# Efficient conversion from Dependency Trees to Abstract Syntax Trees in Natural Language Processing

Master's thesis in Computer science and engineering

ANDREAS KÄLLBERG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

# Efficient conversion from Dependency Trees to Abstract Syntax Trees in Natural Language Processing

Connecting Grammatical Framework with Universal Dependencies

ANDREAS KÄLLBERG

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Efficient conversion from Dependency Trees to Abstract Syntax Trees in Natural Language Processing
Connecting Grammatical Framework with Universal Dependencies
ANDREAS KÄLLBERG



Supervisor: Aarne Ranta, Department of Computer Science and Engineering
Examiner: Krasimir Angelov, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Photo of a Mallard by Andreas Källberg

Typeset in LaTeX
Gothenburg, Sweden 2024

Efficient conversion from Dependency Trees to Abstract Syntax Trees in Natural Language Processing
Connecting Grammatical Framework with Universal Dependencies
ANDREAS KÄLLBERG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

This thesis is a contribution to the field of rule based Natural Language Processing. Within rule based NLP there are different grammar formalisms, each with their own strengths and weaknesses. This work presents several improvements to a tool for converting between two such grammar formalisms. First we improved the performance significantly. Secondly we improved the debugging facilities of the tool. Thirdly we made it possible to convert between trees that had previously been too structurally different from each other.

Keywords: Computer, science, computer science, engineering, project, thesis.

# Acknowledgements

I want to thank Aarne Ranta, my supervisor for guiding me throughout my journey.

I want to thank Inari Listenmaa for her support, patience and kindness.

I want to thank my parents Eva Piscator and Ulf Källberg for both emotional and practical support and guidance.

I want to thank the SMU Centre for Computational Law for giving me money while working on this.

Andreas Källberg, Gothenburg, 2024-08-16

# Contents

# List of Figures

# List of Tables

# Glossary

**GF** Grammatical Framework

**UD** Universal Dependencies

**NLP** Natural Language Processing

**POS** Part of Speech

# 1

# Introduction

Since the advent of computers, people have been trying to make them understand natural languages. Today machine learning methods are very popular, but the more traditional method is so called rule-based Natural Language Processing (NLP) and Natural Language Generation (NLG), in which we treat natural languages more like programming languages and write the grammar rules explicitly into the computer in order to parse the text into structured data and/or generate text from abstract data.[1]–[3] In fact, much of the terminology around programming languages comes from linguistics for this exact reason[1]. This work will focus on connecting the two rule-based NLP formalisms of *Grammatical Framework*[4] and *Universal Dependencies*[5].[2]

While machine learning is currently dominating in NLP, using rule-based approaches still provides several important advantages over those based on machine-learning. For example, rule-based approaches are deterministic and more predictable. They make it possible to fix individual bugs without needing to retrain the whole model, hoping that it fixes the issue. They are also more transparent, rather than being a black box as most machine learning based language models are. These properties are particularly important when correctness is crucial, for example in law and medicine.

While technically even the most naive string replacement method[3] is an instance of "rule-based NLP", in practice we tend to use more sophisticated tools, namely *grammar formalisms*.

A grammar formalism is used to describe the connection between a plain text and structured data. To understand what a formalism is, it is useful to introduce the concept of structured data. Consider a text string like "the cat sat on the mat"—for a general-purpose computer program, it is nothing more than an array of characters. But to a human linguist, it is full of structure: "sat" is the main verb, "cat" is the subject, "on the mat" is an adverbial. A grammar formalism is simply a way to represent this grammatical structure in a machine readable way.

There are several different formalisms for describing sentences as trees, all with their own strengths and weaknesses. The two we will be talking about in this paper are

[1] Programming *languages* have a *syntax* and a *grammar* which is *parsed* and lexed into *lexemes*

[2] The code is found on github: https://github.com/GrammaticalFramework/gf-ud

[3] e.g. adding an "s" to the end of a word to make it plural, producing "foots" instead of the correct "feet"

Universal Dependencies (UD), which is based on so called *dependency trees*, and Grammatical Framework (GF), which uses *constituency-based abstract syntax trees*.

## 1.1 Dependency trees and Universal Dependencies

A dependency tree is a tree structure that shows the grammatical relationship between words in a sentence[6], [7]. Each word becomes a node in the tree with the main verb as the root and the edges represent the relation and the direction of the dependency.

The specific standard for dependency trees that this paper is about is called Universal Dependencies (UD)[5]. UD is based on the idea of making a multilingual standard for dependency trees where the same set of tags can be used regardless of which language the sentence is written in.

In Figure 1.1 we can see the sentence "the cat sleeps" analyzed as a UD-tree. The first step is to determine which part-of-speech each word in the sentence belongs to. In this case, "the" is a determiner, "cat" is a noun and "sleeps" is a verb. Next we need to determine the relation between the words: "sleeps" is the root of the sentence, "cat" is the subject of this verb and "the" quantifies (determines) the noun.

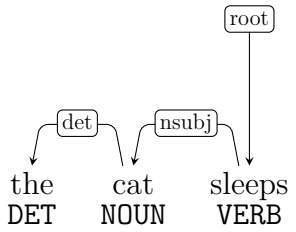



Figure 1.1: The phrase "the cat sleeps" analyzed as a UD tree.

In order to create a tree from a sentence, UD uses supervised machine-learning trained on a large library of manually tagged sentences.[4]

## 1.2 Abstract syntax trees and Grammatical Framework

Another way of describing the grammatical structure of a sentence is through abstract syntax trees, where instead of using words as the nodes in the tree, you use nodes tagged with functions for combining different parts of speech and with the words only being in the leaves.

Grammatical Framework (GF)[8] is a formalism for describing natural language grammars as code, which allows converting between natural language and a language-independent abstract syntax. It is split into a generic resource grammar that covers

[4]This differs from the Large Language Models like GPT, which are based on unsupervised learning trained on untagged text from the internet.

morphology and the language as a whole and application grammars for a more narrow domain which allows a more semantic abstract syntax. When you try to write a grammar that covers a very wide domain, you will often get an over-generating grammar where each sentence can be parsed in many different ways into many different trees, where usually only one is the intended way.

In Figure 1.2, the sentence "The cat sleeps" is analyzed as a GF tree. Instead of having dependencies between words like in UD, the trees is built up of constituency. In this case we have a sentence clause (labeled `PredVP : Cl`) that consists of a noun phrase (labeled `DetCN : NP`) and a verb phrase (labeled `UseV : VP`) and the noun phrase in turn consists of a determiner ("the") and what Grammatical Framework calls a *common noun* (labeled `UseN : CN`) which in turn consists of the noun "cat". The verb "sleeps" constitutes the verb phrase.

In order to build up this tree, we used these functions:

- `UseN : N -> CN` is a function that takes a noun and converts it to a common noun,
- `DetCN : Det -> CN -> NP` is a function that takes a determiner and a common noun and converts it to a noun phrase,
- `UseV : V -> VP` is a function that takes a verb and converts it to a verb phrase
- and finally, `PredVP : NP -> VP -> Cl` is a function that takes a noun phrase and a verb phrase and converts it to a clause

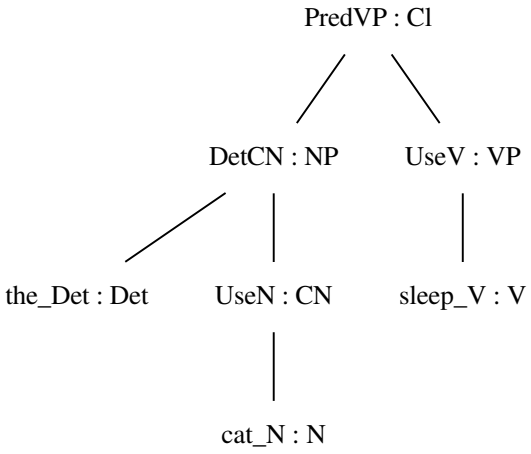



Figure 1.2: The sentence "The cat sleeps" analyzed as a GF tree

## 1.3 Differences between GF and UD

GF is very strict when it comes to following grammar and spelling, which means that it will often refuse to parse sentences if they contain even the smallest error. UD on the other hand uses machine-learning to give some parse tree for all sentences regardless of how many errors they have.

While the machine-learning-based approach of UD allows it to guess the correct tree better for ambiguous sentences and allows it to handle grammatically incorrect sentences better, GF is much more capable when it comes to performing transformations on the sentences, while maintaining correct morphology and grammar. This makes it attractive to parse sentences using UD and then convert the parsed trees to GF trees in order to perform further transformations.

### 1.3.1 Useful synthesis between UD and GF

Prior to this work there existed a proof-of-concept implementation of a tool for converting between the trees for GF and UD, with the help of so-called labels-files containing annotations which describe the mapping between UD labels and GF functions, called gf-ud, which contains both a component for converting from UD to GF, called ud2gf[9][1] and a component for converting from GF to UD, called gf2ud[10][1]. This work will focus on the ud2gf component.

### 1.3.2 Applications for the synthesis of UD and GF

The gf-ud tool has been used for both translation and semantics[11]. Another application has been in concept alignment[12]. It has also been used to analyze legal texts as a part of a Controlled Natural Language for law[13].

## 1.4 What problem does ud2gf solve?

One problem within NLP is converting from text to logic. There are several different paths one could take, each with their own issues, see Figure 1.3 for an overview. The tool gf2ud makes it possible to get the robust parsing available for dependency trees and the ease of conversion to logic of Abstract Syntax Trees.

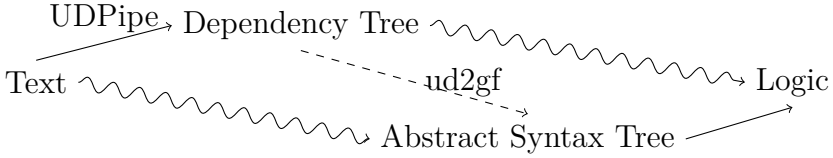



Figure 1.3: An overview of how gf2ud can be helpful in converting from text to logic

Converting from text to dependency trees is a solved problem[14]–[16]. Converting from AST to logic is a solved problem[17]–[19]. Converting from dependency trees to logic is a difficult problem[20], [21]. Converting from natural language text directly to AST is a difficult problem[3], [17], [22]–[24]. By converting from DT to AST ud2gf allows us to use the two solved problems to get a path from text to logic.

[1]These references apply to an older version of gf-ud, from before the one this thesis is based on. A part of the goal of this thesis is to document the later version in addition to documenting the changes made in the process of this thesis.

# 2

# Background and problem

## 2.1 Background

In a project[13] at the Singapore Management University, Center for Computational Law we wanted to convert text into logical form, with the goal of being able to parse and interpret legal text. We initially tried using GF directly, but we ran into issues with it being too strict, so many of the sentences were not parsed at all, either because they contained words not in the dictionary or because it didn't follow the grammatical rules exactly. In order to work around this we used a UD parser, which was based on machine learning, which allowed it to be more robust for unexpected data. However, now the problem of converting into logic became more difficult, so we used the gf2ud tool to convert the UD trees into GF trees. This worked fairly well, but we ran into several problems, described in the next section, which this project aims to solve.

## 2.2 Problem

The original implementation of ud2gf had some limitations. There are primarily three problems that this work aims to fix:

1. Converting UD trees to GF trees quickly became extremely slow for sentences with more than a couple of words and/or when using large GF-grammars, e.g. the GF-grammars based on Wordnet[25].
2. If the structure would differ too much between the representation of a sentence in UD format and as a GF tree, it is not possible to describe the required transformation in the old "labels file" language. See chapter 7 for more details.
3. The third problem is that it can sometimes be difficult to figure out why a rule in a labels-file is not firing, so it would be useful to have a debugging tool to help diagnosing such issues.

## 2.3 Goals and Challenges

1. Analyze algorithms and improve performance. The main challenge here is to find an algorithm for finding matching trees without exponential complexity

on the number of children of a node in the UD tree.

2. Improve flexibility of annotation language, allowing changing the structure of the trees while translating from UD to GF. One challenge here is in figuring out either how to change the algorithms to support these more advanced transformations or to find a way to allow them without needing to change the algorithms.
3. Write a debugging tool, which analyzes exactly what it is that prevents a rule in a labels-file from firing or what prevents that tree from being selected. One challenge here is how to explain to the user the issue for all the possible things that can go wrong. There is also an engineering challenge in making an algorithm that figures out what went wrong and why.
4. Document the version of the tool on which this work is based on, which had changed since what was written in [9]

## 2.4 Limitations

Only the direction of converting UD trees to GF trees is studied here, because that is what was relevant to the application at hand. Furthermore, the two directions are almost completely independent in the implementation.

# 3

# Methods

These methods were used for the different parts of the project

## 3.1 Performance

1. Finding the main source of slowness, which was done with profiling and printf-debugging.
2. Analysing the current algorithms, which are based on brute force, trying all combinations, with some simple filtering.
3. Finding a better algorithm, which avoids exploring paths that could never be the correct answer and which avoids duplicate work.
4. Analysing the algorithmic complexity of both the new and the old algorithm and testing the practical performance to confirm the results.

## 3.2 Flexibility

In order to allow changing the shape of trees when translating from UD to GF, the macro language needs to be expanded. A first prototype of this with minimal code changes has been done by making macro expansion recursive and then representing the code for the transformation in Church-encoding, inspired by lambda-calculus.

This approach can be evaluated by seeing how well it covers different tree shape changes for different trees one would encounter.

It could also be worthwhile to make a more user-friendly version of the advanced macros that can be understood without knowing about Church-encoding.

## 3.3 Debugging tool

Going through each component of the algorithms in order to find where applying a rule can go wrong and add detection for them. Additionally trying out the debugging tool on a real grammar, e.g. in the context of [13], in order to find edge-cases which were not handled by the debugging tool.

# 4

# The old algorithm

In this chapter, we first explain the annotations that are used to describe the mapping between GF trees and UD trees, then we present a high level overview of the algorithm followed by going through a concrete example of how the algorithm works. Finally, we discuss some limitations of the old algorithms, which we cover solutions of in the next chapter.

## 4.1 Overview

GF trees consist of functions and categories, UD trees consist of Part of Speech (POS) annotations and dependency labels and some optional extra annotation.

The ud2gf tool uses annotations in a so called `.labels`-file to describe the relation between the UD labels and the GF functions. These annotations give constraints, which are used to decide which GF functions can be applied where. The algorithm starts by parsing each individual word using GF for each of the possible categories according to the annotations and the Part of Speech. After that the algorithm tries to recursively apply as many functions as possible, following the constraints from the annotations and the UD tree structure. When multiple trees of the same category are generated at the same location in the UD tree, the most complete ones are selected. Here, completeness is a partial order measured in terms of how many words are included in the selected tree. A tree that contains all words that another tree has is considered more complete. Finally backup functions are inserted to include the words that were not included in (one of) the most complete trees.

## 4.2 How annotations work

In order for the program to know which UD POS corresponds to which GF categories (can be multiple) and how the GF functions relate to the UD dependency labels, we need to supply annotations which describe these correspondences. Most of these annotations are bidirectional and are used both by ud2gf and gf2ud. A complete description of these annotations can be found in Appendix A.

First we have the `#cat` annotation, which describes the mapping from GF categories and UD Part of Speech labels:

```
1    #cat GFCategory UD_POS
```

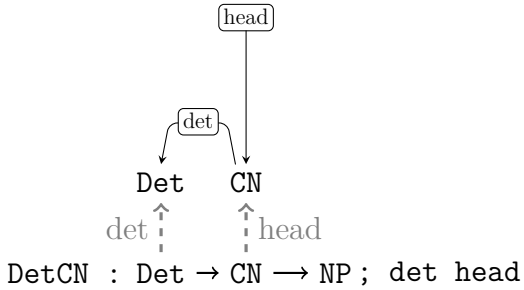



Figure 4.1: The UD shape that the function `DetCN` matches against.

```
2     #cat N NOUN
```

Secondly we have the `#fun` annotation, which describes GF functions and which UD dependency label each argument of the function should have (see Figure 4.1 for graphical representation of such an annotation). The type signature of the GF function is also required. Each function needs to have exactly one argument with the label `head`, corresponding to the head of the current subtree of the UD tree, and any number of arguments corresponding to the direct children of that head in the UD tree, which are labeled with their corresponding UD dependency labels. It is also possible to add other UD annotations in brackets to a label.

```
1     #fun GFFunctionName : FirstArgumentCat -> SecondArgumentCat ->
          ReturnCat ; first_ud_label second_ud_label
2     #fun UseN : N -> CN ; head
3     #fun DetCN : Det -> CN -> NP ; det head
```

The function `UseN` only has a single argument (of type `N`), so that is required to be the head, while the function `DetCN` has two arguments, of types `Det` and `CN`, the first of which has the dependency label `det`, while the second is the head. An illustration of this can be seen in Figure 4.1.

