Heinrich Heine Universität Düsseldorf

Optimierung von Chart Parsing für Baumgrammatiken durch zusätzliche Faktorisierung

Optimization of chart parsing for tree grammars through additional factorization

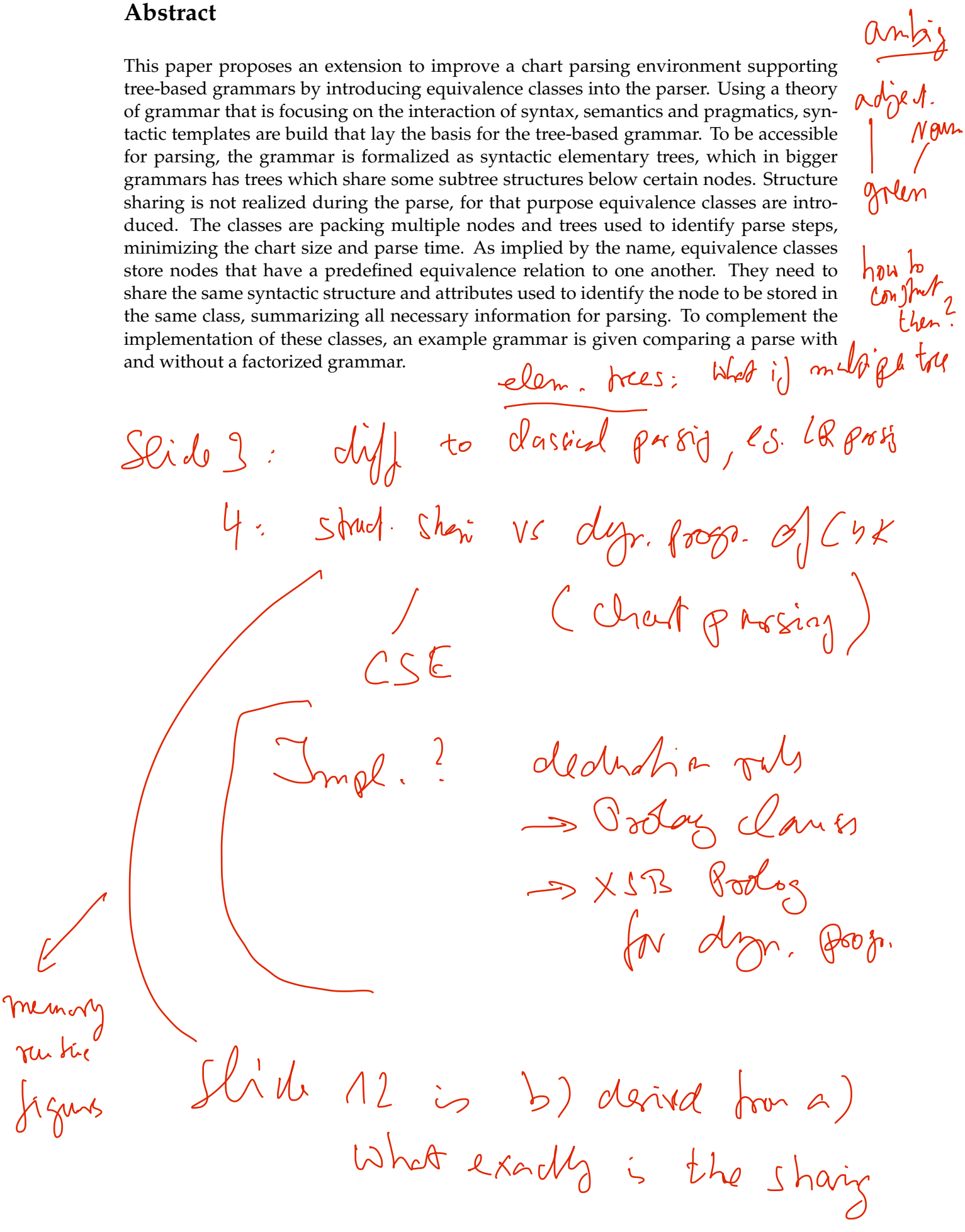
Julia Block
Harffstraße 129a, 40591 Düsseldorf
Matrikelnummer: 2623993

Bachelorarbeit

Date of issue:	21. September 2022
Date of submission:	21. Dezember 2022
Reviewers:	Prof. Dr. Michael Leuschel
	Prof. Dr. Laura Kallmeyer
Supervisors:	Prof. Dr. Laura Kallmeyer
	Dr. Simon Petitjean
	David Arps

Abstract

This paper proposes an extension to improve a chart parsing environment supporting tree-based grammars by introducing equivalence classes into the parser. Using a theory of grammar that is focusing on the interaction of syntax, semantics and pragmatics, syntactic templates are build that lay the basis for the tree-based grammar. To be accessible for parsing, the grammar is formalized as syntactic elementary trees, which in bigger grammars has trees which share some subtree structures below certain nodes. Structure sharing is not realized during the parse, for that purpose equivalence classes are introduced. The classes are packing multiple nodes and trees used to identify parse steps, minimizing the chart size and parse time. As implied by the name, equivalence classes store nodes that have a predefined equivalence relation to one another. They need to share the same syntactic structure and attributes used to identify the node to be stored in the same class, summarizing all necessary information for parsing. To complement the implementation of these classes, an example grammar is given comparing a parse with and without a factorized grammar.



Contents

List of Figures

List of Tables

1 Introduction

The Tuebingen Linguistic Parsing Architecture (TuLiPA, Kallmeyer et al., 2008) is a parsing environment that can be used for parsing and developing tree-based grammars. Having this common framework for natural language processing simplifies sharing resources and comparing formalisms. Currently it uses a parsing schema utilizing single nodes of syntactic trees as pointers to separate parsing steps. As explained further in this thesis, parsing based on nodes can become expensive and time consuming when using large grammars, as structure sharing between trees is not realized. To improve the parser, there are several points to look at, mainly concerning the moment of applying an improvement idea, which can be before or during parsing. Refining during the parse can for example include changing the parsing algorithm. It is however more plausible to first try to improve the input grammar used for parsing.

When trying to improve a grammar, factorization is one of the first techniques that come to mind. Although there are various aspects and implementations, factorization with respect to the parsing domain usually tries to reduce or minimize the grammar that is worked with, therefore optimizing parsing techniques used. This paper puts the focus on grammars formalized as trees and how to reduce the parsing time and size of a chart parser by eliminating redundant parse items. The grammar setting the basis for this proposal is Role and Reference Grammar first introduced by Valin (1993), although it is more commonly known from the book written later (Robert D. Van Valin Jr., 2005). This linguistic theory is heavily based on typological concerns, linking the syntax to semantics and pragmatics. The theory provides syntactic templates, but in order to specify how these templates can be combined to form larger structures and be used in computational analyses, a formalization of RRG is needed. Kallmeyer, Osswald, and Van Valin (2013) proposed a formalization as elementary trees with a Tree Wrapping Grammar (TWG), building on the idea of Tree Adjoining Grammars (TAG Joshi and Schabes, 1997). This tree rewriting system focuses on the composition of elementary templates in RRG, introducing two basic operations to realize argument insertion, long-distance dependencies and adjunction to non-binary trees. Arps (2018) implemented the proposed chart parsing algorithm for RRG presented in Kallmeyer (2017a) as a part of the TuLiPA framework, which is very suited for that extension considering the similarities between RRG and TAG. This part of the code outlines the domain of this proposed factorization and where it is applicable. The focus lies on factorizing the elementary trees before parsing is conducted by realizing ideas of structure sharing from Tomita (1987). The theory packs nodes together which share the same structure below, meaning all daughter nodes are equal and the yield is the same. This lays the base for introducing equivalent classes for subtree structure sharing which fit criteria that is decided upon beforehand, meaning when two nodes share the same structure and additional attributes they are stored in the same equivalent class.

In this paper the basic theories needed for this proposal are introduced first in section 2, namely RRG and TWG, and the idea of subtree sharing is explained. Section 3 gives an overview of the parsing environment TuLiPA as well as an example of the chart parsing algorithm used. The motivation and expected benefits of introducing equivalent classes into the parser are discussed, the classes and implementation is then detailed in the fol-

lowing section 4. The paper follows with an evaluation comparing parses of two example grammars and concluding with a summary and an outlook of possible future improvements.

The code for this thesis can be found at the TuLiPA GitHub Repository: `https://github.com/spetitjean/TuLiPA-frames/tree/factorizing_RRG`, working on the branch `factorizing_RRG`.

2 Introducing basic theories

This section gives a brief introduction to the grammar theory summarized in Robert D. Van Valin Jr. (2005), with focus on the formalization as elementary trees proposed in Kallmeyer et al. (2013). This formalization is used in the parser this proposed factorization tries to improve.

2.1 RRG: Role and Reference Grammar

The grammar theory of Role and Reference Grammar (Robert D. Van Valin Jr., 2005) proposes that any grammatical structure in a sentence can only be fully understood if its semantic and communicative functions are taken into account, because all syntax is relatively motivated by semantic and pragmatic factors (Robert D Van Valin et al., 1997). A key point to RRG is the concept of the *layered structure of the clause* (LSC), which reflects the distinction between arguments, predicates and non-arguments.

The *predicate* is the starting point, being solely encapsulated by the *nucleus* layer. The *core* layer then has the nucleus plus all the arguments of the predicate, sometimes there is an additional *pre-core slot* (PRCS) for wh-words in languages like English. The *periphery* modifies the core with adjunct temporal and locative modifiers. Some languages also have a *left-detached position* (LDP), which is the position of the pre-clausal element in a left-dislocation construction. Each layer can be modified by an *operator*, which can include tense, aspect or modality. In most sentences the predicate is a verb, as seen in Figure 1 taken from Valin (1993, page 67), which shows the layers of a clause with an example. *Give* is the dominating predicate, having two mandatory core arguments, *John* and *to Mary*. The wh-extraction of the object is realized in a pre-core slot. The locative modifier in the periphery *in the library* is attached at the core, and *yesterday* is modifying the whole sentence in clause initial position.

The *logical structure* (LS) of a verb is determined by the group it is assigned to. A set of syntactic and semantic tests decide the group membership of a predicate in a specific sentence. Semantic roles play a big part in the RRG theory, but since frame semantics are not yet included in this proposal of a factorized grammar they are not discussed in this thesis. For further reading consider Valin (1993).

2.2 TWG: Tree Wrapping Grammar for RRG

Tree Wrapping Grammar (TWG) is first introduced in "Tree Wrapping for Role and Reference Grammar" (Kallmeyer, Osswald, and Van Valin, 2013) as a tree rewriting system building on ideas from Tree Adjoining Grammars (TAG) (Joshi and Schabes, 1997). It formalizes RRG further by adding operations to combine the trees described in the previous section, so that the grammar can be used in a parsing schema. TAG and therefore also TWG consists of syntactic *elementary trees*, which can be loosely sorted into initial and auxiliary elementary trees. All elementary trees have a tree structure in which every node represents a syntactic group, e.g. a node with the label „V“ represents the Verb group. Going forth the notation from Joshi and Schabes (1997) is used to denote nodes, meaning

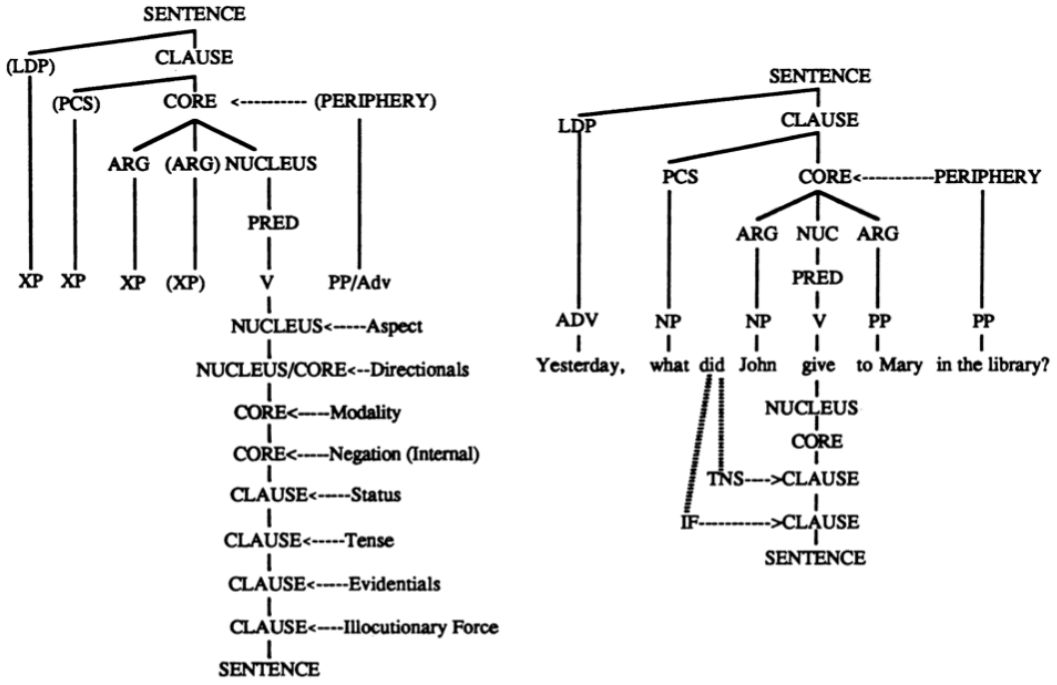


Figure 1: RRG layered structure of the clause (LSC)

interior nodes are all nodes that have daughters and frontier nodes have no daughters.

