Emile 4.1.7

User Guide

Marco Vervoort

email: vervoort@wins.uva.nl

Universiteit van Amsterdam
Faculteit der Wiskunde en Informatica,
Natuurkunde en Sterrenkunde
Plantage Muidergracht 24
1018 TV Amsterdam

Last update: March 18, 2004

Contents

Chapter 1

The Basics of EMILE

1.1 Introduction

Human being are remarkably good in working with natural languages. Even if someone has no knowledge of the formal structure of a language, he or she will be able to tell when 'something' is like 'something else'. For instance, Lewis Caroll's famous poem 'Jabberwocky' starts with

> 'Twas brillig, and the slithy toves
> Did gyre and gimble in the wabe;
> All mimsy were the borogoves
> and the mome raths outgrabe.

Even without Humpty Dumpty's annotations, it is immediately obvious what the *syntactic structure* of the first sentence is: 'brillig' and 'slithy' are adjectives, 'toves' is a noun, 'gyre' and 'gimble' are verbs, etcetera.
So how do we know such things? The short answer is 'from context'. When a sentence starts with ' 'Twas', we are not surprised if the next word is an adjective. Similarly, if a sentence has the pattern 'the (.) did (.) and (.) in the (.)', we expect the missing phrases to be a noun-phrase, two verb-phrases and another noun-phrase, respectively.
The notion of *grammatical type* has many possible definitions. For instance, if we had a *context-free* grammar of a language, we can view each non-terminal symbol as a grammatical type. In general, one of the properties of a grammatical type is, that wherever some expression is used as an expression of that type, other expressions of that type can be substituted without making the sentence ungrammatical. This gives rise to a notion of a grammatical type as a set of expressions together with a set of contexts. For instance, the type 'noun-phrase' could be represented by the set of all noun-phrases, together with the set of all contexts in which a noun-phrase can appear. Combining any of the expressions of a type with any of the contexts will yield a grammatical sentence. Many of these combinations of contexts and expressions, and especially the short ones (in terms of number of words), are likely to appear in actual texts.

In this terminology, we can describe the above phenomenon, as the existence of contexts which are *characteristic* for a type, meaning that whenever something appears in that context, we assume it also belongs to that type. Some types may also have *characteristic expressions*, with the analogous property. It has been conjectured that for any grammatical type in a natural language, the type will have both characteristic contexts and expressions, and furthermore some of these will be relatively *short*.
EMILE[1] 4.1 is a program based on the above concepts. It attempts to learn the grammatical structure of a language from sentences of that language, without being given any prior knowledge of the grammar. For any type in any valid grammar for the language, we can expect context/expression combinations to show up in a sufficiently large sample of sentences of the language. EMILE searches for such clusters of expressions and contexts in the sample, initially confining itself to expressions and contexts that are relatively short. and interprets them as grammatical types. It then tries to find characteristic contexts and expressions, and uses them to extend the types to include longer expressions and contexts. Finally, it formulates derivation rules based on the types found, in the manner of the rules of a context-free grammar. The program can present the grammatical structure found in several ways, as well as use it to parse other sentences or generate new ones.
The theoretical concepts used in EMILE 4.1 are elaborated on in P. Adriaans' articles on EMILE 1.0/2.0 [1] and EMILE 3.0 [2]. in these chapters we will focus on the practical aspects. Note that although EMILE 4.1 is based on the same theoretical concepts as EMILE 3.0, it is not based on the same algorithm. More information on the precursors of EMILE 4.1 may be found in the above articles, as well as in the E. Dörnenburg's Master's Thesis[4].

1.2 Definitions

The three most basic concepts in EMILE are *contexts*, *expressions* and *context/expression pairs*.

Definition 1.2.1 A *context/expression pair* is a sentence split into three parts, for instance

John (makes) tea

Here, 'makes' is called an *expression*, and 'John (.) tea' is called a *context* (with *left-hand side* 'John' and *right-hand side* 'tea').

Definition 1.2.2 We say that an expression e *appears with* a context c, or that the context/expression pair (c, e) has been *encountered*, if $c_l{}^\frown e^\frown c_r$ appears as a sentence in a text, where c_l and c_r are the left-hand side and the right-hand side of c, respectively, and $a^\frown b$ denotes the concatenation of a and b.

[1]EMILE 4.1 is a successor to EMILE 3.0, written by P. Adriaans. The original acronym stands for Entity Modeling Intelligent Learning Engine. It refers to earlier versions of EMILE that also had semantic capacities. The name EMILE is also motivated by the book on education by J.-J. Rousseau.

Remark 1.2.3 Context/expression pairs are not always sensible, as for instance in the sentence

John (drinks coffee, and Mary drinks) tea

where the expression 'drinks coffee, and Mary drinks' appears in the context 'John (.) tea'. EMILE will find such context/expression pairs and attempt to use them in the grammar induction process, But such pairs are usually isolated, i.e. they are not part of any significant clusters. So EMILE will fail to make use of them, and they will be effectively ignored.

As stated before, we view grammatical types in terms of the expressions that belong to that type, and the contexts in which they can appear (as expressions of that type). As such, we define grammatical types as follows:

Definition 1.2.4 In the context of this paper, a grammatical type T is defined as a pair (T_C, T_E), where T_C is a set of contexts, and T_E is a set of expressions. Elements of T_C and T_E are called *primary* contexts and expressions for T.

The intended meaning of this definition is, that all expressions of a type can appear with all of its the contexts.
In natural languages, the type of an expression is not always unambiguous. For instance, the word 'walk' can be both a noun and a verb. Hence 'walk' will not only appear in contexts for noun-phrases, but also in contexts for verb-phrases. The same does not hold for the phrase 'thing': 'thing' only appears in contexts for noun-phrases, and in any such context, any noun can be substituted for 'thing' without making the sentence ungrammatical. We say that 'thing' is *characteristic* for the type 'noun'. Formally,

Definition 1.2.5 An expression of a type T is *characteristic* for T if it only appears with contexts of type T. Similarly, a context of a type T is *characteristic* for T if it only appears with expressions of type T.[2]

Note that a context may have more than one type, so a context appearing with a expression characteristic for a type T may be of other types in addition to being of type T. For instance, 'a thing' is a characteristic expression for the grammatical type of noun-with-particle-phrases, but it can occur in the context 'This is (.)', which is a context for both noun-with-particle-phrases and adjectives.
In these chapters, we will also use *characteristic** and *secondary* expressions and contexts. However, as these definitions are rather dependent on the algorithms, they will be delayed until section 2.4. That section also has several examples of characteristic expressions and contexts.

Notation For any type T, T_E, T_E^{ch}, T_E^* and T_C^{se} denote the sets of primary, characteristic, characteristic* and secondary expressions of T, and T_C, T_C^{ch}, T_C^* and T_C^{se} denote the corresponding sets of contexts.

[2]EMILE changes this definition slightly in implementation, in that contexts and expressions which have been assigned no type at all are completely ignored, i.e. an expression is characteristic if all contexts with which it appears are of type T, or untyped.

The EMILE program also attempts to transform the collection of grammatical types found into a context-free grammar consisting of derivation rules. Such rules generally are of the form

$$[T] \Rightarrow s_0[T_1]s_1[T_2]\ldots[T_k]s_k$$

where $T, T_1, T_2, \ldots, T_k$ are grammatical types, and $s_0, s_1, \ldots, s_k$ are (possibly empty) sequences of words. Given a rule with left-hand side $[T]$, and a sequence of word-sequences and grammatical types containing $[T]$, that appearance of $[T]$ can be replaced by the right-hand side of the rule, (concatenating adjacent word-sequences as necessary). Any sequence which can be obtained from another sequence by such rule applications, is said to be derivable from that sequence. The language of a context-free grammar consists of those word-sequences e such that $[0] \Rightarrow e$ is derivable, where $[0]$ denotes the type of whole sentences.

Chapter 2

The Workings of EMILE

This chapter attempts to give some insight into the reasoning underlying the algorithms of EMILE. We will start with a very simple version of the basic algorithm, and in several steps change it to the full algorithm, at each step elaborating on the motivations for the change.

2.1 1-dimensional clustering

Given a sample of sentences, we want to obtain sets of expressions and contexts that correspond to grammatical types. A simple clustering technique is to extract all possible context/expression combinations from a given sample of sentences, and group together expressions that appear with the same context.

Example 2.1.1 If we take the sample sentences 'John makes tea' and 'John likes tea', we get the following context/expression *matrix*:

	(.) makes tea	John (.) tea	John makes (.)	(.) tea	John (.)	(.)	(,) likes tea	John likes (.)
John	x						x	
makes		x						
tea			x					x
John makes				x				
makes tea					x			
John makes tea						x		
likes		x						
John likes				x				
likes tea					x			
John likes tea						x		

from which we can obtain the clusters

[‘John (.) tea’, {‘makes’, ‘likes’}]
[‘(.) tea’, {‘John makes’, ‘John likes’}]
[‘John (.)’, {‘makes tea’, ‘likes tea’}]
[‘(.)’, {‘John makes tea’, ‘John likes tea’}]

Next, we can group contexts together if they appear with exactly the same expressions.

Example 2.1.2 If we add the sentences ‘John makes coffee’, ‘John likes coffee’ to the previous sample, the relevant part of the context/expression matrix looks like

	John (.) tea	John (.) coffee	John makes (.)	John likes (.)
makes	x	x		
likes	x	x		
tea			x	x
coffee			x	x

which yields the clusters

[{‘John (.) tea’, ‘John (.) coffee’}, {‘makes’, ‘likes’}]
[{‘John makes (.)’, ‘John likes (.)’}, {‘tea’, ‘coffee’}]

As stated before, a grammatical type can be characterized by the expressions that are of that type, and the contexts in which expressions of that type appear. Hence the clusters we find here can be interpreted as grammatical types. For instance, the clusters in the above example could be said to correspond to the grammatical types of ‘verbs’ and ‘nouns’, respectively.

2.2 2-dimensional clustering

One of the flaws in this technique is that it doesn’t properly handle contexts whose type is ambiguous.

Example 2.2.1 If we add the sentences ‘John likes eating’ and ‘John is eating’ to the previous example, the relevant part of the context/expression matrix will look like this:

	John (.) tea	John (.) coffee	John (.) eating	John makes (.)	John likes (.)	John is (.)
makes	x	x				
likes	x	x	x			
is			x			
tea				x	x	
coffee				x	x	
eating					x	x

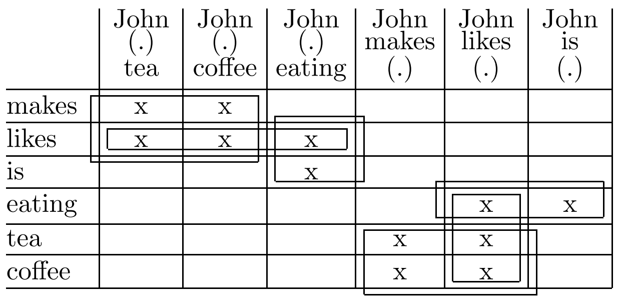
Here we can intuitively identify four grammatical types: noun-phrases, verb-phrases, 'ing'-phrases, and 'verbs-that-appear-with-ing-phrases'-phrases. The context 'John likes (.)' is ambiguous, in the sense that it appears with both noun-phrases and 'ing'-phrases. If we proceed as before, we get the following clusters

[{'John (.) tea', 'John (.) coffee'}, {'makes', 'likes'}]
[{'John (.) eating'}, {'likes', 'is'}]
[{'John makes (.)'}, {'tea', 'coffee'}]
[{'John likes (.)'}, {'tea', 'coffee', 'eating'}]
[{'John is (.)'}, {'eating'}]

i.e. the context 'John likes (.)' is assigned a separate type.

Assigning ambiguous contexts a separate type not only results in a less natural representation, in a later step it will prevent us from correctly identifying the characteristic expressions of a type (as will be demonstrated in Example 2.4.4). A more natural representation would be to allow ambiguous contexts and expressions to belong to multiple types. For this, we need to use a different clustering method. The clustering method EMILE uses is to search for maximum-sized blocks in the matrix. This could be termed *2-dimensional clustering.*

Example 2.2.2 The following picture shows the matrix of the previous example, with the maximum-sized blocks indicated by rectangles.[1]



	John (.) tea	John (.) coffee	John (.) eating	John makes (.)	John likes (.)	John is (.)
makes	x	x				
likes	x	x	x			
is			x			
eating					x	x
tea				x	x	
coffee				x	x	

These blocks correspond to the clusters

[{'John (.) tea', 'John (.) coffee'}, {'makes', 'likes'}]
[{'John (.) eating'}, {'likes', 'is'}]
[{'John makes (.)', 'John likes (.)'}, {'tea', 'coffee'}]
[{'John is (.)', 'John likes (.)'}, {'eating'}]
[{'John (.) tea', 'John (.) coffee', 'John (.) eating'}, {'likes'}]
[{'John likes (.)'}, {'eating', 'tea', 'coffee'}]

The last two clusters correspond to sets of context/expression pairs which are already 'covered' by the other blocks. In a sense these blocks are superfluous.

