Language as Circuits

Jonathon Liu

Wolfson College

University of Oxford

Supervised by Bob Coecke

A thesis submitted for the degree of

MSc Mathematics and Foundations of Computer Science

2021

Acknowledgements

What a strange, yet strangely wonderful year it has been. Firstly, I want to thank Bob for giving me a fascinating new perspective on mathematics and science, as well as for bringing me into the idiosyncratic fold of Oxford quantum. I am very glad I picked this project - it has been a blast.

To Vincent, my second supervisor - thanks for all the discussions, all the insights, all the food, and all the hanging out (working hard, or hardly working?). Of course, much of the work and ideas in this thesis are joint with you.

Thanks to all the people associated with the Quantum Group and/or CQC with whom I had interesting and productive chats - including but not limited to Richie Yeung, Tim Hosgood, Stephen Clark, and Tiffany Duneau. Thanks to those who proofread parts of my thesis - Max Doré and Emily Jennings, and thanks to those who gave me encouragement in the final days.

Thanks to my coursemates on the MFoCS - I wish you all the best and hope to see you around. Thanks to all the wonderful people I met in this raucous town of dreaming spires - it really is the people that make this place so special. You made this year an incredible and unforgettable one in spite of everything.

Finally, eternal gratitude to my parents - everything good that I have done is yours.

Jonathon Liu
October 2021

Abstract

Since its inception roughly a decade ago, there has been a flurry of research surrounding the *compositional-distributional* approach to natural language processing. This approach is so named because it combines distributional models of meaning with compositional theories of grammar and semantics. More concretely, it is based around the *composition* of meaning vectors of individual words for the purpose of obtaining the meaning vector of larger constituents of text. Such an approach to NLP is regarded as a *quantum* one - the underlying theory of string diagrams that it heavily relies on was originally formulated for describing processes in quantum information. As such, in view of the current progress in quantum computation, this approach is potentially on the cusp of being realized in large scale experiments.

This thesis focuses on one significant new theoretical development in this area - the idea of *language circuits*. It is an ambitious extension of earlier frameworks, and aims to provide a way to represent the meanings of entire texts. In this thesis, which consists entirely of theoretical work, we will first provide an extensive coverage of the background theory behind compositional-distributional NLP. In the second half, we launch into an exploration of language circuits - much of this exploration will come in the form of a discussion of ideas. As more concrete contributions, we will propose two different processes that allow us to take as input some suitable sentence, and produce as output the language circuit representation of that sentence. One is based on dependency grammar parsing, and the other is based on providing 'internal wirings'.

Contents

Chapter 1

Introduction

Natural language is, broadly speaking, any language that was the mother tongue of a group of humans [Tra14]. Natural language processing (NLP) is a field in linguistics, computer science, and artificial intelligence concerned with the interaction between computers and natural language. The ultimate aim of NLP is to make computers understand language as well as humans do - whatever that may entail. NLP is the driving force behind technologies like speech recognition, sentiment analysis, virtual assistants, machine translation, and so on. While this field has seen impressive progress in recent years (in particular benefiting from the deep learning revolution), there are still obvious limitations to modern NLP, with one of the overarching issues being a heavy reliance on opaque, structure-less statistical machine learning models. Indeed we are also a long way off from the ultimate aim of achieving human-level understanding of language.

This thesis focuses on a diagrammatic, 'compositional-distributional' (often abbreviated to 'DisCo') approach to NLP. This is a theoretical framework that, at an applied level, seeks to provide an alternative to the currently dominant paradigm of black box statistical models, and at a theoretical level, hopes to yield new insights into language itself. In addition, it is an approach that appears to be particularly amenable to implementation on quantum hardware[CdFMT20]. DisCo-style NLP has its origin in the 2010 paper [CSC10], which proposed a framework (later named 'DisCoCat') that allows one to obtain the distributional meaning of a sentence by composing the distributional meanings of its words. Much work has been done on this and related ideas in the years following. Most recently, a new evolution of this framework was proposed (the eponymous 'language circuits'). This new idea gives a way to 'compose sentences' in order to obtain the distributional meaning of entire texts.

1.1 Structure of the thesis

This thesis aims to provide a thorough exploration of the notion of 'language circuit'.

The first three chapters following the introduction are mainly expository background material. In chapter 2 we present the category-theoretic foundations of our diagrammatic formalism, which serves to make our work mathematically rigorous. The chapter focuses specifically on monoidal categories and the various useful structures one can introduce to them - braidings, cups, caps, spiders. In chapter 3, we present background material on grammar/syntax as it is studied in linguistics. Grammar provides a structural understanding of language, which will be used to inform how we model language. Chapter 4 provides a broad overview of the theoretical work thus far in the diagrammatic, DisCo approach to modelling language. This chapter brings together ideas from the previous two chapters. In particular, we will give a description of DisCoCat.

Chapter 5 contains a discussion on how to diagrammatically model a certain basic fragment of English ('simple sentences') as language circuits. This essentially amounts to deciding how to represent different syntactic categories as circuit components. Many of the design choices here were already made in previous work (e.g. [CW]), but we attempt to motivate and justify these choices, by drawing on established grammatical theories. In this chapter we also propose a simple type theory to categorise the circuit components.

In chapter 6, we propose a concrete 'algorithm' that outputs language circuits corresponding to input text from a certain fragment of English. This algorithm relies on parsing information from a dependency grammar parser. The fragment of English that is dealt with largely consists of the 'simple sentence fragment' discussed in the previous chapter, along with some extra ingredients. At the diagrammatic level, the extra ingredients do not yield new circuit components, but rather they end up corresponding to new 'assembly rules' for the existing simple sentence circuit components.

Finally, in chapter 7, we follow an alternate approach to obtaining language circuits proposed in [CW]. This approach consists of providing a large catalogue of 'internal wirings', which can be plugged into certain DisCoCat diagrams in order to turn them into language circuits. While [CW] provided a catalogue of internal wirings that work with a 'pregroup grammar'-based DisCoCat, in this chapter we do so for a DisCoCat based on a 'combinatory categorial grammar'. Some of the work in [CW] is able to be translated over, but much of it does not.

Chapter 2

Categorical background

In this chapter, we present some requisite mathematical background material on category theory. This background serves to put the string-diagrammatic calculus we use later on solid ground.

In particular, this chapter is centred on monoidal categories, which are one of the main tools of the modern field known as *applied category theory*[FS19]. We begin by introducing monoidal categories, and then present specialised versions of these - braided, symmetric, closed, and finally compact monoidal categories. In the last section we discuss Frobenius algebras, known more colloquially as 'spiders', which are useful gadgets that live in monoidal categories.

In our exposition, we will assume some basic category theoretic knowledge - i.e. the kind of material covered in a first course on category theory (basic examples of categories, commutative diagrams, functors, natural transformations, adjoints). Much of this assumed background can be found in Appendix A.

2.1 Monoidal categories

Monoidal categories, and specialised versions thereof, form the bread and butter of applied category theory. They also come with an elegant and rigorous graphical calculus that can be used for proofs. This section follows closely [HV19].

Definition 2.1.1. *A **monoidal category** is a category C equipped with the following data:*

- *a **tensor product** functor $\otimes : C \times C \to C$*
- *a **unit object** $I \in \mathrm{Ob}(C)$*
- *an **associator** natural isomorphism $(X \otimes Y) \otimes Z \xrightarrow{\alpha_{X,Y,Z}} X \otimes (Y \otimes Z)$*

- *a **left unitor** natural isomorphism* $I \otimes X \xrightarrow{\lambda_X} X$
- *a **right unitor** natural isomorphism* $X \otimes I \xrightarrow{\rho_X} X$.

This data must satisfy the following triangle and pentagon equations for all objects X, Y, Z, W

$$\begin{tikzcd}
(X \otimes I) \otimes Y \arrow[rr, "\alpha_{X,I,Y}"] \arrow[dr, "\rho_X \otimes 1_Y"'] & & X \otimes (I \otimes Y) \\
& X \otimes Y \arrow[ur, "1_X \otimes \lambda_Y"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
(X \otimes (Y \otimes Z)) \otimes W \arrow[rr, "\alpha_{X,Y\otimes Z,W}"] & & X \otimes ((Y \otimes Z) \otimes W) \arrow[d, "1_X \otimes \alpha_{Y,Z,W}"] \\
((X \otimes Y) \otimes Z) \otimes W \arrow[u, "\alpha_{X,Y,Z} \otimes 1_W"] \arrow[dr, "\alpha_{X\otimes Y,Z,W}"'] & & X \otimes (Y \otimes (Z \otimes W)) \\
& (X \otimes Y) \otimes (Z \otimes W) \arrow[ur, "\alpha_{X,Y,Z\otimes W}"'] &
\end{tikzcd}$$

The associator and left and right unitors are often referred to as *structural isomorphisms*.

Example. Set is a monoidal category where

- the categorical tensor product is the Cartesian product of sets ×,
- the unit object is a chosen singleton set $\{\bullet\}$,
- the associators $(X \times Y) \times Z \xrightarrow{\alpha_{X,Y,Z}} X \times (Y \times Z)$ are the functions $((x, y), z) \mapsto (x, (y, z))$,
- the left (resp. right) unitor $\{\bullet\} \times X \xrightarrow{\lambda_X} X$ (resp. $X \times \{\bullet\} \xrightarrow{\rho_X} X$) are the functions $(\bullet, x) \mapsto x$ (resp. $(x, \bullet) \mapsto x$).

Example. Vect$_k$ (vector spaces over the field k) is a monoidal category where

- the categorical tensor product $\otimes : \mathbf{Vect}_k \times \mathbf{Vect}_k \to \mathbf{Vect}_k$ is the usual tensor product of vector spaces,
- the unit object is the 1-dimensional vector space given by the underlying field k,
- the associators $(X \otimes Y) \otimes Z \xrightarrow{\alpha_{X,Y,Z}} X \otimes (Y \otimes Z)$ are the unique linear maps satisfying $(x \otimes y) \otimes z \mapsto x \otimes (y \otimes z)$ for all $x \in X$, $y \in Y$, $z \in Z$,
- the left (resp. right) unitor $k \otimes X \xrightarrow{\lambda_X} X$ (resp. $X \otimes k \xrightarrow{\rho_X} X$) is the unique linear map satisfying $1 \otimes x \mapsto x$ (resp. $x \otimes 1 \mapsto 1$) for all $x \in X$.

By restriction, $\mathbf{FVect}_k$ (finite dimensional k-vector spaces) also forms a monoidal category with these structures.

Example. Note that given a category C, the monoidal structure may not be unique. For instance, we can also make $\mathbf{Vect}_k$ into a monoidal category by taking the categorical tensor product to be the direct sum of vector spaces $\oplus$[1]. However, in this thesis we will only use $\mathbf{Vect}_k$ (and $\mathbf{FVect}_k$) to refer to the monoidal category where the categorical tensor product is the vector space tensor product.

Definition 2.1.2. *A monoidal category is* ***strict monoidal*** *if all the structural isomorphisms are all identity natural transformations, i.e.*

$$\begin{aligned} (X \otimes Y) \otimes Z &= X \otimes (Y \otimes Z) \\ I \otimes X &= X \\ X \otimes I &= X \end{aligned}$$

for all X, Y, Z.

We make a remark here on the so-called *principle of equivalence* in category theory[2]. Stating that two objects of a given category are *equal*, as we do in definition 2.1.2, often leads to a 'breaking of equivalence invariance' and can be problematic. Generally, one should instead say that objects are isomorphic and then (usually) impose some coherence relation on the relevant family of isomorphisms. However, in some later sections (chapter 4, when we construct DisCoCat as a functor from some syntax category to $\mathbf{FVect}_{\mathbb{R}}$) we will somewhat contravene this rule and assert equality of objects, by enforcing choices of various objects.

2.1.1 Monoidal functors, strictification, coherence

Here we introduce the notion of monoidal functors, which are functors that respect the monoidal structure, along with some important results about monoidal categories.

Monoidal functors come in varying levels of strictness and strength, but the kind we are interested in is the strong monoidal functor.

Definition 2.1.3. *A* ***strong monoidal functor*** *$F : C \to C'$ between monoidal categories is a*

[1] Note the direct sum also gives the products (and coproducts) for $\mathbf{Vect}_k$, so this construction would be a 'Cartesian monoidal category', i.e. a monoidal category where the monoidal structure is given by the categorical product.

[2] See `https://ncatlab.org/nlab/show/principle+of+equivalence`.

functor equipped with natural isomorphisms

$$F_0 : I' \cong F(I)$$
$$(F_2)_{A,B} : F(A) \otimes' F(B) \cong F(A \otimes B)$$

such that the following diagrams commute:

- *Associativity:*

$$\begin{tikzcd}
(F(A) \otimes' F(B)) \otimes' F(C) \arrow[r, "\alpha'_{F(A),F(B),F(C)}"] \arrow[d, "(F_2)_{A,B} \otimes' 1_{F(C)}"] & F(A) \otimes' (F(B) \otimes' F(C)) \arrow[d, "1_{F(A)} \otimes' (F_2)_{B,C}"'] \\
F(A \otimes B) \otimes' F(C) \arrow[d, "(F_2)_{A \otimes B,C}"] & F(A) \otimes' F(B \otimes C) \arrow[d, "(F_2)_{A,B \otimes C}"'] \\
F((A \otimes B) \otimes C) \arrow[r, "F(\alpha_{A,B,C})"'] & F(A \otimes (B \otimes C))
\end{tikzcd}$$

- *Unitality:*

$$\begin{tikzcd}
F(A) \otimes' I' \arrow[r, "\rho'_{F(A)}"] \arrow[d, "1_{F(A)} \otimes' F_0"] & F(A) \arrow[d, "F(\rho_A^{-1})"] \\
F(A) \otimes' F(I) \arrow[r, "(F_2)_{A,I}"'] & F(A \otimes I)
\end{tikzcd}
\qquad
\begin{tikzcd}
I' \otimes' F(A) \arrow[r, "\lambda'_{F(A)}"] \arrow[d, "F_0 \otimes' 1_{F(A)}"] & F(A) \arrow[d, "F(\lambda_A^{-1})"] \\
F(I) \otimes' F(A) \arrow[r, "(F_2)_{I,A}"'] & F(I \otimes A)
\end{tikzcd}$$

With monoidal functors, we can then state the following useful theorem.

Theorem 2.1.4 (Strictification theorem, theorem 1.38 [HV19]). *Every monoidal category is monoidally equivalent to a strict monoidal category.*

Without going into the precise details of what a 'monoidal equivalence' is, it suffices to say that monoidal equivalences between two monoidal categories witness that the two monoidal structures are 'essentially the same'. Thus, the strictification theorem essentially justifies the usual laxity with which we treat associativity and unit laws in monoidal categories. That is, we will often write $A \otimes B \otimes C \otimes \ldots$ without regard to parentheses and such, analogously to how we freely write $A \times B \times C \times \ldots$ for Cartesian products of sets.

Another important result about monoidal categories worth noting is the so-called 'coherence theorem', due to Mac Lane[3].

[3] See `https://ncatlab.org/nlab/show/coherence+theorem+for+monoidal+categories`.

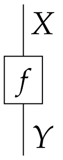
Theorem 2.1.5 (Coherence theorem for monoidal categories)**.** *Given the data of a monoidal category, if the pentagon and triangle equations hold, then any well-typed equation built from associators, unitors, and their inverses holds.*

That is, the triangle and pentagon equations together imply that any possible 'reorganization' of two systems via the structural isomorphisms are equal.

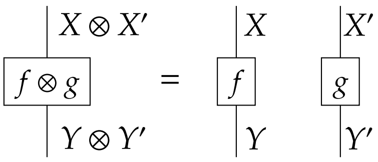
2.1.2 Graphical calculus

If we think of a morphism $f : X \to Y$ as some kind of abstract process taking us from state X to state Y, then the tensor product $\otimes$ can be interpreted as allowing multiple processes to occur concurrently. This notion is captured in a graphical calculus for monoidal categories, where we represent morphisms/processes as boxes with input and output wires corresponding to their domain and codomain. These graphical depictions of morphisms are called **string diagrams**, and are the Poincaré dual of the familiar commutative diagrams[4].

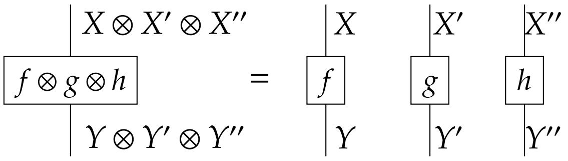
Putting inputs at the top and outputs at the bottom, a morphism $f : X \to Y$ would then be depicted as the following string diagram.



The tensor product of $f : X \to Y$ with $g : X' \to Y'$ is then depicted by simply placing two morphisms next to each other.



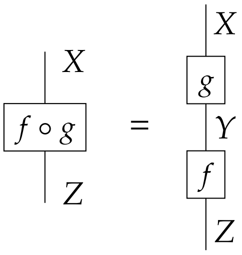
Since the graphical calculus is usually for a strict, or strictified version of some monoidal category (as per theorem 2.1.4), we make no distinction between $(X \otimes Y) \otimes Z$ and $X \otimes (Y \otimes Z)$, and additional tensor products are unambiguously depicted as the following[5].



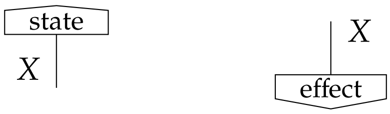
[4] See `https://ncatlab.org/nlab/show/string+diagram`.

[5] There are alternate diagrammatic calculi for not necessarily strict monoidal categories, in which one does keep track of the bracketing, as well as of I objects. We will not consider these alternate systems in this thesis.

Identity morphisms are represented by wires with no boxes. Sequential composition $\circ$ is represented by vertically composing the boxes.



The monoidal unit I is usually thought of as a trivial or empty system. Since the graphical calculus is usually set in a strict monoidal category, the wire for the unit object I is simply not depicted. Hence the identity morphism on I is depicted as an 'empty diagram'. There is also a special graphical depiction for **states** (morphisms that go $I \to X$), and **effects** (morphisms that go $X \to I$).



We will be particularly interested in states - for our applications, a state on object X can be rightly viewed as 'some element of X'.

Example. In **Set**, states of a set X are functions $\{\bullet\} \to X$, which correspond exactly to elements of X by considering the image of $\bullet$.

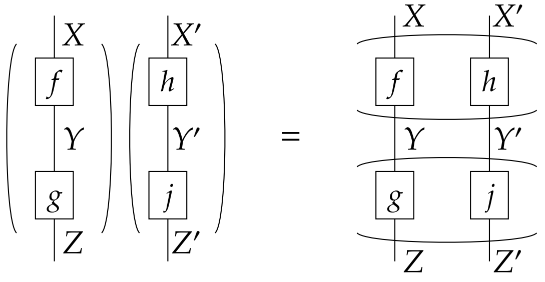
Example. Similarly, in $\mathbf{Vect}_k$, states of X are linear functions $k \to X$, which correspond exactly to vectors in X by considering the image of $1 \in k$.

Many operations in monoidal categories that look unenlightening in symbols become obvious in the diagrammatic calculus. For instance, an important property of monoidal categories is that they satisfy the 'bifunctoriality' or 'interchange law': for any morphisms $f : X \to Y$, $g : Y \to Z$, $h : X' \to Y'$, $j : Y' \to Z'$, we have

$$(g \circ f) \otimes (j \circ h) = (g \otimes j) \circ (f \otimes h). \tag{2.1}$$

Note this is just one of the two functoriality conditions of $\otimes$[6]. In symbols, this law looks like a somewhat mysterious algebraic identity - but in graphical notation, the interchange law becomes

[6] The other functoriality condition is $1_X \otimes 1_Y = 1_{X \otimes Y}$.



where we used brackets to indicate how we are forming the equations on each side. Dropping the brackets, the two sides will look identical, and we see that in the graphical calculus, the interchange law is intuitively natural to the point of being trivial. This illustrates a general theme - in the graphical calculus many algebraic features are are absorbed into the geometry of the plane, of which humans have an excellent intuitive understanding.

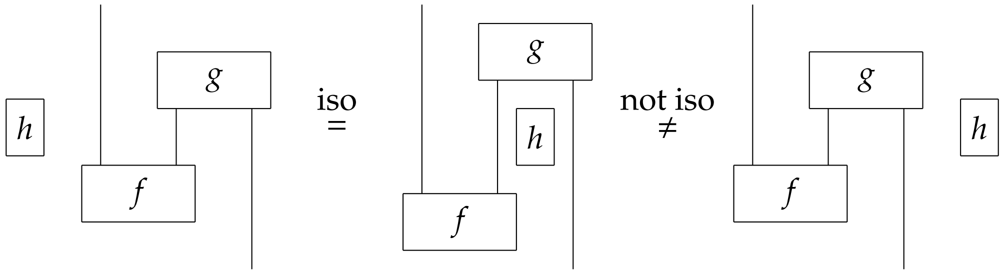
As we will see later, there are many additional structures one can introduce to the basic monoidal category which can be represented by encoding further geometric properties into the string diagram calculus.

For the basic monoidal category without any such additions, the validity of the grammatical calculus is captured in the following 'correctness theorem', which essentially says that the graphical calculus is both sound and complete.

Theorem 2.1.6 (Correctness of the graphical calculus for monoidal categories, theorem 1.8 [HV19])**.** *A well-typed equation between morphisms in a monoidal category follows from the axioms if and only if it holds in the graphical language up to planar isotopy.*

Here two diagrams are considered 'planar isotopic' if one can be deformed continuously into another, within some rectangular region of the plane in which the input and output wires terminate at the top and bottom of the rectangle, and no intersections are introduced into any of the components.

Below are some example of isotopic and non-isotopic diagrams.



2.2 Braided and symmetric monoidal categories

In this section, we introduce some extra structure to the basic monoidal category, namely a 'braiding'.

Definition 2.2.1. *A **braided monoidal category** is a monoidal category equipped with a natural isomorphism*

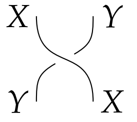
$$X \otimes Y \xrightarrow{\sigma_{X,Y}} Y \otimes X$$

*called a **braiding**, which satisfies the hexagon equations:*

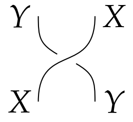
$$\begin{tikzcd}
 & X \otimes (Y \otimes Z) \arrow[r, "\sigma_{X,Y\otimes Z}"] \arrow[dl, "\alpha^{-1}_{X,Y,Z}"'] & (Y \otimes Z) \otimes X & \\
(X \otimes Y) \otimes Z \arrow[dr, "\sigma_{X,Y}\otimes 1_Z"'] & & & Y \otimes (Z \otimes X) \arrow[ul, "\alpha^{-1}_{Y,Z,X}"'] \\
 & (Y \otimes X) \otimes Z \arrow[r, "\alpha_{Y,X,Z}"'] & Y \otimes (X \otimes Z) \arrow[ur, "1_Y\otimes\sigma_{X,Z}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & (X \otimes Y) \otimes Z \arrow[r, "\sigma_{X\otimes Y,Z}"] \arrow[dl, "\alpha_{X,Y,Z}"'] & Z \otimes (X \otimes Y) & \\
X \otimes (Y \otimes Z) \arrow[dr, "1_X\otimes\sigma_{Y,Z}"'] & & & (Z \otimes X) \otimes Y \arrow[ul, "\alpha_{Z,X,Y}"'] \\
 & X \otimes (Z \otimes Y) \arrow[r, "\alpha^{-1}_{X,Z,Y}"'] & (X \otimes Z) \otimes Y \arrow[ur, "\sigma_{X,Z}\otimes 1_Y"'] &
\end{tikzcd}$$

The braiding morphisms are depicted in the graphical notation as the following

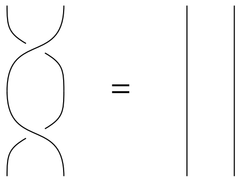


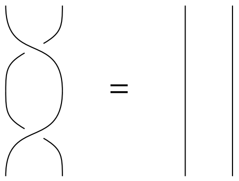
$\sigma_{X,Y} : X \otimes Y \to Y \otimes X$



$\sigma^{-1}_{X,Y} : Y \otimes X \to X \otimes Y$

so that invertibility takes the following intuitive graphical form, mirroring the behaviour of strings.





Braided monoidal categories have their own graphical calculus, expressed via the following correctness theorem. The notion of isotopy used in the calculus is now three-dimensional ('spatial isotopy'), as opposed to the two-dimensional planar isotopy for basic monoidal cateogires. That is, the diagrams for a braided monoidal category are assumed to lie in a cube, with input wires terminating at the lower face and output wires terminating at the upper face. These diagrams are then allowed to be deformed in three-dimensional space.

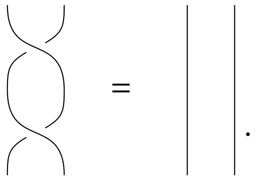
Theorem 2.2.2 (Correctness of graphical calculus for braided monoidal categories, theorem 1.18 [HV19])**.** *A well-typed equation between morphisms in a braided monoidal category follows from the axioms if and only if it holds in the graphical language up to spatial isotopy.*

An important special class of braided monoidal categories is the symmetric monoidal category.

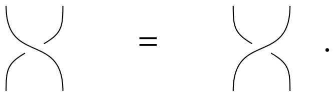
Definition 2.2.3. *A* ***symmetric monoidal category*** *is a braided monoidal category in which the braiding satisfies the symmetry property*

$$\sigma_{Y,X} \circ \sigma_{X,Y} = 1_{X \otimes Y}.$$

In this case we call σ the 'symmetry'. Graphically, the symmetry property corresponds to

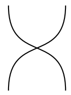


Indeed, it is easy to show that in a symmetric monoidal category, $\sigma_{X,Y} = \sigma^{-1}_{Y,X}$



(2.2)

So in symmetric monoidal categories there is no distinction made between under- and over-crossings, and we can simply draw



for a single type of crossing.

Suppose our diagrams now depict curves in four-dimensional space. Then we can smoothly deform one crossing into the other, in the manner of equation (2.2), by using the extra dimension. In this sense, symmetric monoidal categories have a four-dimensional graphical calculus.

Theorem 2.2.4 (Correctness of the graphical calculus for symmetric monoidal categories, theorem 1.22 [HV19])**.** *A well-typed equation between morphisms in a symmetric monoidal category follows from the axioms if and only if it holds in the graphical language up to graphical equivalence.*

Example. **Set** has a braiding $X \times Y \xrightarrow{\sigma_{X,Y}} Y \times X$ given by the function $(x, y) \mapsto (y, x)$. Clearly this braiding is also a symmetry, so that **Set** is a symmetric monoidal category.

Example. $\mathbf{Vect}_k$ (and also $\mathbf{FVect}_k$) has a braiding, where $X \otimes Y \xrightarrow{\sigma_{X,Y}} Y \otimes X$ is the unique linear map extending $x \otimes y \mapsto y \otimes x$ for all vectors $x \in X$, $y \in Y$. Clearly this braiding is a symmetry, so that $\mathbf{Vect}_k$ (and $\mathbf{FVect}_k$) is a symmetric monoidal category.

2.3 Closed monoidal categories

In this section we introduce the notion of closure. The basic idea is that a category is closed when for any pair of objects X, Y, the collection of morphisms from X to Y can be regarded as forming an object of C itself. In this case, the object is often denoted $[X, Y]$ and referred to as the **internal hom-object** or simply the **internal hom**. Note that this differs from the hom-set $\mathrm{Hom}_C(X, Y)$ which in general is an object of **Set** rather than of C. The term 'closed' in 'closed monoidal category' is hence used in the sense that forming hom-sets does not lead 'out of the category'.

There are various flavours of closed categories - there are definitions of closed category that assume no extra structure (in particular no monoidal structure)[7], and there are special varieties like Cartesian closed categories. For a general monoidal category we have the following formal definitions.

Definition 2.3.1. *A monoidal category C is **right-closed** if for all objects B, the functor $- \otimes B : C \to C$ has a right adjoint*

$$- \otimes B \dashv - \multimapinv B.$$

Using the hom-set definition of adjoint (definition A.3.1), this means that for any object B there is a bijection

$$\Lambda^R_{X,B,Y} : \mathrm{Hom}_C(X \otimes B, Y) \cong \mathrm{Hom}_C(X, Y \multimapinv B) \tag{2.3}$$

natural in X, Y. By a result on 'adjunctions with a parameter' (theorem A.3.3), we may assemble the functors $- \multimapinv B : C \to C$ into a bifunctor $- \multimapinv - : C \times C^{op} \to C$ such that the bijection (2.3) is also natural in B[8]. The operation given by the $\Lambda^R_{X,B,Y}$ map, which consists of taking some morphism $X \otimes B \to Y$ and producing a morphism $X \to Y \multimapinv B$, is called **right-currying**.

[7] See `https://ncatlab.org/nlab/show/closed+category`.

[8] See also `https://ncatlab.org/nlab/show/internal+hom#properties`.

We have an analogous definition for left-closure.

Definition 2.3.2. *A monoidal category C is* ***left-closed*** *if for all objects B, the functor $B \otimes - : C \to C$ has a right adjoint*

$$B \otimes - \dashv B \multimap -.$$

Analogously to the right-closed case, there is a bijection

$$\Lambda^{L}_{B,X,Y} : \mathrm{Hom}_{C}(B \otimes X, Y) \cong \mathrm{Hom}_{C}(X, B \multimap Y)$$

natural in X, Y, and B, where we have extended the $B \multimap - : C \to C$ functors into a bifunctor $- \multimap - : C^{op} \times C \to C$. The operation given by $\Lambda^{L}_{B,X,Y}$ is called **left-currying**.

The point here is that for a general monoidal category, there is not necessarily a way to relate tensoring on the left and tensoring on the right, and hence we will in general have a separate notion of left- and right-closure.

Definition 2.3.3. *If a monoidal category is both left- and right-closed, we call it* ***biclosed****.*

In a *braided* monoidal category, the braiding makes $A \otimes B$ naturally isomorphic to $B \otimes A$, so the distinction between tensoring on the left and tensoring on the right becomes immaterial. Every right-closed braided monoidal category becomes left-closed in a canonical way, and vice versa[9]. That is, a braided monoidal category will be left-closed if and only if it is right-closed. Thus, we may safely speak of a 'braided monoidal closed category', or 'symmetric monoidal closed category' without specifying whether it is left- or right-closed.

In the special case of symmetric monoidal category which we will be interested in, we have the following definition for closure that subsumes the previous ones[10].

Definition 2.3.4. *For a symmetric monoidal category C, an* ***internal hom-functor*** *in C is a functor*

$$[-,-] : C^{op} \times C \to C$$

such that for every object X of C, we have a pair of adjoint functors $- \otimes X \dashv [X,-]$ (both $C \to C$).

If an internal hom-functor exists, we say C is a ***closed symmetric monoidal category****.*

As previously mentioned, the object $[X,Y]$ is usually called the internal hom of X and Y. We have a bijection

$$\mathrm{Hom}_{C}(X \otimes Y, Z) \cong \mathrm{Hom}_{C}(X, [Y,Z]) \tag{2.4}$$

[9] More concretely, note that for any object B, the braiding can be restricted to give a natural isomorphism $- \otimes B \cong B \otimes -$. Then, if a category is right-closed, i.e. $- \otimes B \dashv - \multimapinv B$, by a result on the uniqueness of adjoints (lemma A.3.2), this also gives $B \otimes - \dashv - \multimapinv B$, which means the category is also left-closed.

[10] From `https://ncatlab.org/nlab/show/internal+hom`.

natural in all arguments.

Note that definition 2.3.4 reduces correctly with respect to the previous definitions - a 'closed symmetric monoidal category' in the sense of definition 2.3.4 is precisely a left-/right-/bi-closed monoidal category which is also symmetric[11].

Example. The tautological example of a closed monoidal category is **Set**. Recalling that the Cartesian product makes **Set** a symmetric monoidal category, the internal hom-functor in **Set** is just the usual hom-functor $\mathrm{Hom}_{\mathbf{Set}}(-,-) : \mathbf{Set}^{op} \times \mathbf{Set} \to \mathbf{Set}$, with the natural bijection

$$\mathrm{Hom}_{\mathbf{Set}}(X \times Y, Z) \cong \mathrm{Hom}_{\mathbf{Set}}(X, \mathrm{Hom}_{\mathbf{Set}}(Y, Z))$$

provided by the usual currying of functions. That is, the forward operation here is the following: given an $f : X \times Y \to Z$, we can 'delay the inputs' to obtain a function $X \to \mathrm{Hom}_{\mathbf{Set}}(Y, Z)$ that maps $x \mapsto (f(x, -) : Y \to Z)$. The inverse operation is referred to as 'uncurrying'. It takes as input a function $\hat{f} : X \to \mathrm{Hom}_{\mathbf{Set}}(Y, Z)$ that maps $x \mapsto (f_x : Y \to Z)$ - essentially a family of functions $Y \to Z$ indexed by elements $x \in X$. Uncurrying means we assemble these into a single function $X \times Y \to Z$ that maps $(x, y) \mapsto f_x(y)$. Verifying the naturality of this bijection is a routine exercise.

Example. Recall that given two vector spaces V and W over the same field k, the space of k-linear maps from V to W, denoted $L(V, W)$, is itself a vector space over k. $\mathbf{Vect}_k$ thus also forms a closed symmetric monoidal category, where the internal hom-functor is $L(-,-) : \mathbf{Vect}_k^{op} \times \mathbf{Vect}_k \to \mathbf{Vect}_k$. Note this is distinct from the hom-functor $\mathrm{Hom}_{\mathbf{Vect}_k}(-,-) : \mathbf{Vect}_k^{op} \times \mathbf{Vect}_k \to \mathbf{Set}$. By restriction, $\mathbf{FVect}_k$ is also a closed symmetric monoidal category.

2.4 Compact closed categories

We are now able to introduce compact closed categories. Compactness is about the notion of dual objects, which leads to caps and cups in the graphical calculus.

Definition 2.4.1 (Def 3.1 [HV19]). *In a monoidal category, an object X is* ***left dual*** *to an*

[11] In the forward direction, a symmetric monoidal category with internal hom $[-,-] : C^{op} \times C \to C$ will be right-closed in the sense of definition 2.3.1, by taking $- \multimapinv B = [B, -]$. Therefore it is additionally left-closed (hence biclosed) by the previous discussion, where we may also take $B \multimap - = [B, -]$. In the other direction, if C is closed and symmetric, then the $- \multimapinv B$ functors from right-closure say can then be assembled into the $[-,-]$ internal hom-functor via theorem A.3.3.

object X^r, and X^r is ***right dual*** *to X, written $X \dashv X^r$, if there are morphisms*

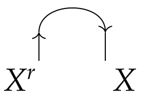
$$\eta_X : I \to X^r \otimes X, \qquad \epsilon_X : X \otimes X^r \to I$$

called the ***unit*** *and* ***counit*** *that satisfy the equations*

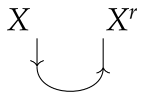
$$\begin{tikzcd}
& X \otimes I \arrow[r, "1_X \otimes \eta_X"] & X \otimes (X^r \otimes X) \arrow[dd, "\alpha^{-1}_{X,X^r,X}"] \\
X \arrow[ur, "\rho_X^{-1}"] & & \\
& I \otimes X \arrow[ul, "\lambda_X"] & (X \otimes X^r) \otimes X \arrow[l, "\epsilon_X \otimes 1_X"]
\end{tikzcd}$$

$$\begin{tikzcd}
& I \otimes X^r \arrow[r, "\eta_X \otimes 1_{X^r}"] & (X^r \otimes X) \otimes X^r \arrow[dd, "\alpha_{X^r,X,X^r}"] \\
X^r \arrow[ur, "\lambda_{X^r}^{-1}"] & & \\
& X^r \otimes I \arrow[ul, "\rho_{X^r}"] & X^r \otimes (X \otimes X^r) \arrow[l, "1_{X^r} \otimes \epsilon_X"]
\end{tikzcd}$$

We will generally use a superscript r to denote the right dual (e.g. $X \dashv X^r$), and a superscript l to denote a left dual (e.g. $X^l \dashv X$). If we have $X \dashv X^r$, we say that the object X is right-dualisable, and X^r is left-dualisable. Graphically, we depict the chosen unit and counit morphisms as 'caps' and 'cups' ('chosen', since cups and caps are not necessarily unique).

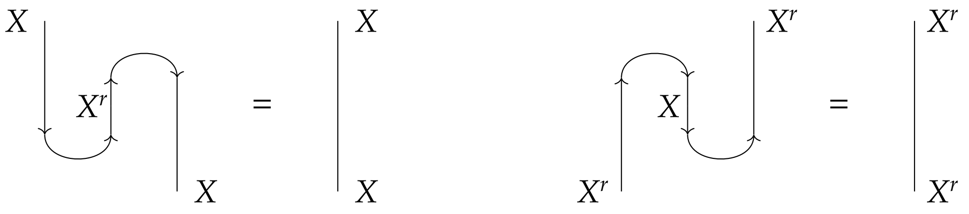


$\eta_X : I \to X^r \otimes X$



$\epsilon_X : X \otimes X^r \to I$

The arrows on the wires capture the duality information of the types, and in effect add orientation to the wires in the graphical calculus. The duality equations then take on a particularly nice form.



Because of this graphical form, these equations are sometimes called 'snake equations' or 'yanking equations'.

A useful result about duals is that they are unique up to isomorphism.

Lemma 2.4.2 (Lem 3.4 [HV19]). *In a monoidal category with $X \dashv X^r$, then $X \dashv R$ if and only if $X^r \cong R$. Similarly, $L \dashv X^r$ if and only if $X \cong L$.*

For a neat graphical proof, refer to the citation.

Lemma 2.4.3 (Lem 3.7 [HV19]). *In a monoidal category, $X \dashv X^r$ and $Y \dashv Y^r$ implies $X \otimes Y \dashv Y^r \otimes X^r$.*

There is also a neat graphical proof for this (see the citation). Using lemma 2.4.2, this result is equivalently expressed as $(A \otimes B)^r \cong B^r \otimes A^r$, and similarly $(A \otimes B)^l \cong B^l \otimes A^l$.

Another property worth noting is that strong monoidal functors preserve duals.

Lemma 2.4.4. *Let $F : \mathbf{C} \to \mathbf{C}'$ be a strong monoidal functor. If $X \dashv X^r$ in $\mathbf{C}$, then $F(X) \dashv F(X^r)$ in $\mathbf{C}'$.*

Put another way, we have $F(A^l) \cong F(A)^l$, and $F(A^r) \cong F(A)^r$. To see this, note that we have the following two compositions of morphisms

$$I' \xrightarrow{F_0} F(I) \xrightarrow{F(\eta_X)} F(X^r \otimes X) \xrightarrow{(F_2)^{-1}_{X^r,X}} F(X^r) \otimes' F(X)$$
$$F(X) \otimes' F(X^r) \xrightarrow{(F_2)_{X,X^r}} F(X \otimes X^r) \xrightarrow{F(\epsilon_X)} F(I) \xrightarrow{F_0^{-1}} I'$$

which we claim are the unit $\eta_{F(X)}$ and counit $\epsilon_{F(X)}$ witnessing the adjunction $F(X) \dashv F(X^r)$[KSPC13].

Monoidal categories with dual objects are called 'rigid monoidal categories'.

Definition 2.4.5. *If every object of a monoidal category $\mathbf{C}$ has a left dual, we say $\mathbf{C}$ is **left-rigid** monoidal category. Analogously, if every object has a right dual we say $\mathbf{C}$ is **right-rigid**. If every object has both a left and right dual, we simply say that $\mathbf{C}$ is a **rigid** monoidal category*[12].

A right-rigid monoidal category (i.e. not necessarily braided) will be left-closed, where we can take the internal hom-functor $B \multimap -$ to be $B^r \otimes -$.

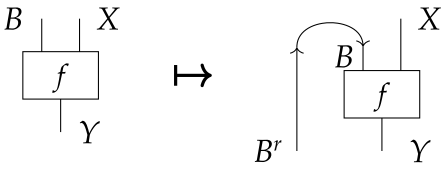
Lemma 2.4.6. *Right-rigid monoidal categories are left-closed, via*

$$B \otimes - \dashv B^r \otimes -$$

for any object B.

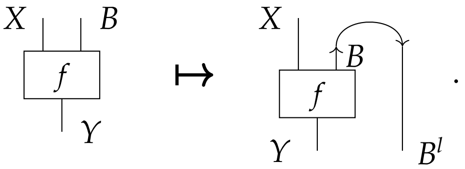
[12] These are sometimes also referred to as *autonomous* monoidal categories.

The easiest way to prove that this forms an adjunction is to exhibit an isomorphism $\mathrm{Hom}_C(B \otimes X, Y) \cong \mathrm{Hom}_C(X, B^r \otimes Y)$ natural in X, Y (left-currying). This is given by bending one of the input wires (i.e. attaching a cap):



This is clearly an isomorphism, since the inverse operation of attaching a cup will undo it. Naturality is also easy to prove with diagrams.

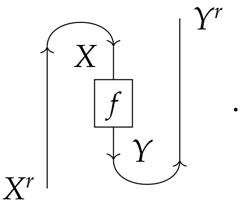
Similarly, a left-rigid monoidal category will be right-closed, since $- \otimes B \dashv - \otimes B^l$, via right-currying. That is, we have $\mathrm{Hom}_C(X \otimes B, Y) \cong \mathrm{Hom}_C(X, Y \otimes B^l)$ natural in X, Y via



So a rigid monoidal category will be biclosed, possessing both left- and right-currying operations.

It turns out that having dual objects is a strong structure that extends functorially to morphisms.

Definition 2.4.7. *In a monoidal category, for a morphism $f : X \to Y$ and chosen dualities $X \dashv X^r$, $Y \dashv Y^r$, the* ***right dual*** *or* ***transpose*** *$f^r : Y^r \to X^r$ is defined as*



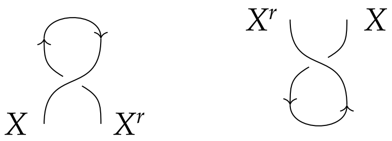
Definition 2.4.8. *In a right-rigid monoidal category where every object X has some chosen right dual X^r, the* ***right dual functor*** *$(-)^r : C \to C^{op}$ is defined on objects as $(X)^r = X^r$ and on morphisms as $(f)^r = f^r$.*

It is straightforward to check that this definition satisfies the functoriality axioms.

Working in a braided monoidal category allows us to drop the left/right bureaucracy.

Lemma 2.4.9. *If a monoidal category is braided, the difference between right/left duals vanishes, i.e. $X \dashv X^*$ implies $X^* \dashv X$.*

So in a braided monoidal category, we can simply speak of X, X^* being 'dual objects', rather than left or right dual. The proof of this lemma is to construct the caps and cups corresponding to $X^* \dashv X$ using braiding.



It is easy to see that this proposed cap/cup pair satisfies the yanking equations.

Finally we arrive at the definition of compact closed category[13]. Essentially, it is a *symmetric* rigid monoidal category.

Definition 2.4.10. *A* ***compact closed category****, or simply a* ***compact category****, is a symmetric monoidal category in which every object is dualizable*[14]*.*

Recalling the canonical closed structure arising from duals in lemma 2.4.6, we see that a compact closed category is in particular a closed symmetric monoidal category, with the internal hom given by $[A, B] \cong A^* \otimes B$. More precisely, we take the internal hom to be the functor

$$(-)^r \otimes - : C^{op} \times C \to C.$$

This justifies the name, i.e. it is what the terminology 'compact closed' is referring to.

Example. The canonical example of a compact closed category is $\mathbf{FVect}_k$[15]. Here the dual object X^* is the usual dual of the vector space X[16]. The unit and counit for the duality $V \dashv V^*$ is given by

$$\begin{aligned} \epsilon_V : V \otimes V^* &\to k \\ v \otimes f &\mapsto f(v) \end{aligned} \qquad\qquad \begin{aligned} \eta_V : k &\to V^* \otimes V \\ 1 &\mapsto \sum_i f_i \otimes e_i \end{aligned}$$

where $f : V \to k$ is a functional in the dual space, $\{e_i\}$ is some chosen (finite) basis for V, and $\{f_i\}$ is the dual basis, i.e. $f_i(e_j) = \delta_{ij}$.

In the special case of $\mathbf{FVect}_{\mathbb{R}}$, for any (finite dimensional) vector space V we may pick a basis and then canonically turn it into a real inner product/Hilbert space by

[13] An important note: in some literature, the term 'compact closed' refers to what we have called 'rigid' categories. This usage occurs in much of the DisCoCat literature. We reiterate that our definition of compact closed includes symmetry.

[14] This definition is from `https://ncatlab.org/nlab/show/compact+closed+category`.

[15] Note that the larger category $\mathbf{Vect}_k$ is *not* compact closed, only closed symmetric monoidal.

[16] See also `https://ncatlab.org/nlab/show/finite-dimensional+vector+space#CompactClosure`.

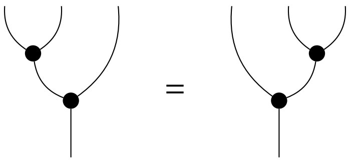
defining the dot product. Then by the Riesz representation theorem we have $V \cong V^*$, and hence $V \dashv V$, in which case $f(v)$ is interpreted as simply the dot product $\vec{f} \cdot \vec{v}$.

2.5 Spiders

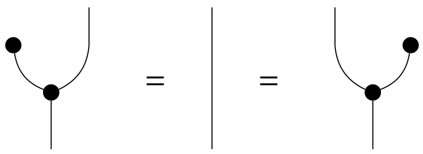
In this section, we discuss Frobenius algebras (also referred to as 'spiders'). These are a useful algebraic gadget that one can introduce to monoidal categories. They are used extensively in categorical quantum mechanics [**?**], and we will make use of them in our linguistics applications.