Initial trees have non-terminal labels on all interior nodes and either terminal labels or a substitution marker on the frontier nodes, which are commonly called leafs. The substitution marker is often an arrow pointing down ↓ next to the label. Lexicalized TAG (LTAG) (Joshi and Schabes, 1997) uses lexicalized trees that have one anchor node which holds the lexical item the tree is associated with. Here the first difference to LTAG is visible, TWG doesn't assume all initial trees are lexicalized. The grammar can also hold tree templates that describe the structure of a sentence without being anchored by a lexical item. Auxiliary trees have the same basic structure as the initial trees, but in addition hold a frontier foot node with a non-terminal label. This node is commonly marked with a star, and is a projection of the root, meaning it has the same label. This foot node later plays a vital part in combining the auxiliary tree with another tree, when using one of TWG's proposed operations. During the parsing process, trees can be combined with different operations to make derived trees. There are two operations in LTAG: substitution and adjunction.

Substitution replaces a marked substitution node in tree a with tree b, provided the root of b has the same label as the substitution node and fits all other imposed criteria, e.g. modality or tense. For every syntactic argument of the predicate, the tree would have a substitution node as a placeholder in the core layer. This is also where the a difference between LTAG and TWG comes up, as LTAG treats clausal complements by adjunction instead of substitution.

Adjoining is used to add modifiers and non mandatory arguments by replacing an interior node with an auxiliary tree. Since the root and the marked foot node of an auxiliary tree must have the same label, they split the interior node they replace and every daughter of that node is added below the foot node.

TWG proposes two new operations to deal with argument insertion and adjunction, *(wrapping) substitution* and *sister adjunction*. For long-distance dependencies wrapping substitution is used to treat more complex causal complements, as shown in the following wh-extraction examples inspired by Arps (2018):

a) Who always ate boar?

b) What did Obelix want to eat?

What is the object of *eat* but is put in clausal-initial position in *b*. Even though there are clauses between the extracted argument and the predicate, their associated relationship stays the same. To realize that in the formalization, *what* should be added to the *eat* tree as an argument. An example composition of *b* can be seen in Figure 2. The complement tree *eat* is split and a subtree is added to the target tree *want*'s substitution node $CO\downarrow$ while the upper part is added above the target trees root *CL*. The splitting point is marked with a dotted edge (*d-edge*) from root to a daughter. That daughter is called *d-daughter* and replaces the substitution node. In this case *CO* from the *eat* tree is the d-daughter.

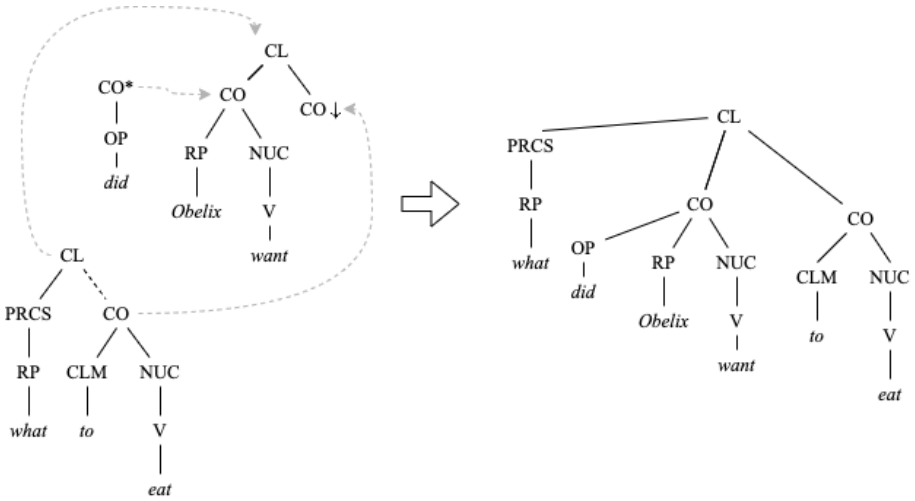


Figure 2: Simple example of wrapping subtitution

Modifiers and functional elements are not counted as arguments and therefore can not be added by substitution, often they are taken to be a periphery of the category they modify. Periphery trees are anchored by periphery elements and their root determines the category they can modify. These trees are added via sister adjunction at any position as new daughters to the modified category. In this example, the tense modifier *did* is sister adjoined at the core of the *want* tree. Trees that are marked for sister adjunction commonly have a star mark $*$ next to their root node. Sister adjunction of functional

elements happens the same way, but these elements can usually not be added at any position under the target node and instead are restricted to only leftmost or rightmost sister adjunction. An example for sister adjunction inspired by Kallmeyer et al. (2013) can be seen in Figure 3.

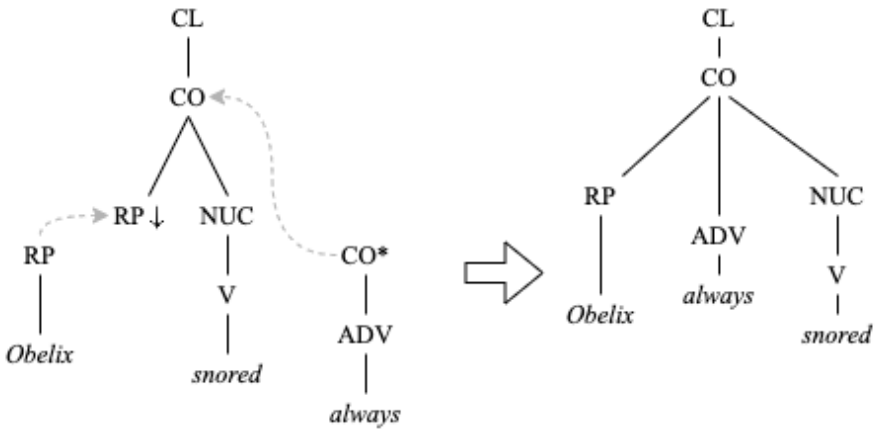


Figure 3: Simple example of substitution and sister adjunction for *Obelix always snored*

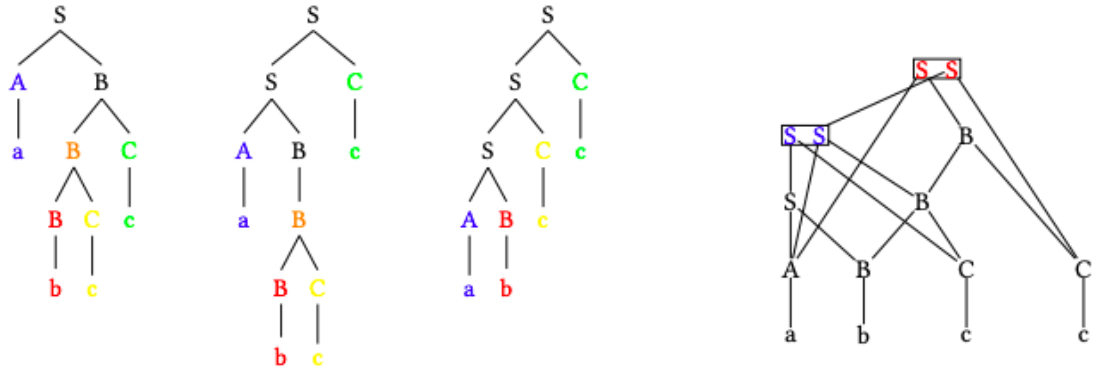
Constraints can be added to nodes, restricting the operations that can be done with it, e.g. no sister adjunction is allowed or only open to specific operators. One way to do this is by adding feature structures to each node, for instance to ensure verb agreement. Usually TWG has specific attribute constraints like number agreement or case marking. Feature structures are discussed in detail in Kallmeyer and Osswald (2017), since this proposal is not using feature structures this part is omitted here. However, introducing feature structures into single trees would bring an even better improvement to the parser, as mentioned again in the conclusion of this paper.

In terms of the outcome, rather than generating strings like most other context-free grammars, TWG generates trees. Therefore the string language of TWG is the yields of all trees that can be derived from the present elementary trees.

2.3 Structure sharing

Elementary trees are based on a large but limited number of syntactic templates and node labels. Even when working with a small grammar, many elementary trees have at least some subtree parts that share the exact same structure. The idea of subtree sharing and local ambiguity packing is taken from Tomita (1987), the goal is to avoid computing partial-results more than once. This can be very time and cost consuming while parsing, especially using a larger grammar that represents a full language, and would cause the number of parse trees produced to grow exponentially. To have a more efficient representation, Tomita proposes two not completely new ideas, *subtree sharing* and *local ambiguity packing*. The idea of subtree sharing in a parser is to group all subtrees from the original elementary trees that share the same structure, depending on the criteria chosen beforehand, so the parser doesn't have to go through the same subtrees more than once. Section

4.1.2 gives deeper insight at possible criteria and detail which ones where chosen for this proposal. Using local ambiguity packing, all nodes of the same category that span the same input are put into a packed node. When combining these two techniques, the outcome is a compact graph of all elementary trees connected and grouped to downsize them, as seen in Figure 4. The example is taken from Kallmeyer (2017b).



(a) Before (b) After

Figure 4: Example of subtree sharing and local ambiguity packing

This idea of structure sharing is laying the base for introducing factorized trees into the parser, taking the form of *equivalence classes*. These classes, as the name suggests, hold objects that are equal when comparing a set of attributes. In this proposal, the objects held by an equivalence class are nodes from elementary trees. Using Tomita's principle of subtree sharing, nodes sharing the same structure are put into the same class. This structure includes the daughters and left sisters of the node, as well as other attributes that are detailed in a later section. The view encapsulated by equivalence classes is always directed below or additionally to the left of the node, mothers or right sisters are not included in the criteria. This is due to the bottom-up left to right nature of the parsing algorithm explained in the following section. For a correct representation of the tree structure, the order of daughter nodes has to be maintained. The data structure holding information about the daughters and left sisters of a class is therefore an ordered collection with duplicates, as sister nodes can have equal classes.

3 The parsing environment: TuLiPA

The Tuebingen Linguistic Parsing Architecture (TuLiPA Kallmeyer et al., 2008) is an open-source parsing environment used to parse several mildly context-sensitive formalisms, including TWG. TuLiPA is implemented in Java and was originally parsing with all input grammars converted into a Range Concatenation Grammar (RCG). Arps (2018) extended a part of the parser to use RRG and the tree composition operations described above, which is the part focused on in this thesis.

3.1 Parsing algorithm

The algorithm used for RRG is proposed in Arps et al. (2019) based on Kallmeyer (2017a) and is a bottom-up chart parsing algorithm. After the input sentence is scanned, the parser moves up through all parse trees. Parse trees are computed by applying the tree composition rules working with *parse items* holding details about the current derivation step. Another similar form of the algorithm is described in Bladier et al. (2020), also using deduction rules. If the parse is successful, one result is a *goal item* that spans the entire input. The output of the parse, i.e. the derivation tree, the derived tree and all elementary trees used in the parse and the derivation steps can be viewed either via a graphical output or stored in an XML file.