Both of these annotations are bidirectional and language-independent, since they only refer to the abstract syntax in GF.

## 4.3 Overview of the old algorithm

The algorithm is a variation of a bottom-up parser, guided by both the structure and the labels of the UD tree.

Similarly to an ordinary bottom up parser, we annotate each word with one or several GF trees and then build up larger trees by combining these from the bottom up. However, rather than starting with a linear string, we instead start with a UD tree, where each node in the UD tree corresponds to a word. And instead of combining neighboring GF trees in the linear string, we combine a GF tree for a word with GF trees from a number of dependents of that word.

More precisely, as mentioned in section 4.2, each syntactic GF function is annotated with exactly one `head` argument and zero or more other arguments with their

respective dependency labels. In an iteration, we pick one GF tree from the head word and one GF tree from each of as many unused dependent words as the function has non-head arguments. Then if both the category of the used GF trees and the dependency labels of the used dependents match the annotations (as illustrated in Figure 4.1), we add the new GF tree to the head word. After this, we keep trying to apply functions on the generated GF trees for the current head word until no more can be applied and then continue to the next word.

### 4.3.1 Example: The black cat

For a very simple initial example, we start with the noun-phrase "the black cat", which has the following UD representation:

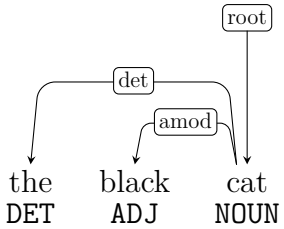



Figure 4.2: "the black cat" as a UD tree

And we use the following GF abstract syntax:

```
cat
  Det CN NP AP N A;
fun
  -- Syntactic functions
  DetCN  : Det -> CN -> NP;
  ModCN  : AP  -> CN -> CN;
  UseN   : N         -> CN;
  PositA : A         -> AP;

  -- Lexical functions, i.e. functions that have no arguments:
  the_Det : Det;
  black_A : A;
  cat_N : N;
```

and the following corresponding dependency configuration (labels-file):

```
#fun DetCN  : Det -> CN -> NP ; det  head
#fun ModCN  : AP  -> CN -> CN ; amod head
#fun UseN   : N         -> CN ; head
#fun PositA : A         -> AP ; head

#cat A                        ; ADJ
#cat Det                      ; DET
#cat N                        ; NOUN
```

Notice how only the lexical categories[1] have a corresponding UD POS. This is a distinction between GF's phrase structure grammar, which uses phrasal categories covering entire phrases, and UD's dependency grammar, where the individual words are annotated with a POS.

The first step is to parse each word in the UD tree into their corresponding lexical GF functions. We use the configuration in the labels file to convert UD POS into their corresponding GF categories.

```
DET -> Det
ADJ -> A
NOUN -> N
```

Next we use the GF parser[2] on each individual lemma (word in the dictionary form), with the category or categories we just got[3]. This gives the tree in Figure 4.3. Notice how the words have been replaced by GF lexical functions and the POS annotations have been replaced by GF Categories. The UD dependency labels still remain.

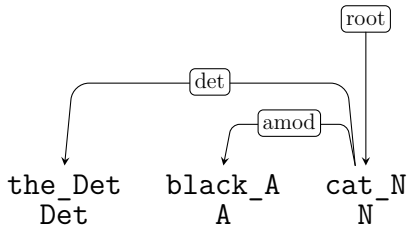



Figure 4.3: "The black cat" as a UD tree, with GF lexicon entries inserted

After converting UD POS annotations to GF categories and parsing words to GF lexical functions, we start traversing the UD tree. The tree is processed from the bottom up, starting with the leaves. Since `cat_N` has unprocessed children we keep going down and reach `the_Det`. We look through the list of available functions and see that the only function that takes a `Det` as an argument takes several arguments and no functions take a `Det` in head position, so it can not be applied to a leaf node like this. There is nothing to be done here, so we continue.

black_A : A

(a) "black" as an adjective : A

Figure 4.4: The available GF trees on the word "black" before the first iteration

[1]Lexical categories are categories that are produced by lexical functions. A lexical function is a function with zero arguments, which usually corresponds to a singular word.

[2]It would also be possible to use the morphoanalyzer, which would be more efficient. However, some GF lexical functions have multiple forms in their inflection tables and we need to know which functions to apply to the lexical function to produce that particular form. An extreme example of this is numerals in the GF standard library, as can be seen in subsection 4.3.2.

[3]The GF parser needs to know which category the result should be in order to be able to parse correctly.

Next we get to "black", with the available tree `black_A` (Figure 4.4) and here we can apply `PositA : A -> AP ; head`, which converts an adjective into an Adjectival Phrase (AP), so now the available trees on "black" are `[black_A : A, PositA black_A : AP]` (Figure 4.4), no more functions can be applied here, so we continue. Figure 4.5 illustrates how the rule for `PositA` matches against this node.

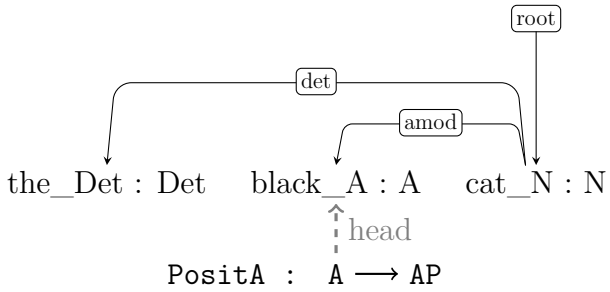



Figure 4.5: An illustration of how the rule for `PositA` matches against the tree `black_A` for the word *black* in the phrase "the black cat".


(a) "black" as an adjective : A (b) "black" as an adjectival phrase: AP

Figure 4.6: The available GF trees on the word "black" after the first iteration

After having processed the children of "cat", the UD tree annotated with GF trees looks like in Figure 4.7.

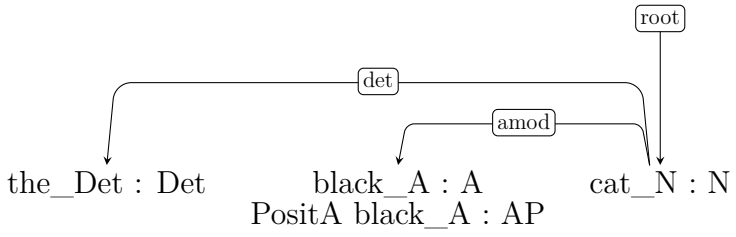



Figure 4.7: An overview of the nested tree, after having processed both the dependent words "the" and "black".

Now that we are done with all the children of "cat", we can start processing it. We have `cat_N : N`.


(a) “cat” : N

Figure 4.8: The available GF trees on the word “cat” before the first iteration

Looking through all the functions, only `UseN : N -> CN ; head` takes an N as an argument, so that's the one that's applied, giving us `UseN cat_N : CN`. Now we have the trees in Figure 4.9


(a) “cat” as a noun : N (b) “cat” as a common noun: CN

Figure 4.9: The available GF trees on the word “cat” after the first iteration

In the next iteration we have a `CN` available for “cat”, which means that, as is illustrated in Figure 4.10, we can apply either of the following functions:

```
1    DetCN : Det -> CN -> NP ; det head
2    ModCN : AP -> CN -> CN ; amod head
```

Let us verify that these are possible to apply: For `DetCN` we need a tree of type *Det* with the *det* UD label for the relation and indeed, the word “the” has the *det* label on the relation to “cat” and we have the tree `the_Det : Det` available on “the”. Secondly we need a *CN* at the head and since our current head is “cat” and we have `UseN cat_N : CN`, that fits perfectly giving us `DetCN the_Det (UseN cat_N): NP`. With similar reasoning for `ModCN`, we have “black” with UD-label *amod* and one of the available trees for “black” is `PositA black_A : AP` which has the correct category, which allows us to construct `ModCN (PositA black_A)(UseN cat_N): CN`. After this we have the trees in Figure 4.11

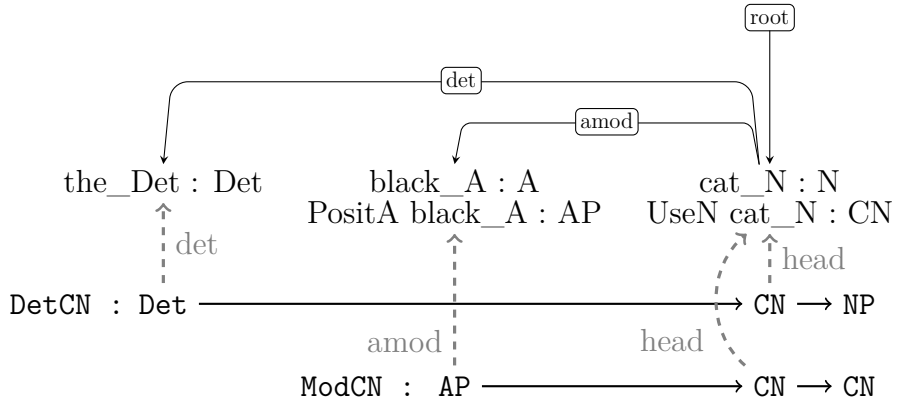
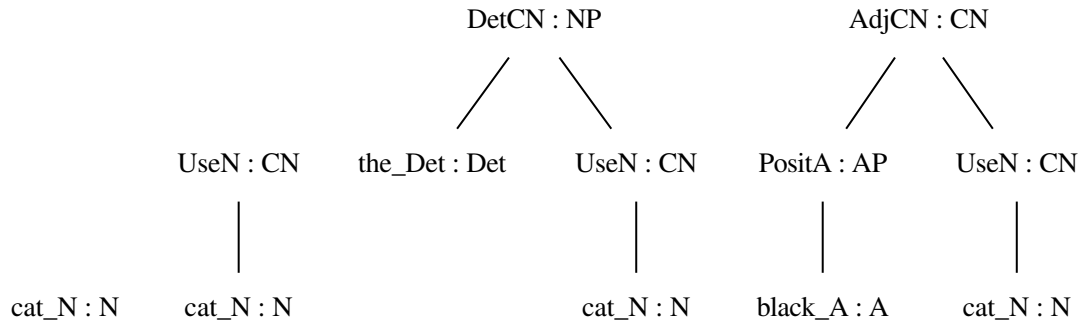



Figure 4.10: An illustration of how `DetCN` matches against the words *the* and *cat* and how `ModCN` matches against the words *black* and *cat* in the phrase "the black cat".




(a) cat : N (b) cat : CN (c) the cat : NP (d) black cat : CN

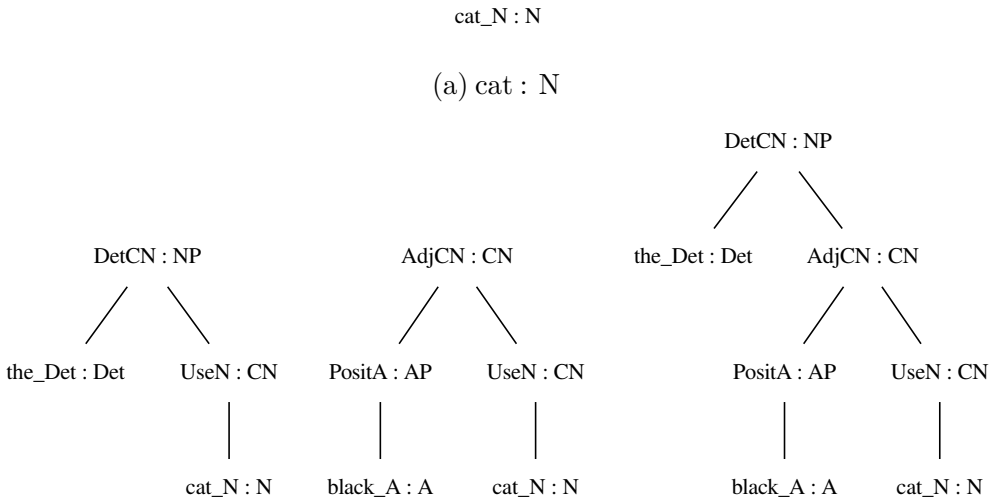
Figure 4.11: The four available trees on the word "cat" after the second iteration.

After this iteration we have two trees on "cat" which both have the same category $CN$, namely

```
1    UseN cat_N : CN
2    ModCN (PositA black_A) (UseN cat_N) : CN
```

and one of them is a strict superset of the other when it comes to words covered, the first only covers "cat" while the second covers both "black" and "cat". This means we can prune the redundant tree `UseN cat_N`.

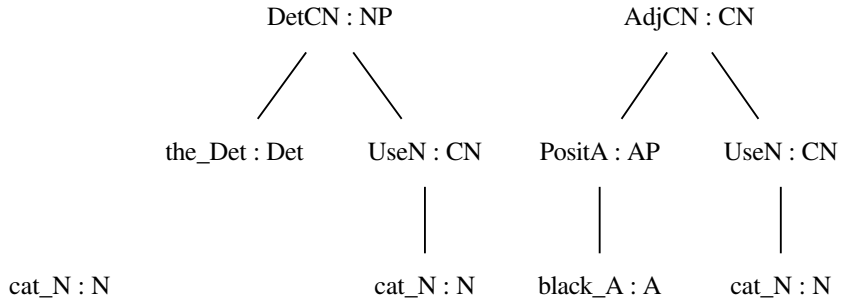
After this pruning, the available trees on the "cat" node can be seen in Figure 4.12.




(a) cat : N

(b) the cat : NP (c) black cat : CN (d) the black cat : NP

Figure 4.13: The four available trees on the word "cat" after the third iteration *before* pruning. Trees (a) and (c) are both subtrees of the final tree (d), while tree (b) is not a subtree of (d), because it connects "the" directly to "cat" ignoring the adjective "black".




(a) cat (b) the cat (c) black cat

Figure 4.12: The three available trees on the word "cat" after the second iteration after pruning. Tree (b) will not be a subtree of the final tree.

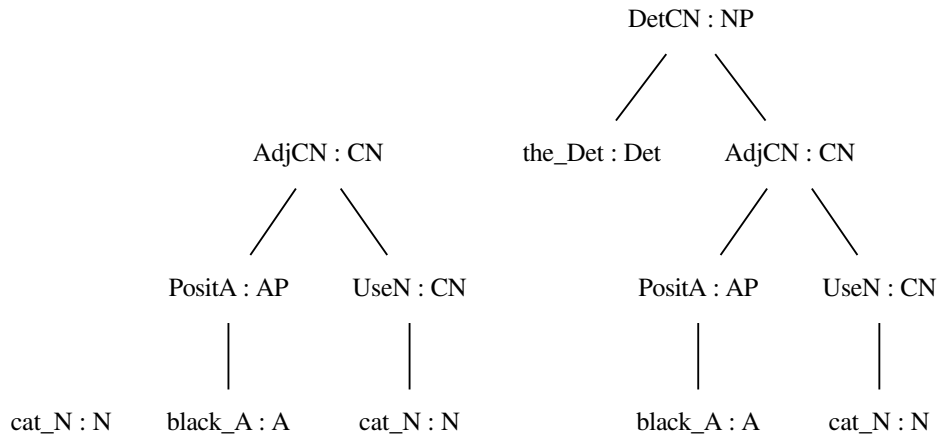
After this step, one function is still possible to apply, namely $DetCN$, which can be applied to our new $CN$ together with `the_Det` like before, giving us the new tree

```
1    DetCN the_Det (ModCN (PositA black_A) (UseN cat_N)) : NP
```

and like before, we get multiple trees of the same category, this time of type $NP$, once again allowing us to prune the less covering tree.

```
1    DetCN the_Det (UseN cat_N) : NP
```

After this no more functions can be applied at the "cat" node, giving us a final set of cat trees as can be seen in Figure 4.14.




(a) cat : N (b) black cat : CN (c) the black cat : NP

Figure 4.14: The three available trees on the word "cat" after the third iteration *after* pruning. Trees (a) and (b) are both subtrees of the final tree (c).

Now finally we have a complete tree which contains all of the words of the phrase, so we will choose the tree in Figure 4.14c as the final tree.

In the end, the data structure used in the calculation will look like in Figure 4.15 or Figure 4.16, with the UD structure outside which we traverse in order to find which parts can be connected and in each node of this tree a list of the GF trees that are possible to construct using the words in the local UD subtree, while conforming to the UD labels. In this case, the word "the" has one GF tree, the word "black" has two possible trees and the word "cat", which depends on the words "the" and "black" has four possible trees.