[1]Please note that the expressions and contexts have been arranged to allow the blocks to be easily indicated: in general, blocks will *not* consist of adjacent context/expression pairs.

The algorithm to find these blocks is very simple: starting from a single context/expression pair, EMILE randomly adds contexts and expressions while ensuring that the resulting block is still contained in the matrix, and keeps adding contexts and expressions until the block can no longer be enlarged. This is done for each context/expression pair that is not already contained in some block. Once all context/expression pairs have been 'covered', the superfluous blocks (those completely covered by other blocks) are discarded.

2.3 Allowing for imperfect data

In the previous section, the requirement for a block was that it was entirely contained within the matrix, i.e. the clustering algorithm did not find a type unless every possible combination of contexts and expressions of that type had actually been encountered and stored in the matrix. This only works if a perfect sample has been provided. In practical use, we need to allow for imperfect samples. There are many context/expression combinations, such as for instance 'John likes evaporating', which are grammatical but nevertheless will appear infrequently, if ever.

To allow EMILE to be used with imperfect samples, two enhancements have been made to the algorithm. First, the requirement that the block is completely contained in the matrix, is weakened to a requirement that the block is *mostly* contained in the matrix. Specifically, a certain percentage of the context/expression pairs of the block as a whole should be contained in the matrix, as well as a certain percentage of the context/expression pairs in each individual row or column. We can express this as

$$\begin{array}{rlll} & \#(M \cap (T_C \times T_E)) \geq & \#(T_C \times T_E) & \cdot\ \texttt{total\_support\%} \\ \forall c \in T_C: & \#(M \cap (\{c\} \times T_E)) \geq & \#T_E & \cdot\ \texttt{context\_support\%} \\ \forall e \in T_E: & \#(M \cap (T_C \times \{e\})) \geq & \#T_C & \cdot\ \texttt{expression\_support\%} \end{array}$$

where M is the set of all encountered context/expression pairs, and the values `XXX_support%` are constants that can be set by the user.

Example 2.3.1 Suppose that the matrix of context/expression pairs EMILE has encountered has the following sub-matrix:

	John makes (.)	John likes (.)	John drinks (.)	John buys (.)
tea	x	x	x	x
coffee	x	x	x	x
lemonade	x	x	x	
soup	x	x	x	x
apples				x

If the settings `context_support%` and `expression_support%` have been set to 75%, and `total_support%` has been set to 80%, then the type represented by the cluster

$$\left[\begin{array}{c}\{\text{'John makes (.)', 'John likes (.)', 'John drinks (.)', 'John buys (.)'}\},\\ \{\text{'tea', 'coffee', 'lemonade', 'soup'}\}\end{array}\right]$$

will be identified, in spite of the fact that one of the context/expression pair of the block, ('John buys (.)', 'lemonade'), does not appear in the matrix. However, the expression 'apples' will not be added to the above type, since it appears with less than `expression_support%` of the contexts.

Secondly, note that of the different expressions and contexts belonging to a grammatical type, it can be expected that the short and medium-length ones (in terms of number of words) will be encountered more often than the long ones. In other words, if we restrict the sample to short and medium-length contexts and expressions, it will be closer to a perfect sample. Implementing this notion, EMILE initially uses only short and medium-length contexts and expressions when searching for grammatical types.

Definition 2.3.2 The contexts and expressions EMILE finds for a type T in the initial clustering algorithm are called the *primary* contexts and expressions of T.

2.4 Characteristic, characteristic* and secondary expressions and contexts

To search for longer expressions and contexts associated with types, EMILE uses characteristic expressions and contexts. We repeat the definition from chapter 1:

Definition 2.4.1 [1.2.5] An expression of a type T is *characteristic* for T if it only appears with contexts of type T. Similarly, a context of a type T is *characteristic* for T if it only appears with expressions of type T.[2]

Occasionally, a type has no characteristic expressions (due to imperfections in the sample or the inherent ambiguity of the type): in such cases, the primary expressions of the type are used in place of the characteristic expressions. We call these the *characteristic** expressions of T, i.e.

Definition 2.4.2 An expression of a type T is *characteristic** for T if it is a characteristic expression for T, or if it is a primary expression for T and T has no characteristic expressions.

[2]Since the types involved usually have not been fully identified yet, EMILE changes this definition slightly in implementation, in that contexts and expressions which have been assigned no type at all are completely ignored, i.e. an expression is characteristic if all contexts with which it appears are of type T, or untyped.

the *characteristic** expressions of T are defined as the characteristic expressions of T if there are any, and as the primary expressions of T otherwise.
The definitions of *characteristic* and *characteristic** contexts of a type T are analogous.
Any untyped context appearing with an characteristic expression of a type T is likely to belong to T as well. Contexts which appear with (a certain percentage of the) characteristic* expressions of T are called *secondary* contexts of T, as opposed to the *primary* contexts found by the clustering algorithm. Analogous for *secondary* expressions. Note that the constraint on the length of primary contexts and expressions does not apply to secondary contexts and expressions, and hence this allows for long contexts and expressions to be associated with types.

Example 2.4.3 In the previous example, for the type represented by the cluster

[{'John likes (.)'}, {'eating', 'tea', 'coffee'}]

'John likes (.)' only appears with 'eating', 'tea' and 'coffee', so it is a characteristic (and hence characteristic*) context for this type. The expression 'eating' also appears with the context 'John is (.)', so it is not a characteristic expression. A similar condition obtains for 'tea' and 'coffee', so the type has no characteristic expressions at all. Consequentially, its primary expressions 'eating', 'tea' and 'coffee' are also its characteristic* expressions.
For the type represented by the cluster

[{'John makes (.)', 'John likes (.)'}, {'tea', 'coffee'}]

all its expressions and contexts are characteristic.

Example 2.4.4 In Example 2.2.1, we used 1-dimensional clustering to obtain the cluster

[{'John makes (.)'}, {'tea', 'coffee'}]

Here, 'tea' and 'coffee' are not characteristic expressions, since they appear with the context 'John likes (.)', which here is not a context belonging to the type. So the type has no characteristic expressions. It is easy to see that when using 1-dimensional clustering, whenever a context is ambiguous[3], all types involved will lack characteristic expressions.

Example 2.4.5 Assume that primary expressions are constrained to be at most 5 words long. If we add the sentence 'John makes really really really really strong coffee' to the sample of the previous example, then the expression 'really really really really strong coffee' will not be added as a primary expression to the type represented by the cluster

[{'John makes (.)', 'John likes (.)'}, {'tea', 'coffee'}]

[3]'Ambiguous' in the sense that the set of expressions it appears with is the union of several smaller sets associated with other contexts

However, since 'John makes (.)' is a characteristic expression of this type, the expression 'really really really really strong coffee' will be associated with the type as a secondary expression.

2.5 Finding rules

The EMILE program also transforms the grammatical types found into derivation rules. For reasons of simplicity, EMILE constructs a context-free grammar rather than a context-sensitive grammar. For this construction, only the sets of expressions associated with the types are needed: the sets of contexts associated with the types are not used in creating the derivation rules.
First, EMILE searches for rules that are *supported*. Obviously, if an expression e belongs to a type T (as a secondary expression), the rule

$$[T] \Rightarrow e$$

is supported. EMILE finds more complex rules, by searching for characteristic* expressions of one type that appear in the secondary expressions of another (or the same) type. For example, if the characteristic* expressions of a type T are

$$\{\text{dog}, \text{cat}, \text{gerbil}\}$$

and the type [0] contains the secondary expressions

$$\{\text{I feed my dog}, \text{I feed my cat}, \text{I feed my gerbil}\}$$

then EMILE will find the rule

$$[0] \Rightarrow \text{I feed my } [T]$$

This process of abstraction is repeated to obtain more abstract rules. Formally, a rule R is considered to be *supported* if it is of the form $[T] \Rightarrow e$ (with e being a secondary expression of T), or if it is of the form $[T] \Rightarrow s_0[T_1]s_1[T_2]\ldots s_k$, $k \geq 1$, and for some $i \in \{1, \ldots, k\}$,

$$\#\{e \in T_E^* \mid R \text{ with } [T_i] \text{ replaced by } e \text{ is supported}\} \geq \#T_E^* \cdot \texttt{rule\_support\%} \tag{2.1}$$

In certain cases, using characteristic* and secondary expressions in this manner allows EMILE to find recursive rules. For instance, a characteristic* expression of the type of sentences S might be

Mary drinks tea

If the maximum length for primary expressions is set to 4 or 5, the sentence

John observes that Mary drinks tea

will be a secondary expression of S, but not a primary or characteristic one. So if there are no other expressions involved, EMILE would derive the rules

$$\begin{aligned} [S] &\Rightarrow \text{Mary drinks tea} \\ [S] &\Rightarrow \text{John observes that } [S] \end{aligned}$$

which would allow the resulting grammar to generate, for instance,

John observes that John observes that John observes that Mary drinks tea

EMILE creates a set of supported rules capable of generating all sentences in the original sample. To reduce the size of this grammar, the program discards from the final output rules which are superfluous, such as rules which are instantiations of other rules[4], and rules for types which aren't referred to in other rules.

Experiments showed that often, EMILE finds several types which where only slight variations of one another. If all these types are referred to in the rules, this results in a much larger ruleset than is necessary. The most recent incarnation of EMILE tries to prevent this by being actively conservative in the number of types used: a set of *used types* is maintained, and only rules using those types are considered for inclusion. This set initially contains only the whole-sentence type [0], and types are added only if this would result in a decrease in the size of the total ruleset.[5]

2.6 Negative Samples

Although EMILE can work with imperfect samples by reducing the support required for grammatical types, there are limits to what can be achieved in this fashion. If we lower the required support *too much*, EMILE will 'overgeneralize', attempt to interpolate patterns which do not exist. On the other hand, if the required support is too high, the sample required to achieve satisfactory results will be rather large, in relation to the underlying grammar of the language.

One way of improving this is by extending the algorithm to allow it to handle negative samples, i.e. samples of sentences which are known to be *not* grammatical. If we could explicitly tell EMILE that the overgeneralizations it finds are not grammatically correct, then we could safely use EMILE with low required support settings.

The current implementation of EMILE contains some basic functionality for this. Sentences can be learned as part of a 'negative sample', and the resulting context/expression-pairs contribute *negatively* to the support of any grammatical type containing them. This will cause EMILE to try to find types which do not contain these context/expression pairs.

[4]I.e. which can be obtained from other rules by replacing a type reference by a secondary expression of that type

[5]EMILE can also be set to allow a small increase: this often results in a more meaningful grammar at the expense of a slightly larger ruleset.

Samples of non-grammatical sentences usually have to be generated on purpose. Since they are mostly used to mark sentences ungrammatical which otherwise would have been considered grammatical, samples of arbitrary non-grammatical sentences, on average, contain less useful information. Because of this, the current implementation of EMILE is able to list sentences whose 'assumed' grammaticality is the most tenuous, corresponding to context/expression pairs for types with very tentative support. This allows for a more 'directed' generation of negative samples, by an external program or 'oracle'.

2.7 Future Developments

There is still a lot of room for improvement. At the moment, the immediate efforts concentrate on experimenting with the various control parameters of EMILE, gauging their effect and their interactions. Of course, to accurately compare different parameter settings, an objective measurement of the results is necessary. Currently, we measure results by running EMILE on a sample text for which we have an annotated parsing available, and comparing that parsing to EMILE's parsing using the EVALB bracket scoring program.

EMILE has some basic interactive facilities: it can generate a list of sentences which it assumes to be grammatical but is not sure about, which could be inspected by a human operator or some other 'oracle' and fed back to EMILE as a positive or negative sample to learn. This functionality could be improved. For instance, at this moment EMILE can make temporary type assignments 'on the fly' when parsing sentences with unknown words: creating an interactive interface for this would allow for a very natural way of 'teaching' EMILE a better grammar.

A possible extension of EMILE is to the algorithm constructing the derivation-rule grammars. Currently EMILE constructs a context-free grammar. It may be possible to adapt EMILE to produce a more sensible context-sensitive grammars, using the sets of contexts produced by the clustering algorithm.

The EMILE program has some superficial similarities to another grammar-analyzing program called ABL[6] written by Menno van Zaanen[6, 7, 8, 9]. The latter uses a completely different algorithm to achieve its results, however. Possibly, the two algorithms could be merged, each complementing the other.

[6]Alignment-Based Learning

Chapter 3

Using EMILE

There are three ways to use Emile: by using the interactive command line interpreter, by starting the program with command line arguments, or by giving it commands in a script. All three are described below.

