Categorical Tools for Natural Language Processing

Giovanni de Felice

Wolfson College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2022

Abstract

This thesis develops the translation between category theory and computational linguistics as a foundation for natural language processing. The three chapters deal with syntax, semantics and pragmatics. First, string diagrams provide a unified model of syntactic structures in formal grammars. Second, functors compute semantics by turning diagrams into logical, tensor, neural or quantum computation. Third, the resulting functorial models can be composed to form games where equilibria are the solutions of language processing tasks. This framework is implemented as part of DisCoPy, the Python library for computing with string diagrams. We describe the correspondence between categorical, linguistic and computational structures, and demonstrate their applications in compositional natural language processing.

Acknowledgements

I would like to thank my supervisor Bob Coecke for introducing me to the wonderland of string diagrams, for supporting me throughout my studies and always encouraging me to switch topics and pursue my ideas. This thesis is the fruit of a thousand discussions, board sessions, smokes and beers with Alexis Toumi. I am grateful to him for having the patience to teach Python to a mathematician, for his loyal friendship and the continual support he has given me in both personal matters and research. I want to thank my examiners Aleks Kissinger and Pawel Sobocinski for their detailed feedback on the first version of this thesis, and their suggestion to integrate the passage from categories to Python code. Thanks also to Andreas Petrossantis, Sebastiano Cultrera and Dimitri Kartsaklis for valuable comments on this manuscript, to Dan Marsden and Prakash Panangaden for providing guidance in my early research, and to Samson Abramsky for prompting me to search into the deeper history of applied category theory.

Among fellow collaborators who have shared their wisdom and passion, for many insightful discussions, I would like to thank Amar Hadzihasanovic, Rui Soares Barbosa, David Reutter, Antonin Delpeuch, Stefano Gogioso, Konstantinos Meichanetzidis, Mario Roman, Elena Di Lavore and Richie Yeung. Among my friends, who have been there for me in times of sadness and of joy, and made me feel at home in Oxford, Rome and Sicily, special thanks to Tommaso Salvatori, Pinar Kolancali, Tommaso Battistini, Emanuele Torlonia, Benedetta Magnano and Pietro Scammacca. Finally, a very special thanks to Nonna Miti for hosting me in her garden and to my mother and father for their loving support.

Contents

Introduction

Since the very beginnings of human inquiry into language, people have investigated the natural processes by which we learn, understand and produce linguistic meaning. Only recently, however, the field of linguistics has become an autonomous scientific discipline. The origins of this modern science are closely interlinked with the birth of mathematical logic at the end of the nineteenth century. In the United States, Peirce founded "semiotics" — the science of signs and their interpretation — while developing graphical calculi for logical inference. At around the same time, in the United Kingdom, Frege and Russell developed formal languages for logic in the search for a Leibnizian "characteristica universalis" while discussing the sense and reference of linguistic phrases.

These mathematical origins initiated a formal and computational approach to linguistics, often referred to as the symbolic tradition, which aims at characterising language understanding in terms of structured logical processes and the automatic manipulation of symbols. On the one hand, it led to the development of mathematical theories of syntax, such as the categorial grammars stemming from the Polish school of logic [Ajd35; Lam58] and Chomsky's influential generative grammars [Cho57]. On the other hand, it allowed for the development of formal approaches to semantics such as Tarski's theory of truth [Tar36; Tar43], which motivated the work of Davidson [Dav67a] and Montague [Mon70b] in extracting the logical form of natural language sentences. From the technological perspective, these theories enabled the design of programming languages, the construction of large-scale databases for storing structured knowledge and linguistic data, as well as the implementation of expert computer systems driven by formal logical rules to reason about this accrued knowledge. Since the 1990s, the symbolic tradition has been challenged by a series of new advances motivated by the importance of context and ambiguity in language use [BP83]. With the growing amount of data and large-scale corpora available on the internet, statistical inference methods based on n-grams, Markov models or Bayesian classifiers allowed for experiments to tackle new problems such as speech recognition and machine translation [MS99]. The distributional representation of meaning in vector spaces [SW97] was found suitable for disambiguating words in context [Sch98] and computing synonymity [TP10]. Furthermore, connectionist models based on neural networks have produced impressive results in the last couple of decades, outperforming previous models on a range of tasks such as language modelling [Ben+03; Mik+10; Vas+17], word sense disambiguation [Nav09], sentiment analysis [Soc+13a; Cha+22], question answering [JM08; LHH20] and machine translation [BCB14; Edu+18].

Driven by large-scale industrial applications, the focus gradually shifted from theoretical enquiries into linguistic phenomena to the practical concern of building highly parallelizable connectionist code for beating state-of-the-art algorithms. Recently, a transformer neural network with billions of parameters (GPT-3) [Bro+20] wrote a Guardian article on why humans have nothing to fear from AI. The reasons for how and why GPT-3 "chose" to compose the text in the way that it did is a mystery and the structure of its mechanism remains a "black box". Connectionist models have shown the importance of the distributional aspect of language and the effectiveness of machine learning techniques in NLP. However, their task-specificity and the difficulty in analysing the underlying processes which concur in their output are limits which need to be addressed. Recent developments in machine learning have shown the importance of taking structure into account when tackling scientific questions in network science [Wu+20], chemistry [Kea+16], biology [Jum+21; Zho+21]. NLP would also benefit from the same grounding in order to analyse and interpret the growing "library of Babel" of natural language data.

Category theory can help build models of language amenable to both linguistic reasoning and numerical computation. Its roots are the same as computational linguistics, as categories were used to link algebra, logic and computation [Law63; Lam86; LS86]. Category theory has since followed a solid thread of applications, from the semantics of programming languages [SRP91; AJ95] to the modelling of a wide range of computational systems, including knowledge-based [Spi12], quantum [AC07; CK17], dataflow [BSZ14], statistical [VKS19] and differentiable [AD19] processes. In the Compositional Distributional models of Coecke et al. [CCS08; CCS10; SCC13] (DisCoCat), categories are used to design models of language in which the meaning of sentences is derived by composition from the distributional embeddings of words. Generalising from this work, language can be viewed as a syntax for arranging symbols together with a functor for interpreting them. Specifically, syntactic structures form a free category of string diagrams, while meaning is computed in categories of numerical functions. Functorial models can then be learnt in data-driven tasks.

The aim of this thesis is to provide a unified framework of mathematical tools to be applied in three important areas of computational linguistics: syntax, semantics and pragmatics. We provide an implementation of this framework in object-oriented Python, by translating categorical concepts into classes and methods. This translation has lead to the development of DisCoPy [dTC21], an open-source Python toolbox for computing with string diagrams and functors. We show the potential of this framework for reasoning about compositional models of language and building structured NLP pipelines. We show the correspondence between categorical and linguistic notions and we describe their implementation as methods and interfaces in DisCoPy. The library is available, with an extensive documentation and testing suite, at:

`https://github.com/oxford-quantum-group/discopy`

In Chapter 1, on syntax, we use the theory of free categories and string diagrams to formalise Chomsky's regular, context-free and unrestricted grammars [Cho56]. With the same tools, the categorial grammars of Ajdiuciewicz [Ajd35], Lambek [Lam58]

and Steedman [Ste00], as well as Lambek's pregroups [Lam99a] and Tesniere's dependency grammars [Tes59; Gai65], are formalised. We lay out the architecture of the syntactic modules of DisCoPy, with interfaces for the corresponding formal models of grammar and functorial reductions between them. The second chapter deals with semantics. We use Lawvere's concept of functorial semantics [Law63] to define several NLP models, including logical, distributional and connectionist approaches. By varying the target semantic category, we recover knowledge-based, tensor-based and quantum models of language, as well as Montague's logical semantics [Mon70b] and connectionist models based on neural networks. The implementation of these models in Python is obtained by defining semantic classes that carry out concrete computation. We describe the implementation of the main semantic modules of DisCoPy and their interface with high-performance libraries for numerical computation. This framework is then applied to the study of the pragmatic aspects of language use in context and the design of NLP tasks in Chapter 3. To this end, we use the recent applications of category theory to statistics [CJ19; Fri20], machine learning [FST19; Cru+21] and game theory [Gha+18] to develop a formal language for modelling compositions of NLP models into games and pragmatic tasks.

The mathematical framework developed in this thesis provides a structural understanding of natural language processing, allowing for both interpreting existing NLP models and building new ones. The proposed software is expected to contribute to the design of language processing systems and their implementation using symbolic, statistical, connectionist and quantum computing.

Contributions

The aim of this thesis is to provide a framework of mathematical tools for computational linguistics. The three chapters are related to syntax, semantics and pragmatics, respectively.

In Chapter 1, a unified theory of formal grammars is provided in terms of free categories and functorial reductions and implemented in object-oriented Python. This work started from discussions with Alexis Toumi, Bob Coecke, Mernoosh Sadrzadeh, Dan Marsden and Konstantinos Meichanetzidis about logical and distributional models of natural language [Coe+18; dMT20; Coe+22]. It was a driving force in the development of DisCoPy [dTC21]. The main contributions are as follows.

1. A unified treatment of *formal grammars* is provided using the theory of string diagrams in free monoidal categories. We cover Chomsky's regular 1.1, context-free 1.2 and unrestricted grammar 1.3, corresponding to free categories, free operads and free monoidal categories respectively. This picture is obtained by aggregating the results of Walters and Lambek [Wal89b; Wal89a; Lam88; STS20]. Using the same tools, we formalise categorial grammars 1.4, as well as pregroups and dependency grammars 1.5 in terms of biclosed and rigid categories. The links between pregroups and rigid categories are known since [PL07], those between categorial grammar and biclosed categories were previously discussed by Lambek [Lam88] but not fully worked out, while the categorical formalisation of dependency grammar is novel. We also introduce the notion of pregroup with coreference for discourse representation in 1.6 [Coe+18] to offer an alternative to the DisCoCirc framework of Coecke [Coe20] which can be implemented with readily available tools. To the best of our knowledge, this is the first time that models from the Chomskyan and categorial traditions appear in the same framework, and that the full correspondence between linguistic and categorical notions is spelled out.

2. *Functorial reductions* between formal grammars are introduced as a convenient intermediate notion between weak and strong equivalences, and used to compute normal forms of context-free grammars in 1.3.3. We use this notion in the remainder of the chapter to capture the relationship between: i) CFGs and categorial grammar (Propositions 1.4.8 and), ii) categorial grammar and biclosed categories (Propositions 1.4.7, 1.4.12 and 1.4.17), iii) categorial and pregroup grammars (Propositions 1.5.7 and 1.5.13) and iv) pregroups, dependency grammars and CFGs in 1.5.2. The latter yields a novel result showing that dependency grammars are the structural intersection of pregroups and CFGs (Theorem 1.5.23).

3. The previously introduced categorical definitions are implemented in *object-oriented Python*. The structure of this chapter follows the architecture of the syntactic modules of DisCoPy, as described at the end of this section. Free categories and syntactic structures are implemented by subclassing `cat.Arrow`

or `monoidal.Diagram`, the core data structures of DisCoPy. Functorial reductions are implemented by calling the corresponding `Functor` class. We interface DisCoPy with linguistic tools for large-scale parsing.

In Chapter 2, functorial semantics is applied to the study of natural language processing models. Once casted in this algebraic framework, it becomes possible to prove complexity results and compare different NLP pipelines, while also implementing these models in DisCoPy. Section 2.4 is based on joint work with Alexis Toumi and Konstantinos Meichanetzidis on relational semantics [dMT20]. Sections 2.5, 2.8 and 2.7 are based on the recent quantum models for NLP introduced with Bob Coecke, Alexis Toumi and Konstantinos Meichanetzidis [Mei+21; Mei+20] and further developed in [Kar+21; TdY22]. We list the main contributions of this chapter.

1. *NLP models* are given a unified theory, formalised as functors from free syntactic categories to concrete categories of numerical structures. These include i) knowledge-based relational models 2.4 casted as functors into the category of relations, ii) tensor network models 2.5 seen as functors into the category of matrices and including factorisation models for knowledge-graph embedding covered in 2.6 iii) quantum NLP models 2.7 casted as functors into the category of quantum circuits, iv) Montague semantics 2.2 given by functors into cartesian closed categories, and v) recurrent and recursive neural network models 2.3 which appear as functors from grammar to the category of functions on euclidean spaces.

2. We prove *complexity results* on the evaluation of these functorial models and related NLP tasks. Expanding on [dMT20], we use relational models to define `NP`-complete entailment and question-answering problems (Propositions 2.4.26, 2.4.31 and 2.4.33). We show that the evaluation of tensor network models is in general #`P`-complete (Proposition 2.5.18) but that it becomes tractable when the input structures come from a dependency grammar (Proposition 2.5.20). We show that the additive approximation of quantum NLP models is a `BQP`-complete problem (Proposition 2.7.18). We also prove results showing that Montague semantics is intractable in its general form (Propositions 2.2.14, 2.2.16).

3. *Montague semantics* is given a detailed formalisation in terms of free cartesian closed categories. This corrects a common misconception in the DisCoCat literature, and in particular in [SCC13; SCC14], where Montague semantics is seen as a functor from pregroup grammars to relations. These logical models are studied in 2.4, but they are distinct from Montague semantics where the lambda calculus and higher-order types play an important role 2.2.

4. We show how to implement *functorial semantics in DisCoPy*. More precisely, the implementation of the categories of tensors and Python functions is described in 2.1. We then give a concrete example of how to solve a knowledge-embedding task in DisCoPy by learning functors 2.8. We define currying and

uncurrying of Python functions 2.2 and use it to give a proof-of-concept implementation of Montague semantics. We define sequential and parallel composition of Tensorflow/Keras models [Cho+15], allowing us to construct recursive neural networks with a DisCoPy functor 2.3.

In Chapter 3, we develop formal diagrammatic tools to model pragmatic scenarios and natural language processing tasks. This Chapter is based on our work with Mario Roman, Elena Di Lavore and Alexis Toumi [de +21], and provides a basis for the formalisation of monoidal streams in the stochastic setting [DdR22]. The contribution for this section is still at a preliminary stage, but the diagrammatic notation succeeds in capturing and generalising a range of approaches found in the literature as follows.

1. *Categorical probability* [Fri20] is applied to the study of discriminative and generative language models using notions from Bayesian statistics in 3.1. We investigate the category of *lenses* [Ril18a] over discrete probability distributions in 3.2 and use it to characterise notions of context, utility and reward for iterated stochastic processes [DdR22] (see Propositions 3.2.4 and 3.2.6). Finally, we apply open games [BHZ19] the study of Markov decision processes and repeated games in 3.3, while giving examples relevant to NLP.

2. Three *NLP applications* of the developed tools are provided in 3.4: i) we discuss reference games [FG12; MP15; BLG16] and give a diagrammatic proof that Bayesian inversion yields a Nash equilibrium (Proposition 3.4.1), ii) we define a question answering game between a teacher and a student and compute the Nash equilibria when the student's strategies are given by relational models [de +21] and iii) we give a compositional approach to word-sense disambiguation as a game between words where strategies are word-senses.

The DisCoPy implementation, carried out with Alexis Toumi [dTC21], is described throughout the first two chapters of this thesis. We focused on i) the passage from categorical definitions to object-oriented Python and ii) the applications of DisCoPy to Natural Language Processing. Every section of Chapter 1 corresponds to a *syntactic module* in DisCoPy, as we detail.

1. In 1.1, we view derivations of *regular* grammars as arrows in the free category and show how these notions are implemented via the core DisCoPy class `cat.Arrow`.

2. In 1.2, we study *context-free* grammars in terms of trees in the free operad and give an implementation of free operads and their algebras in DisCoPy. This is a new `operad` module which has been written for this thesis and features interfaces with NLTK [LB02] for CFG and SpaCy [Hon+20] for dependencies.

3. In 1.3, we formalise Chomsky's *unrestricted* grammars in terms of monoidal signatures and string diagrams and describe the implementation of the key DisCoPy class `monoidal.Diagram`.

4. In 1.4, we formalise *categorial* grammar in terms of free biclosed categories. We give an implementation of `biclosed.Diagram` as a monoidal diagram with `curry` and `uncurry` methods and show its interface with state-of-the-art categorial grammar parsers, as provided by Lambeq [Kar+21].

5. In 1.5, we show that *pregroup* and *dependency* structures are diagrams in free rigid categories. We describe the data structure `rigid.Diagram` and its interface with SpaCy [Hon+20] for dependency parsing.

6. In 1.6, we introduce a notion of pregroup grammar with *coreference* using hypergraphs to represent the syntax of text and discourse. We describe the `hypergraph.Diagram` data structure and show how to interface it with SpaCy's package for neural coreference.

The models studied in Chapter 2 can all be implemented using one of the four *semantic modules* of DisCoPy which we detail.

1. The `tensor` module of DisCoPy implements the category of matrices, as described in 2.1 where we give its implementation in NumPy [Har+20]. We use it in conjunction with Jax [Bra+18] in 2.8 to implement the models introduced in 2.4 and 2.5.

2. The `quantum` module of DisCoPy implements quantum circuits 2.7 and supports interfaces with PyZX [Kv19] and TKET [Siv+20] for optimisation and compilation on quantum hardware. These features are described in our recent work [TdY22].

3. The `function` module of DisCoPy implements the category of functions on Python types, as described in 2.1. We define currying and uncurrying of Python functions 2.2 and use it to give a proof-of-concept implementation of Montague semantics.

4. The `neural` module implements the category of neural networks on euclidean spaces. We describe it in 2.3 as an interface between DisCoPy and TensorFlow/Keras [Cho+15].

A schematic view of the modules in DisCoPy and their interfaces is summarized in Figure 1.

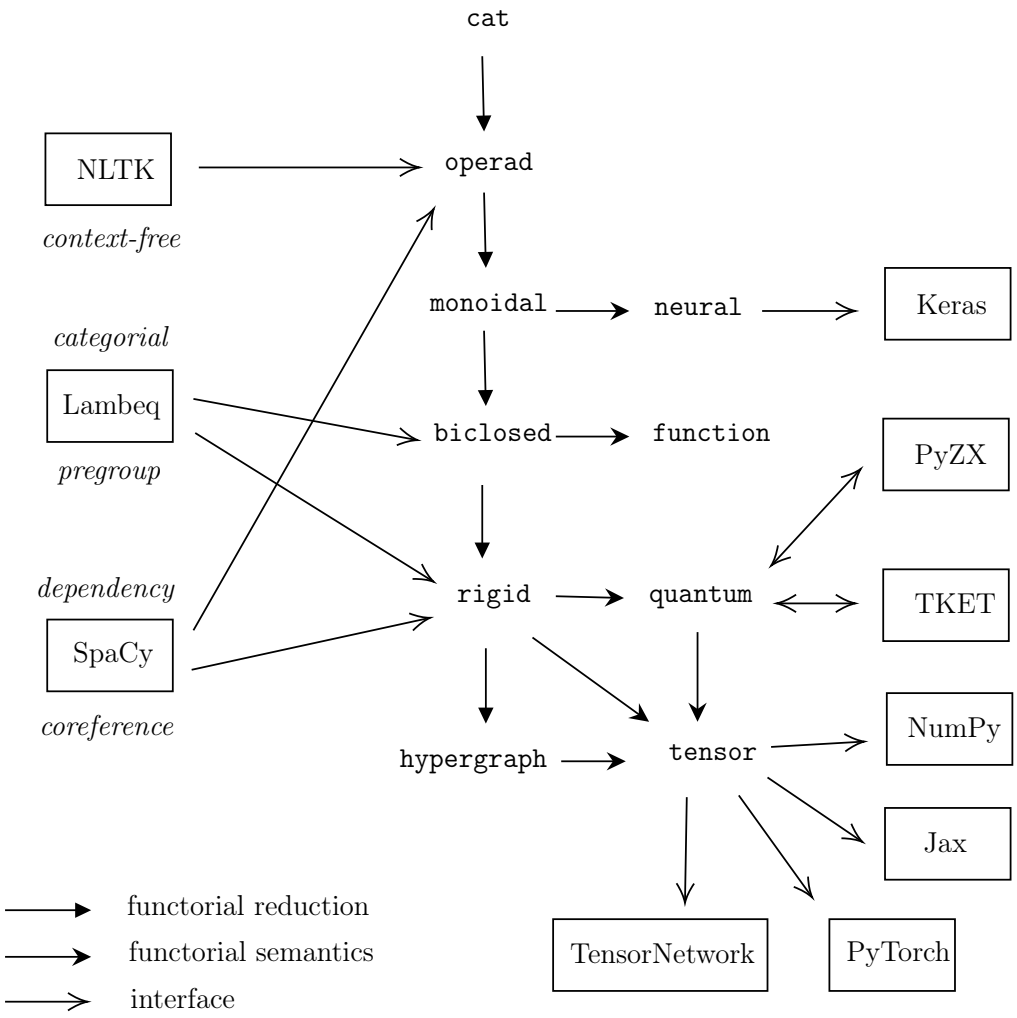


Figure 1: DisCoPy: an interfaced compositional software for NLP

Chapter 1

Diagrams for Syntax

The word "grammar" comes from the Greek $\gamma\rho\acute{\alpha}\mu\mu\alpha$ (gramma), itself from $\gamma\rho\acute{\alpha}\varphi\epsilon\iota\nu$ (graphein) meaning both "to write" and "to draw", and we will represent grammatical structure by drawing diagrams. A *formal grammar* is usually defined by a set of *rewriting rules* on strings. The rewriting process, also called *parsing*, yields a procedure for deciding whether a string of words is grammatical or not.