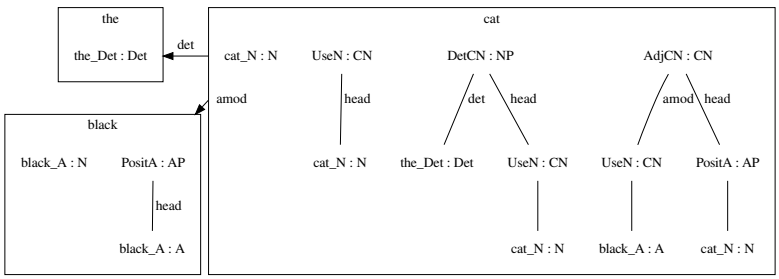



Figure 4.15: An overview of the nested tree, with the UD structure outside and a list of GF-trees at each node

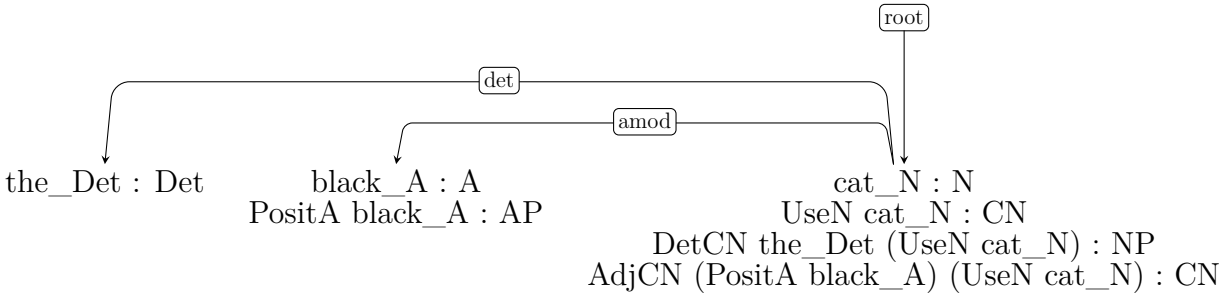



Figure 4.16: An overview of the nested tree, with the UD structure outside and a list of GF-trees at each node.

### 4.3.2 Multiple possible GF trees of the same category

In this simple example, we arrived at a state with intermediate trees that had the same type, and one was a strict subset of the other ("cat" and "black cat"). In a case like that, the old algorithm prunes away the smaller of the trees and continues on to the next iteration.

However, in a different grammar, we might very well end up with several trees that are of the same type, but not subsets of each other. This is the case for numerals in GF standard library, where the trees for numbers like "eight", "eighteen" and "eighty" formed by applying different functions to the underlying digit called `n8`. Figure 4.17 shows an example.

Due to the implementation of the GF grammar, all of the strings "eight", "eighteen" and "eighty" are included in the inflection table of `n8`. This means that if the input contains any of these words, it matches the abstract syntax function `n8`, and then the next steps of the algorithm will start applying all possible functions to it. And since all of these functions are parallel alternatives, the results cannot be pruned like "cat" and "black cat". At the final step, when ud2gf is forced to present the user with only one alternative, the algorithm just picks the first tree from the intermediate list. This means that sometimes the results containing numerals are simply wrong: "I have eight cats" might become "I have eighteen cats".

To compensate for this issue, one can disable the potentially ambiguous functions using `#disable` and use `#auxfun` macros to ensure that all the relevant information is taken into consideration when deciding which trees are valid translations.

## 4.4 Differences between versions

As explained in section 4.3, the algorithm works as follows. For each word, we try to apply all possible functions that match in both type and dependency label against the current head and a selection of children. As a result, we may get zero or more new trees for the given word. Then we want to check if these new trees can, in turn, become the head argument for new function applications. In the old algorithm, we check with *all* the available trees for that word against each available function.

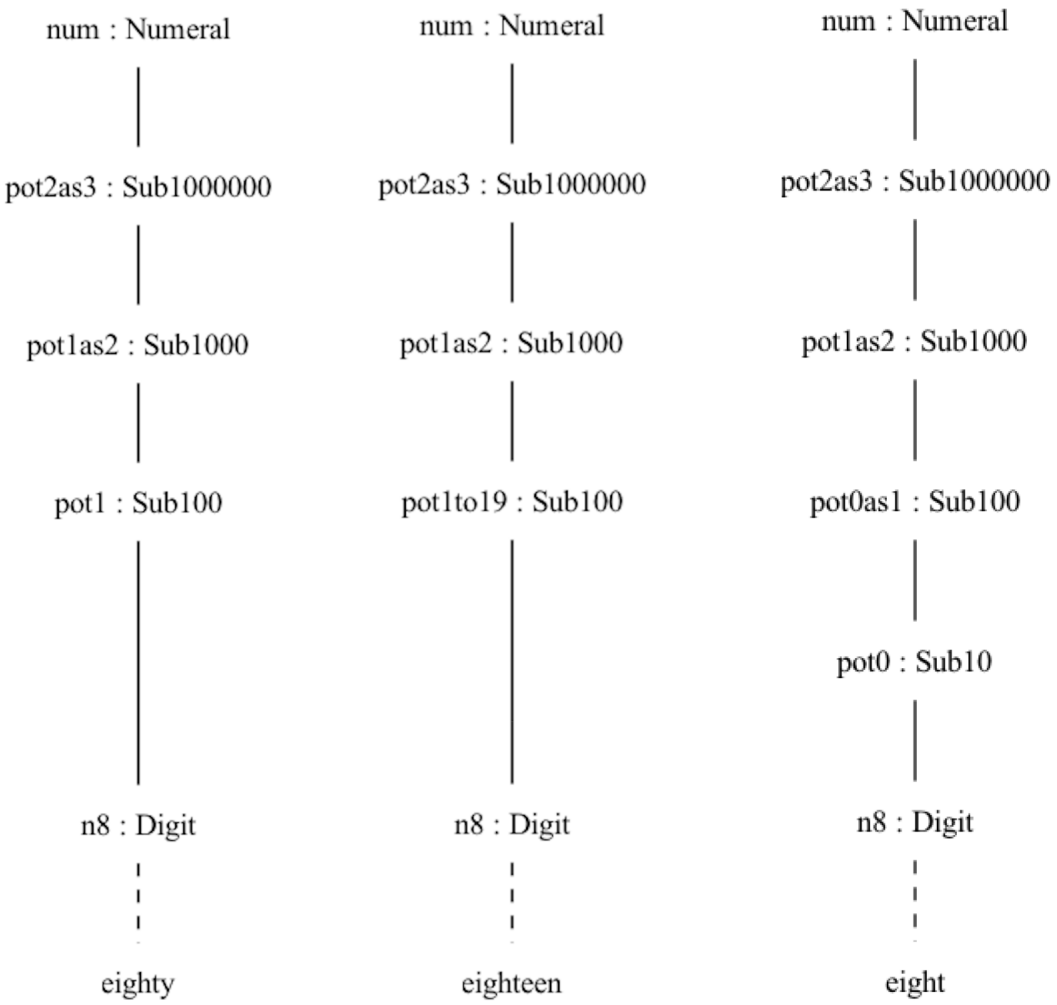



Figure 4.17: Three GF trees formed of the digit `n8`

This means that all the trees that were possible to construct in the previous iteration are still possible to construct, so we will construct them again and will need to remove the duplicates afterwards.

# 5

# The new algorithm

In the new algorithm (version 3), we make two main improvements:

## 5.1 First improvement: faster keepTrying

The first modification was to a function called `keepTrying`, which works with a word (the current head) in the UD tree and keeps trying to build new GF trees for that word based on the currently existing GF trees for the word and for its direct dependents, until no new trees can be built.

Before the improvement, pseudo-code of the algorithm would look roughly like this:

```
1. newTrees <- allFunsLocal(treesAtCurrentHead, childTrees)
2. combined <- deduplicate(treesAtCurrentHead ++ newTrees)
3. If combined != treesAtHead:
4.   treesAtHead <- combined
5.   Go to 1.
```

Here, the function `allFunsLocal` (explained more in section 5.2) tries to apply all possible GF functions to each of the trees at the current head. If the GF function that we try to apply takes more than one argument, then we choose that many trees from the dependents that the current tree has not used yet.

This had the issue that the same trees were calculated many times just to be immediately thrown away. To solve this, we only calculate new trees based on trees we have not seen before at the current head of the UD tree:

```
1. newTrees <- treesAtCurrentHead
2. newTrees <- Apply all matching functions on each of newTrees,
               together with a choice of unused
3. treesAtCurrentHead += newTrees
4. if notEmpty(newTrees): goto 2
5. Deduplicate the list of trees
```

In the first version, we only have one shared list of all the trees, both old and new trees mixed together. This means that the old trees will be used in every single iteration and time after time generate the same new trees. This produces a lot of duplicates that needs to be removed. In the new algorithm, we only generate based on the newest trees from the previous iteration, which eliminates these duplicates.

Since all previous trees were re-generated in each iteration, the old version was $O(n^2T)$ while the new version is $O(nT)$, where $n$ is the number of iterations (which is equal to the max depth of the tree) and $T$ is the number of possible trees at each level.

## 5.2 Second improvement: faster allFunsLocal

The second modification was to a function called `allFunsLocal`, which was responsible for finding all the possible combinations of functions and arguments at the current location in the UD tree that satisfy the constraints defined by the annotations. For example, for the function

```
#fun DetCN  : Det -> CN -> NP ; det  head
```

the first argument needs to come from a child with UD dependency annotation `det` and be a GF tree with category `Det` and the second argument needs to be at the current local head of the UD tree and have a GF category of `CN`.

The old algorithm did this by first listing all possible choices of the previously generated trees for the head and each direct child of the current node and then trying each function for each of those choices. The new algorithm instead starts with each individual function and then only checks choices that could possibly match that function and stops as soon as it has determined that a match is impossible.

Below is pseudocode for the old algorithm for allFunsLocal:

Given the following variables:

```
headNode = A local UD head node with a list of possible GF trees
childNodes = A list of UD child nodes each with a list of
             possible GF trees
gfFunctions = A list of all GF functions that we have available
              and their metadata
```

The algorithm looks as follows

```
1. For each gfFunction: f
2.   For each gf tree in headNode: headTree
3.     filteredChildNodes <- Filter out which childNodes have
                             not already been used by headTree
4.     childCombos <- Generate a list of all combinations, where
                      we select one GF tree from each element in
                      filteredChildNodes
5.     For each childCombo:
6.       For each argument of the function f: arg
7.         Select one element from childCombo or headTree which
             has not been used yet and which matches the required
             UD label and GF category
8.       Generate a list of all valid ways to select one valid
           childNode for each fun arg
```

This algorithm can also be expressed in terms of a sequence of choices:

```
1. f <- Choose a gfFunction
2. headTree <- Choose a gf tree in headNode
3. childCombo <- For each childNode not already used by headTree:
4.     Choose a GF tree from the child node
5. For each argument of the function f: arg
6.   Select one element from childCombo or headTree which has not
       been used yet and which matches the required UD label and
       GF category
7. Generate a list of all valid ways to select one valid
       childNode for each fun arg
```

This algorithm is unfortunately exponential. More precisely:

If we have $f$ functions, each taking $a$ arguments and if the headNode contains $h$ gf-trees and we have $c$ child-nodes, each containing $t$ gf-trees and each argument matches $m$ values from childCombo, we get an algorithmic complexity of:

$$O(fht^c(ac + m^a))$$

We can see that the complexity is exponential with respect to the number of child-nodes, with a base proportional to how many GF trees each of those child-nodes have. The complexity is also exponential with the number of arguments, but this has less of an impact since GF functions always have a fixed number of arguments, unlike UD trees, where a node can have an unlimited number of children, e.g. in conjunctions.

In addition to having an exponential complexity, this also produces duplicate trees. To see this we need to go through an example.

Let's say we have one function `DetCN : Det -> CN -> NP ; det head` which takes two arguments and a UD tree with a head node *cat* and two children *the* and *black*: (See Figure 4.15)

Now, if we were to apply this algorithm, we would generate these lists of combinations:

```
   cat ; head      | the ; det     | black ; amod
   cat_N      : N  | the_Det : Det | PositA black_A : AP
   UseN cat_N : CN |               | black_A        : A

1. cat_N      : N , the_Det : Det, black_A        : A
2. cat_N      : N , the_Det : Det, PositA black_A : AP
3. UseN cat_N : CN, the_Det : Det, black_A        : A
4. UseN cat_N : CN, the_Det : Det, PositA black_A : AP
```

If we now check where the function can be applied, we can see that both 3 and 4 matches, but in both cases they produce the same tree:

```
DetCN the_Det (UseN cat_N)
```

because the only difference is in the unused child node `black`.

In order to solve both the issue of poor complexity and producing duplicates, in the new algorithm, we delay the selection of tree until the latest possible moment.

```
1. For each gfFunction: f
2.   headArg <- Find the argument of f with label "head"
3.   For each headTree in headNode.trees where
              headTree.cat == headArg.cat:
4.     filteredChildNodes <- Filter out which childNodes have not
                             already been used by headTree
5.     For each non-head argument of f: arg
6.       For each node in filteredChildNodes, where
                   node.label == arg.label: childTree
7.         return each tree where childTree.cat == arg.cat
8.     Construct a list of trees using f and each of the valid
         child-trees (if any)
```

We can also express the same thing in terms of a sequence of choices:

```
1. Make a list of trees from every possible combination of
   choices below:
2.   f <- Choose a function from gfFunctions
3.   headArg <- Find the argument of f with label "head"
4.   headTree <- Choose a tree in headNode where
                    headTree.cat == headArg.cat:
5.   filteredChildNodes <- Filter out the childNodes that have not
                               already been used by headTree
6.   childArgTrees <- For each non-head argument of f: arg
7.       childNode <- Choose a node in filteredChildNodes, where
                         node.label == arg.label:
8.       Choose a tree in childNode where childTree.cat == arg.cat
9.   Construct a tree using f, headTree and each of the childArgTrees
```

Now $h_m$ is the number of trees in the head node which *matches* the head argument instead of every tree in the head node.

$$O(fh_m(c+m)^a)$$

The complexity here is in most cases very close to just the number of trees we are generating with only a linear overhead.

Because we delayed the choice of tree until we know we need it we are no longer generating duplicate trees. We also have the ability to stop early, as soon as one of the arguments are impossible to satisfy, we don't need to check the rest.

One possible inefficiency that remains is that if many children have the same label and category as each other we will generate every possible tree that uses these and then in a later stage filter out so we only have a single tree with that specific category.

It might be possible to keep track of these so we don't needlessly produce redundant trees that will soon be thrown away anyways. Another possible solution would be to have ordering constraints, so that with the otherwise equivalent children, they are forced to be used in a specific order. This would be especially relevant with coordinate structures (see also chapter 7), in which each child has the same category and dependency label and the ordering is important.

# 6

# Debugger

## 6.1 Problem description

An issue that was encountered when trying to write annotations for `ud2gf` is that sometimes it can be difficult to figure out why a rule does not fire.

The goal of the debugger project is to automatically give an explanation of why a rule (function label description) doesn't match.

Let's say we have the sentence "colorless green ideas sleep furiously" and believe that the function `DetCN : Det -> CN -> NP ; det head` should be applied to "green" and "ideas" respectively, but it doesn't seem to be used for some reason. Then we would call the `ud2gf` tool with either `dbg="DetCN 2 3"` or `dbg="DetCN green ideas"` and the debugger would give back one of a list of reasons to what prevented it from being applied:

- The function has been manually disabled.
- The function does not exist.
- There are multiple definitions of the function.
- The wrong number of arguments was given to the debugger (meta-error).
- The requested indices are not found within the UD tree.
- The non-head arguments are not direct children of the head argument.
- The non-head arguments have labels that don't match the labels within the UD-tree.
- Required features are missing for some of the arguments.
- No trees with the required category could be found for some of the arguments.
- The function was possible to apply, but was pruned away because a different tree with the same category had higher priority.

If a word exist multiple times in the sentence, you need to use the number form, otherwise you will get an "Ambiguous word" error. In order to see the index number of a word you can either look at the conllu file[1] directly or run `ud2gf` with the `ud` or

[1] Conllu is the standard file format for UD trees.

`ut` argument.

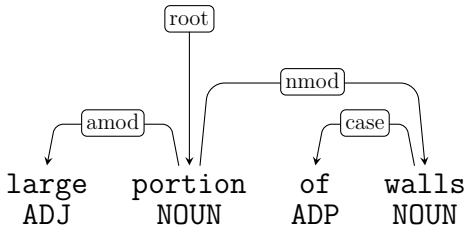
## 6.2 Implementation

The implementation is mostly a case of going through every step of the algorithm and seeing what can go wrong. Some of this process was simplified by the strong type system of Haskell, since it makes edge cases explicit, so in most cases we would get a type error if we missed handling a possible error.

In order to simplify the implementation and reduce code duplication, we first run the main algorithm to generate all possible GF trees in each node of the UD tree and use this result to check what goes wrong with the specific function we are interested in.