2.5.1 Frobenius algebras

Definition 2.5.1. *In a monoidal category, a* ***monoid*** *is a triple* $(X,$ $,$ $)$ *where X is an object, and* $: X \otimes X \to X$ *and* $: I \to X$ *are morphisms that satisfy the associativity*

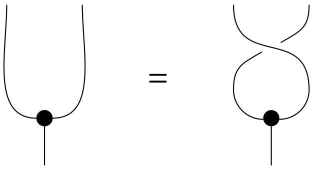


and unitality



equations. The morphism *is called the monoid* ***multiplication****, and* *is called the monoid* ***unit****.*

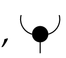
If the monoid additionally satisfies



(2.5)

then we say that it is a ***commutative monoid****.*

Note that in (2.5) it does not matter which braiding we took, since the condition would be equivalent to the one with the alternate choice of braiding.

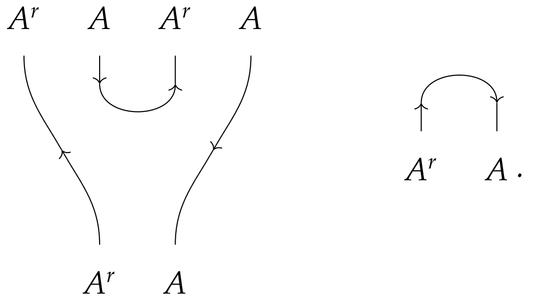
Example. To justify its name, in **Set** this definition coincides exactly with the classic definition of monoid. That is,  is just the monoid multiplication function sending

$(a, b) \mapsto a \cdot b$, and is just the monoid identity.

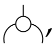
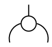
Example. In $\mathbf{Vect}_k$ a monoid is what is usually called a (unital) algebra. A special example is that in $\mathbf{FVect}_k$, a chosen basis $\{e_i\}$ gives a commutative monoid with multiplication defined by $e_i \otimes e_j \mapsto \delta_{ij} e_i$ (i.e. componentwise multiplication) and the unit defined $1 \mapsto \sum_i e_i$ (the vector of 1's).

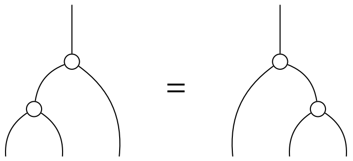
Example. The following is an important general example.

Definition 2.5.2 (Lem 4.11 [HV19]). *In a monoidal category with a chosen duality $A \dashv A^r$, the object $A^r \otimes A$ has a canonical monoid structure called the* ***pair-of-pants*** *monoid, with multiplication and unit*

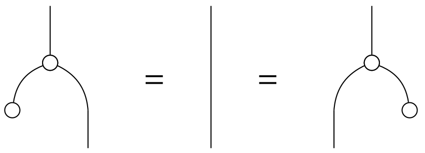


Dualising definition 2.5.1, we obtain the definition of comonoid.

Definition 2.5.3. *In a monoidal category, a* ***comonoid*** *is a triple* (X, ,) *where X is an object, and*  : $X \to X \otimes X$ *and* : $X \to I$ *are morphisms that satisfy the coassociativity*

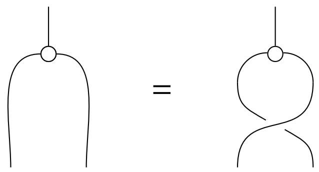


and counitality



equations. The morphism *is called the comonoid* ***comultiplication****, and* *is called the monoid* ***counit****.*

If the comonoid additionally satisfies



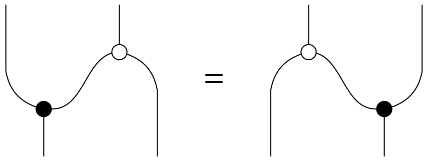
*then we say that it is a **cocommutative comonoid**.*

Example. In **Set**, every set X has a unique cocommutative comonoid structure, where ⋔ is the function $x \mapsto (x, x)$, and ⫰ is the unique function $X \to \{\bullet\}$.

Example. In $\mathbf{FVect}_k$, any basis $\{e_i\}$ gives a cocommutative comonoid, with ⋔ defined by $e_i \mapsto e_i \otimes e_i$, and ⫰ defined by $e_i \mapsto 1$.

Example. In a monoidal category with chosen duality $A \dashv A^r$, there is also a pair-of-pants comonoid on $A \otimes A^r$, defined in the obvious way.

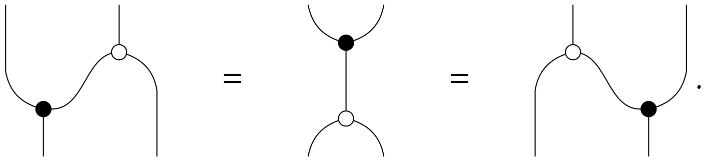
Definition 2.5.4. *In a monoidal category, a **Frobenius structure** (X,* ⋔*,* ⫰*,* ⋎*,* ⫯*) is tuple such that (X,* ⋔*,* ⫰*) is a comonoid, (X,* ⋎*,* ⫯*) is a monoid, and additionally the Frobenius law is satisfied.*



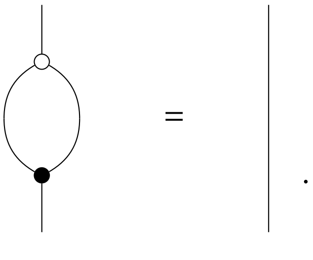
*We call a Frobenius structure **commutative** when its monoid is commutative and its comonoid is cocommutative.*

Frobenius structures satisfy an additional equality, which is sometimes unnecessarily included in the definition.

Lemma 2.5.5 (Extended Frobenius law, lem 5.4 [HV19])**.** *In a monoidal category, a Frobenius structure satisfies*



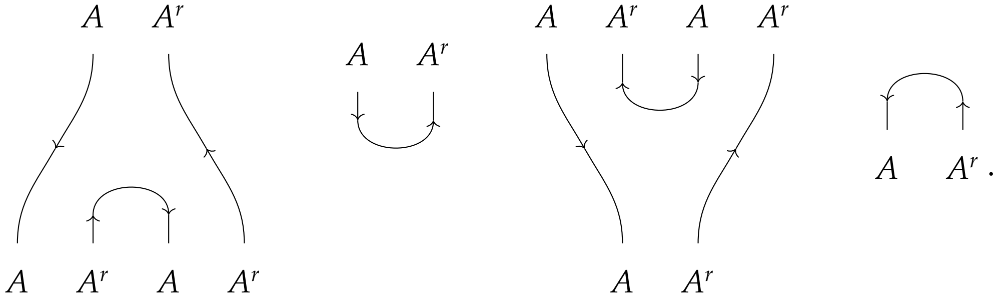
Definition 2.5.6. *In a monoidal category, a pair consisting of comonoid (X,* ⋔*,* ⫰*) and monoid (X,* ⋎*,* ⫯*) is **special** when* ⋔ *is a right inverse of* ⋎ *; i.e.*



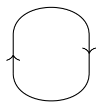
(2.6)

Thus, we speak of 'special Frobenius algebras', which are Frobenius algebras where the multiplication and comultiplication satisfy (2.6).

Example. In a monoidal category, suppose we have $A \dashv A^r$ and $A^r \dashv A$. Then we can construct the pair of pants monoid and comonoid on $A \otimes A^r$ say, which together yield a Frobenius algebra:



Moreover, if we have the following 'normalisation condition'[17]

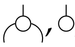
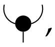
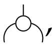
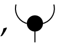
 =

then the pair-of-pants will additionally be a *special* Frobenius algebra.

Example. In **FVect**$_k$, a chosen basis $\{e_i\}$ induces a special commutative Frobenius algebra, by combining the commutative monoid and cocommutative comonoid of previous examples.

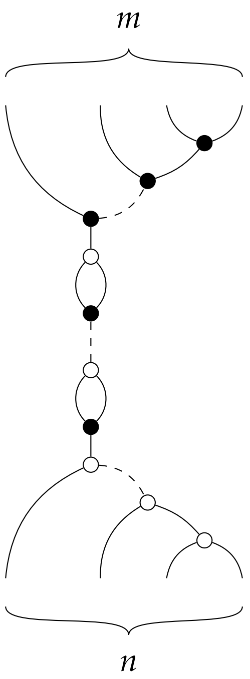
2.5.2 Normal forms

In this subsection, we justify the name 'spider', and describe the spider fusion rule.

Theorem 2.5.7 (Noncommutative spider theorem, theorem 5.21 [HV19]). *In a monoidal category, let* $(X,$ , , ,) *be a Frobenius structure. Any connected morphism* $X^{\otimes m} \to X^{\otimes n}$ *built of finitely many pieces* , , , *and identities, using* $\circ$ *and* $\otimes$*, is equal to the following*

[17] That is, the trace of the identity is equal to the empty diagram.

normal form:

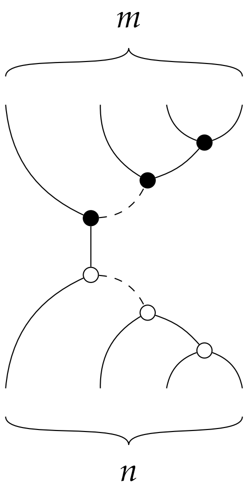


(2.7)

The dashed lines in the diagram indicate repeated subunits. The condition that the morphism be 'connected' means that it must have a graphical representation that has a path between any two vertices.

There are specific normal form theorems for specific Frobenius algebras. The following is the version for special Frobenius algebras (not necessarily commutative).

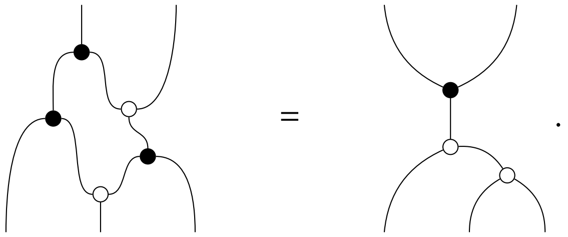
Theorem 2.5.8 (Special noncommutative spider theorem, lem 5.20 [HV19]). *In a monoidal category, let* $(X,$ ⅄, ○, Y, ●$)$ *be a special Frobenius structure. Any connected morphism* $X^{\otimes m} \to X^{\otimes n}$ *built of finitely many pieces* ⅄, ○, Y, ● *and identities, using* $\circ$ *and* $\otimes$*, is equal to the following normal form:*



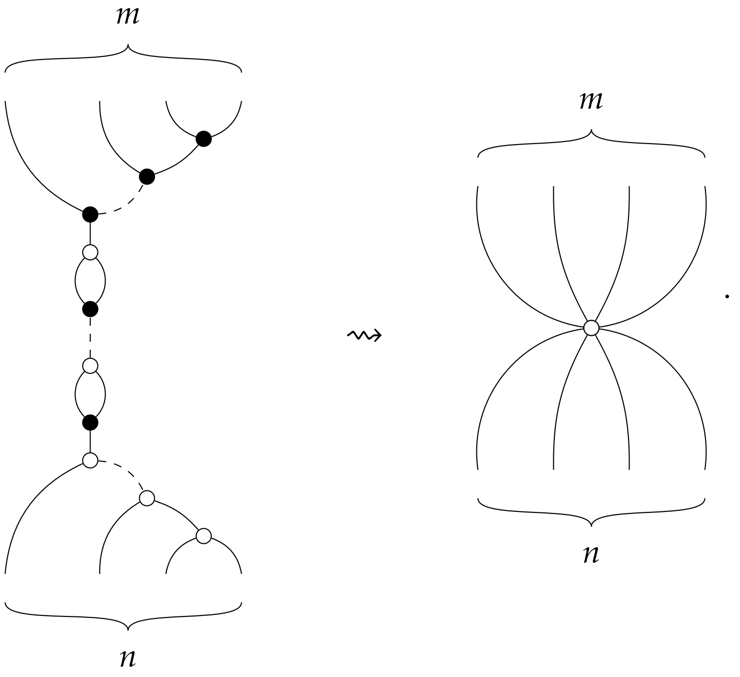
(2.8)

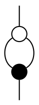
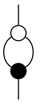
This theorem follows directly from the previous one.

Example. If (X, , , ,) is a special Frobenius algebra in a monoidal category, then theorem 2.5.8 would tell us that

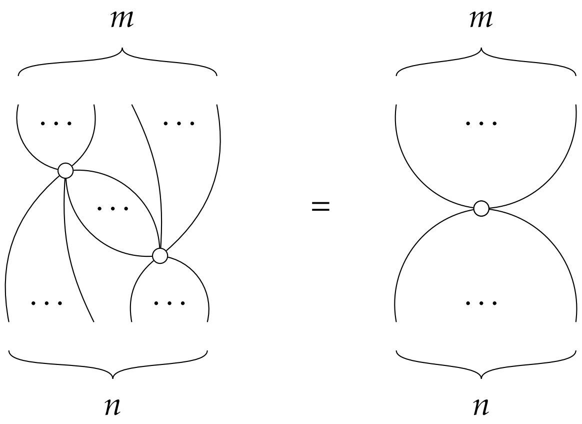


Normal form results for Frobenius structures like the ones above are called spider theorems because the normal forms resemble an ($m + n$)-legged spider. Indeed, we simplify our graphical notation by representing normal forms like (2.7) (or equivalently (2.8) in the case of a special Frobenius algebra) as simply:



Indeed, from now on we will simply use white dots in our graphical notation to represent spiders (and never again use white and black dots to denote monoids and comonoids). Note that in the non-special case, this notation suppresses information about the number of  components in the normal form - so that two spiders with the same number of input and output legs may not represent the same morphism due to having differing numbers of  components. However, in our applications, the spiders will always be *special*, so this problem does not arise, and our spiders are uniquely determined by their number of in/out legs.

With this convention, the spider theorem yields the following graphical rewrite rule known as ‘spider-fusion’:



i.e. given connected spiders of the same species (i.e. of the same underlying Frobenius algebra), we simply merge their heads.

Chapter 3

Grammatical background

This chapter provides background material on grammar, as it might be presented in a textbook for linguists. This material is important since our approach to NLP heavily emphasises the incorporation of grammar and the structural insight that it provides.

We begin with by outlining what exactly 'grammar' entails, and introduce some standard grammatical terms and concepts. We will discuss the dominant 'constituency' view of grammar, which sees sentence structure in terms of nested, hierarchical units of varying size. In particular, we will present a number of 'typelogical grammars' - highly formal systems for grammar, which yield constituency structures, and which form a crucial ingredient in the DisCoCat formalism. In the last section, we introduce a slightly different view of grammar and sentence structure based on 'dependency' - a perspective that turns out to be useful for our applications, as seen in chapters 5, 6).

3.1 The study of grammar

Roughly speaking, the grammar of a natural language is its set of structural constraints. In describing the grammar of a language, the central aim is to explain why speakers recognize certain forms as being 'correct' but reject others as being 'incorrect' [Kro05]. The term grammar can also refer more broadly to the study of such constraints, which includes domains like phonology (the study of patterns of sounds), morphology (the study of word shape), and semantics (the study of meaning, and the relationship between form and meaning). Of particular interest for us is the study of **syntax** - the set of principles that govern sentence structure. Though syntax is generally considered a subset of grammar, we will sometimes use these terms interchangeably - for instance, to say that a sentence is 'grammatical' is to say that it is 'syntactically valid'[Tal19].

To give a basic example, one key description of a language's syntax is the order in which the subject (S), verb (V), and object (O) usually appear in sentences[1]. For instance, English is an SVO language: consider the sentence "Alice ate cereal". On the other hand, Latin is generally SOV: consider the sentence

Servus	puellam	amat
slave	girl	loves

i.e. "the slave loves the girl".

An important note is that the rules of grammar are only concerned with the acceptability of the form itself, rather than the meaning or function expressed (though grammatical structure and meaning are often closely intertwined - see for instance the example of structural ambiguity in section 3.3, in which one sentence can have different meanings depending on how it is parsed). For instance, at the level of sentences, there is a clear distinction between syntax and semantics, in that it is possible for a sentence to be perfectly grammatical (syntactically valid) but have an obscure meaning. A famous example of such a sentence comes from Chomsky[Cho09]:

"Colourless green ideas sleep furiously."

Conversely, we can often make sense of a sentence even if it is not grammatically correct:

"Me Tarzan, you Jane."

We note that while grammar may feel relatively intuitive for humans, such that we can learn to speak without ever being consciously aware of the sophisticated grammar we are using, it is formally very complicated, and is far from being 'solved'. That is, to answer the question of 'which sentences are grammatical', linguists have come up with an enormous number of different, sophisticated grammatical theories and formalisms, all of which provide different answers to this question.

A consequence of this is that the notion of 'grammatical English' is a chimera - it is not well-defined. While there are some sentences that are clearly grammatical and others that are clearly ungrammatical, on the boundaries there are many sentences that may or may not be considered grammatical depending on what grammatical theory one adopts.

[1] We will explain these terms later.

3.2 Some basic grammatical units

An important feature of human language is the fact that larger units are composed of smaller units, and that the arrangement of these smaller units is significant. For example, a sentence is not just a structure-less string of words. Rather, words cluster together to form grammatical units of various sizes; these units are referred to as **constituents**[Kro05]. In this section, we will discuss important examples of such units, and also take this opportunity to introduce some other related grammatical concepts.

At the lowest level (that we are interested in), the *words* of any language can be classified according to their grammatical properties. These classes are traditionally referred to as **parts of speech** (POS) - e.g. noun, verb, etc. Linguists also refer to them as **syntactic categories** (though depending on who you ask, the latter term may also include phrasal categories)[Kro05]. Note these 'categories' have no relation to those of category theory.

Another important grammatical unit is the **clause** - the smallest grammatical unit which can express a complete proposition. To be more precise about this, we first define that the element of meaning which identifies the property or relationship is called the **predicate**. For instance, in the examples

"John is hungry."
"John loves Mary."
"Mary is slapping John."

the words "hungry", "loves", and "is slapping" express the predicates. The individuals (participants) of whom the property or relationship is claimed to be true (in these cases, "John" and 'Mary") are called **arguments**. Note in some languages, predicates may not require any arguments at all - e.g. in many languages statements like "It is raining" can be expressed by a single word, a bare predicate with no arguments. With this in mind, we say then that a clause is a grammatical unit which expresses a single predicate and its arguments[Kro05]. Some clauses ('independent clauses') can stand alone as a sentence - for instance, the clause "I first met her in Paris" in the sentence

"I first met her in Paris where I lived as a child."

However, not all clauses will form standalone sentences - in the above, "where I lived as a child" forms a 'subordinate clause', but clearly does not form a standalone sentence. A sentence may consist of just one clause or it may contain several clauses:

"[My wife told me that [I should introduce her little sister to the captain of the football team]], but [I assumed that [her sister was too shy]]."[Kro05]

Another important grammatical unit is the **phrase**. First, a phrase must be a group of words which form a constituent. Second, a phrase is lower on the grammatical hierarchy than clauses (though sometimes clauses may also be considered a kind of phrase). As a preliminary definition then, let us assume that a phrase is a group of words which can function as a constituent within a simple clause. Just as words may be classified into different categories, so too can we categorize phrases into different phrasal categories. The most 'important' word of a phrase is called the **head** of the phrase[NG18], and we generally name a phrase by the category of its head. So for instance,

"that big fish"

is a noun phrase, here the head is the noun "fish", and

"very beautiful"

is an adjective phrase where the head is the adjective "beautiful". Note though that English noun phrases do not always contain a head noun - in certain contexts a previously mentioned head may be omitted because it is 'understood'. Consider for instance the noun phrases in

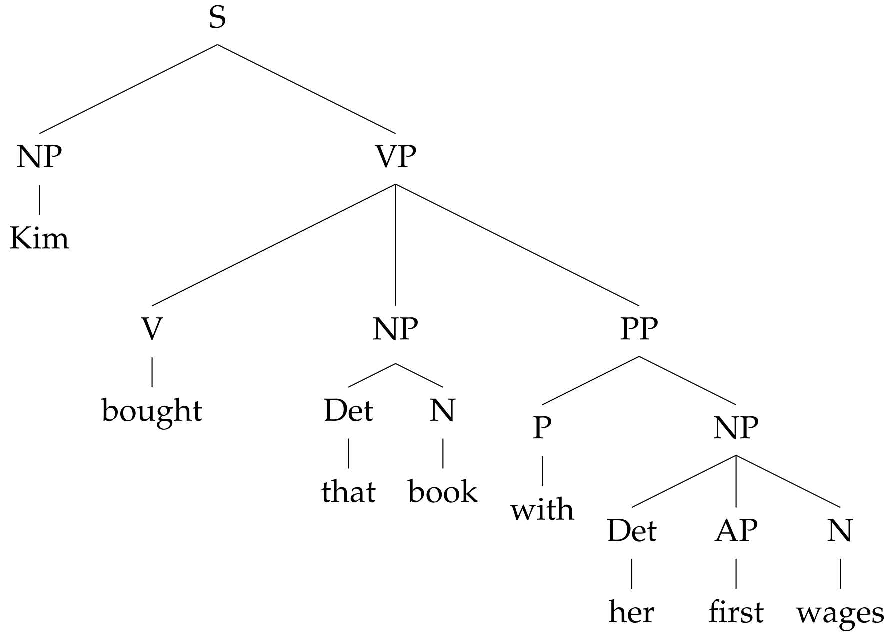
"[The third little pig] was smarter than [the second __]."

The set of grammatical units sentence, clause, phrase, word, (along with the sub-word-level unit called morpheme, which we did not discuss) is adequate for most languages. Each well-formed grammatical unit (e.g. a sentence) is made up of constituents which are themselves well-formed grammatical units, giving a kind of hierarchy. This hierarchy is an important aspect of linguistic structure.

3.3 Constituency grammar

Building on the constituent/hierarchical perspective of the previous section leads naturally to the so-called **phrase-structure** or **constituency** theories of grammar, which is one of the most common and basic ways of analyzing sentence structure.

We begin our discussion of constituency grammar by introducing **tree diagrams** - the most commonly used method of representing information about constituency and linear order. Consider a simple constituency-based tree diagram for the sentence "Kim bought that book with her first wages":



(3.1)

The key ideas are the following. The tree is rooted at the top, and grows down (thus, phrase structure rules are often considered to offer a *top-down* view of sentence structure). Each node of the tree is labelled by some category label (S = sentence, NP = noun phrase, V = verb, Det = determiner etc.), unless it is a **terminal node** which is labelled by an actual word. Evidently, higher nodes correspond to larger constituents.

Given a node A that is directly above and connected to a node B by an edge, we say that A **immediately dominates** B. A node that immediately dominates a set of nodes is their **mother**, and the nodes that are dominated are the **daughters**. Daughters of the same mother are **sister** nodes. If a node A is above node B, and B falls somewhere in the subtree growing down from A, we say that A **dominates** B (as opposed to immediately dominates, though immediate domination also implies domination)[Tal19].

As for how exactly one comes up with a constituency-based tree structure like (3.1), there are various 'syntactic tests' one can perform to check if certain words form constituents. Most importantly though, in constituency grammars, we usually assert that our parse trees must be able to be generated by **phrase structure rules** - rewrite rules of the following form

$$A \rightarrow B \quad C \tag{3.2}$$

which say that a node labelled A can be replaced by two nodes labelled B and C respectively, in that order. Each phrase structure rule thus defines a possible combination of mother and daughter nodes. The fact that there is only one symbol on the left-hand side of equation (3.2) means that the resulting grammar is *context-free*.

For instance, the rules

$$
\begin{aligned}
S &\rightarrow NP \quad VP \\
VP &\rightarrow V \quad (NP) \quad (PP) \\
NP &\rightarrow (Det) \quad (AP) \quad N \\
PP &\rightarrow P \quad NP
\end{aligned}
$$

are sufficient to generate the tree (3.1), where bracketed labels indicate optional arguments.

Note that different phrase structure theories with different rewrite rules will yield different looking trees for the same sentence. That is, given the sentence "Kim brought that book with her first wages", we likely would have produced a somewhat different looking constituency-based tree to (3.1), if we had adopted a different phrase structure formalism. For instance, Lexical Functional Grammar (LFG) and the Minimalist Program (MP) only allow phrase structure rules with strictly binary branching.

With these ideas in place, we can now propose a more concrete definition of constituent: a constituent is a string of words which are exactly those dominated by some node[Kro05]. Thus, the node labels indicate the category of the constituent it dominates.

To illustrate the importance of constituency and sentence structure, consider the phenomenon of **structural ambiguity**, which is where multiple different sentence structures for the same sentence/phrase lead to different interpretations of its meaning. For instance, the sentence

"The short bishop stole the tall bishop's hat"

can be parsed as "The short bishop stole the tall [bishop's hat]" or as "The short bishop stole the [tall bishop]'s hat"[Kro05].

3.4 Parts of speech

In this section we will discuss in more depth word-level syntactic categories. As with much of the grammatical theory we have thus far presented, there are varying approaches which will disagree with each other to some extent. Generally, the list of syntactic categories for English will roughly resemble the following[2].

[2] In addition to the sources provided, see `https://universaldependencies.org/u/pos/index.html` for a useful summary of POS.

- **Open classes**: this refers to classes of words to which we can add new words. For example, the nouns “byte”, “blog”, and “software” are all recent innovations in English, as are the verbs “breathalyse” and “decoke” (to remove carbon deposits from an engine)[Tal19].

  noun: a part of speech typically denoting a person, place, thing, or idea. A notable special kind of noun is a **proper noun**, which is the name of a specific individual, place, or object - for instance, “Bob”, or “North America”.

  (main) **verb**: words that typically describe events and actions, can constitute a minimal predicate in a clause, and govern the number and types of other constituents which may occur in the clause. One of the ways that verbs can be categorised is via the number and kind of arguments that they take - this is referred to as the **valency** of the verb. **Intransitive** verbs are those that take one argument, which we refer to as the **subject**. Examples of intransitive verbs include “Lee sneezed”, or “The volcano erupted”. **Transitive** verbs are those that take two arguments - a subject and an **object**. In this case, the subject is generally understood to be the ‘doer’ of the action, whereas the object is the one ‘acted upon’. Examples include “Lee broke the priceless vase”, and “Alice likes Bob”, where “Lee” and “Alice” are the subjects, and “the priceless vase” and “Bob” are the objects. Finally, we have **ditransitive** verbs, which take three arguments - a subject, a ‘direct’ object, and an **indirect object**. Typically, the participants will be someone performing the action (the subject), an item being acted upon (the direct object), and a recipient (the indirect object). An example is “Alice sent flowers to Lee”, where “Alice” is the subject, “flowers” is the direct object, and “Lee” is the indirect object. Many ditransitive verbs can also be transitive, when we drop the indirect object: “Alice sent flowers” [Tal19]. Note also that while the arguments in the above example sentences are noun phrases (NPs), arguments can in many cases be other categories. For instance, consider “Sam persuaded us to contribute to the cause” where the object is a verb phrase, or “That he came late did not surprise us” where the subject is a clause.

  adjective: words that typically modify nouns and specify their properties.

  adverb: words that typically modify verbs for such categories as time, place, direction or manner. An example is “Alice ran quickly”, A special class of words that are generally considered a subset of adverbs are the **intensifiers**, which cannot modify verbs but rather modify adjectives or other adverbs. The most common intensifier is “very”, which can modify both adjectives (“I am very red”) and adverbs (“I ran very quickly”). Other examples may include “fairly”, “too”, and “incredibly”[NG18].

- **Closed classes**: in contrast to open classes, closed classes contain only a small, fixed number of words, and new words are added very slowly.

  adposition: these are items that occur before (**preposition**) or after (**postposition**[3]) a noun-phrase-like complement, which form a single structure with the complement to express its grammatical and semantic relation to another unit within a clause. Examples include “under the floor”, “towards that conclusion”, and “outside my house”. In many languages, adpositions can take the form of fixed multiword expressions, such as “in spite of”, “because of”, “thanks to”.

  conjunction: these connect words, phrases, or clauses that are called the conjuncts of the conjunctions. This definition may overlap with that of other parts of speech (in particular, with adpositions), so what constitutes a conjunction must be defined for each language. For instance, “after” is a preposition in “he left after the fight”, since the complement her is a NP “the fight”. However, “after” is a conjunction in “he left after they fought”, since the conjunct “they fought” is not NP-like. Important subclasses here are the **coordinating conjunctions** that join items of equal syntactic importance, and **subordinating conjunctions** that join constructions by making one of them a constituent of the other.

  determiners: a word, phrase, or affix that occurs together with a noun or noun phrase and serves to express the reference of that noun or noun phrase in the context. Notable examples include **articles** (“a”, “the”), **demonstratives** (“this”, “that”), and **quantifiers** (“all”, “some”).

  auxiliaries: these are function words, often a verb, that accompany the lexical verb of a verb phrase and expresses grammatical distinctions not carried by the lexical verb, such as person, number, tense, mood, aspect, voice or evidentiality. Some examples include tense auxiliaries (“has done”, “is doing”, “will do”), passive auxiliaries (“was done”, “got done”), modal auxiliaries (“should do”, must do”), and verbal copulas (“He is a teacher”).

  pronouns: words that substitute for nouns or noun phrases, whose meaning is recoverable from the context. Pronouns under this definition function like nouns. Particular subclasses we will be interested in are **reflexive pronouns** (those ending in “-self”, e.g. “herself”, “itself”) and **relative pronouns** (those that introduce relative clauses, e.g. “a cat who eats fish”).

[3] It is generally accepted that the only common postposition in English is the word “ago” - hence, in English we often just use the term preposition to refer to adpositions.

There are also some other classes that are given their own category. These include interjections (“oh”, “ouch”, “bravo”), numerals (i.e. numbers), and particles (function words that must be associated with other words to impart meaning, which do not fall into other universal parts of speech, like “not”, or the possessive marker “’s”).

In addition to the notion of open classes and closed classes, another useful way to categorise words is as either a **function word** or **content word**. Function words are those with little semantic content that primarily express grammatical relationships (some adpositions, some conjunctions, some auxiliaries, articles, etc.). As we can see, the notion of function words broadly overlaps with the notion of closed class. On the other hand, content words possess semantic content and contribute to the meaning of the sentence, and are usually open class words.

3.5 Typelogical grammar

Linguists have developed many frameworks of grammar which aim to give a precise, scientific theory of syntactic rules. Among these frameworks, we will now discuss the formal approach provided by the so-called **typelogical grammars**. These will be of particular interest for our NLP applications.

Typelogical grammar has its roots in the work of Kazimierz Ajdukiewicz in the 1930s [Ajd78], and the later work of Yehoshua Bar-Hillel and Jim Lambek. Broadly, the main aim of this approach is “to obtain an effective rule (or algorithm) for distinguishing sentences from non-sentences” [Lam58].

To this end, one assigns logical types to constituents, which then combine according to some fixed set of rules as arguments and functions. The judgement that a sentence is well-formed or grammatical boils down to performing a deduction proof in a Gentzen-style sequent calculus [Moo14]. That is, a typelogical grammar has two characteristic components: 1) a **type lexicon**, which assigns a set of types to each word, and 2) some type inference rules, which determine how the type of a string of words follows from the types of the constituent words. A string of words that can be shown to have the ‘sentence type’ is considered a valid sentence in this typelogical grammar framework[Bus18].

Traditionally, such typelogical grammars are sometimes also referred to as **categorial grammars** - here ‘category’ is used in the sense of ‘type’, or ‘syntactic category’.

3.5.1 Basic categorial grammar (BCG)

The simplest examples of typelogical grammar are the **basic categorial grammars** (BCG), sometimes also called AB-grammars (after Ajdukiewicz and Bar-Hillel)[BH53].

The basic categorial grammar somewhat resembles the simply typed lambda calculus, though the lambda calculus has only one function type $A \to B$, whereas a categorial grammar typically has two function types. These two types correspond to functions taking the input on the right and on the left respectively. That is, given types A and B, a simple categorial grammar admits function types B/A and $A\backslash B$. The former, B/A, will return a phrase of type B when combined with a phrase of type A to its right. That is, we have the type inference rule

$$\frac{B/A \qquad A}{B}$$

The latter, $A\backslash B$, will return a phrase of type A when combined with a phrase of type B to its left[4] , giving the type inference rule

$$\frac{B \qquad A\backslash B}{A}$$

In basic categorial grammars, these are the only two inference rules via which types can combine.

As a matter of convention, when we have higher order types that include several layers of arguments, we will often drop the brackets and assume that they 'associate to the left'. That is, we may write

$$A\backslash B/C$$

to mean $(A\backslash B)/C$.

With the inference rules in place, we now need to specify the type lexicon. Firstly, we fix some set of primitive types P. These are sometimes also called **basic types** or **atomic types**. Then we freely generate a larger set of types $T(P)$, via the recursive rule that if $X, Y \in T(P)$ then $X\backslash Y$, X/Y are also in $T(P)$. The additional functor types that are generated in this way are called **complex types**, which are disjoint from the primitive types. We then associate each word with some types in $T(P)$. Note that one word may be associated with multiple types, since depending on the context, it can serve different functions - this is the structural ambiguity of words in natural language as discussed

[4] Note that there is another convention in which $A\backslash B$ represents a type that returns type B when combined with a type A to its left, i.e.

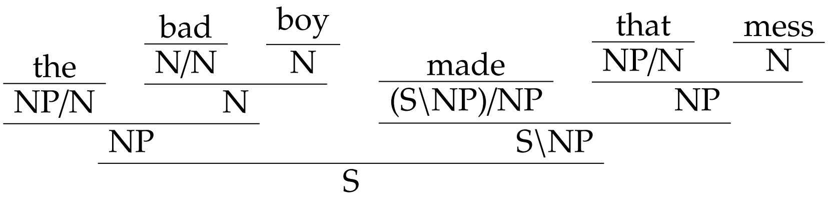
$$\frac{A \qquad A\backslash B}{B}$$

We do not use this convention.

in section 3.3. For instance, the word "and" can connect sentences, or nouns, or verbs, and so on[Bus18].

We now have a basic categorial grammar. A string of words is then considered grammatical according to this grammar, if each word in the string can be assigned a type (as per the lexicon) such that the lexical types of the words in the string can be combined via the two inference rules to form a constituent.

All this is probably best made clear through an example. A simple BCG system for English might have the three basic types *N* (nouns), *NP* (noun phrases), and *S* (sentences). Then an adjective might be assigned the type *N/N*, since if it is followed by a noun, it can combine with the noun to give a composite that is also a noun - e.g. "red" as in "red car". A determiner like "the" has type *NP/N* because it forms a complete noun phrase when followed by a noun, and a transitive verb like "made" has type (*S*\NP)/*NP*, since it absorbs a noun phrase to the right and to the left to return a completed sentence. In this system, the string "the bad boy made that mess" can be shown to be a grammatical sentence, via the deduction



Note that information can also be extracted from the reductions that are made - it shows that the sentence can be parsed as

"[the [bad boy]] [made [that mess]]"

Categorial grammars of this form (with only function application rules) are equivalent in generative capacity to context-free grammars [BHCS60], a degree of expressiveness that is known not to be adequate for natural language. For instance, [BKPZ82] and [Shi85] have shown that certain syntactical constructions in Dutch and Swiss-German give rise to cross-serial dependencies and are beyond context-freeness.

3.5.2 Combinatory categorial grammar (CCG)

The CCG framework is an extension of the basic categorial grammar via additional inference rules which are based on the combinators of combinatory logic [CFC+58]. Refer to [Ste00] for a comprehensive treatment; the PhD thesis [Hoc03][5] also contains

[5] Accessible at `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.135.8543&rep=rep1&type=pdf`.

a useful summary. CCG is more expressive than BCG - [VSW94] showed it is a mildly context-sensitive grammar.

We list the main inference rules of CCG below. Each kind of inference rule comes in forwards and backwards varieties, which are essentially mirror images. Firstly, the usual forward and backward application rules are the same as basic CG.

- Forward application

$$> \frac{\alpha : X/Y \qquad \beta : Y}{\alpha\beta : X}$$

- Backward application

$$< \frac{\alpha : Y \qquad \beta : X\backslash Y}{\alpha\beta : X}$$

In addition, there is composition of functors (also called 'harmonic function composition')

- Forward composition

$$B_{>} \frac{\alpha : X/Y \qquad \beta : Y/Z}{\alpha\beta : X/Z}$$

- Backward composition

$$B_{<} \frac{\alpha : Y\backslash Z \qquad \beta : X\backslash Y}{\alpha\beta : X\backslash Z}$$

Type-raising is an unary rule which reverts the role of functor and argument.

- Forward type-raising

$$T_{>} \frac{\alpha : X}{\alpha : Y/(Y\backslash X)}$$

- Backward type-raising

$$T_{<} \frac{\alpha : X}{\alpha : Y\backslash(Y/X)}$$

Composition and type-raising do not affect the grammar's theoretical power, but allow additional flexibility in the order of composition. For instance, consider the sentence "Alice likes Bob". As per the example BCG system given in subsection 3.5.1, we assign the type $(S\backslash NP)/NP$ to the ditransitive verb "likes". In BCG, we would be forced to contract "likes" with the object noun "Bob" first, and then with the subject noun "Alice", but in CCG we can use composition and type-raising to instead have

$$> \cfrac{B_> \cfrac{T_> \cfrac{\cfrac{\text{Alice}}{NP}}{S/(S\backslash NP)} \qquad \cfrac{\text{likes}}{(S\backslash NP)/NP}}{S/NP} \qquad \cfrac{\text{Bob}}{NP}}{S}$$

There is also a 'crossed' version of composition, that allows to deal with certain phenomena in which words are not found at their canonical positions.

- Forward cross composition

$$BX_> \frac{\alpha : X/Y \qquad \beta : Y\backslash Z}{\alpha\beta : X\backslash Z}$$

- Backward cross composition

$$BX_< \frac{\alpha : Y/Z \qquad \beta : X\backslash Y}{\alpha\beta : X/Z}$$

One example that crossed composition allows us to deal with is the so-called 'heavy NP-shift', which roughly speaking involves reordering (shifting) a 'heavy' noun phrase to the right of its canonical position[YK21]. Consider the sentence "John passed successfully his exam" - canonically, we would expect the noun phrase "his exam" to sit in the position "John passed his exam successfully". Nevertheless, with the backward cross composition rule, we can have

$$< \cfrac{\cfrac{\text{John}}{NP} \qquad > \cfrac{BX_< \cfrac{\cfrac{\text{passed}}{(S\backslash NP)/NP} \qquad \cfrac{\text{sucessfully}}{(S\backslash NP)\backslash(S\backslash NP)}}{(S\backslash NP)/NP} \qquad \cfrac{\text{his exam}}{NP}}{S\backslash NP}}{S}$$

In English this rule is used in a restricted form - Steedman disallows the use of the forward version, and only permits the backward version when $Y = S\backslash NP$ [Ste00].

Another extension is the 'generalized composition' rule. To state this rule, we first introduce a new notation - the **$ convention** defined by Steedman.

Definition 3.5.1. *For a category X, we define $\{X/\$\}$ (resp. $\{X\backslash\$\}$) as the set containing X and all leftward functions (resp. rightward functions) into a category in $\{X/\$\}$ (resp. $\{X\backslash\$\}$).*

This recursive definition essential boils down to the following. $\{X/\$\}$ is the set of types of the form $X/Y_1/Y_2/\ldots/Y_n$, and $\{X\backslash\$\}$ is the set of types of the form $X\backslash Y_1\backslash Y_2\backslash\ldots\backslash Y_n$, and where $n \geq 0$ and the Y_i's are possibly complex. We then use unbracketed $X/\$$ (resp. $X\backslash\$$) to schematize over members of the set $\{X/\$\}$ (resp. $\{X\backslash\$\}$), using subscripts to distinguish different schematizations. That is, $X/\$_1$, $X/\$_2$ denote possibly different types, and $Y/\$_1$ takes the same series of arguments on the right as $X/\$_1$, though the former will return Y and the latter will return X.

We now define generalized composition.

- Generalized forward composition:

$$B^n_> \frac{X/Y \qquad (Y/Z)/\$_1}{(X/Z)/\$_1}$$

- Generalized backward composition:

$$B^n_< \frac{(Y\backslash Z)\backslash\$_1 \qquad X\backslash Y}{(X\backslash Z)\backslash\$_1}$$

This rule effectively allows us to compose while 'ignoring the outer arguments' in one of the types. Each of these rules actually corresponds to a family of rules - one for each arity n of the secondary functor. So for instance, for $n = 2$ we have

$$B^2_< \frac{(Y\backslash Z_1)\backslash Z_2 \qquad X\backslash Y}{(X\backslash Z_1)\backslash Z_2}$$

For $n = 1$ generalised composition reduces to the usual composition: $B^1_> = B_>, B^1_< = B_<$. The $n = 0$ case can be viewed as reducing to the application rule. Without any restriction on the arity n, full context-sensitivity would be obtained. However, there has not yet been any evidence that this is required to capture natural language syntax. Therefore, only schemata up to a bounded arity n (Steedman assumes 4 for English) are allowed in practice [Hoc03]. These generalized composition rules have special significance, since it is argued to be the reason for the beyond context-free generative capacity of CCG - see for example [KKS15].

In addition to the five kinds of rules we have thus far defined (application, composition, type-raising, cross composition, generalized composition), there are some other rules such as 'generalized cross composition', 'substitution', and 'cross substitution', which can be introduced to handle additional grammatical phenomena (see section 2.3 of [Hoc03]). However, these are less frequently used (for instance, by parsers, or in CCGbank), and we will not make use of them. The five kinds of rules we have presented are the more frequently used and arguably the most important ones.

3.5.3 Pregroup grammar

Pregroup grammar (PG) is another formalism that belongs in the tradition of typelogical grammars (though in a sense they are not *strictly* typelogical grammars due to there being no known type-theoretic semantics known for them [Bus18]). They were introduced by Lambek in [Lam97] as a simplification of his original syntactic calculus, and are efficiently parseable. Pregroup grammars have been proved to be weakly equivalent to context-free grammars [Bus01]. In other words, they share the expressive limitations of the original categorial grammars. For an extensive discussion of pregroup grammar, see [Lam08].

First we will describe the mathematics of pregroups.

Definition 3.5.2. *A **partially ordered monoid** $(P, \leq, \cdot, 1)$ is a partially ordered set $(P, \leq)$ equipped with a monoid multiplication $- \cdot -$ and a monoid unit 1, such that multiplication is monotone*[6]*:*

$$x \leq y \wedge x' \leq y' \implies x \cdot x' \leq y \cdot y' \tag{3.3}$$

Definition 3.5.3. *A **pregroup** $(P \leq, \cdot, 1, (-)^l, (-)^r)$ is a partially ordered monoid where every element $p \in P$ has a left adjoint p^l and right adjoint p^r such that*

$$p^l \cdot p \leq 1 \leq p \cdot p^l \tag{3.4a}$$

$$p \cdot p^r \leq 1 \leq p^r \cdot p \tag{3.4b}$$

From the definition of adjoint it can be verified that the following properties hold for pregroups:

$$1^l = 1 = 1^r$$

$$x^{lr} = x = x^{rl}$$

$$(x \cdot y)^l = y^l \cdot x^l \qquad\qquad (x \cdot y)^r = y^r \cdot x^r$$

Note the latter property allows us to verify that the adjoint rules extend to products - for instance, for left adjoints we have

$$(x \cdot y)^l \cdot (x \cdot y) = y^l \cdot x^l \cdot x \cdot y \leq y^l \cdot 1 \cdot y = y^l \cdot y \leq 1. \tag{3.5}$$

We can also show that adjoints are unique. Suppose that p^l, p' are both left adjoints of p. Then using (3.4a), we have $p^l = p^l \cdot 1 \leq p^l \cdot p \cdot p' \leq 1 \cdot p' = p'$. So $p^l \leq p'$, but by symmetry we also have $p' \leq p^l$, and the antisymmetry property of posets gives $p^l = p'$.

[6] Equivalently, we could enforce the condition $p \leq q \implies r \cdot p \cdot s \leq r \cdot q \cdot s$.

Another property is that adjoints are order reversing. That is,

$$p \leq q \implies q^r \leq p^r \text{ and } q^l \leq p^l. \tag{3.6}$$

Starting with $p \leq q$ and multiplying by p^r on the left and q^r on the right gives $p^r \cdot p \cdot q^r \leq p^r \cdot q \cdot q^r$, which reduces to $q^r \leq p^r$. Analogously we can show $q^l \leq p^l$.

With the mathematics out of the way, we describe how pregroups can be used to formalise grammar. Firstly, we presuppose a partially ordered set of basic types. This is analogous to fixing the set of primitive types in the categorial grammars. We use an arrow → instead of ≤ to denote this order, where $X \rightarrow Y$ is interpreted as saying that type X can be reduced to type Y. For instance, if our set of primitive types include the type π for 'subject', and π_3 for 'third person singular subject', then we would have the reduction $\pi_3 \rightarrow \pi$, since a third person singular subject is a special instance of a subject. In proper poset notation this would be written $\pi_3 \leq \pi$, which evokes the fact that π is a 'larger' type that includes π_3.

A simple pregroup system for English (which we will primarily make use of in this thesis) can just have two unrelated primitive types n and s, representing nouns and sentences respectively. This follows the spirit of Lambek's original categorial grammar paper [Lam58], which also only has the basic n and s types.