These structures were studied in mathematics since the 1910s by Thue and later by Post [Pos47] and Markov Jr. [Kus06]. Their investigation was greatly advanced by Chomsky [Cho57], who used them to *generate* grammatical sentences from some basic rewrite rules interpreted as *productions*. He showed that natural restrictions on the allowed production rules form a hierarchy, from unrestricted to regular grammars, which corresponds to models of computation of varying strengths, from Turing machines to deterministic finite state automata [Cho56]. In parallel to Chomsky's seminal work, Lambek developed his syntactic calculus [Lam58], refining and unifying the *categorial grammars* originated in the Polish school of logic [Ajd35]. These are different in spirit from Chomsky's grammars, but they also have tight links with computation as captured by the Lambda calculus [van87; Ste00].

In this chapter, we lay out the correspondence between free categorical structures and linguistic models of grammar. Every level in this hierarchy is implemented with a corresponding class in DisCoPy. In 1.1, we show that regular grammars can be seen as graphs with a labelling homomorphism and their derivations as arrows in the free category, implemented via the core DisCoPy class `cat.Arrow`. In 1.2, we show that context-free grammars are operadic signatures and their derivations trees in the free operad. We give an implementation of free operads as a class `operad.Tree`, interfaced with NLTK [LB02] for context-free parsing. In 1.3, we arrive at Chomsky's unrestricted grammars, captured by monoidal signatures and string diagrams. We discuss varying notions of reduction and normal form for these grammars, and show the implementation of the key DisCoPy class `monoidal.Diagram`. In 1.4, we show that categorial grammars such as the original grammars of Ajdiuciewicz and Bar-Hillel, the Lambek calculus and Combinatory Categorial Grammars (CCGs) can be seen as biclosed signatures and their grammatical reductions as morphisms in free biclosed categories. We give an implementation of `biclosed.Diagram` as a monoidal diagram with `curry` and `uncurry` methods and show its interface with state-of-the-

art categorial grammar parsers, as provided by Lambeq [Kar+21]. In 1.5, we show that pregroups and dependency grammars are both captured by rigid signatures, and their derivations by morphisms in the free rigid category. This leads to the data structure `rigid.Diagram` which we interface with SpaCy [Hon+20] for state-of-the-art dependency parsing. Finally, in 1.6, we introduce a notion of pregroup grammar with coreference using hypergraphs to represent the syntax of text and discourse, and give a proof-of-concept implementation in DisCoPy.

1.1 Arrows

In this section, we introduce three structures: categories, regular grammars and `cat.Arrows`. These cast light on a level-zero correspondence between algebra, linguistics and Python programming. We start by defining categories and their free construction from graphs. Following Walters [Wal89a], regular grammars are defined as graphs together with a labelling homomorphism and their grammatical sentences as labelled paths, i.e. arrows of the corresponding free category. This definition is very similar to the definition of a finite state automaton, and we discuss the equivalence between Walters' notion, Chomsky's original definition and finite automata. We end by introducing the `cat` module, an implementation of free categories and functors which forms the core of DisCoPy.

1.1.1 Categories

Definition 1.1.1 (Simple signature / Directed graph)**.** *A simple signature, or directed graph, G is a collection of vertices G_0 and edges G_1 such that each edge has a domain and a codomain vertex*

$$G_0 \xleftarrow{\texttt{dom}} G_1 \xrightarrow{\texttt{cod}} G_0$$

. A graph homomorphism $\varphi : G \to \Gamma$ is a pair of functions $\varphi_0 : G_0 \to \Gamma_0$ and $\varphi_1 : G_1 \to \Gamma_1$ such that the following diagram commutes:

$$\begin{array}{ccccc} G_0 & \xleftarrow{\texttt{dom}} & G_1 & \xrightarrow{\texttt{cod}} & G_0 \\ \downarrow{\scriptstyle \varphi_0} & & \downarrow{\scriptstyle \varphi_1} & & \downarrow{\scriptstyle \varphi_0} \\ \Gamma_0 & \xleftarrow[\texttt{dom}]{} & \Gamma_1 & \xrightarrow[\texttt{cod}]{} & \Gamma_0 \end{array}$$

We denote by $G(a, b)$ the edges $f \in G_1$ such that $\texttt{dom}(f) = a$ and $\texttt{cod}(f) = b$. We also write $f : a \to b$ to denote an edge $f \in G(a, b)$.

A category is a directed graph with a composition operation, in this context vertices are called *objects* and edges are called *arrows* or *morphisms*.

Definition 1.1.2 (Category)**.** *A category $\mathbf{C}$ is a graph $\mathbf{C}_1 \rightrightarrows \mathbf{C}_0$, where $\mathbf{C}_0$ is a set of* objects*, and $\mathbf{C}_1$ a set of* morphisms*, equipped with a composition operation $\cdot : \mathbf{C}(a, b) \times \mathbf{C}(b, c) \to \mathbf{C}(a, c)$ defined for any $a, b, c \in \mathbf{C}_0$ such that:*

1. *for any $a \in \mathbf{C}_0$ there is an identity morphism $\texttt{id}_a \in \mathbf{C}(a, a)$ (identities).*
2. *for any $f : a \to b$, $f \cdot \texttt{id}_a = f = \texttt{id}_b \cdot f$ (unit law).*
3. *whenever $a \xrightarrow{f} b \xrightarrow{g} c \xrightarrow{h} d$, we have $f \cdot (g \cdot h) = (f \cdot g) \cdot h$ (associativity).*

A functor *$F : \mathbf{C} \to \mathbf{D}$ is a graph homomorphism which respects composition and identities, i.e. for any $a \in \mathbf{C}_0$ $F(\texttt{id}_a) = \texttt{id}_{F(a)}$ and whenever $a \xrightarrow{g} b \xrightarrow{f} c$ in $\mathbf{C}$ we have $F(f \cdot g) = F(f) \cdot F(g)$.*

Given a pair of functors $F, G : \mathbf{C} \to \mathbf{D}$, a natural transformation $\alpha : F \to G$ *is a family of maps $\alpha_a : F(a) \to G(a)$ such that the following diagram commutes:*

$$\begin{array}{ccc} F(a) & \xrightarrow{\alpha_a} & G(a) \\ \downarrow{\scriptstyle F(f)} & & \downarrow{\scriptstyle G(f)} \\ F(b) & \xrightarrow{\alpha_b} & G(b) \end{array} \tag{1.1}$$

for any $f : a \to b$ in $\mathbf{C}$.

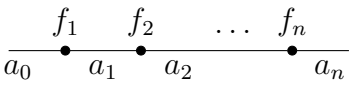
Remark 1.1.3. *The symbol $\to$ appeared remarkably late in the history of symbols with the earliest use registered in Bernard Forest de Belidor's 1737* L'architecture hydraulique*, where it is used to denote the direction of a flow of water. Arguably, it conveys more structured information then its predecessor: the Medieval manicule symbol. Its current mathematical use as the type of a morphism $f : x \to y$ appeared only at the beginning of the 20th century, the first extensive use being registered in Hausdorff [Hau35] to denote group homomorphisms.*

Example 1.1.4 (Basic)**.** *Sets and functions form a category* **Set***. Monoids and monoid homomorphisms for a category* **Mon***. Graphs and graph homomorphisms form a category* **Graph***. Categories and functors form a category* **Cat***.*

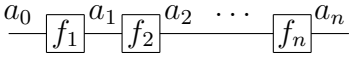
An arrow f in a graph G is a sequence of edges $f \in G_1^*$ such that $\mathtt{cod}(f_i) = \mathtt{dom}(f_{i+1})$, it can be represented graphically as a sequence of arrows:

$$a_0 \xrightarrow{f} a_n = a_0 \xrightarrow{f_1} a_1 \ldots \xrightarrow{f_n} a_n$$

Or as a sequence of vertices and edges:



Or as a sequence of boxes and wires:



Two arrows with matching endpoints can be composed by concatenation.

$$a \xrightarrow{f} b \xrightarrow{g} c = a \xrightarrow{f \cdot g} c$$

Paths on G in fact form a category, denoted $\mathbf{C}(G)$. $\mathbf{C}(G)$ has the property of being the *free category* generated by G [Mac71].

The free category construction is the object part of functor $\mathbf{C} : \mathbf{Graph} \to \mathbf{Cat}$, which associates to any graph homomorphism $\varphi : G \to V$ a functor $\mathbf{C}(\varphi) : \mathbf{C}(G) \to \mathbf{C}(V)$ which relabels the vertices and edges in an arrow. This free construction $\mathbf{C}$ is the *left adjoint* of the functor $U : \mathbf{Cat} \to \mathbf{Graph}$ which forgets the composition operation on arrows. $\mathbf{C}$ is a left adjoint of U in the sense that there is a natural isomorphism:

$$\mathbf{Graph}(G, U(\mathbf{S})) \simeq \mathbf{Cat}(\mathbf{C}(G), \mathbf{S})$$

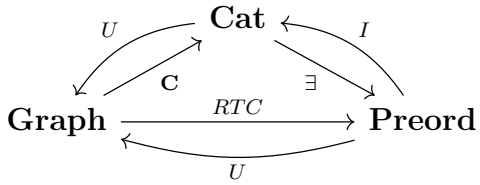
which says that specifying a functor $F : \mathbf{C}(G) \to \mathbf{S}$ is the same as specifying an arrow in $\mathbf{S}$ for every generator in G. This will have important consequences in the context of semantics.

A preorder P is a category with at most one morphism between any two objects. Given $a, b \in P_0$, the hom-set $P(a, b)$ is either the singleton or empty, we write aPb for the corresponding boolean value. Identities and composition of the category, correspond to reflexivity and transitivity of the preorder. Following Lambek [Lam68] and [Str07], we can interpret a preorder as a *logic*, by considering the underlying set as a set of *formulae* and the relation $\leq$ as a *consequence* relation, which is usually denoted $\vdash$ (entails). Reflexivity and transitivity of the consequence relation correspond to the following familiar rules of inference.

$$\frac{}{A \vdash A} \qquad \frac{A \vdash B \qquad B \vdash C}{A \vdash C} \tag{1.2}$$

Given a graph G we can build a preorder by taking the reflexive transitive closure of the relation $G \subseteq G_0 \times G_0$ induced by the graph, $\leq = RTC(G) \subseteq G_0 \times G_0$. We can think of the edges of the graph G as a set of *axioms*, then the free preorder $RTC(G)$ captures the logical consequences of these axioms, and, in this simple setting, we have that $a \implies b$ if and only if there is an arrow from a to b in G.

This construction is analogous to the free category construction on a graph, and is in fact part of a commuting triangle of adjunctions relating topology, algebra and logic.



Where $\exists\mathbf{C}$ is the *preorder collapse* of $\mathbf{C}$, i.e. $a \exists\mathbf{C} b$ if and only if $\exists f \in \mathbf{C}(a, b)$. The functor RTC allows to define the following *decision problem*.

Definition 1.1.5. ∃Path

Input: $G \rightrightarrows B$, $a, b \in N$
Output: $a\ RTC(G)\ b$

The problem ∃Path is also known as the graph accessibility problem, which is complete for NL the complexity class of problems solvable by a non-deterministic Turing machine in logarithmic space [Imm87]. In particular, it is equivalent to 2SAT, the problem of satisfying a formula in conjunctive normal form where each clause has two literals. The reduction from 2SAT to ∃Path is obtained by turning the formula into its graph of implications.

The free construction $\mathbf{C}$ allows to define a *proof-relevant* form of the path problem, where what matters is not only whether a entails b but the way in which a entails b.

Definition 1.1.6. Path

Input: G, $a, b \in G_0$
Output: $f \in \mathbf{C}(G)(a, b)$

From these considerations we deduce that $\texttt{Path} \in \texttt{FNL}$, since it is the *function problem* corresponding to $\exists\texttt{Path}$. These problems correspond to parsing problems for regular grammars.

1.1.2 Regular grammars

We now show how graphs and categories formalise the notion of regular grammar. Fix a finite set of words V, called the *vocabulary*, and let us use V to label the edges in a graph G. The data for such a *labelling* is a function $L : G_1 \to V$ from edges to words, or equivalently a graph homomorphism $L : G \to V$ where V is seen as a graph with one vertex and words as edges. Fix a starting vertex s_0 and a terminal vertex s_1. Given any arrow $f : s_0 \to s_1 \in \mathbf{C}(G)$, we can concatenate the labels for each generator to obtain a string $L^*(f) \in \mathbf{C}(V) = V^*$ where $L^* = \mathbf{C}(L)$ is the function L applied point-wise to arrows. We say that a string $u \in V^*$ is *grammatical* in G whenever there is an arrow $f : s_0 \to s_1$ in G such that $L(f) = u$. We can think of the arrow f as a witness of the grammaticality of u, called a *proof* in logic and a *derivation* in the context of formal grammars.

Definition 1.1.7 (Regular grammar)**.** *A regular grammar is a finite graph G equipped with a homomorphism $L : G \to V$, where V is a set of words called the* vocabulary*, and two specified symbols $s_0, s_1 \in G_0$, the bottom (starting) and top (terminating) symbols. Explicitly it is given by three functions:*

$$\begin{array}{ccccc} G_0 & \xleftarrow{\texttt{dom}} & G_1 & \xrightarrow{\texttt{cod}} & G_0 \\ & & \downarrow{\scriptstyle L} & & \\ & & V & & \end{array}$$

The language generated by G is given by the image of the labelling functor:

$$\mathcal{L}(G) = L^*(\mathbf{C}(G)(s_0, s_1)) \subseteq V^* .$$

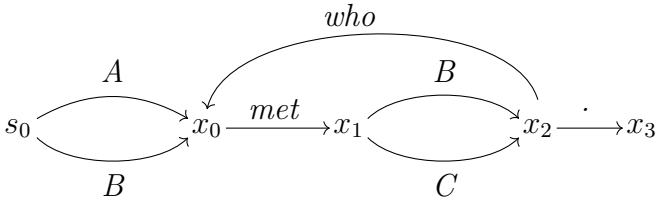
A morphism of regular grammars $\varphi : G \to H$ is a graph homomorphism such that the following triangle commutes:

$$\begin{array}{ccc} G & \xrightarrow{\quad\varphi\quad} & H \\ & {\scriptstyle L}\searrow \quad \swarrow{\scriptstyle L} & \\ & V & \end{array}$$

and such that $\varphi(s_1) = s_1'$, $\varphi(s_0) = s_0'$. These form a category of regular grammars **Reg** *which is the slice or* comma *category of the coslice over the points $\{ s_0, s_1 \}$ of the category of signatures* $\mathbf{Reg} = (2 \backslash \mathbf{Graph})/V$.

Definition 1.1.8 (Regular language)**.** *A regular language is a subset of $X \subseteq V^*$ such that there is a regular grammar G with $\mathcal{L}(G) = X$.*

Example 1.1.9 (SVO)**.** *Consider the regular grammar generated by the following graph:*



An example of sentence in the language $\mathcal{L}(G)$ is "A met B who met C.".

Example 1.1.10 (Commuting diagrams)**.** *Commuting diagrams such as 1.1 can be understood using regular grammars. Indeed, a commuting diagram is a graph G together with a labelling of each edge as a morphism in a category $\mathbf{C}$. Given a pair of vertices $x, y \in G_0$, we get a regular language $\mathcal{L}(G)$ given by arrows from x to y in G. Saying that the diagram G commutes corresponds to the assertion that all strings in $\mathcal{L}(G)$ are equal as morphisms in $\mathbf{C}$.*

We now translate from the original definition by Chomsky to the one above. Recall that a regular grammar is a tuple $G = (N, V, P, s)$ where N is a set of non-terminal symbols with a specified start symbol $s \in N$, V is a vocabulary and P is a set of production rules of the form $A \to aB$ or $A \to a$ or $A \to \epsilon$ where $a \in V$, $A, B \in N$ and ϵ denotes the empty string. We can think of the sentences generated by G as arrows in a free category as follows. Construct the graph $\Sigma = P \rightrightarrows (N + \{ s_1 \})$ where for any production rule in P of the form $A \to aB$ there is an edge $A \xrightarrow{f} B$ with $L(f) = a$ and for any production rule $A \to a$ there is an edge $A \xrightarrow{w} s_1$ with $L(w) = a$. The language generated by G is the image under L of the set of labelled paths $s \to s_1$ in Σ, i.e. $\mathcal{L}(\Sigma) = L^*(\mathbf{C}(\Sigma)(s, s_1))$.

This translation is akin to the construction of a *finite state automaton* from a regular grammar. In fact, the above definition of regular grammar directly is equivalent to the definition of *non-deterministic* finite state automaton (NFA). Given a regular grammar (G, L), we can construct the span

$$V \times G_0 \xleftarrow{L \times \mathtt{dom}} G_1 \xrightarrow{cod} G_0$$

which induces a relation $\mathtt{im}(G_1) \subseteq V \times G_0 \times G_0$, which is precisely the transition table of an NFA with states in G_0, alphabet symbols V and transitions in $\mathtt{im}(G_1)$. If we require that the relation $\mathtt{im}(G_1)$ be a *function* — i.e. for any $(v, a) \in V \times G_0$ there is a unique $b \in G_0$ such that $(v, a, b) \in \mathtt{im}(G_1)$ — then this defines a *deterministic* finite state automaton (DFA). From any NFA, one may build a DFA by blowing up the state space. Indeed relations $X \subseteq V \times G_0 \times G_0$ are the same as functions $V \times G_0 \to \mathcal{P}(G_0)$ where $\mathcal{P}$ denotes the powerset construction. So any NFA $X \subseteq V \times G_0 \times G_0$ can be represented as a DFA $V \times \mathcal{P}(G_0) \to \mathcal{P}(G_0)$.

Now that we have shown how to recover the original definition of regular grammars, consider the following folklore result from formal language theory.

Proposition 1.1.11. *Regular languages are closed under intersection and union.*

Proof. Suppose G and G' are regular grammars, with starting states q_0, q_0' and terminating states q_1, q_1'.

Taking the cartesian product of the underlying graphs $G \times G' = G_1 \times G_1' \rightrightarrows G_0 \times G_0'$ we can define a regular grammar $G \cap G' \subseteq G \times G'$ with starting state (q_0, q_0'), terminating state (q_1, q_1') and such that there is an edge between (a, a') and (b, b') whenever there are edges $a \xrightarrow{f} b$ in G and $a' \xrightarrow{f'} b'$ in G' with the same label $L(f) = L'(f')$. Then an arrow from (q_0, q_0') to (q_1, q_1') in $G \cap G'$ is the same as a pair of arrows $q_0 \to q_1$ in G and $q_0' \to q_1'$ in G'. Therefore $\mathcal{L}(G \cap G') = \mathcal{L}(G) \cap \mathcal{L}(G')$. Proving the first part of the statement.