## 6.3 Example

Let's look at the phrase "large portion of walls", which in UD format looks like this:




First we try to form a tree of words that are not in a head–dependent relation. The tool reports that *large* is not a child of *walls*, and suggests an actual child of *walls* (i.e. *of*) as a hint for the grammar writer.

```
$ cat tests/examples/large_portion_of_walls.conllu | gf-ud ud2gf tests/
    grammars/Test Eng UDS no-backups dbg='AdjCN large walls'

Starting debug for AdjCN:
AdjCN : AP -> CN -> CN ; amod head
Error: Word number 1 ("large") is not a child of 4 ("walls").
    Available children: [("3","of")]
```

Next, we ask the debugger about a correct attachment, but we try to apply a function of the wrong type. The debugger answers with "Incompatible argument labels".

```
$ cat tests/examples/large_portion_of_walls.conllu | gf-ud ud2gf tests/
    grammars/Test Eng UDS no-backups dbg='DetCN large portion'

Starting debug for DetCN:
DetCN : Det -> CN -> NP ; det head
Attempting to build: DetCN large portion
Error: Incompatible argument labels:
 - For "large": Got amod expected det
```

Finally, if we debug a correct application, we get a step by step trace of how the tree is constructed.

```
$ cat tests/examples/large_portion_of_walls.conllu | gf-ud ud2gf tests/
    grammars/Test Eng UDS no-backups dbg='AdjCN large portion'

Starting debug for AdjCN:
AdjCN : AP -> CN -> CN ; amod head
Attempting to build: AdjCN large portion

Argument "large" : AP ; amod:
  Found trees with correct category:
    - (PositA large_A) : AP

Argument "portion" : CN ; head:
  Found trees with correct category:
    - (AdvCN (UseN portion_N) (PrepNP of_Prep (DetCN_aPl (UseN wall_N)))) :
        CN
Trees using AdjCN found in devtree:
    (AdjCN (PositA large_A) (AdvCN (UseN portion_N) (PrepNP of_Prep (
       DetCN_aPl (UseN wall_N))))) : CN
Success!
```

# 7

# Improving flexibility of macros

## 7.1 The macro system

GF trees are on a higher abstraction level than UD trees. In UD, every token is associated with a dependency label. In contrast, any production rule in a GF grammar may introduce new tokens. This difference complicates the mapping between the two formalisms.

In order to overcome some limitations of the conversion from UD to GF, especially the issue with multiple possible matching trees mentioned in subsection 4.3.2, there is a system of macros. A macro is an imaginary GF function that will get converted into an expression of real GF functions after the main conversion step of gf2ud is done.

The full syntax of macros can be found in Appendix A.

### 7.1.1 Example 1: `#auxfun` and matching morphological features

When one GF function corresponds to multiple different dependency labels, a macro can be used to disambiguate which one is meant and to avoid losing the information of that distinction.

For example, the GF grammar contains two ways of building a noun phrase without a determiner: an indefinite plural, using `DetCN aPl_Det : CN -> NP`[1], or mass noun, using `MassNP : CN -> NP`. Knowing which function to choose is trivial for a human reader, because we see it from the form of the word: *water* in "I drink *water*" is a singular mass noun, and *children* in "*Children* are playing" is an indefinite plural. But for the ud2gf algorithm, this is impossible with the tools we have seen so far.

The annotation scheme introduced in section 4.2 operates on abstract syntax only, and thus it has no way to distinguish between those cases, because it has no information about concrete morphological features.

In order to choose the right function depending on features beyond the dependency label, the macros can also match against part of speech and morphological features

[1] Becuase the `aPl_Det` function does not correspond to a word, the algorithm would actually not try to construct it.

using the syntax `dependency_label[Feature=Value]`. Using this syntax, we can make macros for distinguishing between mass nouns and indefinite plurals:

```
1    -- disable plurals as mass terms
2    #disable MassNP
3    #auxfun MassNP_sg cn : CN -> NP = MassNP   cn ; head[Number=Sing]
4    #auxfun DetCN_aPl cn : CN -> NP = DetCN aPl_Det cn ; head[Number=Plur]
```

In order to ensure that the macros are used instead of the original functions, we first `#disable` all the original functions that might get applied too liberally (in this case, only `MassNP`). Next we create macros that check for the correct number of the noun before constructing a noun phrase.

Let's consider the sentence "Children are playing", and what would happen to it if we didn't force mass nouns and indefinite plurals to be introduced via macros.

The algorithm from section 4.3 goes through each word[2], and tries all possible functions that can be applied to them. However, `MassNP` is the only function we can apply to the word *child*: the algorithm would never apply the correct `DetCN aPl_Det` without a macro, because the lexical function `aPl_Det` does not correspond to any word, and thus would be never introduced into the tree.

So the two macros fill two purposes: preventing `MassNP` from being applied too liberally, and making `DetCN aPl_Det` possible in the first place.

Note that none of this would be a problem, if the original phrase had a determiner. If the sentence were instead "Many children are playing", then the NP formed by `MassNP` would still be formed, but discarded as soon as the algorithm enters the next step and tries to combine the noun and the determiner.

### 7.1.2 Example 2: `#auxcat` and syncategorematic words

Another common use case for macros is to handle so-called *syncategorematic* words. Syncategorematic words are words that are introduced as a part of the linearization of some syntactic function, and don't have a GF category or function of their own.

When introducing the problem and its partial solution, in [9] Ranta and Kolachina name as examples copulas, negations, tense auxiliaries and infinitive marks. We reproduce their example of the English copula (the verb "to be") and its treatment in `ud2gf`.

Figures 7.1 and 7.2 show the difference between GF's and UD's treatment of copulas. In the UD tree, the copula has a category and a dependency label `cop`. In the GF tree, there is no special category: the string "is" is introduced by the function `UseAP : AP -> VP` rule, which makes an AP into a predicate.

As a solution, `ud2gf` introduced the notion of *auxiliary categories* and auxiliary functions. This time we use the following auxiliary category and auxiliary function:

[2]Remember that all the words are parsed into a lexical category, which is determined by the `#cat` annotation.

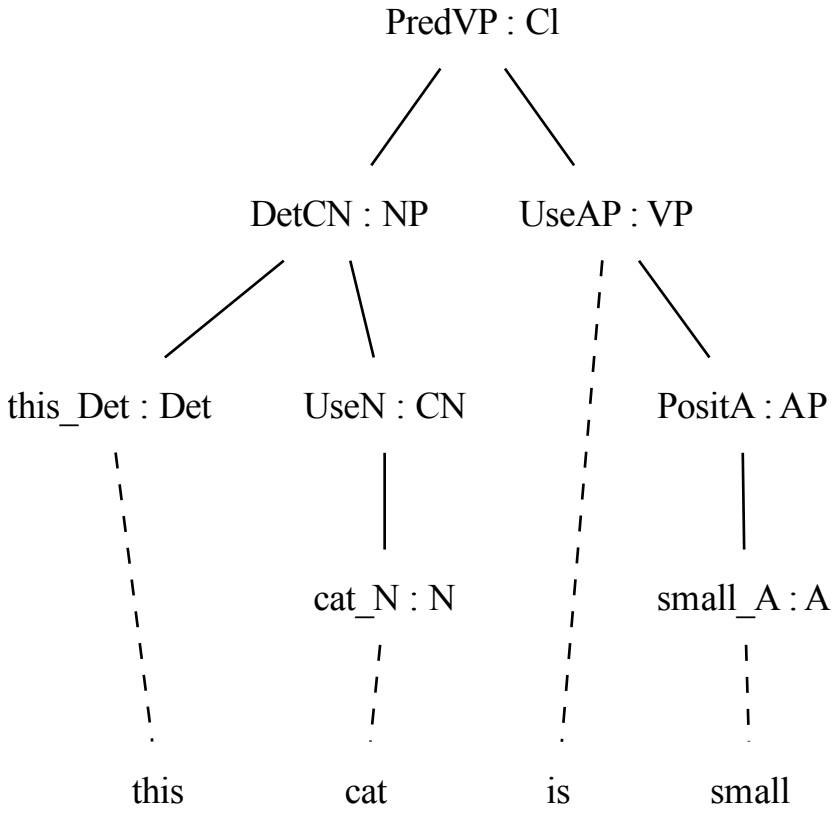



Figure 7.1: The phrase “This cat is small” analysed as a GF tree.

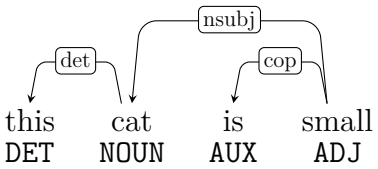



Figure 7.2: The phrase “This cat is small” analysed as a UD tree.

- Auxiliary category for copula: `#auxcat Cop VERB`. This annotation introduces an auxiliary category `Cop` and associates it with the UD part of speech `VERB`.
- Auxiliary function (macro) that recognises the auxiliary category:
  `#auxfun UseAP_ cop ap : Cop -> AP -> VP = UseAP ap ; cop head.`
  The macro takes a copula and an adjectival phrase and then throws away the copula, because the copula (*is*) is created by the `UseAP` function.
- A declaration of the lemma *be*:
  `#lemma DEFAULT_ be Cop cop head.`
  Here we declare that the lemma can be used with all functions (`DEFAULT_`), the lemma is *be*, the auxiliary category should be *Cop*, the dependency label pointing at the lemma should be *cop* and that the *head* argument of the function should be the parent of the copula. Most parts of the `#lemma` declaration are only used for the gf2ud direction. Only the lemma name and category name are used by ud2gf.

However, coordination was still a problem in `ud2gf`. The auxiliary categories and functions were not expressive enough to transform a structure like "the small, cute and fluffy cat" from UD to GF.

## 7.2 Problem: Limitations of the possible tree transformation

As an example of a phrase that can be difficult to convert using the old ud2gf, let us consider the adjectival phrase "small, furry, fluffy and cute" would be described in UD format as in Figure 7.3.

The GF version of the same tree, would look like in Figure 7.4.

Here we can see that in UD, the word "cute" is in the root, while the conjunction "and" is at the bottom of the tree, while in GF the conjunction is a direct child of the root. This transformation can not be preformed by the simple single-layer transformations that were available in the old macro-system.

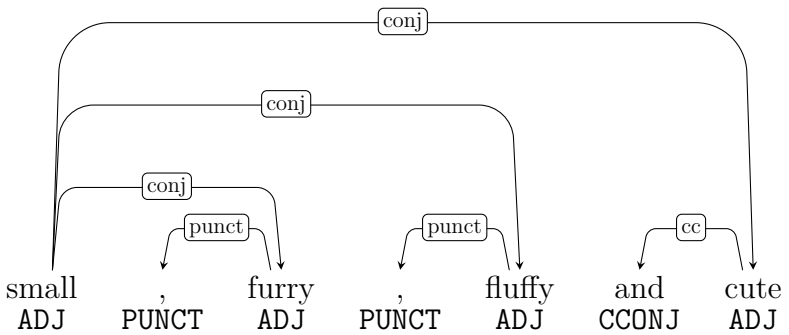



Figure 7.3: The phrase "small, furry, fluffy and cute" as a UD tree in graphical format

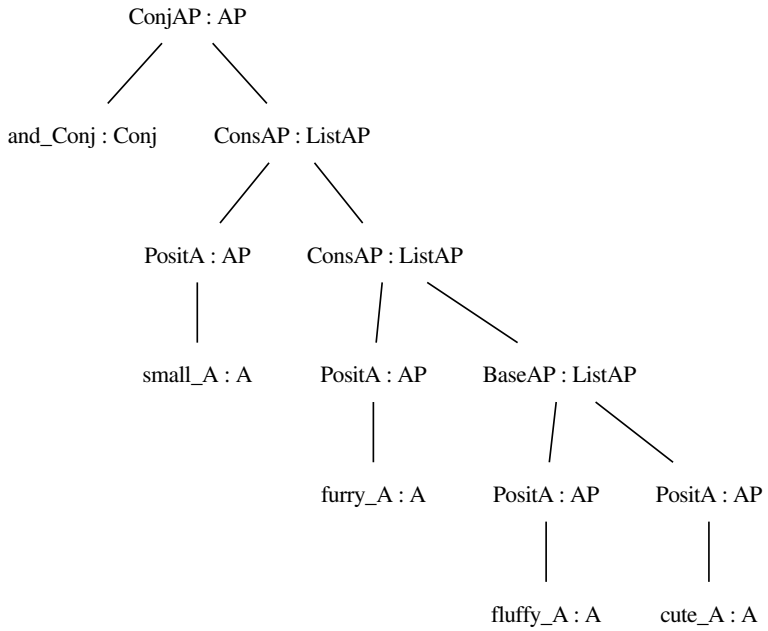



Figure 7.4: The phrase "small, furry, fluffy and cute" as a GF tree in graphical format.

The main issue that made it impossible for the old macros to perform this transformation is that we need to extract several pieces of information from a node and then use them independently.

## 7.3 Solution: Recursive macros

In order to improve the ability to create structurally different trees when converting from UD to GF, the ability for macros to be recursive was added. This means that when a macro is substituted, the resulting head is checked again to see if the result is another macro, until no more substitutions are possible. This makes it possible to encode data using Church encoding, in particular the Church encoding for pairs: a higher order function that takes a binary function as an argument and provides the two items it contains to the inner function. This construction of pairs can also be interpreted as continuation-passing style, where instead of constructing a pair, we take a continuation which would consume the two arguments of the pair.

```
type Pair a b = forall c. (a -> b -> c) -> c
```

In order to construct and consume these pairs we use the following helper functions:

```
1 -- The syntax of macros type signatures does not allow higher order
2 -- functions types, so we use the following abbreviations:
3 -- ab2r = a -> b -> r
4 -- Pair_a_b = ab2r -> r
5 #auxfun MkPair_ a b handler : a -> b -> ab2r -> r = handler a b ; head
    dummy dummy
```

```
6 #auxfun UsePair_ handler pair : ab2r -> Pair_a_b -> r = pair handler ;
    head dummy
```

When using these helpers, you would partially apply the `MkPair_` helper, so it can later get the final `handler` argument from `UsePair_`[3]. Because the helpers are only used by other macros, we don't need any real dependency labels but instead use dummy values to fill them in (here `dummy`, but anything would work).

In addition to pairs, we will also need triples, which work similarly:

```
1 -- Triple_a_b_c = (a -> b -> r) -> r
2 #auxfun MkTriple_ a b c handler : a -> b -> c -> abc2r -> r = handler a b
    c ; head dummy dummy dummy
3 #auxfun UseTriple_ handler triple : abc2r -> Triple_a_b_c -> r = triple
    handler ; head dummy
```

As can be seen in Figure 7.3, each word but the first and last in a coordinate structure has a comma as a dependent in UD. This macro handles that.

```
1 #auxcat Comma PUNCT
2 #lemma DEFAULT_ , Comma punct head
3 #auxfun CommaAP_ ap comma : AP -> Comma -> APComma = ap ; head punct
```

The last word has the conjunction as a child in UD, while the conjunction in GF abstract syntax is inserted at the very top, as can be seen in Figure 7.4. This is where we first use our `MkPair_` helper in order to carry along the last word and the conjunction separately.

```
1 #auxfun AndCuteCont_ and cute : Conj -> AP -> Pair_Conj_AP = MkPair_ and
    cute ; cc head
```

Because lists in GF have a minimum length of two, we need two base cases: one with exactly two conjuncts and one with three or more.

The simplest case is when we only have two words with a conjunction in between ("small and cute"). The two functions below handle this case.

```
1 #auxfun AP2_ small andCute : AP -> Pair_Conj_AP -> AP = UsePair_ (
    AP2_helper_ small) andCute ; head conj
2 #auxfun AP2_helper_ small and cute : AP -> Conj -> AP -> AP = ConjAP and (
    BaseAP small cute) ; head dummy dummy
```

We need a separate `AP2_helper_` function to handle the extra argument `small` because we don't have support for lambdas or closures. We can see that the main function `AP2_` gets both parts of the pair "and cute" as a bundle, while the `AP2_helper_` gets them as separate arguments.

The next case is when we have at least three conjuncts with a conjunction ("small, fluffy and cute"). Here we need to create a triple, with:

[3]The `UsePair_` helper is not strictly necessary, since it can be replaced with function application (`UsePair_ handler pair = pair handler`), but it helps making the code clearer. The `MkPair_` on the other hand is necessary, because we need to partially apply it.

