COGNITIVE SCIENCE SERIES

LOGIC, LINGUISTICS AND COMPUTER SCIENCE SET

Volume 1

Application of Graph Rewriting to Natural Language Processing

Guillaume Bonfante
Bruno Guillaume and Guy Perrier

ISTE

WILEY

Application of Graph Rewriting to Natural Language Processing

Logic, Linguistics and Computer Science Set

coordinated by
Christian Retoré

Volume 1

Application of Graph Rewriting to Natural Language Processing

Guillaume Bonfante
Bruno Guillaume
Guy Perrier

ISTE

WILEY

First published 2018 in Great Britain and the United States by ISTE Ltd and John Wiley & Sons, Inc.

ISTE Ltd
27-37 St George's Road
London SW19 4EU
UK

www.iste.co.uk

John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030
USA

www.wiley.com

Library of Congress Control Number: 2018935039

British Library Cataloguing-in-Publication Data
A CIP record for this book is available from the British Library
ISBN 978-1-78630-096-6

Contents

Introduction

Our purpose in this book is to show how *graph rewriting* may be used as a tool in *natural language processing*. We shall not propose any new linguistic theories to replace the former ones; instead, our aim is to present graph rewriting as a programming language shared by several existing linguistic models, and show that it may be used to represent their concepts and to transform representations into each other in a simple and pragmatic manner. Our approach is intended to include a degree of universality in the way computations are performed, rather than in terms of the object of computation. Heterogeneity is omnipresent in natural languages, as reflected in the linguistic theories described in this book, and is something which must be taken into account in our computation model.

Graph rewriting presents certain characteristics that, in our opinion, makes it particularly suitable for use in natural language processing.

A first thing to note is that language follows rules, such as those commonly referred to as *grammar rules*, some learned from the earliest years of formal education (for example, “use a singular verb with a singular subject”), others that are implicit and generally considered to be “obvious” for a native speaker (for example in French we say “une voiture rouge (a car red)”, but not “une rouge voiture (a red car)”). Each rule only concerns a small number of the elements in a sentence, directly linked by a *relation* (subject to verb, verb to preposition, complement to noun, etc.). These are said to be *local*. Note that these relations may be applied to words or syntagms at any distance from each other within a phrase: for example, a subject may be separated from its verb by a relative.

Note, however, that in everyday language, notably spoken, it is easy to find occurrences of text which only partially respect established rules, if at all. For practical applications, we therefore need to consider language in a variety of forms, and to develop the ability to manage both rules and their real-world application with potential exceptions.

A second important remark with regard to natural language is that it involves a number of forms of ambiguity. Unlike programming languages, which are designed to be unambiguous and carry precise semantics, natural language includes ambiguities on all levels. These may be lexical, as in the phrase *There's a bat in the attic*, where the bat may be a small nocturnal mammal or an item of sports equipment. They may be syntactic, as in the example "call me a cab": does the speaker wish for a cab to be hailed for them, or for us to say "you're a cab"? A further form of ambiguity is discursive: for example, in an anaphora, "She sings songs", who is "she"?

In everyday usage by human speakers, ambiguities often pass unnoticed, as they are resolved by context or external knowledge. In the case of automatic processing, however, ambiguities are much more problematic. In our opinion, a good processing model should permit programmers to choose whether or not to resolve ambiguities, and at which point to do so; as in the case of constraint programming, all solutions should *a priori* be considered possible. The program, rather than the programmer, should be responsible for managing the coexistence of partial solutions.

The study of language, including the different aspects mentioned above, is the main purpose of linguistics. Our aim in this book is to propose automatic methods for handling formal representations of natural language and for carrying out transformations between different representations. We shall make systematic use of existing linguistic models to describe and justify the representations presented here. Detailed explanations and linguistic justifications for each formalism used will not be given here, but we shall provide a sufficiently precise presentation of each case to enable readers to follow our reasoning with no prior linguistic knowledge. References will be given for further study.