Moreover, the disjoint union of graphs $G + G'$ yields a regular grammar $G \cup G'$ by identifying q_0 with q_0' and q_1 with q_1'. Then an arrow $q_0 \to q_1$ in $G \cup G'$ is either an arrow $q_0 \to q_1$ in G or an arrow $q_0' \to q_1'$ in G'. Therefore $\mathcal{L}(G \cup G') = \mathcal{L}(G) \cup \mathcal{L}(G')$. □

The constructions used in this proof are canonical constructions in the category of regular grammars **Reg**. Note that $\mathbf{Reg} = 2\backslash\mathbf{Graph}/V$ is both a slice and a coslice category. Moreover, **Graph** has all limits and colimits. While coslice categories reflect limits and slice categories reflect colimits, we cannot compose these two statements to show that **Reg** has all limits and colimits. However, we can prove explicitly that the constructions defined above give rise to the categorical product and coproduct. We do not know whether **Reg** also has equalizers and coequalizers which would yield all finite limits and colimits.

Proposition 1.1.12. *The intersection $\cap$ of NFAs is the categorical product in* **Reg**. *The union $\cup$ of NFAs is the coproduct in* **Reg**.

Proof. To show the first part. Suppose we have two morphisms of regular grammars $f : H \to G$ and $g : H \to G'$. These must respect the starting and terminating symbols as well as the labelling homomorphism. We can construct a homomorphism of signatures $< f, g >: H \to G \cap G'$ where $G \cap G' = G \times_V G'$ with starting point (q_0, q_0') and endpoint (q_1, q_1') as defined above. $< f, g >$ is given on vertices by $< f, g >_0 (x) = (f_0(x), g_0(x))$ and on arrows by $< f, g >_1 (h) = (f_0(h), g_0(h))$. Since $L(f(h)) = L(h) = L(g(h))$, this defines a morphism of regular grammars $< f, g >: H \to G \cap G'$. There are projections $G \cap G' \to G, G'$ induced by the projections $\pi_0, \pi_1 : G \times G' \to G, G'$, and it is easy to check that $\pi_0 \circ < f, g >= f$ and $\pi_1 \circ < f, g >= g$ where π_0 and π_1 are the projections. Now, suppose that there is some $k : H \to G \cap G'$ with $\pi_0 \circ k = f$ and $\pi_1 \circ k = g$, k then the underlying function $H \to G \times G'$ must be equal to $< f, g >$ and thus also as morphisms of regular grammar $k =< f, g >$. Therefore $G \cap G'$ is the categorical product in **Reg**.

Similarly the union is the coproduct in **Reg**. Given any pair of morphisms $f : G \to H$ and $g : G' \to H$, we may define $[f, g] : G \cup G' \to H$ on vertices by $[f, g](x) = f(x)$ if $x \in G$ and $[f, g](x) = g(x)$ if $x \in G'$ and on similarly on edges. We have that $[f, g](q_0) = f(q_0) = g(q_0') = q_1$ since q_0 and q_0' are identified in $G \cup G'$ and $L([f, g](h))$ is either of these equal expressions $L(f(h)) = L(h) = L(g(h))$, i.e. $[f, g]$ is a morphism of regular grammars. Let $i : G \to G \cup G'$ and $i' : G' \to G \cup G'$ be the injections into the union. We have $[f, g] \circ i = f$ and $[f, g] \circ i' = g$. Moreover, for any other morphism $k : G \cup G' \to H$ with $k \circ i = f$ and $k \circ i' = g$, we must have that $k(h) = f(h)$ whenever

$h \in G \subseteq G \cup G'$ and $k(h) = g$ otherwise, i.e. $k = [f, g]$. Therefore $G \cup G'$ satisfies the universal property of the coproduct in **Reg**. □

We now study the parsing problem for regular grammars. First, consider the *non-emptiness problem*, which takes as input a regular grammar G and returns "no" if $\mathcal{L}(G)$ is empty and "yes" otherwise.

Proposition 1.1.13. *The non-emptiness problem is equivalent to* ∃`Path`

Proof. $\mathcal{L}(G)$ is non-empty if and only if there is an arrow from s_0 to s_1 in G. □

Now, consider the problem of recognizing the language of a regular grammar G, also known as `Parsing`. We define the proof-relevant version which, of course, has a corresponding decision problem ∃`Parsing`.

Definition 1.1.14. `Parsing`

Input: G, $u \in V^*$
Output: $f \in \mathbf{C}(G)(s_0, s_1)$ *such that* $L^*(f) = u$.

Given a string $u \in V^*$, we may build the regular grammar G_u given by the path-graph with edges labelled according to u, so that $\mathcal{L}(G_u) = \{ u \}$. Then the problem of deciding whether there is a parse for u in G reduces to the non-emptiness problem for the intersection $G \cap G_u$. This has the following consequence.

Proposition 1.1.15. ∃`Parsing` *is equivalent to* ∃`Path` *and is thus* `NL`*-complete. Similarly,* `Parsing` *is* `FNL`*-complete.*

At the end of the section, we give a simple algorithm for parsing regular grammars based on the composition of arrows in a free category. In order to model the higher levels of Chomsky's hierarchy, we need to equip our categories with more structure.

1.1.3 cat.Arrow

We have introduced free categories and shown how they appear in formal language theory. These structures have moreover a natural implementation in object-oriented Python, which we now describe. In order to implement a free category in Python we need to define three classes: `cat.Ob` for objects, `cat.Arrow` for morphisms and `cat.Box` for generators. Objects are initialised by providing a name.

Listing 1.1.16. Objects in a free category.

```
class Ob:
    def __init__(self, name):
        self.name = name
```

Arrows, i.e. morphisms of free categories, are given by lists of boxes with matching domains and codomains. In order to initialise a `cat.Arrow`, we provide a domain, a codomain and a list of boxes. The class comes with a method `Arrow.then` for composition and a static method `Arrow.id` for generating identity arrows.

Listing 1.1.17. Arrows in a free category.

```
class Arrow:
    def __init__(self, dom, cod, boxes, _scan=True):
        if not isinstance(dom, Ob) or not isinstance(cod, Ob):
            raise TypeError()
        if _scan:
            scan = dom
            for depth, box in enumerate(boxes):
                if box.dom != scan:
                    raise AxiomError()
                scan = box.cod
            if scan != cod:
                raise AxiomError()
        self.dom, self.cod, self.boxes = dom, cod, boxes

    def then(self, *others):
        if len(others) > 1:
            return self.then(others[0]).then(*others[1:])
        other, = others
        if self.cod != other.dom:
            raise AxiomError()
        return Arrow(self.dom, other.cod, self.boxes + other.boxes, _scan=False))

    @staticmethod
    def id(dom):
        return Arrow(self, dom, dom, [], _scan=False)

    def __rshift__(self, other):
        return self.then(other)

    def __lshift__(self, other):
        return other.then(self)
```

When `_scan == False` *we do not check that the boxes in the arrow compose. This allows us to avoid checking composition multiple times for the same* `Arrow`*. The methods* `__rshift__` *and* `__lshift__` *allow to use the syntax* `f >> g` *and* `g << f` *for the composition of instances of the* `Arrow` *class.*

Finally, generators of the free category are special instances of `Arrow`, initialised by a name, a domain and a codomain.

Listing 1.1.18. Generators of a free category.

```
class Box(Arrow):
    def __init__(self, name, dom, cod):
        self.name, self.dom, self.cod = name, dom, cod
        Arrow.__init__(self, dom, cod, [self], _scan=False)
```

The subclassing mechanism in Python allows for `Box` *to inherit all the* `Arrow` *methods, so that there is essentially no difference between a box and an arrow with one box.*

Remark 1.1.19. *It is often useful to define dataclass methods such as* `__repr__`, `__str__` *and* `__eq__` *to represent, print and check equality of objects in a class. Similarly, other standard methods such as* `__hash__` *may be overwritten and used as syntactic gadgets. We set* `self.name = name` *although, in DisCoPy, this parameter is immutable and e.g.* `Ob.name` *is implemented as a* `@property` *method. In order to remain concise, we will omit these methods when defining further classes.*

We now have all the ingredients to compose arrows in free categories. We check that the axioms of categories hold for `cat.Arrow`s on the nose.

Listing 1.1.20. Axioms of free categories.

```
x, y, z = Ob('x'), Ob('y'), Ob('z')
f, g, h = Box('f', x, y), Box('g', y, z), Box('h', z, x)
assert f >> Arrow.id(y) == f == Arrow.id(x) >> f
assert (f >> g) >> h == f >> g >> h == (f >> g) >> h
```

A signature is a pair of lists, for objects and generators. A homomorphism between signatures is a pair of Python dictionnaries. Functors between the corresponding free categories are initialised by a pair of mappings `ob`, from objects to objects, and `ar` from boxes to arrows. The call method of `Functor` allows to evaluate the image of composite arrows.

Listing 1.1.21. Functors from a free category.

```
class Functor:
    def __init__(self, ob, ar):
        self.ob, self.ar = ob, ar

    def __call__(self, arrow):
        if isinstance(arrow, Ob):
            return self.ob[arrow]
        if isinstance(arrow, Box):
            return self.ar[arrow]
        if isinstance(arrow, Arrow):
            return Arrow.id(self(arrow.dom)).then(*map(self, arrow))
        raise TypeError()
```

We check that the axioms hold.

```
x, y = Ob('x'), Ob('y')
f, g = Box('f', x, y), Box('g', y, x)
F = Functor(ob={x : y, y: x}, ar={f : g, g: f})
assert F(f >> g) == F(f) >> F(g)
assert F(Arrow.id(x)) == Arrow.id(F(x))
```

As a linguistic example, we use the composition method of `Arrow` to write a simple parser for regular grammars.

Listing 1.1.22. Regular grammar parsing.

```
from discopy.cat import Ob, Box, Arrow, AxiomError
s0, x, s1 = Ob('s0'), Ob('x'), Ob('s1')
A, B, C = Box('A', s0, x), Box('B', x, x), Box('A', x, s1)
grammar = [A, B, C]
def is_grammatical(string, grammar):
    arrow = Arrow.id(s0)
    bool = False
    for x in string:
        for box in grammar:
            if box.name == x:
                try:
                    arrow = arrow >> box
                    bool = True
                    break
                except AxiomError:
                    bool = False
        if not bool:
            return False
    return bool
assert is_grammatical("ABBA", grammar)
assert not is_grammatical("ABAB", grammar)
```

So far, we have only showed how to implement *free* categories and functors in Python. However, the same procedure can be repeated. Implementing a category in Python amounts to defining a pair of classes for objects and arrows and a pair of methods for identity and composition. In the case of `Arrow`, and the syntactic structures of this chapter, we are able to check that the axioms hold in Python. In the next chapter, these arrows will be mapped to concrete Python functions, for which equality cannot be checked.

1.2 Trees

Context-free grammars (CFGs) emerged from the linguistic work of Chomsky [Cho56] and are used in many areas of computer science. They are obtained from regular grammars by allowing production rules to have more than one output symbol, resulting in tree-shaped derivations. Following Walters [Wal89a] and Lambek [Lam99b], we formalise CFGs as operadic signatures and their derivations as trees in the corresponding free operad. Morphisms of free operads are trees with labelled nodes and edges. We give an implementation – the `operad` module of DisCoPy – which satisfies the axioms of operads on the nose and interfaces with the library NLTK [LB02].

1.2.1 Operads

Definition 1.2.1 (Operadic signature)**.** *An operadic signature is a pair of functions:*

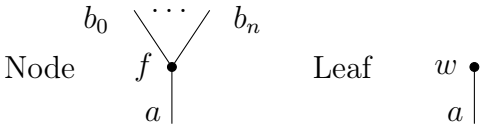
$$G_0^* \xleftarrow{\texttt{dom}} G_1 \xrightarrow{\texttt{cod}} G_0$$

where G_1 is the set of nodes *and G_0 a set of* objects*. A morphism of operadic signatures $\varphi : G \to \Gamma$ is a pair of functions $\varphi_0 : G_0 \to \Gamma_0$, $\varphi_1 : G_1 \to \Gamma_1$ such that the following diagram commutes:*

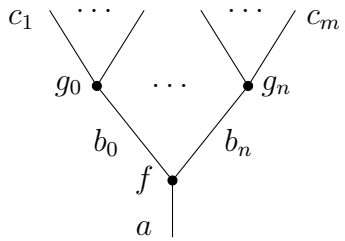
$$\begin{array}{ccccc} G_0^* & \xleftarrow{\texttt{dom}} & G_1 & \xrightarrow{\texttt{cod}} & G_0 \\ \downarrow{\scriptstyle \varphi_0^*} & & \downarrow{\scriptstyle \varphi_1} & & \downarrow{\scriptstyle \varphi_0} \\ \Gamma_0^* & \xleftarrow[\texttt{dom}]{} & \Gamma_1 & \xrightarrow[\texttt{cod}]{} & \Gamma_0 \end{array}$$

With these morphisms, operadic signatures form a category denoted **OpSig***.*

A node or box in an operadic signature is denoted $b_0 \dots b_n \xrightarrow{f} a$. Nodes of the form $a \xrightarrow{w} \epsilon$ for the empty string ϵ are called *leaves*.



Definition 1.2.2 (Operad)**.** *An operad* **O** *is an operadic signature equipped with a composition operation $\cdot : \prod_{b_i \in \vec{b}} \mathbf{Op}(\vec{c_i}, b_i) \times \mathbf{Op}(\vec{b}, a) \to \mathbf{Op}(\vec{c}, a)$ defined for any $a \in \mathbf{O}_0$, $\vec{b}, \vec{c} \in \mathbf{O}_0^*$. Given $f : \vec{b} \to a$ and $g_i : \vec{c_i} \to b_i$, the composition of $\vec{g}$ with f is denoted graphically as follows.*

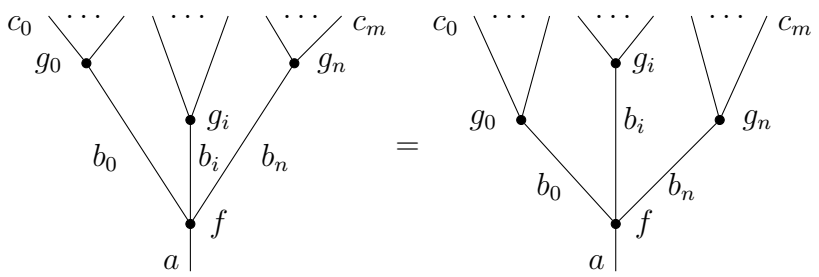


(1.3)

We ask that they satisfy the following axioms:

1. *for any* $a \in \mathbf{O}_0$ *there is an identity morphism* $\mathtt{id}_a \in \mathbf{Op}(a,a)$ *(identities).*

2. *for any* $f : \vec{b} \to a$, $f \cdot \mathtt{id}_a = f = \vec{\mathtt{id}}_{b_i} \cdot f$ *(unit law).*

3. 

(1.4)

An algebra $F : \mathbf{O} \to \mathbf{N}$ *is a morphism of operadic signature which respects composition, i.e. such that whenever* $\vec{c} \xrightarrow{\vec{g}} \vec{b} \xrightarrow{f} a$ *in* $\mathbf{O}$ *we have* $F(\vec{g} \cdot f) = F\vec{(g)} \cdot F(f)$*. With algebras as morphisms, operads form a category* **Operad**.

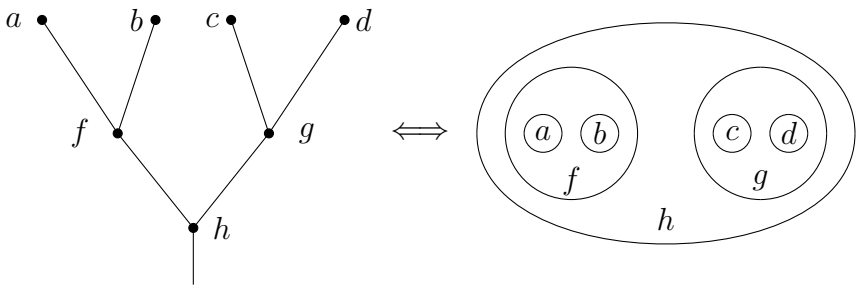
Remark 1.2.3. *The diagrammatic notation we are using is not formal yet, but it will be made rigorous in the following section where we will see that operads are instances of monoidal categories and thus admit a formal graphical language of string diagrams.*

Given an operadic signature G, we may build the free operad over G, denoted $\mathbf{Op}(G)$. Morphism of $\mathbf{Op}(G)$ are labelled trees with nodes from G and edges labelled by elements of the generating objects B. Two trees are equivalent (or congruent) if they can be deformed continuously into each other using the interchanger rules 1.4 repeatedly. The free operad construction is part of the following free-forgetful adjunction.

$$\mathbf{OpSig} \underset{U}{\overset{\mathbf{Op}}{\rightleftarrows}} \mathbf{Operad}$$

This means that for any operadic signature G and operad $\mathbf{O}$, algebras $\mathbf{Op}(G) \to \mathbf{O}$ are in bijective correspondence with morphisms of operadic signatures $G \to U(\mathbf{O})$.

Example 1.2.4 (Peirce's alpha)**.** *Morphisms in an operad can be depicted as trees, or equivalently as a nesting of bubbles:*



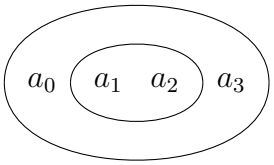
(1.5)

Interestingly, the nesting perspective was adopted by Peirce in his graphical formulation of propositional logic: the alpha graphs *[Pei06]. We may present Peirce's alpha graphs as an operad* α *with a single object* x*, variables as leaves* $a, b, c, d : 1 \to x$*,*

a binary operation $\wedge : xx \to x$ *and a unary operation* $\neg : x \to x$*, together with equations encoding the associativity of* $\wedge$ *and* $\neg\neg = \mathtt{id}_x$*. In order to encode Peirce's rules for "iteration" and "weakening", we would need to work in a preordered operad, but we omit these rules here, see [BT00] for a full axiomatisation. The main purpose here is to note that the nesting notation is sometimes more practical than its tree counterpart. Indeed, since* $\wedge$ *is associative and it is the only binary operation in* α*, when two symbols are put side by side on the page, it is unambiguous that one should take the conjunction* $\wedge$*. Therefore the nested notation simplifies reasoning and we may draw the propositional formula*

$$\neg(a_0 \wedge \neg(a_1 \wedge a_2) \wedge a_3)$$

as the following diagram:



(1.6)

1.2.2 Context-free grammars

Given a finite operadic signature G, we can intepret the nodes in G as *production rules*, the generating objects as *symbols*, and morphisms in the free operad $\mathbf{Op}(G)$ as *derivations*, obtaining the notion of a context-free grammar.