3.1 Emile's command line interpreter

The easiest way to use the Emile program is to start the program without any command line arguments, wait until the prompt > appears, and then type in EMILE commands from the terminal. Emile's interactive command line interpreter has the following features:

- A command can be spread over multiple lines. All lines except the last should end with a backslash (\) to indicate that the command continues on the next line, and Emile should not execute the command(s) yet.
- A line may contain multiple commands, separated by semicolons (;).
- Output may be redirected to a file by appending '> *filename*' or '>> *filename*' to a command (the second form appends the output to a file rather than overwriting it). Appending '| *programname*' to a command will redirect the output to an external program. For instance, you may wish to append '| more:' to a command to redirect output to the 'more' program and thus allow you to page through it.
- Spaces normally separate command arguments. However, command arguments may contain spaces, semicolons, or other special symbols by either putting backslashes before all special symbols, or by enclosing the command argument in single (') or double (") quotes.
- If the Gnu `Readline` library is installed on your system, you may use the Tab-key for command completion, i.e. if you press the TAB key, the EMILE program will attempt to complete the current command name,

variable name or filename argument, and display a list of possible continuations (if appropriate).

- If the Gnu `Readline` library is installed on your system, all commands are stored in a history buffer, and you may call up previous commands by pressing the Arrow-up and Arrow-down keys.
- The Emile command line interpreter uses fuzzy patterns to allow it to recognize abbreviations of command names and options.

3.2 Startup command line options

Emile can be started with the following command line options:

`-d` *file*

`--database` *file*

start with the specified file for storing and retrieving grammar data. If this option is not specified, EMILE uses the file `grammar.dat` as the the default file for storing and retrieving its grammar data.

`-v` [*n*]

`--verbose` [*n*]

start at verbosity level *n* (default: 2). Higher verbosity levels give more information about intermediate stages of computation.

`-q`

`--quiet`

start at verbosity level 0, which disables most command logging.

`-h`

`--help`

print a help screen and exit

`-V`

`--version`

print the version of the program and exit

The long versions of the options are only available if the Gnu Getopt library is installed on your system.
If there are any other startup command line arguments, Emile will interpret them as Emile commands and execute them (instead of starting the interactive command line interpreter). So for instance, by typing

```
emile parse mysentences.txt
```

at the shell prompt, you cause Emile to parse your sentences without having to enter an interactive session with Emile. This can be useful in shell scripts and the like.

3.3 Executing Emile commands from a script

Although it is possible to have Emile execute an entire series of commands as command line arguments to the program invocation, this is not really convenient. A much better way is to put the commands in a script, and execute it by giving the '`script`' command, either as a startup command line argument:

```
emile script mycommands.txt
```

or from within the interactive interpreter. The syntax for command lines in a script is the same as the syntax for command lines in the interactive interpreter.

3.4 Example of usage

Suppose that the file `testdata.txt` contains the following sentences:

```
the fox jumped.  the dog jumped.
the quick brown fox jumped.
the lazy dog jumped.
the fox jumped over the dog.
the dog jumped over the fox.
the quick brown fox jumped over the dog.
the lazy dog jumped over the fox.
the fox jumped over the lazy dog.
the dog jumped over the quick brown fox.
the lazy dog jumped over the quick brown fox.
```

We start the Emile program in the normal fashion. For instance, if we are working on a Unix system, we could type

```
user@host.com 1)emile
Emile version 4.1.7:
Error opening database grammar.dat
Awaiting commands:
>
```

(underlined text denotes text typed by the user). Now Emile is ready to start processing. We want Emile to read in the sentences of the sample, compile the grammar, and display the grammatical rules.

```
> learn testdata.txt
Done learning phrases from file testdata.txt
> compile
```

```
Updating grammar
Extending existing types
Creating new types
Eliminating superfluous types
Finding characteristic and secondary expressions and
contexts
Updating auxiliary secondary matrices
Compiling rules
Starting with whole-sentence and previously used types
Trying for new types
Optimizing rulesets for each used type
Eliminating types that are unused or can be shortcircuited
Updating parsing table and auxiliary rules matrices
Done updating grammar
> show rules

Rules relevant to type [0]:

[0] --> [18] dog jumped .
[0] --> the [4] jumped .
[0] --> [18] dog jumped over the [4] .
[0] --> the [4] jumped over [18] dog .
[4] --> fox
[4] --> quick brown [4]
[18] --> the
[18] --> the lazy
>
```

Now, there is one sentence which logically is missing from the sample. We can look at the ruleset above and see if the grammar Emile has found accepts the missing sentence, but we can also let Emile do all the work:

```
> parse-phrase the quick brown fox jumped over the lazy
dog .
Parsing "the quick brown fox jumped over the lazy dog ."
Result:  (the (quick brown (fox)[4])[4] jumped over (the
lazy)[18] dog .)[0]
>
```

or, to see in general how the grammar found by Emile generalizes the sample,

```
> generate 5 new
Generating 5 new sentences with depth between 0 and 999
(100 tries)
the quick brown fox jumped over the lazy dog .
the dog jumped over the quick brown quick brown fox .
the lazy dog jumped over the quick brown quick brown
```

```
quick brown fox .
the lazy dog jumped over the quick brown quick brown
fox .
the quick brown quick brown quick brown fox jumped .
>
```

If we find that this overgeneralizes the sample, we can try again with higher required support settings:

```
> set support 70
Setting expression_support_percentage to 70
Setting context_support_percentage to 70
Setting secondary_expression_support_percentage to 70
Setting secondary_context_support_percentage to 70
Setting rule_support_percentage to 70
Setting sesp_for_no_characteristics to 70
Setting scsp_for_no_characteristics to 70
Setting rsp_for_no_characteristics to 70
Setting total_support_percentage to 91
> show rules
Updating grammar
Extending existing types
Creating new types
Eliminating superfluous types
Finding characteristic and secondary expressions and
contexts
Updating auxiliary secondary matrices
Compiling rules
Starting with whole-sentence and previously used types
Trying for new types
Optimizing rulesets for each used type
Eliminating types that are unused or can be shortcircuited
Updating parsing table and auxiliary rules matrices
Done updating grammar

Rules relevant to type [0]:

[0] --> the fox jumped over the lazy dog .
[0] --> [65] jumped .
[0] --> [65] jumped over the dog .
[0] --> the [64]
[0] --> the lazy [64]
[64] --> dog jumped .
[64] --> dog jumped over [65] .
[65] --> the fox
[65] --> the quick brown fox
> generate 5 new
```

```
Generating 5 new sentences with depth between 0 and 999
(100 tries)
No new phrases found within 100 tries
>
```

Note that Emile gave an implicit `compile` command at the moment the `show rules` command needed an up-to-date grammar. Actually, we could have done the same the first time, instead of giving an explicit `compile` command.
If, on the other hand, we want to generalize a bit more, we lower the support settings. This time, we also tell Emile to skip the log messages.

```
> quiet
> set support 25
> show rules

Rules relevant to type [0]:

[0] --> the [66] jumped .
[0] --> the [66] jumped over the [66] .
[66] --> fox
[66] --> dog
[66] --> quick brown [66]
[66] --> lazy [66]
> generate 5 new
the quick brown lazy lazy lazy dog jumped over the dog .
the quick brown lazy lazy dog jumped .
the dog jumped over the dog .
the quick brown lazy quick brown dog jumped over the
lazy dog .
the lazy lazy fox jumped over the lazy quick brown fox .
>
```

If we are happy with this, we can quit the program. Emile will automatically save the grammar. To verify this, we enable the logging messages again.

```
> verbose
Command logging reactivated
> quit
Done parsing commands from standard input
Writing data to file (Emile 4.1.7 binary format)
Database grammar.dat saved
user@host 2)
```

3.5 Common Files used with EMILE

EMILE uses the following types of files:

grammar files , which EMILE uses to store and retrieve its grammar data.

These files can be stored in either binary format or ASCII format, depending on the value of the `save_in_ascii_format` setting. Saving it in ASCII format will make it easier to manipulate the file externally (by hand or with other programs): however, saving it in binary format (the default) will reduce filesize and improve save/load speed by a factor 2.

By default, EMILE uses a file named `grammar.dat` to store and retrieve its grammar data. However, this can be easily changed by starting EMILE with the `--database` option, or using the `load` and `save` commands and specifying a filename.

sample texts , which EMILE uses with the `learn`, `unlearn` and `parse` to learn grammars or create parsings commands. These files should be plain (ASCII) text files.

script files , which EMILE uses with the `script` command, as described in the previous section.

output files , created by EMILE when redirecting the output of a command, by appending '> *filename*' or '>> *filename*' to the command in EMILE's command line interpreter.

Appendix A

Installing EMILE

The EMILE program should be able to run on almost any machine that has a C++ compiler installed with the SGI Standard Library. However, at the time of this writing, it has only been tested on the Linux and Solaris Unix platforms, using the Gnu g++ compiler (version egcs-2.91.66, egcs 1.2.2 release, and version g++-2.92.107). We have no specific information about the compile status of Emile with other configurations: presumably most people manage to get it working. If you feel you have a tip to contribute about installing Emile with a specific other configuration, you can send it to the maintainer of the program, at `vervoort@wins.uva.nl`.
To install the program, you first need to obtain a license and download the source code. At the time of this writing, although you do not need to pay to use the EMILE program, it is not distributed freely: the Universiteit van Amsterdam provides designated users with a temporary license for EMILE 4.1 in the context of a well defined research project. To obtain a license and download instructions, you currently have to contact Pieter Adriaans directly. More information can be found on the EMILE webpage, at

```
http://turing.wins.uva.nl/ pietera/Emile/
```

You will be given the URL for a zipped archive containing the source code for the program. After downloading and unzip the archive, first you must generate and configure the Makefile by typing

```
./configure
```

Then you use `make` to compile the EMILE program, by typing

```
make emile
```

EMILE can be compiled in Morpheme (Word) Analysis mode, which is a variant of EMILE optimized for analyzing words consisting of letters (as opposed to analyzing sentences consisting of words). To create this variant of EMILE, type

```
make morpheme
```

The compilation includes the Gnu GetOptLong and Readline libraries if available: due to incompatible licenses, these libraries cannot be shipped with the program. If these libraries are not available, EMILE will be compiled without the history buffer, tab completion or long option handling facilities. The Readline library is available at

http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html

and the GetOptLong library can be found in most distributions of the Gnu GCC compiler.
If you have problems compiling the program, the compilation process can be modified by adding options such as `--disable-optimize` when invoking the `./configure` script. More information can be found in the `INSTALL` file included with the program.

Appendix B

Emile commands

B.1 Overview of commands

`clear`	clear grammar/data/settings/rules/all
`compile`	explicitly recompile the grammar
`exit`	exit the program or the current script
`generate`	generate sentences or phrases
`help`	show help screens
`learn`	learn sentences from files or terminal input
`learn-phrase`	learn a specified sentence
`load`	load the database
`parse`	parse sentences from files or terminal input
`parse-phrase`	parse a specified sentence
`parse-type`	parse a phrase of the specified or an arbitrary type
`quiet`	turn off logging messages
`save`	save the database [to a specific file]
`script`	execute commands from a file (or from terminal input)
`set`	set the value of a settings variable
`shell`	execute a shell command, or enter a subshell
`show`	display various data
`unlearn`	unlearn sentences from input [or from file]
`unlearn-phrase`	unlearn 1 sentence from input [or as given]
`verbose`	turn on comments [or set verbosity level]
`version`	this displays the version number of the program.

B.2 The `clear` command

Syntax

`clear [g|d|s|r|a]`

`new [g|d|s|r|a]`

Synopsis

clear grammar/data/settings/rules/all

This command clears parts or all of the database. The command arguments have the following meaning:

`r[ules]`

Clear the derivation rules found by the program without clearing the grammatical types. This is used mainly to negate the preference for reusing previously used types when searching for new rules.

`g[rammar]`

Clear the grammar, i.e. the rules and all the grammatical types.

`d[ata]`

Clear the sentences read in, as well as the grammar.

`s[ettings]`

Clear all settings

`a[ll]`

Clear all (i.e. settings, sentences, grammar and rules), resetting the program to a virgin state

B.3 The `compile` command

Syntax

`compile`

Synopsis

explicitly recompile the grammar

This command (re-)compiles the grammar and regenerates the ruleset based on the set of sentences read in so far. This is normally done automatically when a command is given that uses the grammar and the grammar is not up-to-date, either because new sentences have been read in or because relevant settings have been changed since the last time the grammar was compiled. The `compile` command explicitly requests a compilation, for instance because you

want the grammar compiled without using it immediately, or because you want to study the logging output.
The behavior of this command is influenced by many settings. For details, please see the individual settings and algorithm descriptions..
The `compile` command is called implicitly when necessary if one of the following commands is used: `generate`, `parse`, `parse-phrase`, `parse-type`, `show all`, `show context(s)`, `show dict`, `show expression(s)`, `show rule(s)`, `show type(s)`. Whether an update would be necessary can be checked by issuing the `show statistics` command and checking the value of the `grammar_updated` entry.

B.4 The `exit` command

Syntax

```
exit [nosave]

quit [nosave]

return [nosave]

x [nosave]

q [nosave]
```

Synopsis

exit the program or the current script

This command exits the program or the current script. If the program is ended, the grammar and sentences are automatically saved if they have changed since the last save, unless the optional 'nosave' argument was used.