3.1.1 Parsing techniques

The parsing techniques used are *parsing as deduction* and *chart parsing*, as described in Kallmeyer (2010). Parsing in connection with deduction is explained in Pereira and Warren (1983) based on previous work on chart parsing by Kay (1986) using the Early context-free parsing algorithm (Earley, 1970), a special case of chart parsing. Rather than using pseudocode to describe the algorithm in detail and already imposing some implementational decisions, deduction rules are used to formalize it, making it usable for different control and data structures. Deduction rules usually consist of three parts, the *antecedent*, *consequent* and *side conditions*, and are represented as follows:

$$\frac{\textit{antecedents}}{\textit{consequents}} \quad \textit{side conditions}$$

All antecedents need to be present and all side conditions must be true, only then can the consequent also be interpreted as true. In our case with TuLiPA, parse items make up the antecedents and consequents, they are detailed further in section 3.1.2. Parse items hold information about the current parse step and which node of which elementary tree is being processed in that step. Let's assume the current node being processed is done and the parser wants to move up to the mother node for further analysis. As an example the *did* tree from Figure 2 is used. Assuming the parser is currently looking at the *did* node as antecedent item, processing it, and after that wants to move up to the mother node *OP* as a consequent item. As a side condition, *did* can not have any sister nodes that have not been processed yet. Since the condition is met and the node has been visited, all constraints are true and therefore the consequent item *[OP]* can be build.

$$\frac{[\textit{did}]}{[OP]} \quad \textit{did has no sister nodes}$$

Rules that have no antecedents and can therefore usually be deduced at the beginning of parsing are called *axioms*, and the wanted result for a successful parse is a *goal item* that can be traced back recursively to the axiom.

The natural language domain has a lot of ambiguity and can often deduce more than one parse tree for a given input sentence. A repercussion of that is that similar parse trees

are computed repeatedly and used at different stages of the parse. To avoid using the same derivation step more than once, they are often stored in a *chart* and an *agenda* in the form of parse items. As stated before, included in these items are information about the current node and elementary tree of the derivation step, and which tree rewriting rule was used to accomplish this parse tree. The agenda lists all items that have not yet been examined during the parse and controls the flow of item deduction. At every derivation step, the next item is removed from the agenda and deduction rules are applied to form new items if possible. That process is repeated until the agenda is empty. More than one deduction rule can be applied to the current item. The chart stores all new items once they are deduced and doesn't allow duplicates. When a new item is deduced, it is checked if the item is already present in the chart. If not, it is added to both the chart and the agenda to be looked at later. If it is, there might be different antecedents than the ones stored and a new set of *backpointers* are added to the existing item and it is not added to the agenda. Backpointers are storing all antecedents that were used to build an item to allow for back tracking in later stages of the parse. For extraction of the resulting parse trees, starting at the goal items, recursively go through the backpointers of every item they are deduced from.

3.1.2 Parse items

To compactly represent all knowledge of the current derivation step and also see what has already been derived, TuLiPA uses parse items. The standard information an item holds is made up of:

- What part of the input sentence it covers
- The position in the derived tree, in this case that would be the Gorn address of the node and the elementary tree it belongs to

Additional important information is attached to the item:

- Pointers to the items it was deduced from
- Any additional details specified in the parsing schema, e.g. a probability or weight when working with statistical parsing algorithms

The input sentence w with $|w| = n$ is indexed starting at 0:

$$w = {}_0\, Obelix\, {}_1\, always\, {}_2\, snored\, {}_3$$

The first items in the parse are created from scanning the words in the input sentence and collecting all elementary trees that have a fitting lexical item corresponding to the words. From these trees, initial parse items are created that point to the lexical node in the tree and are put into the chart at the position of the word they hold. At the beginning of every derivation step, one item gets popped off the agenda and depending on the parse one or more new items are added again.

The parse item looks as follows:

$$[\gamma, p_t, i, j, \langle\langle f_1, g_1, X_1\rangle, ...\rangle, ws?]$$

- γ is the elementary tree
- p is the Gorn address of the node in γ, where the address $(p \cdot m)$ denotes the m-th daughter of p. Root nodes are represented with ϵ.
- The subscript $t \in \{\top, \bot\}$ is referring to the position of the node. BOT position $\bot$ items consider the span below p, items in TOP position $\top$ also consider the span of all left sisters of p
- $ws? \in \{yes, no\}$ is a truth value and indicates whether wrapping substitution is still possible at this node, avoiding double wrapping
- i and j are indices marking the start and end of the item's span, where $0 \leq i \leq j \leq |w| = n$
- $\langle\langle f_1, g_1, X_1\rangle, ...\rangle$ is a list of gaps, where f_1, g_1 with $i \leq f_1 \leq g_1 \leq j$ are marking the start and end span of the d-daughter of a wrapping operation and X_1 is the node label

3.1.3 Deduction rules

This section closely follows Arps (2018). The axiom items are generated first with the Scan-rule that takes no antecedents, as there are no items yet:

Scan: $\dfrac{}{[\gamma, p_\bot, i, i+1, \langle\rangle, no]} \quad label(\gamma, p) = w_{i+1}$

A node's position can be changed from $\bot$ to $\top$ if the node has no left sisters.

No-Left-Sister: $\dfrac{[\gamma, (p \cdot 1)_\bot, i, j, \Gamma, ws?]}{[\gamma, (p \cdot 1)_\top, i, j, \Gamma, ws?]}$

To combine two sister nodes, the left node $(p \cdot m)$ must be in $\top$-position while the right node $(p \cdot (m+1))$ is still in $\bot$-position.

Combine-sisters: $\dfrac{[\gamma, (p \cdot m)_\top, i_1, i_2, \Gamma_1, no], [\gamma, (p \cdot (m+1))_\bot, i_2, i_3, \Gamma_2, no]}{[\gamma, (p \cdot (m+1))_\top, i_1, i_3, \Gamma_1 \circ \Gamma_2, no]}$

When the rightmost daughter $(p \cdot m)$ is reached and the item is in $\top$-position, the parser moves up to p. If p is a d-daughter, *ws?* is set to *yes*.

Move-up: $\dfrac{[\gamma, (p \cdot m)_\top, i, j, \Gamma, no]}{[\gamma, p_\bot, i, j, \Gamma, ws?]} \quad There\ is\ no\ node\ (p \cdot (m+1)); ws? = yes\ if\ p\ is\ a\ d-daughter$

When reaching a root node, all substitution nodes with matching labels are considered as candidates.

Substitution: $\dfrac{[\alpha, \epsilon_\top, i, j, \Gamma, no]}{[\gamma, p\downarrow_\perp, i, j, \Gamma, no]} \quad label(\alpha, \epsilon) = label(\gamma, p\downarrow); \gamma(p\downarrow)\ is\ a\ substitution\ node$

To compute wrapping substitution triggered by a d-daughter, the parser first uses the Predict-wrapping rule to create items from all matching substitution nodes of possible target trees. A gap that holds the span and label of the d-daughter is added. The parser moves up the target trees and once reaching a root node checks if it has the same label as the mother of the d-daughter node. If so, the gap is closed by using the Complete-wrapping (CW) rule and jumping back to the d-daughter.

Predict-wrapping: $\dfrac{[\alpha, p_\perp, i, j, \Gamma, yes]}{[\gamma, p\downarrow_\top, i, j, \langle\langle i, j, label(\gamma, p\downarrow)\rangle\rangle, no]} \quad label(\alpha, p) = label(\gamma, p\downarrow)$

Complete-wrapping: $\dfrac{[\gamma, \epsilon_\top, i, j, \Gamma_1 \circ \langle\langle f_1, f_2, Y\rangle\rangle \circ \Gamma_2, ws?], [\alpha, (p\cdot m)_\perp, f_1, f_2, \Gamma_3, yes]}{[\alpha, (p\cdot m)_\perp, i, j, \Gamma_1 \circ \Gamma_3 \circ \Gamma_2, no]} \quad \begin{array}{l} label(\alpha, p) = label(\gamma, \epsilon); \\ label(\alpha, (p\cdot m)) = Y \end{array}$

Using wrapping substitution with an interior node instead of the root of the wrapped tree is a bit more complex and requires a different composition rule. The mother of the d-daughter node needs to be the root of the wrapping tree, so it can be added via sister adjunction. A backpointer is added to the consequent item, pointing to the antecedent target item. This backpointer serves as a jump back to the target tree after jumping to the wrapping tree and reaching the root node, for that the Jump-back after Generalized CW rule is used.

Generalized CW: $\dfrac{[\gamma, q_\top, i, j, \Gamma_1 \circ \langle\langle f_1, f_2, Y\rangle\rangle \circ \Gamma_2, ws?], [\alpha, m_\perp, f_1, f_2, \Gamma_3, yes]}{[\alpha, m_\perp, i, j, \Gamma_1 \circ \Gamma_3 \circ \Gamma_2, no, [\gamma, q_\top, i, j, \Gamma_1 \circ \langle\langle f_1, f_2, Y\rangle\rangle \circ \Gamma_2, ws?]]} \quad \begin{array}{l} label(\alpha, m) = label(\gamma, q); \\ label(\alpha, m) = Y \\ \alpha(m)\ is\ a\ daughter\ of \alpha(\epsilon) \end{array}$

Jump-back after Generalized CW: $\dfrac{[\alpha, \epsilon_\top, i, j, \Gamma_1, no, [\gamma, q_\top, k, l, \Gamma_2, ws?]]}{[\gamma, q_\top, i, j, \Gamma_1, no]}$

Sister adjunction is handled by two rules displaying two cases, depending on whether the auxiliary tree should be added as the leftmost daughter or further right. The Left-adjoin rule adds an auxiliary tree β as a left sister to the node $\gamma(p\cdot 1)$ and extends the item's span to the left. Whenever β is sister adjoined to the right of at any node $\gamma(p\cdot m > 1)$, the Right-adjoin rule is used. The span of the auxiliary tree is added to the adjoined sister item, expending it to the right.

Left-adjoin: $\dfrac{[\gamma, (p\cdot 1)_\top, i_2, i_3, \Gamma_1, no], [\beta, \epsilon_\top, i_1, i_2, \Gamma_2, no]}{[\gamma, (p\cdot 1)_\top, i_1, i_3, \Gamma_1 \circ \Gamma_2, no]} \quad \begin{array}{l} label(\beta, \epsilon) = label(\gamma, p); \\ \beta\ is\ an\ auxiliary\ tree \end{array}$

Right-adjoin: $\dfrac{[\gamma, (p\cdot m)_\top, i_1, i_2, \Gamma_1, no], [\beta, \epsilon_\top, i_2, i_3, \Gamma_2, no]}{[\gamma, (p\cdot m)_\top, i_1, i_3, \Gamma_1 \circ \Gamma_2, no]} \quad \begin{array}{l} label(\beta, \epsilon) = label(\gamma, p); \\ \beta\ is\ an\ auxiliary\ tree \end{array}$

At the end of parsing there should be at least one goal item, meaning the parse was successful.

Goal item: $[\alpha, \epsilon_\top, 0, |w| = n, \langle\rangle, no] \quad \alpha\ must\ not\ be\ an\ auxiliary\ tree.$

3.1.4 Parse path

Working with TWG, most elementary trees have a frontier node that is marked for anchoring, commonly with a diamond shape next to it. Before the actual parsing starts, these anchor nodes are fitted with a daughter that is the lexical item of the tree, anchoring the tree. After checking the input words, the parser looks for elementary trees which can anchor those as lexical items. Chart and agenda are created and start out empty, the size of the chart corresponding to the number of words in the input sentence.

Starting at the lexical anchor, the parser processes each daughter from left to right and, if applicable, uses one or more of the tree composition operations to form new parse items. When the rightmost daughter of a node is reached, the parser moves up to that node and repeats the same path as before. When a root node is reached, the next step depends on the node type. An auxiliary tree root node marked with a star would trigger the search for an item suitable for sister adjunction. A standard root node is still a possible substitution candidate and substitution would be performed at all substitution nodes with a matching label. An example of wrapping substitution can be seen in 5, taken from Kallmeyer (2017a). When reaching a d-daughter node C_d in tree α_1, all possible substitution nodes are found and the parser continues deducing up from $C\downarrow$ in the wrapped tree γ. The same procedure applies to d-daughter node B_d in tree α_2. The added subtree is kept track of with *gaps*, which store span and label of C_d and B_d in each parse item traversing up to the root of tree γ. Once reaching the root X in γ, the parser jumps back to d-daughter C_d and continues the path to the root of the wrapping tree, closing the gaps during this step.