Definition 1.2.5 (Context-free grammar)**.** *A CFG is a finite operadic signature of the following shape:*

$$(B + V)^* \leftarrow G \to B$$

where B *is a set of non-terminal symbols with a specified sentence symbol* $s \in B$*,* V *is a vocabulary (a.k.a a set of terminal symbols) and* G *is a set of production rules. The language generated by* G *is given by:*

$$\mathcal{L}(G) = \{\, u \in V^* \mid \exists g : u \to s \in \mathbf{Op}(G) \,\}$$

where $\mathbf{Op}(G)$ *is the free operad of labelled trees with nodes from* G*.*

Remark 1.2.6. *Note that the direction of the arrows is the opposite of the usual direction used for CFGs, instead of seeing a derivation as a tree from the sentence symbol* s *to the terminal symbols* $u \in V^*$*, we see it as a tree from* u *to* s*. This, of course, does not change any of the theory.*

Definition 1.2.7 (Context-free language)**.** *A context-free language is a subset* $X \subseteq V^*$ *such that* $X = \mathcal{L}(G)$ *for some context-free grammar* G*.*

Example 1.2.8 (Backus Naur Form)**.** *BNF is a convenient syntax for defining context-free languages recursively. An example is the following expression:*

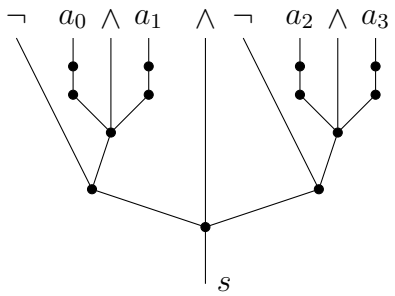
$$s \leftarrow s \wedge s \,|\, \neg s \,|\, a$$

$$a \leftarrow a_0 \,|\, a_1 \,|\, a_2 \,|\, a_3$$

which defines a CFG with seven production rules $\{s \leftarrow s \wedge s\,,\ s \leftarrow \neg s\,,\ s \leftarrow a\, a \leftarrow a_0\, a \leftarrow a_1\, a \leftarrow a_2\, a \leftarrow a_3\}$ *and such that trees with root* s *are well-formed propositional logic formulae with variables in* $\{\, a_0, a_1, a_2, a_3 \,\}$*. An example of a valid propositional formula is*

$$\neg(a_0 \wedge a_1) \ \wedge\ \neg(a_2 \wedge a_3)$$

as witnessed by the following tree:

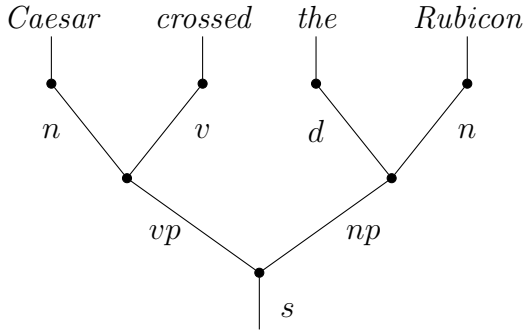


where we have omitted the types of intermediate wires for readability.

Example 1.2.9. *As a linguistic example, let* $B = \{\, n, d, v, vp, np, s \,\}$ *for nouns, prepositions, verbs, verb phrases and prepositional phrases, and let* G *be the CFG defined by the following lexical rules:*

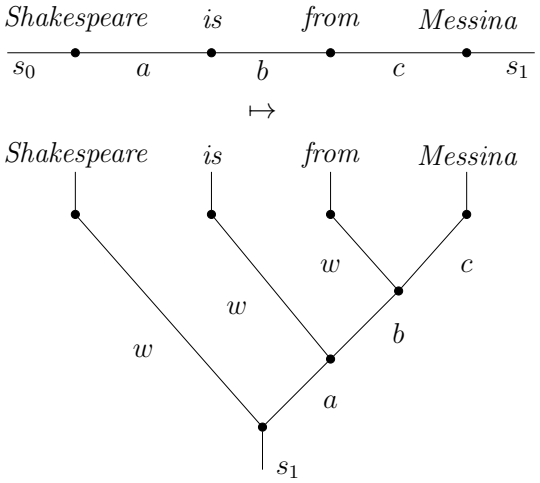
$$\mathit{Caesar} \to n \quad \mathit{the} \to d \quad \mathit{Rubicon} \to n \quad \mathit{crossed} \to v$$

together with the production rules $n \cdot v \to vp$*,* $n \cdot d \to vp$*,* $vp \cdot pp \to s$*. Then the following is a grammatical derivation:*



Example 1.2.10 (Regular grammars revisited)**.** *Any regular grammar yields a CFG.*

The translation is given by turning paths into left-handed trees as follows:



(1.7)

Not all CFGs arise in this way. For example, the language of well-bracketed expressions, defined by the CFG with a single production rule $G = \{ s \leftarrow (s) \}$, cannot be generated by a regular grammar. We can prove this using the pumping lemma. *Indeed, suppose there is a regular grammar G' such that well-bracketed expressions are paths in G'. Let n be the number of vertices in G' and consider the grammatical expression $x = (\dots()\dots)$ with $n+1$ open and $n+1$ closed brackets. If G' parses x, then there must be a path p_0 in G' with labelling $(\dots($ and a path p_1 with labelling $)\dots)$ such that $p_0 \cdot p_1 : s_0 \to s_1$ in G'. By the* pigeon hole *principle, the path p_0 must have a cycle of length $k \geq 1$. Remove this cycle from p_0 to get a new path p_0'. Then $p_0' \cdot p_1$ yields a grammatical expression $x' = (\dots()\dots)$ with $n+1-k$ open brackets and $n+1$ closed brackets. But then x' is not a well-bracketed expression. Therefore regular grammars cannot generate the language of well-bracketed expressions and we deduce that regular languages are* strictly *contained in context-free languages.*

We briefly consider the problem of parsing context-free grammars.

Definition 1.2.11. `CfgParsing`

Input: G, $u \in V^*$
Output: $f \in \mathbf{Op}(G)(u, s)$

This problem can be solved using a *pushdown automaton*, and in fact any language recognized by a pushdown automaton is context-free [Cho56]. The following result was shown independently by several researchers at the end of the 1960s.

Proposition 1.2.12. *[You67; Ear70] Context-free grammars can be parsed in cubic time.*

1.2.3 operad.Tree

Tree data structures are ubiquitous in computer science. They can implemented via the inductive definition: a tree is a root together with a list of trees. Implementing

operads as defined in this Section, presents some extra difficulties in handling types (domains and codomains) and identities. In fact, the concept of "identity tree" is not frequent in the computer science literature. Our implementation of free operads consists in the definition of classes `operad.Tree`, `operad.Box`, `operad.Id` and `operad.Algebra`, corresponding to morphisms (trees), generators (nodes), identities and algebras of free operads, respectively. A `Tree` is initialised by a `root`, instance of `Node`, together with a list of `Trees` called `branches`. Alternatively, it may be built from generating Boxs using the `Tree.__call__` method, this allows for an intuitive syntax which we illustrate below.

Listing 1.2.13. Tree in a free operad.

```
class Tree:
    def __init__(self, root, branches, _scan=True):
        if not isinstance(root, Box):
            raise TypeError()
        if not all([isinstance(branch, Tree) for branch in branches]):
            raise TypeError()
        if _scan and not root.cod == [branch.dom for branch in branches]:
            raise AxiomError()
        self.dom, self.root, self.branches = root.dom, root, branches

    @property
    def cod(self):
        if isinstance(self, Box):
            return self._cod
        else:
            return [x for x in branch.cod for branch in self.branches]

    def __repr__(self):
        return "Tree({}, {})".format(self.root, self.branches)

    def __str__(self):
        if isinstance(self, Box):
            return self.name
        return "{}({})".format(self.root.name,
                               ', '.join(map(Tree.__str__, self.branches)))

    def __call__(self, *others):
        if not others or all([isinstance(other, Id) for other in others]):
            return self
        if isinstance(self, Id):
            return others[0]
        if isinstance(self, Box):
            return Tree(self, list(others))
        if isinstance(self, Tree):
            lengths = [len(branch.cod) for branch in self.branches]
            ranges = [0] + [sum(lengths[:i + 1]) for i in range(len(lengths))]
            branches = [self.branches[i](*others[ranges[i]:ranges[i + 1]])
                        for i in range(len(self.branches))]
            return Tree(self.root, branches, _scan=False)
        raise NotImplementedError()
```

```
    @staticmethod
    def id(dom):
        return Id(dom)

    def __eq__(self, other):
        return self.root == other.root and self.branches == other.branches
```

A `Box` is initialised by label `name`, a domain object `dom` and a list of objects `cod` for the codomain.

Listing 1.2.14. Node in a free operad.

```
class Box(Tree):
    def __init__(self, name, dom, cod):
        if not (isinstance(dom, Ob) and isinstance(cod, list)
                and all([isinstance(x, Ob) for x in cod])):
            return TypeError
        self.name, self.dom, self._cod = name, dom, cod
        Tree.__init__(self, self, [], _scan=False)

    def __repr__(self):
        return "Box('{}', {}, {})".format(self.name, self.dom, self._cod)

    def __hash__(self):
        return hash(repr(self))

    def __eq__(self, other):
        if isinstance(other, Box):
            return self.dom == other.dom and self.cod == other.cod \
                        and self.name == other.name
        if isinstance(other, Tree):
            return other.root == self and other.branches == []
```

An `Id` is a special type of node, which cancels locally when composed with other trees. The cases in which identities must be removed are handled in the `Tree.__call__` method. The `Tree.__init__` method, as it stands, does not check the identity axioms. We will however always use the `__call__` syntax to construct our trees.

Listing 1.2.15. Identity in a free operad.

```
class Id(Box):
    def __init__(self, dom):
        self.dom, self._cod = dom, [dom]
        Box.__init__(self, "Id({})".format(dom), dom, dom)

    def __repr__(self):
        return "Id({})".format(self.dom)
```

We can check that the axioms of operads hold for `Tree.__call__`.

Listing 1.2.16. Axioms of free operads.

```
x, y = Ob('x'), Ob('y')
f, g, h = Box('f', x, [x, x]), Box('g', x, [x, y]), Box('h', x, [y, x])
assert Id(x)(f) == f == f(Id(x), Id(x))
left = f(Id(x), h)(g, Id(x), Id(x))
middle = f(g, h)
right = f(g, Id(x))(Id(x), Id(x), h)
assert left == middle == right == Tree(root=f, branches=[g, h])
```

Listing 1.2.17. We construct the tree from Example 1.2.9.

```
n, d, v, vp, np, s = Ob('N'), Ob('D'), Ob('V'), Ob('VP'), Ob('NP'), Ob('S')
Caesar, crossed = Box('Caesar', n, []), Box('crossed', v, []),
the, Rubicon = Box('the', d, []), Box('Rubicon', n, [])
VP, NP, S = Box('VP', vp, [n, v]), Box('NP', np, [d, n]), Box('S', s, [vp, np])
sentence = S(VP(Caesar, crossed), NP(the, Rubicon))
```

We define the `Algebra` class, which implements operad algebras as defined in 1.2.2 and is initialised by a pair of mappings: `ob` from objects to objects and `ar` from nodes to trees. These implement functorial reductions and functorial semantics of CFGs, as defined in the next section and chapter respectively.

Listing 1.2.18. Algebra of the free operad.

```
class Algebra:
    def __init__(self, ob, ar, cod=Tree):
        self.cod, self.ob, self.ar = cod, ob, ar

    def __call__(self, tree):
        if isinstance(tree, Id):
            return self.cod.id(self.ob[tree])
        if isinstance(tree, Box):
            return self.ar[tree]
        return self.ar[tree.root](*[self(branch) for branch in tree.branches])
```

Note that we parametrised the class algebra over a codomain class, which by default is the free operad `Tree`*. We may build any algebra from the free operad to an operad A by providing a class* `cod=A` *with* `A.id` *and* `A.__call__` *methods. We will see a first example of this when we interface* `Tree` *with* `Diagram` *in the next section. Further examples will be given in Chapter 2.*

We end by interfacing the `operad` module with the library NLTK [LB02].

Listing 1.2.19. Interface between `nltk.Tree` and `operad.Tree`.

```
def from_nltk(tree):
    branches, cod = [], []
    for branch in tree:
        if isinstance(branch, str):
            return Box(branch, Ob(tree.label()), [])
        else:
            branches += [from_nltk(branch)]
            cod += [Ob(branch.label())]
    root = Box(tree.label(), Ob(tree.label()), cod)
    return root(*branches)
```

This code assumes that the tree is generated from a lexicalised CFG. The `operad` *module of DisCoPy contains the more general version. We can now define a grammar in NLTK, parse it, and extract an* `operad.Tree`*. We check that we recover the correct tree for "Caesar crossed the Rubicon".*

```
from nltk import CFG
from nltk.parse import RecursiveDescentParser
grammar = CFG.fromstring("""
S -> VP NP
NP -> D N
VP -> N V
N -> 'Caesar'
V -> 'crossed'
D -> 'the'
N -> 'Rubicon'""")

rd = RecursiveDescentParser(grammar)
for x in rd.parse('Caesar crossed the Rubicon'.split()):
    tree = from_nltk(x)
assert tree == sentence
```

1.3 Diagrams

String diagrams in monoidal categories are the key tool that we use to represent syntactic structures. In this section we introduce *monoidal grammars*, the equivalent of Chomsky's unrestricted type-0 grammars. Their derivations are string diagrams in a free monoidal category. We introduce *functorial reductions* as a structured way of comparing monoidal grammars, and motivate them as a tool to reason about equivalence and normal forms for context-free grammar. String diagrams have a convenient *premonoidal encoding* as lists of layers, which allows to implement the class `monoidal.Diagram` as a subclass of `cat.Arrow`. We give an overview of the `monoidal` module of DisCoPy and its interface with `operad`.

Monoidal category	Type-0 grammar	Python
objects	strings	`Ty`
generators	production rules	`Box`
morphisms	derivations	`Diagram`
functors	reductions	`Functor`

1.3.1 Monoidal categories

Definition 1.3.1 (Monoidal signature)**.** *A monoidal signature G is a signature of the following form:*

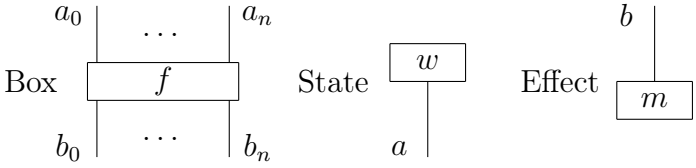
$$G_0^* \xleftarrow{\texttt{dom}} G_1 \xrightarrow{\texttt{cod}} G_0^*$$

. G is a finite monoidal signature if G_1 is finite. A morphism of monoidal signatures $\varphi : G \to \Gamma$ is a pair of maps $\varphi_0 : G_0 \to \Gamma_0$ and $\varphi_1 : G_1 \to \Gamma_1$ such that the following diagram commutes:

$$\begin{array}{ccccc} G_0^* & \xleftarrow{\texttt{dom}} & G_1 & \xrightarrow{\texttt{cod}} & G_0^* \\ \downarrow{\scriptstyle \varphi_0^*} & & \downarrow{\scriptstyle \varphi_1} & & \downarrow{\scriptstyle \varphi_0^*} \\ \Gamma_0^* & \xleftarrow[\texttt{dom}]{} & \Gamma_1 & \xrightarrow[\texttt{cod}]{} & \Gamma_0^* \end{array}$$

With these morphisms, monoidal signatures form a category **MonSig**.

Elements $f : \vec{a} \to \vec{b}$ of G_1 are called *boxes* and are denoted by the following diagram, read from top to bottom, special cases are states and effects with no inputs and outputs respectively.



Definition 1.3.2 (Monoidal category)**.** *A (strict) monoidal category is a category* $\mathbf{C}$ *equipped with a functor* $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ *called the* tensor *and a specified object* $1 \in \mathbf{C}_0$ *called the* unit*, satisfying the following axioms:*

1. $1 \otimes f = f = f \otimes 1$ *(unit law)*
2. $(f \otimes g) \otimes h = f \otimes (g \otimes h)$ *(associativity)*

for any $f, g, h \in \mathbf{C}_1$. A (strict) monoidal functor is a functor that preserves the tensor product on the nose, i.e. such that $F(f \otimes g) = F(f) \otimes F(g)$. The category of monoidal categories and monoidal functors is denoted **MonCat**.

Remark 1.3.3. *The general (non-strict) definition of a monoidal category relaxes the equalities in the unit and associativity laws to the existence of natural isomorphisms, called unitors and associators. These are then required to satisfy some coherence conditions in the form of commuting diagrams, see MacLane [Mac71]. In practice, these natural transformations are not used in calculations. MacLane's coherence theorem ensures that any monoidal category is equivalent to a strict one.*

Given a monoidal signature G we can generate the free monoidal category $\mathbf{MC}(G)$, i.e. there is a free-forgetful adjunction:

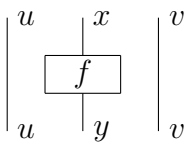
$$\mathbf{MonSig} \underset{U}{\overset{\mathbf{MC}}{\rightleftarrows}} \mathbf{MonCat}$$

This means that for any monoidal signature G and monoidal category $\mathbf{S}$, functors $\mathbf{MC}(G) \to \mathbf{S}$ are in bijective correspondence with morphisms of signatures $G \to U(\mathbf{S})$. The free monoidal category was first characterized by Joyal and Street [JS91] who showed that morphisms in $\mathbf{MC}(G)$ are topological objects called *string diagrams*. We follow the formalisation of Delpeuch and Vicary [DV19a] who provided an equivalent combinatorial definition of string diagrams.

Given a monoidal signature G, we can construct the signature of *layers*:

$$G_0^* \xleftarrow{\texttt{dom}} L(G) = G_0^* \times G_1 \times G_0^* \xrightarrow{\texttt{cod}} G_0^*$$

where for every layer $l = (u, f : x \to y, v) \in L(G)$ we define $\texttt{dom}(l) = uxv$ and $\texttt{cod}(l) = uyv$. A layer $l \in L(G)$ is denoted as follows:



The set of *premonoidal diagrams* $\mathbf{PMC}(G)$ is the set of morphisms of the free category generated by the layers:

$$\mathbf{PMC}(G) = \mathbf{C}(L(G))$$

They are precisely morphisms of free premonoidal categories in the sense of [PR97]. Morphisms $d \in \mathbf{PMC}(G)$ are lists of layers $d = (d_1, \ldots, d_n)$ such that $\texttt{cod}(d_i) = \texttt{dom}(d_{i+1})$. The data for such a diagram may be presented in a more succint way as follows.

Proposition 1.3.4 (Premonoidal encoding)**.** *[DV19a] A premonoidal diagram $d \in \mathbf{PMC}(G)$ is uniquely defined by the following data:*

1. *a domain* $\texttt{dom}(d) \in \Sigma_0^*$,
2. *a codomain* $\texttt{cod}(d) \in \Sigma_0^*$,
3. *a list of boxes* $\texttt{boxes}(d) \in G_1^n$,
4. *a list of offsets* $\texttt{offsets}(d) \in \mathbb{N}^n$.

Where $n \in \mathbb{N}$ *is the* length *of the diagram and the offsets indicate the number of boxes to the left of each wire. This data defines a valid premonoidal diagram if for* $0 < i \leq n$ *we have:*

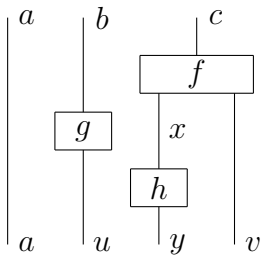
$$\texttt{width}(d)_i \geq \texttt{offsets}(d)_i + |\texttt{dom}(b_i)|$$

where the widths are defined inductively by:

$$\texttt{width}(d)_1 = \texttt{size}(\texttt{dom}(d)) \quad \texttt{width}(d)_{i+1} = \texttt{width}(d)_i + |\texttt{cod}(b_i)| - |\texttt{dom}(b_i)|$$

and $b_i = \texttt{boxes}(d)_i$.

As an example consider the following diagram:

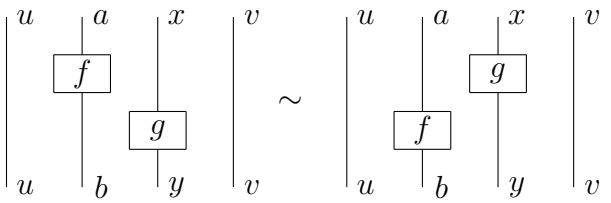


It has the following combinatorial encoding:

$$(\texttt{dom} = abc, \texttt{cod} = auyv, \texttt{boxes} = [f, g, h], \texttt{offsets} = [2, 1, 2])$$

where $f : c \to xv$, $g : b \to u$ and $h : x \to y$ and $a, b, c, x, u, y, v \in \mathcal{G}_0$. This combinatorial encoding is the underlying data-structure of both the online proof assistant Globular [BKV18] and the Python implementation of monoidal categories DisCoPy [dTC21].