B.5 The `generate` command

Syntax

`generate` [*n*] `[new]` [*t*] [*m*] [*M*] [*r*]

`g` [*n*] `[new]` [*t*] [*m*] [*M*] [*r*]

Synopsis

generate sentences or phrases

This command uses the current grammar of derivation rules to generate sentences or phrases. The command takes the following optional arguments:

n

the number of phrases to generate (default: 1) `new`

: indicates that Emile should exclude phrases which are already known (for that type)

t

: the type for which new phrases are to be generated. Default is type 0, which generates new sentences

m

: the minimum depth of the parse tree of the generated phrase (default: 0)

M

: the maximum depth of the parse tree of the generated phrase (default: 999)

r

: the maximum number of retries before giving up when attempting to generate new phrases.

B.6 The `help` command

Syntax

```
help [shortcuts|show|version]
?  [shortcuts|show|version]
```

Synopsis

show help screens

This command displays one of several help screens:

(default)

: a short overview of the available commands.

`shortcuts`

: an overview of the available shortcuts, abbreviations and alternative command names

`show`

: an overview of the different arguments for the `show` command.

`version`

: the version number of the program. This is an alias for the `version` command.

B.7 The `learn` command

Syntax

`learn` [*filename*] . . .

`l` [*filename*] . . .

Synopsis

learn sentences from files or terminal input

This command learns a (positive) sample of sentences from one or more files (i.e. adds the sentences to the set of sentences-read-in). If no filenames are given, input is taken from the terminal (terminated by an end-of-file symbol, usually obtained by pressing Control-D).
If a sentences is reread, the multiplicity of the sentence is increased by 1. This can, for instance, compensate for occurrences of the same sentence in negative samples. To decrease the multiplicity of sentences or read a negative sample of sentences, use the `unlearn` command.
A file is normally parsed into sentences at each period, question mark, exclamation mark, or semicolon, ignoring periods directly following one-letter words (these are considered to be abbreviation points). Sentences are normally divided into words using space(s) as delimiters, but each non-alphanumeric symbol forms its own word. Both behaviors can be altered using settings. See also the descriptions of the `end_of_sentence_markers`, `allow_multi_line_sentences`, `ignore_abbreviation_periods`, `end_of_sentence_regular_expression` and `word_regular_expression` settings.

B.8 The `learn-phrase` command

Syntax

`learn-phrase` [*phrase*]

`lp` [*phrase*]

Synopsis

learn a specified sentence

This command learns one sentence (i.e. adds it to the set of sentences-read-in). If the sentence is not given as (a sequence of) command argument, input is taken from the terminal. If the sentence is given as a (sequence of) command arguments, the normal settings for designating sentence delimiters are ignored, but the setting `word_regular_expression` is still used.
If a sentence is reread, the multiplicity of the sentence is increased by 1. This can, for instance, compensate for occurrences of the same sentence in negative samples. To decrease the multiplicity of sentences or read a negative sample of sentences, use the `unlearn-phrase` command.

B.9 The `load` command

Syntax

`load` [*filename*]

Synopsis

load the database

This command loads sentences, settings, grammatical types and rules from the specified database file, or from the current file if no filename is given. This will undo all changes made since the last time the database was saved.
This command is executed automatically when the Emile program starts, to load the '`grammar.dat`' file (or the file specified with the `--database` command line option).

B.10 The `parse` command

Syntax

`parse` [*filename*] ...

`p` [*filename*] ...

Synopsis

parse sentences from files or terminal input

This command attempts to parse sentences from one or more files, using the rules found for the current grammar. If no filenames are given, input is taken from the terminal (terminated by an end-of-file symbol, usually obtained by pressing Control-D). For the manner in which a file is split into sentences, see the `learn` command.
To find a parsing, the parser may assume that up to parser_tolerance words are not-yet-known expressions of already known types. It attempts to find a parsing which uses as few of these instances as possible. For each sentence, either a single parsing is displayed, or a message that no parsing was found.

B.11 The `parse-phrase` command

Syntax

`parse-phrase` [*phrase*]

`pp` [*phrase*]

Synopsis

parse a specified sentence

This attempts to parse a single sentence. If the sentence is not given as (a sequence of) command argument, input is taken from the terminal. If the sentence is given as a (sequence of) command arguments, the normal settings for designating sentence delimiters are ignored, but the setting `word_regular_expression` is still used.
To find a parsing, the parser may assume that up to parser_tolerance words are not-yet-known expressions of already known types. It attempts to find a parsing which uses as few of these instances as possible. For each sentence, either a single parsing is displayed, or a message that no parsing was found.

B.12 The `parse-type` command

Syntax

`parse-type` *t*|`*` `[`*phrase*`]`

`pt` *t*|`*` `[`*phrase*`]`

Synopsis

parse a phrase of the specified or an arbitrary type

This attempts to parse a single phrase of the specified type. If the phrase is not given as (a sequence of) command argument, input is taken from the terminal. If the phrase is given as a (sequence of) command arguments, the normal settings for designating sentence delimiters are ignored, but the setting `word_regular_expression` is still used.
To find a parsing, the parser may assume that up to parser_tolerance words are not-yet-known expressions of already known types. It attempts to find a parsing which uses as few of these instances as possible. For each sentence, either a single parsing is displayed, or a message that no parsing was found.
A type should be specified as a number, without any enclosing brackets. I.e. to parse a phrase of the base sentence-type '

$$0$$

', you use the command `parse-type 0` *phrase*. If the type is specified as `*`, Emile attempts to find the best parsing for *any* type.

B.13 The `quiet` command

Syntax

`quiet`

Synopsis

turn off logging messages

This sets `verbosity_level` to 0, disabling all non-error logging messages.

B.14 The `save` command

Syntax

`save` [*filename*]

Synopsis

save the database [to a specific file]

This command saves sentences, settings, grammatical types and rules in the specified database file, or in the current file if no filename is given. This will store all changes made since the last time the database was saved.
This command is executed automatically when the Emile program exits, to save the current database file.

B.15 The `script` command

Syntax

`script` [*filename*]...

`batch` [*filename*]...

`.` [*filename*]...

Synopsis

execute commands from a file (or from terminal input)

This command executes commands from one or more files (i.e. adds the sentences to the set of sentences-read-in). Commands are executed as if they were typed on the interactive command line, until an `exit` command or an end-of-file symbol is encountered. The `script` command may be inside a script itself: after the called script finishes, execution continues with the command following the `script` command. If no filenames are given, input is taken from the terminal, which may be useful from inside a script.

B.16 The `set` command

Syntax

`set [setting [=] [value]]`

Synopsis

set the value of a settings variable

This command sets the value of a settings variable. The name of the variable may be separated from the value by spaces, an '=' symbol, or both. For boolean variables, a 'true' value may be specified as '1', 'true' or 'yes': all other values

are interpreted as 'false'. If the value is omitted, the variable is reset to its default value (for non-boolean variables), or toggled (for boolean variables).
If the Gnu `Readline` library is installed on your system, you may use Tab completion to complete variable names.
There are three special invocations of the set command:

`set`

Without any arguments, the `set` command displays the current values of all the settings.

`set support [=] value` This sets *all* support-related values to values derived from the specified value.

`set random_seed [=] value`

This sets the seed for the random number generator to a specific value. The current seed can be displayed by using the `show random_seed` command. This can be used to repeat experiments.

B.17 The `shell` command

Syntax

`shell` [*shellcommand*]

`!` [*shellcommand*]

Synopsis

execute a shell command, or enter a subshell

This command allows you to run shell commands and external programs. For instance, if you are working on a unix system, typing

```
!ls
```

from the Emile command line lists the files in the current work directory.
If you do not give an argument, the Emile program will attempt to start a subshell from which you can run shell commands and external programs.

B.18 The `show` command

Syntax

`show` *arg*

`display` *arg*

`print` *arg*

`p` *arg*

Synopsis

display various data

This command displays various data, such as types found, grammatical rules, statistics, etcetera. Output is commonly piped to an external pager, as in

```
show rules | more
```

The `show` command takes one or more arguments to specify the data to be shown, as follows:

`all`

show all data from the `show contexts`, `show expressions` and `show types` commands.

`assumptions [`*n*`]`

show *n* sentences, not present in the sample of sentences-read-in, that Emile deduces to be grammatical.

`context` *context*

for the specified context, show the expressions and types connected to that context

`context[s]`

for each context, show the expressions and types connected to that context

`dict` *t* `[`*t*`]...`

for the specified type(s), show the one-word expressions connected to that type Types should be specified as numbers with no enclosing brackets, i.e. specify '0' for the whole-sentence type.

`dict`

for each type, show the one-word expressions connected to that type

`expression` *expr*

for the specified expression, show the contexts and types connected to that context

`expression[s]`

for all expressions, show the contexts and types connected to that context

`grammar` *argument*

alias for the `show types` command

`help` *argument*

alias for the help command

`mem`

show estimated memory usage

`mem detailed`

show estimated memory usage in detail

`random_seed`

show current value of randomizer seed

`rule` *t* `[`*t*`]...`

for the specified grammatical type(s), show the derivation rules for that type Types should be specified as numbers with no enclosing brackets, i.e. specify '0' for the whole-sentence type.

`rules`

for each grammatical type that with the current ruleset is relevant to the derivation of sentences, show the derivation rules for that type

`rules all`

for each grammatical types, show the derivation rules for that type

`rules complex`

for each grammatical type, show all rules with two or more type references

`sentences`

show all sentences currently read-in

`settings`

show the current values of the variables

`statistics`

show statistics for the database

`[type]` *t* `[`*t*`]...`

for the specified grammatical type(s), show all contexts, expressions and rules. Types should be specified as numbers with no enclosing brackets, i.e. specify '0' for the whole-sentence type.

`type[s]`

for each grammatical type(s), show all contexts, expressions and rules.

`types by-compression` for each grammatical type(s) (in order of descending compression factor), show all contexts, expressions and rules.

`version`

show the version number of the program (this is an alias for the `version` command).

B.19 The `unlearn` command

Syntax

`unlearn` [*filename*] . . .

`ul` [*filename*] . . .

Synopsis

unlearn sentences from input [or from file]

This command unlearns a (negative) sample of sentences from one or more files (i.e. adds the sentences to the set of negative-sentences-read-in). If no filenames are given, input is taken from the terminal (terminated by an end-of-file symbol, usually obtained by pressing Control-D). For the manner in which a file is split into sentences, see the `learn` command.
If a sentences is reread, the negative multiplicity of the sentence is increased by 1. This can, for instance, compensate for occurrences of the same sentence in positive samples. To decrease the negative multiplicity of sentences or read a positive sample of sentences, use the `learn` command.

B.20 The `unlearn-phrase` command

Syntax

`unlearn-phrase` [*phrase*]

`ulp` [*phrase*]

Synopsis

unlearn 1 sentence from input [or as given]

This command learns one sentence (i.e. adds it to the set of negative-sentences-read-in). If the sentence is not given as (a sequence of) command argument, input is taken from the terminal. If the sentence is given as a (sequence of) command arguments, the normal settings for designating sentence delimiters are ignored, but the `word_regular_expression` setting is still used.
If a sentences is reread, the negative multiplicity of the sentence is increased by 1. This can compensate for occurrences of the same sentence in positive samples, i.e. you can effectively ‘undo’ learning a file by unlearning it, and vice versa.
To decrease the negative multiplicity of sentences or read a positive sample of sentences, use the `learn` command.

B.21 The `verbose` command

Syntax

`verbose` [*n*]

Synopsis

turn on comments [or set verbosity level]

This increases `verbosity_level` by 1 or sets it to the specified value. The higher the value, the more detailed the logging messages. Setting `verbosity_level` to values higher than 2 is not recommended except for debugging purposes.

B.22 The `version` command

Syntax

```
version
```

Synopsis

this displays the version number of the program.

Appendix C

The Algorithms of EMILE

At this moment of writing, the EMILE program consists of about 7800 lines of C++-code. However, most of that is code for data type representation, user interface, utility functions, various optimizations, etcetera: the algorithms themselves are fairly simple. Each of the sections of this chapter focuses on a different algorithm used in EMILE. For each algorithm, a synopsis is given, as well as explicit pseudo-code, and a summary of the constants controlling the algorithm that can be set by the user.
In many of the pseudo-code algorithms, the phrase `for each` occurs to indicate iterating over elements of a collection. The *order* in which the algorithm iterates over these elements should be considered to be nondeterministic. EMILE uses various optimization considerations and a random number generator to determine the actual order.

C.1 Gathering context/expression pairs.

Synopsis

As described in section one-dimensional-clustering-section, EMILE maintains a matrix M of the context/expression pairs it has encountered in positive and negative samples. This routine updates this matrix, given text from some input I.