I.1. Levels of analysis

A variety of linguistic theories exist, offering relatively different visions of natural language. One point that all of these theories have in common is the use of multiple, complementary levels of analysis, from the simplest to the most complex: from the phoneme in speech or the letter in writing to the word, sentence, text or discourse. Our aim here is to provide a model which is sufficiently generic to be compatible with these different levels of analysis and with the different linguistic choices encountered in each theory.

Although graph structures may be used to represent different dimensions of linguistic analysis, in this book, we shall focus essentially on syntax and semantics at sentence level. These two dimensions are unavoidable in terms of language processing, and will allow us to illustrate several aspects of graph rewriting. Furthermore, high-quality annotated corpora are available for use in validating our proposed systems, comparing computed data with reference data.

The purpose of syntax is to represent the structure of a sentence. At this level, lexical units – in practice, essentially what we refer to as words – form the basic building-blocks, and we consider the ways in which these blocks are put together to construct a sentence. There is no canonical way of representing these structures and they may be represented in a number of ways, generally falling into one of two types: syntagmatic or dependency-based representations.

The aim of semantic representation is to transmit the meaning of a sentence. In the most basic terms, it serves to convey "who" did "what", "where", "how", etc. Semantic structure does not, therefore, necessarily follow the linear form of a sentence. In particular, two phrases with very different syntax may have the same semantic representation: these are known as paraphrases. In reality, semantic modeling of language is very complex, due to the existence of ambiguities and non-explicit external references. For this reason, many of the formalisms found in published literature focus on a single area of semantics. This focus may relate to a particular domain (for example legal texts) or semantic phenomena (for example dependency minimal recursion semantics (DMRS) considers the scope of quantifiers, whilst abstract meaning representation (AMR) is devoted to highlighting predicates and their arguments).

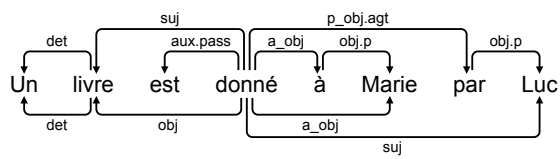
These formalisms all feature more or less explicit elements of formal logic. For a simple transitive sentence, such as *Max hates Luke*, the two proper nouns are interpreted as constants, and the verb is interpreted as a predicate, $Hate$, for which the arguments are the two constants. Logical quantifiers may be used to account for certain determiners. The phrase *"a man enters."* may thus be represented by the first-order logical formula $\exists x(Man(x) \wedge Enter(x))$.

In what follows, we shall discuss a number of visions of syntax and semantics in greater detail, based on published formalisms and on examples drawn from corpora, which reflect current linguistic usage.

There are significant differences between syntactic and semantic structures, and the interface between the two levels is hard to model. Many linguistic models (including Mel'čuk and Chomsky) feature an intermediary level between syntax, as described above, and semantics. This additional level is often referred to as *deep syntax* .

To distinguish between syntax, as presented above, and deep syntax, the first is often referred to as surface syntax or *surface structure*.

These aspects will be discussed in greater detail later. For now, note simply that deep structure represents the highest common denominator between different semantic representation formalisms. To avoid favoring any specific semantic formalism, deep structure uses the same labels as surface structure to describe new relations. For this reason, it may still be referred to as "syntax". Deep structure may, for example, be used to identify new links between a predicate and one of its semantic arguments, which cannot be seen from the surface, to neutralize changes in verb voice (diathesis) or to identify grammatical words, which do not feature in a semantic representation. Deep structure thus ignores certain details that are not relevant in terms of semantics. The following figure is an illustration of a passive voice, with the surface structure shown above and the deep structure shown below, for the French sentence "Un livre est donné à Marie par Luc" (A book is given to Mary by Luc).



I.2. Trees or graphs?