Next, we freely generate a pregroup of these primitive types by taking adjoints and concatenating - this will constitute of the set of possible types for our lexicon. More concretely, for any primitive type p, we can first construct **simple types** $\ldots p^{ll}, p^l, p, p^r, p^{rr} \ldots$. We *posit* that the adjoint conditions (3.4) hold - then we will have the following type reductions ('contractions')

$$p^l \cdot p \rightarrow 1 \qquad p \cdot p^r \rightarrow 1.$$

That is, we may cancel a type with its left adjoint on its left, or with its right adjoint on its right. Note that the adjoint conditions (3.4) also yield the following 'expansions' :

$$1 \rightarrow p \cdot p^l \qquad 1 \rightarrow p^r \cdot p.$$

However, it turns out that for our linguistic application, contractions alone are sufficient for type reduction proofs, and we may ignore expansions[7]. The partial order from our basic types can then be extended to simple types via (3.6).

By 'compound type', or just type, we mean a string $x_1 \ldots x_n$ of simple types with $n \geq 0$. We take the monoidal multiplication to be concatenation of strings (henceforth we omit the monoid multiplication symbols). Thus if $n = 0$ we have the empty string,

[7] For more details, refer to the 'switching lemma' in chapter 27 of [Lam08].

which is our monoidal unit 1 satisfying $1x = x = x1$. Mathematically, we have obtained the *free monoid generated by the set of simple types*. The adjoint properties (3.4) also hold for compound types, via (3.5) The partial order can then be extended from simple types to compound types via (3.3). This completes the construction of the pregroup of possible types for our lexicon.

As usual, a string of words forms a grammatical sentence if each word in the string can be assigned a pregroup type as per the type lexicon, such that the concatenation of the types of all the words in the sentence can be reduced via contractions to the s type. For instance, in the simple system with primitive types n and s, transitive verbs have type $n^r s n^l$, so the sentence "John likes Mary" has the overall type

$$\begin{array}{ccc} \text{John} & \text{likes} & \text{Mary} \\ n & n^r s n^l & n. \end{array}$$

This string can be shown to be a grammatical sentence, via the sequence of reductions

$$n n^r s n^l n \to 1 s n^l n \to s n^l n \to s1 \to s. \tag{3.7}$$

Another way to present a sequence of reductions like the one above is to use 'underlinks' that indicate which types were contracted[Lam08]:

$$\underbrace{n \quad n^r} \quad s \quad \underbrace{n^l \quad n} \tag{3.8}$$

3.5.4 Relationship between PG and CG

It is worth noting that, given a basic categorial grammar with some generating set S of basic types, one can translate all the types and BCG proofs into a pregroup grammar with the same generating set S of basic types. Under this translation, any proof that is possible in the BCG is possible in the PG. Concretely, the translation of types is done by recursively applying the rules

$$p\backslash q \mapsto q^r \cdot p \tag{3.9a}$$

$$p/q \mapsto p \cdot q^l. \tag{3.9b}$$

Indeed, we can check for instance that under this mapping the transitive verb type $(S\backslash N)/N$ in categorial grammar goes to its usual pregroup type $N^r S N^l$. Then, the usual forward and backward application rules of the BCG are realizable in the PG: for example, for instance

$$\frac{\mathrm{X/Y} \qquad \mathrm{Y}}{\mathrm{X}} \qquad \rightsquigarrow \qquad X \cdot Y^l \cdot Y \to X.$$

Of course, the same does not hold in the opposite direction - certain proofs in this PG system are not realizable in the original BCG. Indeed, the type-map presented above is not injective - for instance, both $X\backslash Y/Z$ and $X/Z\backslash Y$ will map to the pregroup type Y^rXZ^l.

Going beyond BCG to CCG, we remark that under this mapping, the forward and backward composition rules of CCG are also realizable in the PG:

$$\frac{\mathrm{X/Y} \qquad \mathrm{Y/Z}}{\mathrm{X/Z}} \qquad \rightsquigarrow \qquad X \cdot Y^l \cdot Y \cdot Z^l \to X \cdot Z^l.$$

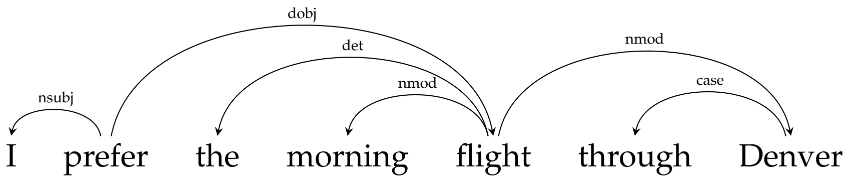
Similarly, the generalised forward and backward composition rules are realizable. However, the type-raising and cross composition rules are not.

3.6 Dependency grammar

This chapter has thus far has focused on a constituency based viewpoint of grammar. In particular, the categorial grammars of section 3.5 yield constituency structures, and pregroup grammar to a lesser extent also reflects a constituent-based view of grammar via its contraction of *adjacent* words.

However, an important parallel concept to constituency is that of **dependency**, which we now discuss. A dependency is essentially a directed edge between two words in a sentence, and **dependency grammar** (DG) is a family of formalisms in which the syntactic structure of a sentence is described solely in terms of the words and dependencies between the words. Often, the dependencies will be labelled with a type, drawn from some fixed inventory of grammatical dependencies - these are called **typed dependency structure**. In DG approaches, phrasal constituents and phrase-structure rules do not play a direct role [JM, Chapter 14].

A typical tree-like structure resulting from dependency analysis is the following.



(3.10)

We can see that some basic examples of dependencies are those between a verb and its arguments. In (3.10), the verb transitive "prefer" is linked to its subject "I" via an 'nsubj' dependency, and to the root of its object ("flight") via a 'dobj' dependency. Linguists have, of course, developed taxonomies of relations that go well beyond the familiar notions of subject and object, some of which we can see in the example. The Universal Dependencies project [NDMG$^+$16] provides a useful inventory of dependency relations that are linguistically motivated, computationally useful, and cross-linguistically applicable[8].

Another characteristic of dependency trees is the one-to-one correspondence between nodes in the tree and words in the sentence. By contrast, phrase structure trees are one-to-one-or-more correspondence, which means that, for every element in a sentence, there is one or more nodes in the tree that correspond to that element. The result of this difference is that dependency structures are minimal compared to their phrase structure counterparts and tend to contain many fewer nodes.

Additionally, in DG approaches the (finite) verb is taken to be the structural center of clause structure. Note for instance in (3.10) that the *root* of the tree is the verb "prefer". Indeed, all other words are either directly or indirectly connected to the verb via a directed path.

A major advantage of dependency grammars is their ability to deal with languages that have a relatively free word order. Dependency grammar abstracts away from word order information, representing only the information that is necessary for the parse.

[8] Refer also to `https://universaldependencies.org/` for more information.

Chapter 4

Diagrammatic NLP background

In this chapter, we give a presentation of the diagrammatic, compositional-distributional approach to modelling language and performing natural langauge processing, which will draw upon the background material in the previous two chapters. 'Compositional-distributional' is often abbreviated to DisCo, thus leading to terms like 'DisCoCat' and 'DisCoCirc'. I refer to this approach to natural language processing as 'diagrammatic NLP', since at the theoretical level, *the central problem at hand is that of modelling language with diagrams*.

We will begin by saying a few words about the motivation behind the idea of compositional-distributional NLP. Then, we will present the original DisCoCat framework based on pregroup grammar, that was proposed in 2010. We will also present an alternate CCG-based DisCoCat framework, which was recently formalised in [YK21] and offers several advantages over the old pregroup-DisCoCat. Finally, we will devote some space to discussing my thoughts regarding the relatively new idea of language circuits - the main object of interest of this thesis. We will also briefly discuss the concept of 'internal wiring' which is the focus of chapter 7.

4.1 Motivation: compositional and distributional semantics

As suggested by its name, the compositional-distributional approach to NLP was conceived of as a synthesis of two paradigms in semantics. In this section we will give a rough sketch of these paradigms, their place in NLP, and how they motivate the DisCo approach. See [Gaz96] for a discussion of these competing paradigms in NLP.

At the level of sentences, the traditional theoretical approach to semantics broadly followed the *syntax-driven*, *compositional* tradition of Montague [JZ21]. An essential feature of this approach is the reliance on the systematic relation between syntax and semantics, as encapsulated in the *Principle of Compositionality*. It reads, in a formulation that is standard nowadays:

> "The meaning of a compound expression is a function of the meanings of its parts and of the way they are syntactically combined."

That is, the meaning of a sentence arises from the way that individual words compose and interact. Concretely, such theories generally provided a pairing of syntactic analysis rules paired with a semantic interpretation. Indeed, the Lambek style categorial grammars discussed in section 3.5, which come with a compositional semantics usually expressed via lambda calculus, can be viewed as realizing Montague's compositionality program in an uncompromising way [Moo14].

In the context of NLP, these semantic theories were traditionally applied in a *symbolic* approach. 'Symbolic' approaches to problems in artificial intelligence are those in which high-level, human-readable symbols are used to represent entities or concepts as well as logic in order to create rules for the concrete manipulation of those symbols, leading to a rule-based system. Put another way, symbolic approaches involve the explicit embedding of human knowledge into computer programs.

Though they were once the dominant paradigm in artificial intelligence, traditional symbolic approaches have largely fallen out of fashion, especially with the deep learning revolution occurring over the last decade or so. In the field of NLP, the modern standard is 'distributional semantics', which is a kind of *black-box*, *statistical* approach that uses vector spaces as a model for meaning. The main idea behind this approach is captured by Firth's famous dictum that "you shall know a word by the company it keeps" [Fir57]. At a practical level, this means the meaning of a word can be learned from a corpus by looking at what other words occur with it within a certain context. One then builds co-occurrence vectors for each word, which then allows us to compare the meanings of words by looking at their vector distance.

Example. To give a simple illustration[1], suppose that we are given a real world training corpus, and want to use it to construct semantic vectors for the words "dog", "cat" and "snake". In a simple model, the basis vectors of our semantic vector space will be annotated with words from the lexicon. So let us choose a set of context words "furry", "pet", "stroke" which will serve as the labels for our basis vectors. The vector space we are working in will then be three dimensional. Additionally, we will consider 'context'

[1] This example is taken from [Gre13], which can be accessed at `https://arxiv.org/pdf/1311.1539.pdf`.

to mean 'words occurring in the same sentence as the target word'. So if in our corpus the word "dog" occurs in the same sentence with the word "furry" twice, and occurs twice with "pet" and once with "stroke", we will set its semantic vector to be

$$\vec{dog} = [2\ 2\ 1]^T.$$

Similarly, suppose we find that "cat" occurs thrice in the same sentence as "furry", once as "pet" and does not occur in the same sentence as "stroke". Likewise, we note that "snake" does not occur in the context of 'furry', but twice in the context of "pet" and "stroke". We will then have the semantic vectors

$$\vec{cat} = [3\ 1\ 0]^T \qquad \vec{snake} = [0\ 2\ 2]^T$$

With these semantic vectors we can now do things like compare their similarity by taking their cosine measure

$$\cos(\vec{v}, \vec{u}) = \frac{\langle \vec{v}, \vec{u} \rangle}{||\vec{v}|| \cdot ||\vec{u}||}.$$

In this case, we would find that "dog" is 10% closer in meaning to "cat" than it is to "snake".

These distributional models have been very useful in many natural language tasks [Cur04, SMR08, Sch98]. However, the same idea does not naturally scale up. For example, larger constituents of text such as phrases or sentences are much more unique than words. As such, there is currently no text corpus available that can provide reliable co-occurrence statistics for anything larger than text segments of more than a few words[Kar15].

Beyond the practical issue of the immense amount of data that would be required to scale up a purely distributional approach, there are other reasons that motivate going this paradigm. The first is that a recent trend within AI is leading back towards more symbolic approaches, due to their superior understandability/explanability. There is an increasing recognition of the importance of models that are more transparent and can be understood by humans, for both ethical and operational reasons[VL20]. Data-driven, machine learning models however, like the distributional semantic model, are essentially opaque and incomprehensible to humans.

Furthermore, while distributional semantic models have been very successful at certain tasks, their opaque nature means that they have done little to advance our actual understanding of language. To quote [Coe21], "it would be a major mistake to only follow the path where empirical success takes us, and ignore that which increases understanding"[2]. Despite the current success of data-driven, statistical models, there remains a need for greater *structural understanding* of the underlying phenomena - understanding which, in the long term, may well yield models that outperform purely data-driven ones.

Finally, the human ability to understand sentences intuitively feels like a compositional mechanism. We can comprehend sentences we have never seen before because we can generate its meaning from the meaning of its constituent words, of which we do have prior understanding.

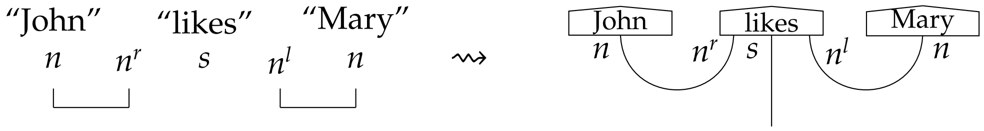
DisCo-style NLP seeks to address these issues. It allows the highly successful vector space approach to NLP to be used for word meanings, but draws upon from formal theories of grammar to compose these word meanings and obtain the meaning of the whole. Through the compositionality, some semblance of structure is imposed, rather than being purely statistical.

This combination yields models that can feasibly deal with the meanings of sentences and even of entire texts. The way all this works is made concrete in the next subsection, where we present the details of DisCoCat, which is the first iteration of the compositional-distributional paradigm.

4.2 DisCoCat (via Pregroup grammar)

The first iteration of diagrammatic NLP program is the **DisCoCat** model (distributional-compositional-categorical) based on pregroup grammar[CSC10]. In this section we give an in depth description of the pregroup-based DisCoCat formalism.

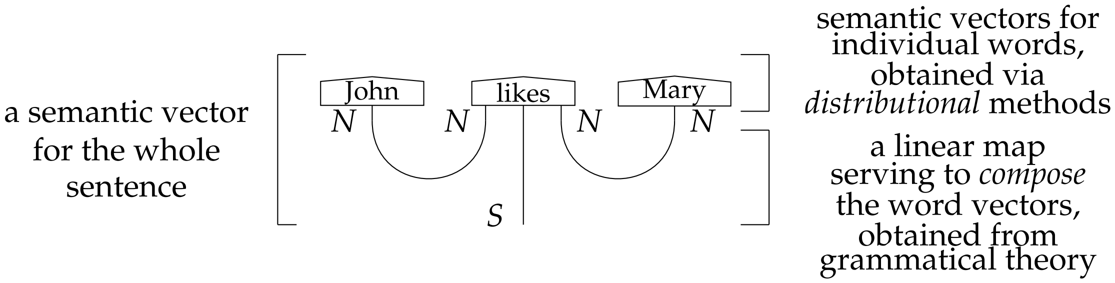
The key insight in the inception of DisCoCat is that pregroup type reduction proofs (especially when visualised using underlinks like in equation (3.8)) correspond exactly to string diagrams involving cups and identities in a general category.



Hence, the type reductions of Pregroup grammar can be 'lifted' to morphisms in a general category. If we then *interpret* the resulting string diagram in the compact-closed category $\mathbf{FVect}_k$, this allows us to make use of the standard NLP practice of encoding word meanings as vectors. Thus, we propose that the semantic vector of the

[2] Refer to this paper for an analogy involving Ptolemy's epicycle model for the movement of celestial bodies.

sentence “John likes Mary” is precisely the state expressed by the string diagram when interpreted in $\mathbf{FVect}_{\mathbb{R}}$:


a semantic vector for the whole sentence
John
likes
Mary
N
N
N
N
S
semantic vectors for individual words, obtained via *distributional* methods
a linear map serving to *compose* the word vectors, obtained from grammatical theory

(4.1)

where N, S are appropriately chosen vector spaces. As promised, this model synthesises distributional and compositional ideas in order to to obtain the meaning of longer text segments like sentences.

In the rest of this section, we will make this idea precise.

4.2.1 Pregroups as a rigid category

Mathematically, the insight that pregroup proofs map to string diagrams is more formally expressed by the fact that the pregroups of PG have the structure of a *rigid monoidal category*.

Lemma 4.2.1. *A pregroup can be viewed as a rigid strict monoidal posetal category,* ***Preg***[3].

Proof. Firstly, a partially ordered monoid is a (posetal) strict monoidal category, where the monoid multiplication $\cdot$ acts as the categorical monoidal product $\otimes$ on objects. Given morphisms $[p \leq r]$ and $[q \leq z]$, their monoidal product is the unique morphism $[p \cdot q \leq r \cdot z]$, which exists by the monotonicity of monoid multiplication (3.3). The associator is then an identity natural transformation, as $(p \cdot q) \cdot r = p \cdot (q \cdot r)$. The monoid unit 1 acts as the categorical monoidal unit, so that the unitors are also identity natural transformations: $1 \cdot p = p$, $p \cdot 1 = p$. The triangle and pentagon equations are trivially satisfied (as is any valid equation between morphisms in posetal categories).

To add in a rigid structure and obtain the category **Preg**, we use the underlying pregroup's adjoints. Specifically, any object p will have a right dual $p \dashv p^r$ with cups and caps given by (3.4b)

$$\eta_p = [1 \leq p^r \cdot p] \qquad\qquad \epsilon_p = [p \cdot p^r \leq 1],$$

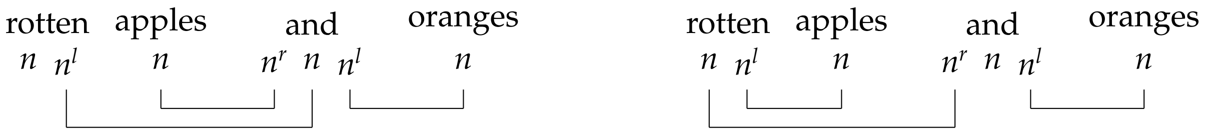
[3] **Preg** is *not* compact closed according to our definition, since it has no braiding and so is not symmetric monoidal. The presence of braiding would imply that the underlying monoid is commutative.

and left dual $p^l \dashv p$ with cups and caps given by (3.4a)

$$\eta_{p^l} = [1 \leq p \cdot p^l] \qquad\qquad \epsilon_{p^l} = [p^l \cdot p \leq 1].$$

The snake equations are trivially satisfied. That is, our categorical caps η and cups ϵ correspond to our ability to perform type expansions contractions of adjoints in the pregroup. □

While the category **Preg** is an obvious choice for categorifying our pregroup grammar, the fact that it is posetal means that all possible pregroup derivations between two given strings of types are identified in a single morphism. This is not desirable, since we would like to distinguish between different derivations that clearly give semantically distinct results. For instance, the structurally ambiguous phrase "rotten apples and oranges" yields two parses



representing "rotten [apples and oranges]" and "[rotten apples] and oranges" respectively.

The solution is to instead use the *free rigid category* over a poset of basic types (as described by Preller and Lambek in [PL07], where they use the term 'compact' rather than 'rigid')[4]. This category, which we will call $\mathbf{C}_F$, is essentially identical to **Preg** except that the hom-set between two strings of types is allowed to have multiple different reductions.

4.2.2 Pregroup → FVect functor

Now we describe the passage from pregroup reductions to a general string diagram. Recall our pregroup proof of the grammaticality of the sentence "John likes Mary" (3.7),

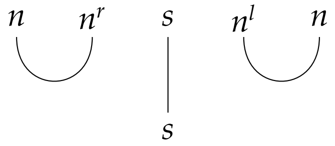
[4] Much of the DisCoCat literature skips over this point, and simply presents pregroup-based DisCoCat as a functor **Preg** → $\mathbf{FVect}_k$. This is a little sloppy and strictly speaking not correct.

$$\begin{tikzcd}
\text{John} & \text{likes} & \text{Mary} \\
n & n^r s n^l \arrow[d, "\epsilon_n \otimes 1_s \otimes 1_{n^l} \otimes 1_n"] & n \\
& 1 s n^l n \arrow[d, "1_{s n^l n}"] & \\
& s n^l n \arrow[d, "1_s \otimes \epsilon_{n^l}"] & \\
& s 1 \arrow[d, "1_s"] & \\
& s &
\end{tikzcd}$$

where we can now interpret each reduction step as a morphism in $\mathbf{C}_F$ built from cups and identities. Or, all at once, a morphism witnessing the grammaticality of this sentence is

$$\epsilon_n \otimes 1_s \otimes \epsilon_{n^l} : n \otimes n^r \otimes s \otimes n^l \otimes n \to s, \tag{4.2}$$

which diagrammatically is depicted as



i.e. essentially the underlink depiction of a pregroup reduction proof (3.8).

But the morphism $\epsilon_n \otimes 1_s \otimes \epsilon_{n^l}$ (along with the corresponding string diagram), which encodes the grammatical proof, can be interpreted in *any* rigid category. As previously discussed, we choose to interpret it in the category $\mathbf{FVect}_{\mathbb{R}}$[5]. More concretely, this act of 'interpretation in another category' consists of defining a strong monoidal functor[6]

$$F : \mathbf{C}_F \to \mathbf{FVect}_{\mathbb{R}}.$$

Defining this functor amounts to choosing an appropriate vector space for each atomic type.

Specifically, given a pregroup grammar with basic types n and s, we choose some vector spaces $N = F(n)$ and $S = F(s)$. To round off the basic types, we have $F(1) = I$ for the monoidal unit I. Now, recall that strong monoidal functors preserve duals

[5] For the purposes of NLP it makes sense to choose the target category of our functor to be $\mathbf{FVect}_{\mathbb{R}}$, but one of the advantages of DisCoCat-type models is that the target category can be any category that has the appropriate structure. For instance, the category of matrices over the semiring of Booleans also has a rigid structure. The original paper [CSC10] describes how choosing this to be the target category results in a Montague-style Boolean-valued semantics.

[6] In the original paper [CSC10], the unification of grammar and vector spaces was achieved by working in the product category $\mathbf{Preg} \times \mathbf{FVect}_k$. This was later recast as the functorial passage we describe here.

(lemma 2.4.4), and also that in $\mathbf{FVect}_{\mathbb{R}}$ not only are left and right duals equivalent due to braiding, but all objects are self dual. Hence we can choose

$$F(x^l) = F(x^r) = F(x)^* = F(x).$$

As per the requirements of a strong monoidal functor, the object map for F extends to compound types via

$$F(x \otimes y) = F(x) \otimes F(y).$$

Example. For instance, the word "likes" is usually typed as a transitive verb: $n^r s n^l$. Under our functor, this pregroup type maps to the vector space

$$F(n^r \otimes s \otimes n^l) = F(n^r) \otimes F(s) \otimes F(n^l) = F(n) \otimes F(s) \otimes F(n) = N \otimes S \otimes N.$$

This means that the meaning of "likes" in the sentence will be encoded as a vector in $N \otimes S \otimes N$.

For the morphism map, we simply translate the identities, cups, and caps from $\mathbb{C}_F$ to $\mathbf{FVect}_{\mathbb{R}}$ in the obvious way:

$$F(1_x) = 1_{F(x)} \qquad F(\epsilon_x) = \epsilon_{F(x)} \qquad F(\eta_x) = \eta_{F(x)}.$$

This fixes the morphism map for all possible morphisms.

Example. The type reduction of the sentence "John likes Mary", which was captured by the morphism constructed from identities and cups (4.2), will be mapped under F to

$$F(\epsilon_n \otimes 1_s \otimes \epsilon_{n^l}) = \epsilon_N \otimes 1_S \otimes \epsilon_N : N \otimes N \otimes S \otimes N \otimes N \to S$$

Finally, the overall meaning of a sentence is concretely calculated by applying the linear map obtained via the pregroup reduction, to the tensor product of the semantic vectors of the individual words.

Example. Suppose the words "John", "likes", and "Mary" have meanings represented by the vectors $\vec{John} \in N$, $\vec{likes} \in N \otimes S \otimes N$, and $\vec{Mary} \in N$. Then the meaning of the sentence "John likes Mary" is the vector

$$F(\epsilon_n \otimes 1_s \otimes \epsilon_{n^l})(\vec{John} \otimes \vec{likes} \otimes \vec{Mary}) = (\epsilon_N \otimes 1_S \otimes \epsilon_N)(\vec{John} \otimes \vec{likes} \otimes \vec{Mary}).$$

Depicted diagrammatically, this is precisely the state depicted in (4.1).

In the usual case that the domain of our functor is $\mathbf{FVect}_{\mathbb{R}}$ (or possibly $\mathbf{FVect}_{\mathbb{C}}$ for quantum computers!) our finite dimensional vector spaces form Hilbert spaces (i.e. have an inner product). Now that our sentences all live in the *same Hilbert space S*, we can take the inner product of any two sentences (or indeed pairs of smaller constituents with the same type) and compare them the same way we did with words in the example in section 4.1.

4.3 CCG-based DisCoCat

While DisCoCat was initially introduced using pregroup grammar as the domain syntax category, more generally one might define a DisCoCat-type language model to be any monoidal functor

$$C \to \text{word meaning}$$

where C is a biclosed category that captures the grammar. In particular, as argued in [CGS13, Gre13], DisCoCat also supports other categorial grammars such as standard Lambek calculus and Lambek-Grishin calculus. In this section we will examine a version of DisCoCat based on CCG which was recently formalised in [YK21].

This version of DisCoCat has a number of advantages beyond the original pregroup-based version. Firstly, as noted in subsection 3.5.3, pregroups are weakly equivalent to context-free grammars and hence not expressive enough for for natural language, whereas CCG was shown to be a mildly context-sensitive grammar (3.5.2). Thus the generative power of our DisCoCat model is increased. Secondly, there are many robust CCG parsers available which have been used to parse large corpora of real data - see, for example [CC07]. This is (at the time of writing) not the case for pregroup grammar. Thus, a CCG-based DisCoCat makes large-scale DisCoCat experiments on sentences of arbitrary grammatical structures possible for the first time. Indeed, in [YK21], a standard CCG parser [YNM17] was used to obtain DisCoCat diagrams for all sentences in the book 'Alice's Adventures in Wonderland'. Further, a web-based tool that allows the conversion of any sentence into a DisCoCat diagram is available at CQC's QNLP website[7].

The CCG-based DisCoCat model we discuss here comes in the form of a functor

$$\mathbf{CCG} \to \mathbf{FVect}_{\mathbb{R}},$$

[7] The tool is available at `https://qnlp.cambridgequantum.com/generate.html`. We will make extensive use of it in this thesis to generate parses.

where the domain category **CCG** is a biclosed monoidal category. Recall that $\mathbf{FVect}_{\mathbb{R}}$, being rigid, has a canonical biclosed structure given by

$$X \multimap - = X^r \otimes - \cong X \otimes - \qquad\qquad - \multimapinv Y = - \otimes Y^l \cong - \otimes Y. \tag{4.3}$$

Fixing a choice of $\multimap$ and $\multimapinv$ is important, as there are other possible choices (due to the problem of equivalence) which lead to slightly different results.

4.3.1 CCG as a biclosed category

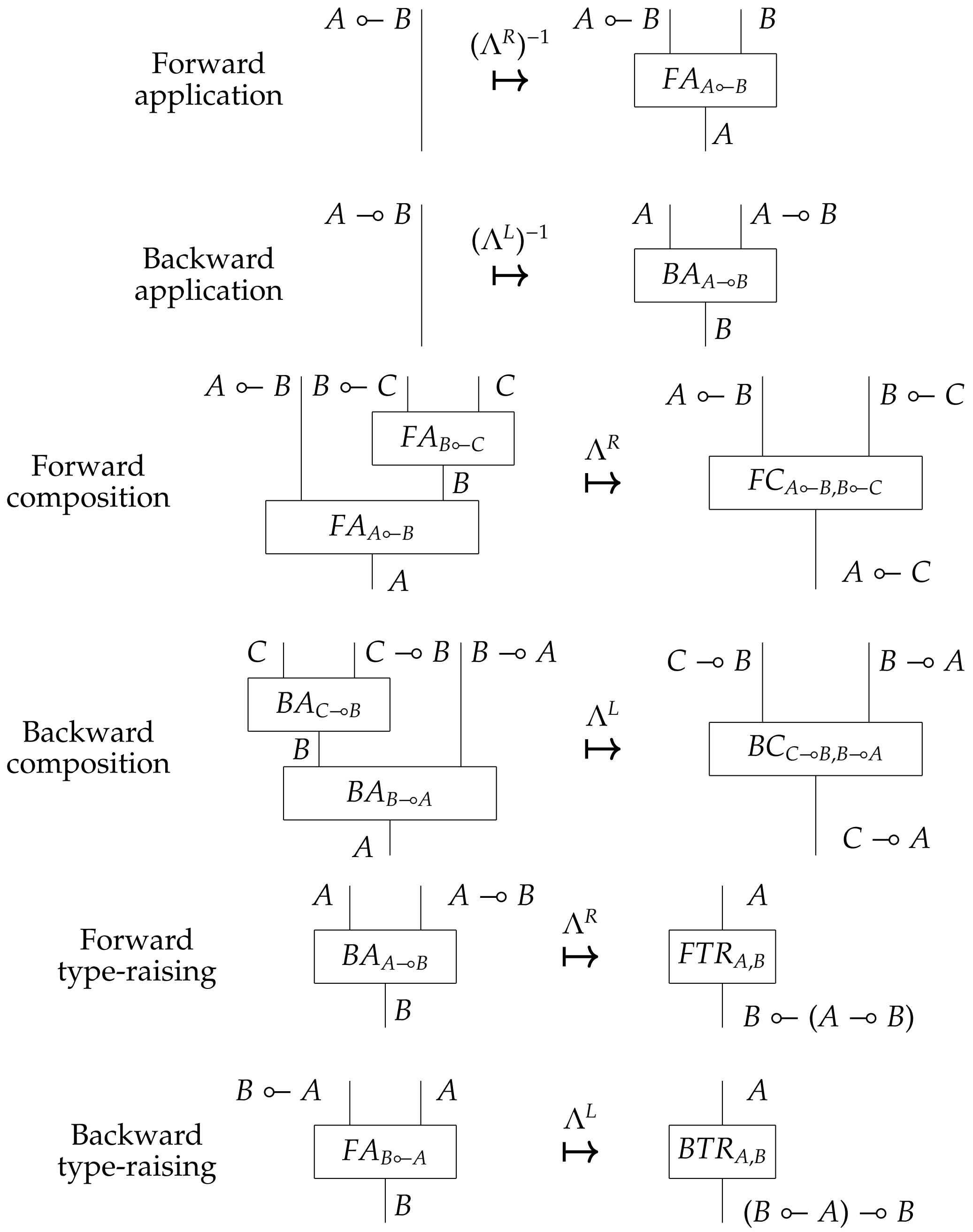
Given a specific CCG system which uses the five kinds of categorial rules described in subsection 3.5.2, we may interpret it as a biclosed category **CCG** in the following way[Gre13][8]. The objects of the category are freely generated from the set of atomic types. That is,

- for every atomic type A, there is an object A in Ob(**CCG**),
- for every type of the form A/B (resp. $A\backslash B$), where A, B are not necessarily atomic, there is some object $A \multimapinv B$ (resp. $B \multimap A$) in Ob(**CCG**),
- and for each sequence of types A, B permitted by the CCG, there is an object $A \otimes B$ in Ob(**CCG**).

As previously, the morphisms between types represent type deduction proofs. The key insight is that many of the basic categorial rules (i.e. all those described in subsection 3.5.2, *except* cross composition) exist naturally in any biclosed category and can emerge solely by currying and uncurrying identity morphisms. Hence any CCG derivation going from some string of types X to Y (using rules other than cross composition) exists as a morphism $X \to Y$ in a biclosed category freely generated over atomic types[YK21].

The morphisms corresponding to application, composition, and type-raising are shown below.

[8] This approach follows from the fact that Lambek Grammars can be viewed as biclosed monoidal categories, a fact observed in original work by Lambek [Lam88].



Note that the forward application and backward application morphisms are just the usual evaluation maps arising from the closed structure.

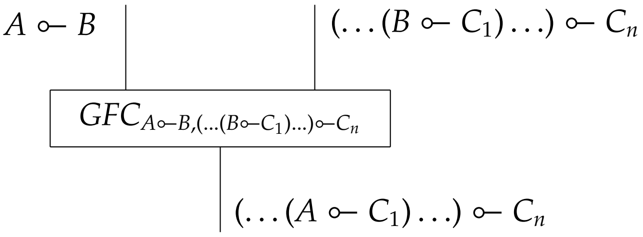
Obtaining the generalised composition requires a recursive construction[9]. Recall that each arity n gives a forward and backward composition rule

$$B^n_> \frac{X/Y \qquad (\ldots((Y/Z_1)/Z_2)\ldots)/Z_n}{(\ldots((X/Z_1)/Z_2)\ldots)/Z_n}$$

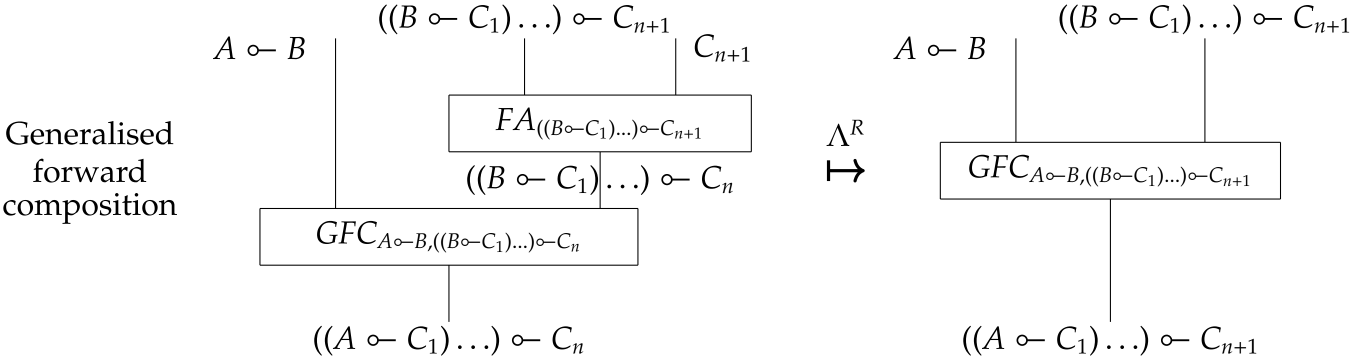
$$B^n_< \frac{(\ldots((Y\backslash Z_1)\backslash Z_2)\ldots)\backslash Z_n \qquad X\backslash Y}{(\ldots((X\backslash Z_1)\backslash Z_2)\ldots)\backslash Z_n}$$

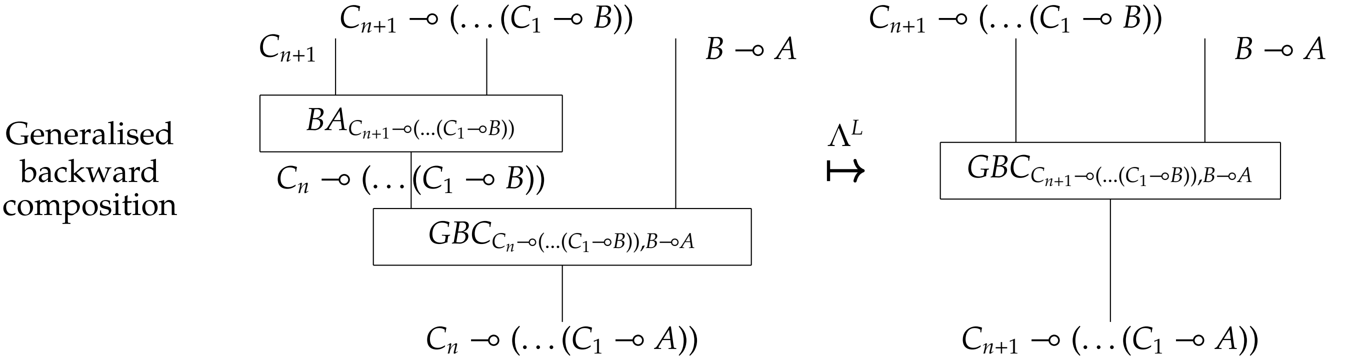
[9] The presentation we give here is somewhat original - the presentation in [YK21] of generalised composition contains errors and is incomplete.

where the $n = 1$ base case reduces to the basic forward and backward composition rules. Then, given the existence of arity n GFC maps

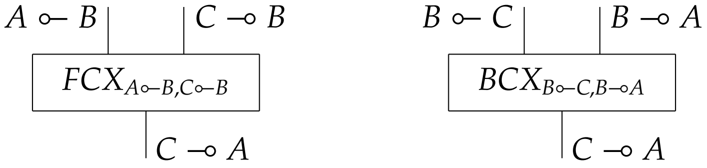

$A \multimapinv B$
$(\dots(B \multimapinv C_1)\dots) \multimapinv C_n$
$GFC_{A \multimapinv B,(\dots(B \multimapinv C_1)\dots) \multimapinv C_n}$
$(\dots(A \multimapinv C_1)\dots) \multimapinv C_n$

we can obtain arity $n + 1$ GFC maps (and analogously given arity n GBC maps we obtain arity $n + 1$ GBC maps) via:


Generalised forward composition
$A \multimapinv B$
$((B \multimapinv C_1)\dots) \multimapinv C_{n+1}$
C_{n+1}
$FA_{((B \multimapinv C_1)\dots) \multimapinv C_{n+1}}$
$((B \multimapinv C_1)\dots) \multimapinv C_n$
$GFC_{A \multimapinv B,((B \multimapinv C_1)\dots) \multimapinv C_n}$
$((A \multimapinv C_1)\dots) \multimapinv C_n$
Λ^R
$\mapsto$
$A \multimapinv B$
$((B \multimapinv C_1)\dots) \multimapinv C_{n+1}$
$GFC_{A \multimapinv B,((B \multimapinv C_1)\dots) \multimapinv C_{n+1}}$
$((A \multimapinv C_1)\dots) \multimapinv C_{n+1}$


Generalised backward composition
C_{n+1}
$C_{n+1} \multimap (\dots(C_1 \multimap B))$
$B \multimap A$
$BA_{C_{n+1} \multimap (\dots(C_1 \multimap B))}$
$C_n \multimap (\dots(C_1 \multimap B))$
$GBC_{C_n \multimap (\dots(C_1 \multimap B)),B \multimap A}$
$C_n \multimap (\dots(C_1 \multimap A))$
Λ^L
$\mapsto$
$C_{n+1} \multimap (\dots(C_1 \multimap B))$
$B \multimap A$
$GBC_{C_{n+1} \multimap (\dots(C_1 \multimap B)),B \multimap A}$
$C_{n+1} \multimap (\dots(C_1 \multimap A))$

Unlike these four types of rules, the cross composition rules do not occur naturally in biclosed setting. Hence, they need to be added to the generating set of the category as the following morphisms:



4.3.2 CCG → FVect functor

Now we define a 'closed monoidal functor'[10]

$$\mathbf{CCG} \to \mathbf{FVect}_{\mathbb{R}}$$

that enables the functorial conversion of CCG rules in a biclosed category into string diagrams involving caps and cups that can be interpreted in vector spaces. As before, we assign each basic type some vector space: $N = F(n)$, $S = F(s)$. The monoidal-ness of the functor imposes $F(I) = I$ and $F(x \otimes y) = F(x) \otimes F(y)$ as before. The closedness of the functor imposes among other conditions, that F respects the internal homs:

$$F(X \multimap Y) = F(X)^r \otimes F(Y) \qquad F(X \multimapinv Y) = F(X) \otimes F(Y)^l. \tag{4.4}$$

We note that this is, in essence, (3.9) - the equation that 'translates' categorial grammar types into pregroup grammar types. This essentially means that at the level of DisCoCat diagrams, the types in categorial grammar-based DisCoCat match the types in the DisCoCat based on the corresponding pregroup grammar.

Further, the closed monoidal condition yields that F respects currying. More concretely, for any diagram $d : A \otimes B \to C$,

$$\begin{aligned} F(\Lambda^L_{A,B,C}(d)) &= \Lambda^L_{F(A),F(B),F(C)}(F(d)) \\ F(\Lambda^R_{A,B,C}(d)) &= \Lambda^R_{F(A),F(B),F(C)}(F(d)). \end{aligned}$$

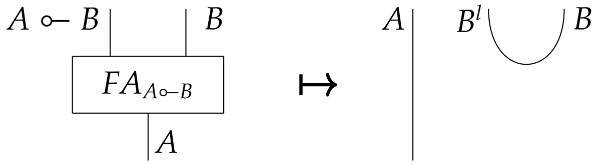
With these constraints in place, we can see how to convert all the rules (sans cross composition) into string diagrams in $\mathbf{FVect}_{\mathbb{R}}$. For instance, the backward application rule is mapped in the following way:

$$\begin{aligned} F(BA_{A\multimap B}) &= F((\Lambda^L_{A,A\multimap B,B})^{-1}(1_{A\multimap B})) \\ &= (\Lambda^L_{FA,FA^r\otimes FB,FB})^{-1}(F(1_{A\multimap B})) \\ &= (\Lambda^L_{FA,FA^r\otimes FB,FB})^{-1}(1_{F(A\multimap B)}) \\ &= (\Lambda^L_{FA,FA^r\otimes FB,FB})^{-1}(1_{FA^r\otimes FB}) \\ &= (\Lambda^L_{FA,FA^r\otimes FB,FB})^{-1}(1_{FA^r} \otimes 1_{FB}). \end{aligned}$$

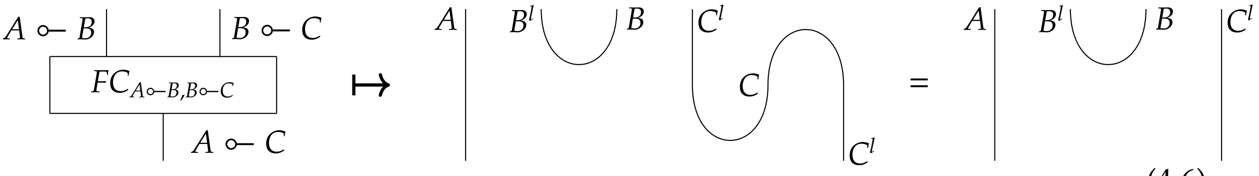
Here we show the translation of all the forward rules into DisCoCat diagrams. The backwards versions are obtained by simply reflecting the diagrams and replacing left

[10] We have skipped over the definition of what this exactly means.

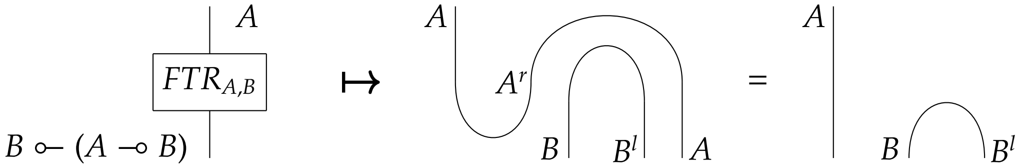
adjoints with right adjoints and vice versa.



(4.5)

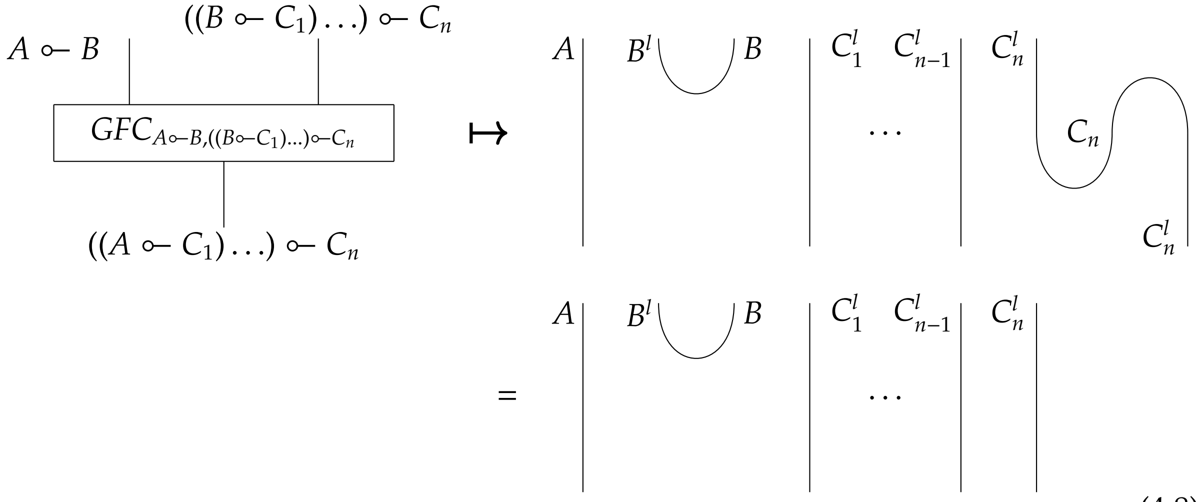


(4.6)



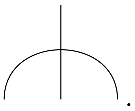
(4.7)

Note that for FTR, the $B \multimapinv (A \multimap B)$ object is mapped to $B \otimes (A \multimap B)^l = B \otimes (A^r \otimes B)^l = B \otimes B^l \otimes A$[11].

