Error Correction using DOP

Menno van Zaanen
School of Computer Studies
University of Leeds
Leeds, LS2 9JT, U.K.
menno@scs.leeds.ac.uk

Abstract

In order to create a robust parser, it is necessary that the parser has a well-defined behaviour on what to do when it is fed with incorrect input. There are several ways to cope with incorrect input. The method described here starts with expanding an Earley parser to let it correct erred input. The corrections consist of inserting or deleting tokens from the input, but other corrections can be simulated by combining these two operations. When parsing an input string, typically more than one derivation can be generated. This is especially the case if the input has been corrected. The Data Oriented Parsing (DOP) model is used to disambiguate among the possible derivations. DOP selects the most probable derivation based on a corpus of derivations that were previously generated by the Earley parser. The complete system, i.e. the expanded Earley parser in combination with DOP, has been successfully tested on correcting corrupted C programs. The next step is to use the system for error correction in natural language processing.

1 Introduction

According to [GJ95] parsing is:

> *the process of structuring a linear representation in accordance with a given grammar.*

A parser[1] takes a sentence[2] as input and, if the sentence is correct according to the grammar, it will generate a structure. This structure describes how the sentence can be generated from the grammar.

Unfortunately, quite often a parser encounters sentences that cannot be generated from the grammar. Since the sentences are not part of the language described by the grammar, the parser cannot generate a structure for them. Since we assume that the language describes all correct sentences, the sentence is incorrect.

Several types of error exist, but we will focus on syntactical errors. Morphological, semantic and pragmatic errors will not be discussed, although morphological error sometimes result in syntactical errors.

When we want to have a parser that can handle sentences that are not part of the language, a mechanism is needed that allows the parser to handle errors that may occur in the input. These mechanisms are called error handling.

First of all some general information on error handling is given, followed by a brief discussion on the used parser. After these sections a description of the disambiguation method is given, concluding the basic concepts needed to understand the rest of the paper. After the description of the basic concepts the implemented systems will be described. When necessary changes in the basic concepts will be discussed there. Finally, a conclusion and some future research will be given.

[1] For more information on parsing, see [GJ95], [AU72], [ASU86] and [WM95].

2 For more information on sentences, languages and grammars, see [Lin90]. 1

2 Basic concepts

2.1 Error handling

A mechanism that handles errors in a parser can be one of the following types:

Error detection *This is the simplest kind of error handling. This method does not try to correct the input, it only detects if the input contains an error. After detecting an error the parser stops parsing and may generate an error message. Most parsers have at least this type of error handling. A parser has the correct-prefix property when it can detect an error at the first symbol in the input that results in a string that is not a prefix of a sentence of the language. A parser has immediate error detection when it can detect an error when the erroneous symbol is first examined. Note that this is a somewhat stronger property, because the parser will not even try processing the erroneous symbol; so a parser with immediate error detection property provides more context of the parse up to that symbol and can gives more extensive information about the error.*

Error recovery *This kind of error handling tries to recover from the error, so parsing may continue. It does not try to correct the input, but it changes the state of the parser, or skips symbols, so the parser can continue parsing.*

Error correction *This is the most advanced way to handle errors. These methods try to change the input in a correct sentence. After correcting the input, parsing can continue. Since this is the type of error handling methods we are most interested in, we will concentrate on this. Error correction methods can be divided into the following types:*

Ad hoc error correction *These error correction methods cannot be generated from the grammar. This means that each grammar uses different error correction code which should be written by someone.*

Local error correction *A local error correction method can be automatically generated from the grammar. To correct an error the method uses information local to the error point, so no context is used. These methods are often easy to implement and quick. Regional error correction*

Regional error correction *methods use some context around the error. This way they can make a better choice of how to continue parsing. This method is most often used in bottom-up parsers, where the error and some context is reduced to a left-hand side of a grammar rule.*

Global error correction *Error correction methods that use the complete program as context are called global error correction methods. These techniques are most often used in general parsers, like the Unger, CYK and Earley parsers. These methods can make the best choice in correcting the error, but are complex and, most often, very slow.*

Several error correction methods have been devised, see for example [ASU86], [GJ95], [LF77], [Lyo74], [L'ev74], [Pai80], [AP72], [Iro63] and [Tho76], but none calculate the most probable correction. The systems that will be treated in this paper will calculate the most probable correction of an error in a sentence.