- the conjunction (“and”)
- the first word in the coordination, which is also the head of the UD tree (“small”)
- the two last conjuncts (“fluffy” and “cute”) which form base of the GF list (`BaseAP fluffy cute`).

```
#auxfun APBaseComma_ small fluffy andCute : AP -> APComma -> Pair_Conj_AP
    -> ConjListAP = UsePair_ (APBaseComma_helper_ small fluffy) andCute ;
    head conj conj
#auxfun APBaseComma_helper_ small fluffy and cute : AP -> APComma -> Conj
    -> AP -> ConjListAP = MkTriple_ and small (BaseAP fluffy cute) ; head
    dummy dummy
```

The next case is for four or more conjuncts with a conjunction (“small, furry, fluffy and cute”). Here we deconstruct the triple, add the middle word (“furry”) to the list (`ConsAP furry (BaseAP fluffy cute)`) and reconstruct the triple.

```
#auxfun APAddComma_ furry and_small_fluffyCute : APComma -> ConjListAP ->
    ConjListAP = UseTriple_ (APAddComma_helper_ furry)
    and_small_fluffyCute ; conj head
#auxfun APAddComma_helper_ furry and small fluffyCute : APComma -> Conj ->
     AP -> ListAP -> ConjListAP = MkTriple_ and small (ConsAP furry
    fluffyCute) ; dummy head
```

Finally, we have can take a complete list and convert it into an adjectival phrase (AP). We deconstruct the triple of: a conjunction (“and”), first content word (“small”) and list of the remaining conjuncts (“furry”, “fluffy”, “cute”) and construct the final coordinate structure: `ConjAP and (ConsAP small furryFluffyCute)` to arrive at the GF tree that can be seen in Figure 7.4.

```
#auxfun ConjListToAP2_ and_small_furryFluffyCute : ConjListAP -> AP =
    UseTriple_ ConjListToAP2_helper_ and_small_furryFluffyCute ; head
#auxfun ConjListToAP2_helper_ and small furryFluffyCute : Conj -> AP ->
    ListAP -> AP = ConjAP and (ConsAP small furryFluffyCute) ; notreal
    head dummy
```

The tree transformation algorithm would construct all the possible subtrees, regardless of if they have consumed all conjuncts or not. However in the end, the one that has consumed all the conjuncts will be selected because it’s the most complete.

The complete code for handling conjunctions using recursive macros can be seen in Appendix B.

## 7.4 Limitations and ideas for further improvements

A more structured implementation of this could be to add (anonymous) records as well as pattern matching to the syntax of macros similar to the concrete syntax of

GF. A suggestion for how it could look can be seen commented out in Appendix B.

It could also be useful to have a more direct approach for describing the structural changes, especially since macros are only available in the `ud2gf` direction, not in the `gf2ud` direction.

# 8

# Results and discussion

## 8.1 Performance

The result of running different versions of the algorithm on the example file called `upto12eng.conllu` using the example grammar `ShallowParse` in the gf-ud git repository can be seen in Figure 8.1. The full list of sentences included in the benchmark can be seen in Appendix D. All benchmarks were perfomed on a 2019 MacBook Pro, with a 2.3 GHz 8-Core Intel Core i9 CPU.

In Figure 8.2 we can see the speedup for each individual sentence. The improvement of the keepTrying function gives a close to linear speedup, while the speedup from the optimized version of allFunsLocal is much larger. The theoretical expected speedup from keepTrying is a quadratic factor that becomes a linear factor based on the depth of the resulting trees. The library Criterion[1] was used to perform the measurements. A full table of results can be found in Appendix C.

In Figure 8.3 we can see that the speedup from the keepTrying improvement is linear with respect to the logarithm of the original code, which matches the expectation of moving from converting a quadratic factor to a linear factor. Looking at the slowest sentence “In Danish, the word may even apply to shallow lagoons”, it takes 83 seconds with the original algorithm and 10 seconds with the keepTrying improvement, giving a speedup factor of 8.3. Looking at the generated tree we can confirm that it has a maximum tree depth of 9 at the top, matching the expectation.

In Figure 8.4 we can see that the allFunsLocal improvement has a speedup factor that is linear on the log-log plot, which indicates that we got an exponential speedup as expected from the theory.

### 8.1.1 Garbage Collection time

As we saw in Figure 8.1 and Table 8.1, when the optimized algorithm is used, over 80% of time is spent on garbage collection. This is in a large part because GHC uses a generational, moving garbage collector[26], which means that the cost of a garbage collection is proportional to the amount of currently living data. In our case we have a large amount of long-living data, which means that every major garbage collection is expensive. The code begins by loading the GF grammar into memory,

[2]http://www.serpentine.com/criterion/

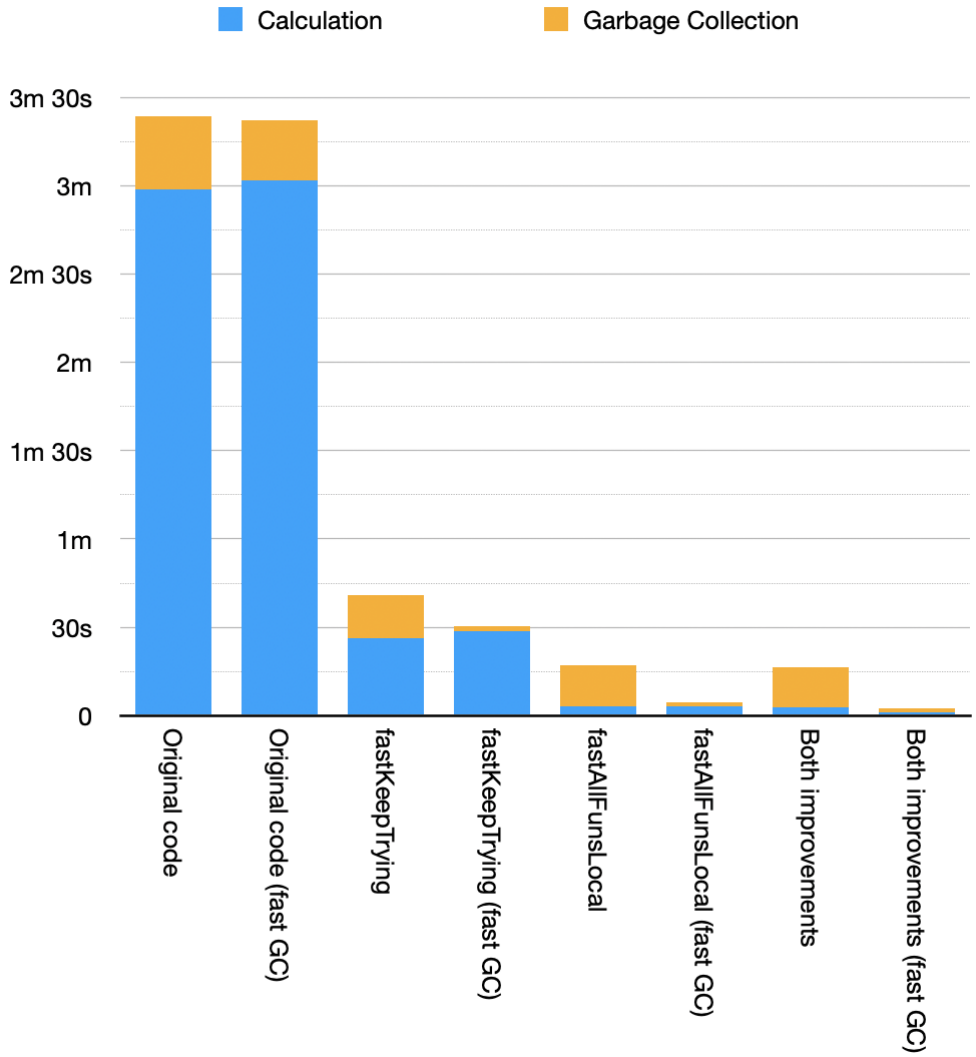



Figure 8.1: The total run time for converting the file `upto12eng.conllu`, including garbage collection and startup time. The bars marked "fast GC" have increased initial heap size to 500MB to reduce the number of unnecessary garbage collections, based on the experiments in subsection 8.1.1. Each measurement was only performed once, so there might be some inaccuracy because of random noise resulting from preemptive multitasking.

| Time (seconds) | Calculation | Garbage Collection | Total |
|---|---|---|---|
| Original code | 2m 58.8s | 24.6s | 3m 23.5s |
| Original code (fast GC) | 3m 01.9s | 20.2s | 3m 22.1s |
| fastKeepTrying | 26.2s | 14.7s | 40.9s |
| fastKeepTrying (fast GC) | 28.6s | 1.7s | 30.3s |
| fastAllFunsLocal | 3.2s | 14.3s | 17.2s |
| fastAllFunsLocal (fast GC) | 3.2s | 1.3s | 4.5s |
| Both improvements | 2.7s | 13.9s | 16.6s |
| Both improvements (fast GC) | 1.25s | 1.25s | 2.5s |

Table 8.1: The total run time for converting the file upto12eng.conllu, including garbage collection and startup time. The bars marked "fast GC" have increased initial heap size to 500MB to reduce the number of unnecessary garbage collections, based on the experiments in subsection 8.1.1. Each measurement was only performed once, so there might be some inaccuracy because of random noise resulting from preemptive multitasking.

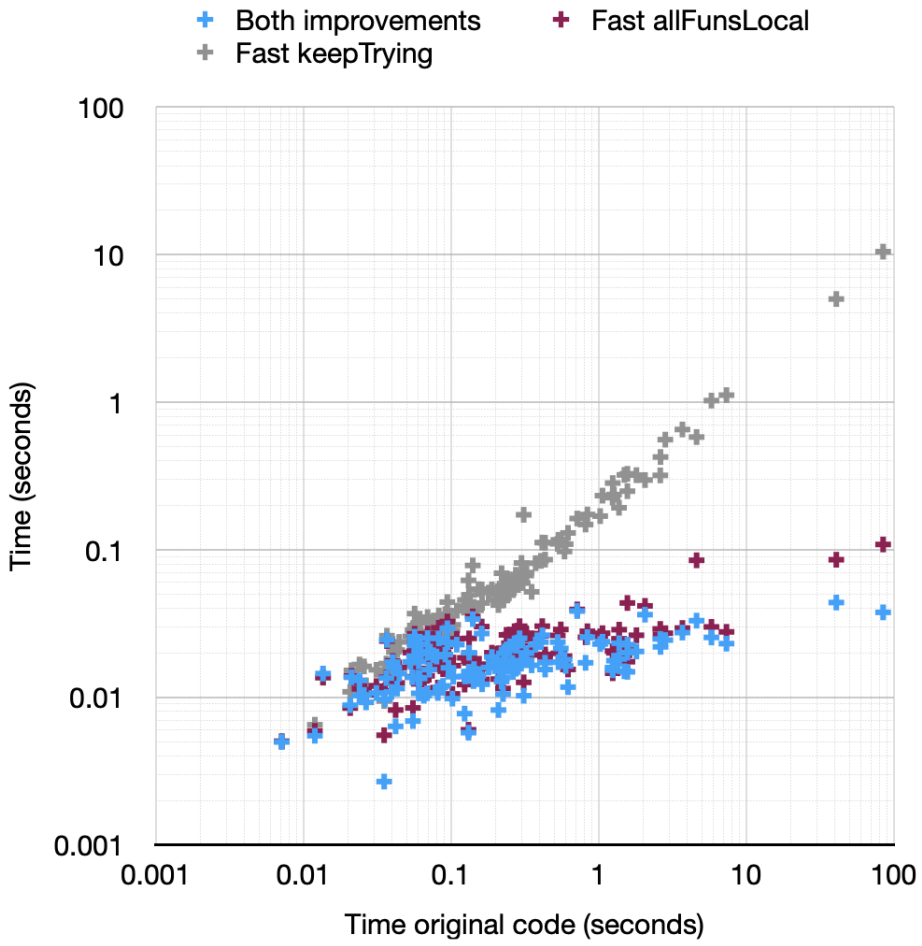



Figure 8.2: A log-log plot of time for each optimization against the time for the original algorithm. Each data point is the average time over several runs for a single sentence. The library Criterion[2] was used to perform the measurements.

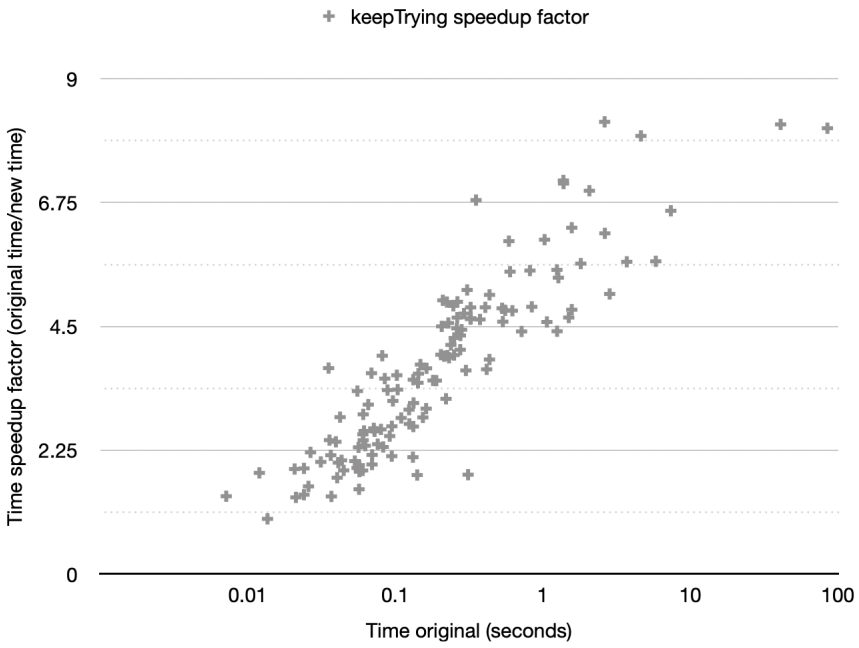



Figure 8.3: A plot of the speedup factor for the improved keepTrying algorithm: new time divided by original time, against the original time taken. A linear speedup is the expected result for converting a quadratic algorithm to a linear algorithm.

which for our test grammar takes up around 100 megabytes. There are several ways to mitigate this, but the easiest solution is to tweak the parameters for the garbage collector to make it wait longer before attempting to collect garbage, which reduces the total number of major garbage collections.

The tool ghc-gc-tool allows automatically determining which parameters are best for this by running the executable with different parameters and plotting the result. In Figure 8.6 we can see the result of running this on the first 60 items of `upto12eng.conllu` with the ShallowParse grammar from the repository. This number was chosen arbitrarily to make the runtime not be unreasonably long.

As can be seen in Figure 8.6b any initial heap size over 256MB drastically reduces the run time and the logs show that the productivity (time spent on non-GC) goes from 15% to 50% and that less than one tenth as many garbage collections are performed. This number depends on the size of the GF grammar and it corresponds to the size of the GF grammar after being loaded into memory. Running the command with the default GC parameters shows that the maximum residency is 180MB, which is slightly less than the optimal GC parameter value .

We also explored other methods for reducing the impact of garbage collection, like compact regions[27], but they provided no improvement over the simpler method of tweaking the GC parameters and in some cases using them made performance worse, because they force all data stored to be fully evaluated.