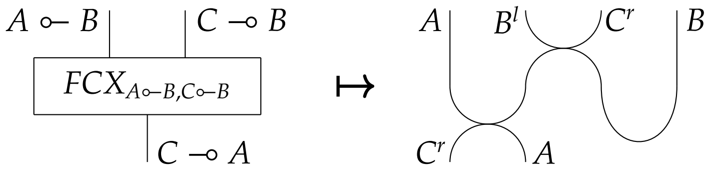


(4.8)

[11] This is a consequence of our choice of taking $(X \otimes Y)^l = Y^l \otimes X^l$, as per lemma 2.4.3, along with $X \multimap Y = X^r \otimes Y$, as per (4.3). However dual objects are not necessarily unique, and the fact that $\mathbf{FVect}_{\mathbb{R}}$ is braided means we could also have taken $(A \otimes B)^l = A^l \otimes B^l$ (this duality is witnessed by *crossed* cups and caps). In this case, in $\mathbf{FVect}_{\mathbb{R}}$ we would have $B \multimapinv (A \multimap B) = B \otimes (A \multimap B)^l = B \otimes (A^r \otimes B)^l = B \otimes A \otimes B^l$. Alternately and even more directly, we could take $V^l = V^r = V^* = V$ strict in $\mathbf{FVect}_{\mathbb{R}}$, which would yield $B \multimapinv (A \multimap B) = B \otimes A \otimes B$. In either such case, the currying operation $\Lambda^R_{A,A \multimap B,B}$ we use to construct the FTR morphism would have consisted of attaching some kind of crossed cap, and the FTR morphism itself would look like



Finally, we deal with cross composition. This rule introduces a crossing between the involved types. Since DisCoCat diagrams can live in a symmetric monoidal category, we can thus use the symmetry and map the cross composition morphism to the following string diagram:



(4.9)

We note that the use of unrestricted swaps in the semantics category would allow every possible permutation of words, which defeats the point of grammar and results in a maximally overgenerating model. However, in the context of DisCoCat and the syntax-semantic functor, it is the responsibility of the grammar to pose restrictions on how the semantic form is generated. Thus for our purposes we are only interested in the morphisms in the semantics category that are mapped onto from morphisms in the syntax category. We ignore the other morphisms in the semantics category (which would allow arbitrary permutations).

4.4 Outline of language circuits

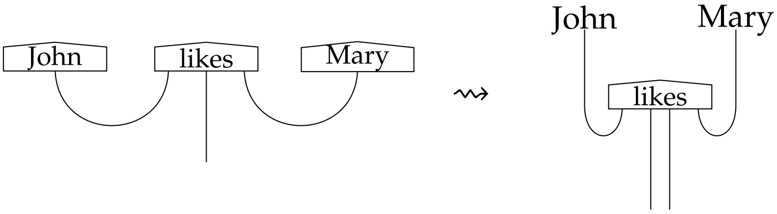
In this section we briefly outline the recently introduced notion of **language circuits** which are the main object of study of this project. Initially referred to by 'DisCoCirc' [Coe21], this framework builds upon the foundation of DisCoCat and represents a significant advancement of the diagrammatic approach to NLP.

We note that because language circuits are a very new development, many of the fundamental ideas behind them are still in flux. Indeed, questions regarding what exactly constitutes a language circuit, or how one arrives at this notion of language circuit in the first place, are still fairly open. As such, this section will contain mainly non-precise, philosophical discussion about what language circuits are/ought to be. More concrete and detailed discussion will be presented in chapter 5, which attempts to simultaneously motivate and construct the concept of language circuit.

4.4.1 The broad idea

Whereas DisCoCat provided a method for composing *words* so that we may obtain the meaning of the *sentence* they form, language circuits take this to the next level, by enabling the composition of *sentences* so that we may obtain the meaning of the *text*[12] that they form.

The way this is achieved is by *viewing sentences as input/output processes that modify the meanings of certain words*. Loosely, a DisCoCat diagram for the sentence "John likes Mary" will be transformed into the following language circuit:

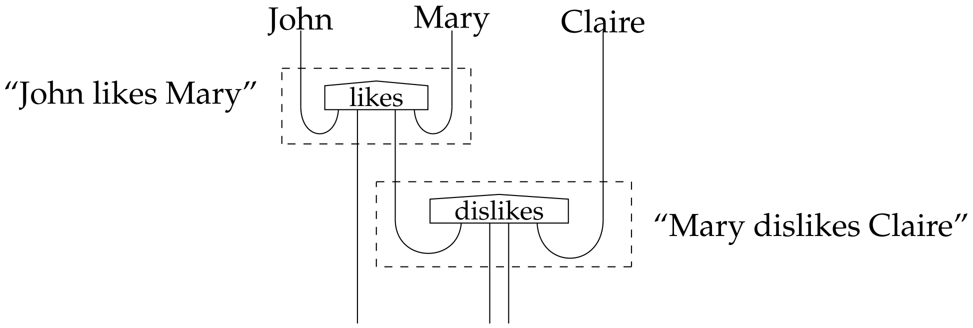


Note that in the circuit, the nouns/actors "John" and "Mary" each correspond to a wire (we call these **noun wires**). A sentence essentially manifests as a gate that acts upon and changes the meaning of the noun wires in the circuit. In this case the meaning of the sentence "John likes Mary" is captured by a single gate corresponding to the transitive verb "likes", which acts upon John and Mary. In this way, the meanings of words that are associated with noun wires are allowed to *evolve* as we proceed through the text, rather than remaining static. Furthermore note that the noun wires remain open at the end of the circuit, in contrast to DisCoCat where the output at the end of any diagram representing a sentence is always a single S type wire. The fact that these noun wires remain open at the end of the circuit is what allows for nontrivial composition with additional sentences.

To give an example, consider the 'text' consisting of the following two sentences

"John likes Mary.
Mary dislikes Claire."

We compose the sentence circuits to form a larger circuit that captures the overall meaning of the text:

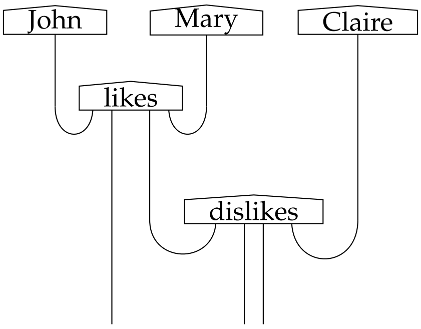


(4.10)

Once we have obtained such a circuit that represents the whole text, if we want to concretely calculate the meaning of the text as say, a semantic vector, we just plug

[12] For our purposes, we consider a 'text' to be a list of sentences.

'initial states' for each of the actors "John", "Mary", and "Claire" into the top of the circuit, at the appropriate noun wires.



What are the advantages of such a circuit representation of text? Firstly, such circuits bear some resemblance to quantum circuit diagrams - which is no coincidence given the string-diagrammatic framework was originally formulated to handle quantum information. A hope then is that language circuits would lend themselves to implementation on quantum computers, if and when appropriate hardware becomes available. In fact, DisCoCat was recently successfully implemented on quantum hardware [MTdFC20, LPM+21].

At a more theoretical level though, these circuits provide a 'higher-dimensional' representation of meaning which is potentially more natural and universal than the one-dimensional syntax of human language. That is, humans communicate using one-dimensional strings of symbols[13]. To quote [CW], "much of the complexity of grammar is due to the fact that human language is a one-dimensional vehicle for higher-dimensional content" - think of the bureaucratic conventions and stylistic features like SVO-order which vary across different languages. In expressing meaning through circuits rather than strings, we do away with the complexity and artificiality of these bookkeeping conventions, and *distill meaning down to an essential core*[14]. Indeed, we would expect the language circuit for "John likes Mary. Mary dislikes Claire" to look like (4.10) regardless if it the text was written in English or Latin.

4.4.2 Our approach to defining and motivating the concept of language circuit

What exactly is a language circuit? The best way to conceive of what we are attempting to do with language circuits seems to be: *we are finding ways to use circuit-like diagrams*

[13] This is true for natural language, but also true of other 'languages' like mathematics, much of which is also expressed through strings of symbols

[14] An analogy involving mathematics comes from the aforementioned example of the bifunctoriality condition (2.1), which manifests as a bookkeeping condition in linear symbolic form but is invisible in string diagrams, since it is already embedded within the logic of diagrams.

to represent the factual information expressed by sentences. Beyond this, there is as of yet no precise answer to this question. Historically, the paper [CW] and [Coe21] both present somewhat ad-hoc paths towards obtaining language circuits. In these papers, the springboard for the concept of language circuits is the pre-existing pregroup-based DisCoCat model, along with intuitions derived from previous work on the so-called 'internal wirings' (see section 4.5).

While I believe useful insights and intuitions may be garnered from these papers, I am somewhat sceptical about the strength of their motivations/derivations of language circuits. While language circuits clearly bear some relation to DisCoCat models, the concept of a 'language circuit' does not canonically 'fall out of' or obviously follow from the DisCoCat framework. While one may argue for a direct correspondence between, say, the pregroup-based DisCoCat model of section 4.2 and language circuits for very simple sentences, any such correspondence starts to break down for more complicated expressions, as I will argue in chapter 5.

The paper [CW] in particular relied heavily on providing internal wirings for words (which we also do for CCG-DisCoCat in chapter 7) as a means to explicitly construct a map from a pregroup-DisCoCat model (with a bespoke pregroup grammar system) to language circuits.

$$\text{DisCoCat} \xrightarrow{\text{internal wirings}} \text{language circuits}$$

However, I have always been of the opinion the internal wirings in this paper were *derived by working backward from some pre-existing notion of language circuits*, rather than being the thing that motivates language circuits in the first place.

It is my view that we should introduce language circuits by forgetting about DisCoCat and instead, unapologetically *assert* some basic, loose principles, and seeing what follows. These basic principles are that a language circuit for a given text consists of

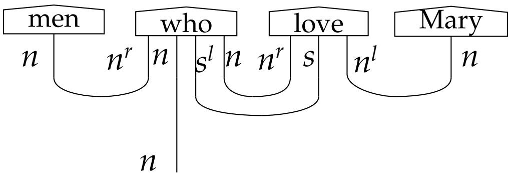
- a collection of *noun wires*, which are acted on/modified by
- *gates*, which capture the content of the sentences in the text.

With these principles in mind, we then proceed to address the main question of how to go about modelling sentences as gates in a circuit, doing so 'in the spirit of DisCoCat'. That is, our approach is still compositional-distributional, seeking to compose the constituents of sentences in ways guided by grammar. Indeed, basic cases will align closely with DisCoCat, such as the way we model verbs, adjectives, adverbs, etc (though again, we emphasise that at some point the analogy with existing DisCoCat models necessarily breaks down or ceases to be useful). The result of this approach and philosophy is the exposition of language circuits provided by chapter 5.

4.5 Outline of internal wirings

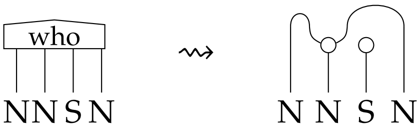
In this section we discuss the notion of 'internal wirings', which predate the idea of language circuits and have recently been thought of as potentially offering a bridge between DisCoCat and DisCoCirc. The main idea behind internal wirings is that we can provide *additional internal structure* to the word states in DisCoCat, through the use of wires and spiders. At a theoretical level, the extra structure provided by such internal wirings gives us more room to play around with our DisCoCat diagrams - though there may also be practical reasons for breaking down word states by providing additional internal structure [KS14]. As with the language circuits of the previous section, the exact role to be played by internal wirings is still unsettled. As such, this section will also just give a broad sketch of the main ideas. More detailed work can be found in chapter 7, in which we explicitly provide a catalogue of internal wirings that allows one to obtain language circuits from a certain CCG system.

The first time internal wirings are introduced for word states is in [SCC13]. Here, internal structure was provided for 'subject relative pronouns' and 'object relative pronouns' in the context of the pregroup-based DisCoCat. As an example of the former, the pronoun "who" in the phrase "men who love Mary" is a subject relative pronoun, which receives pregroup type $n^r n s^l n$. The DisCoCat diagram for this noun phrase is



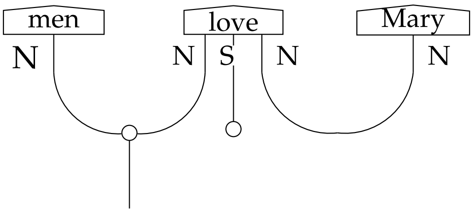
(4.11)

Recall that in passing to $\mathbf{FVect}_{\mathbb{R}}$, we replace the types n, n^l, n^r with N and s, s^l with S. The paper then proposed that one can replace the "who" state with the following morphism built with wires and spiders:

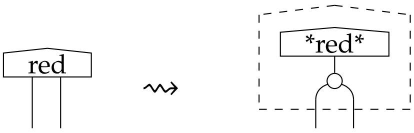


(4.12)

This extra structure allows for additional graphical manipulation of the DisCoCat diagram - plugging (4.12) into the diagram (4.11) and simplifying, we obtain the following simplified form:



While for functional words like the aforementioned subject relative pronoun, the entire meaning of a word may be satisfactorily captured by wiring alone, we can also have cases where internal wiring provides extra structure but also retains some 'black-boxedness'. For instance, in [Coe21] the following internal structure was proposed for adjectives



Note this wiring essentially turns adjectives into the spider-form we see later (section 5.3).

Many other explorations and ideas involving internal wiring were proposed (see for instance [SCC14, CLM18, Coe21]. Mostly recently, a systematic catalogue of internal wirings was proposed in the [CW], where the internal wirings were intnded to provide a passage from DisCoCat to language circuits.

Chapter 5

Language circuits for simple sentences

Natural languages like English are devilishly complicated - so to begin with, we first consider a restricted, basic fragment (corresponding to what we will call 'simple sentences'). In section 5.1, which forms the meat of this chapter, we will define the simple sentence fragment, and go about motivating our choices regarding how to model each item of this fragment as a circuit component. The motivation comes from existing grammatical frameworks - the previously discussed typelogical grammars, as well as dependency grammar. In section 5.2 we discuss the behaviour of the circuit components we have introduced, and describe a typing system for them. Section 5.3 describes how these components can be represented in an alternative 'spider-form' - this will be used in our internal wirings in chapter 7.

We note that though this chapter deals solely with the 'simple sentence fragment', in chapters 6 and 7 we will go beyond this fragment with the introduction of some extra parts.

5.1 Modelling simple sentences

The fragment of English we investigate in this chapter is restricted, but nevertheless incorporates many of the key parts of speech and contains much of the open word classes. Specifically, we consider the fragment built from the following lexical categories:

- *nouns* (not including pronouns)
- some *determiners* (e.g. "the" and "a"), which we will essentially ignore and merge with nouns to form noun phrases. That is, we treat NPs like "the flowers" or "a cat" as constituting a singular noun state. We will generally avoid quantifiers ("all", "some").

- *verbs* that take noun phrases as inputs, i.e. intransitive, transitive, and ditransitive verbs with arguments restricted to NPs. We generally stay away from 'linking verbs' that take can take adjectival phrases, verbal phrases, or clauses as inputs, and also ignore auxiliary verbs for now.

- *adjectives*

- *adverbs*, which we take to mean words that directly modify verbs

- *intensifiers*, which we take to be words that modify adverbs or adjectives (though we noted that traditionally, these are considered to be a kind of adverb).

- *adpositions* that connect a sentence with a noun phrase. This excludes adpositions that connect two sentences - "Bob drinks until Alice arrives" - which might better be classified as a conjunction. This also excludes adpositions that connect two noun phrases - "the building by the sea".

A main characteristic of this fragment is that the grammatical 'complements' are generally always noun phrases, rather than clauses or other kinds of phrases[1]. For instance, consider the adposition/subordinating conjunction "until". In our fragment we can have "Bob drinks beer until dawn", but not "Bob drinks beer until Alice arrives", since in the former sentence "until" joins the main sentence "Bob drinks beer" with a noun phrase "dawn", whereas in the latter sentence it joins the main sentence with another sentence "Alice arrives". In a similar vein, note our exclusion of copular or linking verbs that take as a complement something that is not a noun phrase - examples of forbidden sentences are "Bob *is* happy about the game", "Bob *dreamt* that he won", "I *like* going to the beach", "the distinction *became* clear".

We will roughly define a **simple sentence** to mean a grammatical sentence that consists only of the parts of speech we listed in the previous subsection . We will call the fragment of English we are considering in this chapter the **simple sentence fragment**.

A consequence of this definition should be that *simple sentences only contain one verb*[2]. One can argue that a sentence like "I enjoy living in Oxford" 1) falls into this fragment and 2) seemingly contains two verbs, "enjoy" and "living". However, there are at least two ways to counter this claim. Firstly, we could say that "living" here is a noun rather than a verb ("living" as in "the act of living"), which is a perfectly grammatically valid answer. Alternately though, if we want to view "living" as a verb, we could argue

[1] When we expand our fragment beyond simple sentences in section REF by introducing some extra parts of speech, we will allow a few extra kinds of complements, but still *no clausal complements*.

[2] I am unsure how one would prove this is the case, but it seems to hold intuitively.

that this sentence actually does *not* fall into the simple fragment, because then the verb "enjoy" links "I" with a verb phrase complement "living in Oxford". Recall then that we excluded such instances of linking verbs from our fragment.

As we proceed with modelling the components of simple sentences and examining some motivating examples, two points that I want to contend are the following.

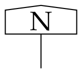
Claim 5.1.1. *1. We do not/should not rigorously follow existing typelogical grammar frameworks to decide how language circuits should look, and*

2. if anything, dependency grammar more closely reflects what we are trying to achieve.

Throughout this chapter, we will rely on the previously mentioned CQC web tool (`https://qnlp.cambridgequantum.com/generate.html`) to generate CCG parses of sentences and phrases. Meanwhile, spaCy (specifically, the displaCy visualizer at `https://explosion.ai/demos/displacy` with the English-en_core_web_sm v3.1.0 model) will be used to generate dependency parses.

5.1.1 Nouns

In general, nouns will be modelled as states



that are plugged into the noun wires.

There is more to this story, however. In fact, I think that one of the fundamental conceptual questions that is yet to be fully addressed in the theory of language circuits is:

How do we assign noun wires?

The first thing to note regarding this question, is that *noun wires do not really correspond to nouns, but to 'entities'*. Consider for instance:

"Bob drinks at the black bar. Alice drinks at the red bar."

Here the "black bar" and "red bar" are distinct entities, despite both sharing the same underlying noun "bar". It would not make sense for "black bar" and "red bar" to end up on the same wire labelled by the noun "bar". To borrow from discourse representation theory [GBM20] (which shares many similarities with our theory of language circuits), given a text, what we need is a universe of 'discourse referents' which represent the *objects under discussion*. These 'objects under discussion' should index our noun wires. So, in the case of the black and red bars, a solution may be to have two wires - one for the black bar and one for the red bar.

This view of noun wires as representing entities opens up a whole trove of other questions. Consider, for instance the following tricky scenario, where we initially treat two discourse referents in the text (e.g. Clark Kent, Superman) as separate entities, but later find out they are actually the same entity (Clark Kent is Superman in disguise).

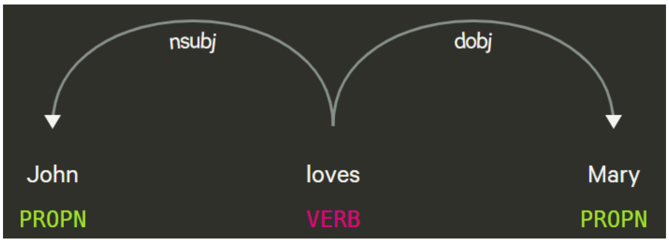
Another conundrum revolves around the question 'which objects/nouns deserve to be assigned noun wires?' The simplest answer to this question would be to go with the obvious choice - all nouns constitute an 'object under discussion' and are worthy of a noun wire. However, recall that noun wires correspond to words with dynamic meanings that change as the text progresses, in stark contrast to all the other words, which are gates that are presumably fixed. So, if we are dealing with a sentence like "Bob likes swimming", it would be strange for "swimming" to be privileged with a noun wire. Its meaning is fairly static - indeed we would happily treat the verb "swims" as a static gate. On the other hand, if our text is an article explaining the act of swimming - "Swimming is an individual or team racing sport that requires the use of one's entire body to move through water . . . " - it seems sensible for "swimming" to have a dynamic meaning. As another example, suppose we have a text "Five men walk along the road" - it does not make sense for the "five men" to constitute one entity. These examples should show that the simplistic idea that 'every noun is an entity' breaks down in some scenarios.

The discussion in this subsection lays out some of the ambiguities regarding deciding how noun wires ought to work. The examples we have given demonstrate that there is still some thinking to be done regarding this point. For our current purposes though, we choose to avoid these complications, and leave sorting them out to future work. In this thesis, we will limit our noun wires to to clearly defined individual entities - named people like "Alice" and "Bob", or physical objects like "the bar" or "flowers" of which there is a single copy (rather than having both "red bar" and "black bar").

5.1.2 Verbs

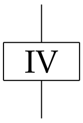
The simplest sentences we consider consist of a single verb and the nouns that serve as its arguments. In many ways, *a verb forms the basic core of a sentence*. Hence, we will begin our consideration of circuit components proper with verbs.

The key idea here will be that *verbs should be modelled as gates that act on some number of noun wires*. We have seen in typelogical grammars (section 3.5) the idea that verbs can be viewed as entities that accept some number of nouns as inputs in specific argument slots and return a well-formed sentence. For instance, recall the CG type of a transitive verb $(S\backslash N)/N$, or the pregroup type of $n^r s n^l$. The notion that a verb has some fixed set of noun arguments (subject, direct object etc.) is also reflected in dependency grammar.

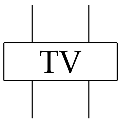


These observations motivate the following circuit representations for verbs, which are distinguished by their valency:

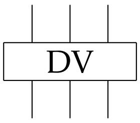
- intransitive verb



- transitive verb



- ditransitive verb



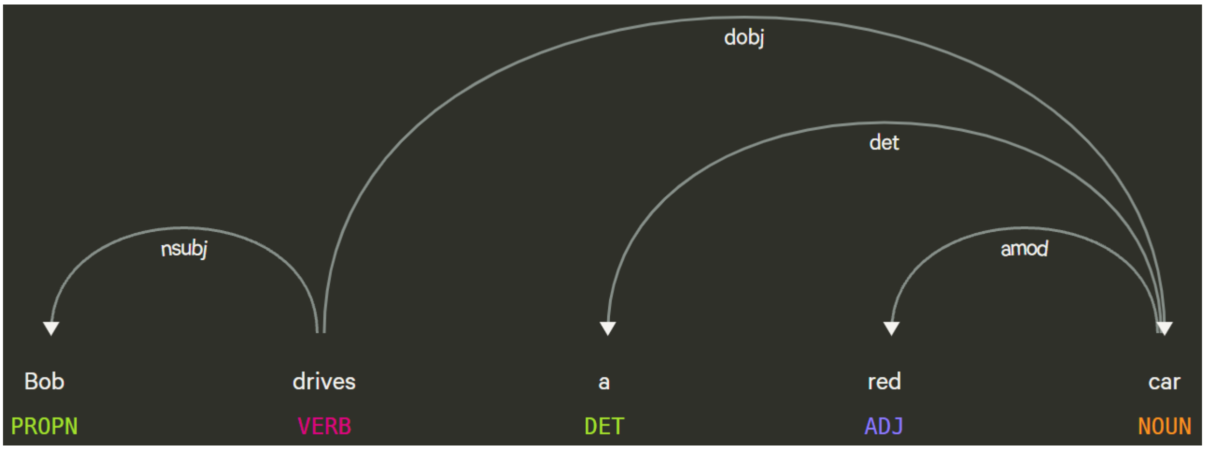
We adopt the convetion that the left-most noun wire always corresponds to the subject of the verb, the second from the left corresponds to the (direct) object, and the third wire corresponds to the indirect object (when such wires exist).

For English, the valency of verbs usually stops at $n = 3$, but in principle there is no reason to stop here. One may conceive of a race of aliens who can communicate using verbs of even higher valency.

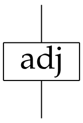
In support of claim 5.1.1, an attractive feature of dependency grammar is that, as mentioned in section 3.6, the verb is the structural center of the clause. This seems to coincide with the privileged position that verbs have in our theory of language circuits as the core component of a gate.

5.1.3 Adjectives

Adjectives are also fairly straightforward. A standard CG typing for an adjective like "red" in "red car" is N/N (or nn^l in pregroup form), indicating that it acts on a noun to return a modified noun. Similarly, in dependency parsing we see that adjectives simply serve to modify a noun (here via an 'amod' dependency).



Hence, adjectives are gates modifying single nouns, and so their circuit components look like intransitive verbs

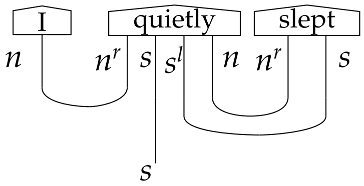


It is worth noting that in English, some adjectives are 'predicative', meaning they come after the noun their modify, often linked via a copula: "Jack is handsome". (The form in "red car" is called attributive). A CG parsing of this sentence may no longer assign a 'noun to noun' type for the adjective. Nevertheless, in these cases we think the most natural choice is still to view the adjective as a single noun gate acting on "Jack". The copula "is" will simply be made to vanish - refer to the discussion on "is" in subsection 6.2.1.

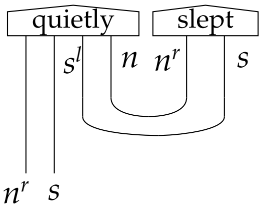
5.1.4 Adverbs

Whereas verbs act on and modify nouns, the key idea for adverbs is that *an adverb is something that modifies a verb*.

In a typelogical grammar, an adverb for an intransitive verb might be typed $S\backslash N/(S\backslash N)$, (or $n^r s s^l n$ in pregroups), reflecting that it takes in an intransitive verb $(S\backslash N)$ on the right to return another intransitive verb $S\backslash N$. The CG/PG proof of a sentence like "I quietly slept" is captured in the DisCoCat diagram

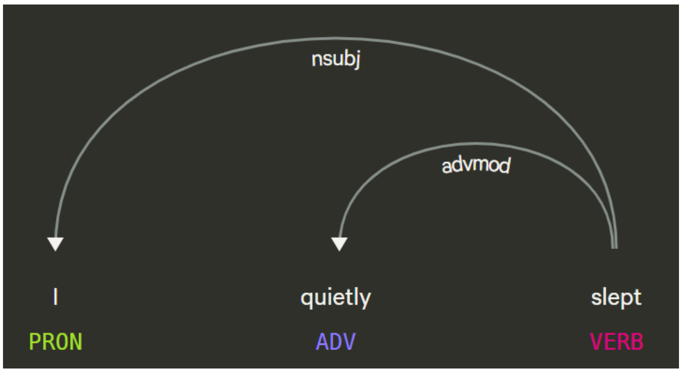


If we consider the pregroup/CG cancellation between "quietly" and "slept" to have occurred first (which is indeed what happens in the CG proof), ignoring for now the cancellation between "I" and "slept",



then we see that the adverb "quietly" can be viewed as taking in the intransitive verb "slept" and returning a new intransitive verb "quietly slept" (CG type $S\backslash N$)[3].

Meanwhile, the dependency grammar parse of the sentence "I quietly slept"



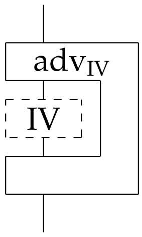
(5.1)

also indicates that the adverb "quietly" simply modifies the verb "slept", and additionally that "slept" then just acts as the intransitive verb it normally is.

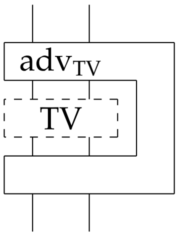
So, based on these observations we posit that adverbs should be a a component that takes in a verb of valency n as input and return a verb of the same valency as output. We again distinguish between adverbs based on the valency of the verbs they accept.

- adv_{IV}

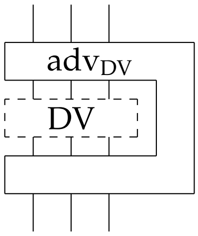
[3] So while for the case of verbs, we had to replace the S type with some tensor product of N types to properly turn them into some kind of language circuit input/output gate, for the case of adverbs we did not have to make any adjustments. The view of adverbs as endomorphisms on verbs is inherent in the PG/CG formalism.



- adv$_{\text{TV}}$

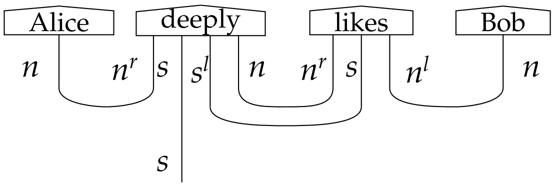


- adv$_{\text{DV}}$

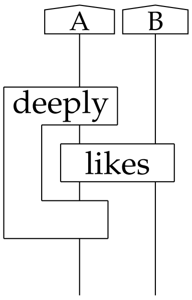


In circuit jargon these kinds of components are referred to as **combs**. Their category-theoretic foundation is explored in [Rom20].

We now note an example in which our above proposal for the shape of adverbs *disagrees* with established categorial grammar. This is the first of a number of examples which support claim 5.1.1. Consider the sentence "Alice deeply likes Bob", in which "deeply" is an adverb for a transitive verb, and which has the following CCG parse (given by the web tool).



Note that in this DisCoCat diagram, "Bob" is fed directly into the verb "likes". Topologically, this DisCoCat diagram more closely resembles a circuit with the following connectivity[4]:

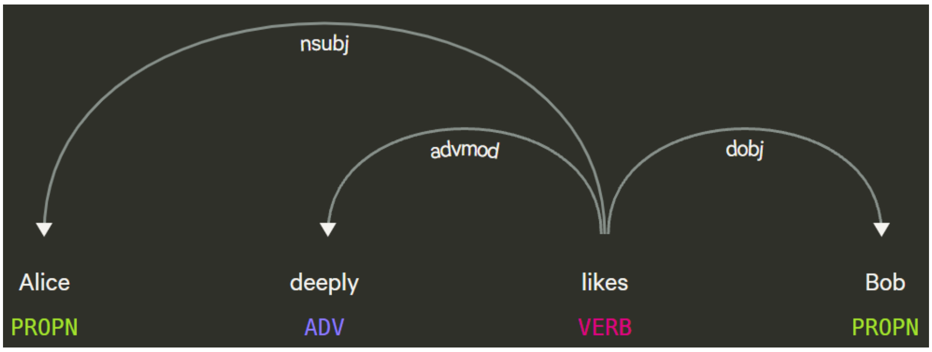


(5.2)

This however, seems somewhat unnatural at a theoretical level - why should the "deeply" comb only modify Alice's noun wire? This example (and those in the following subsections 5.1.5, 5.1.6) show that we probably should not faithfully follow pre-established categorial grammar when designing our theory of language circuits and circuit components.

The fact that the last input of "likes" remains free here, is an artefact of the fact that categorial grammars were designed to act as a bookkeeping system for linear word order. More precisely, in this case, the parser has deemed it most efficient/grammatically natural to merge "likes Bob" into an intransitive verb-like constituent.

By contrast, the spaCy parse for this sentence



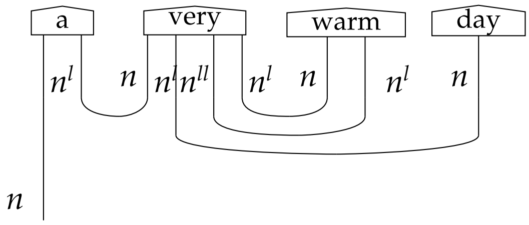
looks *essentially the same* as the parse (5.1), except the intransitive verb has been swapped out for a transitive one. Both dependency parses reflect the same principle that the adverb *simply modifies* the ditransitive verb through an 'advmod' dependency.

5.1.5 Intensifiers

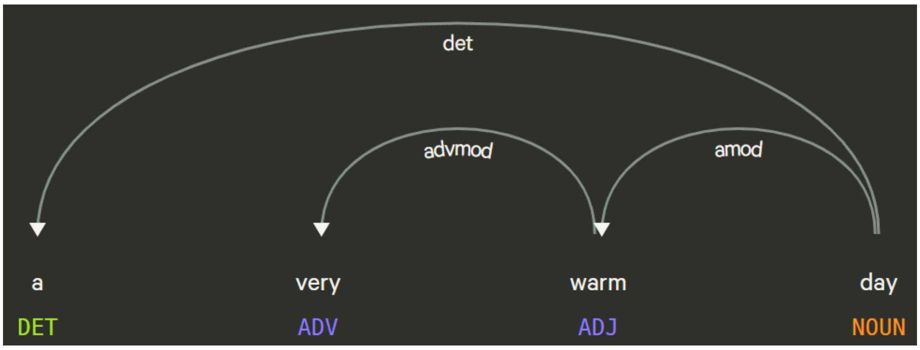
For intensifiers, which we defined to be words like "very" that modify adverbs or adjectives, the natural assertion is that they should be *higher-order combs that modify adjective/adverb combs*.

[4] Especially when replacing the S type in the DisCoCat diagram with $N \otimes N$ wrapped up in a wrapping gadget.

We can find the following examples to motivate this idea. For the case of intensifiers that modify adjectives, consider the sentence "Today is a very warm day", where "warm" is an adjective that is modified by the intensifier "very". The web tool gives the following CCG parse for the noun phrase "a very warm day":



We see that the adjective "warm" is fully absorbed by "very" as an input. The composite "very warm" then acts as an adjective (i.e. it has pregroup type nn^l) that takes in the noun "day" and returns a noun. Meanwhile, the dependency parse for this phrase



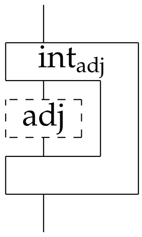
(5.3)

indicates also that the intensifier (here tagged as an adverb) simply modifies the adjective "warm" via an 'advmod' dependency.

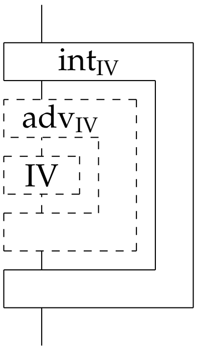
For examples involving intensifiers acting on intransitive verbs, one can look at the parses for the sentences "Bob rather quickly ran", or "Bob ran rather quickly". Here one will see something similar, where the intensifier fully absorbs the adverb, returning a composite that acts again as an adverb.

We thus propose the following circuit components.

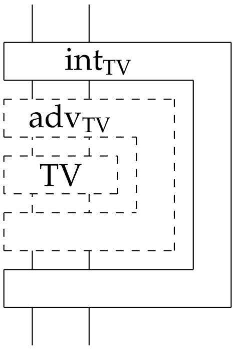
- int_{adj}



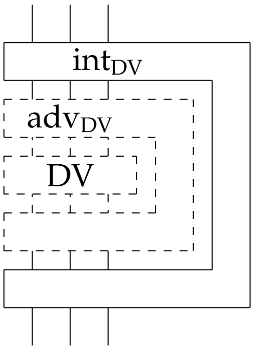
- int_{IV}



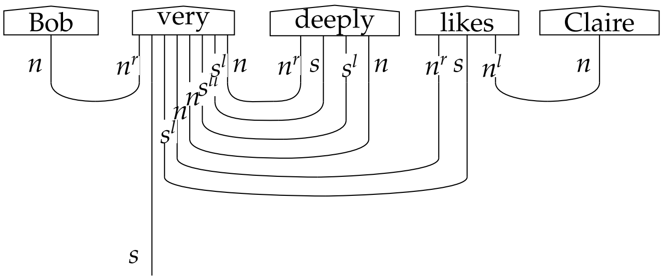
- int$_{TV}$



- int$_{DV}$

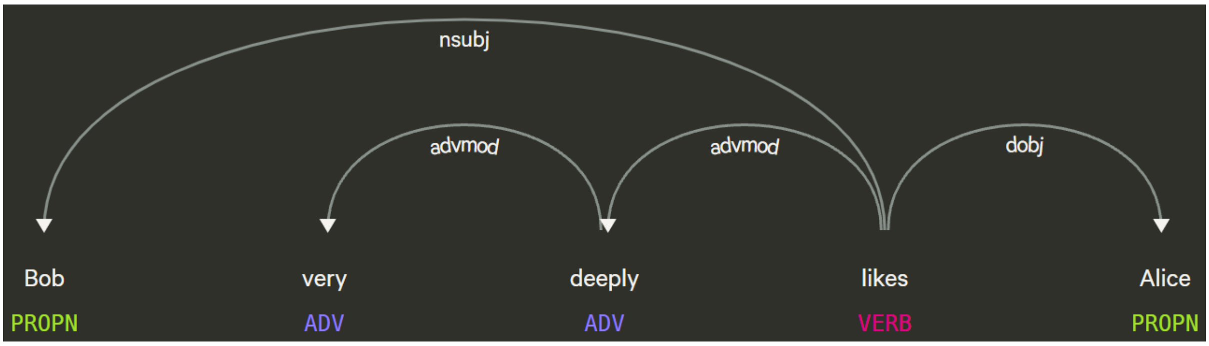


As with adverbs in the previous subsection, the fairly natural proposal for intensifiers above, fails to cohere with categorial grammar for transitive and ditransitive verbs. Looking at the CCG parse for "Bob very deeply likes Claire",



we see a similar issue to the case of adverbs, where the connectivity of the DisCoCat diagram obtained from a CCG parse does not match our desired connectivity. Here the verb “likes” is not ‘fully consumed’ by “very” (although “very” does fully consume “deeply”). This means the composite “very deeply” does not quite match the expected type of an adverb.

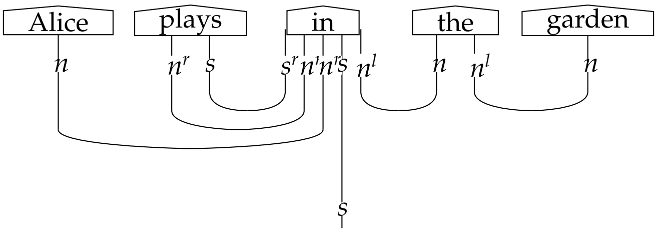
By contrast, dependency parsing once again remains more consistent with our proposal - “very” just acts on another adverb through an ‘advmod’ dependency just like in (5.3).



5.1.6 Adpositions

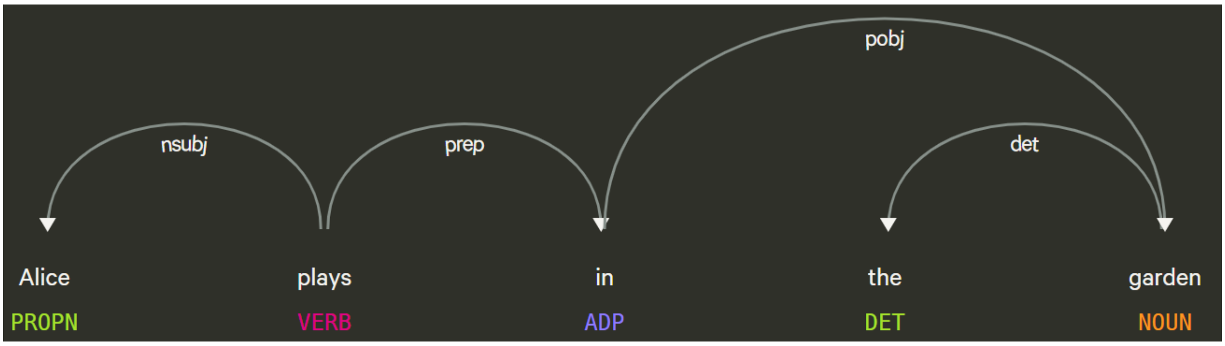
The last ingredient we will include in this simple sentence fragment are adpositions that connect a sentence with a noun phrase. This kind of adposition can be viewed as *acting on an input verb by increasing its valency by 1*. That is, it will be a comb that accepts a verb on n noun wires and returns a new verb on $n + 1$ noun wires.

To justify this idea, consider the CCG parse of “Alice plays in the garden”.



Here, the adposition “in” absorbs the intransitive very “plays” and returns a transitive verb “plays in” (pregroup type $n^r s n^l$).

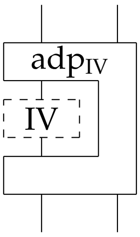
From the point of view of dependency grammar, this interpretation of adpositions is more slightly more tenuous but still supported.



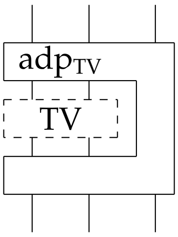
Here, we see that the adposition serves as a kind of intermediate node that links the verb “plays” with an additional noun argument, the ‘pobj’ (prepositional object).

Thus we propose the following circuit components.

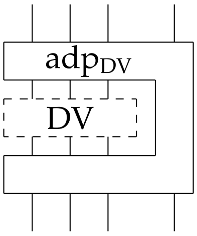
- adp_{IV}



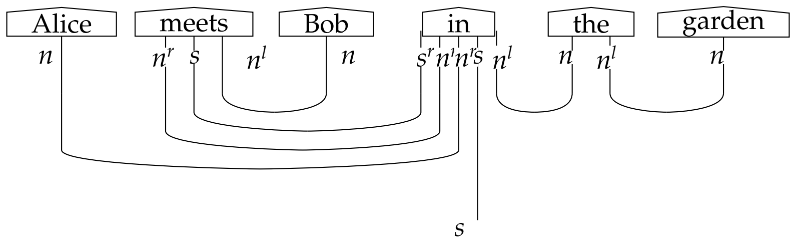
- adp_{TV}



- adp_{DV}



As with adverbs and intensifiers, our proposal does not match with the categorial parses for transitive and ditransitive verbs. In the sentence “Alice meets Bob in the garden”,



we get connectivity issues since the adposition "in" does not fully absorb the transitive verb "meets". Instead "Bob" is fed directly into "meets", instead of into "in".

5.1.7 A word on discrepancies with CCG parses

In subsections 5.1.4, 5.1.5, and 5.1.6, we have given a number of examples to show that our idea of modelling various grammatical features as 'functions' that fully absorb some input to produce some output, does not always coincide with the CCG typings provided by the parser. These examples are intended to support claim 5.1.1.

In such cases, as a counterargument to this point, one might provide the following argument. The statistical aspects of parsers means that for any sentence, there may be a large number of alternate possible parsings that are not shown, since they are deemed suboptimal. One of these discarded parsings may match our desired connectivity, and hence fully support our language circuit theory!

While this is true, I think that if we are picking and choosing the parsings, then at a methodological level we are not really *starting from typelogical grammar to decide what language circuits look like*, but rather *using our internal idea of what connectivity we want as the starting point*.

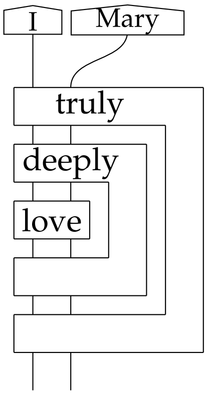
Indeed, if by some method or other we always enforce PG or CG typings for syntactic categories that matches our language circuit theory, we *are not really using the structure of the typelogical grammar*, but rather simply expressing our functional view of syntactic categories through that type-grammatical formalism. Indeed, this is what was done in the original internal wiring paper [CW], and what we do in chapter 7 - we cook up our own PG/CCG typing system that allows us to realize the connectivity between the components that we desire.

5.2 Comb inputs and typing

Based on the categorisation provided in the previous section, combs seem fairly rigid in terms of the kinds of inputs they accept. However, one thing to note is that these combs do not necessarily *exactly* take in the kinds of inputs that were depicted as dashed boxes in the previous diagrams. There is a certain degree of freedom here with respect to their inputs.

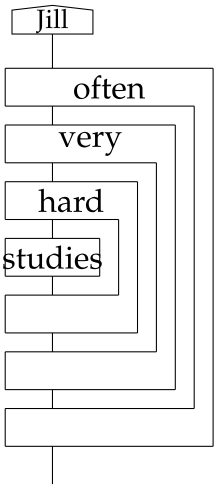
To make this point clearer, consider a few examples.

1. Stacking: intuitively it seems like it should be possible to *stack* certain combs together. Consider a sentence with stacked adverbs, like "I truly, deeply love Mary". Here "truly" and "deeply" are both adverbs modifying the verb "love", which intuitively yields a language circuit like

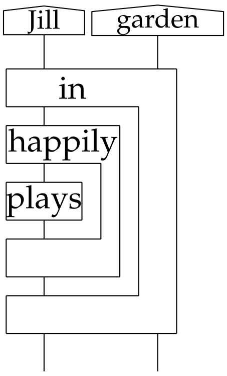
   The point being made here is that in the previous diagrams we depicted adverb combs as accepting a single, bare/unadorned verb, but in this example, the "truly" adverb comb accepts as input an adverb-verb composite "deeply love".

   As another example of stacking, consider the sentence "Jill often studies very hard", which leads to adverbs stacked together with an intensifier. Specifically, "often" and "hard" are both adverbs modifying "studies", and "hard" is itself modified by the intensifier "very". This intuitively could result in a circuit like

   where there is an intensifier sandwiched between two adverbs.

2. Our adpositions provide another clear cut example of combs accepting inputs that do not exactly follow the dashed boxes in the diagrams. An example of an adposition that accepts a bare verb would be “Jill plays in the garden”, where the adposition “in” takes the verb “plays” as input. However, the sentence “Jill happily plays in the garden” is clearly grammatical, and intuitively gives the circuit



i.e. the adposition “in” accepts a verb-adverb composite “plays happily”.

One way to account for these phenomena is that the dashed boxes in the previous diagrams just indicated the *type* of the inputs of a given comb (in the sense of type theory). At the level of typing, a bare intransitive verb like “plays” and a verb-adverb composite like “happily plays” both have the type of an intransitive verb, and so both can fit into the adp_{IV} “in”. Similarly, in the earlier example “Jill often studies very hard”, the “studies very hard” composite consisting of intensifier-adverb-verb has the type of an intransitive verb, and so can be inserted into the adv_{IV} “often”.