Algorithm

```
sub learn_sentences($I$)
    while (there is input to be read) do
        read the input $I$ up to the next end-of-sentence marker;
        set $s$ := the sentence read, converted to a sequence of words;
        if (length($s$) $\leq$ maximum_sentence_length) then
            for each triple $(c_l, e, c_r)$ with $c_l$^$e$^$c_r = s$ do
                increment $M(c_l$^"(.)"^$c_r, e)$
            end for
        end if
    end while
    set $M^+ := \{(c,e) \mid M(c,e) > 0\}$;
    set $M^- := \{(c,e) \mid M(c,e) < 0\}$;
end sub

sub unlearn_sentences($I$)
    while (there is input to be read) do
        read the input $I$ up to the next end-of-sentence marker;
        set $s$ := the sentence read, converted to a sequence of words;
        if (length($s$) $\leq$ maximum_sentence_length) then
            for each triple $(c_l, e, c_r)$ with $c_l\frown e\frown c_r = s$ do
                decrement $M(c_l\frown$“(.)”$\frown c_r, e)$
            end for
        end if
    end while
    set $M^+ := \{(c,e) \mid M(c,e) > 0\}$;
    set $M^- := \{(c,e) \mid M(c,e) < 0\}$;
end sub
```

Relevant user settings

`maximum_sentence_length`
: sentences longer than this are ignored.

`end_of_sentence_markers`
: a set of characters that mark the end of a sentence.

`allow_multi_line_sentences`
: a boolean variable, indicating whether sentences are allowed to span multiple lines.

`ignore_abbreviation_periods`
: a boolean variable, indicating whether to consider periods following a single letter to be abbreviation periods or end-of-sentence markers.

`end_of_sentence_regular_expression`
: a regular expression that can be used to specify end-of-sentence markers.

`word_regular_expression`
: a regular expression that can be used to specify the words a sentence should be split in.

Notes

If a sentence occurs in both a positive and a negative sample, the two instances cancel one another out. Hence, a sentence counts as positive if it occurs more often in positive samples, and as negative if it occurs more often in negative samples.

For reasons of efficiency, contexts and expressions are not directly used as elements of the matrix. Instead, the actual contexts and expressions are stored in a table, and references to the entries in the table are used in the matrix.

There is a compilation option to put Emile in 'Morpheme Analysis' mode (see chapter A). In this mode, each word (using whitespace as a delimiter) is treated as a separate element of S, and is split into single characters for analysis. The settings related to end-of-sentence marking are ignored.

A sentence is converted into a sequence of words before it is searched for context/expression pairs. Characters not contained in a match for `word_regular_expression` function as word separators where necessary and are otherwise ignored.

An end-of-sentence marker is considered to be part of the sentence it is ending.

C.2 Extracting the grammatical types from the matrix

Synopsis

The program maintains a set G of grammatical types with sufficient support and size, covering the matrix M^+ of positively encountered context/expression pairs (wherever coverable). This routine updates G for any changes in the settings or M^+.

Details

As described in section 2.3, a type $T = (T_C, T_E)$ is specified by sets T_C and T_E of *primary* contexts and expressions. It is considered to have sufficient *support* if a certain percentage (given by the `total_support_percentage` setting) of the context/expression pairs of the block $T_C \times T_E$ as a whole is contained in the matrix of positively encountered context/expression pairs M^+. Furthermore, the same should hold for each individual row or column (with percentages given by the `context_support_percentage` and `expression_support_percentage` settings). Finally, as described in section 2.6, any negatively encountered context/expression pairs within $T_C \times T_E$ actively detract from this support. Formally, T should satisfy the following three conditions:

$$\begin{aligned} &\#(M^+ \cap (T_C \times T_E)) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (T_C \times T_E)) \\ &\quad \geq \#(T_C \times T_E) \cdot \texttt{total\_support\_percentage/100} \qquad \text{(C.1)} \\ \forall c \in T_C : &\#(M^+ \cap (\{c\} \times T_E)) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (\{c\} \times T_E)) \\ &\quad \geq \#T_E \cdot \texttt{context\_support\_percentage/100} \qquad \text{(C.2)} \end{aligned}$$

$$\forall e \in T_E : \#(M^+ \cap (T_C \times \{e\})) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (T_C \times \{e\})) \geq \#T_C \cdot \texttt{expression\_support\_percentage/100} \quad (C.3)$$

If the `use_multiplicities` setting has a value of `true`, these conditions are modified to take into account that T_E, T_C and M^+ are multisets. As the modified conditions are rather complex and not very insightful, we will omit them.

The program maintains a set G of grammatical types with sufficient support, with contexts of length at most `maximum_primary_context_length`, and expressions of length at most `maximum_primary_expression_length` All these types are of maximal size (under the constraint of having sufficient support), and all are at or above a certain minimum size (as indicated by `minimum_contexts_per_type` and `minimum_expressions_per_type`).

An element $(c, e) \in M^+$ is considered *covered* by a type T if $(c, e) \in T_C \times T_E$. This routine updates and enlarges G so that every element $(c, e) \in M^+$ (that can be covered by a type of minimum size) is covered by a type in G.

Algorithm

```
sub expand_grammar($G$)
    for each $T \in G$ do
        call enlarge_grammatical_type($T$);
        if (($\#T_C <$ minimum_contexts_per_type)
                or ($\#T_E <$ minimum_expressions_per_type)) then
            remove $T$ from $G$;
        end if
    end for
    for each $(c, e) \in M^+$ do
        if $(\neg\exists T \in G : (c, e) \in T_C \times T_E)$ then
            set $T := (\{c\}, \{e\})$;
            call enlarge_grammatical_type($T$);
            if (($\#T_C \geq$ minimum_contexts_per_type)
                    or ($\#T_E \geq$ minimum_expressions_per_type)) then
                insert $T$ into $G$;
            end if
        end if
    end for
end sub

sub enlarge_grammatical_type($T$)
    repeat
        set $X := \{c'$ a context $\mid (T_C \cup \{c'\}, T_E)$ has sufficient support,
                        $\text{length}(c) \leq$ maximum_primary_context_length$\}$,
                $\cup\{e'$ an expression $\mid (T_C, T_E \cup \{e'\})$ has sufficient support,
                        length($e$) $\leq$ maximum_primary_expression_length};
    if ($X \neq \emptyset$) then
        nondeterministically select $x$ from $X$;
        if ($x$ is a context) then
            insert $x$ into $T_C$;
        else
            insert $x$ into $T_E$;
            end if
        end if
    until ($X = \emptyset$);
end sub
```

Relevant user settings

`maximum_primary_context_length`

`maximum_primary_expression_length`

maximum length of potential primary contexts and expressions.

`total_support_percentage`

`context_support_percentage`

`expression_support_percentage`

the support required from the matrix for the submatrix of the type as a whole, and for individual contexts and expressions of each type.

`negative_entry_support_multiplier`

the relative importance of negative samples.

`minimum_contexts_per_type`

`minimum_expressions_per_type`

types with fewer contexts or expressions than indicated by these settings are discarded.

`use_multiplicities`

whether or not to make more use of multiplicities, by treating all sets as multisets and modifying the conditions types should satisfy (computationally expensive).

Notes

The selection of x from X is not nondeterministic, but based on the amount of support that would be added to the grammatical type.

Due to the discarding of types of very small size, some elements $(c, e) \in M^+$ nay be uncoverable.

The type [0] is always set to the type of whole sentences, with '((),())' as a secondary context and all positively encountered sentences as secondary expressions.

The lower the settings for required support, the *larger* the types will tend to be. Usually this will decrease the *number* of types found.

Complexity

The `enlarge_grammatical_type` routine maintains a significant amount of auxiliary data to avoid having to repeat calculations to collect the set X. Initialization of the auxiliary data has an execution time of order $O(\#\{(c,e) \in M^+ \mid c \in T_C \vee e \in T_E\})$, while each iteration of the `repeat..until` loop has an average-case execution time of order $O(\#(T_C \cup T_E))$ and a worst-case execution time of order $O(\#\{(c,e) \in M^+ \mid c \in T_C \vee e \in T_E\})$.

C.3 Eliminating superfluous types

Synopsis

It is possible that the contribution of some type to the coverage of G is made (nearly) superfluous by types found later, i.e. most or all of the context/expression pairs that are covered by that type, are also covered by other types of G. This routine eliminates such types.

Algorithm

```
sub eliminate_superfluous_types($G$)
    set cover(*) $= 0$;
    for each $T \in G$ do
        for each $(c,e) \in M^+ \cap (T_C \times T_E)$ do
            increment cover($c$,$e$);
        end for
    end for
    for each $T \in G$ do
        if ($\#\{(c,e) \in M^+ \cap (T_C \times T_E) \mid \text{cover}(c,e) = 1\}$
                $<$ type_usefulness_required) then
            remove $T$ from $G$;
            for each $(c,e) \in M^+ \cap (T_C \times T_E)$ do
                decrement cover($c$,$e$);
            end for
        end if
    end for
end sub
```

Relevant user settings

`type_usefulness_required`

this variable determines how useful a type has to be (in terms of contributions to the coverage of G) in order to not be discarded.

Notes

The types are checked in order of increasing absolute total support. This means that if the matrix can be covered by either a lot of small types or a single big one, probability favors the latter result.

C.4 Identifying characteristic, secondary and negative contexts and expressions

Synopsis

As described in section 2.4, Emile uses the short expressions and contexts it found for each type in the previous algorithms to find, first, characteristic expressions and contexts, and then (possibly long) ‘negative’ and ‘secondary’ ones.

Details

For each type T, this routine finds

the set T_E^{ch} of characteristic expressions of T
: those expressions which only appear with contexts of no type or of type T.

the set T_C^{ch} of characteristic contexts of T
: those contexts which only appear with expressions of no type or of type T.

the sets T_E^* of characteristic* expressions of T
: the characteristic expressions of T if there are any, otherwise defaulting to the primary expressions of T.

the sets T_C^* of characteristic* contexts of T
: the characteristic contexts of T if there are any, otherwise defaulting to the primary contexts of T.

the set T_E^{se} of secondary expressions of T
: the primary expressions of T, and those expressions e that occur with its characteristic* contexts often enough, i.e. that satisfy

$$\begin{aligned}\#(M^+ \cap (T_C^* \times \{e\})) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (T_C^* \times \{e\})) \\ \geq \quad \#T_C^* \cdot \texttt{sec\_context\_support\_percentage/100} \qquad \text{(C.4)}\end{aligned}$$

the set T_C^{se} of secondary contexts of T
: the primary contexts of T, and those contexts c that occur with its characteristic* expressions often enough, i.e. that satisfy

$$\begin{aligned}\#(M^+ \cap (\{c\} \times T_E^*)) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (\{c\} \times T_E^*)) \\ \geq \quad \#T_E^* \cdot \texttt{sec\_expression\_support\_percentage/100} \quad \text{(C.5)}\end{aligned}$$

the set T_E^- of negative expressions of T

those expressions e that occur negatively with its characteristic* contexts often enough, i.e. that satisfy

$$\begin{aligned}\#(M^+ \cap (T_C^* \times \{e\})) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (T_C^* \times \{e\}))& \\ < \; 0 & \qquad \text{(C.6)}\end{aligned}$$

the set T_C^- of negative contexts of T

those contexts c that occur negatively with its characteristic* expressions often enough, i.e. that satisfy

$$\begin{aligned}\#(M^+ \cap (\{c\} \times T_E^*)) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (\{c\} \times T_E^*))& \\ < \; 0 & \qquad \text{(C.7)}\end{aligned}$$

Algorithm

```
sub identify_characteristic_and_secondary_aspects(G)
    for each $T \in G$ do
        set $T_C^{ch} := \emptyset$
        for each context $c \in T_C$ do
            if $\forall e : [(c, e) \in M^+ \rightarrow (e \in T_E \vee \neg\exists U \in G : e \in U_E)]$ then
                insert $c$ into $T_C^{ch}$;
            end if
        end for
        set $T_E^{ch} := \emptyset$
        for each expression $e \in T_E$ do
            if $\forall c : [(c, e) \in M^+ \rightarrow (u \in T_C \vee \neg\exists U \in G : c \in U_C)]$ then
                insert $e$ into $T_E^{ch}$;
            end if
        end for
        if $(T_C^{ch} \neq \emptyset)$ then
            set $T_C^* := T_C^{ch}$;
        else
            set $T_C^* := T_C$;
        end if
        if $(T_E^{ch} \neq \emptyset)$ then
            set $T_E^* := T_E^{ch}$;
        else
            set $T_E^* := T_E$;
        end if
        set $T_C^{se} := T_C$;
        set $T_C^- := \emptyset$
        for each context $c$ do
            set $n := \#(M^+ \cap (\{c\} \times T_E^*))$
                    $- \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (\{c\} \times T_E^*))$;
```

```
          if ($n \geq \#T_E^* \cdot$ sec_context_support_percentage/100) then
             insert $c$ into $T_C^{se}$;
          else if ($n < 0$) then
             insert $c$ into $T_C^-$;
          end if
       end for
       set $T_E^{se} := T_E$;
       set $T_E^- := \emptyset$
       for each expression $e$ do
          set $n := \#(M^+ \cap (T_C^* \times \{e\}))$
                $-$neg_entry_supp_mult $\cdot \#(M^- \cap (T_C^* \times \{e\}))$;
          if ($n \geq \#T_C^* \cdot$ sec_expression_support_percentage/100) then
             insert $e$ into $T_E^{se}$;
          else if ($n < 0$) then
             insert $e$ into $T_E^-$;
          end if
       end for
    end for
end sub
```

Relevant user settings

`secondary_context_support_percentage`

`secondary_expression_support_percentage`
  the support required for the secondary contexts of each type.