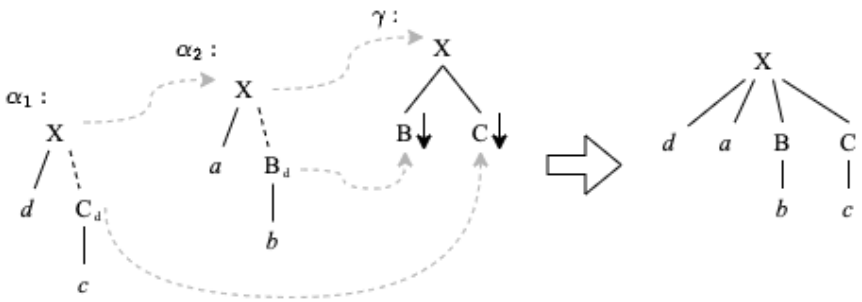


Figure 5: Example of wrapping substitution for *dabc*

3.2 Example chart

Table 1 shows an example chart that depicts the parse from Figure 6 for input sentence *Obelix always snored*, as seen already in section 2.2. Items 1-3 are scans of the input words, followed by all items changing their position to $\top$ in 4-6, as none of the terminal nodes has a left sister. The parser now applies all rules used to traverse up the single elementary trees until reaching the root node in the *Obelix* tree in item 10. There substitution is applied and the substitution node of the *snored* trees is build in item 13. Now usually the next step would be to perform Combine-sister with item 15, seeing as the nodes are sisters in the elementary trees, but as we haven't adjoined the auxiliary tree *always* yet, the

spans don't match up. Once the root of *always* is reached and the node is right-adjoined, it extends the span of item 18 holding the left sister, allowing for the Combine-sisters rule being applied in the next step building item 19. Now the parser simply moves up to the root of the *snored* tree. Item 23 is the goal item, being in a root in $\top$-position spanning the whole input.

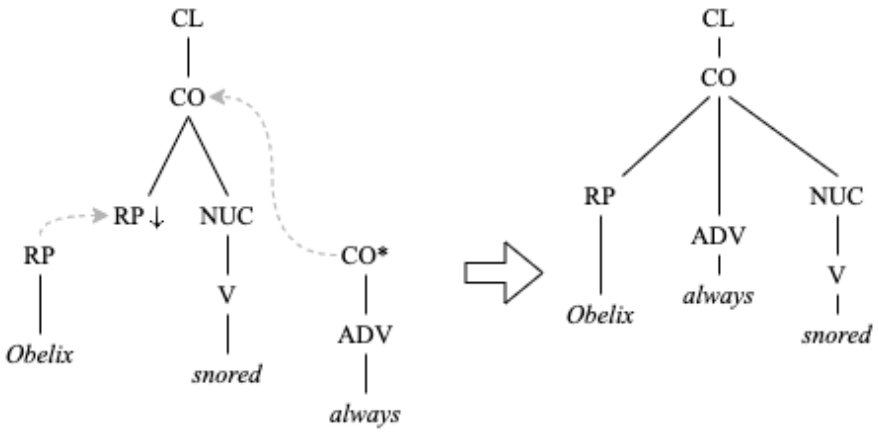


Figure 6: Simple example of substitution and sister adjunction for *Obelix always snored*

3.3 Expected benefits of factorizing

The parser has its own structure sharing, meaning one elementary tree can be used in different contexts without being computed again, but it doesn't realize structure sharing with subtrees between different elementary trees. This means when using bigger grammars with a lot of similarly structured syntactic trees, many irrelevant parse items are computed and enlarge chart size and parse time. Taking again the example trees from Figure 6, assume there is a second tree that is anchored by *snored* and has a slightly different syntactic structure. For the sake of explanation imagine the *CO* node has a third, rightmost daughter $X\downarrow$. As the different structure appears higher up in the tree, during chart parsing both trees are traversed node for node, each one building a separate parse item. Once the *NUC* node is reached, no new items could be deducted in the tree with *X*, as the input sentence *Obelix always snored* has no more words to form the missing daughter substitution tree. Still, many redundant and useless items are build until that point, expanding the chart size.

Using equivalence classes instead of the single node and elementary tree can improve both. An equivalence class is in essence the compact representation of a syntactic subtree structure shared by more than one node. Because one equivalence class can hold multiple nodes, which would all have been different parse items due to the lack of subtree sharing in the parser, there are less items after factorizing and therefore a smaller chart. Having less items to process also shortens the parse time when parsing long or very nested sentences.

	Item	Rules
1	$[Obelix, 1_{\perp}, 0, 1, \langle\rangle, no]$	Scan(Obelix)
2	$[always, 1.1_{\perp}, 1, 2, \langle\rangle, no]$	Scan(always)
3	$[snored, 1.2.1.1_{\perp}, 2, 3, \langle\rangle, no]$	Scan(laughs)
4	$[Obelix, 1_{\top}, 0, 1, \langle\rangle, no]$	NLS(1)
5	$[always, 1.1_{\top}, 1, 2, \langle\rangle, no]$	NLS(2)
6	$[snored, 1.2.1.1_{\top}, 2, 3, \langle\rangle, no]$	NLS(3)
7	$[Obelix, \epsilon_{\perp}, 0, 1, \langle\rangle, no]$	Move-up(4)
8	$[always, 1_{\perp}, 1, 2, \langle\rangle, no]$	Move-up(5)
9	$[snored, 1.2.1_{\perp}, 2, 3, \langle\rangle, no]$	Move-up(6)
10	$[Obelix, \epsilon_{\top}, 0, 1, \langle\rangle, no]$	NLS(7)
11	$[always, 1_{\top}, 1, 2, \langle\rangle, no]$	NLS(8)
12	$[snored, 1.2.1_{\top}, 2, 3, \langle\rangle, no]$	NLS(9)
13	$[snored, 1.1_{\perp}, 0, 1, \langle\rangle, no]$	Substitute(10)
14	$[always, \epsilon_{\perp}, 1, 2, \langle\rangle, no]$	Move-up(11)
15	$[snored, 1.2_{\perp}, 2, 3, \langle\rangle, no]$	Move-up(12)
16	$[snored, 1.1_{\top}, 0, 1, \langle\rangle, no]$	NLS(13)
17	$[always, \epsilon_{\top}, 1, 2, \langle\rangle, no]$	NLS(14)
18	$[snored, 1.1_{\top}, 0, 2, \langle\rangle, no]$	Right-adjoin(17+16)
19	$[snored, 1.2_{\top}, 0, 3, \langle\rangle, no]$	Combine-sister(18+15)
20	$[snored, 1_{\perp}, 0, 3, \langle\rangle, no]$	Move-up(19)
21	$[snored, 1_{\top}, 0, 3, \langle\rangle, no]$	NLS(20)
22	$[snored, \epsilon_{\perp}, 0, 3, \langle\rangle, no]$	Move-up(21)
23	$[snored, \epsilon_{\top}, 0, 3, \langle\rangle, no]$	NLS(22)

Table 1: Parsing "Obelix always snored"

4 Implementing equivalence classes

To factorize the elementary trees used in RRG, all nodes are analyzed after anchoring and put into a corresponding equivalence class. This section shows how the classes are computed and which criteria are used to determine a nodes equivalence class. After that, the necessary changes to parse items and deduction rules are formalized.

4.1 The equivalence algorithm

In terms of changing the algorithm idea not much is altered from Arps (2018) concerning the anchoring and parse path. The factorizer is an additional part of the parser, sitting between anchoring and the beginning of deduction. There, all equivalence classes are built and nodes are sorted. The factorizer itself stores a list of both types of classes, *BOT(bottom)* and *TOP* classes, both modeling the $\perp(bottom)$- and $\top(top)$-position of a node during parsing. The path through the elementary trees has three main parts:

(i) Starting at the root, find the leftmost leaf by traversing down from mother to first daughter. (Top-down, depth-first search)

(ii) Sort node into either fitting existing equivalence classes or create new ones.

(iii) When all daughters of a node p are sorted, move up to p. Otherwise repeat step (i) with p as root and search for lowest unvisited node. (Bottom-up sort)

This is further detailed in Section 4.1.3, before that the elements needed for the algorithm are introduced. First, an overview of the two equivalence class categories which encapsulate two different views of the node, TOP and BOT is given, followed by the criteria for classifying the nodes and the attributes used.

4.1.1 TOP and BOT classes

The division into TOP and BOT (bottom) equivalence classes is in line with the idea of bottom-up left to right parsing, both representing a different view of the node and what part of the tree was already processed. The BOT class looks at everything below a node, meaning when a parse item is created that holds the BOT class (C_{BOT}), all daughter classes $(C \cdot m)$ have been parsed, as well as their daughter classes and so on. A TOP class adds to that by also taking all left sisters of the node it encapsulates into account. When a parse item holding a TOP class is created, it means not only all daughters but also all left sister classes have been parsed.

As seen in Figure 7 every BOT class stores their respective TOP classes, which in turn inherits all attribute fields and adds a few more. Table 2 shows which properties are reserved for TOP classes. The equivalence classes doesn't maintain easily accessible information about the elementary tree structure, since the main idea is to break down trees. Nodes assigned to a class C are stored in `factorizedTrees` with the elementary tree they were in and their address, but trying to access details about for example its mother node would be hard to get. This however is a crucial knowledge for parse operations, such as the Move-up rule. To compensate for that loss of information, every TOP class stores its mother class during the sorting. When a new node is added to a class, if the mother is not already saved it is added to `possibleMothers`. The BOT class of the mother node is stored as well as a truth value indicating whether the current node is the rightmost daughter. This is also important for the Move-up rule, as the parser needs to know whether all daughters of a class have been processed. Although the map of factorized trees is present in both class types they can hold different nodes and are therefore not always equal. Two nodes can share a BOT class, but one might have a left sisters while the other has not, so they would be in different TOP classes and not sharing a map.

4.1.2 Attributes that are compared

As seen in table 2, defining characteristics of a node are considered during sorting. The basic label, which also represents the category of the node, e.g. "V", is the biggest divisor. The lists storing a node's daughters and left sisters allow for duplicate classes and are ordered collections, meaning the order of nodes is maintained. This is especially important for rules applying to sister classes. There are a few additional attributes to be taken into account when sorting TOP classes, as the view of the node it stores also involves comparing all left sisters. Some attributes were needed for the tree composition operations

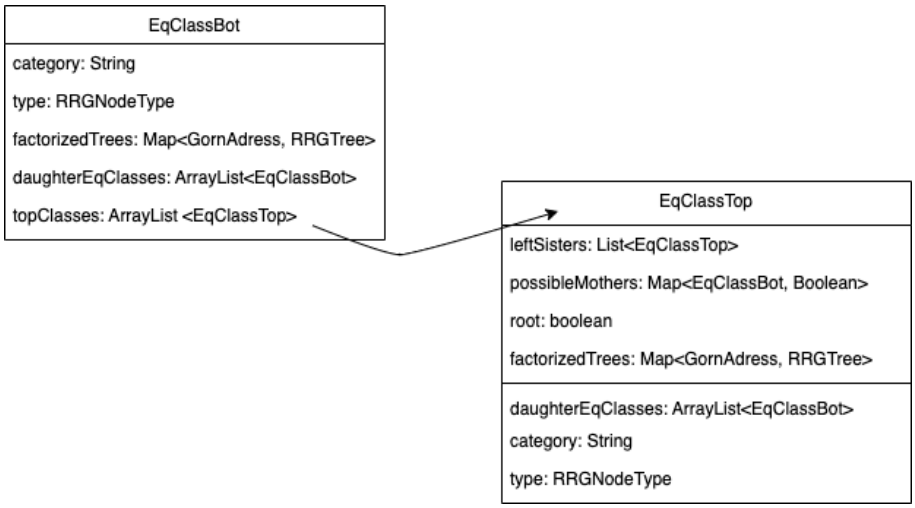


Figure 7: UML diagram of top and bottom equivalence classes

performed during the actual parse, e.g. substitution has to know whether the current equivalence class holds a root item, which is only relevant to TOP classes. Types are also important in both classes for certain deduction rules, a class of type *STAR* for example is needed to trigger sister adjunction. An exemplary sorting of two trees α and β can be seen in table 3, the trees are shown in Figure 8. The first column lists the built BOT classes in the form:

$$\langle C, T, \langle\langle \gamma, (p \cdot m)\rangle, ...\rangle, \langle D_1, D_2, ...\rangle\rangle$$

where

- *C* is the category of the class
- *T* is the node type, e.g. *STD* for standard nodes or *SUBST* for substitution nodes
- $\langle\langle \gamma, (p \cdot m)\rangle, ...\rangle$ is the list of nodes this class encapsulates. γ is the elementary tree and $(p \cdot m)$ the Gorn address
- $\langle D_1, D_2, ...\rangle$ is the list of daughter BOT classes

The second column lists the built TOP classes next to their respective BOT class in the form:

$$\langle \cdots, \langle LS_1, LS_2\rangle, \langle\langle M_1, rightmost?\rangle, ...\rangle, \langle\langle \gamma, (p \cdot m)\rangle, ...\rangle, root?\rangle$$

where

- $\cdots$ is a placeholder for all attributes inherited from the BOT class: category, type and daughters.