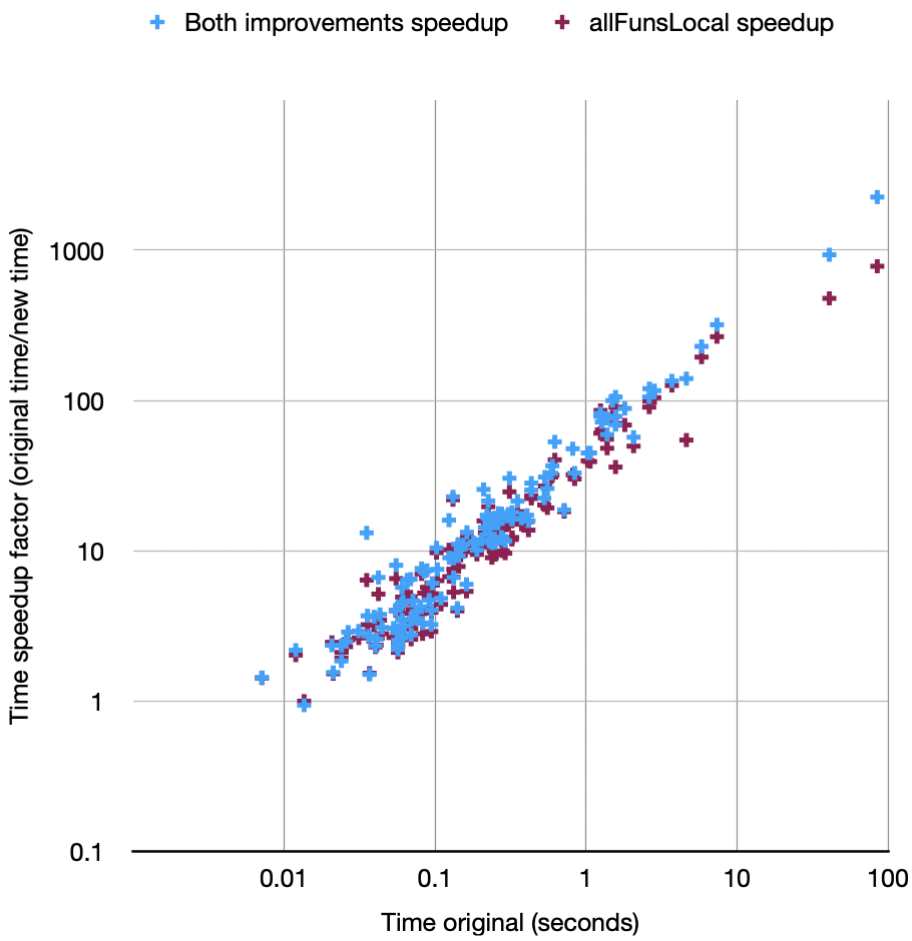
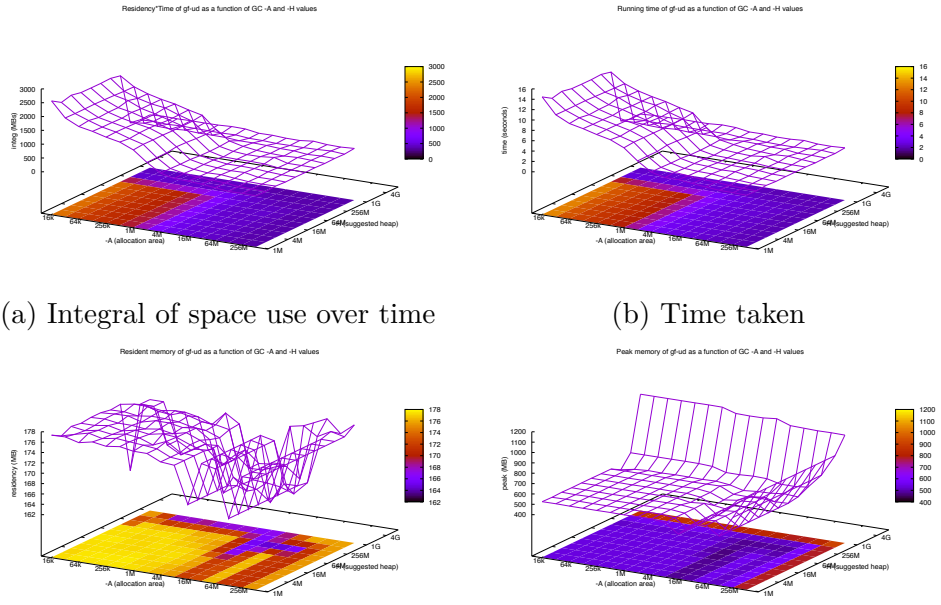



Figure 8.4: A log-log plot of the speedup factor for the improved allFunsLocal algorithm: new time divided by original time, against the original time taken. The linear pattern indicates an exponential speedup.

```
$ ghc-gc-tune -t pdf -spr gf-ud ud2gf grammars/ShallowParse Eng Text
...
gf-ud +RTS -A32768 -H134217728 -RTS ud2gf grammars/ShallowParse Eng Text
    <<GCs 125563, peak   538, resident 177.67m, MUT 1.728s, GC 9.609s>>
gf-ud +RTS -A32768 -H268435456 -RTS ud2gf grammars/ShallowParse Eng Text
    <<GCs   8655, peak   509, resident 176.64m, MUT 1.211s, GC 1.776s>>
...
```

Figure 8.5: Selected lines from the output for the ghc-gc-tune command.




(a) Integral of space use over time

(b) Time taken

(c) Resident space usage

(d) Peak space usage

Figure 8.6: The integral of space usage over time for different GC parameters.

A yet unexplored option is to try using the GF C runtime instead of the GF Haskell runtime. This should also solve the garbage collection issues, since C is not a garbage collected language and the C heap is not touched by the Haskell garbage collector.

## 8.2 Correctness

Because the accuracy of the conversion depends on manually written annotation files, evaluating the accuracy of the conversion is outside of the scope for this paper.

The optimization of the `allFunsLocal` function had no impact on the generated trees. However, in some cases the optimization of the `keepTrying` functions caused different trees to be picked. The different trees are equally valid according to the defined constraints in the annotations and some of them are better and some are worse fits in the example trees we evaluated. See subsection 4.3.2 for more details on the multiple possible trees.

## 8.3 Debugging

The debugging tool that was written in this project allows finding what prevented an #auxfun macro from being used to convert a tree and anecdotal evidence points towards it being helpful when writing annotations.

## 8.4 Flexibility

The new recursive macros made it possible to express more significant changes in tree shapes when converting between UD and GF. The motivating example is *coordination*: structures like "big *or* small" or "cats, dogs *and* capybaras". Before the recursive macros, it was only possible to convert structures with exactly 2 conjuncts, since those had sufficiently similar structure, but now ud2gf can convert coordinate structures of arbitrary length.

## 8.5 Use in robust parsing

As mentioned in section 2.1, the improvement of ud2gf was done as a part of the SMU CCLAW project, with the goal of using ud2gf as a part of a pipeline for parsing unrestricted text in the legal domain. Experiments on that were performed on a small scale, but the results were deemed unsatisfactory for that specific use. A major part of the problem was the correctness of udpipe itself: legal text contains many uncommon structures, and the initial parses were inaccurate relatively often. Given the dissatisfaction with the approach, we never did a quantitative evaluation of the results.

Anecdotally, we found the macro system useful in recovering from parser errors, but

it was a lot of tedious work[3], with a long tail of fixes that only applied to a single sentence.

[3]Example auxfuns written to recover from errors in udpipe output can be found at `https://github.com/smucclaw/sandbox/blob/default/inari/ud/copied-from-dsl/grammars/UDAppEng.labels#L105-L136`

# 9

# Conclusion and future work

## 9.1 Summary

In this project we set out to improve the overall usability of ud2gf, focusing on issues with performance, difficulty in debugging and lack of flexibility regarding differing tree shapes.

**Performance** We reduced some combinatorial explosions and prevented same candidates being generated multiple times in the tree generation phase. The performance was greatly improved, with the slowest sentence in our test improving from 85 seconds to 44 milliseconds. One of the optimizations improved the performance by a linear factor depending on the depth of the tree, while the other optimization improved the performance by an exponential factor depending on the width of the tree.

We could also see that garbage collection can have a large impact on the performance of Haskell programs. However, in our case it could easily be alleviated, without a significant impact on total memory use, by increasing the initial heap size for the program.

Another future option to improve performance would be to try using the GF C runtime instead of the GF Haskell runtime.

**Debugging** We added a debugger feature, where the user can query ud2gf for an explanation why a certain function was not applied. This feature was much less mature compared to the performance improvements, but the single user who tested it gave positive feedback.

**Tree shape flexibility** We made the macro system recursive, which made it possible to do conversions that require more context than just the immediate children. The improved macro system solved the issue, in that we can now express things we could not do before, but the syntax is not very user-friendly.

## 9.2 Future work

The translation described by a labels file is not one-to-one and there are often many possible GF trees that a UD tree could be translated to. The possible trees are currently ranked by completeness, as in how many of the words are included in the generated tree. However this ranking is incomplete and in case two possible trees, with the same GF category, cover the same words, an arbitrary tree will be chosen. A better choice could be to also check the linearization of these trees and rank those whose linearization is more similar to the original string higher. It would also be possible to completely exclude trees with differing linearization, but that would run counter to the goal of robustness.

There are still many cases that the conversion can not handle, for example: If a GF grammar has functions of type `: A -> A`) or some combination of functions that can achieve that type, we can get an infinite loop of that function being repeatedly applied, so it is important that such functions are not included in the labels file. This is not detected automatically, so it is up to the user to notice it and fix the issue.

The debugging tool is mostly a rough, but useful, prototype, so it would benefit from more polishing.

The expanded macro capabilities would benefit from having their own syntax to get a nicer and more user-friendly interface.

A suggestion that has come up was to use a variation of the chart parsing algorithm instead of a bottom-up parser to further improve the performance. It is however not trivial to adopt the algorithm to start from a tree instead of a linear sequence of words, so further work is neecessary. Furthermore, the performance improvements were already sufficent for the application at hand, so it was not deemed necessary.

# Bibliography


[1] N. Chomsky, *Syntactic Structures*. The Hague: Mouton, 1957.

[2] J. Lambek, "The mathematics of sentence structure," *American Mathematical Monthly*, vol. 65, pp. 154–170, 1958.

[3] H. B. Curry, "Some logical aspects of grammatical structure," *Structure of language and its mathematical aspects*, vol. 12, pp. 56–68, 1961.

[4] A. Ranta, *Grammatical Framework: Programming with Multilingual Grammars*. Stanford: CSLI Publications, 2011.

[5] J. Nivre, M.-C. de Marneffe, F. Ginter, *et al.*, "Universal Dependencies v1: A Multilingual Treebank Collection," in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, Portorož, Slovenia: European Language Resources Association (ELRA), May 2016, pp. 1659–1666. [Online]. Available: `https://www.aclweb.org/anthology/L16-1262`.

[6] L. Tesnière, *Elements of structural syntax*. John Benjamins Publishing Company, 2015.

[7] J. Nivre, *Inductive dependency parsing*. Springer, 2006.

[8] A. Ranta, "Grammatical Framework: A Type-Theoretical Grammar Formalism," *The Journal of Functional Programming*, vol. 14(2), pp. 145–189, 2004.

[9] A. Ranta and P. Kolachina, "From Universal Dependencies to Abstract Syntax," in *Proceedings of the 1st Workshop on Universal Dependencies*, Linköping University Electronic Press, May 2017.

[10] P. Kolachina and A. Ranta, "From Abstract Syntax to Universal Dependencies," *Linguistic Issues in Language Technology*, vol. 13, pp. 1–57, 3 2016.

[11] A. Ranta, K. Angelov, N. Gruzitis, and P. Kolachina, "Abstract Syntax as Interlingua: Scaling Up the Grammatical Framework from Controlled Languages to Robust Pipelines," *Computational Linguistics*, vol. 46, no. 2, pp. 425–486, Jun. 2020, `https://doi.org/10.1162/coli_a_00378`.

[12] A. Masciolini and A. Ranta, "Grammar-based concept alignment for domain-specific machine translation," in *Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL 2020/21)*, `https://aclanthology.org/2021.cnl-1.2.pdf`, 2021.

[13] I. Listenmaa, M. Hanafiah, R. Cheong, and A. Källberg, "Towards CNL-based verbalization of computational contracts," in *Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL 2020/21)*, Amsterdam, Netherlands: Special Interest Group on Controlled Natural Language, Sep. 2021. [Online]. Available: `https://aclanthology.org/2021.cnl-1.10`.

[14] "Inductive dependency parsing," in *Inductive Dependency Parsing*. Dordrecht: Springer Netherlands, 2006, pp. 87–120, ISBN: 978-1-4020-4889-0. DOI: 10.1007/1-4020-4889-0_4. [Online]. Available: https://doi.org/10.1007/1-4020-4889-0_4.

[15] M. Straka, J. Hajič, and J. Straková, "UDPipe: Trainable pipeline for processing CoNLL-U files performing tokenization, morphological analysis, POS tagging and parsing," in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, N. Calzolari, K. Choukri, T. Declerck, *et al.*, Eds., Portorož, Slovenia: European Language Resources Association (ELRA), May 2016, pp. 4290–4297. [Online]. Available: https://aclanthology.org/L16-1680.

[16] J. Kanerva, F. Ginter, and S. Pyysalo, "Turku enhanced parser pipeline: From raw text to enhanced graphs in the IWPT 2020 shared task," in *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, G. Bouma, Y. Matsumoto, S. Oepen, *et al.*, Eds., Online: Association for Computational Linguistics, Jul. 2020, pp. 162–173. DOI: 10.18653/v1/2020.iwpt-1.17. [Online]. Available: https://aclanthology.org/2020.iwpt-1.17.

[17] R. Montague, "The proper treatment of quantification in ordinary english," in *Approaches to Natural Language*, P. Suppes, J. Moravcsik, and J. Hintikka, Eds., Dordrecht, 1973, pp. 221–242.

[18] A. Ranta, "Computational Semantics in Type Theory," *Mathematics and Social Sciences*, vol. 165, pp. 31–57, 2004.

[19] A. Ranta, I. Listenmaa, J. Soh, and M. W. Wong, "An end-to-end pipeline from law text to logical formulas," in *Legal Knowledge and Information Systems*, IOS Press, 2022, pp. 237–242.

[20] S. Reddy, O. Täckström, M. Collins, *et al.*, "Transforming dependency structures to logical forms for semantic parsing," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 127–140, 2016.

[21] A. Ranta, "Explainable machine translation with interlingual trees as certificates," 2017.

[22] A. Ranta, "Translating between language and logic: What is easy and what is difficult," in *Automated Deduction – CADE-23*, N. Bjørner and V. Sofronie-Stokkermans, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 5–25, ISBN: 978-3-642-22438-6.

[23] J.-P. Bernardy and S. Chatzikyriakidis, "A Type-Theoretical system for the FraCaS test suite: Grammatical Framework meets Coq," in *IWCS 2017*, 2017.

[24] P. J. Landin, "The next 700 programming languages," *Commun. ACM*, vol. 9, no. 3, pp. 157–166, Mar. 1966, ISSN: 0001-0782. DOI: 10.1145/365230.365257. [Online]. Available: https://doi.org/10.1145/365230.365257.

[25] K. Angelov and G. Lobanov, "Predicting translation equivalents in linked wordnets," in *The 26th International Conference on Computational Linguistics (COLING 2016)*, 2016, p. 26.

[26] D. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," *ACM Sigplan notices*, vol. 19, no. 5, pp. 157–167, 1984.

[27] E. Z. Yang, G. Campagna, Ö. S. Ağacan, A. El-Hassany, A. Kulkarni, and R. R. Newton, "Efficient communication and collection with compact normal forms," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015, pp. 362–374.

# A

# Annotation syntax

Annotations are required in order for gf-ud to know how the GF categories and functions relate to the UD Part of Speech tags and dependency labels. There are two main categories of annotations: Abstract annotations that are associated with a GF abstract syntax and are language independent and Concrete annotations that are associated with a GF concrete syntax and are language specific.

The following appendix is based on the original documentation[1] written by Aarne Ranta and updated to include new syntax.

## A.1 Abstract annotations

These annotations are associated with an abstract syntax and are in general not specific for a concrete language, but in some cases additional abstract annotations may be needed for a specific language.

**Function**: `#fun`

The `#fun` annotation describes GF functions and which UD dependency label each argument of the function should have (see Figure 4.1 for graphical representation of such an annotation). The type signature of the GF function is also required. Each function needs to have at least one head with the label `head` and any number of children which are labeled with their corresponding UD dependency labels.

```
1  #fun GFFunctionName : FirstArgumentCat -> SecondArgumentCat ->
       ReturnCat ; first_ud_label second_ud_label
2  #fun UseN : N -> CN ; head
3  #fun DetCN : Det -> CN -> NP ; det head
```

The function `UseN` only has a single argument (of type `N`), so that is required to be the head, while the function `DetCN` has two arguments, of types `Det` and `CN`, the first of which has the dependency label `det`, while the second is the head.

It is also possible to match the arguments by their inflectional features. The syntax for that is to put the feature and its value in square brackets, attached to a label. Subtypes for the relations, such as `nsubj:pass`, can be pattern matched with a wildcard `*`.

[1] `https://github.com/GrammaticalFramework/gf-ud/blob/master/doc/annotations.md`

```
#fun SingularNP : Det -> CN -> NP ; det head[Number=Sg]
#fun AnySubject : NP -> VP -> Cl ; nsubj:* head
```

**Category**: `#cat`

The `#cat` annotation describes the mapping from GF categories to UD POS tags.

```
#cat GFCategory UD_POS
#cat N NOUN
```

Many categories can have the same POS tag. Only those categories that have lexical items (i.e. zero-place functions linearized to words) need this annotation.

Both function and category annotations must be unique in order for `gf2ud` to work. However, in the `ud2gf` direction, two kinds of deviations may be needed: `#auxcat` and `#auxfun`, which are presented in Section A.2.1.

**Annotation guideline**: `#guidelines`

```
#guidelines UD2
```

This flag is used to indicate what guidelines the annotations are checked against (if any). The default it UD2. It means checking that each label and POS tag is an actually existing UD2 label or tag. An alternative is `none`, meaning that no checks are performed.