Premonoidal diagrams are a useful intermediate step to define the *free monoidal category* $\mathbf{MC}(G)$ over a monoidal signature G. Indeed, morphisms in $\mathbf{MC}(G)$ are equivalence classes of the quotient of $\mathbf{PMC}(G)$ by the *interchanger rules*, given by the following relation:



(1.8)

The following result was proved by Delpeuch and Vicary [DV19a], who showed how to transalte between the combinatorial definition of diagrams and the definition of Joyal and Street as planar progressive graphs up to planar isotopy [JS91].

Proposition 1.3.5. *[DV19a]*

$$\mathbf{MC}(G) \simeq \mathbf{PMC}(G)/\sim$$

Given two representatives $d, d' : u \to v \in \mathbf{PMC}(G)$ we may check whether they are equal in $\mathbf{MC}(G)$ in cubic time [DV19a]. Assuming d and d' are boundary connected diagrams, this is done by turning d and d' into their interchanger normal form, which is obtained by applying the interchanger rules 1.8 from left to right repeatedly. For disconnected diagrams the normalization requires more machinery but can still be performed efficiently, see [DV19a] for details.

1.3.2 Monoidal grammars

Monoidal categories appear in linguistics as a result of the following change of terminology. Given a monoidal signature G, we may think of the objects in G_0^* as *strings* of symbols, the generating arrows in G_1 as *production rules* and morphisms in $\mathbf{MC}(G)$ as *derivations*. We directly obtain a definition of Chomsky's generative grammars, or string rewriting system, using the notion of a monoidal signature.

Definition 1.3.6 (Monoidal Grammar)**.** *A monoidal grammar is a finite monoidal signature G of the following shape:*

$$(V + B)^* \xleftarrow{\texttt{dom}} G \xrightarrow{\texttt{cod}} (V + B)^*$$

where V is a set of words called the vocabulary*, and B is a set of* symbols *with $s \in B$ the sentence symbol. An utterance $u \in V^*$ is grammatical if there is a string diagram $g : u \to s$ in $\mathbf{MC}(G)$, i.e. the language generated by G is given by:*

$$\mathcal{L}(G) = \{\, u \in V^* \mid \exists f \in \mathbf{MC}(G)(u, s) \,\}$$

The free monoidal category $\mathbf{MC}(G)$ is called the category of derivations *of G.*

Remark 1.3.7. *Any context-free grammar as defined in 1.2.5 yields directly a monoidal grammar.*

Proposition 1.3.8. *The languages generated by monoidal grammars are equivalent to the languages generated by Chomsky's unrestricted grammars.*

Proof. Unrestricted grammars are defined as finite relations $P \subseteq (V+B)^* \times (V+B)^*$ where V is a set of terminal symbols, B a set of non-terminals and P is a set of prodution rules [Cho56]. The only difference between this definition and monoidal grammars is that the latter allow more than one production rule between pairs of strings. However, a string u is in the language if *there exists* a derivation $g : u \to s$. Therefore the languages are equivalent. □

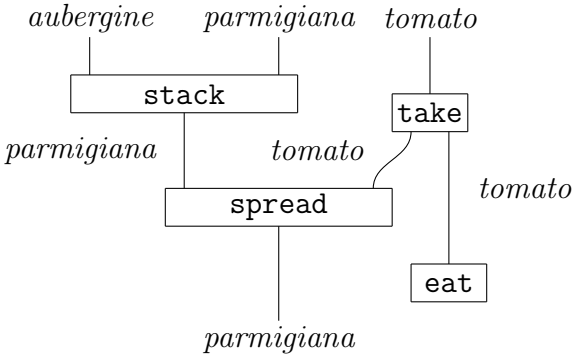
We define recursively enumerable languages as those generated by monoidal grammars, or equivalently by Chomsky's unrestricted grammars, or equivalently those recognized by Turing machines as discussed in the next paragraph.

Definition 1.3.9 (Recursively enumerable language)**.** *A recursively enumerable language is a subset $X \subseteq V^*$ such that $X = \mathcal{L}(G)$ for some monoidal grammar G.*

Example 1.3.10 (Cooking recipes)**.** *As an example of a monoidal grammar, let $V = \{aubergine, tomato, parmigiana\}$ be a set of cooking ingredients, $B = \{ parmigiana \}$ be a set of dishes and let G be a set of cooking steps, e.g.*

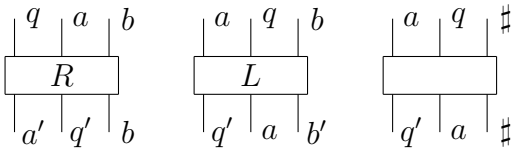
$$\texttt{stack} : a\, p \to p \quad \texttt{take} : t \to t\, t \quad \texttt{spread} : t\, p \to p \quad \texttt{eat} : t \to 1.$$

Then the derivations $u \to parmigiana$ in $\mathbf{MC}(G)$ with $u \in V^*$ are cooking recipes to make parmigiana. For instance, the following is a valid cooking recipe:



Recall that a Turing machine is given by a tuple $(Q, \Gamma, \sharp, V, \delta, q_0, s)$ where Q is a finite set of *states*, Γ is a finite set of *alphabet* symbols, $\sharp \in \Gamma$ is the *blank* symbol, $V \subseteq \Gamma \backslash \{ \sharp \}$ is the set of *input* symbols, $q_0 \in Q$ is the *initial* state, $s \in Q$ is the *accepting* state and $\delta \subseteq ((Q \backslash \{ s \}) \times \Gamma) \times (Q \times \Gamma) \times \{ L, R \}$ is a *transition* table, specifying the next state in Q from a current state, the symbol in Γ to overwrite the current symbol pointed by the head and the next head movement (left or right). At the start of the computation, the machine is in state q_0 and the tape contains a string of initial symbols $u \in V^*$ followed by the blank symbol $\sharp$ indicating the end of the tape. The computation is then performed by applying transitions according to δ until the accepting state s is reached. We assume that the transition δ doesn't overwrite the blank symbol and that it leaves it at the end of the tape.

A Turing machine may be encoded in a monoidal grammar as follows. The set of non-terminal symbols is $B = (\Gamma \backslash V) + Q$, the set of terminal symbols is V and the sentence type is $s \in B$. The production rules in G are given by:



(1.9)

for all $a, a', b, b' \in \Gamma \backslash \{ \sharp \}$, $q, q' \in Q$ such that $\delta((q, a), (q', a', R)) = 1$ and $\delta((q, b), (q', b', L)) = 1$ and $\delta((q, \sharp), (q', \sharp, L)) = 1$. Note that the last rule ensures that the blank symbol $\sharp$ is always left at the end of the tape and never overwritten. Then we have that morphisms in $\mathbf{MC}(G)(q_0\, w\, \sharp, u\, s\, v)$ are terminating runs of the Turing machine. In order to express these runs as morphisms $w \to s$ we may erase the content of the tape once we reach the accepting state by adding a production rule $xsy \to s$ to G for any $x, y \in B$. Using this encoding we can prove the following proposition.

Proposition 1.3.11. *The parsing problem for monoidal grammars is undecidable.*

Proof. The encoding of Turing machines into monoidal grammars given above reduces the problem of parsing monoidal grammars to the Halting problem for Turing machines. Therefore it is an undecidable problem. □

1.3.3 Functorial reductions

We now come to the question of reduction and equivalence for grammars. Several definitions are available in the literature and we introduce three alternative notions of varying strengths. The most established notion of equivalence between CFGs — known as *weak equivalence* — judges a grammar from the language it generates.

Definition 1.3.12 (Weak reduction)**.** *Let G and G' be monoidal grammars over the same vocabulary V. We say that G reduces weakly to G', denoted $G \leq G'$, if $\mathcal{L}(G) \subseteq \mathcal{L}(G')$. G is weakly equivalent to G' if $\mathcal{L}(G) = \mathcal{L}(G')$.*

Even if two grammars are weakly equivalent, they may generate their sentences in completely different ways. This motivates the definition of a stronger notion of equivalence, which does not only ask for the generated languages to be equal, but also for the corresponding derivations to be the same. This notion has been studied in a line of work connecting context-free grammars (CFGs) and algebraic signatures [HR76; Gog+77; BM20], which we discuss below.

Definition 1.3.13 (Strong reduction)**.** *Let G and G' be monoidal grammars over the same vocabulary V. A strong reduction from G to G' is a morphism of monoidal signatures $f : G \to G'$ such that $f_0(v) = v$ for any $v \in V$ and $f_0(s) = s'$. With strong reductions as morphisms, monoidal grammars over V form a subcategory of* **MonSig** *denoted* $\mathbf{Grammar}_V$*. We say that G and G' are strongly equivalent if they are isomorphic in* $\mathbf{Grammar}_V$*.*

Note first that strong reduction subsumes its weak counterpart. Indeed given a morphism $f : G \to G'$, we get a functor $\mathbf{MC}(f) : \mathbf{MC}(G) \to \mathbf{MC}(G')$ mapping syntax trees of one grammar into syntax trees of the other. Since $f_0(v) = v$ for all $v \in V$ and $f_0(s) = s'$, there is an induced function $\mathbf{MC}(f) : \mathbf{MC}(G)(u, s) \to \mathbf{MC}(G')(u, s')$ for any $u \in V^*$, which implies that $\mathcal{L}(G) \subseteq \mathcal{L}(G')$.

A strong reduction f is a consistent relabelling of the nodes and types of the underlying operadic signature. This often results in too strict of a notion, since it relates very few grammars together. We introduce an intermediate notion of reduction between grammars, which we call *functorial reduction*.

Definition 1.3.14 (Functorial reduction)**.** *Let G and G' be monoidal grammars over the same vocabulary V. A functorial reduction from G to G' is a functor $F : \mathbf{MC}(G) \to \mathbf{MC}(G')$ such that $F_0(v) = v$ for all $v \in V$ and $F_0(s) = s'$. A functorial equivalence between G and G' is a pair of functors $F : \mathbf{MC}(G) \to \mathbf{MC}(G')$ and $F' : \mathbf{MC}(G) \to \mathbf{MC}(G')$. With functorial reductions as morphisms, monoidal grammars over V form a category* $\mathbf{Grammar}_V$*.*

Remark 1.3.15. *The passage from strong to functorial reductions can be seen as as a Kleisli category construction. The free operad functor* $\mathbf{MC} : \mathbf{MonSig} \to \mathbf{MonCat}$ *induces a monad* $U \circ \mathbf{MC} : \mathbf{MonSig} \to \mathbf{MonSig}$*. We can construct the Kleisli category* $\mathbf{Kl}(U \circ \mathbf{MC})$ *with objects given by operadic signatures and morphisms given by morphisms of signatures* $f : G \to U\mathbf{MC}(G')$*. Equivalently, a morphism* $G \to G'$ *in* $\mathbf{Kl}(U \circ \mathbf{MC})$ *is a functor* $\mathbf{MC}(G) \to \mathbf{MC}(G')$ *since* $\mathbf{MC} \dashv U$*.* $\mathbf{MC}$ *to* $\mathbf{Cfg}_V$ *we still have an adjunction* $\mathbf{MC} \dashv U$ *and the following equivalence:*

$$\mathbf{Grammar}_V \simeq \mathbf{Kl}(U \circ \mathbf{MC}).$$

Functorial reductions can be computed in logarithmic space. We give a proof, an alternative proof is given by the code for `Functor.__call__`.

Proposition 1.3.16. *Funtorial reductions can be computed in log-space* `L`*.*

Proof. Let G and G' be monoidal grammars, a functorial reduction from G to G' is a functor $F : \mathbf{MC}(G) \to \mathbf{MC}(G')$. By the universal property of the free monoidal category $\mathbf{MC}(G)$ the data for such a functor is a finite homomorphism of signatures $G \to U(\mathbf{MC}(G'))$, i.e. a collection of morphisms $\{ F(f) \}_{f \in G}$. Consider the problem which takes as input a diagram $g : x \to y \in \mathbf{MC}(G)$ and a functorial reduction $F : G \to U(\mathbf{MC}(G'))$ and outputs $F(g) \in \mathbf{MC}(G')$. We assume that we have premonoidal representations of g and $F(f)$ for every production rule $f \in G$, i.e. they all come as a pair of lists for boxes and offsets. In order to compute $F(g)$ we run through the list of boxes and replace each box f of g by $F(f)$ adding the offset of f to every offset in $F(f)$. This can be computed using a constant number of counters (one for the index of the box in the list, one for the offset and one for the pointer to $F(f)$) thus functorial reductions are in logspace. □

From the monoidal definition of weak, strong and functorial reduction we derive the corresponding notions for regular and CFGs using the following diagram.

$$\begin{array}{ccccc} \mathbf{Cat} & \hookrightarrow & \mathbf{Operad} & \hookrightarrow & \mathbf{MonCat} \\ {\scriptstyle U}\downarrow\uparrow{\scriptstyle \mathbf{C}} & & {\scriptstyle U}\downarrow\uparrow{\scriptstyle \mathbf{Op}} & & {\scriptstyle U}\downarrow\uparrow{\scriptstyle \mathbf{MC}} \\ \mathbf{Graph} & \hookrightarrow & \mathbf{OpSig} & \hookrightarrow & \mathbf{MonSig} \end{array}$$

We can now compare regular, context-free and unrestricted grammars via the notion of reduction.

Proposition 1.3.17. *Any regular grammar strongly reduces to a CFG and any CFG to a monoidal grammar.*

Proof. This is done by proving that the injections in the diagram above exist and make the diagram commute. □

The functorial notion of reduction sits in-between the weak and strong notions. As shown above, any strong reduction $f : G \to G'$ induces a functorial reduction $\mathbf{MC}(f) : \mathbf{MC}(G) \to \mathbf{MC}(G')$ via the free construction $\mathbf{MC} : \mathbf{MonSig} \to \mathbf{MonCat}$ and

the existence of such a functor induces an inclusion of the corresponding languages. However, not all functorial reductions are strong. It is well-known that any CFG can be lexicalised without losing expressivity, the resulting grammar is functorially and not strongly equivalent to the original CFG.

Definition 1.3.18 (Lexicalised CFG)**.** *A lexicalised CFG is an operadic signature of the following shape:*

$$B^* + V \leftarrow G \rightarrow B$$

In other words, all production rules involving terminal symbols in V are of the form $w \to b$ for $w \in V$ and $b \in B$. These are called lexical rules.

Proposition 1.3.19. *Any CFG is functorially (and not strongly) equivalent to a lexicalised CFG.*

Proof. For any CFG $(B+V)^* \leftarrow G \rightarrow B$ we can build a lexicalised CFG $B'^* + V \leftarrow G' \rightarrow B'$ where $B' = B+V$ and $G' = G + \{ v \to v \}_{v \in V}$, where we distinguish between the two copies of V. There is a functor $F : \mathbf{MC}(G) \to \mathbf{MC}(G')$ given on objects by $F_0(x) = x$ for $x \in B+V$ and on arrows $f : \vec{y} \to x$ by $F_1(f) = \vec{g} \cdot f$ where $g_i : y_i \to y_i$ is the identity if $y_i \in B$ and is a lexical rule $y_i \to y_i$ if $y_i \in V$. Similarly for the other direction, there is a functor $F' : \mathbf{MC}(G') \to \mathbf{MC}(G)$ given on objects by $F'_0(x) = x$ for all $x \in B+V$ and on arrows by $F'_1(f) = f$ for $f \in G$ and $F_1(v \to v) = \mathtt{id}_v$. Therefore G and G' are functorially equivalent.

Note that even the G' is *not* strongly equivalent to G. Indeed strong equivalence would imply that there is a bijection between the underlying sets of symbols, i.e. $|B+V| = |B+2V|$ which is only true for grammars over an empty vocabulary. □

We now study two useful normal forms for CFGs.

Definition 1.3.20 (Chomsky normal form)**.** *A CFG G is in Chomsky normal form if it has the following shape:*

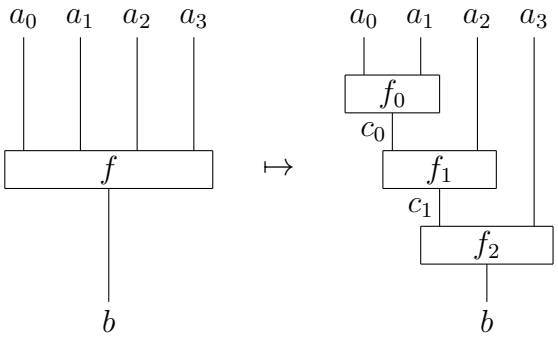
$$B^2 + V \leftarrow G \rightarrow B$$

i.e. all production rules are of the form $w \to a$ or $bc \to a$ for $w \in V$ and $a, b, c \in B$.

Proposition 1.3.21. *Any CFG G is weakly equivalent to a CFG G' in Chomsky normal form, such that the reduction from G to G' is functorial.*

Proof. Fix any CFG G. Without loss of generality, we may assume that G is lexicalised $B^* + V \leftarrow G \rightarrow B$. In order to construct G', we start by setting $G' = \{ f : \vec{a} \to b \in G \mid |\vec{a}| \leq 2 \}$. Then, for any production rule $f : \vec{a} \to b$ in G such that $k = |\vec{a}| > 2$ we add $k-1$ new rules f_i and $k-2$ new symbols c_i to G' given by $\{ f_i : c_i a_{i+1} \to c_{i+1} \}_{i=0}^{k-2}$ where $c_0 = a_0$ and $c_{k-1} = b$. This yields a CFG G' in Chomsky normal form. There is a functorial reduction from G to G' given by mapping production rules f in G to left-handed trees with f_is as nodes, as in the

following example:



(1.10)

This implies that $\mathcal{L}(G) \subseteq \mathcal{L}(G')$. Now suppose $\vec{u} \in \mathcal{L}(G')$, i.e. there is a tree $g : \vec{u} \to s$ in $\mathbf{Op}(G')$. By construction, if some $f_i : c_i a_{i+1} \to c_{i+1}$ appears as an node in g then all the f_is must appear as a sequence of nodes in g, therefore g is in the image of a tree in $\mathbf{Op}(G)(\vec{u}, s)$ and $\vec{u} \in \mathcal{L}(G)$. Therefore $\mathcal{L}(G) = \mathcal{L}(G')$. □

Since the reduction from a CFG to its Chomsky normal form is functorial, the translation can be performed in logspace. Indeed, we will show in the next section that the problem of applying a functor between free monoidal categories (of which operad algebras are an example) is in `NL`. We end with an even weaker example of equivalence between grammars.

Definition 1.3.22 (Greibach normal form)**.** *A CFG G is in Greibach normal form if it has the following shape:*

$$V \times B^* \leftarrow G \to B$$

i.e. every production rule is of the form $wb \to a$ for $a \in B$, $b \in B^*$ and $w \in V$.

Proposition 1.3.23. *[Gre65] Any CFG is weakly equivalent to one in Greibach normal form and the conversion can be performed in poly-time.*

We will use these normal forms in the next section, when we discuss functorial reductions between categorial grammars and their relationship with context-free grammars.

1.3.4 monoidal.Diagram

We now present `monoidal`, the key module of DisCoPy which allows to compute with diagrams in free monoidal categories. We have defined free monoidal categories via the concepts of layers and premonoidal diagrams. These have a natural implementation in object-oriented Python, consisting in the definition of classes `monoidal.Ty`, `monoidal.Layer` and `monoidal.Diagram`, for types, layers and diagrams in free premonoidal categories, respectively. Types are tuples of objects, equipped with a `.tensor` method for the monoidal product.