- $\langle LS_1, LS_2 \rangle$ are the left sister TOP classes
- $\langle \langle M_1, rightmost? \rangle, ... \rangle$ is a map of possible mother classes, with M_i being the mother BOT class and *rightmost?* is a flag marking the rightmost daughter class
- $\langle \langle \gamma, (p \cdot m) \rangle, ... \rangle$ is a map of the nodes encapsulated by this TOP class, the elements are described above
- *root?* is a flag marking a root node

The lexical items are processed first, both nodes *b* being sorted into the same BOT and TOP class, as they both have no daughters or left sisters and are of the same node type *LEX*. The substitution node *A* has no counterpart in tree β so the BOT and TOP class only hold one node of type *SUBST* with no daughters and left sisters.The mother nodes *B* are in the same BOT class, because they both have one daughter class *b* and are of type *ANCH* as they are anchoring the lexical item. However, they are not in the same TOP class seeing as node B_α has a left sister *A* and B_β has no left sisters. Once the mother nodes are sorted, their class is added to the daughter classes *b* map of possible mothers. The truth value set to *yes* as they are the rightmost daughters. The root nodes of both trees do have the same label, i.e. category *X*, and it could indicate sorting them into the same class. Seeing as X_α has two daughters and additionally X_β is an auxiliary tree root node of type *STAR*, they are neither in the same BOT nor the same TOP class. As they are both root nodes, they have no possible mother classes and the *root?* flag is set to *yes*.

Attributes	**Formalization**	
Category	String; Example: "V", "NP", "John"	Category of the node, can relate to lexical item V (Verb), NP (noun phrase) or to structure and for arguments CL (clause), Periphery
Node Type	RRGNodeType $\in$ {STD, SUBST, ANCH, LEX, STAR, DDAUGHTER}	STD (Standard), SUBST (substitution leaf node), ANCH, LEX, STAR (root node of a tree used for sister adjunction), D-Daughter (d-daughter for wrapping substitution, marks the d-edge)
Daughter nodes	List of BOT classes	Tree structure below this node needs to be equal
Additional attributes of TOP classes		
Left sisters	List of TOP classes	All left sister classes need to be equal
Root	Truth value	Indicates whether this node is a root node

Table 2: Criteria for equivalence class sorting

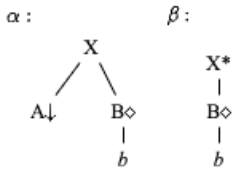


Figure 8: Initial and auxiliary tree to be sorted into classes

BOT classes	**TOP classes**
b $\langle$"b", LEX, $\langle\langle\alpha,(2.1)\rangle,\langle\beta,(1.1)\rangle\rangle,\langle\rangle\rangle$	b $\langle\cdots,\langle\rangle,\langle\langle B,yes\rangle\rangle,\langle\langle\alpha,(2.1)\rangle,\langle\beta,(1.1)\rangle\rangle,no\rangle$
A $\langle$"A", $SUBST$, $\langle\langle\alpha,(1)\rangle\rangle,\langle\rangle\rangle$	A $\langle\cdots,\langle\rangle,\langle\langle X_\alpha,no\rangle\rangle,\langle\langle\alpha,(1)\rangle\rangle,no\rangle$
B $\langle$"B", $ANCH$, $\langle\langle\alpha,(2)\rangle,\langle\beta,(1)\rangle\rangle,\langle b\rangle\rangle$	B_α $\langle\cdots,\langle A\rangle,\langle\langle X_\alpha,yes\rangle\rangle,\langle\langle\alpha,(2)\rangle\rangle,no\rangle$ B_β $\langle\cdots,\langle\rangle,\langle\langle X_\beta,yes\rangle\rangle,\langle\langle\beta,(1)\rangle\rangle,no\rangle$
X_α $\langle$"X", STD, $\langle\langle\alpha,\epsilon\rangle\rangle,\langle A,B\rangle\rangle$	X_α $\langle\cdots,\langle\rangle,\langle\rangle,\langle\langle\alpha,\epsilon\rangle\rangle,yes\rangle$
X_β $\langle$"X", $STAR$, $\langle\langle\beta,\epsilon\rangle\rangle,\langle B\rangle\rangle$	X_β $\langle\cdots,\langle\rangle,\langle\rangle,\langle\langle\beta,\epsilon\rangle\rangle,yes\rangle$

Table 3: Equivalence classes for example trees

4.1.3 Implementation strategy

The core sorting and implementing is done in the class `FactorizingInterface` as seen in detail in Figure 9. During anchoring, the elementary trees are already filtered, the only trees that are passed on to the factorizer either:

(i) have a lexical item that matches a word in the input sentence or

(ii) have no anchors and no lexical nodes

The factorizer loops through the `anchoredTrees` and each tree γ is given over to the method `checkDaughters`. Starting at $\gamma(\epsilon)$ as the current node, the method collects all child nodes. If daughter node $(\epsilon \cdot 1)$ also has children, continue with that as the current node until getting to a leaf $(p \cdot 1)$. Leaves can be either lexical nodes from anchoring or substitution nodes. The method `checkLeafClasses` compares the leaf with all existing BOT classes, considering the attributes mentioned above, and either adds the node to a class or creates a new one. Once the BOT class *bc* is established, the current node is compared against *bc*'s list of already stored TOP classes. Since the factorizer is going from left to right, existing left sisters would have already been processed and are used to sort the node into a TOP class or establish new one and add it to *bc*. $(p \cdot 1)$ is now stored in two views, TOP and BOT. The factorizer now classifies all other daughters $(p \cdot (m > 1))$ of p to move up to p and with the list of daughters sort it into a BOT class. Because of our

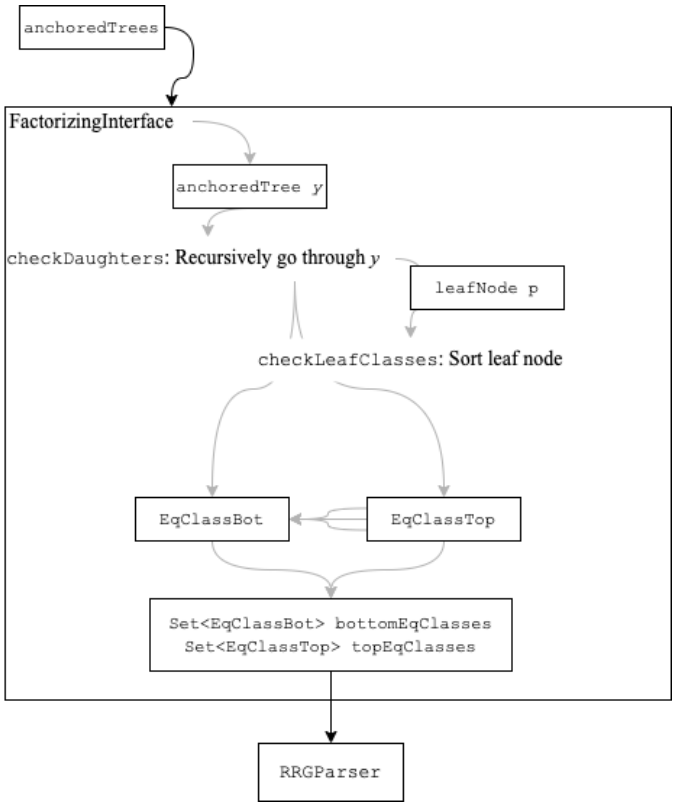


Figure 9: Classes and methods involved in factorizing

left-to-right factorizing schema, p has to be the first or only daughter of its mother node $(t \cdot p)$, meaning there are no left sisters to use while deciding on a TOP class. Figure 10 shows the path of the factorizer, the dotted lines meaning no sorting is done and the next unvisited node is tracked.

Starting in tree α at the root A_α, the `checkDaughters` collects all child nodes and continues with the first daughter C. Since this node is not a leaf, the method is called recursively with C as the new current node and all children are collected again. As the only daughter and leaf node c is given over to `checkLeafClasses`. As there are no BOT or TOP classes built yet, new ones are created, presently only holding c. This was the only daughter node, so the factorizer can move up to C and also create new BOT and TOP classes. C can not be added to b_α's classes, they do not share the same category and have different daughters. Now it is not possible to move up to A_α before also processing the nodes other daughters. First, the next child B_α is analysed and all child nodes are looked at. The only daughter and leaf b_α is handled exactly the same as the first leaf node, building new BOT and TOP classes. Same goes for the mother node B_α and once that is sorted, the root node A_α can also be put into newly built classes. When handling the second tree β is where the criteria of sorting the nodes into classes is really visible. Again starting at the root A_β and recursively going down to b_β, the node is sorted into the existing BOT and TOP class of b_α. They share the same label, daughter and sister nodes are both empty. Moving up to B_β, the node is added to the BOT class of b_α. A new TOP class has to be built however,

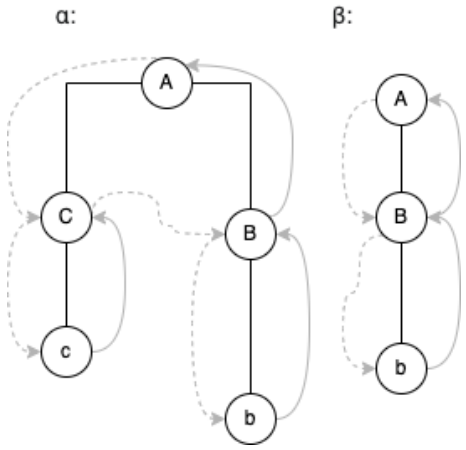


Figure 10: Equivalence class sorting path

as b_β has no left sister in contradiction to b_α. The BOT and TOP classes of the root X_β are clearly also new when compared to X_α, e.g. considering daughters.

4.2 Necessary changes to the parser

This is where the integration of the new equivalence classes into the existing RRG parser happens. As the classes are used to identify a derivation step instead of the single node and elementary tree, the parse items and deduction rules need to be adapted.

4.2.1 New parse items

Most of the elements explained in section 3.1.2 are kept in the new parse items.

$$[C_t, i, j, \langle\langle f_1, g_1, X_1\rangle, ...\rangle, ws?]$$

Visibly the only change to the items is the elementary tree and nodes Gorn address being replaced by an equivalence class *C* with $t \in TOP, BOT$. The class itself contains all necessary details like node type for deciding on the *ws?*-flag. The form of the classes and the elements they hold is detailed in section 4.1.2.

4.2.2 New deduction rules

Changing the parse items means changing all antecedent and consequent items of the deduction rules. The position of the item needs to be known to the parser right at the beginning, deciding over the subset of deduction rules applicable, e.g. a BOT class cannot be an antecedent of the Move-up rule. When trying to use a deduction rule with the current item, there are two major parts:

- Checking the requirements of the deduction rule and comparing antecedents with the current item
- Applying the deduction rule and deducting new consequent parse items

Figure 11 is a snippet taken out of Arps (2018) showing the core classes from TuLiPA involved in parsing RRG. Most changes needed to be done in `RequirementFinder` and `Deducer`, which perform the two derivational operations mentioned above.

`RequirementFinder` checks if the current item fits an antecedent item of a deduction rule, as well as fulfilling all side conditions only relying on the antecedent. Some conditions have restrictions that can only be verified when a consequent candidate is present, e.g. label equality. All requirements like that concerning attributes of the equivalence class in the item are used as filters. A subset of classes is used to form model items mirroring real parse items. The benefit of a model is that not all elements an item is made up by need to be present. Looking again at the Combine-sister rule in section 3.1.3, the span of both antecedent items is restricted, but only the end span of the left and the start span of the right sister are specified. The span of the right sister could extend far to the right, but the rule would still be applicable. When examining a rule possessing two antecedents from which on fits the current item, the chart is searched for the second antecedent represented by a model and can return multiple candidate items. Only when all antecedents are found and all side conditions are met the `Deducer` is triggered. It takes the antecedent item candidates and applies the chosen deduction rule, returning one or more new parse items. The parser then adds the items to chart and agenda or just appends a new backpointer, depending on the novelty of the items. For readability the elementary tree and Gorn address $\langle[\gamma, p_t\rangle]$ in the old parse items are denoted as Δ. As seen in the section before, *C* denotes the equivalence class of an item.

Scan and Goal item These two largely remain unchanged, only now holding equivalence class *C* with *t* = *BOT* and *t* = *TOP* replacing Δ respectively. The Scan rule builds items based on the label of *C* fitting one of the input words. The goal item requires *C* to be a root TOP class.

No-Left-Sister (NLS) Here the parser can utilize the stored list of possible classes contained inside each BOT class. In the antecedent BOT class, a subset of all possible TOP classes that have an empty list of left sisters is determined as consequent candidates.