The notion of trees has come to be used as the underlying mathematical structure for syntax, following Chomsky and the idea of syntagmatic structures. The tree representation is a natural result of the recursive process by which a component is described from its direct subcomponents. In dependency representations, as introduced by Tesnière, linguistic information is expressed as binary relations between atomic lexical units. These units may be considered as nodes, and the binary relations as arcs between the nodes, thus forming a graph. In a slightly less direct manner, dependencies are also governed by a syntagmatic vision of syntax, naturally leading to the exclusion of all dependency structures, which do not follow a tree pattern. In practice, in most corpora and tools, dependency relations are organized in such a way that one word in a sentence is considered as the root of the structure, with each other node as the target of one, and only one, relation. The structure is then a tree.

This book is intended to promote a systematic and unified usage of graph representations. Trees are considered to facilitate processing and to simplify analytical algorithms. However, the grounds for this argument are not particularly solid, and, as we shall see through a number of experiments, the processing cost of graphs, in practice, is acceptable. Furthermore, the tools presented in what follows have been designed to permit use with a tree representation at no extra cost.

While the exclusive use of tree structures may seem permissible in the field of syntactic structures, it is much more problematic on other levels, notably for semantic structures. A single entity may play a role for different predicates at the same time, and thus becomes the target of a relation for each of these roles. At the very least, this results in the creation of acylic graphs; in practice, it means that a graph is almost always produced. The existing formalisms for semantics, which we have chosen to present below (AMR and DMRS), thus make full use of graph structures.

Even at syntactic level, trees are not sufficient. If we wish to enrich a structure with deep syntax information (such as the subjects of infinitives, or the antecedents of relative pronouns), we obtain a structure involving cycles, justifying the use of a graph. Graphs also allow us to simultaneously account for several linguistic levels in a uniform manner (for example syntactic

structure and the linear order of words). Note that, in practice, tree-based formalisms often include ad hoc mechanisms, such as coindexing, to represent relations, which lie outside of the tree structure. Graphs allow us to treat these mechanisms in a uniform manner.

I.3. Linguistically annotated corpora

Whilst the introspective work carried out by lexicographers and linguists is often essential for the creation of dictionaries and grammars (inventories of rules) via the study of linguistic constructs, their usage and their limitations, it is not always sufficient. Large-scale corpora may be used as a means of considering other aspects of linguistics. In linguistic terms, corpus-based research enables us to observe the usage frequency of certain constructions and to study variations in language in accordance with a variety of parameters: geographic, historical or in terms of the type of text in question (literature, journalism, technical text, etc.). As we have seen, language use does not always obey those rules described by linguists. Even if a construction or usage found in a corpus is considered to be incorrect, it must be taken into account in the context of applications.

Linguistic approaches based on artificial intelligence and, more generally, on probabilities, use observational corpora for their learning phase. These corpora are also used as references for tool validation.

Raw corpora (collections of text) may be used to carry out a number of tasks, described above. However, for many applications, and for more complex linguistic research tasks, this raw text is not sufficient, and additional linguistic information is required; in this case, we use annotated corpora. The creation of these corpora is a tedious and time-consuming process. We intend to address this issue in this book, notably by proposing tools both for preparing (pre-annotating) corpora and for maintaining and correcting existing corpora. One solution often used to create annotated resources according to precise linguistic choices is to transform pre-existing resources, in the most automatic way possible. Most of the corpora used in the universal dependencies (UD) project[1] are corpora which had already been annotated in

1 http://universaldependencies.org

the context of other projects, converted into UD format. We shall consider this type of application in greater detail later.

I.4. Graph rewriting

Our purpose here is to show how graph rewriting may be used as a model for natural language processing. The principle at the heart of rewriting is to break down transformations into a series of elementary transformations, which are easier to describe and to control. More specifically, rewriting consists of executing rules, i.e. (1) using patterns to describe the local application conditions of an elementary transformation and (2) using local commands to describe the transformation of the graph.