Thus, the observations of this subsection motivate the introduction of a type system for our circuit components, which serve to more rigorously characterise their behaviour in terms of the kinds of inputs and outputs they have. The type system essentially has one basic type $\mathtt{N}$, and $\times$ and $\to$ operations. In the notation $\times$ takes precedence over $\to$, i.e. $\mathtt{N} \times \mathtt{N} \to \mathtt{N} \times \mathtt{N} = (\mathtt{N} \times \mathtt{N}) \to (\mathtt{N} \times \mathtt{N})$.

- **nouns**
  Type: $\mathtt{N}$

- verbs

  - **intransitive verbs**
    Type: $\mathtt{IV} := \mathtt{N} \to \mathtt{N}$

- **transitive verbs**
  Type: $\texttt{TV} := \texttt{N} \times \texttt{N} \to \texttt{N} \times \texttt{N}$

- **ditransitive verbs**
  Type: $\texttt{DV} := \texttt{N} \times \texttt{N} \times \texttt{N} \to \texttt{N} \times \texttt{N} \times \texttt{N}$

- **adjectives**
  Type: $\texttt{adj} := \texttt{N} \to \texttt{N}$

  (Note the type is the same as $\texttt{IV}$, reflecting the fact that they look the same as circuit components. This ends up being slightly problematic - see for instance the intensifier for adjectives listed below.)

- adverbs
  - **adverb$_{\textbf{IV}}$**
    Type: $\texttt{adv}_{\texttt{IV}} := \texttt{IV} \to \texttt{IV} = (\texttt{N} \to \texttt{N}) \to (\texttt{N} \to \texttt{N})$

  - **adverb$_{\textbf{TV}}$**
    Type: $\texttt{adv}_{\texttt{TV}} := \texttt{TV} \to \texttt{TV} = (\texttt{N} \times \texttt{N} \to \texttt{N} \times \texttt{N}) \to (\texttt{N} \times \texttt{N} \to \texttt{N} \times \texttt{N})$

  - **adverb$_{\textbf{DV}}$**
    Type: $\texttt{adv}_{\texttt{DV}} := \texttt{DV} \to \texttt{DV} = (\texttt{N} \times \texttt{N} \times \texttt{N} \to \texttt{N} \times \texttt{N} \times \texttt{N}) \to (\texttt{N} \times \texttt{N} \times \texttt{N} \to \texttt{N} \times \texttt{N} \times \texttt{N})$

- intensifiers
  - **intensifier$_{\textbf{adj}}$**
    Type: $\texttt{int}_{\texttt{adj}} := \texttt{adj} \to \texttt{adj} = (\texttt{N} \to \texttt{N}) \to (\texttt{N} \to \texttt{N})$ (A problem arises here since adjectives and IVs have same type. Hence, at face value, our type system suggests that adjective-intensifiers can also take IVs as arguments. Modification to our type system is probably necessary to account for this issue.)

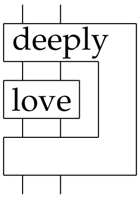
  - **intensifier$_{\textbf{IV}}$**
    Type: $\texttt{int}_{\texttt{IV}} := \texttt{adv}_{\texttt{IV}} \to \texttt{adv}_{\texttt{IV}} = ((\texttt{N} \to \texttt{N}) \to (\texttt{N} \to \texttt{N})) \to ((\texttt{N} \to \texttt{N}) \to (\texttt{N} \to \texttt{N}))$

  - **intensifier$_{\textbf{TV}}$**
    Type: $\texttt{int}_{\texttt{TV}} := \texttt{adv}_{\texttt{TV}} \to \texttt{adv}_{\texttt{TV}} = ((\texttt{N} \times \texttt{N} \to \texttt{N} \times \texttt{N}) \to (\texttt{N} \times \texttt{N} \to \texttt{N} \times \texttt{N})) \to ((\texttt{N} \times \texttt{N} \to \texttt{N} \times \texttt{N}) \to (\texttt{N} \times \texttt{N} \to \texttt{N} \times \texttt{N}))$

- **intensifier$_{\mathrm{DV}}$**
  Type: $\mathtt{int}_{\mathtt{DV}} := \mathtt{adv}_{\mathtt{DV}} \to \mathtt{adv}_{\mathtt{DV}} = ((\mathtt{N} \times \mathtt{N} \times \mathtt{N} \to \mathtt{N} \times \mathtt{N} \times \mathtt{N}) \to (\mathtt{N} \times \mathtt{N} \times \mathtt{N} \to \mathtt{N} \times \mathtt{N} \times \mathtt{N})) \to ((\mathtt{N} \times \mathtt{N} \times \mathtt{N} \to \mathtt{N} \times \mathtt{N} \times \mathtt{N}) \to (\mathtt{N} \times \mathtt{N} \times \mathtt{N} \to \mathtt{N} \times \mathtt{N} \times \mathtt{N}))$

- adpositions
  - **adposition$_{\mathrm{IV}}$**
    Type: $\mathtt{adp}_{\mathtt{IV}} := \mathtt{IV} \to \mathtt{TV} = (\mathtt{N} \to \mathtt{N}) \to (\mathtt{N} \times \mathtt{N} \to \mathtt{N} \times \mathtt{N})$
  - **adposition$_{\mathrm{TV}}$**
    Type: $\mathtt{adp}_{\mathtt{TV}} := \mathtt{TV} \to \mathtt{DV} = (\mathtt{N} \times \mathtt{N} \to \mathtt{N} \times \mathtt{N}) \to (\mathtt{N} \times \mathtt{N} \times \mathtt{N} \to \mathtt{N} \times \mathtt{N} \times \mathtt{N})$
  - **adposition$_{\mathrm{DV}}$**
    Type: $\mathtt{adp}_{\mathtt{DV}} := \mathtt{DV} \to (\mathtt{N} \times \mathtt{N} \times \mathtt{N} \times \mathtt{N} \to \mathtt{N} \times \mathtt{N} \times \mathtt{N} \times \mathtt{N}) = (\mathtt{N} \times \mathtt{N} \times \mathtt{N} \to \mathtt{N} \times \mathtt{N} \times \mathtt{N}) \to (\mathtt{N} \times \mathtt{N} \times \mathtt{N} \times \mathtt{N} \to \mathtt{N} \times \mathtt{N} \times \mathtt{N} \times \mathtt{N})$

We can then extend this type system to composite circuit components. This will deal with the earlier examples in this section. For instance, the component "deeply love"



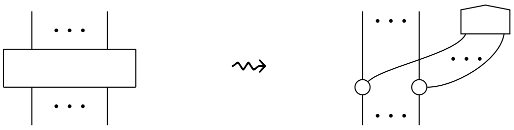
has the type of a $\mathtt{TV}$, since it consists of an $\mathtt{adv}_{\mathtt{TV}} = \mathtt{TV} \to \mathtt{TV}$ "deeply" in which the $\mathtt{TV}$ argument slot has been filled with the $\mathtt{TV}$ "love". This $\mathtt{TV}$-typed composite of "deeply loves" can then fit into the $\mathtt{adv}_{\mathtt{TV}}$ "truly", to return a further $\mathtt{TV}$-typed component "truly deeply love".

For the second example, the composite "happily plays" has the type $\mathtt{IV}$, and hence can fit into the argument slot of "in", which has type $\mathtt{adp}_{\mathtt{IV}} := \mathtt{IV} \to \mathtt{TV}$.

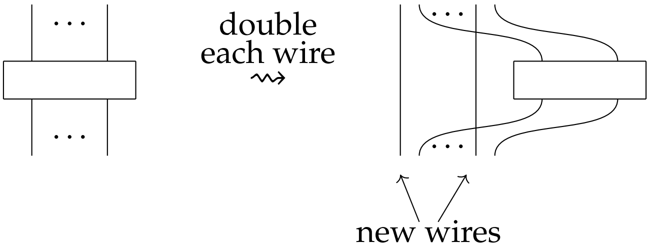
5.3 Spider-form of circuit components

As noted in [CW], we can without loss of generality assume that the circuit components we have just listed come in a 'spider-form', for an appropriate choice of wires and (non-commutative) spiders.

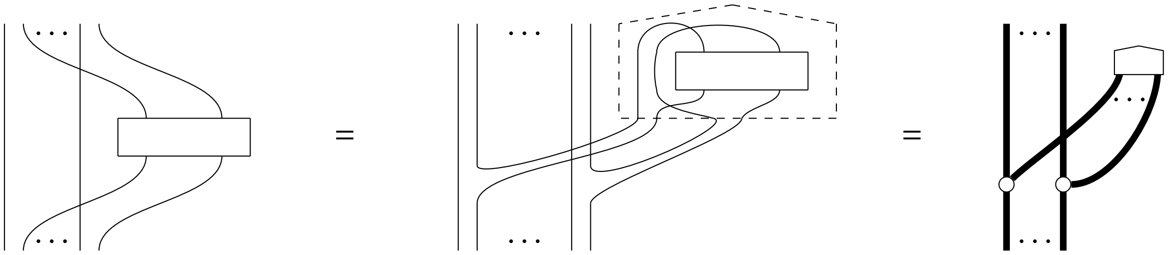
For standard gates, like our adjectives and verbs, we may without loss of generality assume that they come in the following spider-form



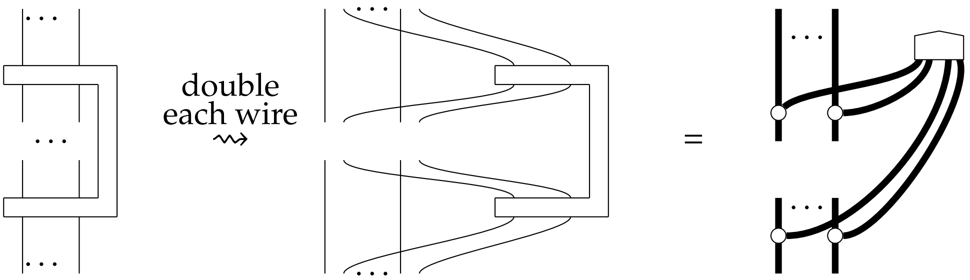
This is achieved by first doubling every wire - i.e. inserting dummy identity wires that play no role.



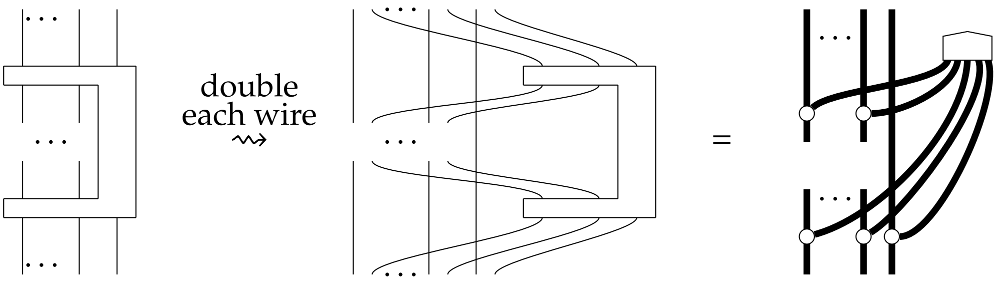
Then we bend the diagram so that the pairs of wires form pair-of-pants spiders (subsection 2.5.1).



We use the same process to obtain a spider-form for combs that have n input wires and n output wires inside the comb (i.e. our adverbs and intensifiers)



Meanwhile, for our adpositions, which have $n+1$ input wires, but only n output wires inside the comb, we have the following spider-form.



By representing all gates and combs with wires and states, spider-form gives us a link with DisCoCat framework, where all the words are states.

Chapter 6

A text-to-circuit algorithm via dependency parsing

In this chapter, we discuss a process for going from a *text* in a specific fragment of English to a *language circuit* representation of that text, using information from the aforementioned spaCy dependency grammar parser (`https://explosion.ai/demos/displacy`). This algorithm will consist of two parts. The first part, presented in section 6.1, is a text-to-circuit algorithm for simple sentences as discussed in chapter 5. For the second part, discussed in section 6.2, we expand the fragment of English that is dealt with by incorporating some extra parts of speech. At the algorithmic/language circuit level, these extra POS will correspond to circuit assembly rules involving simple sentences, rather than the introduction of new circuit components.

6.1 Simple sentence algorithm

In this section, we propose an algorithm that takes as input a *simple sentence*, along with parsing information, and produces as output a language circuit representation of that sentence.

Simple sentences (section 5.1) are built up from constituents that each correspond to some kind of gate or comb. The combs all take a specific kind of input and give a specific kind of output. Hence, the process of going from a given simple sentence to a language circuit is just a matter of 1) figuring out the shape of the circuit components we ought to associate with each word in the sentence, and then 2) figuring out how these circuit components plug into each other.

To address these questions, we must defer to the information we get from parsing the sentence according to some grammatical framework. Each of the formalisms we have mentioned (PG, CG, dependency) could probably be useful. However, I am of the view that for simple sentences, dependency grammar should be sufficient by itself for giving an idea of how to plug components together. Indeed, as noted in claim 5.1.1 I think that dependency grammar captures what we are looking for much more naturally than typelogical grammars. Supporting this claim, we will see in this chapter that dependency relations very often exactly match the input/output information we desire.

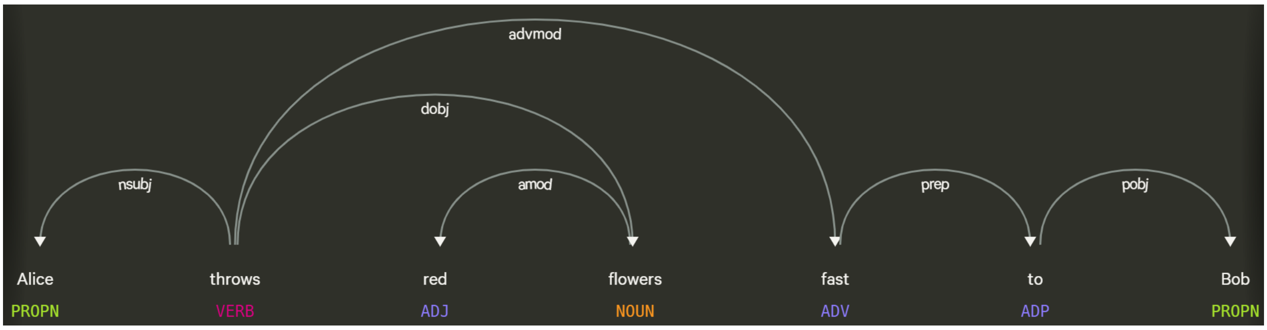
There are many readily available dependency parsers. For the purpose of this project and of this chapter, we have chosen to use the spaCy parser, but if this approach is to be explored further, it would be well worth experimenting with different dependency parsers, for instance [CC07].

6.1.1 Firstly, an example

Before we jump into the precise details of how our algorithm works, it is probably instructive to look at an example that will illustrate the main ideas. Consider the simple sentence

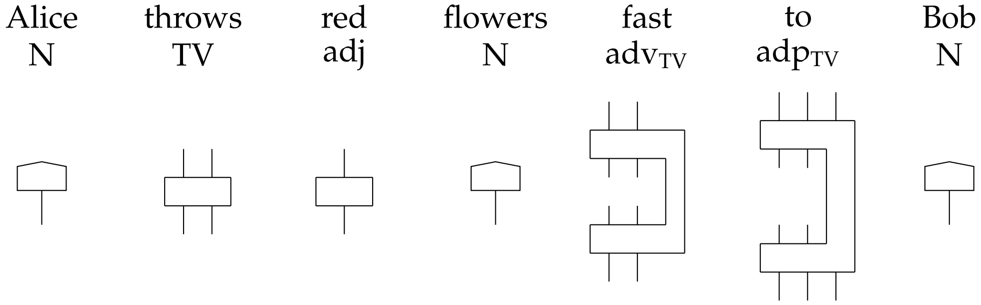
"Alice throws red flowers fast to Bob."

Running it through spaCy gives the following POS tags and dependencies.



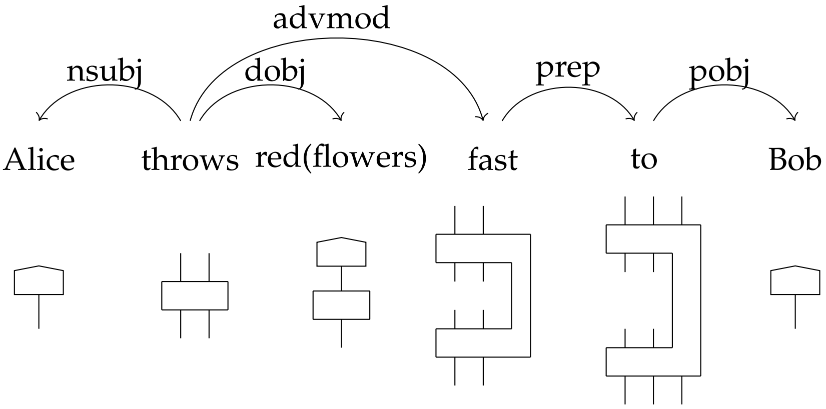
We want to use this information firstly to associate each word with the correct circuit component. We know that nouns (including proper nouns) correspond to states, and adjectives to gates on one noun wire. However, an immediate issue that arises when we consider verbs, adverbs, and adpositions is that in order to determine the shape of their circuit component, we need *valency information*. For instance, to know what kind of gate we should associate with the verb "throws", we need to know whether it is intransitive, transitive, or ditransitive.

We propose that this *valency information can be inferred by looking at the dependencies*. It seems reasonable to argue that since "throws" has two noun dependencies (an 'nsubj' and a 'dobj'), it is transitive. Meanwhile, the adverb "fast" acts on "throws", and it seems reasonable to type it as an adv_{TV}. Finally, since the adposition "to" acts on "fast", it seems reasonable to also type it as adp_{TV}. With this, we can now assign the following circuit component shapes:

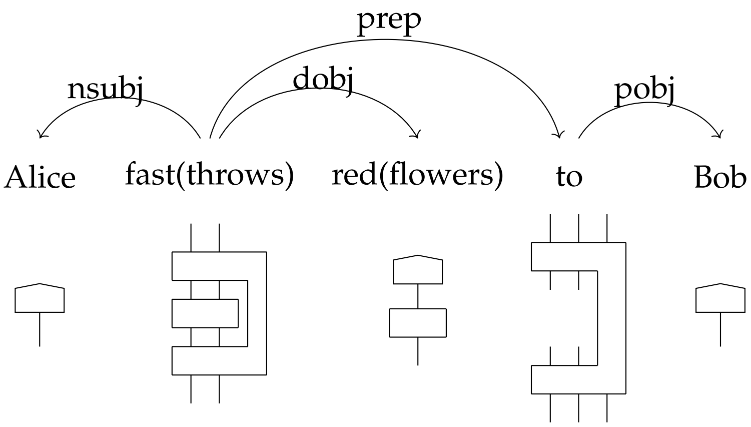


The first step is thus complete. Now, these components need to fit together somehow. How we propose to do this is by 1) picking a directed edge in the dependency tree provided by spaCy, 2) quotienting the endpoints of this edge, which means performing a graph theoretic edge-contraction on the dependency graph whilst plugging together the two circuit components at the endpoints, and 3) repeating this process until we are left with a language circuit. The only caveat is that *we have to pick the edges in a good order* - some 'bad' choices of edge order will lead to the failure to build a circuit.

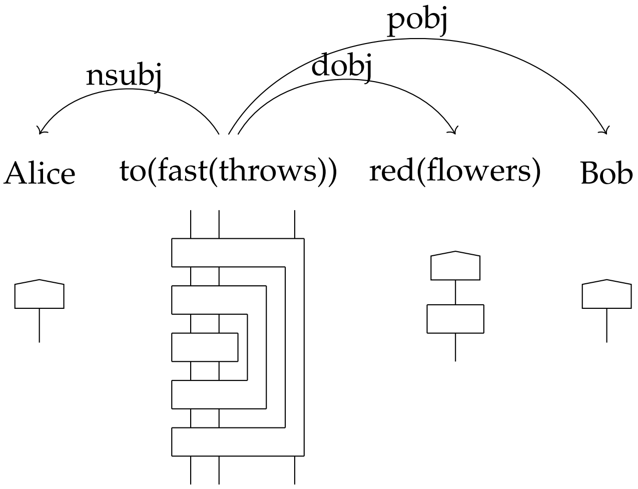
To illustrate this process, suppose that for the first step we choose to contract the 'amod' edge going from "flowers" to "red" - that means we plug together the circuit components for "red" and "flowers", to obtain



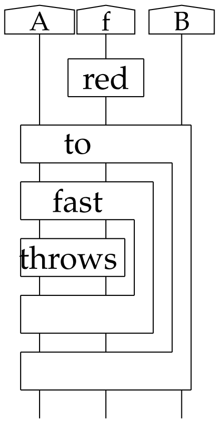
Next, let us quotient the 'advmod' edge, combining "throws" and "fast" into "throws fast":



Now all the remaining edges involve plugging in nouns, except the ‘prep’ edge from “throws fast” to “to”, so we quotient it next.



We now basically have a language circuit awaiting noun inputs - all that remains is to plug in these nouns. Clearly, the three noun wire inputs of the circuit going from left to right correspond to the ‘nsubj’, ‘dobj’, and ‘pobj’ dependencies respectively. Performing these edge quotients gives the final circuit



corresponding to the component to(fast(throws))(Alice, red(flowers), Bob).

To recap, there were a few algorithmic ingredients needed. Firstly, we needed to know the kind of graphical 'plugging in' operation that corresponds to each dependency type - e.g. we had to know that a 'pobj' dependency between a preposition and a noun means we plug the noun into the auxiliary noun input of the preposition comb. Secondly, we needed to know which order to pick the edges in. In the following sections, we will make this idea precise.

6.1.2 Preprocessing the spaCy parse, handling exceptions

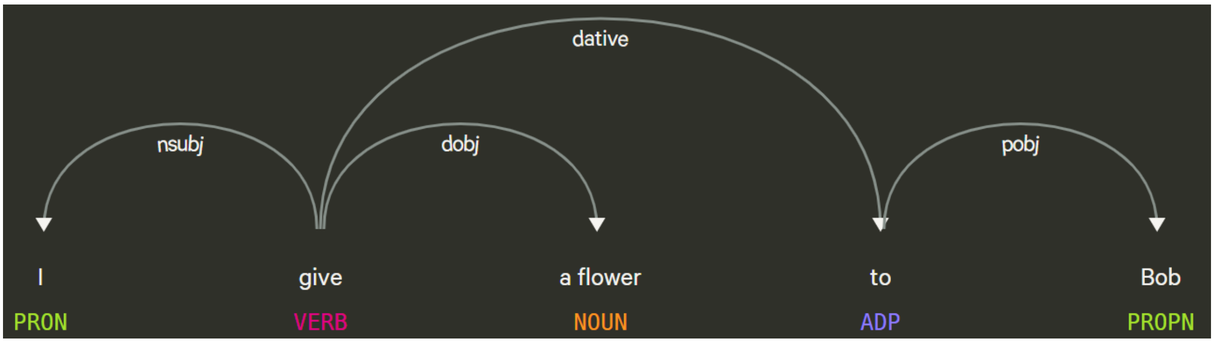
Certain kinds of sentences will lead to 'exception cases', where the grammatical information provided by the spaCy parse does not quite match with the grammatical information we desire. In these cases, the spaCy data must first be adjusted on a case-by-case basis. This is done before we proceed with inferring valency and performing the component plugging-together.

Below, I give some examples of exceptions that I have discovered, and how to handle/fix each one.

1. dative dependency of a verb links to an intermediate adposition rather than directly to the noun:

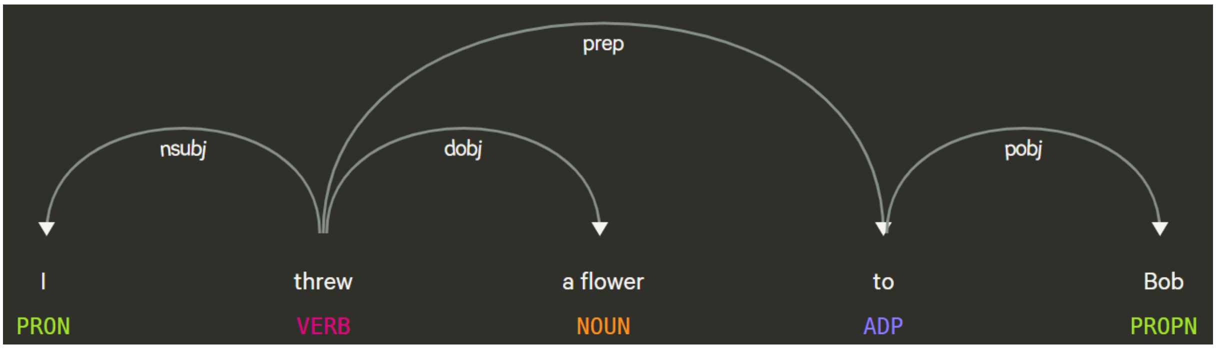
   Usually, spaCy connects a verb with its subject via a 'nsubj' dependency, with its (direct) object via a 'dobj' dependency, and what we would call the indirect object is connected by a 'dative' dependency.

   However, an example that does not quite follow this rule is the following: "I give a flower to Bob".



   Here it is fairly clear that *if "give" is viewed as a ditransitive verb*, then the dative/indirect object should be "Bob". However, in the spaCy parse the 'dative' dependency connects to an intermediate adposition that then connects with "Bob".

Note that the parse we would have expected, *based on the way we have thus far been dealing with adpositions*, would have labelled "gives" as a *transitive* rather than ditransitive verb. Then the 'dative' dependency here would instead be a 'prep' dependency. This kind of parse is in fact what spaCy gives for the sentence "I threw a flower to Bob".

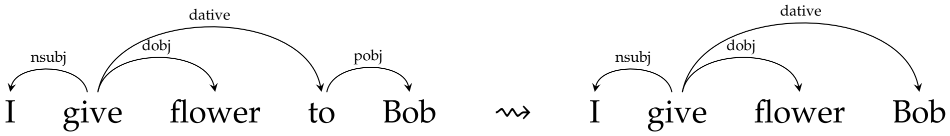


Nevertheless, the spaCy parse for "I give a flower to Bob" is probably more natural and correct than our expected parse - the spaCy parse reflects that "I give a flower to Bob" is semantically equivalent to "I give Bob a flower", and in the latter sentence "give" is a ditransitive verb with direct and indirect objects.

As for how we deal with this exception case in our algorithm, the solution is fairly natural. We refer to the fix as 'adposition deletion', and in future will apply it generally.

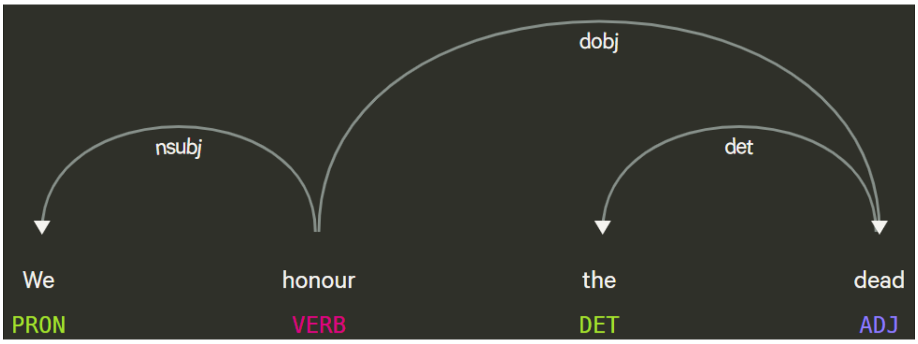
Definition 6.1.1. *If, in a dependency graph, we have a ditransitive verb where the dative object is connected via an adposition (e.g. "to"), we perform* ***adposition deletion*** *on the graph. That is, we delete the adposition, and manually reconnect the dative dependency to the 'pobj'.*

In the above example, this would look like



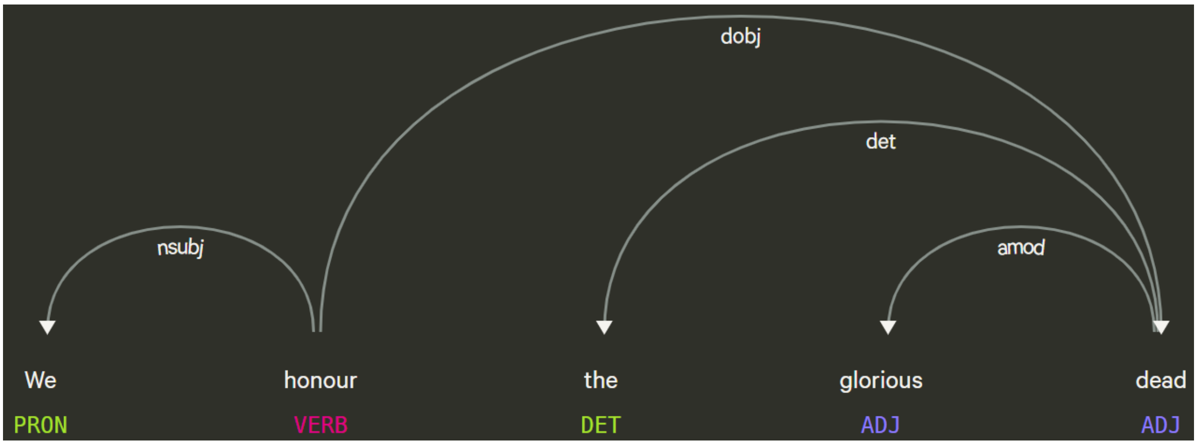
2. noun phrases where the head is not parsed as a noun:

   In a sentence like "We honour the dead", the word "dead" is parsed by spaCy as an adjective.



However, we would (probably) like "the dead" to constitute a noun state. Then everything else would behave normally. A similar example is "We honour the fallen", where "fallen" is labelled a verb.

Building on these examples, consider "We honour the glorious dead", where "glorious" is an adjective modifying another adjective. Similarly, "We honour the glorious fallen", has an adjective modifying a verb.



Motivated by these examples, a possible fix seems to be that *if a noun dependency of a verb (in these cases the 'dobj' dependency) links the verb to an adjective or another verb instead of a noun, simply relabel the target as a noun.*

3. exceptions arising from incorrect parsing:

   Statistical parsers will not always be correct 100% of the time - that is, they will sometimes return parses that are evidently grammatically incorrect. For instance, in the sentence "I pass Tom the ball", the verb "pass" somehow ends up with two 'dobj' dependencies - "Tom" and "the ball" - rather than a 'dobj' and 'dative'. It is not generally considered grammatically invalid for a verb to have two direct objects. Indeed, if we replace "Tom" with "him", to get "I pass him the ball", then "him" is correctly identified as the dative rather than a direct object.

To fix cases like these, a catalogue of incorrect cases could be built up.

4. there are also several ‘exceptions’ that arise due to the fact that we have not yet decided how we ought to handle certain grammatical constructions. Specifically, the parts of speech we included in simple sentences can often combine in other ways than those that we had considered.

   For instance, in the phrase “office chair” both words are parsed as nouns, with a ‘compound’ dependency joining them. We can of course represent “office chair” with a single noun state, but if we want to include some further substructure within the state, how we do so will depend on how we decide to deal with compound nouns - something we have not discussed.

   In the noun phrase “moving day”, moving is a verb, and the two are joined via an ‘npadvmod’ dependency. Another example of an ‘npadvmod’ dependency is in the construction “IBM earned 5 dollars a share”. These ‘npadvmod’ depedencies have also yet to be discussed.

Of course, there are likely many more exceptions along these lines. Based on these examples, it appears that these exceptions arise due to either 1) the statistical aspects of spaCy leading to an *incorrect* parsing, 2) a correct parsing but one with grammatical constructions we do not yet know how to deal with, or 3) a correct parsing but one where the underlying grammar model for spaCy does not match what we want.

For the third case, the hope would be that as we discover more of these ‘true’ exceptions, we can just fix them by adding rules to this initial pre-processing stage of the algorithm.

6.1.3 Inferring valency

We need to infer the extra valency information for each word from their dependencies. This is necessary so that we can associate specific circuit components with the words.

A general theme for the order of inference here seems to be *infer the valencies of lower order components first, then those of higher order*. For instance, as a comb that sends verbs to verbs, an adverb is of a higher order than verbs. Our adverb-intensifiers, which are things that send adverbs to adverbs, are of yet higher order.

1. We begin by inferring the valency of the verbs - this is just a simple matter of checking the number of subject/object dependencies that each verb has. So a verb with only an ‘nsubj’ dependency is intransitive, one with ‘nsubj’ and ‘dobj’ is transitive, and one with ‘nsubj’, ‘dobj’, and ‘dative’ is ditransitive.

2. Next, we infer the 'valency' of words tagged as adverbs[1]. Firstly, we look at those that modify a verb via an 'advmod' dependency - these are true adverbs by our definition (section 5.1). Their 'valency' is just the valency of the verb that is being modified.

   Next, what we call intensifiers are generally treated by spaCy as adverbs that modify adjectives or other adverbs. Clearly, an adverb that modifies an adjective is our $\mathrm{int}_{\mathrm{adj}}$ (with only one possible valency), whereas the valency of an intensifier modifying an adverb is just the same as the valency of the adverb.

3. Finally, we infer the valency of adpositions. Again, this consists of following the 'prep' dependency to the verb that the adposition is acting on, and taking that to be the valency. Note that having applied 'adposition deletion' in the stage where we modify the parse data, we have hopefully eliminated any adpositions that do not have a 'prep' dependency.

6.1.4 Order for contracting edges

Once we have the valency information sorted, it remains to perform the necessary dependency edge contractions/plugging in operations. There are two separate questions here - the question of which order to contract edges, and the question of what graphical plugging in operations the different dependencies correspond to. In this subsection we address the former.

Recall that the order of edge contraction matters because certain choices of edge order will cause the algorithm to fail to produce a circuit, because at some step the remaining components are unable to plug together. We make the note that at this stage, there is currently no motivation for specifying some order of edge contraction beyond this desire to ensure our algorithm works.

As with inferring valency, there is a general theme here based on higher and lower order components. In this case, we want to *go from higher order to lower order* - that is, performing our edge contractions in the order intensifiers → adverbs → verbs → nouns. This rule seems fairly natural to impose, and seems to give the right outcomes.

Concretely, we propose the following rules regarding the order of edge contraction.

1. contract 'advmod' and 'amod' edges first, as these correspond to the application of adjective and adverb (including intensifier) combs. This should not lead to any clashes - i.e. 'advmod' and 'amod' dependencies seem like that should apply to different circuit components.

[1] This is an abuse of terminology - nevertheless, we will use 'valency' to refer to the information provided by the subscripts in types like $\mathtt{adp}_{\mathtt{TV}}$, $\mathtt{int}_{\mathtt{IV}}$ etc.

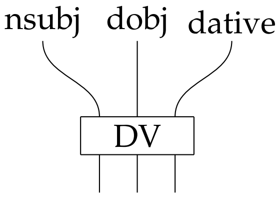
2. contract ‘prep’ edges - i.e., apply adpositions. Note we need to do this after the combs are applied as an artefact of the way we assign valencies. This is because applying a preposition to some composite verb-like component will lead to a change in valency.

3. contract subject/object dependencies (i.e. ‘nsubj’, ‘dobj’, ‘dative’) last, as these correspond to plugging in noun states into the top of the language circuit.

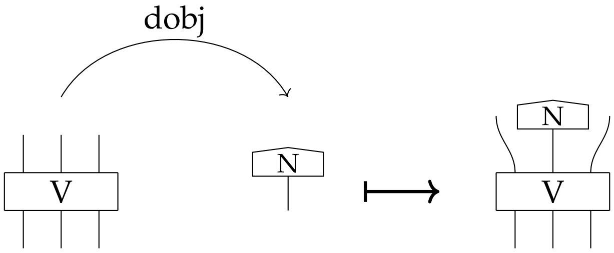
6.1.5 Dependencies vs graphical circuit operations

Now we address the final question - matching edge contractions with specific graphical plugging operations.

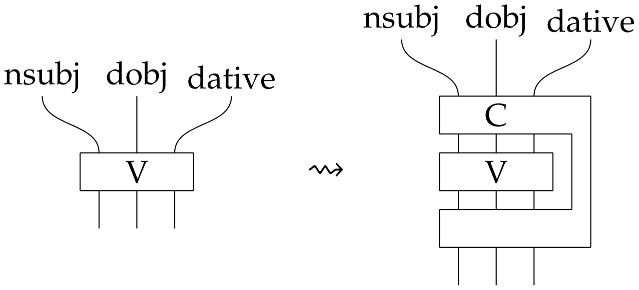
1. Rules for plugging in nouns (‘nsubj’, ‘dobj’, ‘dative’): given a bare verb, the noun dependencies correspond to plugging in states at the following wires (if they exist):



i.e., we have rules of the following form

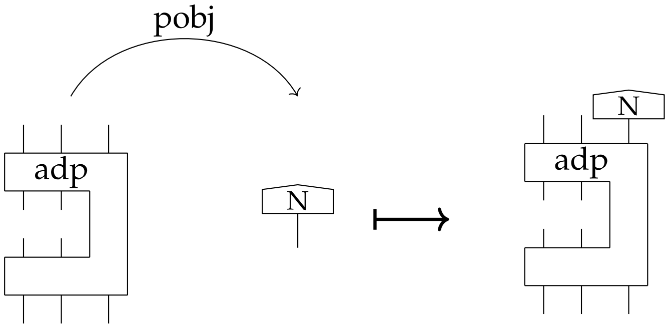


When a verb V is modified by an adverb/intensifier comb C to form a composite verb gate, the valency of the resulting verb does not change. Recalling that we want these noun dependencies to be contracted last, the new composite component consisting of V and C will still have the same noun dependencies of the original verb V. The location that we plug in the corresponding nouns just gets changed to the inputs of the comb C:



2. Applying adverb/intensifier combs (e.g. 'advmod', 'amod' dependencies): this should be fairly straightforward, as there is only way way to do this.

3. Prepositions ('pobj'): the 'pobj' dependency of a preposition corresponds to the new noun input introduced by the preposition, i.e. we have rules of the form



As a general remark: if we do want to seriously pursue this approach, it would be useful to delve more deeply into the specific grammar formalism that the English spaCy model is based on. This way we could more systematically analyse if our algorithm works, rather than just randomly coming up with example cases and exceptions.

6.1.6 Pseudocode

Algorithm 1 A simple sentence text-to-circuit algorithm

```
input: English sentence
obtain spaCy parse of the sentence
preprocessing of spaCy data:
    merge trivial determiners with their nouns into a single noun phrase represented
    by a noun state
    perform adposition deletion on intermediate adpositions
    relabel verbs, adjectives that are the target of nsubj, dobj, or dative dependencies
    as nouns
    more rules here as necessary...
infer valency information:
    infer valency of verbs by looking at their subject/object dependencies
    infer valency of adverbs based on the verbs they modify
    infer valency of intensifiers based the adverbs they modify
    infer valency of adpositions based on the verbs they modify
associate a specific circuit component shape with each word based on POS tags and
inferred valency
perform edge quotienting/component plugging operations
    adv, advmod edges first
    other edges...
    subject/object edges last
return completed language circuit
```

6.2 Expanded algorithm

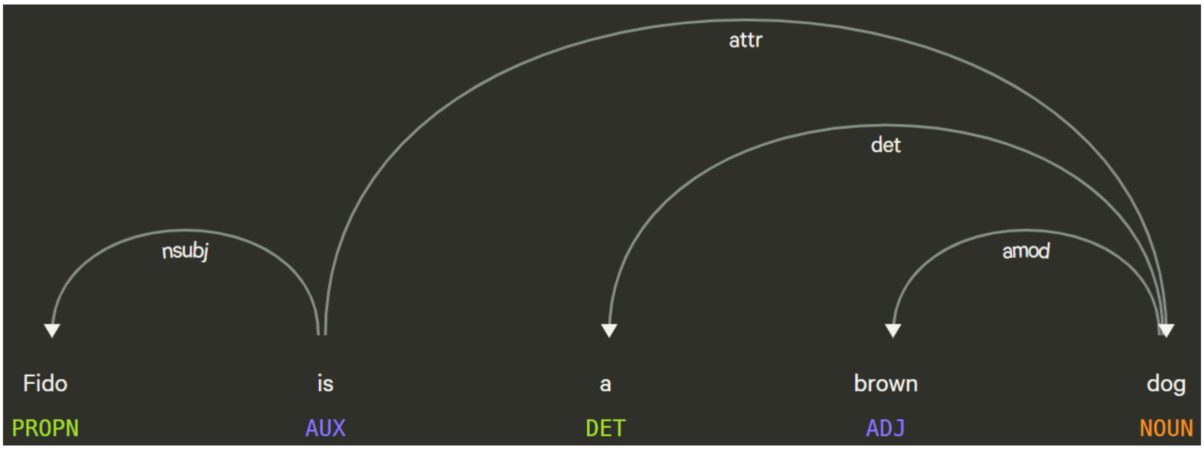
Having (hopefully) dealt with simple sentences, the next thing we do is to expand the kinds of sentences we can deal with, by throwing in some extra parts of speech/lexical categories into the mix. We pick those that, at the language circuit level, should manifest not as new types of circuit components involving boxes, but rather as additional assembly rules (involving wiring and possibly spiders) for our established basic circuit components. Also, we stay in a fragment of English that excludes more complicated complements, as per the discussion in section 5.1.

In contrast to the simple sentence algorithm of the previous section, which was fairly complete and natural, this second part of the algorithm intuitively feels much more incomplete and ad hoc. It is incomplete, in the sense that we basically have tacked on a few extra POS to our fragment of English, but are still missing lots of things. It is more ad hoc, in that the simple sentence algorithm feels very naturally aligned with dependency parses and required adjustment of the dependency parse information only in exceptional cases, whereas the algorithm presented here consists exclusively of rules for manipulating certain dependency parses.

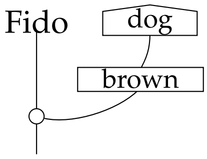
6.2.1 The copula “is”

We deal with two cases - when “is” links the subject to either a noun phrase or an adjectival phrase. Note this also includes different forms like “am”, “was”, “were”, etc.

The following is an example of a noun phrase complement.

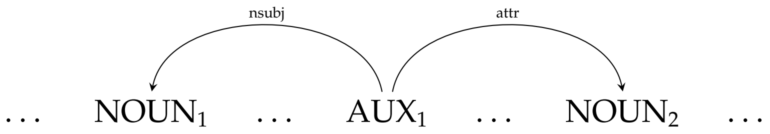


We would probably like this sentence to yield the following language circuit.



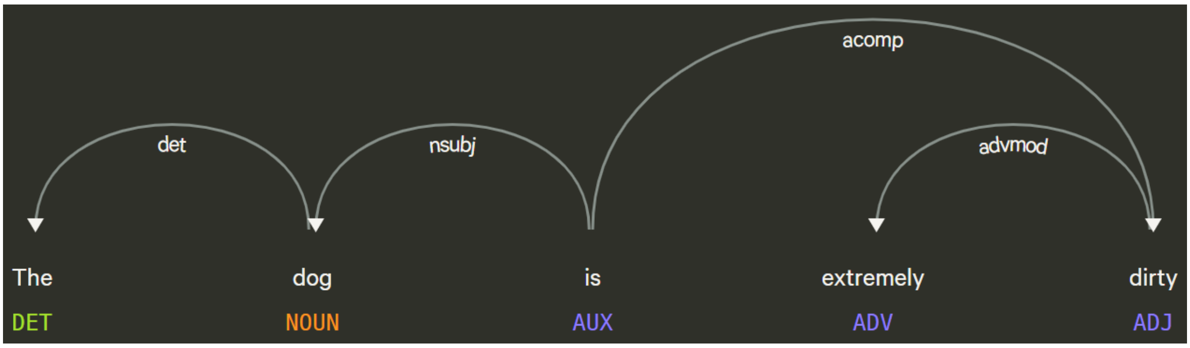
Note that in the parse, the AUX “is” has ‘nsubj’ (linking to the subject) and ‘attr’ (linking to the attributive noun in the complement) dependencies.

The general rule here is that when we have a dependency parse of the form



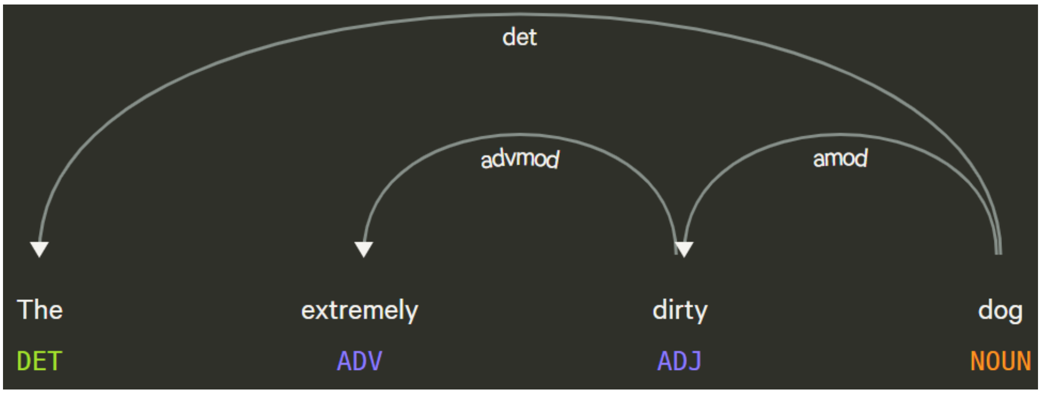
is that 1) we do all the relevant edge contractions for NOUN$_2$, to obtain the circuit component for that noun phrase, and then 2) attach this noun phrase circuit component to the NOUN$_1$ wire by means of a spider. Then 3) we parse the remainder of the sentence.