Combine-sisters This rule is a bit trickier, since the parser can no longer simply use Δ to determine if two items hold sister classes. There are two cases that need to be considered:

(i) The current item *i* is the left sister antecedent with equivalence class C_{TOP}. As only left sisters are stored inside the class, the parser can not access any right sister directly. Instead, a subset $M = \langle m_1, m_2, ...\rangle$ is computed from all possible mothers of C_{TOP} where the truth value is *no*, indicating the class has at least one right sister RS_{BOT}. Next, all immediate right sisters $RS^i_{BOT} \in m_i$ are collected and put into model items with which a fitting right sister antecedent is

searched in the chart. If one is found, the parser proceeds as before and builds a consequent item holding RS_{TOP} with a span extended to the left.

(ii) The current item i is the right sister antecedent with equivalence class C_{BOT}. Since in a BOT class, the left sisters are again not directly accessible and need to be extracted from the possible TOP classes of C_{BOT}. These are filtered by checking whether their stored list of left sister classes has at least one sister LS_{TOP}. Next, all immediate left sisters $LS^1_{TOP}, LS^2_{TOP}, \ldots$ are collected and put into model items with which a fitting left sister antecedent is searched in the chart. If one is found, the parser proceeds as before and builds a consequent item holding C_{TOP} with a span extended to the left.

Move-up This is where the truth value stored with every possible mother in a TOP class is very useful. The list is filtered and all mother classes where the current class is the rightmost daughter are used to build a new item.

Substitution Since the antecedent item holds a TOP class, the parser can easily deduce whether it is a root class and substitution is applicable. The factorizer has many methods for filtering all existing TOP and BOT equivalence classes, in this case all classes of type *SUBST* with a category matching the current class are gathered and from them new consequent items are build.

Predict-wrapping Basically operating the same as the Substitution rule, all fitting substitution classes are collected by the factorizer. Only this time the current class need not be a root class, but a d-daughter (i.e. of type *DDAUGHTER*), and of course as before a gap is added to the consequent item corresponding to the d-daughter item.

Complete-wrapping and Generalized Complete-wrapping The gap element in the parse items remains largely unchanged, simply holding an adapted parse item. The only different requirement to the target antecedent item between the two wrapping cases is whether the TOP class is a root or an interior class. Both are working with the same search from `RequirementFinder` to find either antecedent item in the chart. Finding a filler item for the gap is more complex as again the mother class needs to be accessed to compare labels. The following steps are taken to find filler items:

1) In the chart, find all items that have a span and label that matches the gap and hold a BOT class
2) Get all possible mothers from all TOP classes of the BOT class
3) Filter and collect the mothers for root classes and matching label to the target root

Jump-back after generalized CW Since this rule is only applied to TOP classes, finding root classes is simple, as well as checking for a jump back item. When building the consequent item, the only change is using the equivalence class instead of Δ.

Left- and Right-adjoin Both rules need to first determine whether the current item is the root antecedent or the sister antecedent it adjoins at.

i) The current item class is the root of an auxiliary tree, meaning a TOP root class of type STAR. With a model item constraining the span to either start where the root item ends or vice versa, find all possible sister class candidates in the chart it could left- or right-adjoin to and filter for TOP classes. To assure sister adjunction is applicable, there has to be a mother class of the sister candidate sharing a category with the root class. Since the candidates are TOP classes, mothers are easily accessible and can be compared to the root. Once that condition is met and there is still a suitable sister candidate left, the consequent item is build holding the sister TOP class and depending on the placement of the root item, the span is extended by the root span to either the left or the right.

ii) The current item is a possible sister antecedent. Seeing as the only requirement is not being a d-daughter or root and a TOP class, many items are tried for adjoining, this is however necessary as the root item could have been built before and was removed from the agenda. The `RequirementFinder` searches for possible root candidates for both left- or right-adjunction in much the same way as it does for sister candidates described above. Modeling an item with the right span to find suitable root items marked for sister adjunction and compare the candidates to the current item's possible mother classes. With both antecedent items found apply left- or right-adjunction and built the consequent item.

-.rrg.parser
`RRGParser`: controls the chart parsing algorithm
`SimpleRRGParseChart`: implementation of a `ParseChart`
`SimpleRRGParseItem`: implementation of a `ParseItem`
`RequirementFinder`: checks for antecedent items
`Deducer`: applies deduction rules to items
`Backpointer`: part of the chart, for extraction
`enum Operation`: identifiers for deduction rules

Figure 11: Part of the classes used for parsing RRG in TuLiPA.

4.3 Example chart comparison

The elementary trees used for this example are shown in Figure 12. The two trees with the lexical item *laughs* hold almost the same syntactic structure except for an extra substitution node as the rightmost daughter of the root, for example adding "at"-prepositions e.g. "Mary always laughs at Obelix". Parsing the example sentence "Mary always laughs" using parse items holding elementary tree and node address is seen in table 4. Items 1-4 are scans of the input words, followed by all items changing their position to $\top$ in 5-8, as none of the terminal nodes has a left sister. Since both *laughs*-trees have a fitting lexical anchor and the parser does not concern itself with the full structure of the tree yet, both

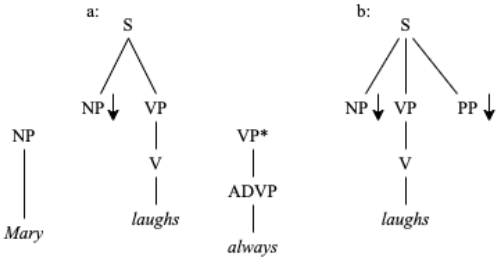


Figure 12: Elementary trees for the chart example

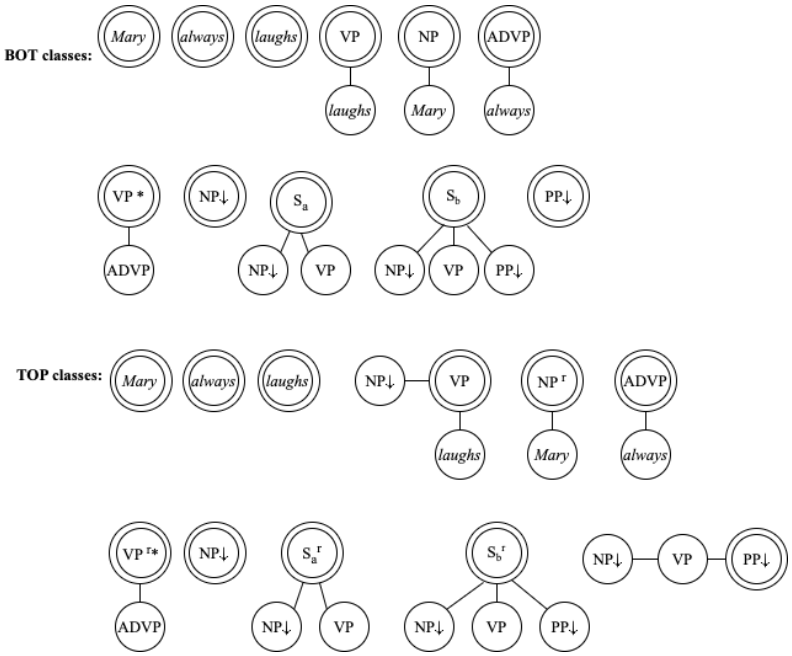


Figure 13: TOP and BOT equivalence classes for the chart example

items are build. The parser now applies all rules used to traverse up the single elementary trees until reaching the root node in the *Mary* tree, where substitution is applied to both substitution nodes of the *laughs* trees in items 15 and 16. Now usually the next step would be to perform Combine-sister with items 11 and 12, seeing as the nodes are sisters in the elementary trees, but since we haven't sister adjoined the auxiliary tree *always* yet, the spans don't match up. Once the root of *always* is reached and the node is left-adjoined, it extends the span of both items 20 and 21, allowing for the Combine-sisters rule being applied and items 24 and 25 being build. Here the parser tries to move up to the root node, but it only works for $laughs_a$ as $laughs_b$ is still needing another right sister to complete all daughters of the root. Item 27 is the goal item, being in a root in $\top$-position spanning the whole input. Some parse items like 11 and 12 never lead to a goal item and some derivation steps are redundant, especially for both *laughs* trees, but in this example they do not blow up the chart exponentially.

There is still a visible difference to table 5 in which the parser is using the new form of parse items including equivalence classes. Figure 13 shows all equivalence classes for

	Item	Rules
1	$[Mary, 1_{\bot}, 0, 1, \langle\rangle, no]$	Scan(Mary)
2	$[always, 1.1_{\bot}, 1, 2, \langle\rangle, no]$	Scan(always)
3	$[laughs_a, 2.1_{\bot}, 2, 3, \langle\rangle, no]$	Scan(laughs)
4	$[laughs_b, 2.1_{\bot}, 2, 3, \langle\rangle, no]$	Scan(laughs)
5	$[Mary, 1_{\top}, 0, 1, \langle\rangle, no]$	NLS(1)
6	$[always, 1.1_{\top}, 1, 2, \langle\rangle, no]$	NLS(2)
7	$[laughs_a, 2.1_{\top}, 2, 3, \langle\rangle, no]$	NLS(3)
8	$[laughs_b, 2.1_{\top}, 2, 3, \langle\rangle, no]$	NLS(4)
9	$[Mary, \epsilon_{\bot}, 0, 1, \langle\rangle, no]$	Move-up(5)
10	$[always, 1_{\bot}, 1, 2, \langle\rangle, no]$	Move-up(6)
11	$[laughs_a, 2_{\bot}, 2, 3, \langle\rangle, no]$	Move-up(7)
12	$[laughs_b, 2_{\bot}, 2, 3, \langle\rangle, no]$	Move-up(8)
13	$[Mary, \epsilon_{\top}, 0, 1, \langle\rangle, no]$	NLS(9)
14	$[always, 1_{\top}, 1, 2, \langle\rangle, no]$	NLS(10)
15	$[laughs_a, 1_{\bot}, 0, 1, \langle\rangle, no]$	Substitute(13)
16	$[laughs_b, 1_{\bot}, 0, 1, \langle\rangle, no]$	Substitute(13)
17	$[always, \epsilon_{\bot}, 1, 2, \langle\rangle, no]$	Move-up(14)
18	$[laughs_a, 1_{\top}, 0, 1, \langle\rangle, no]$	NLS(15)
19	$[laughs_b, 1_{\top}, 0, 1, \langle\rangle, no]$	NLS(16)
20	$[always, \epsilon_{\top}, 1, 2, \langle\rangle, no]$	NLS(17)
21	$[laughs_a, 2.1_{\top}, 1, 3, \langle\rangle, no]$	Left-adjoin(20+7)
22	$[laughs_b, 2.1_{\top}, 1, 3, \langle\rangle, no]$	Left-adjoin(20+8)
23	$[laughs_a, 2_{\bot}, 1, 3, \langle\rangle, no]$	Move-up(21)
24	$[laughs_b, 2_{\bot}, 1, 3, \langle\rangle, no]$	Move-up(22)
25	$[laughs_a, 2_{\top}, 0, 3, \langle\rangle, no]$	Combine-sisters(23+15)
26	$[laughs_b, 2_{\top}, 0, 3, \langle\rangle, no]$	Combine-sisters(24+16)
27	$[laughs_a, \epsilon_{\bot}, 0, 3, \langle\rangle, no]$	Move-up(25)
28	$[laughs_a, \epsilon_{\top}, 0, 3, \langle\rangle, no]$	NLS(27)

Table 4: Parsing "Mary always laughs" without equivalence classes

the example elementary trees. The double circled classes are the main ones, the others are either daughter or left sister classes. All nodes from the original elementary trees are put into equivalence classes and here TOP and BOT classes have the same quantity. Here is where the factorizing is utilized, as all nodes with a matching label leading up to *S* from the *laughs* trees are put into the same equivalence class. Only s_a and s_b are in a different class as they have different daughters. Even in this small example the benefits of factorizing is already visible in the shortened chart. Some items present in the old chart holding nodes from the *laughs*-trees are combined, as the nodes processed there are sharing the same equivalence classes in the new chart. The following list shows which