Listing 1.3.24. Types of free monoidal categories.

```
from discopy import cat

class Ty(cat.Ob):
    def __init__(self, *objects):
        self.objects = tuple(
            x if isinstance(x, cat.Ob) else cat.Ob(x) for x in objects)
        super().__init__(self)

    def tensor(self, *others):
        for other in others:
            if not isinstance(other, Ty):
                raise TypeError()
        objects = self.objects + [x for t in others for x in t.objects]
        return Ty(*objects)

    def __matmul__(self, other):
        return self.tensor(other)
```

Layers are instances of `cat.Box`, initialised by triples (u, f, v) for a pair of types u, v and a box f.

Listing 1.3.25. Layer in a free premonoidal category.

```
class Layer(cat.Box):
    def __init__(self, left, box, right):
        self.left, self.box, self.right = left, box, right
        dom, cod = left @ box.dom @ right, left @ box.cod @ right
        super().__init__("Layer", dom, cod)
```

DisCoPy diagrams are initialised by a domain, a codomain, a list of boxes and a list of offsets. It comes with methods `Diagram.then`, `Diagram.tensor` and `Diagram.id` for composing, tensoring and generating identity diagrams. As well as a method for `normal_form` which allows to check monoidal equality of two premonoidal diagrams.

Listing 1.3.26. Diagram in a free premonoidal category.

```
class Diagram(cat.Arrow):
    def __init__(self, dom, cod, boxes, offsets, layers=None):
        if layers is None:
            layers = cat.Id(dom)
            for box, off in zip(boxes, offsets):
                left = layers.cod[:off] if layers else dom[:off]
                right = layers.cod[off + len(box.dom):]\
                    if layers else dom[off + len(box.dom):]
                layers = layers >> Layer(left, box, right)
            layers = layers >> cat.Id(cod)
        self.boxes, self.layers, self.offsets = boxes, layers, tuple(offsets)
        super().__init__(dom, cod, layers, _scan=False)

    def then(self, *others):
        if len(others) > 1:
            return self.then(others[0]).then(*others[1:])
```

```
        other, = others
        return Diagram(self.dom, other.cod,
                       self.boxes + other.boxes,
                       self.offsets + other.offsets,
                       layers=self.layers >> other.layers)

    def tensor(self, other):
        dom, cod = self.dom @ other.dom, self.cod @ other.cod
        boxes = self.boxes + other.boxes
        offsets = self.offsets + [n + len(self.cod) for n in other.offsets]
        layers = cat.Id(dom)
        for left, box, right in self.layers:
            layers = layers >> Layer(left, box, right @ other.dom)
        for left, box, right in other.layers:
            layers = layers >> Layer(self.cod @ left, box, right)
        return Diagram(dom, cod, boxes, offsets, layers=layers)

    @staticmethod
    def id(dom):
        return Diagram(dom, dom, [], [], layers=cat.Id(dom))

    def interchange(self, i, j, left=False):
        ...

    def normal_form(self, normalizer=None, **params):
        ...

    def draw(self, **params):
        ...
```

`Diagram`*s always a carry a* `cat.Arrow` *called* `layers`*, which may be thought of as a witness that the diagram is well-typed. If no* `layers` *are provided, the* `Diagram.__init__` *method computes the layers and checks that they compose. A* `cat.AxiomError` *is returned when the layers do not compose. The* `Diagram.interchange` *method allows to change the order of layers in a diagram when they commute, and returns an* `InterchangerError` *when they don't. The* `Diagram.normal_form` *method implements the algorithm of [DV19a], see the* `rewriting` *module of DisCoPy. The* `Diagram.draw` *method is implemented in the* `drawing` *module and allows to render a diagram via matplotlib [] as well as generating a TikZ [] output for academic papers.*

Finally, a `Box` is initialised by a name together with domain and codomain types.

Listing 1.3.27. Box in a free monoidal category.

```
class Box(cat.Box, Diagram):
    def __init__(self, name, dom, cod, **params):
        cat.Box.__init__(self, name, dom, cod, **params)
        layer = Layer(dom[0:0], self, dom[0:0])
        layers = cat.Arrow(dom, cod, [layer], _scan=False)
        Diagram.__init__(self, dom, cod, [self], [0], layers=layers)
```

We check that the axioms for monoidal categories hold up to interchanger.

Listing 1.3.28. Axioms of free monoidal categories

```
x, y, z, w = Ty('x'), Ty('y'), Ty('z'), Ty('w')
f0, f1, f2 = Box('f0', x, y), Box('f1', z, w), Box('f2', z, w)
d = Id(x) @ f1 >> f0 @ Id(w)
assert f0 @ (f1 @ f2) == (f0 @ f1) @ f2
assert f0 @ Diagram.id(Ty()) == f0 == Diagram.id(Ty()) @ f0
assert d == (f0 @ f1).interchange(0, 1)
assert f0 @ f1 == d.interchange(0, 1)
```

Functorial reductions are implemented via the `monoidal.Functor` class, initialised by a pair of mappings: `ob` from objects to types and `ar` from boxes to diagrams. It comes with a `__call__` method that scans through a diagram a replaces each box and identity wire with its image under the mapping.

Listing 1.3.29. Monoidal functor.

```
class Functor(cat.Functor):
    def __init__(self, ob, ar, cod=(Ty, Diagram)):
        super().__init__(ob, ar)

    def __call__(self, diagram):
        if isinstance(diagram, Ty):
            return self.cod[0].tensor(*[self.ob[Ty(x)] for x in diagram])
        if isinstance(diagram, Box):
            return super().__call__(diagram)
        if isinstance(diagram, Diagram):
            scan, result = diagram.dom, self.cod[1].id(self(diagram.dom))
            for box, off in zip(diagram.boxes, diagram.offsets):
                id_l = self.cod[1].id(self(scan[:off]))
                id_r = self.cod[1].id(self(scan[off + len(box.dom):]))
                result = result >> id_l @ self(box) @ id_r
                scan = scan[:off] @ box.cod @ scan[off + len(box.dom):]
            return result
        raise TypeError()
```

We check that the axioms hold.

```
x, y, z = Ty('x'), Ty('y'), Ty('z')
f0, f1, f2 = Box('f0', x, y), Box('f1', y, z), Box('f2', z, x)
F = Functor(ob={x: y, y: z, z: x}, ar={f0: f1, f1: f2, f2: f0})
assert F(f0 >> f1) == F(f0) >> F(f1)
assert F(f0 @ f1) == F(f0) @ F(f1)
assert F(f0 @ f1 >> f1 @ f2) == F(f0) @ F(f1) >> F(f1) @ F(f2)
```

Any `operad.Tree` can be turned into an equivalent `monoidal.Diagram`. We show how this interface is built by overriding the `__call__` method of `operad.Algebra`.

Listing 1.3.30. Interface with `operad.Tree`.

```
from discopy import operad

class Algebra(operad.Algebra):
    def __init__(self, ob, ar, cod=Diagram, contravariant=False):
```

```
        self.contravariant = contravariant
        super().__init__(self, ob, ar, cod=cod)

    def __call__(self, tree):
        if isinstance(tree, operad.Id):
            return self.cod.id(self.ob[tree.dom])
        if isinstance(tree, operad.Box):
            return self.ar[tree]
        box = self.ar[tree.root]
        if isinstance(box, monoidal.Diagram):
            if self.contravariant:
                return box << monoidal.Diagram.tensor(
                    *[self(branch) for branch in tree.branches])
            return box >> monoidal.Diagram.tensor(
                *[self(branch) for branch in tree.branches])
        return box(*[self(branch) for branch in tree.branches])


ob2ty = lambda ob: Ty(ob)
node2box = lambda node: Box(node.name, Ty(node.dom), Ty(*node.cod))
t2d = Algebra(ob2ty, node2box, cod=Diagram)
node2box_c = lambda node: Box(node.name, Ty(*node.cod), Ty(node.dom))
t2d_c = Algebra(ob2ty, node2boxc, cod=Diagram, contravariant=True)

def tree2diagram(tree, contravariant=False):
    if contravariant:
        return t2dc(tree)
    return t2d(tree)
```

1.4 Categorial grammar

The *categorial* tradition of formal grammars originated in the works of Ajdukiewicz [Ajd35] and Bar-Hillel [Bar53], their formalisms are now known as AB grammars [Bus16]. They analyse language syntax by assigning to every word a *type* generated from basic types using two operations: \ (under) and / (over). Strings of types are then parsed according to the following basic reductions:

$$\alpha\,(\alpha\backslash\beta) \to \beta \qquad (\alpha/\beta)\,\beta \to \alpha \tag{1.11}$$

The slash notation α/β, replacing the earlier fraction $\frac{\alpha}{\beta}$, is due to Lambek who unified categorial grammars in an algebraic foundation known as the *Lambek calculus*, first presented in his seminal 1958 paper [Lam58]. With the advent of Chomsky's theories of syntax in the 1960s, categorial grammars were disregarded for almost twenty years [Lam88]. They were revived in the 1980s by several researchers such as Buszowski in Poland, van Benthem and Moortgat in the Netherlands, as witnessed in the 1988 books [OBW88; Moo88].

One reason for this revival is the proximity between categorial grammars and logic. Indeed, the original rewrite rules (1.11) can be seen as versions of modus ponens in a Gentzen style proof system [Lam99b]. Another reason for this revival, is the proximity between categorial grammars, the typed Lambda calculus [Chu40] and the semantic calculi of Curry [Cur61] and Montague [Mon73]. Indeed, one of the best intuitions for categorial grammars comes from interpreting the slash type α/β as the type of a *function* with input of type β and output of type α, and the reduction rules (1.11) as function application. From this perspective, the syntactic process of recognizing a sentence has the same form as the semantic process of understanding it [Ste00]. We will see the implications of this philosophy in 2.2.

Although he did not mention categories in his original paper [Lam58], Lambek had in mind the connections between linguistics and category theory all along, as mentioned in [Lam88]. Indeed the reductions in (1.11) and those introduced by Lambek, correspond to the morphisms of *free biclosed categories*, which admit a neat description as deductions in a Gentzen style proof system. This leads to a fruitful parallel between algebra, proof theory and categorial grammar, summarized in the following table.

Categories	Logic	Linguistics	Python
Biclosed category	Proof system	Categorial grammar	DisCoPy
objects	formulas	types	`biclosed.Ty`
generators	axioms	lexicon	`biclosed.Box`
morphisms	proof trees	reductions	`biclosed.Diagram`

We start by defining biclosed categories and a general notion of biclosed grammar. Then the section will unfold as we unwrap the definition, meeting the three most prominent variants of categorial grammars: AB grammars [Bus16], the Lambek calculus [Lam58] and Combinatory Categorial Grammars [Ste00]. We end by giving an implementation of free biclosed categories as a class `biclosed.Diagram` with methods for currying and uncurrying.

1.4.1 Biclosed categories

Definition 1.4.1 (Biclosed signature)**.** *A biclosed signature G is a collection of generators G_1 and basic types G_0 together with a pair of maps:*

$$T(G_0) \xleftarrow{\texttt{dom}} G_1 \xrightarrow{\texttt{cod}} T(G_0)$$

where $T(G_0)$ is the set of biclosed types, given by the following inductive definition:

$$T(B) \ni \alpha = a \in B \mid \alpha \otimes \alpha \mid \alpha\backslash\alpha \mid \alpha/\alpha \tag{1.12}$$

A morphism of biclosed signatures $\varphi : G \to \Gamma$ is a pair of maps $\varphi_1 : G_1 \to \Gamma_1$ and $\varphi_0 : G_0 \to \Gamma_0$ such that the diagram with the signature morphisms commutes. The category of biclosed signatures is denoted **BcSig**

Definition 1.4.2 (Biclosed category)**.** *A biclosed monoidal category $\mathbf{C}$ is a monoidal category equipped with two bifunctors $-\backslash- : \mathbf{C}^{op} \times \mathbf{C} \to \mathbf{C}$ and $-/- : \mathbf{C} \times \mathbf{C}^{op} \to \mathbf{C}$ such that for any object a, $a \otimes - \dashv a\backslash-$ and $- \otimes a \dashv -/a$. Explicitly, we have the following isomorphisms natural in $a, b, c \in \mathbf{C}_0$:*

$$\mathbf{C}(a, c/b) \simeq \mathbf{C}(a \otimes b, c) \simeq \mathbf{C}(b, a\backslash c) \tag{1.13}$$

These isomorphisms are often called currying *(when $\otimes$ is replaced by $\backslash$ or $/$) and* uncurrying*. With monoidal functors as morphisms, biclosed categories form a category denoted* **BcCat**.

Morphisms in free biclosed category can be described as the valid deductions in a proof system defined as follows. The axioms of monoidal categories may be expressed as the following rules of inference:

$$\frac{}{a \to a}\ \texttt{id} \qquad \frac{a \to b \qquad b \to c}{a \to c}\ \circ \qquad \frac{a \to b \qquad c \to d}{a \otimes c \to b \otimes d}\ \otimes \tag{1.14}$$

Additionally, the defining adjunctions of biclosed categories may be expressed as follows:

$$a \otimes b \to c \quad \text{iff} \quad a \to c/b \quad \text{iff} \quad b \to a\backslash c \tag{1.15}$$

Given a biclosed signature G, the free biclosed category $\mathbf{BC}(G)$ contains all the morphisms that can be derived using the inference rules 1.15 and 1.14 from the signature seen as a set of axioms for the deductive system. $\mathbf{BC}$ is part of an adjunction relating biclosed signatures and biclosed categories:

$$\mathbf{BcSig} \underset{U}{\overset{\mathbf{BC}}{\rightleftarrows}} \mathbf{BcCat}$$

We can now define a general notion of biclosed grammar, the equivalent of monoidal grammars in a biclosed setting.

Definition 1.4.3. *A biclosed grammar G is a biclosed signature of the following shape:*

$$T(B+V) \xleftarrow{\texttt{dom}} G \xrightarrow{\texttt{cod}} T(B+V)$$

where V is a vocabulary *and B is a set of* basic types*. The language generated by a G is given by:*

$$\mathcal{L}(G) = \{\, u \in V^* \mid \exists g : u \to s \in \mathbf{BC}(G) \,\}$$

where $\mathbf{BC}(G)$ is the free biclosed category generated by G. We say that G is lexicalised *when it has the following shape:*

$$V \xleftarrow{\texttt{dom}} G \xrightarrow{\texttt{cod}} T(B)$$

As we will see, AB grammars, Lambek grammars and Combinatory Categorial grammars all reduce functorially to biclosed grammars. However, biclosed grammars can have an infinite number of rules of inference, obtained by iterating over the isomorphism (1.15). Interestingly, these rules have been discovered progressively throughout the history of categorial grammars.

1.4.2 Ajdiuciewicz

We discuss the classical categorial grammars of Ajdiuciewicz and Bar-Hillel, known as AB grammars [Bus16]. The types of the original AB grammars are given by the following inductive definition:

$$T_{AB}(B) \ni \alpha = a \in B \mid \alpha\backslash\alpha \mid \alpha/\alpha \;.$$

Given a vocabulary V, the *lexicon* is usually defined as a relation $\Delta \subseteq V \times T_{AB}(B)$ assigning a set of candidate types $\Delta(w)$ to each word $w \in V$. Given an utterance $u = w_0 \dots w_n \in V^*$, one can prove that u is a grammatical sentence by producing a reduction $t_0 \dots t_n \to s$ for some $t_i \in \Delta(w_i)$, generated by the basic reductions in (1.11).

Definition 1.4.4 (AB grammar)**.** *An AB grammar is a tuple $G = (V, B, \Delta, s)$ where V is a vocabulary, B is a finite set of basic types, and $\Delta \subseteq V \times T_{AB}(B)$ is a finite relation, called the* lexicon*. The* rules *of AB grammars are given by the following monoidal signature:*

$$R_{AB} = \left\{ \begin{array}{ccc} a & a\backslash b & \qquad\qquad b/a \quad a \\ \vert & \vert & \qquad\qquad \vert \quad \vert \\ \boxed{\texttt{app}} & , & \qquad\qquad \boxed{\texttt{app}} \\ \vert & & \qquad\qquad \vert \\ b & & \qquad\qquad b \end{array} \right\}_{a,b \in T_{AB}(B)} .$$

Then the language generated by G is given by:

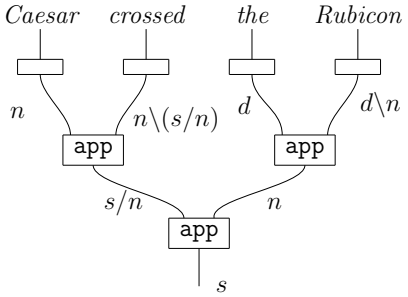
$$\mathcal{L}(G) = \{\, u \in V^* \,:\, \exists g \in \mathbf{MC}(\Delta + R_{AB})(u, s) \,\}$$

Remark 1.4.5. *Sometimes, categorial grammarians use a different notation where $a\backslash b$ is used in place of $b\backslash a$. We find the notation used here more intuitive: we write $a \otimes a\backslash b \to b$ instead of $a \otimes b\backslash a \to b$. Ours is in fact the notation used in the original paper by Lambek [Lam58].*

Example 1.4.6 (Application)**.** *As an example take the vocabulary $V = \{Caesar, crossed, the, Rubicon\}$ and basic types $B = \{s, n, d\}$ for sentence, noun and determinant types. We define the lexicon Δ by:*

$$\Delta(Caesar) = \{\, n \,\} \quad \Delta(crossed) = \{\, n\backslash(s/n) \,\} \quad \Delta(the) = \{\, d \,\} \quad \Delta(Rubicon) = \{\, d\backslash n \,\}$$

Then the sentence "John wrote a dictionnary" is grammatical as witnessed by the following reduction:



Proposition 1.4.7. *AB grammars reduce functorially to biclosed grammars.*

Proof. It is sufficient to show that the rules R_{AB} can be derived from the axioms of biclosed categories, i.e. that they are morphisms in any free biclosed category. Let a, b be objects in a free biclosed category. We derive the forward application rule as follows.

$$\dfrac{\dfrac{}{a/b \to a/b}\,\texttt{id}}{a/b \otimes b \to a}\,1.15$$

Similarly, one may derive backward application $\texttt{app}^{<} : b \otimes b\backslash a \to b$. □

AB grammars are weakly equivalent to context-free grammars [Ret05] as shown by the following pair of propositions.

Proposition 1.4.8. *For any AB grammar there is a functorially equivalent context-free grammar in Chomsky normal form.*

Proof. The only difficulty in this proof comes from the fact that R_{AB} is an infinite set, whereas context-free grammars must be defined over a finite set of symbols and production rules. Given an AB grammar $G = (V, B, \Delta)$, define $X = \{x \in T_{AB}(B) | \exists (w, t) \in \Delta \text{such that} x \subseteq t\}$ where we write $x \subseteq t$ to say that x is a sub-type of t. Note that X is finite. Now let $P = \{r \in R_{AB} | \texttt{dom}(r) \in X\}$. Note that also P is a finite set. Then $X \leftarrow P + \Delta \to (X + V)^*$ forms a lexicalised context-free grammar where each production rule has at most arity 2, i.e. $P + \Delta$ is in Chomsky normal form. There is a functorial reduction $\mathbf{MC}(P + \Delta) \to \mathbf{MC}(R_{AB} + \Delta)$ induced by the injection $P \hookrightarrow R_{AB}$, also there is a functorial reduction $\mathbf{MC}(R_{AB}+\Delta) \to \mathbf{MC}(P+\Delta)$ which sends all the types in $T_{AB}(B)\backslash X$ to the unit and acts as the identity on the rest. Therefore G is functorially equivalent to $P + \Delta$. □

Proposition 1.4.9. *Any context-free grammar in Greibach normal form reduces functorially to an AB grammar.*

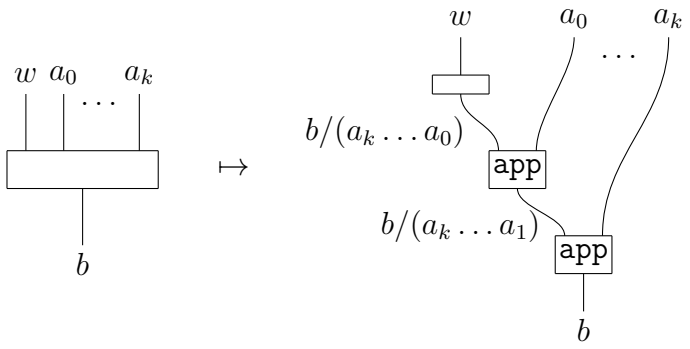
Proof. Recall that a CFG is in Greibach normal form when it has the shape:

$$B \leftarrow G \rightarrow V \times B^*$$

We can rewrite this as a signature of the following shape:

$$V \leftarrow G \rightarrow B \times B^*$$

This yields a relation $G \subseteq V \times (B \times B^*)$. We define the lexicon $\Delta \subseteq V \times T(B)$ by $\Delta(w) = G(w)_0 / \mathtt{inv}(G(w)_1)$ where $\mathtt{inv}$ inverts the order of the basic types. Then there is a functorial reduction $\mathbf{MC}(G) \rightarrow \mathbf{MC}(\Delta + R_{AB})$ given on any production rule $wa_0 \dots a_k \rightarrow b$ in G by:



□

1.4.3 Lambek

In his seminal paper [Lam58], Lambek introduced two new types of rules to the original AB grammars: *composition* and *type-raising*. Several extensions have been considered since, including non-associative [MR12c] and multimodal [MR12b] versions. Here, we focus on the original (associative) Lambek calculus which is easier to cast in a biclosed setting and already captures a large fragment of natural language.