The following is an example of an adjectival phrase complement.



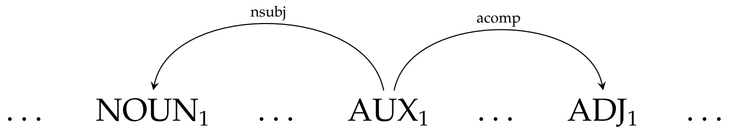
Note that the AUX "is" has 'nsubj' and 'acomp' dependencies. The 'acomp' dependency links to the head adjective of the adjectival complement.

We would expect this sentence to yield a circuit equivalent to the noun phrase

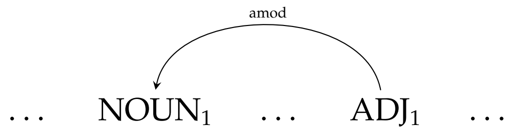


This is topologically the same as the first parse, except the AUX and its dependencies have been deleted, and replaced with an 'amod' dependency.

The general rule that we impose, is that when we have a dependency parse of the form

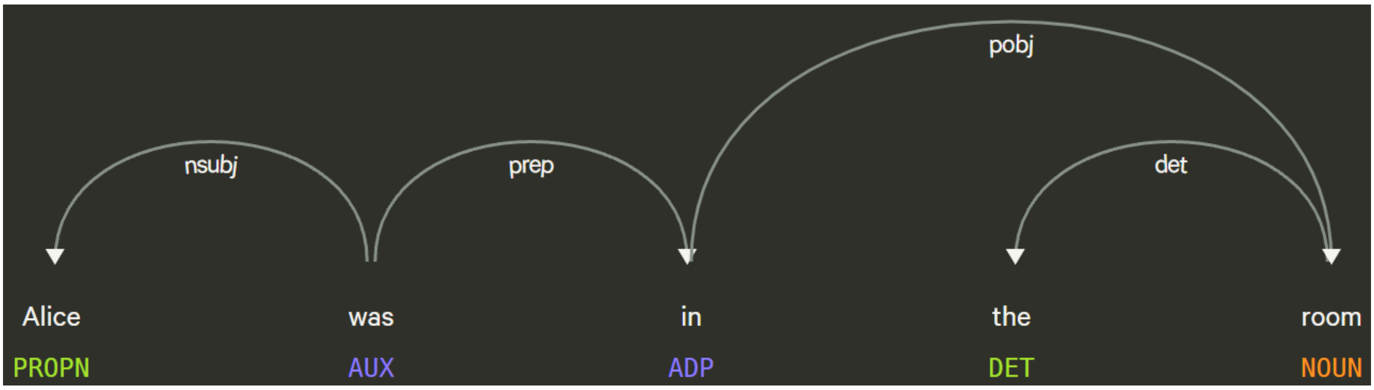


we convert this to a parse of the form



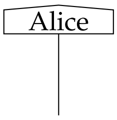
That is, we delete the "is" term/graph vertex, and create an 'amod' dependency going from ADJ_1 to the NOUN_1. All other edges/vertices of the dependency graph are left unchanged.

A further interesting example to note in this subsection is the sentence "Alice was in the room" (or equivalently the inverse copular form "In the room was Alice").



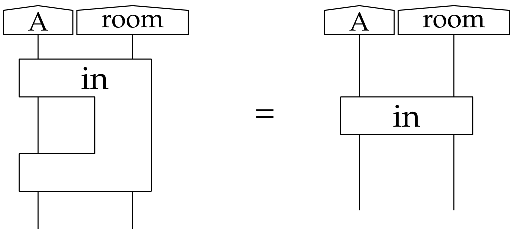
Here the complement is an adpositional phrase “in my room”, so it does not quite fall within the range of examples we dealt with. However, an alternate possibility is to treat the AUX “was” as a intransitive verb described by an identity wire.

“Alice is.” $\rightsquigarrow$



In this case the adposition “in” is a special adp_{IV} taking an identity wire input, which functionally becomes the same as a transitive verb:

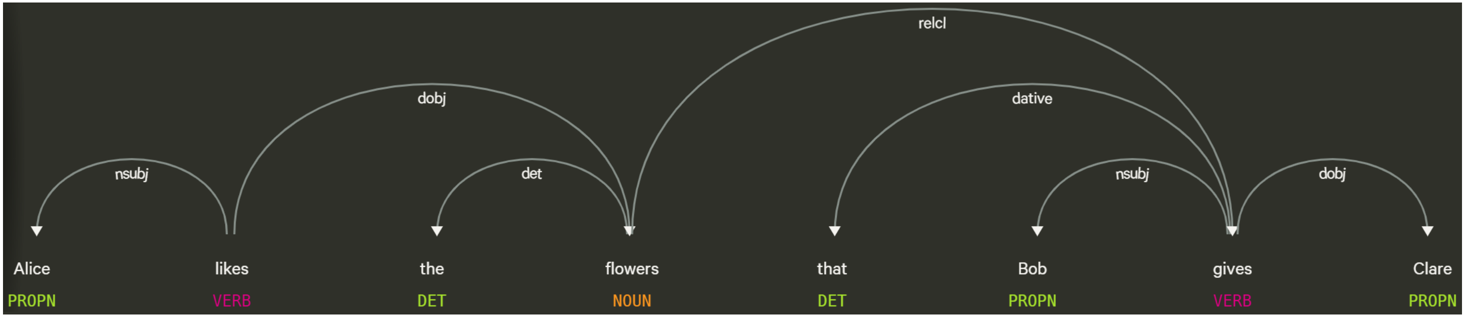
“Alice is in the room.” $\rightsquigarrow$



Adopting this idea allows us to deal with the examples above, in which the AUX has an adpositional complement, where the adposition (as usual) joins to an additional noun phrase.

6.2.2 Relative pronouns

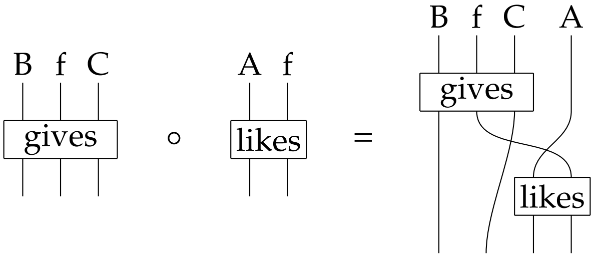
Consider the sentence “Alice likes the flowers that Bob gives Clare”, with the following parse.



We want this to yield the circuit equivalent of the two sentences

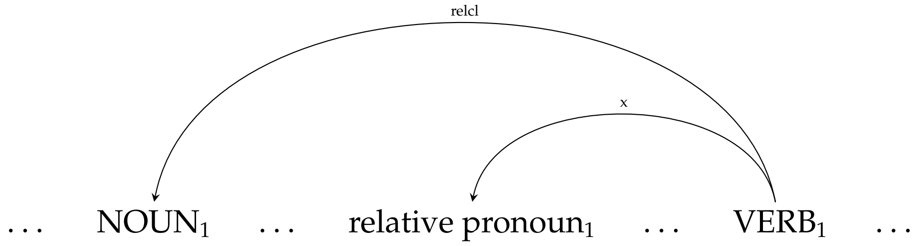
"Bob gives Clare the flowers. Alice likes the flowers."

That is, the following circuit

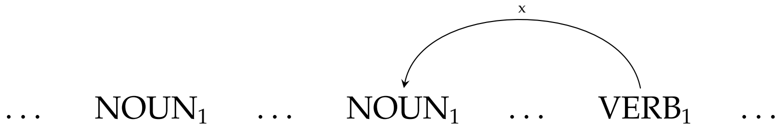


Note that in the parse, what we call a relative pronoun ("that") is parsed as a DET. It acts as the 'dative' argument of the verb "gives", which itself is linked to "flowers" by a 'relcl' (relative clause) dependency.

The general rule is the following - given a parse


relcl
x
. . . NOUN$_1$ . . . relative pronoun$_1$ . . . VERB$_1$ . . .

where there is some dependency 'x' between VERB$_1$ and relative pronoun, we modify this to a parse



That is, we change the original dependency graph by 1) deleting the 'relcl' dependency, and 2) replacing the relative pronoun with a *copy* of the original NOUN$_1$. As usual, all other edges/vertices of the dependency graph are left unchanged. Note this may cause the dependency graph to become *disconnected* (reflecting that we split our original sentence into two sentences), but nevertheless the resulting dependency graph can be treated according to our usual methods.

6.2.3 Passive auxiliaries

Passive auxiliaries can introduce a fair bit of complexity. In the most basic form, a passive auxiliary is used to switch the order of the noun arguments. For instance,

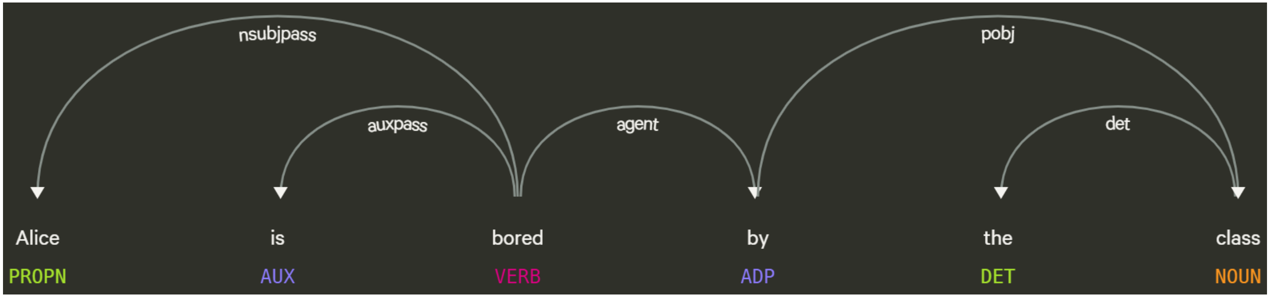
"Alice is bored by the class."

is equivalent to

“The class bores Alice.”

and in the latter sentence, “the class” is the subject, whereas Alice is the (direct) object.

The parse for the former sentence is

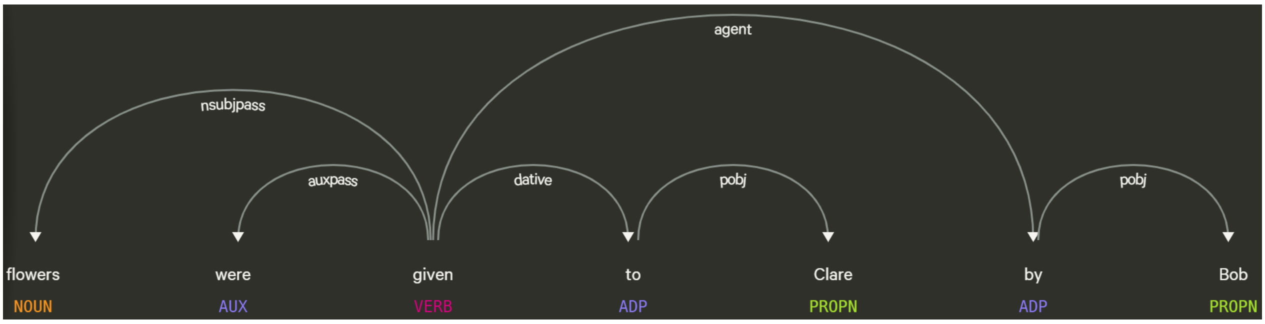


Note that the ‘nsubjpass’ dependency here corresponds to the direct object argument of “bored”, and the ‘agent’ dependency points (via an intermediate adposition) to the subject of “bored”.

The above example involved the reordering of arguments for a *transitive* verb (“bored”). Let us have a look at passive auxiliaries applied to the *ditransitive* verb “gives”, which yields more possible permutations of arguments.

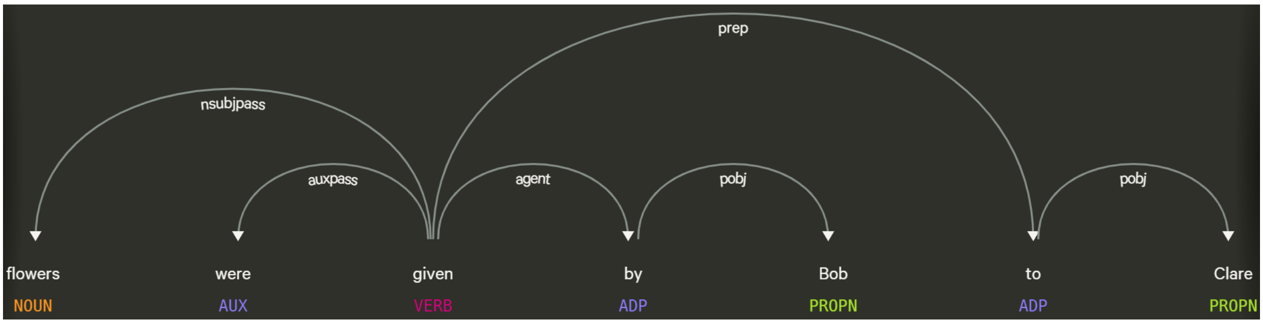
Consider the sentence “Bob gives Clare flowers”, which is in its simplest, canonical form. We have two variants where the direct object (“flowers”) comes first:

- “flowers were given to Clare by Bob”

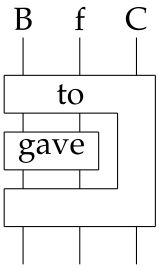


Here, ‘nsubjpass’ ⇝ direct object, ‘dative’ correctly points to the dative (indirect object) via an intermediate adposition, and ‘agent’ ⇝ subject.

- “flowers were given by Bob to Clare”

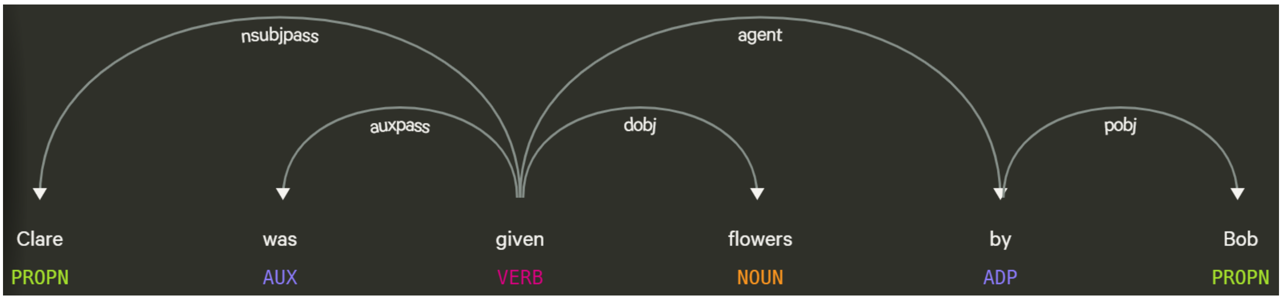


This version, where the order of Bob and Clare are reversed, is topologically similar to the first, except the 'dative' dependency is relabelled as 'prep'. If we treat "gave" as a transitive verb here instead of ditransitive (based on the number of non-prep noun arguments it has), and again interpret 'agent' $\rightsquigarrow$ subject, which leaves 'nsubjpass' $\rightsquigarrow$ (direct) object, we would end up with a circuit of the form



We also have another variant where the indirect object/dative ("Clare") comes first:

- "Clare was given flowers by Bob"

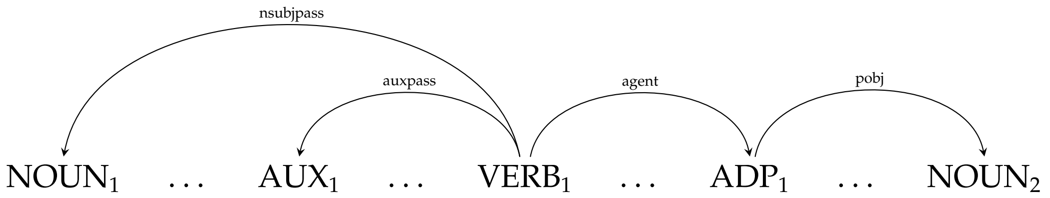


Here, 'dobj' points correctly to the direct object, 'agent' again points to the subject, leaving 'nsubjpass' to point to the dative.

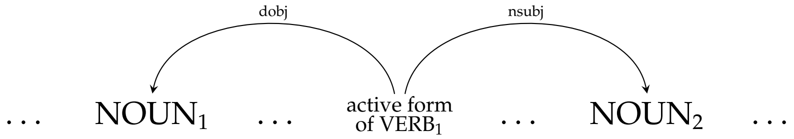
Based on these examples, we will propose a rule for these basic forms of passive auxiliaries.

The general rule is essentially the following. We firstly identify a passive voice verb by its ‘auxpass’ dependency. If necessary, we swap the verb from the passive form to the active form (e.g. “given” to “gave”). Then, we identify the noun pointed to by the ‘agent’ dependency as the ‘subject’ of the verb (possibly involving an intermediate adposition, which will be deleted). Then, we note if there are any ‘dobj’ or ‘dative’ dependencies. Finally, the ‘nsubjpass’ dependency is relabelled as whatever noun argument remains.

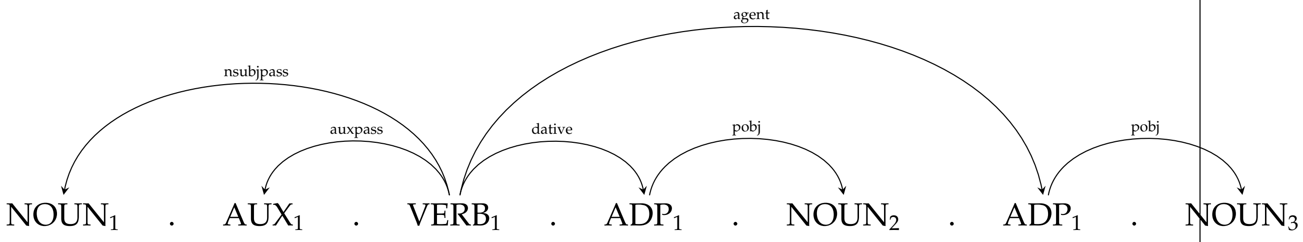
More explicitly, if we have



we reason that VERB$_1$ is transitive, since it has ‘agent’ and ‘nsubjpass’ dependencies. Since the ‘agent’ corresponds to the ‘subject’, the ‘nsubjpass’ must be the (direct) object. So we delete AUX$_1$ and ADP$_1$, and relabel the dependencies, to get the amended dependency graph.

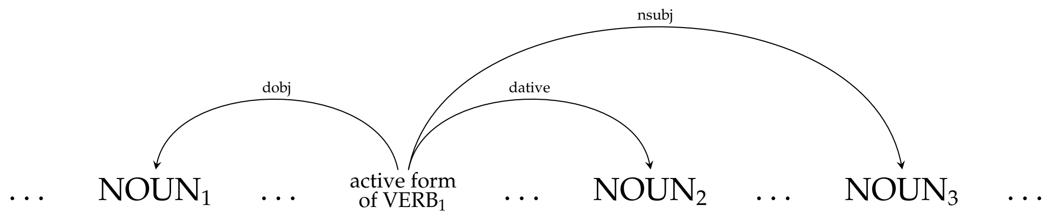


In the case of ditransitive verbs we have the following: given



(6.1)

we reason that VERB$_1$ is ditransitive, as it has ‘agent’/‘subject’, ‘dative’, and ‘nsubjpass’ dependencies. So here the ‘nsubjpass’ must be the direct object, and the result, after adposition-deletion and relabelling, is



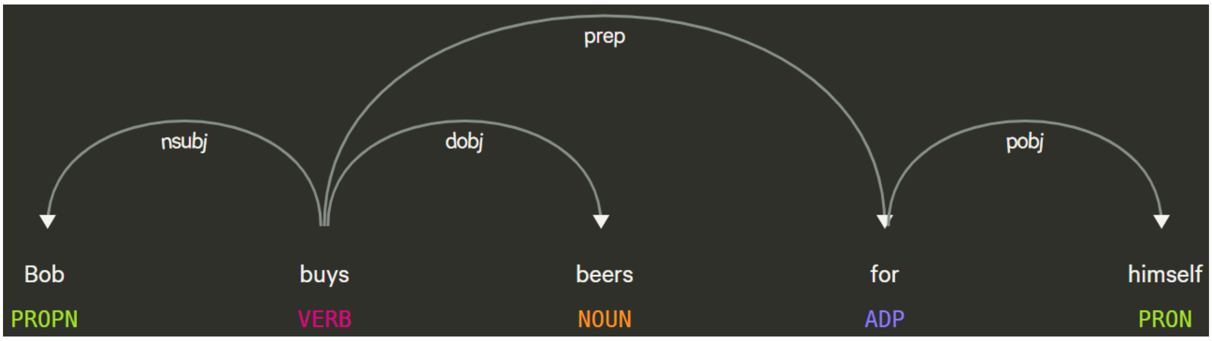
In the case where we have the parse (6.1), except with a 'dobj' label instead of the 'dative' label, then we reason that the 'nsubjpass' should be relabelled to 'dative'.

The examples we have discussed so far simply involve passive auxiliaries with 'agent' dependencies, that serve to reorder the arguments of a verb. We do not, however, introduce passive auxiliaries in their full generality. The issue is that passive auxiliaries can also be used to suppress certain noun arguments of verbs, and we have yet to decide how to model these/relate these to their usual form. As an example, the sentence "The lawn is mown" is roughly semantically equivalent to "(Someone) mowed the lawn" - i.e. the subject of "mow" is suppressed in the passive form.

6.2.4 Reflexive pronouns

Reflexive pronouns are those ending in "-self". An example sentence is "himself" in "Bob buys beers for himself." Intuitively, reflexive pronouns stand in for some other noun/entity, and allows us to feed the same noun wire into multiple inputs of a gate. However, they are problematic for a couple of reasons.

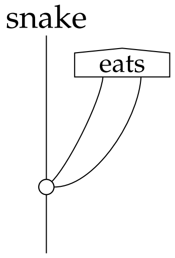
Firstly, parsers (including the spaCy and CCG parser we have been using) *do not identify which noun a relative pronoun is standing in for*. So we need information beyond the parse data from spaCy to deal with these. For instance, this is the spaCy parse for the example sentence.



Nothing here tells us that "himself" is referring to "Bob".

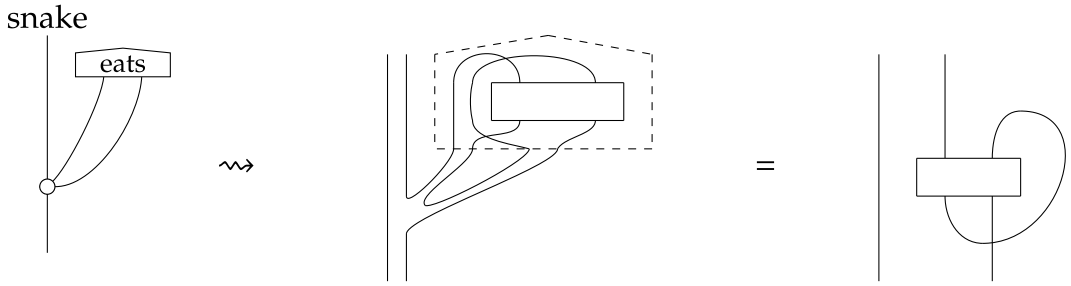
Secondly, it is not clear how we should represent the effect of reflexive pronouns circuit-diagrammatically. There seem to be a couple of options, each with pros and cons.

Let us use the example "Snake eats itself" to demonstrate this. "Eats" is a transitive verb, which takes in the "snake" as both its subject and object. If we use the spider-form of the transitive verb gate, intuitively the circuit should look something like this:



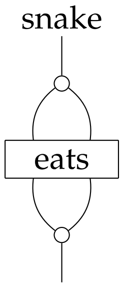
(6.2)

If we undo the derivation we did to arrive at spider-form in the first place, which would entail replacing the wires with doubled wires and spider with the pair-of-pants spider, we find



i.e. this suggests that we ought to deal with reflexive pronouns by using caps and cups to feed the outputs of a verb back into its inputs. A possible downside of this approach is that it seemingly disrupts the causality/one-way-flow of our circuits. Such an approach means that we can have loops in our circuit.

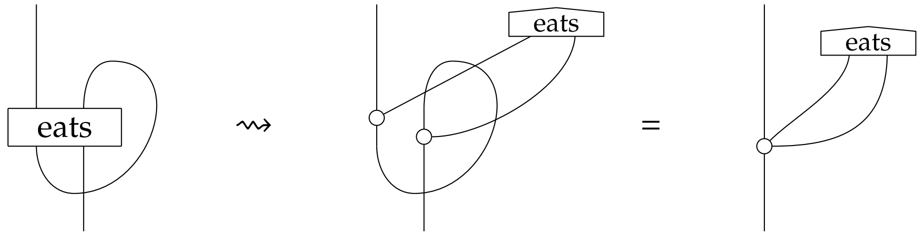
Alternately, another intuitive solution is to use spiders instead of caps and cups, to copy and merge a single noun wire as required. That is, "snake eats itself" would be represented as



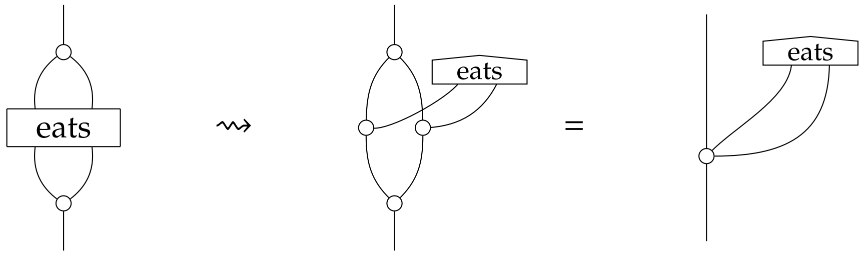
.

A downside of this approach is that the presence of spiders seems to prohibit the argument we make in section 5.3 for showing that we can without loss of generality assume our gates are in spider-form.

However, it is interesting to note that if we replace the gate form with the spider-form, then both choices reduce to the intuitively expected circuit (6.2). For the first proposal involving cups and caps, we have:



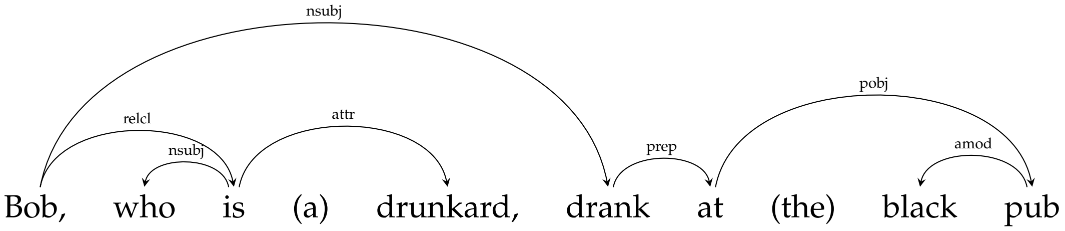
For the second proposal, we have:



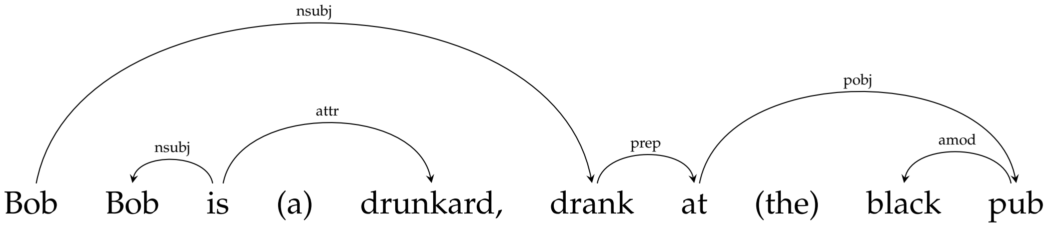
6.2.5 An example

The following sentence includes both a relative pronoun and the copula "is" (with a noun phrase complement)

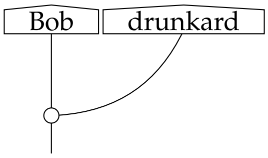
"Bob, who is a drunkard, drank at the black pub"



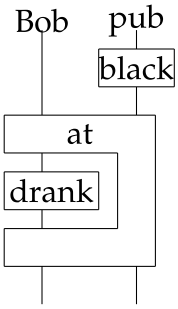
First, we apply the relative pronoun rule, where we delete the 'relcl' dependency and replace the relative pronoun with the named noun:



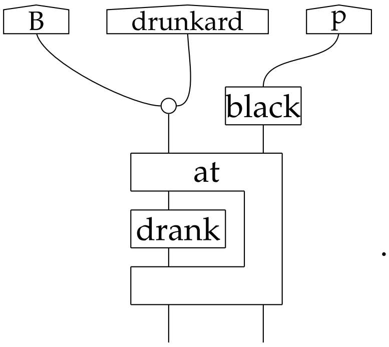
Note this disconnects the graph. Then "Bob is a drunkard" is dealt with using the proposed rule for "is", yielding a component



The remaining part "Bob drank at the black pub" is just a simple sentence, yielding the circuit



Composing, we get the final circuit



6.2.6 Concluding remarks

Obviously, there are issues to be worked out regarding the *order* in which we apply the rules in the second part of the algorithm. More generally, the specifics of this algorithm are probably better worked out in tandem with the process of actually trying to implement it, in that working with real life texts and examples can help identify problems with our current proposal, which can then be iteratively fixed.

Chapter 7

CCG-compatible internal wirings

In this chapter, we follow closely the approach of [CW], and provide a large catalogue of 'internal wirings' that are compatible with a specific, purpose-built CCG. As we will see, these internal wirings allow us to go from DisCoCat diagrams (based this CCG) to language circuits. Thus, this represents an alternative to chapter 6, as another potential way to obtain language circuits from text. Note that [CW] provided internal wirings for a certain *pregroup*-based DisCoCat model - thus, the wirings provided there and the wirings in this chapter end up looking somewhat different.

We roughly sketch the idea behind this chapter in section 7.1, and in section 7.2 we argue that although the set of wirings we have come up with in this chapter was intended to work with a CCG-DisCoCat, they would also feasibly work in an appropriate pregroup based setting. Section 7.3 is the catalogue of internal wirings, and forms the bulk of this chapter.

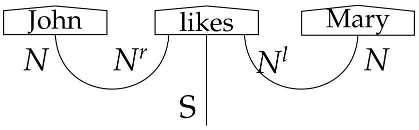
7.1 The overall idea

As previously mentioned, the process can be depicted as

$$\text{DisCoCat diagram} \xrightarrow{\text{internal wirings}} \text{language circuit} \tag{7.1}$$

In this section, we will not fully describe the process as given [CW], but instead provide a pregroup-based example to broadly sketch how it works.

Consider the sentence "John likes Mary", parsed using pregroup grammar, to obtain the following DisCoCat diagram.



The first departure from conventional DisCoCat is that we will work in a version of pregroup grammar which replaces the usual sentence type S via some bracketed tensor product of N types. That is, under the hood of any given S type is some *bracketed* product of N types $[N \otimes \ldots \otimes N]$, where the number of N's may vary for different instances S. Roughly, what we have done is distinguish sentence types by how many noun wires are involved in the sentence. In our example, the type of the transitive verb "likes" will be

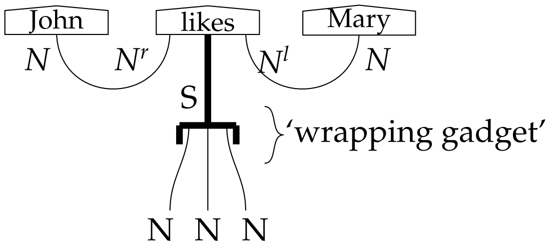
$$N^r S N^l \rightsquigarrow N^r [NN] N^l,$$

i.e. the S type here is replaced by $[NN]$. Note that left and right adjunction can simply pass inside the brackets:

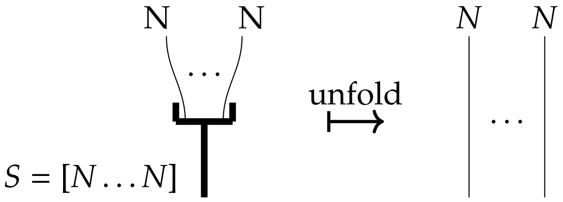
$$S^l = [N \ldots N]^l = [N^l \ldots N^l]$$

etc.

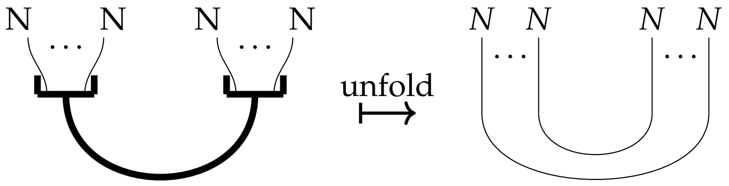
This also leads to some new graphical technicalities. In our diagrams, we denote S types by thick wires, which can be 'unwrapped' to reveal the underlying product of N types:



The indicated graphical gadget that does the 'wrapping' and 'unwrapping' of thick S wires is referred to as a **wrapping gadget**. By **unfolding**, we mean dropping the restrictions imposed by the wrapping gadget, or in other words revealing the N wires underlying all the S wires in our graphical calculus:

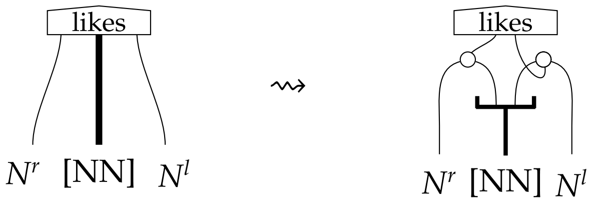


Cups (and analogously, caps) carry over to S wires in the following way:

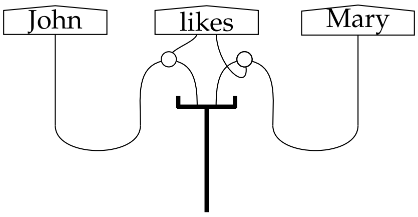


Next, we come up with a large catalogue of internal wirings for various categories of words, and substitute these into our DisCoCat diagram. These categories will roughly follow those introduced in 5.1, as well as those discussed in 6.2 - however for the purposes of this chapter we will often need to fine-grain these into further subcategories.

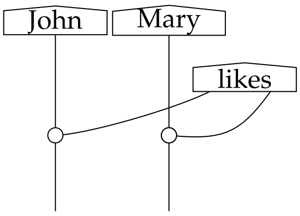
Considering again our example sentence, nouns are unchanged, but transitive verbs form an entry in the catalogue. The internal wiring we provide in this chapter for transitive verbs such as "likes" is the same as that provided in [CW]:



Thus, performing this substitution/rewrite, the DisCoCat diagram becomes



Finally, *unwrapping* the sentence wire returns the language circuit we would expect for the sentence "John likes Mary", where the verb "likes" is in spider-form:



.

Thus, the ingredients of this process are

1. a bespoke typelogical grammar system $\mathcal{T}$ (which has been purpose-built for such a process)
2. a corresponding catalogue of internal wirings, to work with $\mathcal{T}$-based DisCoCat
3. an algorithm (primarily involving plugging in the internal wirings from our catalogue) that takes us from a $\mathcal{T}$-based DisCocat diagram to a language circuit

We note that in order for everything to work as desired, the PG/CCG $\mathcal{T}$ generally needs to be different from standard PG/CCGs involving N and S types. This difference arises not only in the replacement of S types with $[N \dots N]$, but also in terms of its *complexity*. That is, $\mathcal{T}$ will generally need to contain much longer, unwieldy types than what would normally appear in more standard PGs/CCGs.

While in the example we gave in section 7.1 ("John likes Mary") the pregroup type of our intransitive verb "likes" more or less looks the same as it usually does, this is not usually the case. For instance, in [CW], the pregroup type given for a certain subcategory of 'attributive adverb' is

$$N^r N N^r [N][[N N^r][N]]^l N,$$

which is rather more complex than the type of an adverb in any standard pregroup grammar. Something similar holds for many of the pregroup typings provided in [CW].

The main reason for the introduction of extra complexity to $\mathcal{T}$ is that the extra complexity is necessary in order to generate additional underlinks (i.e. cups between words) in the DisCoCat diagrams, *such that the connectivity of the $\mathcal{T}$-based DisCoCat diagrams can more closely resemble the connectivity of the language circuits we hope to obtain*. Broadly speaking, standard PG or CCG systems do not yield the right kind of connectivity between word states that will allow us to recover our desired language circuits via internal wirings. This ties in with claim 5.1.1, and the related observations in section 5.1 regarding connectivity (e.g. see (5.2)).

Even more complexity is introduced to $\mathcal{T}$ if one wants this bespoke grammatical system to capture certain grammatical features (such as the stacking of certain predicative adverbs) that may not naturally occur in it. Indeed this is responsible for much of the additional type-complexity and internal wiring complexity in [CW].

7.2 Applicability to PG-DisCoCat

In this section, we discuss how the typings and internal wirings we provide here, even though originally intended for a CCG based model, could also work well in the right pregroup based model. This is why the title of this chapter refers to 'CCG-compatible' internal wirings - they are compatible with CCG, but are not exclusive to CCG, and could also be used with pregroups.

Recall from subsection 3.5.4 that there is a certain degree of inter-operability between CCG and PG, by mapping CG types to PG types under (3.9). We noted that the basic application and composition rules of CCG are realizable via pregroup contractions, though type-raising is not. We also recall that once we functorially pass to DisCoCat diagrams, the types of words in CCG-DisCoCat look the same as the types in the corresponding pregroup-DisCoCat, due to (4.4). Indeed, interpreted string-diagrammatically, the resulting cups/connectivity one obtains from the CCG rules (e.g. (4.5)) are the same as one would get from the corresponding pregroup contractions.

As it turns out, for the CCG system we devise in this chapter (i.e. in section 7.3), many of the basic example expressions can be parsed using only 1) application rules, and 2) type-raising and composition rules in conjunction for the purpose of flexibility in the order of argument application. In this case, the type-raising-and-composition combination can be realized in the pregroup grammar, as the overall effect is just that of a contraction. For instance, the derivation

$$B_{>}\,\dfrac{T_{>}\,\dfrac{X}{Z/(Z\backslash X)} \qquad Z\backslash X/A}{Z/A}$$

just corresponds to $X \cdot X^{r}ZA^{l} \to ZA^{l}$ in pregroups. Indeed, both the CCG version of this derivation and the PG version of this derivation leads to the same final DisCoCat string diagram.

Thus, it seems that although the system we have devised in this chapter would also work reasonably well in the equivalent PG-DisCoCat. Hence, in our internal wiring catalogue, in addition to providing the CCG type for a particular internal wiring diagram, we also provide corresponding pregroup type.

7.3 Catalogue of internal wirings

This section contains a big catalogue of internal wirings, along with their associated CCG (and pregroup) types. Note the 'pregroup types' are essentially the types of the words in DisCoCat, where we have kept track of left and right duals (though such accountancy is unnecessary in $\mathbf{FVect}_{\mathbb{R}}$). Each internal wiring will come with an example sentence or phrase.

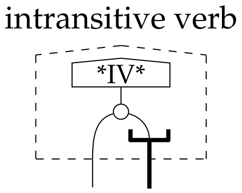
Many of the wirings provided here (e.g. verbs, adjectives, relative pronouns) are essentially the same as those in [CW]. However, many of the other wirings differ - in particular, the ones in this chapter are often *simpler*. This is likely a result of the fact that I have not bothered to ensure that certain grammatical behaviours hold in this system (such as the stacking of certain adverbs). We note also that the internal wiring system we come up in this chapter is *necessarily* distinct from the one in [CW], since, as remarked in subsection 3.5.4, not everything that can be done in pregroups, can be done in categorial grammar. As such, it was not possible to simply 'port over' the wirings from pregroups to CCG.

The catalogue provided here is very much incomplete, and is only meant to serve as a proof of concept. In particular, more subcategories could be introduced to deal with additional grammatical cases.

7.3.1 Verbs

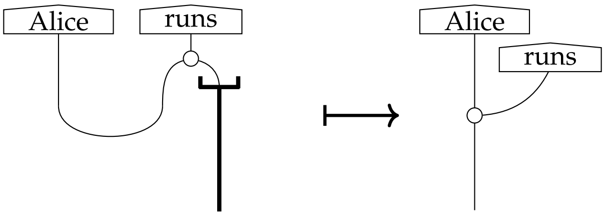
We separate these into the usual subcategories based on valency (intransitive, transitive, ditransitive). The general theme is that they take in nouns to return sentences.

Intransitive verbs (IV)

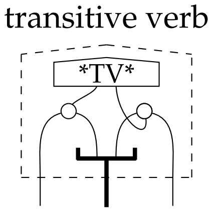


CCG: $[N]\backslash N$
Pregroup: $N^r[N]$
Example: "Alice runs."

$$<\ \dfrac{\dfrac{\text{Alice}}{N} \qquad \dfrac{\text{runs}}{[N]\backslash N}}{[N]}$$

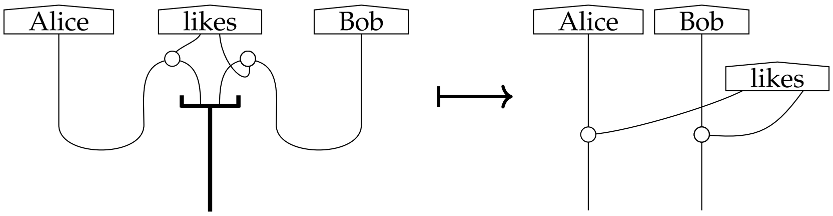


Transitive verbs (TV)



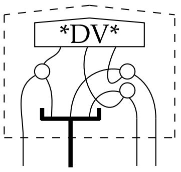
CCG: $[NN]\backslash N/N$
Pregroup: $N^r[NN]N^l$
This CCG type is essentially $\approx IV/N$.
Example: "Alice likes Bob."

$$\dfrac{\dfrac{\text{Alice}}{N} \qquad \dfrac{\dfrac{\text{likes}}{[NN]\backslash N/N} \quad \dfrac{\text{Bob}}{N}}{[NN]\backslash N}>}{[NN]}<$$



Ditransitive verb (DV)

ditransitive verb



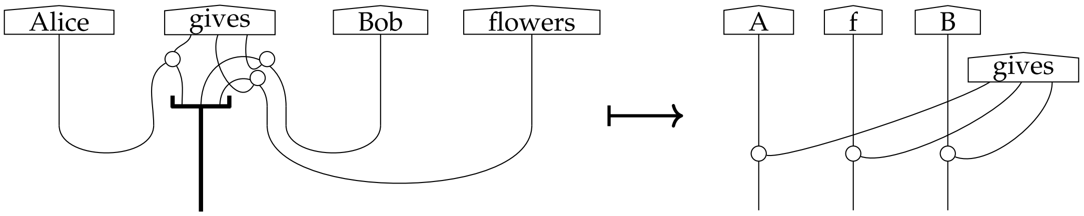
CCG: $[NNN]\backslash N/N/N$

Pregroup: $N^r[NNN]N^lN^l$

This CCG type is essentially $\approx TV/N$.

Example: "Alice gives Bob flowers."

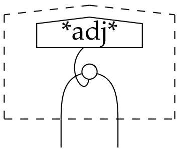
$$\dfrac{\dfrac{\text{Alice}}{N} \qquad \dfrac{\dfrac{\dfrac{\text{gives}}{[NNN]\backslash N/N/N} \quad \dfrac{\text{Bob}}{N}}{[NNN]\backslash N/N}> \quad \dfrac{\text{flowers}}{N}}{[NNN]\backslash N}>}{[NNN]}<$$



7.3.2 Adjectives

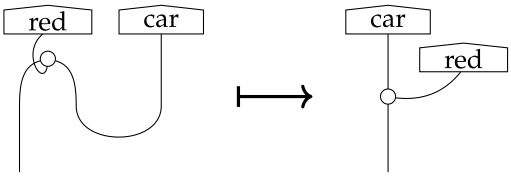
These take in a noun and return a noun.

adjective



CCG: N/N
Pregroup: NN^l
Example: "red car"

$$> \frac{\dfrac{\text{red}}{N/N} \quad \dfrac{\text{car}}{N}}{\mathrm{N}}$$

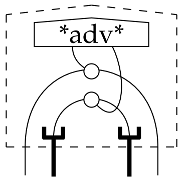


7.3.3 Adverbs

We subcategorise these based on both the valency of the input verb, as well as based on whether the adverb occurs before or after the verb (attributive or predicative).