## A.2 Concrete annotations

**Syncategorematic words**: `#word`

```
#word is  be Mood=Ind|Number=Sing|Person=3|Tense=Pres|VerbForm=Fin
```

These annotations are used for words that appear syncategorematically, i.e. elsewhere than as linearizations of zero-place functions. The lemma should be defined in a `#lemma` annotation

**Syncategorematic lemmas**: `#lemma`

```
#lemma UseAP,UseAdv,UseNP be Cop cop head
```

This means that when any of the functions `UseAP, UseAdv, UseN` introduces the lemma `be`, it is to be marked with the category `Cop` and the label `cop`. Its head will be the head of the same construction. Notice that the category is typically an auxiliary category (see `#auxcat` below). A shortcut exists for saying that a lemma has the same properties in all functions, except those explicitly defined:

```
#lemma DEFAULT_ be Cop cop head
```

**Morphological tags**: `#morpho`

```
#morpho V,V2 2 Mood=Ind|Tense=Past|VerbForm=Fin
```

This annotations tells how the PGF coordinates (numbers) are converted to morphological tags of UD. The conversion depends on the category, since different categories

have different PGF layouts. To find out what the coordinates mean, one can use `gfud check-annotations`, which returns examples such as

```
morpho mapping missing:
#morpho N 1 -- s Sg Gen A-bomb's
#morpho N 3 -- s Pl Gen A-bombs'
```

This means that all fields except 1 and 3 have been defined for the category N. One can also use the text dump of the PGF invoked by the `print_grammar` command of GF:

```
> import ShallowParse.pgf
> print_grammar
  V := range  [C32 .. C32]
       labels ["s Inf"
               "s PresSg3"
               "s Past"
               "s PastPart"
               "s PresPart"]
```

The numbering of labels starts from 0. Thus the third label, `s Past`, is the one with number 2. Comparing with a UD2 treebank shows that the morphological tag to be used is `Mood=Ind|Tense=Past|VerbForm=Fin`.

**Discontinuous constituents**: `#discont ,head`

```
#discont V2 0-4,head   5,ADV,advmod,head  6,ADP,case,obj
```

This is applied to discontinuous words, such as V2 with its verb particle (field 5) and preposition (field 6). The head fields are those that contain actual verb forms. The other fields need a POS tag, the word's own label, and the label of the word in the construction that becomes the head. In this example, the particle (5) is linked to the head of the construction (i.e. the verb), whereas the preposition (6) is linked to the object.

*NB: the target label functionality is currently not implemented, but all fields are linked to the head.*

**Multiwords**: `#multiword <category> <head-position> <label>`:

```
#multiword Prep head-first fixed
```

This annotation tells that, in a multiword preposition, the first word is the head, and the subsequent words are its dependents with the label `fixed`. This follows the guidelines in https://universaldependencies.org/u/overview/specific-syntax.html; however, existing treebanks don't seem to follow this strictly. The guideline example is *in spite of*, where *in* is the `head` and the other words are `fixed`.

*NB: #multiword works so far only in the gf2ude direction, and head-last is not supported at all yet*

**Change of label**: `#change <label> > <label> <condition>`:

```
#change obj > obl above case
#change det > nmod:poss features Poss=Yes|PronType=Prs
```

These annotations are used at the last step of gf2ud to capture discrepancies between the grammar and the annotation standard that are difficult to define in another way. The `above` condition changes the label in a node that dominates a certain label. In the example shown, the `case` label appears in object that has a preposition, which is a concrete syntax feature. The `features` condition changes the label if the node has certain morphological features. In the example, possessive pronouns are get the label `nmod:poss` instead of the usual `det`.

### A.2.1 Special concrete annotations for ud2gf

The previously shown annotations are used in both directions, gf2ud and ud2gf. The ud2gf direction uses some additional annotations, listed here:

**Alternative function**: `#altfun`

```
#altfun ComplV2 head obl
```

This is needed in `ud2gf` for reading normal UD, because the complement of a V2 verb can be labelled either `obj` of `obl` depending on the case governed by the verb.

**Disabled function**: `#disable`

```
#disable UseAP
```

says that the function `UseAP` is not to be used in `ud2gf`. The reason is that it is overshadowed by a macro function introducing an auxiliary copula.

**Auxiliary category**: `#auxcat <abstract syntax category> <POS tag>`

```
#auxcat Cop AUX
```

The category `Cop` is introduced in the concrete syntax annotation for the syncategorematic copula verb (see below). In the `ud2gf` direction, it is recognized when such a verb, marked by the POS tag `AUX`, is encountered in the dependency tree. This is done by means of auxiliary functions, whose format is somewhat complex, as they also have to define these extra functions in terms of standard ones.

**Auxiliary function**: `#auxfun`

```
1 #auxfun <new abstract syntax function> <argument variables> : <type> = <
      definition> ; <labels-and-features>|
2
3 #auxfun MkVPS_Perf have vp : Have -> VP -> VPS = MkVPS (TTAnt AAnter
      TPres) PPos vp ; aux[Tense=Pres] head
```

The number of argument variables and labels must match the type. The type can be built from both ordinary and auxiliary categories. As the definition shows here, the auxiliary category `Have` argument is ignored. But the important thing is that - its presence in the UD tree is recognized, ensuring that all words are taken into account. - it sets the tense and anteriority of the VPS formed

Auxiliary functions do not necessarily contain auxiliary categories: they can be just macros collecting the applications of many functions.

### Recursive macros

*New with this work:* Auxiliary functions can also be defined in terms of other auxfuns, in which case they will be recursively evaluated. This is useful for transformations that require a larger context than the immediate children of the tree.

The following helper macros can be used for capturing multiple nodes in a pair that can then later be deconstructed somewhere higher up in the tree:

```
1 -- ** Generic, only used inside other macros **
2 -- Pair_a_b = (a -> b -> r) -> r
3 #auxfun MkPair_ a b handler : a -> b -> ab2r -> r = handler a b ; head
      dummy nonexistent
4 #auxfun UsePair_ handler pair : ab2r -> Pair_a_b -> r = pair handler ;
      head dummy
5 -- Triple_a_b_c = (a -> b -> r) -> r
6 #auxfun MkTriple_ a b c handler : a -> b -> c -> abc2r -> r = handler a b
      c ; head dummy nonexistent nope
7 #auxfun UseTriple_ handler triple : abc2r -> Triple_a_b_c -> r = triple
      handler ; head dummy
```

When consuming a pair, you'll typically need a second helper macro (in this example `AP_helper_`), because there are no closures in this simple expansion, everything.

```
1 #auxfun AndCuteCont_ and cute : Conj -> AP -> Pair_Conj_AP = MkPair_ and
      cute ; cc head
2
3 #auxfun AP2_ small andCute : AP -> Pair_Conj_AP -> AP = UsePair_ (
      AP2_helper_ small) andCute ; head conj
4 #auxfun AP2_helper_ small and cute : AP -> Conj -> AP -> AP = ConjAP and (
      BaseAP small cute) ; head dummy nonexistent
```

The helper macros that are only intended for use inside other macros have dummy labels and dummy types, since they are not intended to be matched by the main algorithm.

# B

# Conjunction annotation code

Below is the complete annotation code for implementing conjunctions for adjective phrases using the recursive macros.

```
1  -----------------
2  -- Handling lists
3  -- This has to be repeated for every category
4  #auxcat Comma PUNCT
5
6  -- ** Generic, only used inside other macros **
7  -- Pair_a_b = (a -> b -> r) -> r
8  #auxfun MkPair_ a b handler : a -> b -> ab2r -> r = handler a b ; head
       dummy nonexistent
9  #auxfun UsePair_ handler pair : ab2r -> Pair_a_b -> r = pair handler ;
       head dummy
10 -- Triple_a_b_c = (a -> b -> r) -> r
11 #auxfun MkTriple_ a b c handler : a -> b -> c -> abc2r -> r = handler a b
       c ; head dummy nonexistent nope
12 #auxfun UseTriple_ handler triple : abc2r -> Triple_a_b_c -> r = triple
       handler ; head dummy
13
14 -- ** AP **
15 -- fluffy and cute
16 #auxfun CommaAP_ ap comma : AP -> Conj -> APComma = ap ; head punct[LEMMA
       =\,]
17
18 #auxfun AndCuteCont_ and cute : Conj -> AP -> Pair_Conj_AP = MkPair_ and
       cute ; cc head
19 -- If we had pattern matching, the above function could have looked like
       this
20 -- #auxfun AndCutePatternMatch_ and cute : Conj -> AP -> AP2AP = MkAP2AP
       and cute ; cc head
21
22 #auxfun AP2_ small andCute : AP -> Pair_Conj_AP -> AP = UsePair_ (
       AP2_helper_ small) andCute ; head conj
23 #auxfun AP2_helper_ small and cute : AP -> Conj -> AP -> AP = ConjAP and (
       BaseAP small cute) ; head dummy nonexistent
24 -- If we had pattern matching, the above two functions could have been
       replaced by this
```

```
25 -- #auxfun AP2_ small (MkAP2AP and cute) : AP -> AP2AP -> AP = ConjAP and
       (BaseAP small cute) ; head conj
26
27 #auxfun APBaseComma_ small fluffy andCute : AP -> APComma -> Pair_Conj_AP
       -> ConjListAP = UsePair_ (APBaseComma_helper_ small fluffy) andCute ;
       head conj conj
28 #auxfun APBaseComma_helper_ small fluffy and cute : AP -> APComma -> Conj
       -> AP -> ConjListAP = MkTriple_ and small (BaseAP fluffy cute) ; head
       dummy dummy
29 -- #auxfun APBaseComma_ small fluffy (MkAP2AP and cute) : AP -> APComma ->
        AP2AP -> ConjListAP = ConjConsAP and small (BaseAP fluffy cute) ;
       head conj conj
30
31 #auxfun ConjListToAP2_ and_small_furryFluffyCute : ConjListAP -> AP =
       UseTriple_ ConjListToAP2_helper_ and_small_furryFluffyCute ; head
32 #auxfun ConjListToAP2_helper_ and small furryFluffyCute : Conj -> AP ->
       ListAP -> AP = ConjAP and (ConsAP small furryFluffyCute) ; notreal
       head dummy
33 -- #auxfun ConjListToAP2_ (ConjConsAP and small furryFluffyCute) :
       ConjListAP -> AP = ConjAP and (ConsAP small furryFluffyCute) ; head
34
35 #auxfun APAddComma_ furry and_small_fluffyCute : APComma -> ConjListAP ->
       ConjListAP = UseTriple_ (APAddComma_helper_ furry)
       and_small_fluffyCute ; conj head
36 #auxfun APAddComma_helper_ furry and small fluffyCute : APComma -> Conj ->
        AP -> ListAP -> ConjListAP = MkTriple_ and small (ConsAP furry
       fluffyCute) ; dummy head
37 -- #auxfun APAddComma_ furry (ConjConsAP and small fluffyAndCute) :
       APComma -> ConjListAP -> ConjListAP = ConjConsAP and small (ConsAP
       furry fluffyAndCute) ; conj head
```

# C

# Complete benchmark results

The benchmark was performed on the branch `old-benchmark` at `https://github.com/anka-213/gf-ud/tree/old-benchmark`, using the command:

```
$ stack bench --ba "--csv results.csv"
```

Below is a complete table of the results, sorted by time taken for the original algorithm:

| Name | Original algorithm | Both improvements | Fast keepTrying | Fast allFunsLocal |
|---|---|---|---|---|
| 47: Drop the mic . | 7ms | 5ms | 5ms | 5ms |
| 45: The dress is contemporary . | 12ms | 5ms | 7ms | 6ms |
| 08: Who are they ? | 14ms | 14ms | 14ms | 14ms |
| 37: I spotted a few . | 21ms | 9ms | 11ms | 8ms |
| 70: Investigation and expeditions to the island continue . | 21ms | 14ms | 15ms | 14ms |
| 32: And what about Australia 's position ? | 24ms | 13ms | 17ms | 12ms |
| 17: Then the commercial ends . | 24ms | 10ms | 13ms | 11ms |
| 57: Not all transformations in the region have been successful . | 26ms | 10ms | 16ms | 11ms |
| 130: The chalet burned completely down . | 27ms | 9ms | 12ms | 10ms |
| 106: The reason for advertising the video in Germany is unclear . | 31ms | 11ms | 15ms | 12ms |
| 97: John of Gaunt died in 1399 . | 35ms | 3ms | 9ms | 6ms |
| 129: The ruins were later built over . | 36ms | 10ms | 15ms | 11ms |

| Name | Original algorithm | Both improvements | Fast keepTrying | Fast allFunsLocal |
|---|---|---|---|---|
| 33: Conservationists welcomed the commission 's announcement . | 37ms | 13ms | 17ms | 14ms |
| 55: There is no parade and there never has been . | 37ms | 24ms | 26ms | 24ms |
| 108: The exchange of Barrosos caused a big stir . | 40ms | 11ms | 16ms | 12ms |
| 123: They were primarily on hills . | 40ms | 17ms | 23ms | 18ms |
| 50: Is series two working so far ? | 41ms | 16ms | 20ms | 17ms |
| 74: Aldrin has been married three times . | 42ms | 6ms | 15ms | 8ms |
| 61: Moreover , many of the Macedonian and Persian elite intermarried . | 43ms | 11ms | 21ms | 13ms |
| 75: Two measure the lengths of lunar months . | 45ms | 14ms | 24ms | 16ms |
| 76: Its importance resides in two facts . | 53ms | 17ms | 26ms | 20ms |
| 22: Who can stop this Australia side ? | 55ms | 14ms | 29ms | 14ms |
| 112: France does n't have a good reputation . | 55ms | 7ms | 17ms | 8ms |
| 101: Production of the smartphone model was completely discontinued . | 56ms | 19ms | 24ms | 21ms |
| 12: That 's what keeps us coming back for more . | 57ms | 26ms | 37ms | 27ms |
| 121: There are different theories about the reasons for leaving the place . | 57ms | 22ms | 29ms | 23ms |
| 118: And what about the parties in what , in historical rights ? | 57ms | 24ms | 31ms | 25ms |

| Name | Original algorithm | Both improvements | Fast keepTrying | Fast allFunsLocal |
|---|---|---|---|---|
| 67: The study of volcanoes is called volcanology , sometimes spelled vulcanology . | 60ms | 23ms | 32ms | 23ms |
| 15: She was 84 years old . | 60ms | 14ms | 24ms | 16ms |
| 24: He 's spoken in favour of torture . | 61ms | 17ms | 25ms | 19ms |
| 128: The displaced nuns were moved to the Eibingen cloister . | 61ms | 11ms | 21ms | 12ms |
| 114: It will go for assessment . | 61ms | 13ms | 24ms | 14ms |
| 56: The Yas Marina Circuit website has exact timings . | 63ms | 20ms | 27ms | 21ms |
| 44: They will play on Saturday , 10 June . | 66ms | 10ms | 21ms | 13ms |
| 34: Only 50 were marketplaces . | 69ms | 11ms | 19ms | 14ms |
| 18: His skill in getting answers for taxpayers will be sorely missed . | 69ms | 25ms | 35ms | 27ms |
| 31: Let 's just say he 's wrong . | 70ms | 20ms | 32ms | 23ms |
| 81: This city - state emerged in the same period as Sukhothai . | 71ms | 15ms | 27ms | 18ms |
| 62: Current land reclamation projects include extending the district of Fontvieille . | 72ms | 20ms | 28ms | 22ms |
| 72: Each poem narrates only a part of the war . | 76ms | 19ms | 32ms | 21ms |
| 16: Still , there are questions left unanswered . | 80ms | 19ms | 30ms | 21ms |
| 80: Catherine of Russia was also very satisfied . | 81ms | 11ms | 21ms | 12ms |
| 119: He believes that nobody waiting for us waits for us . | 83ms | 25ms | 36ms | 30ms |

| Name | Original algorithm | Both improvements | Fast keepTrying | Fast allFunsLocal |
|---|---|---|---|---|
| 19: More than 330 crew are onboard the ship . | 85ms | 12ms | 24ms | 16ms |
| 96: Bouchard suffered a shocking three - set loss . | 88ms | 12ms | 26ms | 15ms |
| 39: A small town with two minarets glides by . | 92ms | 19ms | 37ms | 22ms |
| 90: George was appalled by what he saw as their loose morals . | 95ms | 29ms | 44ms | 33ms |
| 111: Are workers allowed to keep religious objects on their desks ? | 95ms | 24ms | 35ms | 24ms |
| 14: The new iron guidelines mean more donors are needed . | 96ms | 16ms | 31ms | 19ms |
| 20: People got killed there . | 102ms | 10ms | 28ms | 11ms |
| 60: The multi-ethnic Achaemenid army possessed many soldiers from the Balkans . | 104ms | 14ms | 31ms | 16ms |
| 105: Do you argue with your alarm clock ? | 110ms | 23ms | 39ms | 25ms |
| 59: The 2019 Winter Universiade will be hosted by Krasnoyarsk . | 124ms | 8ms | 41ms | 12ms |
| 94: At least 330,000 people , including 10,000 technicians , were involved . | 124ms | 14ms | 46ms | 18ms |
| 107: The issue might not be over for Barroso . | 132ms | 6ms | 62ms | 6ms |
| 13: The current waiting period is eight weeks . | 132ms | 15ms | 37ms | 18ms |