The types of the Lambek calculus are precisely the biclosed types given in 1.17. This is not a coincidence since Lambek was aware of biclosed categories when he introduced his calculus [Lam88].

Definition 1.4.10 (Lambek grammar)**.** *A Lambek grammar is a tuple $G = (V, B, \Delta, s)$ where V is a vocabulary, B is a finite set of basic types and $\Delta \subseteq V \times T(B)$ is a* lexicon*. The rules of Lambek grammars are given by the following monoidal signature:*

$$R_{LG} = \left\{ \begin{array}{ccccccc} a\ \ a\backslash b & & b/a\ \ a & & a\backslash b\ \ b\backslash c & & a/b\ \ b/c \\ \boxed{\mathtt{app}} & , & \boxed{\mathtt{app}} & , & \boxed{\mathtt{comp}} & , & \boxed{\mathtt{comp}} \\ b & & b & & a\backslash c & & a/c \end{array} \right. \quad \left. \begin{array}{ccc} a & , & a \\ \boxed{\mathtt{tyr}} & & \boxed{\mathtt{tyr}} \\ b/(a\backslash b) & & (b/a)\backslash b \end{array} \right\}_{a,b,c \in T(B)} .$$

Then the language generated by G is given by:

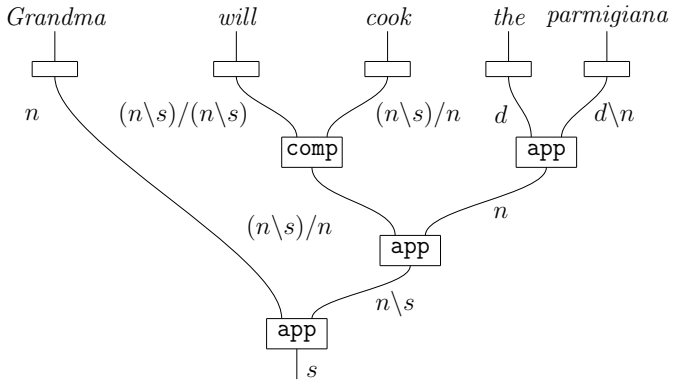
$$\mathcal{L}(G) = \{\, u \in V^* \,:\, \exists g \in \mathbf{MC}(\Delta + R_{LG})(u, s) \,\}$$

Example 1.4.11 (Composition)**.** *We adapt an example from [Ste87]. Consider the following lexicon:*

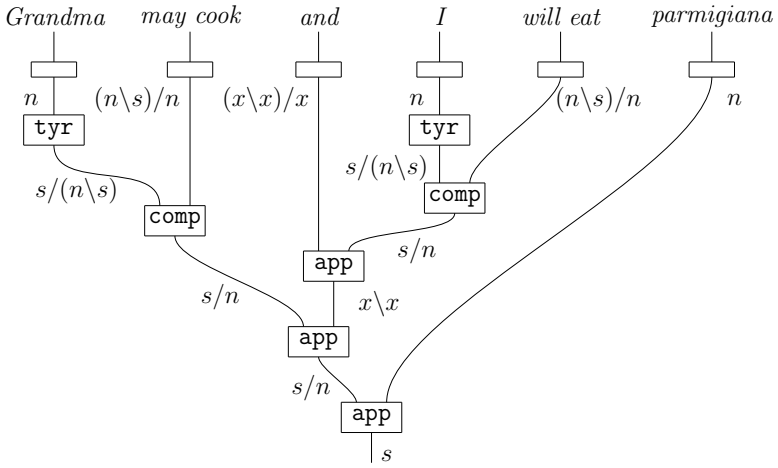
$$\Delta(I) = \Delta(Grandma) = \{\, n \,\} \quad \Delta(the) = \{\, d \,\} \quad \Delta(parmigiana) = \{\, d\backslash n \,\}$$

$$\Delta(will) = \Delta(may) = \{\, (n\backslash s)/(n\backslash s) \,\} \quad \Delta(eat) = \Delta(cook) = \{\, n\backslash s/n \,\}$$

The following is a grammatical sentence, parsed using the composition rule:



And the following sentence requires the use of type-raising:



where $x = s/n$ and we have omitted the composition of modalities (will, may) with their verbs (cook, eat).

Proposition 1.4.12. *Lambek grammars reduce functorially to biclosed grammars.*

Proof. It is sufficient to show that the rules of Lambek grammars R_{LG} can be derived from the axioms of biclosed categories. We already derived forward and backward application in 1.4.7. The following proof tree shows that the forward composition rule follows from the axioms of biclosed categories.

$$\dfrac{\dfrac{\dfrac{}{a/b \to a/b}\,\mathtt{id} \quad \dfrac{\dfrac{}{b/c \to b/c}\,\mathtt{id}}{b/c \otimes c \to b}\,1.15}{a/b \otimes b/c \otimes c \to a}\,\otimes \text{ and } \circ\mathtt{app}}{a/b \otimes b/c \to a/c}\,1.15$$

A similar argument derives the backward composition rule $\texttt{comp}^< : a/b \otimes a\backslash c \to c/b$. Forward type-raising is derived as follows.

$$\dfrac{\dfrac{\dfrac{}{a\backslash b \to a\backslash b}\,\texttt{id}}{a \otimes a\backslash b \to b}\,1.15}{a \to b/(a\backslash b)}\,1.15$$

And a similar argument derives backward type-raising $\texttt{tyr}^< : a \to (b/a)\backslash b$. □

It was conjectured by Chomsky that any Lambek grammar is weakly equivalent to a context-free grammar, i.e. that the language recognised by Lambek calculi is context-free. This conjecture was proved in 1993 by Pentus [Pen93], calling for a more expressive version of categorial grammars.

1.4.4 Combinatory

In the 1980s, researchers interested in the syntax of natural language started recognizing that certain grammatical sentences naturally involve crossed dependencies between words, a phenomenon that cannot be captured by context-free grammars. These are somewhat rare in English, but they abound in Dutch for instance [Bre+82]. An English example is the sentence "I explained to John maths" which is often allowed to mean "I explained maths to John". Modeling cross-serial dependencies was one of the main motivations for the development the Tree-adjoining grammars of Joshi [Jos85] and the Combinatory Categorial grammars (CCGs) of Steedman [Ste87; Ste00]. These were later shown to be weakly equivalent to linear indexed grammars [VW94], making them all "mildly-context sensitive" according to the definition of Joshi [Jos85].

CCGs extend the earlier categorial grammars by adding a *crossed composition* rule which allows for controlled crossed dependencies within a sentence, and is given by the following pair of reductions:

$$\texttt{xcomp}^> : a/b \otimes c\backslash b \to c\backslash a \qquad \texttt{xcomp}^< : a/b \otimes a\backslash c \to c/b$$

We start by defining CCGs as monoidal grammars, and then discuss how they relate to biclosed categories.

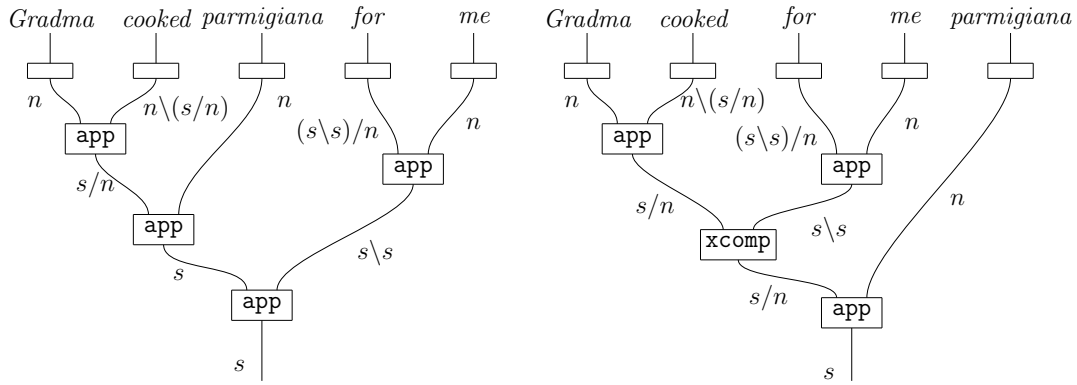
Definition 1.4.13 (Combinatory Categorial Grammar)**.** *A CCG is a tuple* $G = (V, B, \Delta, s)$ *where* V *is a vocabulary,* B *is a finite set of basic types and* $\Delta \subseteq V \times T(B)$ *is a* lexicon*. The* rules *of CCGs are given by the following monoidal signature:*

$$R_{CCG} = R_{LG} + \left\{ \begin{array}{c} a/b \;\; c\backslash b \\ \boxed{\texttt{xcomp}} \\ c\backslash a \end{array} \;,\; \begin{array}{c} a/b \;\; a\backslash c \\ \boxed{\texttt{xcomp}} \\ c/b \end{array} \right\}_{a,b,c \in T(B)} .$$

Then the language generated by G *is given by:*

$$\mathcal{L}(G) = \{\, u \in V^* \,:\, \exists g \in \mathbf{MC}(\Delta + R_{CCG})(u, s) \,\}$$

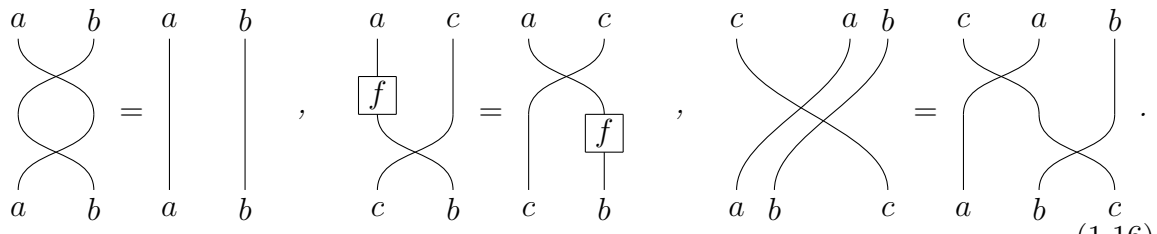
Example 1.4.14 (Crossed composition)**.** *Taking the grammar from Example and adding two lexical entries we may derive the following grammatical sentences:*



Note that the first one can be parsed already in an AB grammar, whereas the second requires the crossed composition rule.

It was first shown by Moortgat [Moo88] that the crossed composition rule cannot be derived from the axioms of the Lambek calculus. In our context, this implies that we cannot derive `xcomp` from the axioms of biclosed categories. Of course, CCGs may be seen as biclosed grammars by adding the crossed composition rules $R_{CCG} - R_{LG}$ as generators in the signature. However, it is interesting to note that these rules can be derived from *closed categories*, the *symmetric* version of biclosed categories:.

Definition 1.4.15 (Symmetric monoidal category)**.** *A symmetric monoidal category* $\mathbf{C}$ *is a monoidal category equipped with a natural transformation* $\texttt{swap} : a \otimes b \to b \otimes a$ *satisfying:*



(1.16)

for any $a, b, c \in \mathbf{C}_0$ *and* $f : a \to b \in \mathbf{C}_1$.

Definition 1.4.16 (Closed category)**.** *A closed category is a symmetric biclosed category.*

Proposition 1.4.17. *The crossed composition rule follows from the axioms of closed categories.*

Proof. Let a, b, c be objects in the free closed category with no generators. The following proof tree shows that the backward crossed composition rule follows from the axioms of closed categories.

$$\dfrac{\dfrac{\dfrac{}{a/b \to a/b}\,\texttt{id} \quad \dfrac{\dfrac{}{c\backslash b \to c\backslash b}\,\texttt{id}}{c \otimes c\backslash b \to b}\,1.15}{a/b \otimes c \otimes c\backslash b \to a}\,\otimes \text{ and } \circ\texttt{app}}{a/b \otimes c\backslash b \otimes c \to a}\,\circ\texttt{swap}}{a/b \otimes c\backslash b \to a\backslash c}\,1.15$$

Note the crucial use of pre-composition with `swap` in the penultimate inference. A similar argument derives the forward crossed composition rule $a/b \otimes a\backslash c \to c/b$. □

One needs to be very careful in adding permutations and symmetry in a formal grammar. The risk is to lose any notion of word order. It was initially believed that adding the crossed composition rule, coupled with type-rasing, would collapse the calculus to permutation invariance [Wil03]. However, as argued by Steedman this is not the case: out of the $n!$ possible permutations of a string of n types, CCGs only allow $S(n)$ permutations where $S(n)$ is the nth Large Schroder number [Ste19]. Thus the crossed composition rule only permits limited crossed dependencies within a sentence. This claim is also verified empirically by the successful performance of large scale CCG parsers [Yos+19].

If we consider the complexity of parsing categorial grammars, it turns out that parsing the Lambek calculus is `NP`-complete [Sav12]. Even when the order of types is bounded the best parsing algorithms for the Lambek calculus run in $O(n^5)$ [Fow08]. Parsing CCGs is in `P` with respect to the size of the input sentence, but it becomes exponential when also the grammar is taken in the input [KSJ18]. This is in contrast to CFGs or the mildly context-sensitive tree-adjoining grammars where parsing is computable in $O(n^3)$ in both the size of input grammar and input sentence. This called for a simpler version of the Lambek calculus, with a more efficient parsing procedure, but still keeping the advantages of the categorial lexicalised formalism and the proximity to semantics.

1.4.5 biclosed.Diagram

We wish to upgrade the `monoidal.Diagram` class to represent diagrams in free biclosed categories. In order to achieve this, we first need to upgrade the `Ty` class, to handle the types $T(B)$ of free biclosed categories. A `biclosed.Ty` is an instance of `monoidal.Ty` with extra methods for under $\backslash$ and over $/$ operations. Recall the context-free grammar of biclosed types:

$$T(B) \ni \alpha = a \in B \mid \alpha \otimes \alpha \mid \alpha\backslash\alpha \mid \alpha/\alpha \tag{1.17}$$

Elements of $T(B)$ are trees built from leaves B and binary nodes $\otimes$, $\backslash$ and $/$. It may be implemented in object-oriented programming, by subclassing `monoidal.Ty` for the tensor and using the native tree structure of classes. We give a working version of `biclosed.Ty`, detailing also some of the standard methods.

Listing 1.4.18. Types in free biclosed categories

```
from discopy import monoidal
```

```
class Ty(monoidal.Ty):
    def __init__(self, *objects, left=None, right=None):
        self.left, self.right = left, right
        super().__init__(*objects)

    def __lshift__(self, other):
        return Over(self, other)

    def __rshift__(self, other):
        return Under(self, other)

    def __matmul__(self, other):
        return Ty(*(self @ other))

    @staticmethod
    def upgrade(old):
        if len(old) == 1 and isinstance(old[0], (Over, Under)):
            return old[0]
        return Ty(*old.objects)


class Over(Ty):
    def __init__(self, left=None, right=None):
        Ty.__init__(self, self)

    def __repr__(self):
        return "Over({}, {})".format(repr(self.left), repr(self.right))

    def __str__(self):
        return "({} << {})".format(self.left, self.right)

    def __eq__(self, other):
        if not isinstance(other, Over):
            return False
        return self.left == other.left and self.right == other.right


class Under(Ty):
    def __init__(self, left=None, right=None):
        Ty.__init__(self, self)

    def __repr__(self):
        return "Under({}, {})".format(repr(self.left), repr(self.right))

    def __str__(self):
        return "({} >> {})".format(self.left, self.right)

    def __eq__(self, other):
        if not isinstance(other, Under):
            return False
        return self.left == other.left and self.right == other.right
```

The `Ty.upgrade` *method allows for compatibility between the tensor methods* `__matmul__` *of* `biclosed` *and* `monoidal` *types. The upgrading mechanism is further desribed in*

the documentation of `monoidal`.

We illustrate the syntax of biclosed types.

```
x = Ty('x')
assert x >> x << x == Over(Under(Ty('x'), Ty('x')), Ty('x'))
assert x >> (x << x) == Under(Ty('x'), Over(Ty('x'), Ty('x')))
x0, x1, y0, y1, m = Ty('x0'), Ty('x1'), Ty('y0'), Ty('y1'), Ty('m')
lens = (x0 >> m @ y0) @ ( m @ x1 >> y1)
assert lens == Ty(Under(Ty('x0'), Ty('m', 'y0')), Under(Ty('m', 'x1'), Ty('y1')))
```

A `biclosed.Diagram` is a `monoidal.Diagram` with domain and codomain `biclosed.Ty`s, together with a pair of static methods `curry()` and `uncurry()` implementing the defining isomorphism of biclosed categories (1.15). In fact, we can store the information of how a biclosed diagram is constructed by using two special subclasses of `Box`, which record every application of the currying morphisms. Thus a diagram in a biclosed category is a tree built from generating boxes using `id`, `then`, `tensor`, `Curry` and `UnCurry`.

Listing 1.4.19. Diagrams in free biclosed categories

```
from discopy import monoidal

@monoidal.Diagram.subclass
class Diagram(monoidal.Diagram):

    def curry(self, n_wires=1, left=False):
        return Curry(self, n_wires, left)

    def uncurry(self):
        return UnCurry(self)

class Id(monoidal.Id, Diagram):


class Box(monoidal.Box, Diagram):


class Curry(Box):
    def __init__(self, diagram, n_wires=1, left=False):
        name = "Curry({})".format(diagram)
        if left:
            dom = diagram.dom[n_wires:]
            cod = diagram.dom[:n_wires] >> diagram.cod
        else:
            dom = diagram.dom[:-n_wires]
            cod = diagram.cod << diagram.dom[-n_wires or len(diagram.dom):]
        self.diagram, self.n_wires, self.left = diagram, n_wires, left
        super().__init__(name, dom, cod)


class UnCurry(Box):
    def __init__(self, diagram):
        name = "UnCurry({})".format(diagram)
```

```
        self.diagram = diagram
        if isinstance(diagram.cod, Over):
            dom = diagram.dom @ diagram.cod.right
            cod = diagram.dom.left
            super().__init__(name, dom, cod)
        elif isinstance(diagram.cod, Under):
            dom = diagram.dom.left @ diagram.dom
            cod = diagram.dom.right
            super().__init__(name, dom, cod)
        else:
            super().__init__(name, diagram.dom, diagram.cod)
```

We give a simple implementation of free biclosed categories which does not impose the axioms of free biclosed categories, `UnCurry(Curry(f)) == f` *and naturality. IT can be implemented on syntax by adding* `if` *statement in the inits, and upgrading* `Curry` *and* `UnCurry` *to subclasses of* `biclosed.Diagram`*..*

Finally, a `biclosed.Functor` is a `monoidal.Functor` whose call method has a predefined mapping for all structural boxes in `biclosed`. It thus allows to map any `biclosed.Diagram` into a codomain class `cod` equipped with `curry` and `uncurry` methods.