2.2 Earley parser

Since most error correction methods depend on the use of a certain parser, we will describe the parser used in this paper briefly.

For this project the Earley parser was chosen ([Ear70]). This parser was chosen because of several reasons:

- [illegible]

- *The Earley parser can generate more than one derivation (at the same time).*
- *The Earley parser can be easily adapted to parse with trees.*
- *It is a top-down, bottom-up parser and can handle left recursion.*
- *The Earley parser has better average time complexity than CYK.*

The notation that is used in this paper is different from the notation Earley used, so an overview is given in figure 1. Note that Earley described his parser with k-look-ahead. In this paper no look-ahead is used, so k = 0.

Figure 1 Notation used in the Earley parser
Rewrite rule ! S!NP VP
Item ! S!NPfflVP
State ! [S!NPfflVP@0]
Stateset ! stateset 0, `the'
[S!fflNP VP@0]
[NP!fflDET N@0]
[DET!ffla@0]
[DET!fflthe@0]

2.3 Disambiguation

The underlying idea of the error correction models described in this paper is that, by changing the (incorrect) input, one or more possible parses can be generated.

A mechanism is needed to select an appropriate parse if more than one possible parses are possible. In general, not some random parse, but the most probable one is wanted. From this most probable parse the most probable correction can be computed.

To find the most probable derivation a disambiguation system is needed. The system used in this paper is called Data Oriented Parsing.

2.3.1 Data Oriented Parsing

Although the Data Oriented Parsing method (DOP) (see [Bod95], [BS96], [Sima], [Simb] and [Bod98]) can be instantiated with many different parameters, this paper describes the ``standard'' DOP system as described as DOP 1 in [Bod95]. This means that the parameters are chosen as follows:

Sentence analyses *In DOP 1 the sentence analyses are syntactically labeled phrase structure trees. These trees are in the form as if they were generated by a ``standard'' parser, based on a context-free grammar.*

Sub-analyses *The sub-analyses used in DOP 1 are elementary trees. Elementary trees are tree structures that can be used to build complete sentence analyses. See figure 2 for some example elementary trees.*

Combination operations *In DOP 1 the only combination operation is rightmost substitution. The rightmost substitution operator ``glues'' the sub-analyses, the elementary trees, together to form a sentence analysis. See figure 2 for an example of the rightmost substitution operator.*

Combination probability *In DOP 1 combination probabilities are defined as ``the probability of selecting a subtree among all corpus-subtrees that could be substituted on a certain non-terminal leaf node''. The chance of selecting subtree t from the structured corpus is*

p(t) = #ft 2 Corpusg
#fv 2 Corpusiroot(t) = root(v)g

This is the ratio between the number of occurrences of a subtree and the total number of occurrences of subtrees with the same root label. (This implies that the probabilities of the subtrees with the same root label sum up to one). To calculate the probability of two combined subtrees, we need the probabilities of both subtrees (we get these from the structured corpus). The probability of the combination of the subtrees is then the product of the probabilities of both the subtrees.

Figure 2 Multiple ways to build the same tree structure

S
NP VP
V NP
sees Mary
ffi
NP
John
=
S
NP VP
John
V NP
sees Mary
S
NPVP
John
ffi
VP
V NP
sees Mary
=
S
NP VP
John
V NP
sees Mary

Since more than one derivation may lead to the same parse, unlike in a stochastic context-free grammar, the Viterbi algorithm cannot be used. Instead the Monte Carlo algorithm is used ([Bod98]) to select the most probable parse from the parse forest. The Viterbi algorithm could have been used if it was not possible to have multiple derivations leading to the same parse.

DOP 1 selects the most probable parse from a parse forest (e.g. generated by an Earley parser) based on the number of occurrences of elementary trees that can be generated from a corpus of annotated sentences.