One of the ideas behind this theory is that transformations are described based on a linguistic analysis that, as we have seen, is highly suited to local analysis approaches. Additionally, rewriting is not dependent on the formalism used, and can successfully manage several coexisting linguistic levels. Typically, it may be applied to composite graphs, made up of heterogeneous links (for example those which are both syntactic and semantic). Furthermore, rewriting does not impose the order, nor the location, in which rules are applied. In practice, this means that programmers no longer need to consider algorithm design and planning, freeing them to focus on the linguistic aspects of the problem in question. A fourth point to note is that the computation model is intrinsically non-deterministic; two "contradictory" rules may be applied to the same location in the same graph. This phenomenon occurs in cases of linguistic ambiguity (whether lexical, syntactic or semantic) where two options are available (in the phrase *he sees the girl with the telescope*, who has the *telescope*?), each corresponding to a rule. Based on a strategy, the programmer may choose to continue processing using both possibilities, or to prefer one option over the other.

We shall discuss the graph rewriting formalism used in detail later, but for now, we shall simply outline its main characteristics. Following standard usage in rewriting, the "left part" of the rule describes the conditions of application, while the "right part" describes the effect of the rule on the host structure.

The left part of a rule, known as the *pattern*, is described by a graph (which will be searched for in the host graph for modification) and by a set of negative constraints, which allow for better control of the context in which rules are

applied. The left part can also include rule parameters in the form of external lexical information. Graph pattern recognition is an NP-complete problem and, as such, is potentially difficult for practical applications; however, this is not an issue in this specific case, as the patterns are small (rarely more than five nodes) and the searches are carried out in graphs of a few dozen (or, at most, a few hundred) nodes. Moreover, patterns often present a tree structure, in which case searches are extremely efficient.

The right part of rules includes atomic commands (edge creation, edge deletion) that describe transformations applied to the graph at local level. There are also more global commands (shift) that allow us to manage connections between an identified pattern and the rest of the graph. There are limitations in terms of the creation of new nodes: commands exist for this purpose, but new nodes have a specific status. Most systems work without creating new nodes, a fact which may be exploited in improving the efficiency of rewriting.

Global transformations may involve a large number of intermediary steps, described by a large number of rules (several hundred in the examples presented later). We therefore need to control the way in which rules are applied during transformations. To do this, the set of rules for a system is organized in a modular fashion, featuring packages, for grouping coherent sub-sets of rules, and strategies, which describe the order and way of applying rules.

The notion of graph rewriting raises mathematical definition issues, notably in describing the way in which local transformations interact with the context of the pattern of the rule. One approach is based on category theory and has two main variants, SPO (*Single Pushout*) and DPO (*Double Pushout*) [ROZ 97]. Another approach uses logic [COU 12], drawing on the decidability of monadic second-order logic. These approaches are not suitable for our purposes. To the best of our knowledge, the graphs in question do not have an underlying algebraic structure or the limiting parameters (such as tree width) necessary for a logical approach. Furthermore, we need to use `shift` type commands, which are not compatible with current approaches to category theory. Readers may wish to consider the theoretical aspect underpinning the form of rewriting used here independently.

Here, we shall provide a more operational presentation of rewriting and rules, focusing on language suitable for natural language processing. We have identified a number of key elements to bear in mind in relation to this subject:

– negative conditions are essential to avoid over-interpretation;

– modules/packages are also necessary, as without them, the process of designing rewriting systems becomes inextricable;

– we need a strong link to lexicons, otherwise thousands of rules may come into play, making rewriting difficult to design and ineffective;

– a notion of strategy is required for the sequential organization of modules and the resolution of ambiguities.

The work presented in this book was carried out using GREW, a generic graph rewriting tool that responds to the requirements listed above. We used this tool to create systems of rules for each of the applications described later in the book. Other tools can be found in the literature, along with a few descriptions of graph rewriting used in the context of language processing (e.g. [HYV 84, BOH 01, CRO 05, JIJ 07, BÉD 09, CHA 10]). However, to the best of our knowledge, widely-used generic graph rewriting systems, with the capacity to operate on several levels of language description, are few and far between (the Ogre system is a notable exception [RIB 12]). A system of this type will be proposed here, with a description of a wide range of possible applications of this approach for language processing.