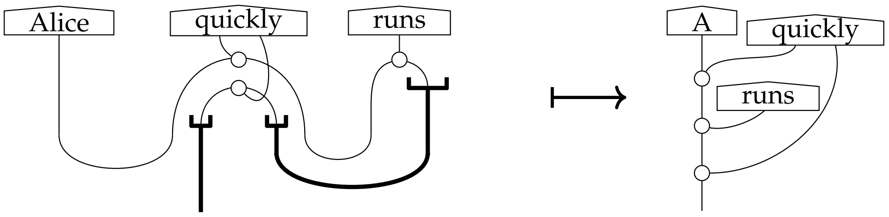
attr. adv$_{\mathrm{IV}}$

attributive adverb$_{\mathrm{IV}}$



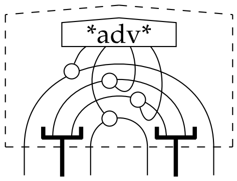
CCG: $[N]\backslash N/([N]\backslash N)$
Pregroup: $N^r[N][N^l]N$
This CCG type is exactly = IV/IV.
Example: "Alice quickly runs."

$$< \frac{\dfrac{\text{Alice}}{N} \quad > \dfrac{\dfrac{\text{quickly}}{[N]\backslash N/([N]\backslash N)} \quad \dfrac{\text{runs}}{[N]\backslash N}}{[N]\backslash N}}{[N]}$$



attr. adv$_{\mathrm{TV}}$

attributive adverb$_{\mathrm{TV}}$



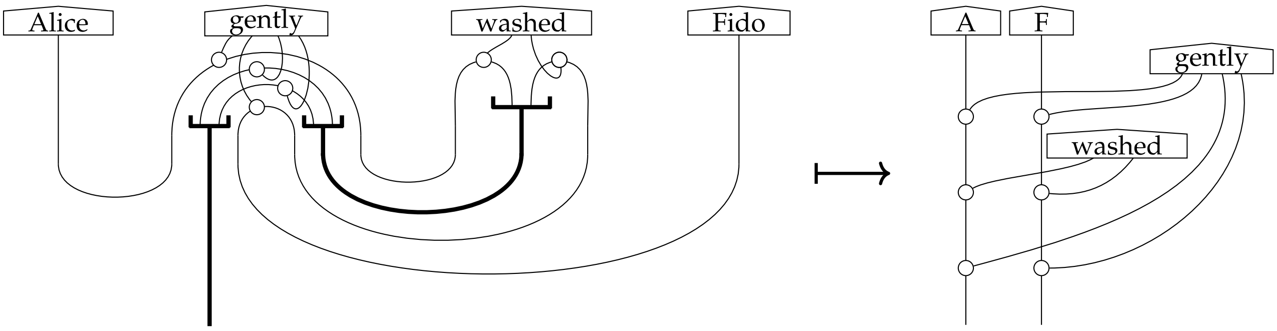
CCG: $[NN]\backslash N/N/([NN]\backslash N/N)$

Pregroup: $N^r[NN]N^lN^{ll}[N^lN^l]N$

This CCG type is exactly = TV/TV.

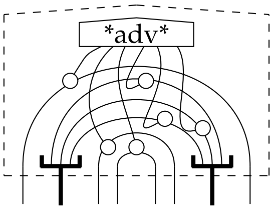
Example: "Alice gently washed Fido."

$$\dfrac{\dfrac{\text{Alice}}{N} \qquad \dfrac{\dfrac{\dfrac{\text{gently}}{[NN]\backslash N/N/([NN]\backslash N/N)} \quad \dfrac{\text{washed}}{[NN]\backslash N/N}}{[NN]\backslash N/N}{>} \quad \dfrac{\text{Fido}}{N}}{[NN]\backslash N}{>}}{[NN]}{<}$$



attr. adv$_{\mathrm{DV}}$

attributive adverb$_{\mathrm{DV}}$



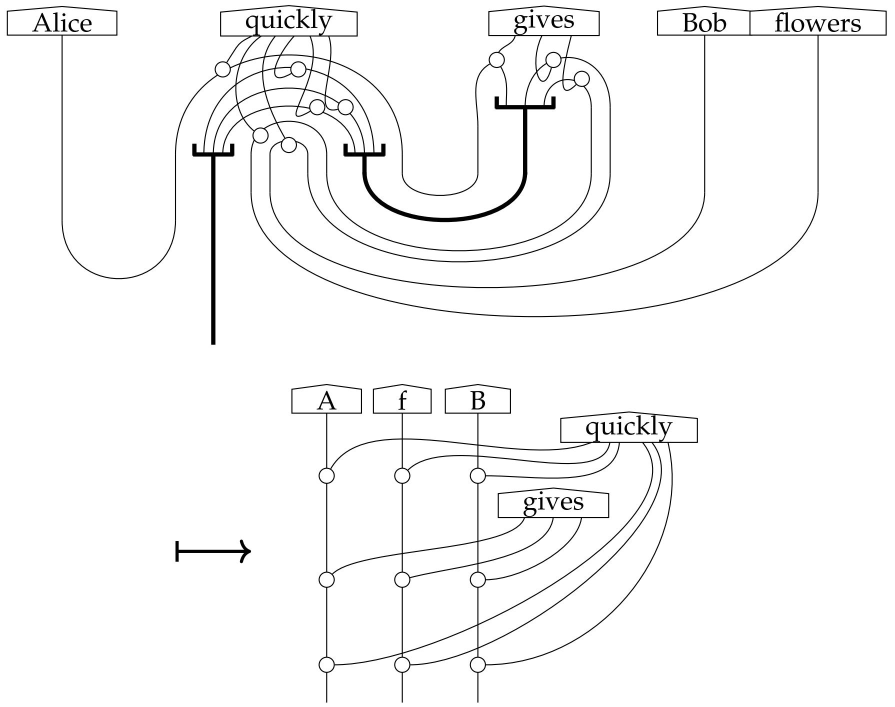
CCG: $[NNN]\backslash N/N/N/([NNN]\backslash N/N/N)$

Pregroup: $N^r[NNN]N^lN^lN^{ll}N^{ll}[N^lN^lN^l]N$

This CCG type is exactly = DV/DV.

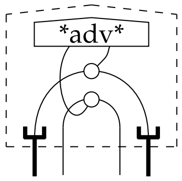
Example: "Alice quickly gives Bob flowers."

$$\dfrac{\dfrac{\text{Alice}}{N} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\text{quickly}}{[NNN]\backslash N/N/N/([NNN]\backslash N/N/N)} \quad \dfrac{\text{gives}}{[NNN]\backslash N/N/N}}{[NNN]\backslash N/N/N}> \quad \dfrac{\text{Bob}}{N}}{[NNN]\backslash N/N}> \quad \dfrac{\text{flowers}}{N}}{[NNN]\backslash N}>}{[NNN]}<$$



pred. adv$_{\text{IV}}$

predicative adverb$_{\text{IV}}$



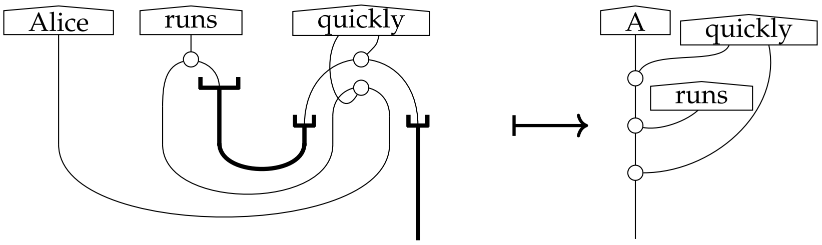
CCG: $[N]\backslash N\backslash([N]\backslash N)$

Pregroup: $[N^r]N^{rr}N^r[N]$

This CCG type is exactly = $IV\backslash IV$.

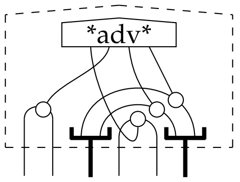
Example: "Alice runs quickly."

$$\dfrac{\dfrac{\text{Alice}}{N} \qquad \dfrac{\dfrac{\text{runs}}{[N]\backslash N} \quad \dfrac{\text{quickly}}{[N]\backslash N\backslash([N]\backslash N)}}{[N]\backslash N}<}{[N]}<$$



pred. adv$_{\mathrm{TV}}$

predicative adverb$_{\mathrm{TV}}$



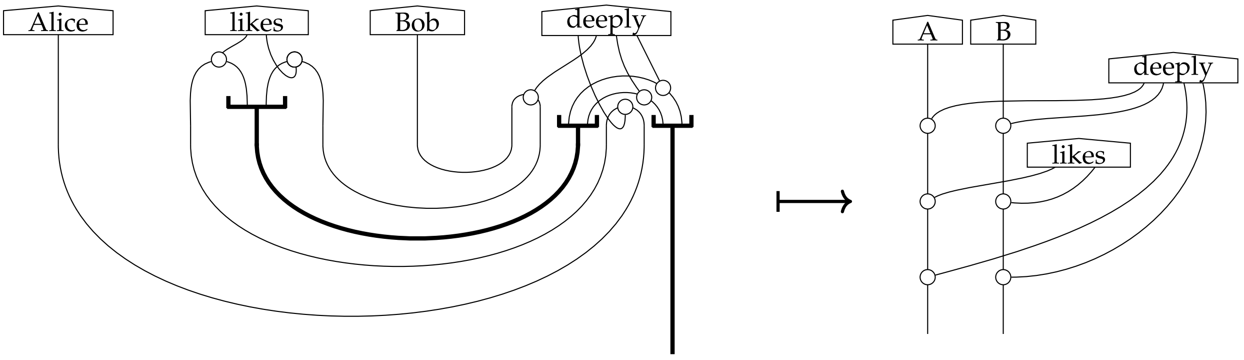
CCG: $[NN]\backslash N\backslash([NN]\backslash N/N)\backslash N$

Pregroup: $N^rN[N^rN^r]N^{rr}N^r[NN]$

This CCG type is essentially $\approx IV\backslash TV\backslash N$.

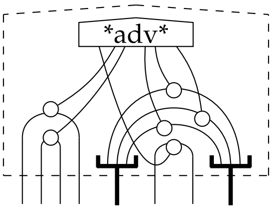
Example: "Alice likes Bob <u>deeply</u>."

$$\dfrac{\dfrac{\text{Alice}}{N}\quad \dfrac{\dfrac{\text{likes}}{[NN]\backslash N/N}\quad \dfrac{\dfrac{\text{Bob}}{N}\quad \dfrac{\text{deeply}}{[NN]\backslash N\backslash([NN]\backslash N/N)\backslash N}}{[NN]\backslash N\backslash([NN]\backslash N/N)}<}{[NN]\backslash N}<}{[NN]}<$$



pred. adv$_{\mathrm{DV}}$

predicative adverb$_{\mathrm{DV}}$



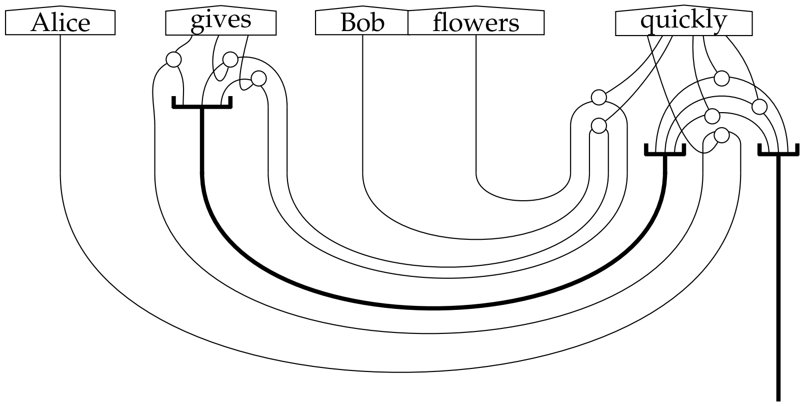
CCG: $[NNN]\backslash N\backslash([NNN]\backslash N/N/N)\backslash N\backslash N$

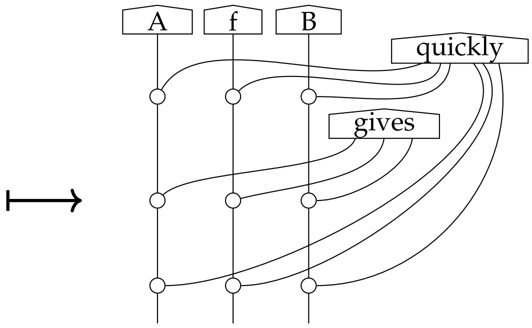
Pregroup: $N^rN^rNN[N^rN^rN^r]N^{rr}N^r[NNN]$

This CCG type is essentially $\approx IV\backslash DV\backslash N\backslash N$.

Example: "Alice gives Bob flowers quickly."

$$< \frac{\frac{\text{Alice}}{N} \quad < \frac{\frac{\text{gives}}{[NNN]\backslash N/N/N} \quad < \frac{\frac{\text{Bob}}{N} \quad < \frac{\frac{\text{flowers}}{N} \quad \frac{\text{quickly}}{[NNN]\backslash N\backslash([NNN]\backslash N/N/N)\backslash N\backslash N}}{[NNN]\backslash N\backslash([NNN]\backslash N/N/N)\backslash N}}{[NNN]\backslash N\backslash([NNN]\backslash N/N/N)}}{[NNN]\backslash N}}{[NNN]}$$





Comments

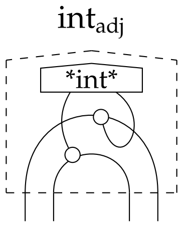
We note that the adverb typings we have provided allow for stacking of attributive adverbs, and of predicative adverbs in the case of IV valency - e.g. "Alice thoroughly, gently washed Fido". It does *not* seem to allow for stacking of pred. adv$_{\mathrm{TV}}$ and pred. adv$_{\mathrm{DV}}$.

Also, there is arguably a third obvious subcategory which we did not deal with - the case where the adverb appears before the sentence, like in "Suddenly, Alice spoke."

7.3.4 Intensifiers

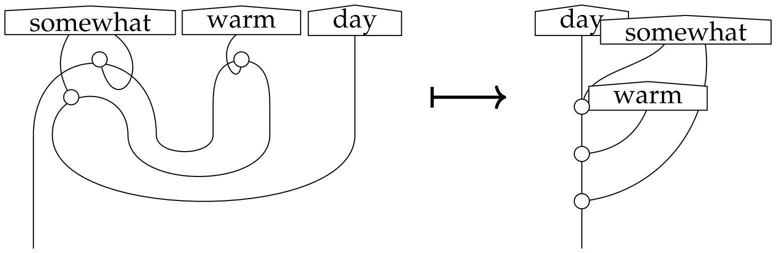
As with adverbs, these need to be subcategorised based according to the adverb subcategories. However, there is no further subcategorisation introduced, since intensifiers generally only appear before the adjective or adverb they modify - i.e. we do not say "red very". The general theme is that they take in adjectives/adverbs and return adjectives/adverbs.

int$_{\textbf{adj}}$

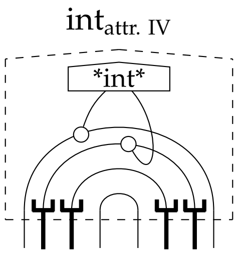


CCG: $N/N/(N/N)$
Pregroup: $NN^lN^{ll}N^l$
This CCG type is exactly = adj/adj
Example: "somewhat warm day"

$$\dfrac{\dfrac{\dfrac{\text{somewhat}}{N/N/(N/N)} \quad \dfrac{\text{warm}}{N/N}}{N/N}> \qquad \dfrac{\text{day}}{N}}{N}>$$



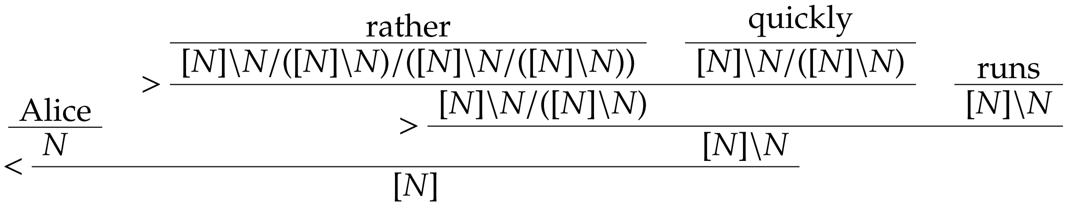
int$_{\textbf{attr. IV}}$

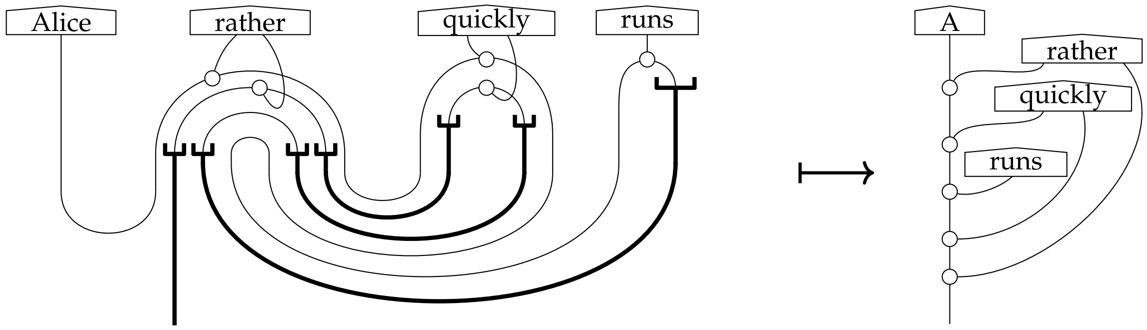


CCG: $[N]\backslash N/([N]\backslash N)/([N]\backslash N/([N]\backslash N))$
Pregroup: $N^r[N][N^l]NN^l[N^{ll}][N^l]N$

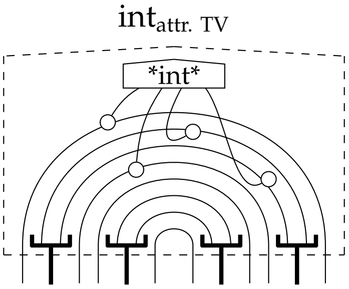
This CCG type is exactly = attr. adv_{IV}/attr. adv_{IV}

Example: "Alice rather quickly runs."


rather
$[N]\backslash N/([N]\backslash N)/([N]\backslash N/([N]\backslash N))$
quickly
$[N]\backslash N/([N]\backslash N)$
runs
$[N]\backslash N$
Alice
N
>
$[N]\backslash N/([N]\backslash N)$
>
$[N]\backslash N$
<
$[N]$



int$_{\text{attr. TV}}$

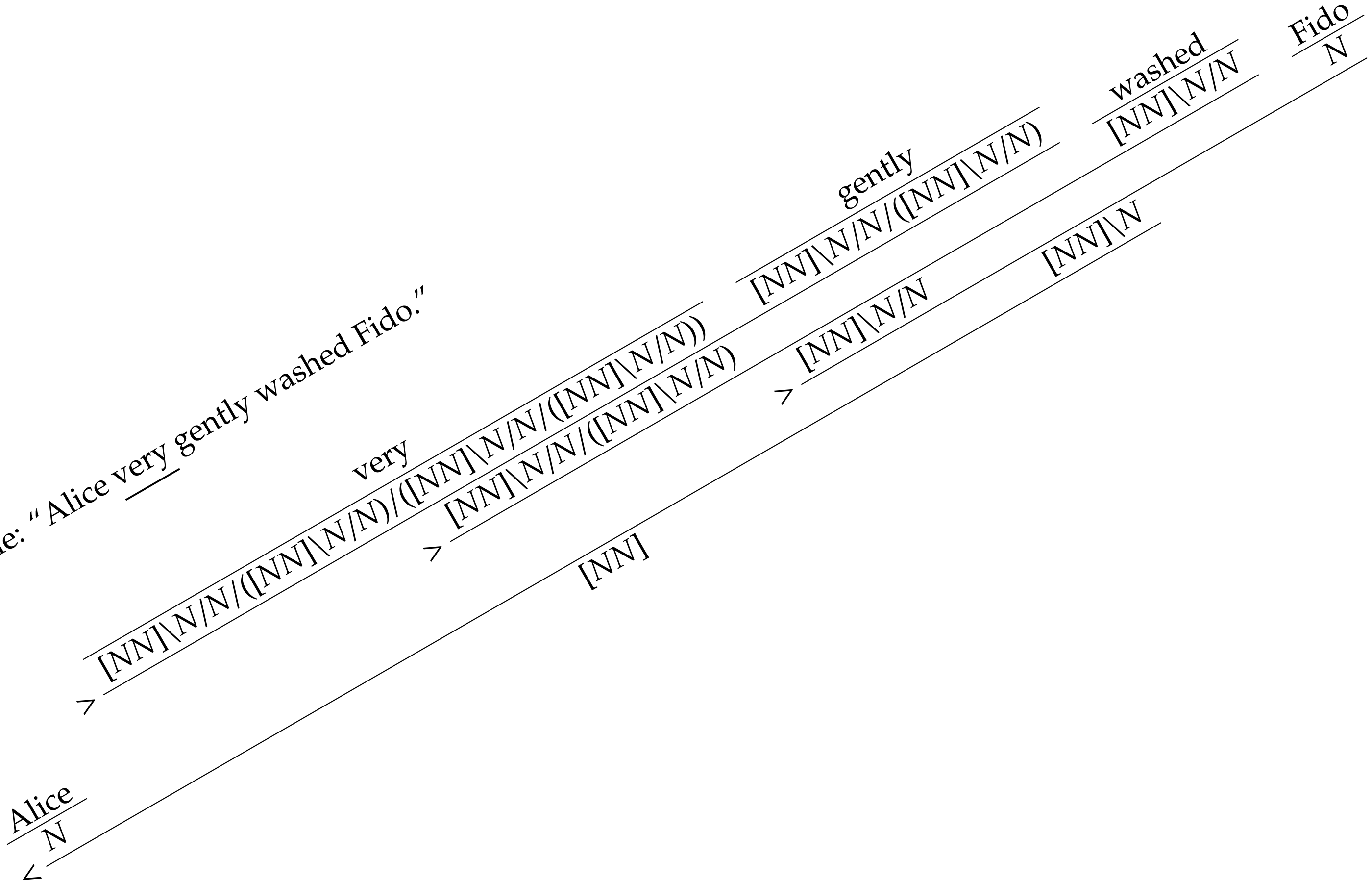


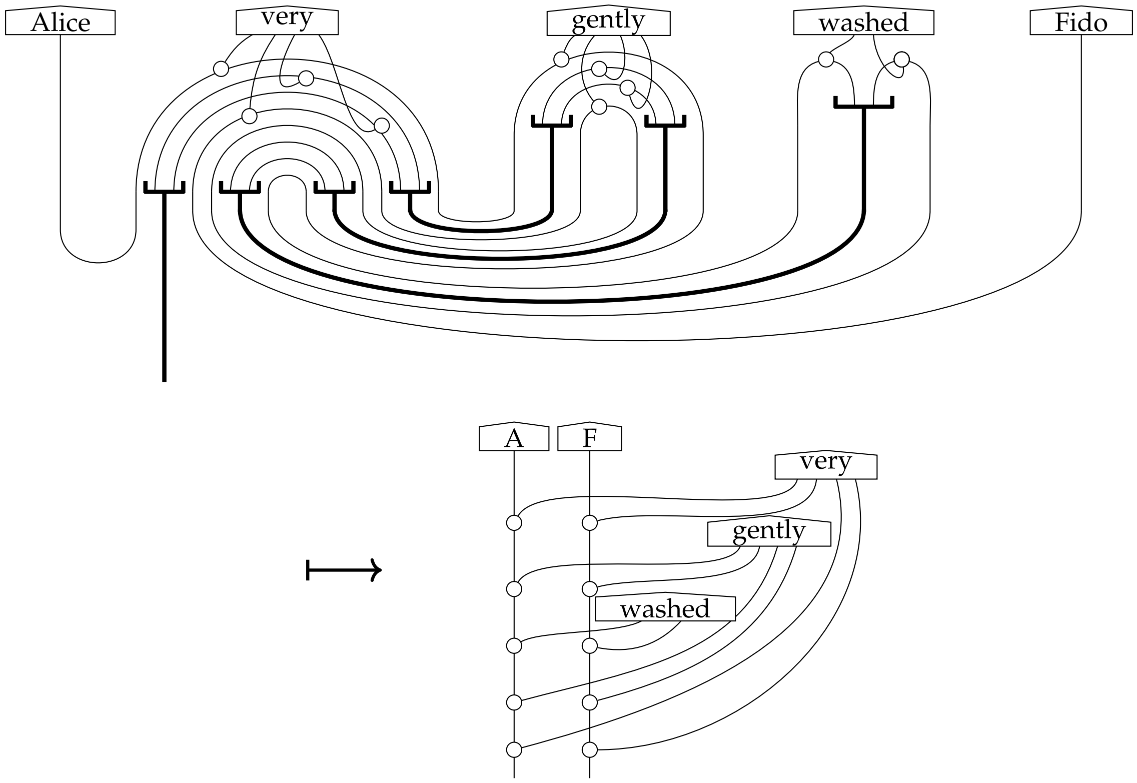
CCG: $[NN]\backslash N/N/([NN]\backslash N/N)/([NN]\backslash N/N/([NN]\backslash N/N))$

Pregroup: $N^r[NN]N^lN^{ll}[N^lN^l]NN^l[N^{ll}N^{ll}]N^{lll}N^{ll}[N^lN^l]N$

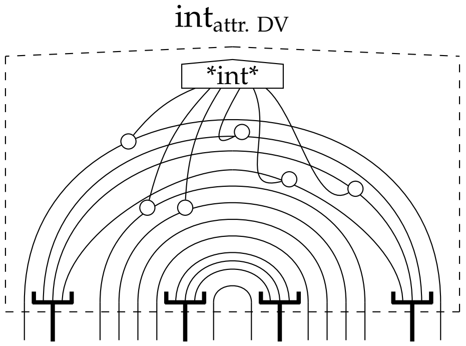
This CCG typing is exactly = attr. adv_{TV} / attr. adv_{TV}

Example: “Alice very gently washed Fido.”





int$_{\text{attr. DV}}$

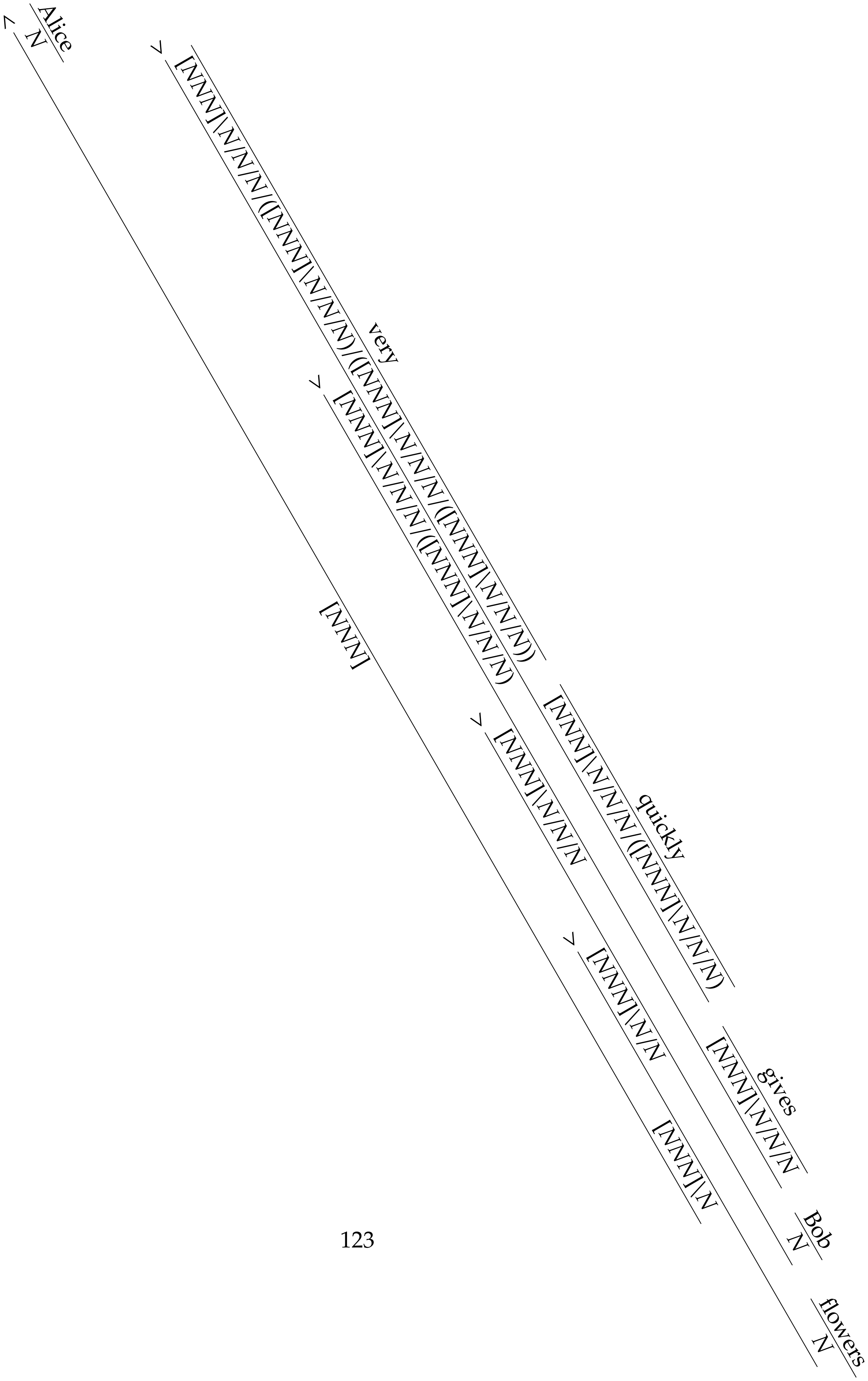


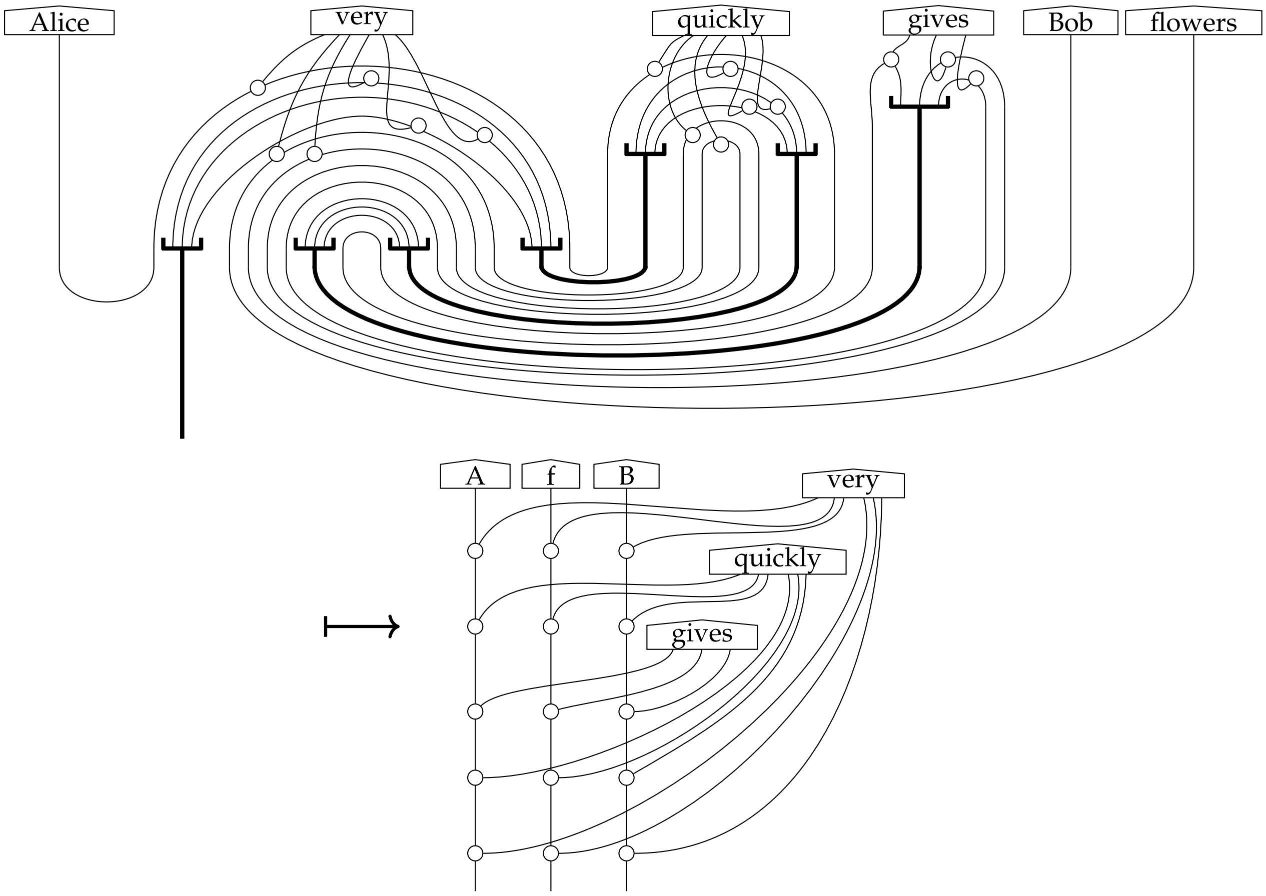
CCG: $[NNN]\backslash N/N/N/([NNN]\backslash N/N/N)/([NNN]\backslash N/N/N/([NNN]\backslash N/N/N))$

Pregroup: $N^r[NNN]N^lN^lN^{ll}N^{ll}[N^lN^lN^l]NN^l[N^{ll}N^{ll}N^{ll}]N^{lll}N^{lll}N^{ll}N^{ll}[N^lN^lN^l]N$

This CCG typing is exactly = attr. adv$_{\text{DV}}$ / attr. adv$_{\text{DV}}$

Example: “Alice $\underline{\text{very}}$ quickly gives Bob flowers.”


Alice
N
very
[N\N]\N/N/N/([N\N]\N/N/N)/([N\N]\N/N/N)
quickly
[N\N]\N/N/N/([N\N]\N/N/N)
gives
[N\N]\N/N/N
Bob
N
flowers
N
>
[N\N]\N/N/N/([N\N]\N/N/N)
>
[N\N]\N/N/N
>
[N\N]\N/N
>
[N\N]\N
<
[N\N]



Comments

We note that we could have chosen to type attributive intensifiers the same way we typed attributive adverbs - attributive adverbs are able to stack, and we would have obtained the correct connectivity with such an approach. However, typing intensifiers the same way we type adverbs would allow for ungrammatical phrases like “very runs”. Thus, we opted to differentiate the two.

We note also that there are also obvious intensifier cases for the predicative adverbs, which we have skipped over:

int$_{\textbf{pred. IV}}$

“Alice runs too quickly.”

int$_{\textbf{pred. TV}}$

“Alice likes Bob very deeply.”

int$_{\textbf{pred. DV}}$

“Alice gives Bob flowers <u>very</u> quickly.”

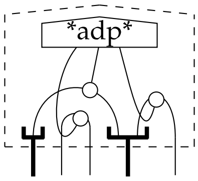
We also avoided dealing with cases where the adverb appears before the sentence: “<u>Rather</u> suddenly, Alice spoke”.

7.3.5 Adpositions

Adpositions are subcategorised based on the valency of the verb they modify. We have not considered cases where the adpositions appear at the front of the sentence - for instance, “<u>On</u> Saturday, Alice met Bob”.

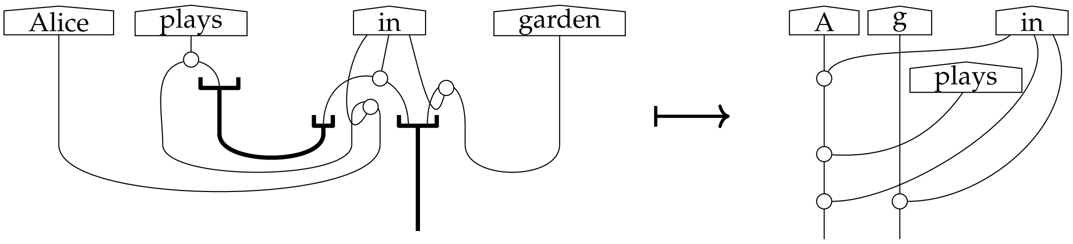
adp$_{\textbf{IV}}$

adposition$_{\text{IV}}$

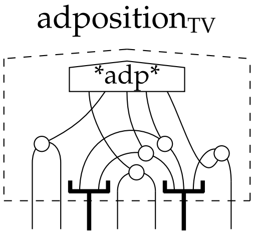


CCG: $[NN]\backslash N/N\backslash([N]\backslash N)$
Pregroup: $[N^r]N^{rr}N^r[NN]N^l$
This CCG type is exactly = $TV\backslash IV$
Example: “Alice plays <u>in</u> the garden.”

$$\cfrac{\cfrac{\text{Alice}}{N} \quad \cfrac{\cfrac{\cfrac{\text{plays}}{[N]\backslash N} \quad \cfrac{\text{in}}{[NN]\backslash N/N\backslash([N]\backslash N)}}{[NN]\backslash N/N}{<} \quad \cfrac{\text{garden}}{N}}{[NN]\backslash N}{>}}{[NN]}{<}$$



adp$_{\text{TV}}$



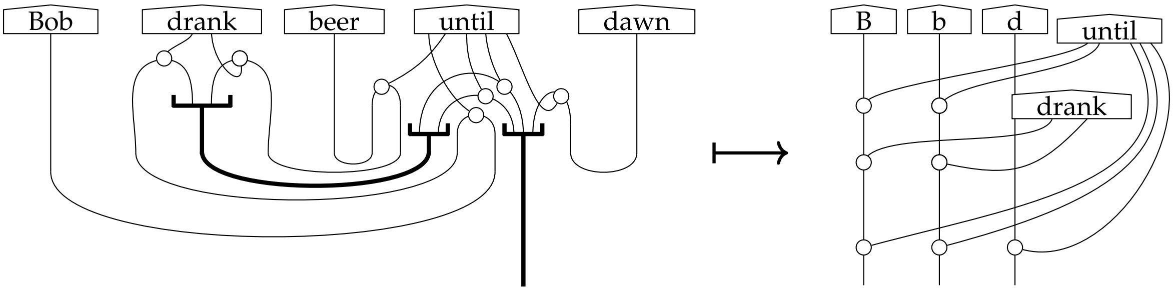
CCG: $[NNN]\backslash N/N\backslash([NN]\backslash N/N)\backslash N$

Pregroup: $N^rN[N^rN^r]N^{rr}[NNN]N^l$

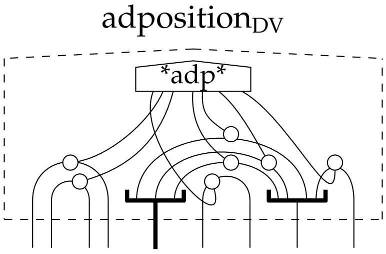
This CCG type is essentially $\approx TV\backslash TV\backslash N$

Example: "Bob drank beer <u>until</u> dawn."

$$\frac{\frac{\text{Bob}}{N} \qquad \frac{\frac{\frac{\text{drank}}{[NN]\backslash N/N} \qquad \frac{\frac{\text{beer}}{N}\qquad \frac{\text{until}}{[NNN]\backslash N/N\backslash([NN]\backslash N/N)\backslash N}}{[NNN]\backslash N/N\backslash([NN]\backslash N/N)}<}{[NNN]\backslash N/N}< \qquad \frac{\text{dawn}}{N}}{[NNN]\backslash N}>}{[NNN]}<$$



adp$_{\text{DV}}$

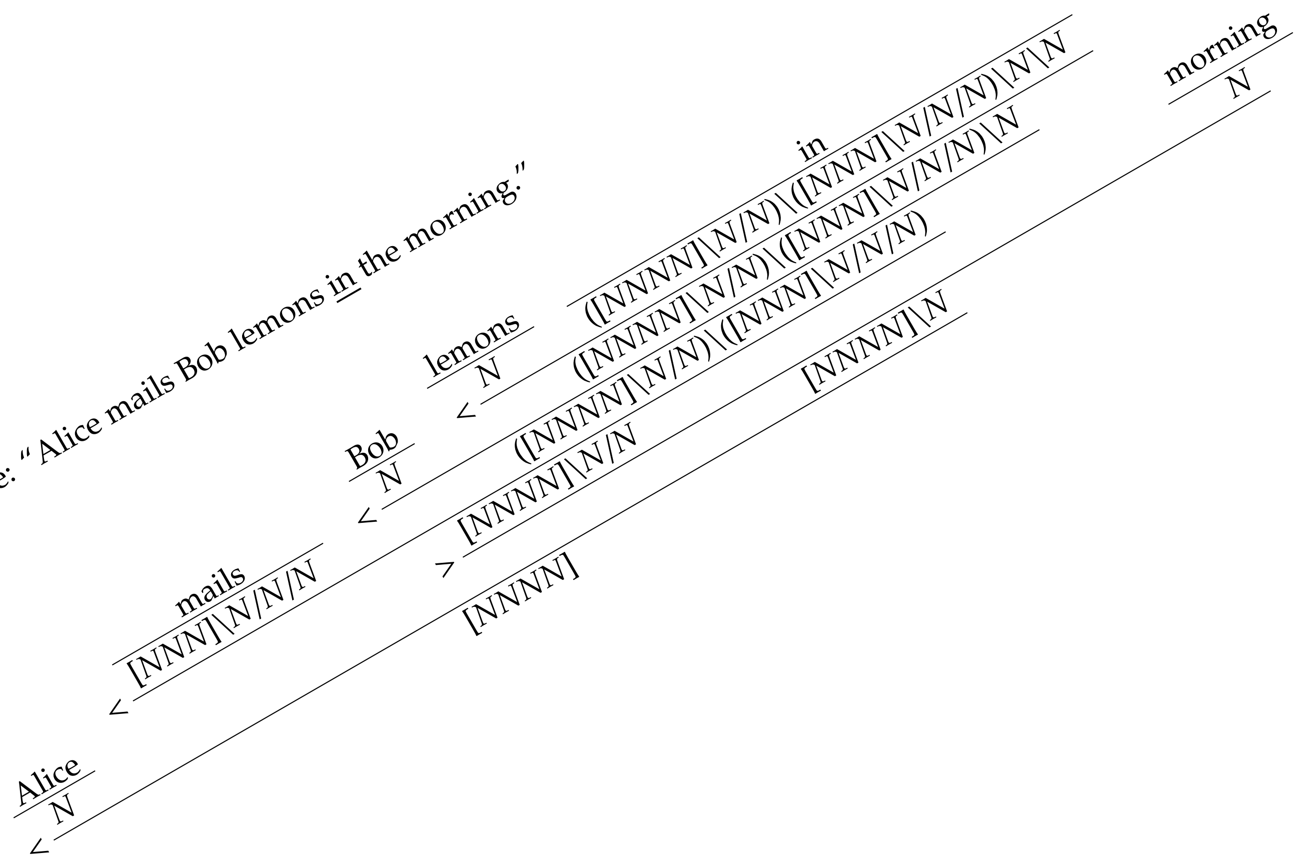


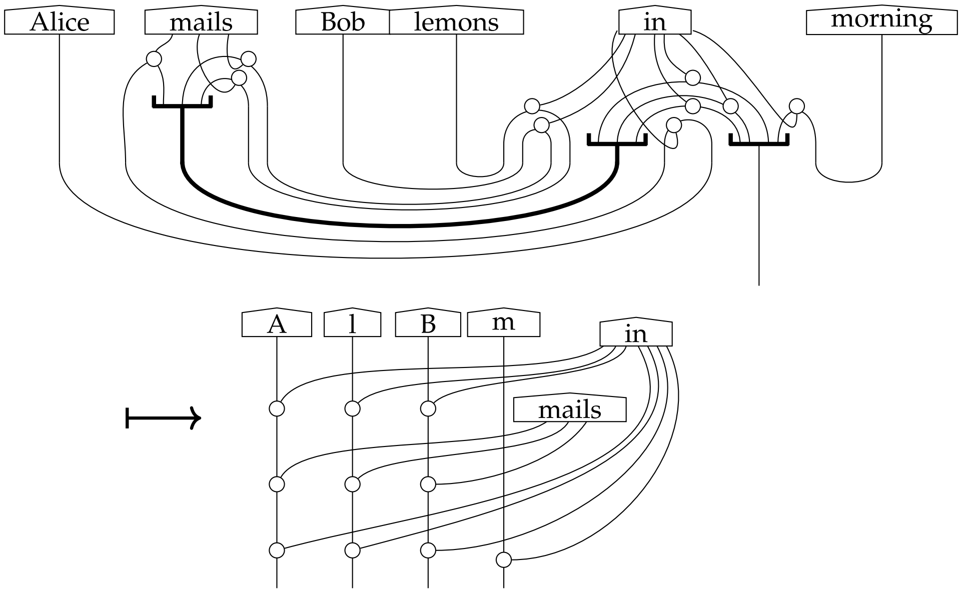
CCG: $([NNNN]\backslash N/N)\backslash([NNN]\backslash N/N/N)\backslash N\backslash N$

Pregroup: $N^rN^rNN[N^rN^rN^r]N^{rr}N^r[NNNN]N^l$

This CCG type is essentially $TV\backslash DV\backslash N\backslash N$

Example: "Alice mails Bob lemons in the morning."


Alice
N
mails
[NNN]\N/N/N
Bob
N
lemons
N
in
([NNNN]\N/N)\([NNN]\N/N/N)\N\N
morning
N
<
<
<
<
([NNNN]\N/N)\([NNN]\N/N/N)\N
([NNNN]\N/N)\([NNN]\N/N/N)
[NNNN]\N/N
>
[NNNN]\N
[NNNN]



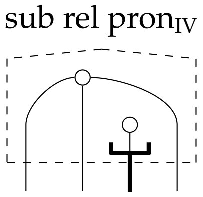
Comments

A problem with the typings we have come up with thus far is that adpositions do not interact well with TV and DV predicative adverbs. That is, these types do not seem to allow us to properly parse a sentence like “Bob drank beer merrily until dawn.” Problems like these can be resolved by introducing more fine-grained, case-based subcategories.