Although other stochastic disambiguation/parsing systems have been proposed ([LF77], [Tho76] and [Cha93]), the DOP system is used here because DOP has a higher parse accuracy than an stochastic context-free grammar (SCFG) and DOP uses global dependencies.

3 Implementations

This section describes the different systems we have compared. All systems treated here are based on an (extended) Earley parser and the DOP system.

3.1 DOP 2

> *As a first possible solution to the problem of unknown words we consider the model DOP 2, which is a very simple extension of DOP 1: substitute all lexical categories for an unknown word, and estimate the most probable parse of the ''sentence'' by means of DOP 1. The selected parse of an input sentence is then defined as the attachment of the unknown words to their corresponding lexical categories in the estimated most probable parse. Thus, unknown words are assigned a lexical category such that their ''surrounding'' partial parse has maximal probability. ([Bod95])*

This system allows us to parse sentences with unknown words. Syntactical errors can also be seen as unknown words. To correct the error in the sentence it is suffices to know the most probable lexical category of the error. This is exactly what DOP 2 calculates.

Because it is not known beforehand where a possible error is located in the program (which is known with unknown words), we slightly modified DOP 2 in that it inserts all possible lexical categories at the point where the error was detected during parsing. At the error point it might be the case that not all lexical categories are appropriate because of the context at the error point, so less categories will be inserted. And because of this, disambiguating needs to consider less possibilities at the error point, which speeds up disambiguating.

3.1.1 Results

The advantages of DOP 2 are that DOP 2 replaces error tokens with tokens from the class of the most probable lexical category (calculated using DOP 1). This system is easy to implement and understand because it does not use any complex algorithms.

But DOP 2 has some disadvantages as well. When the system encounters a sentence containing an error that was introduced by inserting or deleting tokens, DOP 2 will generate a list of errors for (in the worst case) each token following the error. Another disadvantage is that DOP 2 will only work when replacing tokens generates a valid sentence. DOP 2 cannot change the length of the sentence, because it only ''changes'' tokens. DOP 2 only changes tokens after (and including) the error. So if by changing these tokens no valid sentence can be generated, DOP 2 fails[3].

3.2 DOPPER 1

One of the main disadvantages of DOP 2 is that the length of the input cannot be changed. We defined another system that assumed symbols could have been inserted or deleted. DOPPER 1[4] is an extension of the DOP 2 method. It can correct errors using insertions and deletions and it can also (in some cases) simulate the DOP 2 correction (replacement of symbols). This is done by inserting a symbol and directly after that deleting another symbol.

Instead of changing the error token, DOPPER 1 tries to delete or insert one token. This allows the length of the input to change, unlike DOP 2. Again DOP 1 is used to disambiguate between the possible parses.

[3] *See also the example in figure 6.*

[4] *DOPPER stands for Data Oriented Parsing used in Persistent Error Recovery.*

In order to insert or delete tokens, the Earley parser had to be modified. Two algorithms are needed, one used for inserting a token and another one for deleting a token. To implement this the Earley parser was changed in some ways. The changes are shown in figure 3. These changes were necessary to find out how a certain parse had been generated. The algorithm to do this is clear when looking at the way the parse is generated from the parse forest *and can be found in [vZ97]:66.*

Figure 3 Changes to the standard Earley parser
State ! [S!NPfflVP@0 ,mode]
Mode ! fmatch, delete, insertg
Stateset ! stateset 0, ‘the’
[S!fflNP VP@0 ,match]
[NP!fflDET N@0 ,match]
[DET!ffla@0 ,insert]
[DET!fflthe@0 ,delete]

3.2.1 Handling an inserted symbol

When the parser encounters an error, it may assume the error token had been inserted, so the method to solve the error is by deleting the input token. This is done by ‘‘skipping’’ the input token. This can be accomplished by pretending to have read the symbol, which means we can just copy some items in the Earley parser to the next state. We have chosen to copy only the items which try to accept a terminal. So non-terminals with their trees cannot be deleted in one operation.

If [X!OEffla/@k ,m] is an item(tree) in state l , we add the item(tree) [X!OEffla/@k ,delete] to state l + 1 . This way it is just as if the symbol in the input (6= a) was not part of the input (we skipped the symbol and tried the same item again on the next symbol in the input).
A graphical description of this algorithm can be found in figure 4.