	Item	Rules
1	$[Mary_{\perp}, 0, 1, \langle\rangle, no]$	Scan(Mary)
2	$[always_{\perp}, 1, 2, \langle\rangle, no]$	Scan(always)
3	$[laughs_{\perp}, 2, 3, \langle\rangle, no]$	Scan(laughs)
4	$[Mary_{\top}, 0, 1, \langle\rangle, no]$	NLS(1)
5	$[always_{\top}, 1, 2, \langle\rangle, no]$	NLS(2)
6	$[laughs_{\top}, 2, 3, \langle\rangle, no]$	NLS(3)
7	$\left[NP^{r}_{\perp}, 0, 1, \langle\rangle, no\right]$	Move-up(4)
8	$[ADVP_{\perp}, 1, 2, \langle\rangle, no]$	Move-up(5)
9	$[VP_{\perp}, 2, 3, \langle\rangle, no]$	Move-up(6)
10	$\left[NP^{r}_{\top}, 0, 1, \langle\rangle, no\right]$	NLS(7)
11	$[ADVP_{\top}, 1, 2, \langle\rangle, no]$	NLS(8)
12	$[NP_{\perp}\downarrow, 0, 1, \langle\rangle, no]$	Substitute(10)
13	$[VP_{bot}*, 1, 2, \langle\rangle, no]$	Move-up(11)
14	$[NP_{\top}\downarrow, 0, 1, \langle\rangle, no]$	NLS(12)
15	$[VP_{top}*, 1, 2, \langle\rangle, no]$	NLS(13)
16	$[laughs_{\top}, 1, 3, \langle\rangle, no]$	Left-adjoin(15+6)
17	$[VP\perp, 1, 3, \langle\rangle, no]$	Move-up(16)
18	$[VP\top, 0, 3, \langle\rangle, no]$	Combine-sisters(17+14)
19	$[S_a, \epsilon_{\perp}, 0, 3, \langle\rangle, no]$	Move-up(18)
20	$[S_a, \epsilon_{\top}, 0, 3, \langle\rangle, no]$	NLS(19)

Table 5: Parsing "Mary always laughs" with equivalence classes

items from the old chart correspond to which items in the new chart:

$$
\begin{aligned}
1 &\longrightarrow 1 \\
2 &\longrightarrow 2 \\
3 + 4 &\longrightarrow 3 \\
5 &\longrightarrow 4 \\
6 &\longrightarrow 5 \\
7 + 8 &\longrightarrow 6 \\
9 &\longrightarrow 7 \\
10 &\longrightarrow 8 \\
11 + 12 &\longrightarrow 9 \\
13 &\longrightarrow 10 \\
14 &\longrightarrow 11
\end{aligned}
\qquad
\begin{aligned}
15 + 16 &\longrightarrow 12 \\
17 &\longrightarrow 13 \\
18 + 19 &\longrightarrow 14 \\
20 &\longrightarrow 15 \\
21 + 22 &\longrightarrow 16 \\
23 + 24 &\longrightarrow 17 \\
25 + 26 &\longrightarrow 18 \\
27 &\longrightarrow 19 \\
27 &\longrightarrow 20
\end{aligned}
$$

5 Evaluation

To evaluate whether the introduction of equivalent classes into the parser has enhanced the process, a small example grammar will be used. First, the grammar is detailed and all

elementary trees are shown. Next, the equivalence classes build for the single nodes are explained and in the end two charts are compared, one using trees and one using classes.

5.1 Toy Grammar

The toy grammar used for this example consists of eleven elementary trees, covering the following sentences:

- Obelix snored
- who snored
- Obelix always snored
- who always snored
- Obelix ate boar
- who ate boar
- Obelix always ate boar
- who always ate boar
- Obelix wanted to eat boar
- what did Obelix want to eat

To suitably show an improvement in chart size, the longest sentence is chosen for parsing:

$$w = {}_0\,what\,{}_1\,did\,{}_2\,Obelix\,{}_3\,want\,{}_4\,to\,{}_5\,eat\,{}_6$$

Once the input is passed to the parser, the grammar is searched for trees that can anchor all words from the sentence. The elementary trees which are passed on for parsing are shown in Figure 14. The sub- and superscript correspond to the nodes BOT and TOP equivalent class respectively, as numbered in table 6 in the next section. Before the parse the structure of the possible result tree is not known, so there are four trees that have *"eat"* as a lexical item.

5.2 Toy grammar equivalence classes

Table 6 shows the equivalent classes for the toy grammar trees in the format known from section 4.1.2:

BOT class: $\langle Category, Type, \langle\langle tree, (p \cdot m)\rangle, ...\rangle, \langle Daughter_1, D_2, ...\rangle\rangle$

TOP class: $\langle \cdots, \langle Left\ Sister_1, LS_2\rangle, ...\langle\langle Possible\ Mother_1, rightmost?\rangle, ...\rangle, \langle\langle tree, (p \cdot m)\rangle, ...\rangle, root?\rangle$

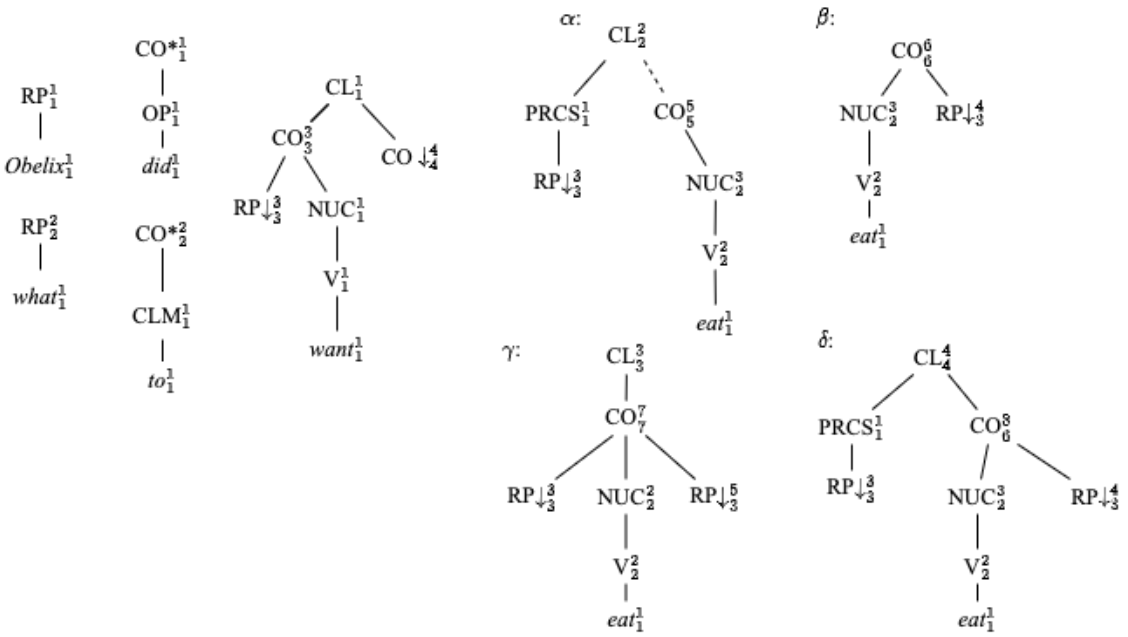


Figure 14: Trees for parsing Toy Grammar example *"What did Obelix want to eat?"*

As the grammar is quite small, most classes are unique and the only structure sharing is between the five trees holding a predicate, mostly between the *eat*-trees. The substitution nodes $_{RP}\downarrow$ of all five trees can be sorted into the same BOT equivalence class, as they all do not have daughters and share label and type. However there are three separate TOP classes, representing the variety of left sister classes from zero to two. Almost every node with label *CO* is already put in a different BOT class considering the diversion in daughters, except the one from trees eat_β and eat_δ sharing the exact same daughter classes. The benefit of factorization comes into play when looking at the nucleus layer of the *eat*-trees, which shows a perfect example of subtree sharing in the corresponding equivalent classes. Except for the *NUC* node in tree eat_γ, which has an extra left sister, all nodes *eat*, *V* and *NUC* share a BOT and TOP class.

Table 6: Equivalence classes for toy grammar

BOT classes	**TOP classes**
Obelix^1 $\langle$"$Obelix$", LEX, $\langle\langle Obelix, (1)\rangle\rangle$, $\langle\rangle\rangle$	Obelix^1 $\langle\cdots, \langle\rangle, \langle\langle RP^1_{BOT}, yes\rangle\rangle, \langle\langle Obelix, (1)\rangle\rangle, no\rangle$
RP^1 $\langle$"RP", $ANCH$, $\langle\langle Obelix, (\epsilon)\rangle\rangle$, $\langle Obelix\rangle\rangle$	RP^1 $\langle\cdots, \langle\rangle, \langle\rangle, \langle\langle Obelix, (\epsilon)\rangle\rangle\rangle, yes\rangle$
what^1 $\langle$"$what$", LEX, $\langle\langle what, (1)\rangle\rangle$, $\langle\rangle\rangle$	what^1 $\langle\cdots, \langle\rangle, \langle\langle RP^2_{BOT}, yes\rangle\rangle, \langle\langle what, (1)\rangle\rangle, no\rangle$
RP^2 $\langle$"RP", $ANCH$, $\langle\langle what, (\epsilon)\rangle\rangle$, $\langle what^1_{BOT}\rangle\rangle$	RP^2 $\langle\cdots, \langle\rangle, \langle\rangle, \langle\langle what, (\epsilon)\rangle\rangle\rangle, yes\rangle$
did^1 $\langle$"did", LEX, $\langle\langle did, (1.1)\rangle\rangle$, $\langle\rangle\rangle$	did^1 $\langle\cdots, \langle\rangle, \langle\langle OP^1_{BOT}, yes\rangle\rangle, \langle\langle did, (1.1)\rangle\rangle, no\rangle$
Continued on next page	