I.5. Practical issues

Whilst natural language may be manifested both orally and in writing, speech poses a number of specific problems (such as signal processing, disfluence and phonetic ambiguity), which will not be discussed here; for simplicity's sake, we have chosen to focus on written language.

As mentioned, we worked on both the syntactic and semantic levels. The language used in validating and applying our approach to large bodies of real data was French. Figure I.1 shows the different linguistic levels considered in the examples presented in this book (horizontal boxes), along with one or more existing linguistic formats.

Our aim here is to study ways of programming conversions between formats. These transformations may take place within a linguistic level (shown by the horizontal arrows on the diagram) and permit automatic conversion of data between different linguistic descriptions on that level. They may also operate between levels (descending arrows in the diagram), acting as automatic syntactic or semantic analysis tools[2]. These different transformations will be discussed in detail later.

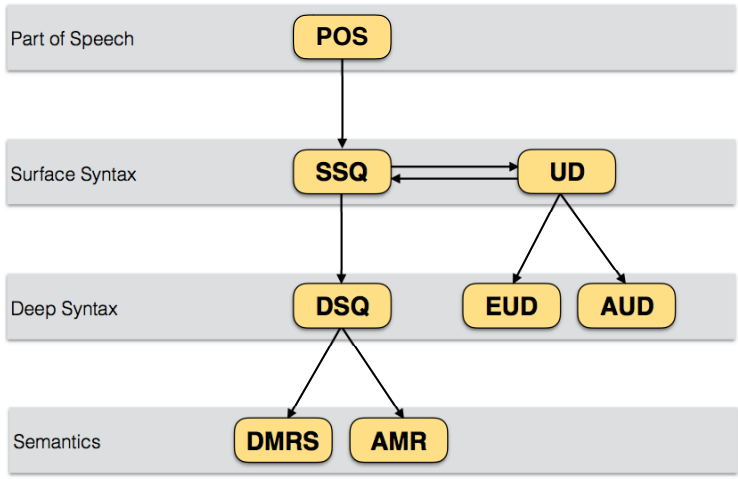


Figure I.1. *Formats and rewriting systems considered in this book*

Our tools and methods have been tested using two freely available corpora, annotated using dependency syntax and made up of text in French. The first corpus is SEQUOIA[3], made up of 3099 sentences from a variety of domains: the press (the *annodis_er* subcorpus), texts issued by the European parliament (the *Europar.550* sub-corpus), medical notices (the *emea-fr-dev* and *emea-fr-test* subcorpora), and French Wikipedia (the *frwiki_50.1000* sub-corpus). It was originally annotated using constituents, following the French Treebank annotation scheme (FTB) [ABE 04]. It was then converted

2 We have yet to attempt transformations in the opposite direction (upward arrows); this would be useful for text generation.

3 https://deep-sequoia.inria.fr

automatically into a surface dependency form [CAN 12b], with long-distance dependencies corrected by hand [CAN 12a]. Finally, SEQUOIA was annotated in deep dependency form [CAN 14]. Although the FTB annotation scheme used here predates SEQUOIA by a number of years, we shall refer to it as the SEQUOIA format here, as we have only used it in conjunction with the SEQUOIA corpus.

The second corpus used here is part of the Universal Dependencies project[4] (UD). The aim of the UD project is to define a common annotation scheme for as many languages as possible, and to coordinate the creation of a set of corpora for these languages. This is no easy task, as the annotation schemes used in existing corpora tend to be language specific. The general annotation guide for UD specifies a certain number of choices that corpus developers must follow and complete for their particular language. In practice, this general guide is not yet set in stone and is still subject to discussion. The UD_FRENCH corpus is one of the French-language corpora in UD. It is made up of around 16000 phrases drawn from different types of texts (blog posts, news articles, consumer reviews and Wikipedia). It was annotated within the context of the Google DataSet project [MCD 13] with purely manual data validation. The annotations were then converted automatically for integration into the UD project (UD version 1.0, January 2015). Five new versions have since been issued, most recently version 2.0 (March 2017). Each version has come with new verifications, corrections and enrichments, many thanks to the use of the tools presented in this book. However, the current corpus has yet to be subject to systematic manual validation.