Figure 4 Correcting an inserted symbol

3.2.2. Handling a deleted symbol

When we expect a token to be deleted from the "original" input, we try to insert the possible tokens. We do this by the algorithm given in algorithm 1.

This rather complex algorithm is necessary, because we are ``illegally'' inserting items in a stateset. To let the parser continue parsing correctly, we first tried to just add the items into the stateset, but this did not work. When an item was complete, it could be used multiple times to reduce other items. Because of this, it would be (at least) difficult, if not impossible, to get the correct disambiguation information after parsing. In order to stop items being used multiple times, some sort of separator had to be set in the stateset. This way it is possible to see which items are already used an which still can be used.

This algorithm inserts all possible tokens in a certain place, but does not insert multiple tokens followed by each other. It only handles only one deleted token at a time. To let it handle multiple deleted tokens, the algorithm has to be started multiple times.

3.2.3 Probability of corrections

The insertion and deletion algorithms change the input in a certain way. When the probability of a parse based on corrected input, the probability of the insertion and deletion has to be known.

There are several ways of implementing this probability. We have used a simple approach, by setting the probability of the insertion and deletion to a small pre-determined value. Better insertion and deletion probabilities should take into account

- *different insertion and deletion probabilities,*
- *context dependent probabilities,*
- *corpus dependent probabilities and*
- *writer dependent probabilities.*

3.2.4 Results

The advantages of DOPPER 1 over DOP 2 are that with DOPPER 1 the length of the sentence can change and that DOPPER 1 calculates the most probable correction by inserting or deleting one token per error token. Replacement, like in DOP 2, can be simulated using insertion and deletion.

The major disadvantage of DOPPER 1 is that not all corrections are possible. E.g. multiple deletions are not possible, because the system will select the insertion of a token before deleting the second token.

3.3 DOPPER 2

DOPPER 2 is an extension of DOPPER 1 that allows all possible corrections. It is exactly like DOPPER 1, with the one difference that multiple insertions and deletions are possible.

This is done by applying the algorithm 1 and the algorithm in figure 4 multiple times. The deletion algorithm can be applied until there are no more tokens left on the input, but the insertion algorithm has to be stopped at a certain point. This is because recursive non-terminals may generate sentences of infinite length.

The stop condition on applying the insertion algorithm can be set in different ways. Stop after some predetermined finite insertions, or when the probability of inserting another token gets below a threshold value, are examples of stop conditions.

3.3.1 Results

DOPPER 2 has some advantages over DOPPER 1. Instead of DOPPER 1, DOPPER 2 can generate all possible corrections and the input is corrected into the most probable input according to the most probable parse.

Unfortunately, DOPPER 2 does not work correct when the real error initiated before the error point. This is possible because the Earley parser has the correct-prefix property. Although the parsed part of the input is a correct prefix, the ``actual'' error may have been instantiated before the error point.

3.4 DOPPER 3

DOPPER 3 solves the disadvantage of DOPPER 2. DOPPER 2 corrects the input from the point an error is found on. To correct an error that was before the error point, the entire input needs to be considered for correction. DOPPER 3 does exactly this. DOPPER 2 uses the remaining input (after the error point) as context, DOPPER 3 uses the entire input as context. This means that when an error is found, parsing is restarted with the entire input.

3.4.1 Results

The major advantage of DOPPER 3 is that errors are corrected like DOPPER 2 only they may be corrected even before the error point. Note that DOPPER 3 is a global error correction method, while DOPPER 1 and DOPPER 2 are only regional and DOP 2 is only local. The disadvantage of DOPPER 3 is that because parsing is restarted, it takes more time to calculate the correction.

4 Examples

4.1 Example 1

In figure 5, the DOP 2 method works good. It parses until the error point, the second ‘1’. DOP 2 tries to change the second ‘1’ into something else, so the parsing may continue. The ‘;’ was most probable, so DOP 2 changes the second ‘1’ into the ‘;’.