7.3.6 Relative pronouns

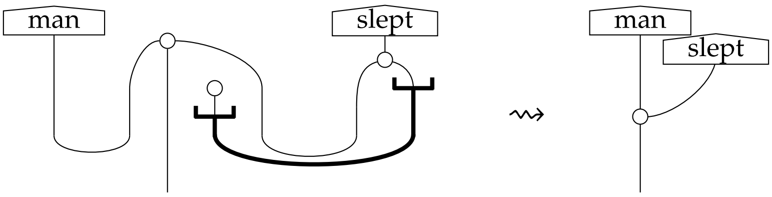
For this subsection, we note that in the TV and DV cases, the language circuits we obtain feature noun states that are directly fed into verbs, and do not reappear as open noun wires. In order to recover the proper set of noun wires, we need to insert a copy spider on the desired nouns. This step was also discussed in [CW].

sub rel pron$_{\text{IV}}$

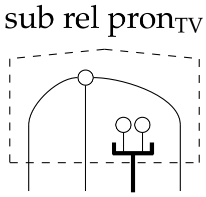


CCG: $N\backslash N/([N]\backslash N)$
Pregroup: $N^r N[N^l]N$
Example: “man who slept”

$$\dfrac{\dfrac{\text{man}}{N} \qquad \dfrac{\dfrac{\text{who}}{N\backslash N/([N]\backslash N)} \qquad \dfrac{\text{slept}}{[N]\backslash N}}{N\backslash N}>}{N}<$$



sub rel pron$_{\text{TV}}$

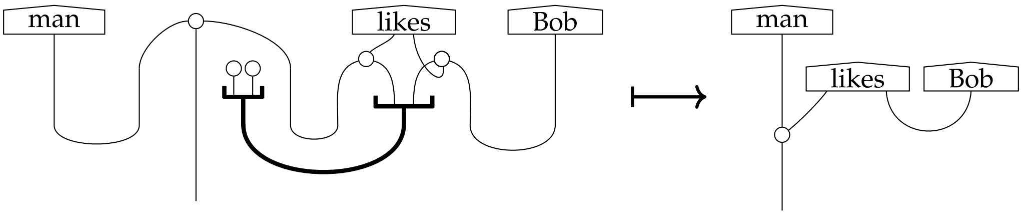


CCG: $N\backslash N/([NN]\backslash N)$

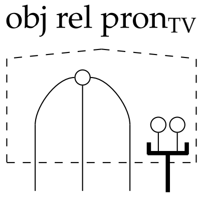
Pregroup: $N^r N[N^l N^l]N$

Example: "man <u>who</u> likes Bob"

$$\dfrac{\dfrac{\text{man}}{N} \qquad \dfrac{\dfrac{\text{who}}{N\backslash N/([NN]\backslash N)} \qquad \dfrac{\dfrac{\text{likes}}{[NN]\backslash N/N} \qquad \dfrac{\text{Bob}}{N}}{[NN]\backslash N}>}{N\backslash N}>}{N}<$$



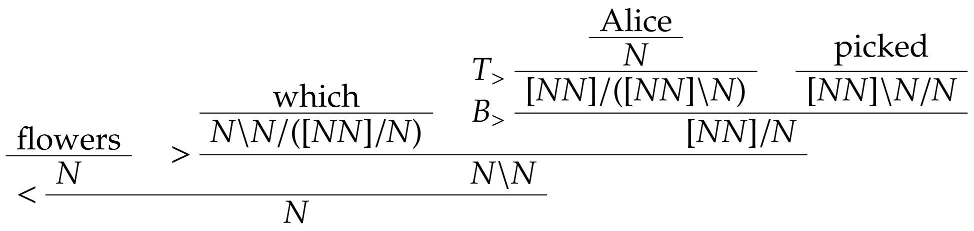
obj rel pron$_{\text{TV}}$

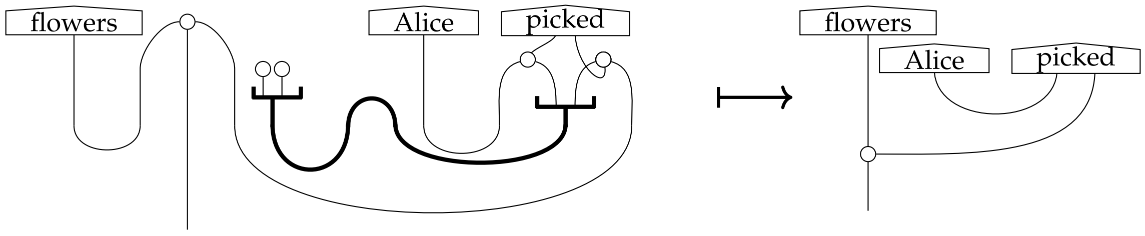


CCG: $N\backslash N/([NN]/N)$

Pregroup: $N^r N N^{ll}[N^l N^l]$

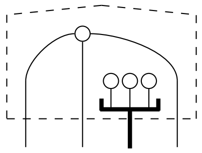
Example: "flowers <u>which</u> Alice picked"





sub rel pron$_{\text{DV}}$

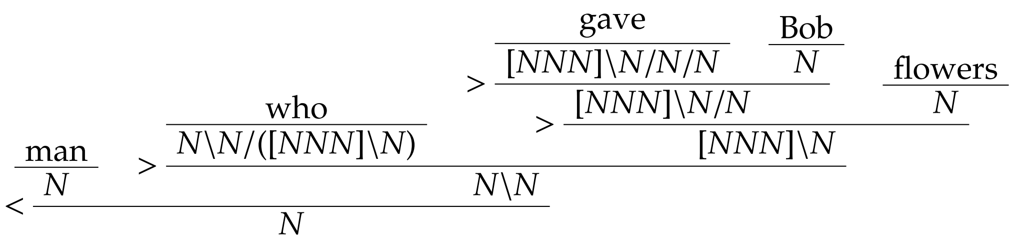
sub rel pron$_{\text{DV}}$

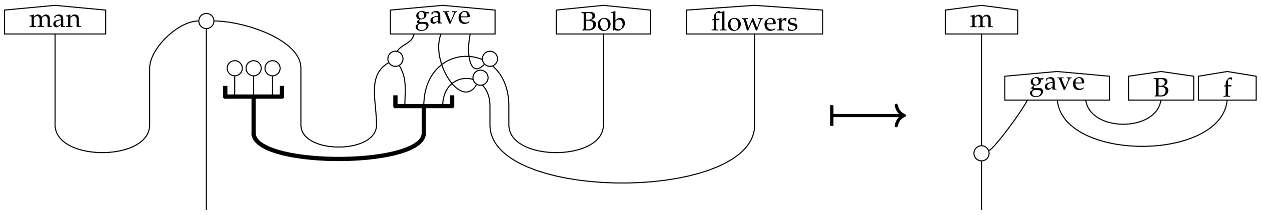


CCG: $N\backslash N/([NNN]\backslash N)$

Pregroup: $N^rN[N^lN^lN^l]N$

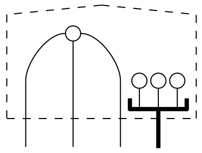
Example: "man <u>who</u> gave Bob flowers"



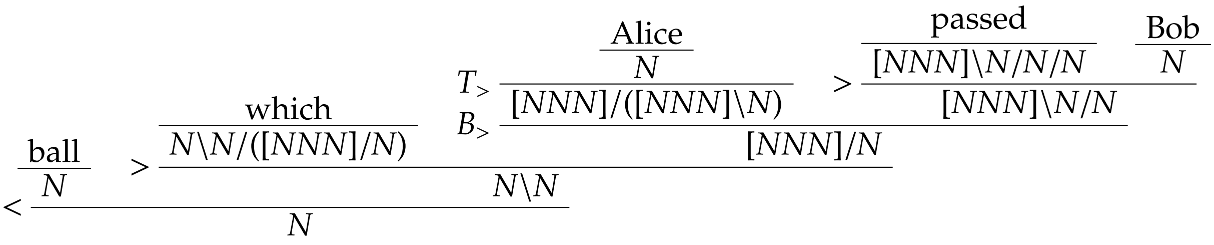


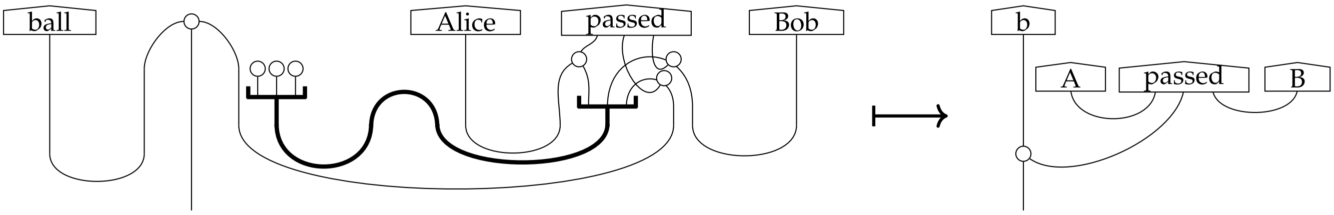
(dir) obj rel pron$_{\text{DV}}$

dobj rel pron$_{\text{DV}}$



CCG: $N\backslash N/([NNN]/N)$
Pregroup: $N^r N N^{ll}[N^l N^l N^l]$
Example: "ball <u>which</u> Alice passed Bob"

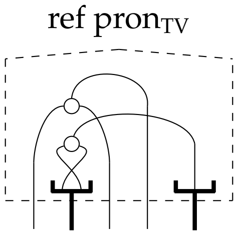




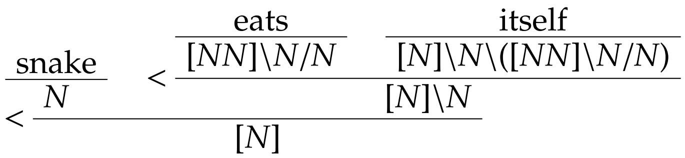
7.3.7 Reflexive pronouns

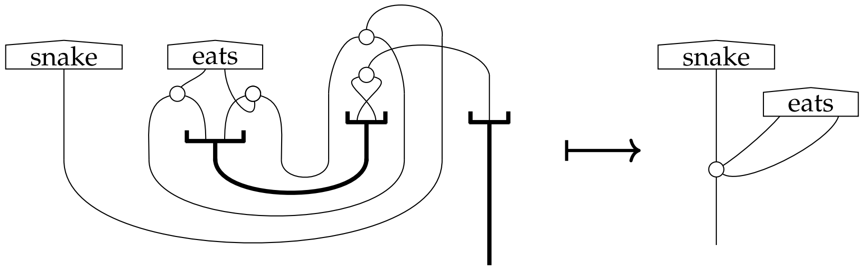
These take in a verb of valency n and return a verb of valency $n-1$. We note that our wirings differ slightly from those in [CW], as our version contains an extra braid.

ref pron$_{\mathrm{TV}}$

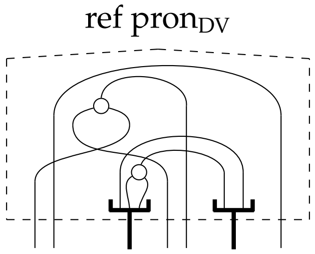


CCG: $[N]\backslash N\backslash([NN]\backslash N/N)$
Pregroup: $N[N^r N^r]N^{rr}N^r[N]$
This CCG typing is exactly $= IV\backslash TV$
Example: "The snake eats <u>itself</u>."





rel pron$_{\text{DV}}$



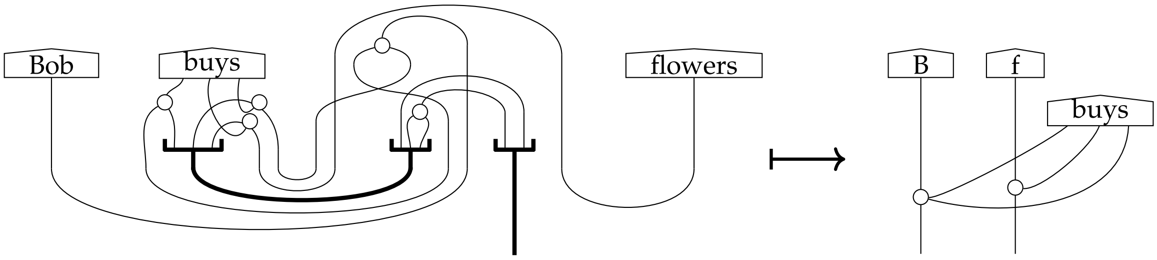
CCG: $[NN]\backslash N/N\backslash([NNN]\backslash N/N/N)$

Pregroup: $NN[N^rN^rN^r]N^{rr}N^r[NN]N^l$

This CCG typing is exactly = $TV\backslash DV$

Example: "Bob buys <u>himself</u> flowers."

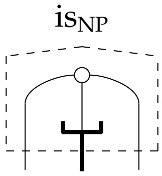
$$< \dfrac{\dfrac{\text{Bob}}{N} \qquad > \dfrac{< \dfrac{\dfrac{\text{buys}}{[NNN]\backslash N/N/N} \qquad \dfrac{\text{himself}}{[NN]\backslash N/N\backslash([NNN]\backslash N/N/N)}}{[NN]\backslash N/N} \qquad \dfrac{\text{flowers}}{N}}{[NN]\backslash N}}{[NN]}$$



7.3.8 Copula "is"

We deal with the two cases of "is" discussed in section 6.2 - one that links to a noun, and one that links to an adjective.

$\mathbf{is}_{\mathbf{NP}}$



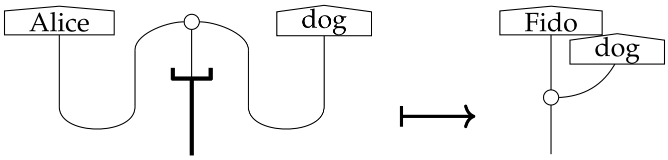
CCG: $[N]\backslash N/N$

Pregroup: $N^l[N]N^r$

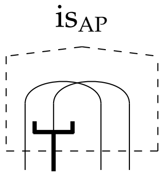
This CCG typing is exactly = IV/N

Example: “Fido is a dog.”

$$\cfrac{\cfrac{\text{Fido}}{N} \qquad \cfrac{\cfrac{\text{is}}{[N]\backslash N/N} \qquad \cfrac{\text{a dog}}{N}}{[N]\backslash N}\,>}{[N]}\,<$$



$\mathbf{is}_{\mathbf{AP}}$



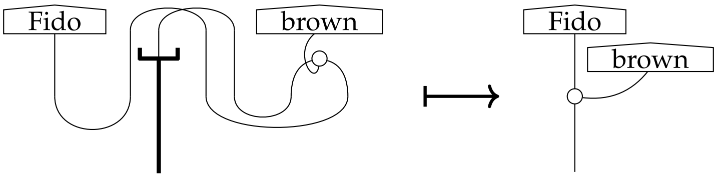
CCG: $[N]\backslash N/(N/N)$

Pregroup: $N^r[N]N^{ll}N^l$

This CCG type is exactly = IV/adj

Example: “Fido is brown.”

$$\cfrac{\cfrac{\text{Fido}}{N} \qquad \cfrac{\cfrac{\text{is}}{[N]\backslash N/(N/N)} \qquad \cfrac{\text{brown}}{N/N}}{[N]\backslash N}\,>}{[N]}\,<$$



Chapter 8

Conclusion and further remarks

To conclude, we list some of the contributions of this thesis.

- We attempted to motivate our theory of language circuits by looking at parses of example sentences based on established grammatical frameworks like CCG and DG. Based on these, we contended that the kind of grammatical structures that arise from our language circuit theory are closer to dependency structures than constituency structures. The latter places heavy emphasis on grouping together adjacent words into constituents, whereas our language circuits do not seem to care much for linear word order.

- We proposed a typing system which helps to formalise the exact behaviour of the different circuit components corresponding to different syntactic categories.

- We proposed an algorithm for converting (suitable) sentences into language circuit representations, using information given by the spaCy dependency parser. The same idea could be and perhaps would be better applied to other dependency parsers, where we have more knowledge/control over the underlying grammatical model.

- We proposed a catalogue of internal wirings and an associated CCG system, which can convert DisCoCat diagrams based on that CCG to language circuits. We noted that this catalogue of internal wirings would also be largely compatible with an appropriate pregroup-based DisCoCat.

We end on some open questions and speculations regarding language circuits - some of which have been discussed in the body of the thesis.

- As mentioned in subsection 5.1.1 - how do we assign noun wires? To put the concern another way, in our theory of language circuits there is a significant delineation between active words which receive wires and 'live on as entities', and all other words which seem to be static, one-off components. Yet in normal language and grammar, there is no clear delineation to be found which can motivate this. Why then is having noun wires a natural thing to do, and how do we decide which words to raise to this privileged active status? Perhaps there is no canonical choice of words to elevate to noun wires, and instead we simply make the choice on a case by case basis, according to our needs.

- What role exactly do internal wirings play? The approach of [CW] and chapter 7 seems rather unwieldy - we need to come up with fairly complicated, original PG/CCG systems. Of course, since these are original grammatical systems, there are no existing parsers, hampering their usefulness at a practical level. Perhaps for practical applications it would be better to first choose some established CCG say, and then attempt to provide internal wirings based on it, even if the resulting connectivity/topology of our language circuits does not quite match the theoretical ideal of chapter 5.1.

- How do we expand the fragment of English that we can model with diagrams? Without much effort, one can already see some obvious and intuitive extensions, which were not included in this thesis only due to constraints on space and time - for instance, adpositions/conjunctions that link sentences with sentences ("Bob drank until Alice arrived"). However, there are some other examples (e.g. "Bob dreamt he saw Alice" - do we need some sort of separate world for Bob's dream?) which seem like they will require a fair bit of extra thought.

Appendix A

Basic category theory

A.1 Basic concepts

Definition A.1.1. *A* ***category*** C *consists of*

- *a collection*[1] *of* ***objects*** $\mathrm{Ob}(C)$*. Objects are denoted by* $X, Y, Z, \ldots$
- *a collection of* ***morphisms*** $\mathrm{Mor}(C)$*. Morphisms are denoted* $f, g, h, \ldots$

such that

- *For each morphism* f*, there are given objects* $\mathrm{dom}(f)$*,* $\mathrm{cod}(f)$*, called the* ***domain*** *and* ***codomain*** *of* f*. We write* $f : A \to B$ *to indicate* $A = \mathrm{dom}(f)$*,* $B = \mathrm{cod}(f)$*.*
- *Given morphisms* $f : X \to Y$ *and* $g : Y \to Z$*, there is a morphism* $g \circ f : X \to Z$ *called the* ***composite*** *of* f *and* g*.*
- *For each object* X*, there is an* ***identity morphism*** $1_X : X \to X$*.*

These data are required to satisfy two laws:

- *Unit:* $f \circ 1_X = 1_Y \circ f = f$ *for all* $f : X \to Y$*.*
- *Associativity:* $h \circ (g \circ f) = (h \circ g) \circ f$ *for all* $f : X \to Y, g : Y \to Z, h : Z \to W$*.*

Morphisms in a category are also referred to as 'arrows' or 'maps'. Associativity means we may freely drop the parentheses and unambiguously write $h \circ g \circ f$. Often we will further drop the composition symbol and just write hgf.

[1] Russell's paradox implies that there is no set whose elements are "all sets", so for instance Ob(**Set**) is not itself a set. This is the reason the vague term "collection" is often used in definitions of category. We will ignore these underlying set-theoretic issues, as most introductory treatments of category theory do - indeed the search for the most useful set-theoretical foundations for category theory is itself a topic of research[Rie17].

Due to the set-theoretic issues discussed in the footnote, it is often useful to introduce adjectives that explicitly describe the size of a category.

Definition A.1.2. *A category is* ***locally small*** *if between any pair of objects, there is only a set's worth of morphisms.*

Definition A.1.3. *Given objects X, Y in a locally small category $\mathbf{C}$, we write*

$$\mathrm{Hom}_C(X, Y)$$

to denote the set of morphisms going $X \to Y$[2]. Such a set is called a ***hom-set****.*

There are many different names for morphisms with special properties. The most important is the notion of an isomorphism.

Definition A.1.4. *An* ***isomorphism*** *in a category $\mathbf{C}$ is an arrow $f : X \to Y$ such that there exists an arrow $h : Y \to X$ - the* ***inverse*** *of f - satisfying*

$$f \circ h = 1_Y, \qquad h \circ f = 1_X.$$

We usually write f^{-1} for the inverse of f, and say that X and Y are isomorphic - written $X \cong Y$ - if there exists an isomorphism $X \to Y$.

Example. Many familiar varities of mathematical objects form a category. The most basic is **Set**, the category whose objects are sets, and whose morphisms are functions between sets. Another elementary example is $\mathbf{Vect}_k$, the category whose objects are vector fields over the field k, and whose morphisms are k-linear transformations. We in particular will be interested in $\mathbf{FVect}_k$, the subcategory of finite dimensional vector spaces over the field k. Note all of these categories are locally small.

Example. Recall that a partial order on a set P is binary relation $\leq$ such that for all $x, y, z \in P$,

- $x \leq x$ (reflexivity)
- $x \leq y \wedge y \leq z$ implies $x \leq z$ (transitivity)
- $x \leq y \wedge y \leq x$ implies $x = y$ (antisymmetry).

A poset is set with a partial order. However, an equivalent definition of poset is as a category such that

[2] An alternate notation that is used for this hom-set is $C(X, Y)$.

- for any objects x, y, $\mathrm{Mor}(C)$ has at most one element
- if there are morphisms $x \to y$ and $y \to x$ then $x = y$.

That is, the objects of the posetal category is taken to be the set P, and the presence of a morphism from $x \to y$ indicates that $x \leq y$[3]. So the reflexivity rule is the presence of identity maps, and transitivity is composition of morphisms. Antisymmetry is captured by the additional requirement. A useful fact about posetal categories is that any equational statement between morphisms (e.g. any commutative diagram) is trivially satisfied, since there can only be one morphism between any two objects.

Definition A.1.5. *Given categories C, $\mathcal{D}$, there is a* ***product category*** *$C \times \mathcal{D}$ whose*

- *objects are ordered pairs (X, Y) where $X \in \mathrm{Ob}(C)$, $Y \in \mathrm{Ob}(\mathcal{D})$,*
- *morphisms are ordered pairs $(f, g) : (X, Y) \to (X', Y')$ where $f : X \to X' \in \mathrm{Mor}(C)$ and $g : Y \to Y' \in \mathrm{Mor}(\mathcal{D})$, and*
- *in which composition and identities are defined component-wise:*

$$\begin{aligned}(f', g') \circ (f, g) &= (f' \circ f, g' \circ g)\\ 1_{(X,Y)} &= (1_X, 1_Y).\end{aligned}$$

There is an obvious generalisation to products on $n \geq 3$ categories.

An important notion in category theory is that of duality. Given a category C, we can obtain an opposite or 'dual' category by formally 'reversing the directions of the morphisms'.

Definition A.1.6. *Given a category C, the* ***opposite category*** *C^{op} has*

- *the same objects as C, i.e. $\mathrm{Ob}(C^{op}) = \mathrm{Ob}(C)$*
- *a morphism $f^{op} : Y \to X$ for each morphism $f : X \to Y \in \mathrm{Mor}(C)$, and no others besides, i.e. $\mathrm{Hom}_{C^{op}}(Y, X) = \mathrm{Hom}_C(X, Y)$.*

Composition and identities are inherited from C, although there is a reversal of order for composition:

- *given $f^{op} : Y \to X$, $g^{op} : Z \to Y$, their composite is $f^{op} \circ g^{op} := (g \circ f)^{op}$*
- *the identity morphism for X in C^{op} is $1_X^{op} : X \to X$.*

Essentially, the process of 'reversing the directions of arrows', or 'exchanging the domains and codomains' exhibits a syntactical self-duality satisfied by the axioms for a category.

[3] We will use this convention, rather than taking $x \to y$ as meaning that $x \geq y$.

A.2 Functors and natural transformations

A key tenet of category theory is that objects should be considered together with an accompanying notion of structure-preserving morphism. Categories themselves are mathematical objects, and the structure-preserving morphisms between categories are functors.

Definition A.2.1. *Let C, $\mathcal{D}$ be categories. A* ***functor*** *$F : C \to \mathcal{D}$ consists of an object map assigning an object FX in $\mathrm{Ob}(\mathcal{D})$ to every X in $\mathrm{Ob}(C)$, and a morphism map assigning a morphism $Ff : FX \to FY$ in $\mathrm{Mor}(\mathcal{D})$ to every morphism $f : X \to Y$ in $\mathrm{Mor}(C)$, such that the* ***functoriality*** *conditions are satisfied:*

- *$F(fg) = F(f)F(g)$ for all morphisms $f : X \to Y$, $g : Y \to Z$ in C, and*
- *$F(1_X) = 1_{FX}$ for all $X \in \mathrm{Ob}(C)$.*

That is, a functor preserves domains and codomains, identity morphisms, and composition. Note that we use the same symbol to refer to both the object and morphism map - in practice this will not cause confusion.

Applying again the notion of duality, we note that a functor $F : C \to \mathcal{D}$ can be equivalently viewed as a functor $F : C^{op} \to \mathcal{D}^{op}$. For the new $C^{op} \to \mathcal{D}^{op}$ functor, simply take the same object map as the original, and for morphisms map $f^{op} : X \to Y$ to $(Ff)^{op}$. Although these functors technically have different domains and codomains and thus may be considered distinct functors, it is standard to use F to denote both the original functor $F : C \to \mathcal{D}$ and the 'opposite' functor $F : C^{op} \to \mathcal{D}^{op}$.

A **bifunctor** is just a functor whose domain is a product category. A **multifunctor** generalises this notion to a product of n categories.

Definition A.2.2. *Given a category C, the* ***bivariant hom-functor***

$$\mathrm{Hom}_C(-,-) : C^{op} \times C \to \mathbf{Set}$$

defined

$$\mathrm{Hom}_C(-,-)(X,Y) := \mathrm{Hom}_C(X,Y), \qquad \mathrm{Hom}_C(-,-)(f^{op}, f') := g \mapsto f' \circ g \circ f$$

This functor is sometimes also just called the 'hom-functor'.

Next we introduce the notion of naturality.

Definition A.2.3. *For functors $F, G : C \to \mathcal{D}$, a* ***natural transformation*** *$\eta : F \Rightarrow G$ is a family of maps $\{\eta_X : FX \to GX\}_{X \in \mathrm{Ob}(C)}$ in $\mathcal{D}$ such that for all maps $X \xrightarrow{f} Y$ in C, the 'naturality square'*

$$\begin{tikzcd}
FX \arrow[r, "\eta_X"] \arrow[d, "Ff"'] & GX \arrow[d, "Gf"] \\
FY \arrow[r, "\eta_Y"'] & GY
\end{tikzcd}$$

commutes.

To quote from [Rie17], in practice it is often most elegant to define a natural transformation by making a statement of the form 'the arrows A are natural'. This means that the collection of arrows defines the components of a natural transformation, leaving implicit the correct choices of domain and codomain functors, and source and target categories. Here A should be a collection of morphisms in a clearly identifiable (target) category, whose domains and codomains are defined using a common 'variable' (an object of the source category). If this variable is X, one might say 'the arrows A are natural in X' to emphasize the domain object whose component is being described.

Note also that if we have a family of isomorphisms $\theta_{A,B} : F(A,B) \to G(A,B)$ in $\mathcal{E}$ for all $(A,B) \in \mathcal{C} \times \mathcal{D}$, then saying that $\theta_{A,B}$ is natural in (A,B) together is the same as saying it is natural in A (for arbitrary fixed B) and separately natural in B (for arbitrary fixed A). To see this, check the naturality squares.

When every η_X map of a natural transformation is an isomorphism, we call $\eta : F \Rightarrow G$ a **natural isomorphism**. In this case it is easy to see that we also get a natural transformation $\eta^{-1} : G \Rightarrow F$. Explicitly writing out the naturality squares for η and η^{-1} will be essentially the same, except the direction of the η maps will be reversed.

There are two ways to compose natural transformations.

Definition A.2.4. *Given functors $F, G, H : \mathcal{C} \to \mathcal{D}$ and natural transformations $\eta : F \Rightarrow G$, $\epsilon : G \Rightarrow H$, the **vertical composition** $\epsilon \cdot \eta : F \Rightarrow H$ is a natural transformation defined by the morphisms $(\epsilon \cdot \eta)_X := \epsilon_X \circ \eta_X$*

Note then that given a natural isomorphism $\eta : F \Rightarrow G$, we indeed have $\eta \cdot \eta^{-1} = 1_F$ and $\eta^{-1} \cdot \eta = 1_G$, where $1_F : F \Rightarrow F$, $1_G : G \Rightarrow G$ are identity natural transformations.

The second way to compose natural transformations is horizontal composition.

Definition A.2.5. *Given a pair of natural transformations*

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=60, "F", ""{name=F, below}] \arrow[r, bend right=60, "G"', ""{name=G, above}] & \mathcal{D} \arrow[r, bend left=60, "H", ""{name=H, below}] \arrow[r, bend right=60, "K"', ""{name=K, above}] & \mathcal{E}
\arrow[Rightarrow, from=F, to=G, "\epsilon"]
\arrow[Rightarrow, from=H, to=K, "\eta"]
\end{tikzcd}$$

*the **horizontal composition** is a natural transformation $\epsilon * \eta : HF \Rightarrow KG$ whose component at $X \in \mathrm{Ob}(\mathcal{C})$ is the composite of the following commutative square*

$$\begin{tikzcd}
HFX \arrow[r, "\eta_{FX}"] \arrow[d, "H\epsilon_X"'] \arrow[dr, dashed, "(\eta{*}\epsilon)_X"] & KFX \arrow[d, "K\epsilon_X"] \\
HGX \arrow[r, "\eta_{GX}"'] & KGX
\end{tikzcd}$$

It is worth noting that both vertical and horizontal composition preserves isomorphism. That is, if ϵ, η and natural isomorphisms that can be vertically composed, then evidently the vertical composite $\epsilon \cdot \eta$ is also a natural isomorphism. Additionally, if natural isomorphisms ϵ, η can be horizontally composed, then the horizontal composite $\epsilon * \eta$ is also a natural isomorphism (this can be seen from the fact that functors preserve isomorphisms).

As a special case of horizontal composition we have **whiskering** - the composition of a natural transformation with a functor, or more precisely, horizontal composition in the case when either $F = G$ or $H = K$. In its the most general form, consider the functors F, H, K, L and natural transformation ϵ:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "F"] & \mathcal{D} \arrow[r, bend left=50, "H", ""{name=U, below}] \arrow[r, bend right=50, "K"', ""{name=D, above}] & \mathcal{E} \arrow[r, "L"] & \mathcal{F}
\arrow[Rightarrow, from=U, to=D, "\epsilon"]
\end{tikzcd}$$

Then $L\epsilon F : LHF \Rightarrow LKF$, with components $(L\epsilon F)_X := L\epsilon_{FX}$ is a natural transformation. Concretely, the natural transformation $L\epsilon F$ is the horizontal composite $1_L * \epsilon * 1_F$.

A.3 Adjunction

There are perhaps three basic (equivalent) definitions on adjoint functor. The one we will use is the definition in terms of natural isomorphisms.

Definition A.3.1. *If we have functors $F : \mathcal{C} \to \mathcal{D}$, $G : \mathcal{D} \to \mathcal{C}$ and a family of isomorphisms*

$$\mathrm{Hom}_{\mathcal{C}}(A, G(B)) \cong \mathrm{Hom}_{\mathcal{D}}(F(A), B)$$

*natural in A and B, we write $F \dashv G$ and say F, G form an **adjunction**. F is called the **left adjoint** and G is called the **right adjoint**.*

That is, there is a natural isomorphism $\theta : \mathrm{Hom}_{\mathcal{C}}(-, G(-)) \Rightarrow \mathrm{Hom}_{\mathcal{D}}(F(-), -)$, where the functors go from $\mathcal{C}^{op} \times \mathcal{D} \to \mathbf{Set}$. Equivalently, for all $A \in \mathrm{Ob}(\mathcal{C})$, $B \in \mathrm{Ob}(\mathcal{D})$ there is an isomorphism of sets

$$\theta_{A,B} : \mathrm{Hom}_{\mathcal{C}}(A, G(B)) \to \mathrm{Hom}_{\mathcal{D}}(F(A), B)$$

such that for all $g : A' \to A$ in $\mathcal{C}$, and $h : B \to B'$ in $\mathcal{D}$, the following diagram commutes

$$\begin{tikzcd}
\mathrm{Hom}_{\mathcal{C}}(A, G(B)) \arrow[r, "\theta_{A,B}"] \arrow[d, "\mathrm{Hom}_{\mathcal{C}}(g,Gh)"'] & \mathrm{Hom}_{\mathcal{D}}(F(A), B) \arrow[d, "\mathrm{Hom}_{\mathcal{D}}(Fg,h)"] \\
\mathrm{Hom}_{\mathcal{C}}(A', G(B')) \arrow[r, "\theta_{A',B'}"'] & \mathrm{Hom}_{\mathcal{D}}(F(A'), B')
\end{tikzcd}$$

Starting from the top left and taking any $f : A \to G(B)$ in $\mathcal{C}$, we will have

$$h \circ \theta_{A,B}(f) \circ Fg = \theta_{A',B'}(Gh \circ f \circ g).$$

The morphisms $f : A \to G(B)$ in $\mathcal{C}$ and $g : F(A) \to B$ in $\mathcal{D}$ which correspond to each other under $\theta_{A,B}$ are said to be **adjuncts**. That is, f is the right-adjunct of g, and g the left-adjunct of f[4].

The following is a useful technical lemma about the uniqueness of adjoints.

Lemma A.3.2. *If we have an adjunction $F \dashv G$, and there is another functor F that is naturally isomorphic to F (i.e. $F \cong F'$), then we also have $F' \dashv G$.*

This can be seen from the hom-set definition of adjoint. Suppose we have $F : \mathcal{C} \to \mathcal{D}$, and $G : \mathcal{D} \to \mathcal{C}$. We can extend the natural isomorphism $F \cong F'$ to a natural isomorphism

$$\begin{tikzcd}
\mathcal{C}^{op} \times \mathcal{D} \arrow[r, bend left=50, "F \times 1_{\mathcal{D}}", ""{name=U, below}] \arrow[r, bend right=50, "F' \times 1_{\mathcal{D}}"', ""{name=L, above}] & \mathcal{D}^{op} \times \mathcal{D}
\arrow[Rightarrow, from=U, to=L]
\end{tikzcd}$$

and then whisker with $\mathrm{Hom}_{\mathcal{D}}(-,-)$ to get a natural isomorphism

$$\begin{tikzcd}
\mathcal{C}^{op} \times \mathcal{D} \arrow[r, bend left=50, "F \times 1_{\mathcal{D}}", ""{name=U, below}] \arrow[r, bend right=50, "F' \times 1_{\mathcal{D}}"', ""{name=L, above}] & \mathcal{D}^{op} \times \mathcal{D} \arrow[r, "\mathrm{Hom}_{\mathcal{D}}(-,-)"'] & \mathbf{Set}
\arrow[Rightarrow, from=U, to=L]
\end{tikzcd}$$

Then $F \dashv G$ gives a natural isomorphism $\mathrm{Hom}_{\mathcal{C}}(-, G(-)) \Rightarrow \mathrm{Hom}_{\mathcal{D}}(F(-), -)$, with which we can vertically compose, to get the desired natural isomorphism $\mathrm{Hom}_{\mathcal{C}}(-, G(-)) \Rightarrow \mathrm{Hom}_{\mathcal{D}}(F'(-), -)$ witnessing the $F' \dashv G$ adjunction:

[4] See `https://ncatlab.org/nlab/show/adjunct`.

$$\begin{tikzcd}
\mathcal{C}^{op} \times \mathcal{D} \arrow[r, bend left=50, "F \times 1_{\mathcal{D}}"] \arrow[r, bend right=50, "F' \times 1_{\mathcal{D}}"'] \arrow[r, phantom, "\Downarrow"] \arrow[rr, bend left=80, "\mathrm{Hom}_{\mathcal{C}}(-, G(-))"] \arrow[rr, phantom, bend left=45, "\Downarrow"] & \mathcal{D}^{op} \times \mathcal{D} \arrow[r, "\mathrm{Hom}_{\mathcal{D}}(-,-)"'] & \mathbf{Set}
\end{tikzcd}$$

Another useful technical result is the following, which can be found in Mac Lane.

Theorem A.3.3 (Adjunctions with a parameter, theorem 3 p102 [ML13]). *Let $F : \mathcal{C} \times \mathcal{D} \to \mathcal{E}$ be a bifunctor, and for every object D in $\mathcal{D}$, let $F_D : \mathcal{C} \to \mathcal{E}$ denote the functor $F(-, D)$. If every $F_D : \mathcal{C} \to \mathcal{D}$ is left adjoint to a $G_D : \mathcal{E} \to \mathcal{C}$ via*

$$\phi^{D}_{C,E} : \mathrm{Hom}_{\mathcal{C}}(C, G_D(E)) \cong \mathrm{Hom}_{\mathcal{E}}(F_D(C), E)$$

(which for any D is natural in C, E), then there is a unique bifunctor $G : \mathcal{D}^{op} \times \mathcal{E} \to \mathcal{C}$ such that $G_D = G(D, -)$ and $\phi^{D}_{C,E} : \mathrm{Hom}_{\mathcal{C}}(C, G(D, E)) \cong \mathrm{Hom}_{\mathcal{E}}(F(C, D), E)$ is also natural in D.

Bibliography

[Ajd78] Kazimierz Ajdukiewicz. Syntactic connexion (1936). In *The Scientific World-Perspective and Other Essays, 1931–1963*, pages 118–139. Springer, 1978.

[BH53] Yehoshua Bar-Hillel. A quasi-arithmetical notation for syntactic description. *Language*, 29(1):47–58, 1953.

[BHCS60] Yehoshua Bar-Hillel, Gaifman C, and Eli Shamir. On categorial and phrase-structure grammars. *Bulletin of the Research Council of Israel*, F(9):155–166, 1960.

[BKPZ82] Joan Bresnan, Ronald M Kaplan, Stanley Peters, and Annie Zaenen. Cross-serial dependencies in dutch. In *The formal complexity of natural language*, pages 286–319. Springer, 1982.

[Bus01] Wojciech Buszkowski. Lambek grammars based on pregroups. In *International Conference on Logical Aspects of Computational Linguistics*, pages 95–109. Springer, 2001.

[Bus18] Wojciech Buszkowski. Categorial grammars and their logics. *The Lvov-Warsaw School. Past and Present*, pages 91–115, 2018.

[CC07] Stephen Clark and James R Curran. Wide-coverage efficient statistical parsing with ccg and log-linear models. *Computational Linguistics*, 33(4):493–552, 2007.

[CdFMT20] Bob Coecke, Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. Foundations for near-term quantum natural language processing. *arXiv preprint arXiv:2012.03755*, 2020.

[CFC$^+$58] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory logic*, volume 1. North-Holland Amsterdam, 1958.

[CGS13] Bob Coecke, Edward Grefenstette, and Mehrnoosh Sadrzadeh. Lambek vs. lambek: Functorial vector space semantics and string diagrams for lambek calculus. *Annals of pure and applied logic*, 164(11):1079–1100, 2013.

[Cho09] Noam Chomsky. *Syntactic structures*. De Gruyter Mouton, 2009.

[CLM18] Bob Coecke, Martha Lewis, and Dan Marsden. Internal wiring of cartesian verbs and prepositions. *arXiv preprint arXiv:1811.05770*, 2018.

[Coe21] Bob Coecke. The mathematics of text structure. In *Joachim Lambek: The Interplay of Mathematics, Logic, and Linguistics*, pages 181–217. Springer, 2021.

[CSC10] Bob Coecke, Mehrnoosh Sadrzadeh, and Stephen Clark. Mathematical foundations for a compositional distributional model of meaning. *Linguistic Analysis*, 36:345–384, 2010.

[Cur04] James Richard Curran. *From distributional to semantic similarity*. PhD thesis, University of Edinburgh, 2004.

[CW] Bob Coecke and Vincent Wang. Distilling grammar into circuits (or, dehumanising grammar for efficient machine use). forthcoming.

[Fir57] John R Firth. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*, 1957.

[FS19] Brendan Fong and David I. Spivak. *An invitation to applied category theory : seven sketches in compositionality*. Cambridge, 2019.

[Gaz96] Gerald Gazdar. Paradigm merger in natural language processing. 1996.

[GBM20] Bart Geurts, David I. Beaver, and Emar Maier. Discourse Representation Theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2020 edition, 2020.

[Gre13] Edward Grefenstette. *Category-Theoretic Quantitative Compositional Distributional Models of Natural Language Semantics*. PhD thesis, University of Oxford, 2013.

[Hoc03] Julia Hockenmaier. *Data and Models for Statistical Parsing with Combinatory Categorial Grammar*. PhD thesis, University of Edinburgh, 2003.

[HV19] Chris Heunen and Jamie Vicary. *Categories for quantum theory: an introduction*. Oxford University Press, 2019.

[JM] Dan Jurafsky and James H. Martin. *Speech & language processing*. Third edition. forthcoming.

[JZ21] Theo M. V. Janssen and Thomas Ede Zimmermann. Montague Semantics. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2021 edition, 2021.

[Kar15] Dimitrios Kartsaklis. *Compositional Distributional Semantics with Compact Closed Categories and Frobenius Algebras*. PhD thesis, University of Oxford, 2015.

[KKS15] Marco Kuhlmann, Alexander Koller, and Giorgio Satta. Lexicalization and generative power in ccg. *Computational Linguistics*, 41(2):215–247, 2015.

[Kro05] Paul R Kroeger. *Analyzing grammar: An introduction*. Cambridge University Press, 2005.

[KS14] Dimitri Kartsaklis and Mehrnoosh Sadrzadeh. A study of entanglement in a categorical framework of natural language. *arXiv preprint arXiv:1405.2874*, 2014.

[KSPC13] Dimitri Kartsaklis, Mehrnoosh Sadrzadeh, Stephen Pulman, and Bob Coecke. Reasoning about meaning in natural language with compact closed categories and frobenius algebras. *Logic and algebraic structures in quantum computing*, page 199, 2013.

[Lam58] Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.

[Lam88] Joachim Lambek. Categorial and categorical grammars. In *Categorial grammars and natural language structures*, pages 297–317. Springer, 1988.

[Lam97] Joachim Lambek. Type grammar revisited. In *International conference on logical aspects of computational linguistics*, pages 1–27. Springer, 1997.

[Lam08] Joachim Lambek. *From word to sentence : a computational algebraic approach to grammar*. Polimetrica, 2008.

[LPM$^+$21] Robin Lorenz, Anna Pearson, Konstantinos Meichanetzidis, Dimitri Kartsaklis, and Bob Coecke. Qnlp in practice: Running compositional models of meaning on a quantum computer. *arXiv preprint arXiv:2102.12846*, 2021.

[ML13] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[Moo14] Michael Moortgat. Typelogical Grammar. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2014 edition, 2014.

[MTdFC20] Konstantinos Meichanetzidis, Alexis Toumi, Giovanni de Felice, and Bob Coecke. Grammar-aware question-answering on quantum computers. *arXiv preprint arXiv:2012.03756*, 2020.

[NDMG$^+$16] Joakim Nivre, Marie-Catherine De Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, et al. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 1659–1666, 2016.

[NG18] Gerald Nelson and Sidney Greenbaum. *An introduction to English grammar*. Routledge, 2018.

[PL07] Anne Preller and Joachim Lambek. Free compact 2-categories. *Mathematical Structures in Computer Science*, 17(2):309–340, 2007.

[Rie17] Emily Riehl. *Category theory in context*. Courier Dover Publications, 2017.

[Rom20] Mario Román. Open diagrams via coend calculus. *arXiv preprint arXiv:2004.04526*, 2020.

[SCC13] Mehrnoosh Sadrzadeh, Stephen Clark, and Bob Coecke. The frobenius anatomy of word meanings i: subject and object relative pronouns. *Journal of Logic and Computation*, 23(6):1293–1317, 2013.

[SCC14] Mehrnoosh Sadrzadeh, Stephen Clark, and Bob Coecke. The frobenius anatomy of word meanings ii: possessive relative pronouns. *Journal of Logic and Computation*, 26(2):785–815, 2014.

[Sch98] Hinrich Schütze. Automatic word sense discrimination. *Computational linguistics*, 24(1):97–123, 1998.

[Shi85] Stuart M Shieber. Evidence against the context-freeness of natural language. In *Philosophy, language, and artificial intelligence*, pages 79–89. Springer, 1985.

[SMR08] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[Ste00] Mark Steedman. *The syntactic process*, volume 24. MIT press Cambridge, MA, 2000.

[Tal19] Maggie Tallerman. *Understanding syntax*. Routledge, 2019.

[Tra14] Robert Lawrence Trask. *Introducing linguistics: a graphic guide*. Icon Books Ltd, 2014.

[VL20] Giulia Vilone and Luca Longo. Explainable artificial intelligence: a systematic review. *arXiv preprint arXiv:2006.00093*, 2020.

[VSW94] Krishnamurti Vijay-Shanker and David J Weir. The equivalence of four extensions of context-free grammars. *Mathematical systems theory*, 27(6):511–546, 1994.

[YK21] Richie Yeung and Dimitri Kartsaklis. A ccg-based version of the discocat framework. *arXiv preprint arXiv:2105.07720*, 2021.

[YNM17] Masashi Yoshikawa, Hiroshi Noji, and Yuji Matsumoto. A* CCG parsing with a supertag and dependency factored model. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 277–287, Vancouver, Canada, July 2017. Association for Computational Linguistics.