I.6. Plan of the book

Chapter 1 of this book provides a practical presentation of the notions used throughout. Readers may wish to familiarize themselves with graph handling in PYTHON and with the use of GREW to express rewriting rules and the graph transformations, which will be discussed later. The following four chapters alternate between linguistic presentations, describing the levels of analysis in question and examples of application. Chapter 2 is devoted to syntax (distinguishing between surface syntax and deep structure), while Chapter 4

4 http://universaldependencies.org

focuses on the issue of semantic representation (via two proposed semantic formalization frameworks, AMR and DMRS). Each of these chapters is followed by an example of application to graph rewriting systems, working with the linguistic frameworks in question. Thus, Chapter 3 concerns the application of rewriting to transforming syntactic annotations, and Chapter 5 covers the use of rewriting in computing semantic representations. In Chapter 6, we shall return to syntax, specifically syntactic analysis through graph rewriting; although the aim in this case is complementary to that found in Chapter 3, the system in question is more complex, and we thus thought it best to devote a separate chapter to the subject. The last two chapters constitute a review of the notions presented previously, including rigorous mathematical definitions, in Chapter 7, designed for use in studying the properties of the calculation model presented in Chapter 8, notably with regard to termination and confluence. Most chapters also include exercises and sections devoted to "good practice". We hope that these elements will be of use to the reader in gaining a fuller understanding of the notions and tools in question, enabling them to be used for a wide variety of purposes.

The work presented here is the fruit of several years of collaborative work by the three authors. It would be hard to specify precisely which author is responsible for contributions, the three played complementary roles. Guillaume Bonfante provided the basis for the mathematical elements, notably the contents of the final two chapters. Bruno Guillaume is the developer behind the GREW tool, while Guy Perrier developed most of the rewriting systems described in the book, and contributed to the chapters describing these systems, along with the linguistic aspects of the book. The authors wish to thank Mathieu Morey for his participation in the early stages of work on this subject [MOR 11], alongside Marie Candito and Djamé Seddah, with whom they worked [CAN 14, CAN 17].

This book includes elements contained in a number of existing publications: [BON 10, BON 11a, BON 11b, PER 12, GUI 12, BON 13a, BON 13b, CAN 14, GUI 15b, GUI 15a, CAN 17]. All of the tools and resources presented in this book are freely available for download at http://grew.fr. All of the graphs used to illustrate the examples in this book can be found at the following link: www.iste.co.uk/bonfante/language.zip.

1

Programming with Graphs

In this chapter, we shall discuss elements of programming for graphs. Our work is based on PYTHON, a language widely used for natural language processing, as in the case of the NLTK library[1] (Natural Language ToolKit), used in our work. However, the elements presented here can easily be translated into another language. Several different data structures may be used to manage graphs. We chose to use dictionaries; this structure is elementary (i.e. unencapsulated), reasonably efficient and extensible. For what follows, we recommend opening an interactive PYTHON session[2].

Notes for advanced programmers: by choosing such a primitive structure, we do not have the option to use sophisticated error management mechanisms. There is no domain (or type) verification, no identifier verification, etc. Generally speaking, we shall restrict ourselves to the bare minimum in this area for reasons of time and space. Furthermore, we have chosen not to use encapsulation so that the structure remains as transparent as possible. Defining a class for graphs should make it easier to implement major projects. Readers are encouraged to take a more rigorous approach to that presented here after reading the book. Finally, note that the algorithms used here are not always optimal; once again, our primary aim is to improve readability.