DOPPER 1 does not work correctly. It parses until the error point and tries to insert a token or delete a token. Because deleting the second ‘1’ does not let the parsing continue, inserting a symbol is the only way. DOPPER 1 inserts a ‘;’ and parsing may continue. Then again an error is found, when DOPPER 1 considers the first ‘g’ directly following the second ‘1’. the second ‘1’.

Again deleting a symbol does not let the parsing continue, so inserting a symbol is again the only way. DOPPER 1 inserts another ‘;’ and parsing can complete.

DOPPER 2 can delete and insert multiple tokens at the error point. Therefore DOPPER 2 does not make the error by DOPPER 1. At the error point DOPPER 2 deletes the ‘1’ and inserts a ‘;’. This way a more probable parse can be generated.

DOPPER 3 is in this case the same as the DOPPER 2 method. DOPPER 3 finds the error and starts all over again, generating the same parse as the DOPPER 2 method.

Figure 5 Example 1

Input program:

```
    int
    main(void) {
        if(1) {
            return 1 1
        }
    }
```

DOP 2	DOPPER 1	DOPPER 2	DOPPER 3
Corrections:			
match `int'	match `int'	match `int'	match `int'
match `main'	match `main'	match `main'	match `main
match `('	match `('	match `('	match `('
match `void'	match `void'	match `void'	match `void'
match `)'	match `)'	match `)'	match `)'
match `{'	match `{'	match `{'	match `{'
match `if'	match `if'	match `if'	match `if'
match `('	match `('	match `('	match '('

match `1'	match `1'	match `1'	match `1'
match ')'	match `)'	match `)'	match `)'
match `{'	match `{'	match `{'	match `}'
match `return'	match `return'	match `return'	match `retur
match '1'	match '1'	match '1'	match '1'
change `1' -> `;'	**insert** `;'	**delete** `1'	**delete** `1'
match `}'	match `1'	**insert** `;'	**insert** `;'
match **'}'**	**insert** ‘;’	match ‘}’	match ’}’
	match ’}’	match ’}’	match ’}’
	match ’}’		
Corrected programs:			
int	int	int	int
main(void) {	main(void){	main(void) {	main(void) {
if(1) {	if(1) {	if(1) {	if(1) {
return 1;	return 1; 1;	return 1;	ret
}	}	}	}
}	}	}	}

__4.2 Example 2__

In example 2, figure 6, DOP 2 does not work very well. DOP 2 parses until the error point, the ‘;’. It then changes the `;' in a `)'. Parsing can continue, but following the `)' a `;' should be present, so the next error corrected by changing the `g' into a `;'. The parsing can continue again, but the program misses a `g'. Unfortunately there are no more tokens to change, so DOP 2 has to give up.

DOPPER 1 works better this time. It parses until it reaches the error point. At that point it may insert or delete one token, because deletion of a token does not let parsing continue, a token is inserted. DOPPER 1 inserts a `)'. Parsing can continue until the end of the program.

DOPPER 2 works exactly the same way as DOPPER 1 this time. It parses until it reaches the error point. At that point it may insert or delete one or more tokens. Just like with DOPPER 1, inserting a `)' is enough.

DOPPER 3 does an even better job. The standard parser parses until an error is found and DOPPER 3 is started. DOPPER 3 starts parsing all over again and at each token it tries if changing tokens in the input generates a more probable parse. Double `(''s are not very probable, so DOPPER 3 removes the second `('. This way an even more probable parse is generated.

__Figure 6 Example 2__

Input program:

```
    int

    main(void) {

        ((1+1);

    }
```

DOP 2	DOPPER 1	DOPPER 2	DOPPER 3
Corrections:			
match `int'	match `int'	match `int'	match `int'
match `main'	match `main'	match `main'	match `main'
match `('	match `('	match `('	match `('
match `void'	match `void'	match `void'	match `void'
match `)'	match `)'	match `)'	match `)'
match `{'	match `{'	match `{'	match `{'
match `('	match `('	match `('	match `('
match `('	match `('	match `('	**delete `('**
match `1'	match `1'	match `1'	match `1'
match `+'	match `+'	match `+'	match `+'
match `1'	match `1'	match `1'	match `1'
match `)'	match `)'	match `)'	match `)'
change `;' -> `)'	**insert** `)'	**insert** `)'	match `;'
change `}' -> `;'	match `;'	match `;'	match `}'
fail	match ‘}’	match ‘}’	
Corrected programs:			
	int	int	int
	main(void){	main(void) {	main(void) {
fail(no program)	((1+1));	((1+1));	(1+1);
	}	}	}