`scsp_for_no_characteristics`

`sesp_for_no_characteristics_percentage`
  settings to be used instead of the normal secondary support settings whenever a type has no characteristic expressions or contexts.

`negative_entry_support_multiplier`
  this variable indicates the relative importance of negative samples.

Notes

If a type has no characteristic expressions, the characteristic* expressions default to the primary expressions. If required support percentages are low, this can result in a single expression being taken as indicative of several types, which could cause an unwieldy total number of secondary contexts to be found. To prevent this, EMILE will never consider a single non-characteristic expression to suffice as indicator of the usability of a type. Furthermore, the setting `scsp_for_no_characteristics`, is provided, to be used instead of the setting `secondary_context_support_percentage` when a type has no characteristic expressions. Similarly, when a type has no characteristic contexts, EMILE will not consider a single non-characteristic context to suffice as indicator of the usability of a type, and the setting `sesp_for_no_characteristics_percentage` is used instead of `secondary_expression_support_percentage` to find secondary expressions.

C.5 Deriving grammatical rules

Synopsis

As described in section 2.5, Emile uses the grammatical types it finds to infer *grammatical derivation rules* of the form:

$$r : [T] \Rightarrow s_0[T_1]s_1[T_2]\ldots s_k$$

EMILE tries to find a set of rules which are *supported*, are capable of generating the original sample, and cannot easily be reduced in number.

Details

All rules of the form $r : [T] \Rightarrow s_0$ with s_0 a secondary expression of T are automatically considered *supported*. Otherwise, for $k \geq 1$, a rule $r : [T] \Rightarrow s_0[T_1]s_1[T_2]\ldots s_k$ is considered to be *supported* if substituting characteristic expressions for one of its type references results in an already supported rule often enough, or formally, if for some $i \leq k$,

$$\begin{aligned} &\#\{e \in (T_i)^*_E \mid r \text{ with } e \text{ replacing } [T_i] \text{ has support}\} \\ &\qquad \geq \quad \#(T_i)^*_E \cdot \texttt{rule\_support\_percentage} \end{aligned} \tag{C.8}$$

Furthermore, a rule should not directly generate (too many) negative expressions for the type, or formally,

$$\begin{aligned} &\#\{(e_1,\ldots,e_k) \in \Pi_{i=1}^k (T_i)^*_E \mid s_0{}^\frown e_1{}^\frown s_1{}^\frown \ldots {}^\frown s_k \in T_E^{se}\} \\ &-\texttt{neg\_ent\_supp\_mult} \cdot \#\{(e_1,\ldots,e_k) \in \Pi_{i=1}^k (T_i)^*_E \mid s_0{}^\frown e_1{}^\frown \ldots {}^\frown s_k \in T_E^-\} \\ &\qquad \geq \quad \Pi_{i=1}^k \#((T_i)^*_E) \end{aligned} \tag{C.9}$$

An *instantiation* of a rule $r : [T] \Rightarrow s_0[T_1]s_1 \ldots s_k$, $k \geq 0$, is an expression $e \in T_E^{se}$ which can be obtained by replacing the type references $[T_1], \ldots, [T_k]$ in r by expressions $e'_1 \in (T_1)^{se}_e, \ldots, e'_k \in (T_k)^{se}_e$. A rule r for some type $[T]$ is considered to be *covered* by other rules for $[T]$ if all of its instantiations are also instantiations of one or more of the other rules.

EMILE tries to find a set of supported rules which contains no rules covered by other rules, is capable of generating the original sample, and cannot easily be reduced in size. To do this, the program maintains a set of used types V_{used}, which initially contains only the whole-sentence type [0]. Whenever a type is added to V_{used}, the program gathers all supported rules for all types of V_{used} which only use types from V_{used} (note that for this purpose, rules for a type are considered to be using that type). Then rules which are covered by other rules are eliminated, until a set of rules R is obtained in which every rule has at least one instantiation which is not shared with any other rule. The program adds types to V_{used} as long as this will not result in a large increase in the size size(R) of the resulting ruleset (measured in number of words and type references used).

Algorithm

```
sub derive_rules($G, R$)
    for each $T \in G$ do
        set $R_T^{sup} := \{[T] \Rightarrow e \mid e \in T_E^{se}\}$;
        set $R_T^{sup} := R_T^{sup} \cup \{r \mid r$ is supported by $R_T^{sup}$. $r$ uses $[T]$ and only $[T]\}$;
        set $R_T := R_T^{sup}$;
        for each $r \in R_T$ do
            if $(\exists r' \in R_T : r' \neq r \wedge r'$ covers $r)$ then
                remove $r$ from $R_T$;
            end if
        end for
    end for
    set $V_{used} := \emptyset$;
    set $R := \emptyset$;
    set $R^{sup} := \emptyset$;
    set $T_{add} := [0]$;
    repeat
        insert $T_{add}$ in $V_{used}$;
        set $R^{sup} := R^{sup} \cup R_{(T_{add})}^{sup}$;
        set $R := R_{(T_{add})}$;
        for each $T \in G$ do
            if $(T \notin V_{used})$ then
                set $R_T^{add} := \left\{ r \;\middle|\; \begin{array}{c} r \text{ is supported by } R^{sup} \cup R_T^{sup}, \\ r \text{ uses both } [T] \text{ and } [T_{add}], \\ r \text{ uses no types outside } V_{used} \cup \{T\} \end{array} \right\}$;
                set $R_T^{sup} := R_T^{sup} \cup R_T^{add}$;
                set $R_T := R \cup (R_T \cap R_T^{sup}) \cup R_T^{add}$;
                for each $r \in R_T$ do
                    if $(\exists r' \in R_T : r' \neq r \wedge r'$ covers $r)$ then
                        remove $r$ from $R_T$;
                    end if
                end for
            end if
        end for
        if $(\exists T :$ size$(R_T) <$ size$(R) +$ ruleset_increase_disallowed$)$ then
            select $T_{add}$ from $G$ such that $T_{add} \notin V_{used}$ and size$(R_T)$ is minimal
        end if
    until $(\neg\exists T :$ size$(R_T) <$ size$(R) +$ ruleset_increase_disallowed$)$;
    for each $r \in R$ do
        if ($R - \{r\}$ covers $r$) then
            remove $r$ from $R$;
        end if
    end for
end sub
```

Relevant user settings

`rule_support_percentage`
: the support required for a rule before it is considered for inclusion in the grammar.

`rsp_for_no_characteristics`
: setting to be used instead of the normal rule support setting whenever a type has no characteristic expressions.

`ruleset_increase_disallowed`
: one more than the maximum number of words and symbols that using a type may add to the ruleset

`negative_entry_support_multiplier`
: the relative importance of negative samples.

Notes

The sets R_T are actually stored as changes to be applied to R.

For reasons of efficiency this routine only checks whether rules are covered by single other rules, everywhere except at the very end. This decreases computation time, and also allows for some other optimizations.

Covered rules are eliminated in order of increasing complexity. This means that the final result will not contain any rule of which an abstraction exists that uses only types in V_{used} and is supported.

Rules of the form $[T] \Rightarrow [U]$ are only allowed if $\#T_E^{se} > \#U_E^{se}$, to prevent loops.

For types $T \notin V_{used}$, the rules in R_T for $[T]$ are calculated during rule generation, because otherwise it cannot be checked how adding T to V_{used} would affect the size of the ruleset R. These rules can be displayed with the `show rules all` command.

If a type has no characteristic expressions, the characteristic* expressions default to the primary expressions. If required support percentages are low, this can result in a single expression being taken as indicative of several types. To prevent this, EMILE will never consider a single non-characteristic expression to suffice as indicator of the usability of a type. Furthermore, the setting `rsp_for_no_characteristics`, is provided, to be used instead of the setting `rule_support_percentage` when a type has no characteristic expressions. The default value for this setting is 51.

C.6 Short-circuiting superfluous types

Synopsis

If a type $T \in G$ has only a single rule in R, or if there is only one reference to $[T]$, then we can decrease the size of the ruleset, and remove T from the set of used types, by substituting the rules of $[T]$ for all references to $[T]$.

Algorithm

```
sub short-circuit_types($G, R$)
    for each $[T] \in V_{used}$ do
        if ($R$ contains only one rule for $[T]$) then
            let $r \in R$ be the unique rule for $[T]$;
            for each rule $r' \in R$ referring to $[T]$ do
                remove $r'$ from $R$;
                substitute $r$ for $[T]$ in $r'$;
                insert $r'$ into $R$;
            end for
            remove $[T]$ from $V_{used}$;
        end if
        if ($R$ contains only one reference to $[T]$) then
            if (shortcircuiting would decrease size($R$) by at least ruleset_increase_disallowed then
                let $r' \in R$ be the unique rule referring to $[T]$;
                remove $r'$ from $R$;
                for each rule $r \in R$ for $[T]$ do
                    substitute $r$ for $[T]$ in $r'$;
                    insert $r'$ into $R$;
                end for
                remove $[T]$ from $V_{used}$;
            end if
        end if
    end for
end sub
```

Relevant user settings

`ruleset_increase_disallowed`

One more than the maximum number of words and symbols that using a type may add to the ruleset (or discarding a type must subtract from the ruleset)

Notes

Using this algorithm results in a smaller ruleset. However, it also loses some information which could be useful in parsing, for finding the structure of parsed sentences.

C.7 Parsing a sentence

Synopsis

Given a set of rules, we will sometimes want to see whether a given sentence is parseable with those rules. Furthermore, we will want to see what types unknown words must be assigned in order to make the sentence parseable. EMILE can check for parseability while allowing up to a user-settable number of words to be assigned arbitrary types, using a recursive algorithm.

Algorithm

```
function parse_phrase($R$,$s$,$[T]$)
    if $(([T] \Rightarrow s) \in R)$ then
        return $0$;
    else if (length($s$) $= 1$) then
        return $1$;
    else
        set $n := \infty$;
        for each rule $([T] \Rightarrow s_0[T_0]s_1 \ldots s_k) \in R$ do
            for each sequence $(s'_1, \ldots, s'_k)$ with $s_0 \frown s'_1 \frown s_1 \frown \ldots \frown s'_k \frown s_k = s$ do
                set n $= \min(n, \sum_{i=1}^{k}$ parse_phrase$(R, s'_i, T_i))$;
            end for
        end for
        return $n$;
    end if
end function

function parse_sentence($R$,$s$)
    if (parse_phrase$(R, s, [0]) \leq$ parser_tolerance) then
        return true;
    else
        return false;
    end if
end function
```

Relevant user settings

`parser_tolerance`

This setting is the maximum number of words to which Emile will assign or reassign a type 'on the fly' in order to make parsing of a sentence possible.

Notes

Emile can assign types 'on the fly' to single words (not to larger expressions), in order to parse sentences with unknown words. The program keeps track of the number of 'on the fly' type-assignments used in the search: if a particular search cannot possibly result in a better parsing than the best one found up to now, the search is aborted.

Appendix D

The Settings of EMILE

D.1 Overview of settings

EMILE's behavior is controlled by some 30 different settings parameters. This is a lot, but little effort has been made to decrease this, since little is known about the effect and relative importance of different parameters and combinations thereof. Indeed, one of the purposes of the program is to *experiment* with the effect of combinations of different settings, in order to find out which ones are important.

However, in order to make it easier to start experimenting, we have marked the settings which we assume to be most significant with an asterisk (∗) in the overview below. The reader can experiment with these settings first, and then fine-tune with the other settings if desired.