1 http://www.nltk.org

2 Our presentation is in PYTHON3, but PYTHON2 can be used almost as-is.

Application of Graph Rewriting to Natural Language Processing, First Edition.
Guillaume Bonfante, Bruno Guillaume and Guy Perrier.

Notes for "beginner" programmers: this book is not intended as an introduction to PYTHON*, and we presume that readers have some knowledge of the language, specifically with regard to the use of lists, dictionaries and sets.*

The question will be approached from a mathematical perspective in Chapter 7, but for now, we shall simply make use of an intuitive definition of graphs. A graph is a set of nodes connected by labeled edges. The nodes are also labeled (with a phonological form, a feature structure, a logical predicate, etc.). The examples of graphs used in this chapter are dependency structures, which simply connect words in a sentence using syntactic functions. The nodes in these graphs are words (`W1`, `W2`, ..., `W5` in the example below), the edges are links (suj, obj, det) and the labels on the nodes provide the phonology associated with each node. We shall consider the linguistic aspects of dependency structures in Chapter 2.

Note that it is important to distinguish between nodes and their labels. This enables us to differentiate between the two occurrences of *"the"* in the graph above, corresponding to the two nodes `W1` and `W4`.

In what follows, the nodes in the figures will not be named for ease of reading, but they can be found in the code in the form of strings : `'W1'`, `'W2'`, etc.

1.1. Creating a graph

A graph is represented using a dictionary. Let us start with a graph with no nodes or edges.

```
g = dict()
```

The nodes are dictionary keys. The value corresponding to key `v` is a pair `(a, sucs)` made up of a label `a` and of the list `sucs` of labeled edges starting from `v`. Let us add a node `'W1'` labeled *"the"* to `g`:

```
g['W1'] = ('the', [])
```

Now, add a second and a third node, with the edges that connect them:

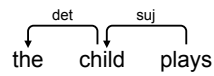
```
g['W2'] = ('child', [])
g['W3'] = ('plays', [])
g['W3'][1].append(('suj', 'W2'))
g['W2'][1].append(('det', 'W1'))
```

and print the result:

```
g
```

```
{'W1': ('the', []), 'W2': ('child', [('det', 'W1')]), 'W3':
('plays', [('suj', 'W2')])}
```

The last box shows the output from the PYTHON interpreter. This graph is represented in the following form:



Let us return to the list of successors of a node. This is given in the form of a list of pairs `(e, t)`, indicating the label `e` of the edge and the identifier `t` of the target node. In our example, the list of successors of node `'W2'` is given by `g['W2'][1]`. It contains a single pair `('det', 'W1')` indicating that the node `'W1'` corresponding to *"the"* is the determiner of `'W2'`, i.e. the common noun *"child"*.

In practice, it is easier to use construction functions:

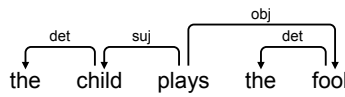
```
def add_node(g, u, a):
  #Add a node u labeled a in graph g
  g[u] = (a, [])

def add_edge(g, u, e, v):
  # Add an edge labeled e from u to v in graph g
  if (e, v) not in g[u][1]:
    g[u][1].append( (e, v) )
```

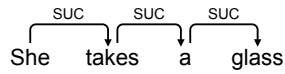
This may be used as follows:

```
add_node(g, 'W4', 'the')
add_node(g, 'W5', 'fool')
add_edge(g, 'W3', 'obj', 'W5')
add_edge(g, 'W5', 'det', 'W4')
```

to construct the dependency structure of the sentence *"the child plays the fool"*:



Let us end with the segmentation of a sentence into words. This is represented as a *flat* graph, connecting words in their order; we add an edge, `'SUC'`, between each word and its successor. Thus, for the sentence *"She takes a glass"*, we obtain:



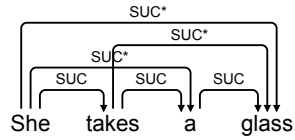
The following solution integrates the NLTK segmenter:

```
import nltk
word_list = nltk.word_tokenize("She takes a glass")
word_graph = dict()
for i in range(len(word_list)):
    add_node(word_graph, 'W%s' %  i, word_list[i])
for i in range(len(word_list) - 1):
    add_edge(word_graph, 'W%s' % i, 'SUC', 'W%s' % (i + 1))
word_graph
```

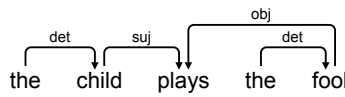
```
{'W3': ('glass', []), 'W1': ('takes', [('SUC', 'W2')]),
'W2': ('a', [('SUC', 'W3')]), 'W0': ('She', [('SUC',
'W1')])}
```

Readers may wish to practice using the two exercises as follows.

EXERCISE 1.1.– *Finish constructing the following flat graph so that there is a* `'SUC*'` *edge between each word and one of its distant successors. For example, the chain* "She takes a glass" *will be transformed as follows:*



EXERCISE 1.2.– *Write a function to compute a graph in which all of the edges have been reversed. For example, we go from the dependency structure of* "the child plays the fool" *to:*



1.2. Feature structures

So far, node labels have been limited to their phonological form, i.e. a string of characters. Richer forms of structure, namely feature structures, may be required. Once again, we shall use a dictionary:

```
fs_plays = {'phon' : 'plays', 'cat' : 'V'}
```

The `fs_plays` dictionary designates a feature structure with two features, `'phon'` and `'cat'`, with respective values `'plays'` and `'V'`. To find the category `'cat'` of the feature structure `fs_plays`, we apply:

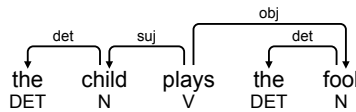
```
fs_plays['cat']
```

```
V
```

Let us reconstruct our initial sentence, taking account of feature structures:

```
g = dict()
add_node(g, 'W1', {'phon' : 'the', 'cat' : 'DET'} )
add_node(g, 'W2', {'phon' : 'child', 'cat' : 'N'} )
add_node(g, 'W3', {'phon' : 'plays', 'cat' : 'V'} )
add_node(g, 'W4', {'phon' : 'the', 'cat' : 'DET'})
add_node(g, 'W5', {'phon' : 'fool', 'cat' : 'N'})
add_edge(g, 'W2', 'det', 'W1')
add_edge(g, 'W3', 'suj', 'W2')
add_edge(g, 'W3', 'obj', 'W5')
add_edge(g, 'W5', 'det', 'W4')
```

The corresponding graph representation[3] is:



The "Part Of Speech" (POS) labeling found in NLTK[4] may also be used to construct a richer graph. The following solution shows how this labeling may be integrated in the form of a dictionary.

```
import nltk
word_list = nltk.word_tokenize("She takes a glass")
tag_list = nltk.pos_tag(word_list)
feat_list = [{'phon':n[0], 'cat':n[1]} for n in tag_list]
t_graph = {'W%s' % i : (feat_list[i], [])
           for i in range(len(tag_list))}
for i in range(len(tag_list)-1):
    add_edge(t_graph, 'W%s' % i, 'SUC', 'W%s' % (i+1))
t_graph
```

```
{'W3': ({'phon': 'glass', 'cat': 'NN'}, []), 'W1': ({'phon'
    :
'takes', 'cat': 'VBZ'}, [('SUC', 'W2')]), 'W2': ({'phon':
'a', 'cat': 'DT'}, [('SUC', 'W3')]), 'W0': ({'phon': 'She',
'cat': 'PRP'}, [('SUC', 'W1')])}
```

1.3. Information searches

To find the label or feature structure of a node, we use:

```
g['W4'][0]
```

```
{'phon': 'the', 'cat': 'DET'}
```

or the function:

```
def get_label(g, u):
    return g[u][0]
```

3 The feature names `phon` and `cat` are not shown, as these are always present in applications. Other features are noted, for example `num=sing`.

4 The PennTreeBank tagset is used by default in NLTK.