Table 6 – continued from previous page

BOT classes	TOP classes
OP^1 $\langle$"OP", $ANCH$, $\langle\langle did, (1)\rangle\rangle$, $\langle\rangle did^1_{BOT}\rangle$	OP^1 $\langle\cdots, \langle\rangle, \langle\langle CO^1_{BOT}, yes\rangle\rangle, \langle\langle did, (1)\rangle\rangle, no\rangle$
CO^1 $\langle$"CO", $STAR$, $\langle\langle did, (\epsilon)\rangle\rangle$, $\langle OP^1_{BOT}\rangle\rangle$	CO^1 $\langle\cdots, \langle\rangle, \langle\rangle, \langle\langle did, (\epsilon)\rangle\rangle\rangle, yes\rangle$
to^1 $\langle$"to", LEX, $\langle\langle to, (1.1)\rangle\rangle, \langle\rangle\rangle$	to^1 $\langle\cdots, \langle\rangle, \langle\langle CLM^1_{BOT}, yes\rangle\rangle, \langle\langle to, (1.1)\rangle\rangle, no\rangle$
CLM^1 $\langle$"CLM", $ANCH$, $\langle\langle to, (1)\rangle\rangle, \langle\rangle to^1_{BOT}\rangle$	CLM^1 $\langle\cdots, \langle\rangle, \langle\langle CO^2_{BOT}, yes\rangle\rangle, \langle\langle to, (1)\rangle\rangle, no\rangle$
CO^2 $\langle$"CO", $STAR$, $\langle\langle to, (\epsilon)\rangle\rangle$, $\langle CLM^1_{BOT}\rangle\rangle$	CO^2 $\langle\cdots, \langle\rangle, \langle\rangle, \langle\langle to, (\epsilon)\rangle\rangle\rangle, yes\rangle$
want^1 $\langle$"$want$", LEX, $\langle\langle want, (1.2.1.1)\rangle\rangle, \langle\rangle\rangle$	want^1 $\langle\cdots, \langle\rangle, \langle\langle V^1_{BOT}, yes\rangle\rangle, \langle\langle want, (1.2.1.1)\rangle\rangle, no\rangle$
V^1 $\langle$"V", $ANCH$, $\langle\langle want, (1.2.1)\rangle\rangle, \langle want^1_{BOT}\rangle\rangle$	V^1 $\langle\cdots, \langle\rangle, \langle\langle NUC^1_{BOT}, yes\rangle\rangle, \langle\langle want, (1.2.1)\rangle\rangle, no\rangle$
NUC^1 $\langle$"NUC", STD, $\langle\langle want, (1.2)\rangle\rangle, \langle V^1_{BOT}\rangle\rangle$	NUC^1 $\langle\cdots, \langle RP^3_{TOP}\rangle, \langle\langle CO^3_{BOT}, yes\rangle\rangle, \langle\langle want, (1.2)\rangle\rangle, no\rangle$
RP^3 $\langle$"RP", $SUBST$, $\langle\langle want, (1.1)\rangle$, $\langle eat_\alpha, (1.1)\rangle, \langle eat_\beta, (2)\rangle, \langle eat_\gamma, (1.1)\rangle, \langle eat_\gamma, (1.3)\rangle$, $\langle eat_\delta, (1.1)\rangle, \langle eat_\delta, (2.2)\rangle\rangle, \langle\rangle\rangle$	RP^3 $\langle\cdots, \langle\rangle, \langle\langle CO^3_{BOT}, no\rangle, \langle PRCS^1_{BOT}, yes\rangle, \langle CO^7_{BOT}, no\rangle\rangle$, $\langle\langle want, (1.1)\rangle, \langle eat_\alpha, (1.1)\rangle, \langle eat_\gamma, (1.1)\rangle$, $\langle eat_\delta, (1.1)\rangle\rangle, no\rangle$ RP^4 $\langle\cdots, \langle NUC^3_{TOP}\rangle, \langle\langle CO^6_{BOT}, yes\rangle\rangle, \langle\langle eat_\beta, (2)\rangle, \langle eat_\delta,$ $(2.2)\rangle\rangle, no\rangle$ RP^5 $\langle\cdots, \langle RP^3_{TOP}, NUC^2_{TOP}\rangle, \langle\langle CO^7_{BOT}, yes\rangle\rangle, \langle\langle eat_\gamma,$ $(1.3)\rangle\rangle, no\rangle$
CO^3 $\langle$"CO", STD, $\langle\langle want, (1)\rangle\rangle, \langle RP^3_{BOT}, NUC^1_{BOT}\rangle\rangle$	CO^3 $\langle\cdots, \langle\rangle, \langle\langle CL^1_{BOT}, no\rangle\rangle, \langle\langle want, (1)\rangle\rangle, no\rangle$
CO^4 $\langle$"CO", $SUBST$, $\langle\langle want, (2)\rangle\rangle, \langle\rangle\rangle$	CO^4 $\langle\cdots, \langle CO^3_{TOP}\rangle, \langle\langle CL^1_{BOT}, yes\rangle\rangle, \langle\langle want, (2)\rangle\rangle, no\rangle$
CL^1 $\langle$"CL", STD, $\langle\langle want, (\epsilon)\rangle\rangle, \langle CO^3_{BOT}, CO^4_{BOT}\rangle\rangle$	CL^1 $\langle\cdots, \langle\rangle, \langle\rangle, \langle\langle want, (\epsilon)\rangle\rangle\rangle, yes\rangle$
eat^1 $\langle$"eat", LEX, $\langle\langle eat_\alpha, (2.1.1.1)\rangle, \langle eat_\beta, (1.1.1)\rangle$, $\langle eat_\gamma, (1.2.1.1)\rangle, \langle eat_\delta, (2.1.1.1)\rangle\rangle, \langle\rangle\rangle$	eat^1 $\langle\cdots, \langle\rangle, \langle\langle V^2_{BOT}, yes\rangle\rangle, \langle\langle eat_\alpha, (2.1.1.1)\rangle, \langle eat_\beta, (1.1.1)\rangle$, $\langle eat_\gamma, (1.2.1.1)\rangle, \langle eat_\delta, (2.1.1.1)\rangle\rangle, no\rangle$
V^2 $\langle$"V", $ANCH$, $\langle\langle eat_\alpha, (2.1.1)\rangle, \langle eat_\beta, (1.1)\rangle, \langle eat_\gamma, (1.2.1)\rangle$, $\langle eat_\delta, (2.1.1)\rangle\rangle, \langle eat^1_{BOT}\rangle\rangle$	V^2 $\langle\cdots, \langle\rangle, \langle\langle NUC^2_{BOT}, yes\rangle\rangle, \langle\langle eat_\alpha, (2.1.1)\rangle, \langle eat_\beta, (1.1)\rangle$, $\langle eat_\gamma, (1.2.1)\rangle, \langle eat_\delta, (2.1.1)\rangle\rangle, no\rangle$
NUC^2 $\langle$"NUC", STD, $\langle\langle eat_\alpha, (2.1)\rangle, \langle eat_\beta, (1)\rangle, \langle eat_\gamma, (1.2)\rangle$, $\langle eat_\delta, (2.1)\rangle\rangle, \langle V^2_{BOT}\rangle\rangle$	NUC^2 $\langle\cdots, \langle RP^3_{TOP}\rangle, \langle\langle COBOT^7, no\rangle\rangle, \langle\langle eat_\gamma, (1.2)\rangle\rangle, no\rangle$ NUC^3 $\langle\cdots, \langle\rangle, \langle\langle COBOT^5, yes\rangle, \langle COBOT^6, no\rangle\rangle, \langle\langle eat_\alpha,$ $(2.1)\rangle, \langle eat_\beta, (1)\rangle, \langle eat_\delta, (2.1)\rangle\rangle, no\rangle$

Continued on next page

Table 6 – continued from previous page

BOT classes	TOP classes
CO^5 $\langle$"CO", $DDAUGHTER$, $\langle\langle eat_\alpha, (2)\rangle\rangle$, $\langle NUC^2_{BOT}\rangle\rangle$	CO^5 $\langle \cdots, \langle PRCS^1_{TOP}\rangle, \langle\langle CL^2_{BOT}, yes\rangle\rangle, \langle\langle eat_\alpha, (2)\rangle\rangle, no\rangle$
PRCS^1 $\langle$"$PCRS$", STD, $\langle\langle eat_\alpha, (1)\rangle, \langle eat_\delta, (1)\rangle\rangle, \langle RP^3_{BOT}\rangle\rangle$	PRCS^1 $\langle \cdots, \langle\rangle, \langle\langle CLBOT^2, no\rangle, \langle CLBOT^4, no\rangle\rangle, \langle\langle eat_\alpha, (1)\rangle,$ $\langle eat_\delta, (1)\rangle\rangle, no\rangle$
CL^2 $\langle$"CL", STD, $\langle\langle eat_\alpha, (\epsilon)\rangle\rangle, \langle PRCS^1_{BOT}, CO^5_{BOT}\rangle\rangle$	CL^2 $\langle \cdots, \langle\rangle, \langle\rangle, \langle\langle eat_\alpha, (\epsilon)\rangle\rangle, yes\rangle$
CO^6 $\langle$"CO", STD, $\langle\langle eat_\beta, (\epsilon)\rangle, eat_\delta, (2)\rangle\rangle, \langle NUC^2_{BOT},$ $RP^3_{BOT}\rangle\rangle$	CO^6 $\langle \cdots, \langle\rangle, \langle\rangle, \langle\langle eat_\beta, (\epsilon)\rangle\rangle, yes\rangle$ CO^8 $\langle \cdots, \langle PRCS^1_{BOT}\rangle, \langle\langle CLBOT^4, yes\rangle\rangle, \langle eat_\delta, (2)\rangle\rangle, no\rangle$
CO^7 $\langle$"CO", STD, $\langle\langle eat_\gamma, (1)\rangle\rangle, \langle RP^3_{BOT}, NUC^2_{BOT}, RP^3_{BOT}\rangle\rangle$	CO^7 $\langle \cdots, \langle\rangle, \langle\langle CLBOT^3, yes\rangle\rangle, \langle\langle eat_\gamma, (1)\rangle\rangle, no\rangle$
CL^3 $\langle$"CL", STD, $\langle\langle eat_\gamma, (\epsilon)\rangle\rangle, \langle CO^7_{BOT}\rangle\rangle$	CL^3 $\langle \cdots, \langle\rangle, \langle\rangle, \langle\langle eat_\gamma, (\epsilon)\rangle\rangle, yes\rangle$
CL^4 $\langle$"CL", STD, $\langle\langle eat_\delta, (\epsilon)\rangle\rangle, \langle PRCS^1_{BOT}, CO^6_{BOT}\rangle\rangle$	CL^4 $\langle \cdots, \langle\rangle, \langle\rangle, \langle\langle eat_\delta, (\epsilon)\rangle\rangle, yes\rangle$

5.3 Chart comparison

The difference in parse time is negligible when working with such a minimal grammar, but when comparing the charts of both unfactorized and factorized parses there is a visible improvement, i.e. a shortened chart missing redundant items. As the structure sharing between elementary trees is incorporated into the equivalence classes and therefore the parse items, all individual derivation steps moving up the nucleus layer of the four *eat*-trees are packed into the new parse items. A comparison between the charts is seen in table 7, even with this small grammar, the chart size is reduced by 15 items. Since the charts are still quite large, they are not printed fully. The goal item is the desired result for a successful parse and both charts end up holding one spanning the entire input $|w| = 6$.

	Old chart	**New chart**
Chart size	94	79
Goal item	$[\alpha, \epsilon_\top, 0, 6, \langle\rangle, no]$	$\boxed{CL^2_{TOP}, 0, 6, \langle\rangle, no}$

Table 7: Comparison of charts

6 Conclusion

The main goal of this thesis was to enhance a chart parser by adding a factorizing step before deduction, reducing the input grammar used. The example comparison in the previous section indicates a successful improvement to the parsing environment, proving the efficiency of the factorizing idea proposed in this paper. The chart storing all parse items representing deduction steps is noticeably smaller after factorizing. The factorizer takes a tree-based grammar and introduces structure sharing via equivalent classes. Subtrees sharing a structure are packed into the same class, minimizing the parse items built and reducing the size of the parse chart. The factorizer extension was added to the TuLiPA parsing environment, based on Role and Reference Grammar formalized and composed using Tree Writing Grammar.

The approach proposed in this paper provides a basis for future research aiming at improving grammar theories. For further development, additional constraints could be introduced into single trees by using feature structures. This would fasten the addition of operators and modifiers via sister adjunction, reducing the redundant parse items even more. Extraction of the derived trees from the goal items might become more complicated when using factorized trees, as the original elementary tree structure is no longer retained. Adding extraction to a factorized grammar is also a point for further consideration.

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 21. Dezember 2022

Julia Block

References

David Arps (2018). "Parsing Role and Reference Grammar". Bachelor thesis. Heinrich-Heine-Universität.

David Arps, Tatiana Bladier, and Laura Kallmeyer (2019). "Chart-based RRG parsing for automatically extracted and hand-crafted RRG grammars". In: *University at Buffalo, Role and Reference Grammar RRG Conference*.

Tatiana Bladier, Jakub Waszczuk, and Laura Kallmeyer (Dec. 2020). "Statistical Parsing of Tree Wrapping Grammars". In: *Proceedings of the 28th International Conference on Computational Linguistics*. International Committee on Computational Linguistics, pp. 6759–6766.

Jay Earley (Feb. 1970). "An Efficient Context-Free Parsing Algorithm". In: *Commun. ACM* 13.2, pp. 94–102.

Aravind K Joshi and Yves Schabes (1997). "Tree-adjoining grammars". In: *Handbook of formal languages*. Springer, pp. 69–123.

Laura Kallmeyer (2010). *Parsing beyond context-free grammars*. Springer Science & Business Media.

Laura Kallmeyer (2017a). "RRG Parsing". Unpublished article.

Laura Kallmeyer (2017b). *Tomita's Parser: Generalized LR Parsing*. Lecture notes, Heinrich-Heine-Universität.

Laura Kallmeyer, Timm Lichte, Wolfgang Maier, Yannick Parmentier, Johannes Dellert, and Kilian Evang (2008). "TuLiPA: Towards a multi-formalism parsing environment for grammar engineering". In: *Proceedings of the Workshop on Grammar Engineering Across Frameworks GEAF*.

Laura Kallmeyer and Rainer Osswald (2017). "Combining predicate-argument structure and operator projection: Clause structure in Role and Reference Grammar". In: *Proceedings of the 13th international workshop on tree adjoining grammars and related formalisms*, pp. 61–70.

Laura Kallmeyer, Rainer Osswald, and Robert D. Van Valin (2013). "Tree wrapping for role and reference grammar". In: *Formal Grammar. 17th and 18th International Conferences, FG 2012 Opole, Poland, August 2012, Revised Selected Papers*. Ed. by Glyn Morrill and Mark-Jan Nederhof. Springer, pp. 175–190.

M Kay (1986). "Algorithm Schemata and Data Structures in Syntactic Processing". In: *Readings in Natural Language Processing*. Morgan Kaufmann Publishers Inc., pp. 35–70.

Fernando CN Pereira and David HD Warren (1983). "Parsing as deduction". In: *21st annual meeting of the association for computational linguistics*, pp. 137–144.

Masaru Tomita (1987). "An efficient augmented-context-free parsing algorithm". In: *Computational linguistics* 13, pp. 31–46.

Robert D. Van Valin (Jan. 1993). "Role and reference grammar". In: *Work Papers of the Summer Institute of Linguistics, University of North Dakota Session* 37.1.

Robert D Van Valin, Robert D Van Valin Jr, Randy J LaPolla, et al. (1997). *Syntax: Structure, meaning, and function*. Cambridge University Press.

Robert D. Van Valin Jr. (2005). *Exploring the Syntax-Semantics Interface*. Cambridge University Press.