∗ `allow_multi_line_sentences`	whether sentences in an input sample can span multiple lines
`context_support_percentage`	support required for the individual primary contexts of a grammatical type
`database_filename`	name of the current database file
`end_of_sentence_markers`	set of characters which can mark the end of a sentence
`end_of_sentence_regular_expression`	regular expression defining possible end-of-sentence markers
`expression_support_percentage`	support required for the individual primary expressions of a grammatical type
`ignore_abbreviation_periods`	whether end-of-sentence markers should exclude abbreviation periods
∗ `maximum_primary_context_length`	the maximum size of the primary contexts of a grammatical type

∗ `maximum_primary_expression_length`	the maximum size of the primary expressions of a grammatical type
`maximum_sentence_length`	the maximum size of sentences read from the input sample
∗ `minimum_contexts_per_type`	minimum number of primary contexts of a grammatical type
∗ `minimum_expressions_per_type`	minimum number of primary expressions of a grammatical type
`negative_entry_support_multiplier`	relative importance of negative samples in calculating support
`parser_tolerance`	maximum number of words of unknown or altered type when parsing
`random_seed`	current seed value for the random number generator
`rsp_for_no_characteristics`	support required for a rule before it is considered for inclusion in the grammar, for types with no characteristic expressions.
`rule_support_percentage`	support required for a rule before it is considered for inclusion in the grammar, for types with characteristic expressions.
∗ `ruleset_increase_disallowed`	one more than the maximum number of words and symbols that using a type may add to the ruleset
`save_in_ascii_format`	whether the save-file will be in ASCII format
`scsp_for_no_characteristics`	support required for the secondary contexts of a grammatical type with no characteristic expressions
`secondary_context_support_percentage`	support required for the secondary contexts of a grammatical type with characteristic expressions
`secondary_expression_support_percentage`	support required for the secondary expressions of a grammatical type with characteristic contexts
`sesp_for_no_characteristics_percentage`	support required for the secondary expressions of a grammatical type with no characteristic contexts
∗ `support`	pseudo-setting for all required-support-related settings
`total_support_percentage`	support required from the matrix for a type as a whole

`type_usefulness_required`	contribution to coverage required of types
`use_multiplicities`	whether to make more use of multiplicities
`verbosity_level`	verbosity of logging messages
`word_regular_expression`	regular expression defining words

D.2 The `allow_multi_line_sentences` setting

Name

`allow_multi_line_sentences`

Synopsis

whether sentences in an input sample can span multiple lines

Type

boolean

Default value

`true`

Used in

Gathering context/expression pairs (C.1)

If this variable is set to `true`, a sentence can span multiple lines in an input sample. Otherwise, the end of an input line is considered to indicate the end of the current sentence. In either case, the `end_of_sentence_markers` also indicate the end of the sentence, as does an empty line (i.e. two end-of-lines separated by nothing but whitespace).

D.3 The `context_support_percentage` setting

Name

`context_support_percentage`

Synopsis

support required for the individual primary contexts of a grammatical type

Type

numeric

Default value

75

Used in

Extracting the grammatical types from the matrix (C.2)

This variable control the support required from the matrix (as a percentage) for the individual contexts of each type, i.e. how many of the expressions of a type a context of that type should occur with, as described in section 2.3. Formally, types are required to satisfy

$$\forall c \in T_C : \#(M^+ \cap (\{c\} \times T_E)) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (\{c\} \times T_E)) \geq \#T_E \cdot \texttt{context\_support\_pct/100} \quad \text{(D.1)}$$

The lower this value, the larger the size of the types found, and (probably) the lower the number of types found. This setting should be used in conjunction with the `expression_support_percentage` and `total_support_percentage` settings. Lowering only one of these settings will usually have only little effect.

D.4 The `database_filename` setting

Name

`database_filename`

Synopsis

name of the current database file

Type

text

Default value

"grammar.dat", or as given by the '-d' option at startup

Used in

the `load` and `save` commands

This variable contains the name of the current database file. Whenever a `load` or `save` command is given without specifying a database file, this value is used. Conversely, whenever a `load` or `save` command specifies a database file, this variable is set.

D.5 The `end_of_sentence_markers` setting

Name

`end_of_sentence_markers`

Synopsis

set of characters which can mark the end of a sentence

Type

text

Default value

".?!;"

Used in

Gathering context/expression pairs (C.1)

A set of characters that EMILE interprets as marking the end of a sentence in an input sample. The use of periods as end-of-sentence-markers may be modified by the `ignore_abbreviation_periods` setting.
Note that an end-of-sentence marker is considered to be part of the sentence it is ending.

D.6 The `end_of_sentence_regular_expression` setting

Name

`end_of_sentence_regular_expression`

Synopsis

regular expression defining possible end-of-sentence markers

Type

text

Default value

""

Used in

Gathering context/expression pairs (C.1)

With this setting you can specify the end-of-sentence markers for input samples as a regular expression, instead of using the `allow_multi_line_sentences`, `end_of_sentence_markers` and `ignore_abbreviation_periods` settings.
The latter settings are only active if `end_of_sentence_regular_expression` is empty (the default value).
Although this setting provides a very comprehensive mechanism for indicating end-of-sentence markers, using the other settings is often much more intelligible: for instance, the default settings correspond to to the regular expression

```
[!;?]|\n[␣\r\n\t]*\n|\r[␣\r\n\t]*\r|^\.|[^a-zA-Z]\.|[^␣\r\n\t].\.
```

When using regular expressions to mark the end of sentences, newlines have to be explicitly included in the regular expression in order to be taken into account. Note that on non-Unix systems, the '`\n`' symbol refers to the end-of-line marker customary on that system. The carriage return symbol '`\r`' can be used if there is a need to explicitly take into account line endings of non-Unix files when working on a Unix system.
The syntax used for regular expressions is the Extended Regular Expressions syntax as defined in the `regex`(5) Unix man page. Additionally, the standard C escape sequences are recognized, i.e. '`\n`' for newline, '`\t`' for tab etcetera. When setting regular expression variables, the Emile command interpreter passes through backslashes in the original input, so you can type

```
set end_of_sentence_regular_expression = '\n'
```

instead of

```
set end_of_sentence_regular_expression = '\\n'
```

Note that an end-of-sentence marker is considered to be part of the sentence it is ending.

D.7 The `expression_support_percentage` setting

Name

`expression_support_percentage`

Synopsis

support required for the individual primary expressions of a grammatical type

Type

numeric

Default value

75

Used in

Extracting the grammatical types from the matrix (C.2)

This variable control the support required from the matrix (as a percentage) for the individual expressions of each type, i.e. how many of the contexts of a type an expression of that type should occur with, as described in section 2.3. Formally, types are required to satisfy

$$\forall e \in T_E : \#(M^+ \cap (T_C \times \{e\})) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (T_C \times \{e\})) \geq \#T_C \cdot \texttt{expression\_support\_pct/100} \quad \text{(D.2)}$$

The lower this value, the larger the size of the types found, and (probably) the lower the number of types found. This setting should be used in conjunction with the `context_support_percentage` and `total_support_percentage` settings. Lowering only one of these settings will usually have only little effect.

D.8 The `ignore_abbreviation_periods` setting

Name

`ignore_abbreviation_periods`

Synopsis

whether end-of-sentence markers should exclude abbreviation periods

Type

boolean

Default value

`true`

Used in

Gathering context/expression pairs (C.1)

If this setting is `true`, and the period symbol '.' is used as an end-of-sentence-marker, then periods following a single letter in input samples are considered to be abbreviation periods instead of end-of-sentence markers.

D.9 The `maximum_primary_context_length` setting

Name

`maximum_primary_context_length`

Synopsis

the maximum size of the primary contexts of a grammatical type

Type

numeric

Default value

6

Used in

Extracting the grammatical types from the matrix (C.2)

In order to ensure that the set of types found will converge if sufficiently many sentences are read, the search space is limited to primary contexts of bounded size. This setting controls the maximum size (in words) of the contexts which Emile will consider for primary contexts of a type.
Lowering this setting will usually dramatically increase Emile's speed: however, it also decreases Emile's potential to find composite types and complex rules. The value of this setting should be at least as large as the size (in words) of the smallest context of any type you want Emile to find. Useful values are usually 4-7.

D.10 The `maximum_primary_expression_length` setting

Name

`maximum_primary_expression_length`

Synopsis

the maximum size of the primary expressions of a grammatical type

Type

numeric

Default value

6

Used in

Extracting the grammatical types from the matrix (C.2)

In order to ensure that the set of types found will converge if sufficiently many sentences are read, the search space can be limited to primary expressions of bounded size. This setting controls the maximum size (in words) of the expressions which Emile will consider for primary expressions of a type.
Lowering this setting will usually dramatically increase Emile's speed: however, it also decreases Emile's potential to find composite types and complex rules. The value of this setting should be at least as large as the size (in words) of the smallest expression of any type you want Emile to find. Useful values are usually 4-7.

D.11 The `maximum_sentence_length` setting

Name

`maximum_sentence_length`

Synopsis

the maximum size of sentences read from the input sample

Type

numeric

Default value

999

Used in

Gathering context/expression pairs (C.1)

Sentences longer than this are ignored when reading input samples. Very long sentences usually do not contribute much to the rules of a grammar. This setting should have a value at least 3 times as large as the values of the `maximum_primary_expression/context_length` settings, as otherwise Emile will likely miss a few subcontexts it could have used to create rules.

D.12 The `minimum_contexts_per_type` setting

Name

`minimum_contexts_per_type`

Synopsis

minimum number of primary contexts of a grammatical type

Type

numeric

Default value

1

Used in

Extracting the grammatical types from the matrix (C.2)

For purposes of constructing a grammar, types of extremely small size usually are not very interesting. Types with fewer primary contexts than indicated by this setting are discarded. Note that this may cause some elements from the matrix to be uncoverable. Increasing this setting will decrease the number of types found, as well as the time Emile needs to generate the ruleset.

D.13 The `minimum_expressions_per_type` setting

Name

`minimum_expressions_per_type`

Synopsis

minimum number of primary expressions of a grammatical type

Type

numeric

Default value

2

Used in

Extracting the grammatical types from the matrix (C.2)

For purposes of constructing a grammar, types of extremely small size usually are not very interesting. Types with fewer primary expressions than indicated by this setting are discarded. Note that this may cause some elements from the matrix to be uncoverable. Increasing this setting will decrease the number of types found, as well as the time Emile needs to generate the ruleset.

D.14 The `negative_entry_support_multiplier` setting

Name

`negative_entry_support_multiplier`

Synopsis

relative importance of negative samples in calculating support

Type

numeric

Default value

1

Used in

Extracting the grammatical types from the matrix (C.2)

Identifying characteristic, secondary and negative contexts and expressions (C.4)

Deriving grammatical rules (C.5)

This variable indicates the relative importance of negative samples. In essence, every negative context/expression pair counters exactly this number of other positive context/expression pairs, rendering both them and itself nonexistent as far as support for a type or rule is concerned. The higher this value, the more Emile will try to avoid creating types or rules with context/expression combinations which occur in negative samples.
Note that this does not affect the manner in which positive and negative samples containing the same sentence counter one another. Such samples are canceled out on a one-to-one basis.

D.15 The `parser_tolerance` setting

Name

`parser_tolerance`

Synopsis

maximum number of words of unknown or altered type when parsing

Type

numeric

Default value

0

Used in

Parsing a sentence (C.7)

Emile can assign types ‘on the fly’ to single words (not to larger expressions), in order to parse sentences with unknown words. This setting is the maximum number of words to which Emile will assign or reassign a type in this fashion in order to make parsing of a sentence possible.

D.16 The `random_seed` setting

Name

`random_seed`

Synopsis

current seed value for the random number generator

Type

numeric

Default value

machine-dependent

Used in

> wherever the program makes a nondeterministic choice or iterates over the elements of a set in a nondeterministic order, i.e. in all algorithms except 'Gathering context/expression pairs',

Not truly a setting, this variable is the current seed value for the random number generator. It changes after every use of the random number generator (i.e. when parsing, generating sentences, or creating the grammar and ruleset). It can be used to recreate conditions exactly, so as to allow repetitions of experiments.

D.17 The `rsp_for_no_characteristics` setting

Name

> `rsp_for_no_characteristics`

Synopsis

> support required for a rule before it is considered for inclusion in the grammar, for types with no characteristic expressions.

Type

> numeric

Default value

> 51

Used in

> Deriving grammatical rules (C.5)

This variable controls the support required for a rule (as a percentage) before it is considered for inclusion in the grammar, for types with no characteristic expressions. It is a counterpart to the `rule_support_percentage` setting.
If a type has no characteristic expressions, the characteristic* expressions default to the primary expressions. If required support percentages are low, this can result in a single expression being taken as indicative of several types, which could cause an unwieldy number of supported rules to be found. To prevent this, EMILE will never consider a single non-characteristic expression to suffice as indicator of the usability of a type. Furthermore, the setting `rsp_for_no_characteristics` is provided, to be used instead of the `rule_support_percentage` setting when a type has no characteristic expressions.

D.18 The `rule_support_percentage` setting

Name

> `rule_support_percentage`

Synopsis

support required for a rule before it is considered for inclusion in the grammar, for types with characteristic expressions.

Type

numeric

Default value

50

Used in

Deriving grammatical rules (C.5)

This variable controls the support required for a rule (as a percentage) before it is considered for inclusion in the grammar, i.e. how many of the characteristic* expressions of a type can be substituted for one of the type references in the rule, while resulting in an already supported rule, as described in section C.5. Formally, a supported rule $r : [T] \Rightarrow s_0[T_1]s_1[T_2]\ldots s_k$ with $k > 1$ has to satisfy, for some $i \leq k$,

$$\#\{e \in (T_i)^*_E \mid r \text{ with } e \text{ replacing } [T_i] \text{ has support}\} \geq \#(T_i)^*_E \cdot \texttt{rule\_support\_pct} \tag{D.3}$$

The lower this value, the more rules will be found.
This setting is used for types with characteristic contexts. For types without characteristic contexts, the `rsp_for_no_characteristics` setting is used instead.