| Name | Original algorithm | Both improvements | Fast keepTrying | Fast allFunsLocal |
|---|---|---|---|---|
| 86: He graduated and obtained an M.A. on 21 April 1882 . | 132ms | 15ms | 49ms | 20ms |
| 88: Only 3000 copies were published of the first edition . | 133ms | 20ms | 43ms | 25ms |
| 40: It is his dream to end his career here . | 141ms | 34ms | 78ms | 35ms |
| 49: All the medics were armed , except me . | 142ms | 13ms | 41ms | 13ms |
| 84: With population growth , new indigenous quarters were created . | 143ms | 16ms | 39ms | 18ms |
| 87: He then returned to Kirriemuir . | 148ms | 14ms | 39ms | 16ms |
| 27: In this context , railing against trade makes sense . | 154ms | 14ms | 54ms | 16ms |
| 51: I do n't know why I chose her ... | 162ms | 27ms | 54ms | 30ms |
| 115: Durán acts acts as spokesman and Ángel Pintado as treasurer . | 163ms | 12ms | 44ms | 13ms |
| 92: Louis Post Dispatch called it one of LaBeouf 's best performances . | 180ms | 16ms | 51ms | 17ms |
| 23: They have one crack at redemption , beating England . | 189ms | 19ms | 54ms | 20ms |
| 77: But the impact of Hispania in the newcomers was also big . | 203ms | 17ms | 51ms | 19ms |
| 117: This department now faces new challenges . | 206ms | 17ms | 46ms | 20ms |
| 63: In June to August 2010 famine struck the Sahel . | 210ms | 8ms | 42ms | 13ms |

| Name | Original algorithm | Both improvements | Fast keepTrying | Fast allFunsLocal |
|---|---|---|---|---|
| 43: The consumer can boost the demand for change . | 214ms | 15ms | 54ms | 16ms |
| 120: The festive dedication took place on April 30 , 1955 . | 220ms | 13ms | 69ms | 18ms |
| 73: This was by boat from continental Europe . | 223ms | 14ms | 45ms | 16ms |
| 113: The hit song is " Geronimo " by Sheppard . | 225ms | 11ms | 57ms | 11ms |
| 79: Philip next marched against his southern enemies . | 229ms | 20ms | 50ms | 21ms |
| 25: I also struggle with passwords . | 231ms | 18ms | 59ms | 21ms |
| 109: The good numbers in Asia promptly pushed the stock markets up . | 238ms | 21ms | 57ms | 26ms |
| 82: The Army performed well in combat in Cuba . | 247ms | 16ms | 51ms | 20ms |
| 09: Not everyone can rise above it . | 250ms | 16ms | 63ms | 17ms |
| 99: Von Beust justified the cost increases as lack of detailed planning . | 251ms | 22ms | 58ms | 27ms |
| 66: The Danevirke has remained in German possession ever since . | 261ms | 17ms | 58ms | 21ms |
| 58: She spoke to CNN Style about the experience . | 263ms | 16ms | 56ms | 19ms |
| 21: He worked for the BBC for a decade . | 263ms | 15ms | 53ms | 18ms |
| 125: The hymn was well received and the audience demanded an encore . | 275ms | 22ms | 68ms | 23ms |

| Name | Original algorithm | Both improvements | Fast keepTrying | Fast allFunsLocal |
|---|---|---|---|---|
| 93: The car burst into flames , and Kenseth walked away . | 276ms | 16ms | 64ms | 17ms |
| 41: I also wonder whether the Davis Cup played a part . | 281ms | 23ms | 63ms | 28ms |
| 83: British cavalry troopers also received excellent mounted swordsmanship training . | 292ms | 25ms | 62ms | 30ms |
| 69: It was declared a wildlife sanctuary in 1975 . | 301ms | 18ms | 81ms | 20ms |
| 03: Maybe the dress code was too stuffy . | 307ms | 17ms | 59ms | 20ms |
| 02: $ 5,000 per person , the maximum allowed . | 311ms | 10ms | 172ms | 13ms |
| 68: They generally do not explode catastrophically . | 323ms | 20ms | 70ms | 27ms |
| 05: The scheme makes money through sponsorship and advertising . | 324ms | 18ms | 67ms | 21ms |
| 07: Shenzhen 's traffic police have opted for unconventional penalties before . | 324ms | 20ms | 70ms | 26ms |
| 78: Dominican priest Heinrich Kramer was assistant to the Archbishop of Salzburg . | 352ms | 16ms | 52ms | 19ms |
| 38: Back on the train , we continue southwards . | 375ms | 23ms | 81ms | 25ms |
| 95: These were almost completely forgotten until after Smith 's death . | 407ms | 24ms | 84ms | 26ms |
| 91: Her 1981 album Wild West was one of her biggest sellers . | 416ms | 26ms | 112ms | 30ms |

| Name | Original algorithm | Both improvements | Fast keepTrying | Fast allFunsLocal |
|---|---|---|---|---|
| 36: Day three , I was back on the EMicro . | 434ms | 17ms | 111ms | 19ms |
| 01: The new spending is fueled by Clinton 's large bank account . | 435ms | 15ms | 86ms | 19ms |
| 85: Lenny is a persistent bachelor who has poor luck with women . | 529ms | 24ms | 110ms | 26ms |
| 71: These plant families are still present in Papua New Guinea . | 535ms | 17ms | 117ms | 20ms |
| 126: I declare the first international Olympic games over . | 555ms | 21ms | 116ms | 29ms |
| 29: That 's just legitimately horrendous . | 588ms | 18ms | 97ms | 19ms |
| 89: Her latest nonfiction is about Margaret Douglas , Countess of Lennox . | 599ms | 16ms | 109ms | 19ms |
| 53: North Carolina is ground zero in this election . | 621ms | 12ms | 130ms | 15ms |
| 10: That 's not what we need in our country , folks . | 717ms | 38ms | 163ms | 39ms |
| 122: It contains a monument to Martin Luther King , Jr. | 817ms | 17ms | 148ms | 26ms |
| 124: In addition , its process of gilding copper is technologically noteworthy . | 841ms | 26ms | 173ms | 28ms |
| 54: I was just a boy with muddy shoes . | 1s 26ms | 23ms | 169ms | 26ms |
| 28: Fast forward to 2016 and this is increasingly worthy of attention . | 1s 65ms | 24ms | 232ms | 27ms |
| 102: On the other hand , Vine was art in six seconds . | 1s 245ms | 16ms | 225ms | 21ms |

| Name | Original algorithm | Both improvements | Fast keepTrying | Fast allFunsLocal |
|---|---|---|---|---|
| 46: In theory , if done right , it 's undetectable . | 1s 248ms | 15ms | 283ms | 15ms |
| 65: Like fjords , freshwater lakes are often deep . | 1s 273ms | 18ms | 236ms | 20ms |
| 04: It 's like a super power sometimes . | 1s 377ms | 18ms | 192ms | 25ms |
| 42: Or is it an expensive standard or prepayment tariff ? | 1s 380ms | 23ms | 194ms | 29ms |
| 110: According to the programme , she will speak at 23.45s . | 1s 497ms | 15ms | 321ms | 19ms |
| 100: Simon Krätschmer gropes around alone through the dilapidated , sinister barrack . | 1s 569ms | 23ms | 249ms | 44ms |
| 30: “ I loved the tropical colours , ” he says . | 1s 570ms | 20ms | 327ms | 22ms |
| 127: Today , expansive ruins can be viewed there . | 1s 570ms | 15ms | 327ms | 17ms |
| 98: Kühn can only shake his head . | 1s 806ms | 20ms | 320ms | 26ms |
| 52: This is a homeland security issue of the most existential kind . | 2s 65ms | 36ms | 297ms | 42ms |
| 103: And now he is also world champion . | 2s 621ms | 25ms | 319ms | 29ms |
| 48: The result , then , is hardly the cat 's pyjamas . | 2s 630ms | 22ms | 425ms | 27ms |
| 116: Its management , however , has n't been devoid of criticism . | 2s 835ms | 24ms | 557ms | 27ms |
| 11: Our cellphones are so much more than phones these days . | 3s 703ms | 27ms | 653ms | 29ms |
| 06: Previously the jets had only been seen by bloggers . | 4s 619ms | 33ms | 580ms | 85ms |

| Name | Original algorithm | Both improvements | Fast keepTrying | Fast allFunsLocal |
|---|---|---|---|---|
| 35: I do n't call it a beast lightly . | 5s 824ms | 25ms | 1s 25ms | 30ms |
| 104: It is now only unclear , in which one . | 7s 365ms | 23ms | 1s 116ms | 28ms |
| 26: I can just do that with my life . | 40s 861ms | 44ms | 5s 0ms | 86ms |
| 64: In Danish , the word may even apply to shallow lagoons . | 1m 24s 835ms | 38ms | 10s 471ms | 109ms |

# D

# upto12eng.txt

Below are the conllu codes for the first few sentences from file `upto12eng.txt` which was used for the benchmarks. The examples comes from the English UD treebank with sentences of at most 12 words.

The complete file can be found at: https://github.com/GrammaticalFramework/gf-ud/blob/master/upto12eng.conllu

```
# sent_id = n01002042
# text = The new spending is fueled by Clinton's large bank account.
1ThethetheDETDTDefinite=Def|PronType=Art3det3:det_
2newnewADJJJDegree=Pos3amod3:amod_
3spendingspendingNOUNNNNumber=Sing5nsubj:pass5:nsubj:pass_
4isbeAUXVBZMood=Ind|Number=Sing|Person=3|Tense=Pres|VerbForm=Fin5aux:pass5:aux:pass
5fueledfuelVERBVBNTense=Past|VerbForm=Part0root0:root_
6bybyADPIN_11case11:case_
7ClintonClintonPROPNNNPNumber=Sing11nmod:poss11:nmod:possSpaceAfter=No
8's'sPARTPOS_7case7:case_
9largelargeADJJJDegree=Pos11amod11:amod_
10bankbankNOUNNNNumber=Sing11compound11:compound_
11accountaccountNOUNNNNumber=Sing5obl5:obl:bySpaceAfter=No
12..PUNCT._5punct5:punct_

# newdoc id = n01003
# sent_id = n01003007
# text = $5,000 per person, the maximum allowed.
1$$SYM$_0root0:rootSpaceAfter=No
25,0005,000NUMCDNumType=Card1nummod1:nummod_
3perperADPIN_4case4:case_
4personpersonNOUNNNNumber=Sing1nmod1:nmod:perSpaceAfter=No
5,,PUNCT,_1punct1:punct_
6thethetheDETDTDefinite=Def|PronType=Art7det7:det_
7maximummaximumNOUNNNNumber=Sing1appos1:appos_
8allowedallowVERBVBNTense=Past|VerbForm=Part7acl7:aclSpaceAfter=No
9..PUNCT._1punct1:punct_

# sent_id = n01003013
```

```
# text = Maybe the dress code was too stuffy.
1MaybemaybeADVRB_7advmod7:advmod_
2thetheDETDTDefinite=Def|PronType=Art4det4:det_
3dressdressNOUNNNNumber=Sing4compound4:compound_
4codecodeNOUNNNNumber=Sing7nsubj7:nsubj_
5wasbeAUXVBDMood=Ind|Number=Sing|Person=3|Tense=Past|VerbForm=Fin7cop7:cop_
6tootooADVRB_7advmod7:advmod_
7stuffystuffyADJJJDegree=Pos0root0:rootSpaceAfter=No
8..PUNCT._7punct7:punct_
```

And here is the full list of sentences:

```
The new spending is fueled by Clinton 's large bank account .
$ 5,000 per person , the maximum allowed .
Maybe the dress code was too stuffy .
It 's like a super power sometimes .
The scheme makes money through sponsorship and advertising .
Previously the jets had only been seen by bloggers .
Shenzhen 's traffic police have opted for unconventional penalties before .
Who are they ?
Not everyone can rise above it .
That 's not what we need in our country , folks .
Our cellphones are so much more than phones these days .
That 's what keeps us coming back for more .
The current waiting period is eight weeks .
The new iron guidelines mean more donors are needed .
She was 84 years old .
Still , there are questions left unanswered .
Then the commercial ends .
His skill in getting answers for taxpayers will be sorely missed .
More than 330 crew are onboard the ship .
People got killed there .
He worked for the BBC for a decade .
Who can stop this Australia side ?
They have one crack at redemption , beating England .
He 's spoken in favour of torture .
I also struggle with passwords .
I can just do that with my life .
In this context , railing against trade makes sense .
Fast forward to 2016 and this is increasingly worthy of attention .
That 's just legitimately horrendous .
" I loved the tropical colours , " he says .
Let 's just say he 's wrong .
And what about Australia 's position ?
Conservationists welcomed the commission 's announcement .
Only 50 were marketplaces .
```

I do n’t call it a beast lightly .
Day three , I was back on the EMicro .
I spotted a few .
Back on the train , we continue southwards .
A small town with two minarets glides by .
It is his dream to end his career here .
I also wonder whether the Davis Cup played a part .
Or is it an expensive standard or prepayment tariff ?
The consumer can boost the demand for change .
They will play on Saturday , 10 June .
The dress is contemporary .
In theory , if done right , it ’s un-detectable .
Drop the mic .
The result , then , is hardly the cat 's pyjamas .
All the medics were armed , except me .
Is series two working so far ?
I do n't know why I chose her ...
This is a homeland security issue of the most existential kind .
North Carolina is ground zero in this election .
I was just a boy with muddy shoes .
There is no parade and there never has been .
The Yas Marina Circuit website has exact timings .
Not all transformations in the region have been successful .
She spoke to CNN Style about the experience .
The 2019 Winter Universiade will be hosted by Krasnoyarsk .
The multi-ethnic Achaemenid army possessed many soldiers from the Balkans .
Moreover , many of the Macedonian and Persian elite intermarried .
Current land reclamation projects include extending the district of Fontvieille .
In June to August 2010 famine struck the Sahel .
In Danish , the word may even apply to shallow lagoons .
Like fjords , freshwater lakes are often deep .
The Danevirke has remained in German possession ever since .
The study of volcanoes is called volcanology , sometimes spelled vulcanology .
They generally do not explode catastrophically .
It was declared a wildlife sanctuary in 1975 .
Investigation and expeditions to the island continue .
These plant families are still present in Papua New Guinea .
Each poem narrates only a part of the war .
This was by boat from continental Europe .
Aldrin has been married three times .
Two measure the lengths of lunar months .
Its importance resides in two facts .
But the impact of Hispania in the newcomers was also big .
Dominican priest Heinrich Kramer was assistant to the Archbishop of Salzburg .
Philip next marched against his southern enemies .
Catherine of Russia was also very satisfied .

This city - state emerged in the same period as Sukhothai .
The Army performed well in combat in Cuba .
British cavalry troopers also received excellent mounted swordsmanship training .
With population growth , new indigenous quarters were created .
Lenny is a persistent bachelor who has poor luck with women .
He graduated and obtained an M.A. on 21 April 1882 .
He then returned to Kirriemuir .
Only 3000 copies were published of the first edition .
Her latest non-fiction is about Margaret Douglas , Countess of Lennox .
George was appalled by what he saw as their loose morals .
Her 1981 album Wild West was one of her biggest sellers .
Louis Post Dispatch called it one of LaBeouf 's best performances .
The car burst into flames , and Kenseth walked away .
At least 330,000 people , including 10,000 technicians , were involved .
These were almost completely forgotten until after Smith 's death .
Bouchard suffered a shocking three - set loss .
John of Gaunt died in 1399 .
Kühn can only shake his head .
Von Beust justified the cost increases as lack of detailed planning .
Simon Krätschmer gropes around alone through the dilapidated , sinister barrack .
Production of the smartphone model was completely discontinued .
On the other hand , Vine was art in six seconds .
And now he is also world champion .
It is now only unclear , in which one .
Do you argue with your alarm clock ?
The reason for advertising the video in Germany is unclear .
The issue might not be over for Barroso .
The exchange of Barrosos caused a big stir .
The good numbers in Asia promptly pushed the stock markets up .
According to the programme , she will speak at 23.45 .
Are workers allowed to keep religious objects on their desks ?
France does n't have a good reputation .
The hit song is " Geronimo " by Sheppard .
It will go for assessment .
Durán acts acts as spokesman and Ángel Pintado as treasurer .
Its management , however , has n't been devoid of criticism .
This department now faces new challenges .
And what about the parties in what , in historical rights ?
He believes that nobody waiting for us waits for us .
The festive dedication took place on April 30 , 1955 .
There are different theories about the reasons for leaving the place .
It contains a monument to Martin Luther King , Jr.
They were primarily on hills .
In addition , its process of gilding copper is technologically noteworthy .
The hymn was well received and the audience demanded an encore .
I declare the first international Olympic games over .

Today , expansive ruins can be viewed there .
The displaced nuns were moved to the Eibingen cloister .
The ruins were later built over .
The chalet burned completely down .