5 Conclusion and future research

In this paper we have discussed several error correction methods. DOP 2 which corrects errors by replacing token, DOPPER 1 which corrects errors by inserting or deleting one token, DOPPER 2 which corrects errors by inserting or deleting multiple tokens and DOPPER 3 which corrects errors by inserting or deleting multiple tokens, but uses the entire program as context.

As can be seen in the examples, DOPPER 2 and DOPPER 3 perform well, but this is subject to the quality of the corpus. The systems might be improved if better systems are used to calculate the probability of inserting and deleting tokens.

At the moment this system has been tested on C programs, but it might as well work on natural language. The main problems here are that in natural language we do not know the complete grammar and that multiple errors in natural language might be dependent on each other.
That is why our interest also focuses on error correction using grammar adjustment. These systems could possibly be used to bootstrap a grammar.

References

[AP72] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parsing for context-free languages. SIAM J. Computing, 1(4):305--312, December 1972.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers - Principles, Techniques, and Tools. Addison-Wesley Publishing Company, 1986.

Addison-Wesley Publishing Company, 1986.

[AU72] Alfred V. Avo and Jeffrey D. Ullman. The Theory of Parsing, Translation, and Compiling - Volume 1: Parsing. Prentice-Hall, Inc., 1972.

[Bod95] R. Bod. Enriching linguistics with statistics: Performance models of natural language. Master's thesis, Universiteit van Amsterdam, 1995.

[Bod98] R. Bod. Beyond Grammar - An Experience-Based Theory of Language. CSLI Publications: Center for the Study of Language and Information, 1998.

[BS96] Rens Bod and Remko Scha. Data-oriented language processing: An overview. Technical report, ILLC: Institute for logic, language and computation, University of Amsterdam, 1996.

[Cha93] Eugene Charniak. Statistical Language Learning. The MIT Press, 1993.

[Ear70] Jay Early. An efficient context-free parsing algorithm. Communications of the ACM, 13(2):94-102, February 1970.

[GJ95] Dick Grune and Ceriel Jacobs. Parsing Techniques - A Practical Guide. Printout by the Authors, 1995.

[Iro63] E.T. Irons. An error-correcting parse algorithm. Communications of the ACM, 6(11):669- 673, November 1963.

[L'ev74] J.P. L'evy. Automatic correction of syntax-errors in programming languages. Acta Informatica, 4:271-292, 1974.

[LF77] Shin-Yee Lu and King-Sun Fu. Stochastic error-correcting syntax analysis for recognition of noisy patterns. IEEE Transactions on computers, C-26(12):1268-1276, December 1977.

[Lin90] Peter Linz. An Introduction to Formal Languages and Automata. D.C. heath and Company, 1990.

[Lyo74] Gordon Lyon. Syntax-directed least-errors analysis for context-free languages: A practical approach. Communications of the ACM, 17(1):3--14, January 1974.

[Pai80] Ajit B. Pai. Global context recovery: A new strategy for syntactic error recovery by table-driven parsers. ACM Transactions on Programming Languages and Systems, 2(1):18-41, January 1980.

[Sima] K. Sima'an. Learning efficient parsing - with application to data oriented parsing and speech understanding. Research Institute for Language and Speech, Utrecht University.

[Simb] K. Sima'an. An optimized algorithm for data oriented parsing. Utrecht University.

[Tho76] Richard A. Thompson. Language correction using probabilistic grammars. IEEE Transactions on Computers, C-25(3):275--286, March 1976.

[vZ97] Menno van Zaanen. Error correction using dop. Master's thesis, Vrije Universiteit, Amsterdam, 1997.

[WM95] Reinhard Wilhelm and Dieter Maurer. Compiler Design. Addison-Wesley Publishing Company, 1995.