D.19 The `ruleset_increase_disallowed` setting

Name

`ruleset_increase_disallowed`

Synopsis

one more than maximum number of words and symbols that using a type may add to the ruleset

Type

numeric

Default value

1

Used in

Deriving grammatical rules (C.5)

When creating a ruleset, EMILE starts by using only references to the whole-sentence type, and adds types to the set of used types V_{used} as long as this will not result in a large increase in the size of the resulting ruleset (measured in number of words and type references used). This variable determines how wasteful a type is allowed to be (in terms of the resulting increase in the number of words and type references used in the rules) in order to be used when creating the ruleset.
Setting this to 0 means that only types that actively reduces the size of the ruleset are included, setting this to 1 will include types as long as they don't actually increase the size of the ruleset. Higher values will allow more types to be included (allowing for more structures to be identified when displaying the ruleset or parsing sentences), at the expense of increasing the size of the resulting ruleset.

D.20 The `save_in_ascii_format` setting

Name

`save_in_ascii_format`

Synopsis

whether the save-file will be in ASCII format

Type

boolean

Default value

`false`

Used in

the `save` and `load` commands.

This variable controls whether Emile saves its database in ASCII format or binary format. Saving it in ASCII format will make it easier to manipulate the file externally (by hand or with other programs): however, saving it in binary format (the default) will reduce filesize and improve save/load speed by a factor 2. If you are not going to examine or manipulate the save-file externally, you can leave this setting on `false`.

D.21 The `scsp_for_no_characteristics` setting

Name

`scsp_for_no_characteristics`

Synopsis

support required for the secondary contexts of a grammatical type with no characteristic expressions

Type

numeric

Default value

51

Used in

Identifying characteristic, secondary and negative contexts and expressions (C.4)

This setting controls the support required (as a percentage) for the secondary contexts of a grammatical type with no characteristic expressions. It is a counterpart to the `secondary_context_support_percentage` setting,
If a type has no characteristic expressions, the characteristic* expressions default to the primary expressions. If required support percentages are low, this can result in a single expression being taken as indicative of several types, which could cause an unwieldy total number of secondary contexts to be found.
To prevent this, EMILE will never consider a single non-characteristic expression to suffice as indicator of the usability of a type. Furthermore, the setting `scsp_for_no_characteristics` is provided, to be used instead of the `secondary_context_support_percentage` setting when a type has no characteristic expressions.

D.22 The `secondary_context_support_percentage` setting

Name

`secondary_context_support_percentage`

Synopsis

support required for the secondary contexts of a grammatical type with characteristic expressions

Type

numeric

Default value

50

Used in

Identifying characteristic, secondary and negative contexts and expressions (C.4)

This setting controls the support required (as a percentage) for the secondary contexts of each type, i.e. the number of characteristic* expressions that context should occur with, as described in section C.4. Formally, secondary contexts should satisfy

$$\begin{aligned} \#(M^+ \cap (\{c\} \times T_E^*)) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (\{c\} \times T_E^*)) \\ \geq \quad \#T_E^* \cdot \texttt{sec\_expression\_support\_pct/100} \end{aligned} \tag{D.4}$$

Lower values for this setting will increase the number of secondary contexts found. Note that the effects of this setting are independent of the value of `secondary_expression_support_percentage`. (unlike with the settings for primary contexts and expressions).
This setting is used for types with characteristic expressions. For types without characteristic expressions, the `scsp_for_no_characteristics` setting is used instead.

D.23 The `secondary_expression_support_percentage` setting

Name

`secondary_expression_support_percentage`

Synopsis

support required for the secondary expressions of a grammatical type with characteristic contexts

Type

numeric

Default value

50

Used in

Identifying characteristic, secondary and negative contexts and expressions (C.4)

This setting controls the support required (as a percentage) for the secondary expressions of each type, i.e. the number of characteristic* contexts that expression should occur with, as described in section C.4. Formally, secondary expressions should satisfy

$$\begin{aligned} \#(M^+ \cap (T_C^* \times \{e\})) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (T_C^* \times \{e\})) \\ \geq \quad \#T_C^* \cdot \texttt{sec\_context\_support\_pct/100} \end{aligned} \tag{D.5}$$

Lower values for this setting will increase the number of secondary or expressions found. Note that the effects of this setting are independent the value of `secondary_context_support_percentage`. (unlike with the settings for primary contexts and expressions).
This setting is used for types with characteristic contexts. For types without characteristic contexts, the `sesp_for_no_characteristics` setting is used instead.

D.24 The `sesp_for_no_characteristics_percentage` setting

Name

`sesp_for_no_characteristics_percentage`

Synopsis

support required for the secondary expressions of a grammatical type with no characteristic contexts

Type

numeric

Default value

51

Used in

Identifying characteristic, secondary and negative contexts and expressions (C.4)

This setting controls the support required (as a percentage) for the secondary contexts of a grammatical type with no characteristic expressions. It is a counterpart to the `secondary_expression_support_percentage` setting.
If a type has no characteristic contexts, the characteristic* contexts default to the primary contexts. If required support percentages are low, this can result in a single context being taken as indicative of several types, which could cause an unwieldy total number of secondary expressions to be found. To prevent this, EMILE will never consider a single non-characteristic context to suffice as indicator of the usability of a type. Furthermore, the setting `sesp_for_no_characteristics` is provided, to be used instead of the `secondary_expression_support_percentag` setting when a type has no characteristic contexts.

D.25 The `support` setting

Name

`support`

Synopsis

pseudo-setting for all required-support-related settings

Type

numeric

Default value

50

Used in

the `set support` command

Not truly a setting, this pseudo-variable is used in the ‘`set support`’ command to set all support percentage settings with a single command. If this ‘variable’ is assigned a value n, the following settings are set to n:

```
expression_support_percentage
context_support_percentage
secondary_expression_support_percentage
secondary_context_support_percentage
rule_support_percentage
sesp_for_no_characteristics
scsp_for_no_characteristics
rsp_for_no_characteristics
```

Additionally, `total_support_percentage` is set to $100 - (100 - n)^2/100$.

D.26 The `total_support_percentage` setting

Name

`total_support_percentage`

Synopsis

support required from the matrix for a type as a whole

Type

numeric

Default value

50

Used in

Extracting the grammatical types from the matrix (C.2)

This variable controls the support required from the matrix (as a percentage) for a type as a whole, i.e. how many of the context/expression pairs of a type should occur in the matrix of positively encountered context/expression pairs, as described in section 2.3. Formally, types are required to satisfy

$$\begin{aligned}\#(M^+ \cap (T_C \times T_E)) - \texttt{neg\_entry\_supp\_mult} \cdot \#(M^- \cap (T_C \times T_E)) \\ \geq \quad \#(T_C \times T_E) \cdot \texttt{total\_support\_pct/100}\end{aligned} \qquad (D.6)$$

The lower this value, the larger the size of the types found, and (probably) the lower the number of types found. This setting should be used in conjunction with the `context_support_percentage` and `expression_support_percentage` settings. Lowering only one of these settings will usually have only little effect.

D.27 The `type_usefulness_required` setting

Name

`type_usefulness_required`

Synopsis

contribution to coverage required of types

Type

numeric

Default value

1

Used in

Eliminating superfluous types (C.3)

This variable determines how useful a type has to be (in terms of contributions to the coverage of G) in order to not be discarded, as described in section C.3. Setting this to 0 will prevent types from being discarded, setting this to 1 eliminates only types which do not contribute anything. Setting this to a high value will eliminate all but a few types of large size, which may decrease the time Emile needs to generate the ruleset.

D.28 The `use_multiplicities` setting

Name

`use_multiplicities`

Synopsis

whether to make more use of multiplicities

Type

boolean

Default value

`false`

Used in

Extracting the grammatical types from the matrix (C.2)

Eliminating superfluous types (C.3)

If this boolean variable is set to `true`, Emile will try to make more use of multiplicities of sentences in the matrix, by treating all sets as multisets. In practice this has turned out to be computationally expensive without changing the result much, so by default this option is turned off. Note that this option is only used when computing grammatical types, not when searching for derivation rules.

D.29 The `verbosity_level` setting

Name

`verbosity_level`

Synopsis

verbosity of logging messages

Type

numeric

Default value

1

Used in

all algorithms and commands

This variable controls the verbosity of Emile's logging messages. The higher the value, the more detailed the logging messages. The `quiet` command will set `verbosity_level` to 0, which disables all non-error logging messages. The `verbose` command will increase this setting or set it to the specified value.
The default value of this variable is 1. Setting `verbosity_level` to values higher than 2 is not recommended except for debugging purposes.

D.30 The `word_regular_expression` setting

Name

`word_regular_expression`

Synopsis

regular expression defining words

Type

text

Default value

“”

Used in

Gathering context/expression pairs (C.1)

A regular expression defining words. A sentence is converted into a sequence of words before it is searched for context/expression pairs. Characters not contained in a match for `word_regular_expression` function as word separators where necessary and are otherwise ignored. If this variable is empty, a word is taken to be a nonempty sequence of alphanumeric characters, or a single non-alphanumeric, non-whitespace symbol. This corresponds to the regular expression

```
[a-zA-Z0-9_\']+|[^a-zA-Z0-9_\'␣\r\n\t-]
```

The syntax used for regular expressions is the Extended Regular Expressions syntax as defined in the **`regex`**(5) Unix man page. Additionally, the standard C escape sequences are recognized, i.e. ‘`\n`’ for newline, ‘`\t`’ for tab etcetera. When setting regular expression variables, the Emile command interpreter passes through backslashes in the original input, so you can type

```
set word_regular_expression = '[^\t\n ]*'
```

instead of

```
set word_regular_expression = '[^\\t\\n ]*'
```

Appendix E

Table of Symbols

c - a context

e - an expression

(c, e) - a context/expression pair

s - a sentence

S - a collection of sentences

M - the matrix counting the number of occurrences of context/expression pairs

M^+ - the set of context/expression pairs that have been positively encountered.

M^- - the set of context/expression pairs that have been negatively encountered.

T - a grammatical type

$[T]$ - a reference to type T inside a grammatical rule

$[0]$ - a reference to the grammatical type of whole sentences

G - the current grammar of grammatical types

T_C - the set of primary contexts of the grammatical type T

T_E - the set of primary expressions of the grammatical type T

T_C^{ch} - the set of characteristic contexts of the grammatical type T

T_E^{ch} - the set of characteristic expressions of the grammatical type T

T_C^* - the set of characteristic$*$ contexts of the grammatical type T

T_E^* - the set of characteristic$*$ expressions of the grammatical type T

T_C^{se} - the set of secondary contexts of the grammatical type T

T_E^{se} - the set of secondary expressions of the grammatical type T

T_C^- - the set of negative contexts of the grammatical type T

T_E^- - the set of negative expressions of the grammatical type T

r - a grammatical rule of the form $[T] \Rightarrow s_0[T_1]s_1[T_2] \ldots s_k$

R^{sup} - the current set of supported grammatical rules

R - the current set of supported, not-superseded grammatical rules

V_{used} - the set of types used by the current set of grammatical rules

R_T^{sup} - the set of supported grammatical rules that would result from adding T to V_{used}

R_T - the set of supported, not-superseded grammatical rules that would result from adding T to V_{used}

Bibliography

[1] **P. Adriaans**, *Language Learning from a Categorial Perspective*, PhD thesis, University of Amsterdam, the Netherlands, 1992.

[2] **P. Adriaans**, *Learning Shallow Context-Free Languages under Simple Distributions*, ILLC Research Report PP-1999-13, Institute for Logic, Language and Computation, Amsterdam, the Netherlands, 1999.

[3] **P. Adriaans, M. Trautwein, M. Vervoort**, *The Emile Approach to Grammar Induction put into Practice*, in preparation.

[4] **E. Dörnenburg**, *Extension of the EMILE algorithm for inductive learning of context-free grammars for natural languages*, Master's Thesis, University of Dortmund, United Kingdom, 1997.

[5] **M. Vervoort**, *Games, Walks and Grammars: Problems I've Worked On*, PhD Thesis, University of Amsterdam, the Netherlands, 2000.

[6] **M. van Zaanen**, *ABL: Alignment-Based Learning*, in: *Proceedings of the 18th International Conference on Computational Linguistics (COLING)* (page 961-967), Saarbrücken, Germany, 2000.

[7] **M. van Zaanen**, *Bootstrapping Syntax and Recursion using Alignment-Based Learning*, in: **P. Langley (editor)**, *Proceedings of the Seventeenth International Conference on Machine Learning* (page 1063-1070), Stanford University, United States, 2000.

[8] **M. van Zaanen**, *Bootstrapping Structure into Language: Alignment-Based Learning*, PhD thesis, University of Leeds, United Kingdom, submitted.

[9] **M. van Zaanen, P. Adriaans** *Alignment-Based Learning versus EMILE: A Comparison*, in *Proceedings of the Belgian-Dutch Conference on Artificial Intelligence (BNAIC)*, Amsterdam, the Netherlands, 2001.