Listing 1.4.20. Functors from free biclosed categories

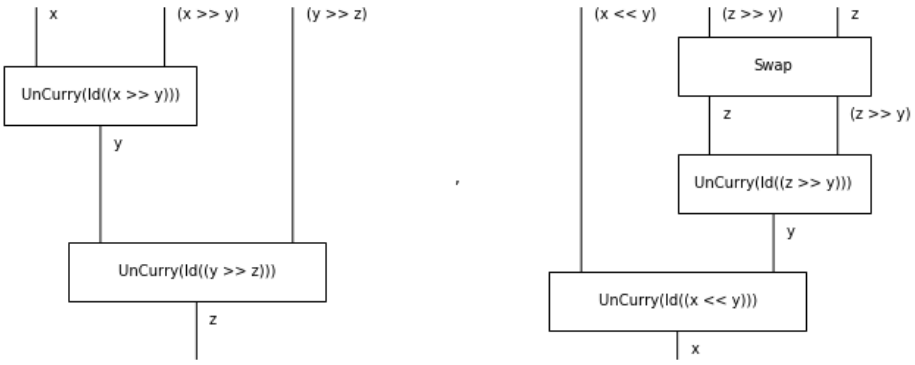
```
class Functor(monoidal.Functor):
    def __init__(self, ob, ar, cod=(Ty, Diagram)):
        self.cod = cod
        super().__init__(ob, ar, ob_factory=cod[0], ar_factory=cod[1])

    def __call__(self, diagram):
        if isinstance(diagram, Over):
            return self(diagram.left) << self(diagram.right)
        if isinstance(diagram, Under):
            return self(diagram.left) >> self(diagram.right)
        if isinstance(diagram, Ty) and len(diagram) > 1:
            return self.cod[0].tensor(*[
                self(diagram[i: i + 1]) for i in range(len(diagram))])
        if isinstance(diagram, Id):
            return self.cod[1].id(self(diagram.dom))
        if isinstance(diagram, Curry):
            n_wires = len(self(getattr(
                diagram.cod, 'left' if diagram.left else 'right')))
            return self.cod[1].curry(
                self(diagram.diagram), n_wires, diagram.left)
        if isinstance(diagram, UnCurry):
            return self.cod[1].uncurry(self(diagram.diagram))
        return super().__call__(diagram)
```

We recover the rules of categorial grammars (in historical order) by constructing them from identities in the free biclosed category with no generators.

Listing 1.4.21. Categorial grammars and the free biclosed category

```
# Adjiuciewicz
FA = lambda a, b: UnCurry(Id(a >> b))
assert FA(x, y).dom == x @ (x >> y) and FA(x, y).cod == y
BA = lambda a, b: UnCurry(Id(b << a))
assert BA(x, y).dom == (y << x) @ x and BA(x, y).cod == y

# Lambek
proofFC = FA(x, y) @ Id(y >> z) >> FA(y, z)
FC = Curry(proofFC, left=True)
assert FC.dom == (x >> y) @ (y >> z) and FC.cod == (x >> z)
BC = Curry(Id(x << y) @ BA(z, y) >> BA(y, x))
assert BC.dom == (x << y) @ (y << z) and BC.cod == (x << z)
TYR = Curry(UnCurry(Id(x >> y)))
assert TYR.dom == x and TYR.cod == (y << (x >> y))

# Steedman
Swap = lambda a, b: Box('Swap', a @ b, b @ a)
proofBX = Id(x << y) @ (Swap(z >> y, z) >> FA(z, y)) >> BA(y, x)
BX = Curry(proofBX)
assert BX.dom == (x << y) @ (z >> y) and BX.cod == (x << z)
proofFX = (Swap(x, y << x) >> BA(x, y)) @ Id(y >> z) >> FA(y, z)
FX = Curry(proofFX, left=True)
assert FX.dom == (y << x) @ (y >> z) and FX.cod == (x >> z)
```

The assertions above are alternative proofs of Propositions 1.4.7, 1.4.12 and 1.4.17. We draw the proofs for forward composition (`proofFC`) and backwards crossed composition (`proofBX`).



1.5 Pregroups and dependencies

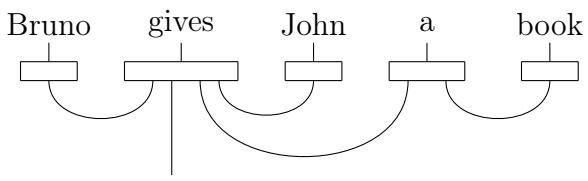
In his 1897 paper "The logic of relatives" [Pei97], Peirce makes an analogy between the sentence "John gives John to John" and the molecule of ammonia.

The intuition that words within a sentence are connected by "bonds", as atoms in a molecule, is the basis of Peirce's diagrammatic approach to logical reasoning, and of his analysis of the concept of *valency* in grammar [Ask19]. This intuition resurfaced in the work of Lucien Tesniére [Tes59] who analysed the valency of a large number of lexical items, in an approach to syntax that became known as *dependency grammar* [Hay64; Gai65]. These bear a striking resemblance to Lambek's pregroup grammars [Lam08] and its developments in the DisCoCat framework of Coecke, Sadrzadeh et al. [CCS08; SCC13; SCC14].

In this section, we formalise both Pregroup Grammar (PG) and Dependency Grammar (DG) in the language of *free rigid categories*. Once casted in this algebraic framework, the similarities between PG and DG become apparent. We show that dependency grammars are structurally equivalent to both pregroup grammar and context-free grammar, i.e. their derivations are tree-shaped rigid diagrams. We end by describing the implementation of the class `rigid.Diagram` and we interface it with SpaCy's dependency parser [Hon+20].

1.5.1 Pregroups and rigid categories

Pregroup grammars were introduced by Lambek in 1999 [Lam99a]. Arising from a non-commutative fragment of Girard's linear logic [CL02] they refine and simplify the earlier Lambek calculus discussed in Section 1.4.3. As shown by Buszowski, pregroup grammars are weakly equivalent to context-free grammars [BM07]. However, the syntactic structures generated by pregroup grammars differ from those of a CFG. Instead of trees, pregroups parse sentences by assigning to them a nested pattern of *cups* or "links" as in the following example.



In the strict sense of the word, a pregroup is a preordered monoid where each object has a left and a right adjoint [Lam99a]. We formalise pregroup grammars in terms of *rigid categories* which categorify the notion of pregroup by upgrading the preorder to a category. Going from inequalities in a preordered monoid to arrows in a monoidal category allows both to reason about syntactic ambiguity — as discussed by Lambek

and Preller [PL07] — as well as to define pregroup semantics as a monoidal functor, an observation which lead to the development of the compositional distributional models of Coecke et al. [CCS10] and which will form the basis of the next chapter on semantics.

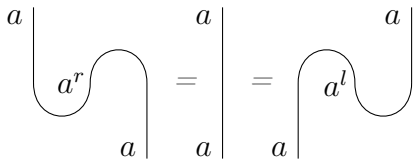
Given a set of basic types B, the set of pregroup types $P(B)$ is defined as follows.

$$P(B) \ni t ::= b \in B \,|\, t^r \,|\, t^l \,|\, t \otimes t.$$

we can use it to define the notion of a rigid signature.

Definition 1.5.1 (Rigid signature)**.** *A rigid signature is a graph $\Sigma = \Sigma_1 \rightrightarrows P(\Sigma_0)$. A morphism of rigid signatures $\Sigma \to \Gamma$ is a pair of maps $\Sigma_1 \to \Gamma_1$ and $\Sigma_0 \to \Gamma_0$ satisfying the obvious commuting diagram. Rigid signatures and their morphisms form a category denoted* **RgSig**.

Definition 1.5.2 (Rigid category)**.** *A rigid category* **C** *is a monoidal category such that each object a has a left adjoint a^l and a right adjoint a^r. In other words there are morphisms $a^l \otimes a \to 1$, $1 \to a \otimes a^l$, $a \otimes a^r \to 1$ and $1 \to a^r \otimes a^l$, denoted as cups and caps and satisfying the snake equations:*



(1.18)

The category of rigid categories and monoidal functors is denoted **RigidCat**.

Proposition 1.5.3. *Rigid categories are biclosed, with $a\backslash b = a^r \otimes b$ and $b/a = b \otimes a^l$.*

Given a rigid signature Σ we can generate the *free rigid category*

$$\mathbf{RC}(\Sigma) = \mathbf{MC}(\Sigma + \{\, cups, caps \,\})/\sim_{snake}$$

where $\sim_{snake}$ is the equivalence relation on diagrams induced by the snake equations (1.18). Rigid categories are called *compact 2-categories* with one object by Lambek and Preller [PL07], who showed that **RC** defines an adjunction between rigid signatures and rigid categories.

$$\mathbf{RgSig} \underset{U}{\overset{\mathbf{RC}}{\rightleftarrows}} \mathbf{RgCat}$$

We start by defining a general notion of *rigid grammar*, a subclass of biclosed grammars.

Definition 1.5.4. *A rigid grammar G is a rigid signature of the following shape:*

$$P(B + V) \xleftarrow{\texttt{dom}} G \xrightarrow{\texttt{cod}} P(B + V)$$

where V is a vocabulary *and B is a set of* basic types*. The language generated by G is given by:*

$$\mathcal{L}(G) = \{\, u \in V^* \,|\, \exists g : u \to s \in \mathbf{RC}(G) \,\}$$

where $\mathbf{RC}(G)$ is the free rigid category generated by G.

A pregroup grammar is a lexicalised rigid grammar defined as follows.

Definition 1.5.5 (Pregroup grammar)**.** *A pregroup grammar is a tuple* $G = (V, B, \Delta, I, s)$ *where* V *is a* vocabulary*,* B *is a finite set of basic types,* $G \subseteq V \times P(B)$ *is a* lexicon *assigning a set of possible pregroup types to each word, and* $I \subseteq B \times B$ *is a finite set of* induced steps*. The language generated by* G *is given by:*

$$\mathcal{L}(G) = \{\, u \in V^* \,:\, \exists g \in \mathbf{RC}(G)(u, s) \,\}$$

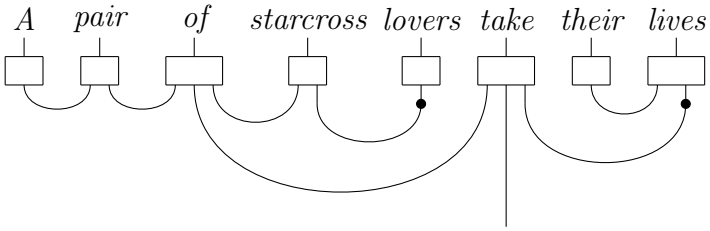
where $\mathbf{RC}(G) := \mathbf{RC}(\Delta + I)$*.*

Example 1.5.6. *Fix the basic types* $B = \{\, s, n, n_1, d, d_1 \,\}$ *for sentence, noun, plural noun, determinant and plural determinant and consider the following pregroup lexicon:*

$$\Delta(\mathit{pair}) = \{\, d^r\, n \,\},\ \Delta(\mathit{lives}) = \{\, d_1^r\, n_1 \,\},\ \Delta(\mathit{lovers}) = \{\, n_1 \,\},\ \Delta(\mathit{starcross}) = \{\, n\, n^l \,\},$$

$$\Delta(\mathit{take}) = \{\, n^r\, s\, n^l \,\},\quad \Delta(\mathit{of}) = \{\, n^r\, n\, n^l \,\},\quad \Delta(A) = \{\, d \,\},\quad \Delta(\mathit{their}) = \{\, d_1 \,\}.$$

and one induced step $I = \{\, n_1 \to n \,\}$*. Then the following is a grammatical sentence:*

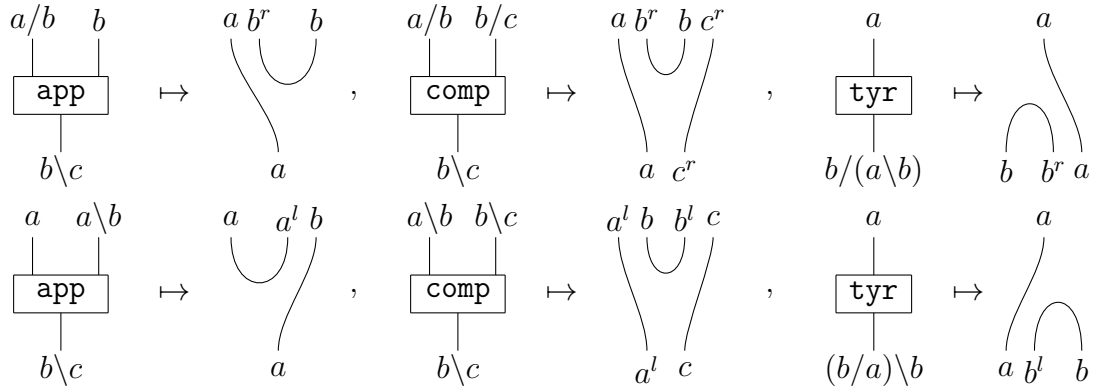


where we omitted the types for readability, and we denoted the induced step using a black node.

The tight connections between categorial grammars and pregroups were discussed in [Bus07], they are evermore apparent from a categorical perspective: since rigid categories are biclosed, there is a canonical way of mapping the reductions of a categorial grammar to pregroups.

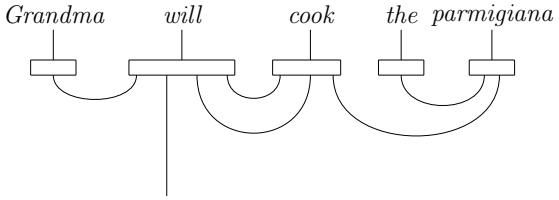
Proposition 1.5.7. *For any Lambek grammar* G *there is a pregroup grammar* G' *with a functorial reduction* $\mathbf{MC}(G) \to \mathbf{RC}(G')$*.*

Proof. The translation works as follows:



□

Example 1.5.8. *Consider the categorial parsing of the sentence "Grandma will cook the parmigiana" from Example 1.4.14. The categorial type of "will" is given by* $(n\backslash s)/(n\backslash s)$ *which translates to the pregroup type* $(n^r s)(n^r s)^l = n^r s\, s^l n$*, the transitive verb type* $(n\backslash s)/n$ *for "cook" translates to* $n^r\, s\, n^l$*, and similarly for the other types. Translating the categorial reduction according to the mapping above, we obtain the following pregroup reduction:*



One advantage of pregroups over categorial grammars is that they can be parsed more efficiently. This is a consequence of the following lemma, proved by Lambek in [Lam99a].

Proposition 1.5.9 (Switching lemma)**.** *For any pregroup grammar* G *and any reduction* $t \to s$ *in* $\mathbf{RC}(G)$*, there is a type* t' *such that* $t \to s = t \to t' \to s$ *and* $t \to t'$ *doesn't use contractions (cups),* $t' \to s$ *doesn't use expansions (caps).*

Remark 1.5.10. *Note that the equivalent lemma for categorial grammars would state that all instances of the type-raising rule can appear after all instances of the composition and application rules. This is however not the case.*

A direct corollary of this lemma, is that any sentence $u \in V^*$ may be parsed using only contractions (cups). This drastically reduces the search space for a reduction, with the consequence that pregroup grammars can be parsed efficiently.

Proposition 1.5.11. *Pregroup grammars can be parsed in polynomial time [DP05; Mor11] and in linear-time in linguistically justified restricted cases [Pre07].*

As discussed in Section 1.4.4, linguists have observed that certain grammatical sentences naturally involve crossed dependencies between words [Sta04]. Although the planarity assumption is justified in several cases of interest [Can06], grammars with crossed dependencies allow for more flexibility when parsing natural languages. In order to model these phenomena with pregroup grammars, we need to step out of (planar) rigid categories and allow for (restricted) permutations. These can be represented in the symmetric version of rigid categories, known as *compact closed* categories.

Definition 1.5.12 (Compact-closed)**.** *A compact-closed category is a rigid category (1.18) that is also symmetric (1.16).*

Given a pregroup grammar G, we can generate the free compact-closed category defined by:

$$\mathbf{CC}(G) = \mathbf{RC}(G + \mathtt{swap})/\sim_{\mathtt{sym}}$$

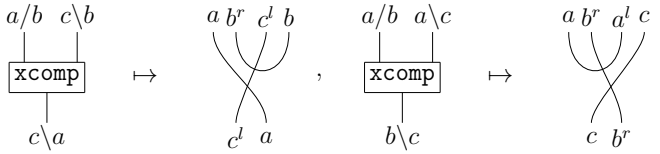
where $\sim_{\texttt{sym}}$ is the congruence induced by the axioms for the symmetry (1.16). Notice that in a compact-closed category $a^r \simeq a^l$ for any object a, see e.g. [HV19]. Pregroup reductions in the free rigid category $\mathbf{RC}(G)$ can of course be mapped in $\mathbf{CC}(G)$ via the canonical rigid functor which forgets the ^{r} and ^{l} adjoint structure.

We cannot use free compact-closed categories directly to parse sentences. If we only ask for a morphism $g : u \to s$ in $\mathbf{CC}(G)$ in order to show that the string u is grammatical, then any permutation of the words in u would also count as grammatical, and we would lose any notion of word order. In practice, the use of the swap must be restricted to only special cases. These were discussed in 1.4, where we saw that the crossed composition rule of combinatory grammars is suitable for modeling these restricted cases.

In recent work [YK21], Yeung and Kartsaklis introduced a translation from CCG grammars to pregroup grammars which allows to build a diagram in the free compact-closed category over a pregroup grammar from any derivation of a CCG. This is useful in practical applications since it allows to turn the output of state-of-the-art CCG parsers such as [YNM17] into compact-closed diagrams. The translation is captured by the following proposition.

Proposition 1.5.13. *For any combinatory grammar G there is a pregroup grammar G' with a canonical functorial reduction $G \to \mathbf{CC}(G')$.*

Proof. The translation is the same as 1.5.7, together with the following mapping for the crossed composition rules:



(1.19)

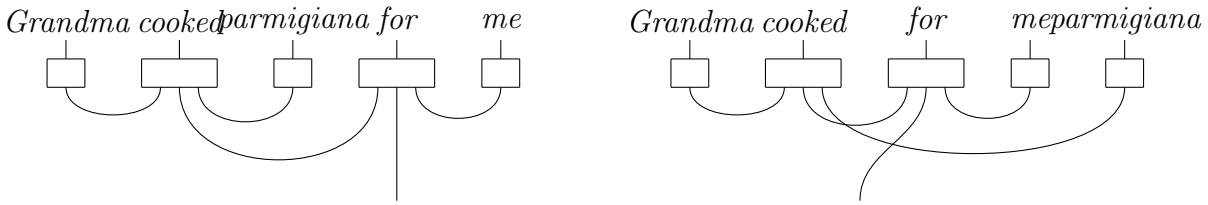
□

Representing the crossed composition rule in compact-closed categories, allows to reason diagrammatically about equivalent syntactic structures.

Example 1.5.14. *As an example consider the pregroup grammar G with basic types $B = \{ n, s \}$ and lexicon given by:*

$$\Delta(\textit{cooked}) = \{ n^r s n^l \} \quad \Delta(\textit{me}) = \Delta(\textit{Grandma}) = \Delta(\textit{parmigiana}) = \{ n \} \quad \Delta(\textit{for}) = \{ s^l s n^l \}$$

Then using the grammar from Example 1.4.14, we can map the two CCG parses to the following compact-closed diagrams in $\mathbf{CC}(G)$, even though the second is not grammatical in a planar pregroup grammar.



If we interpret the wires for words as the unit of the tensor, then these two diagrams are equal in $\mathbf{CC}(G)$ but not when seen in a biclosed category.

1.5.2 Dependency grammars are pregroups

Dependency grammars arose from the work of Lucien Tesniere in the 1950s [Tes59]. It was made formal in the 1960s by Hays [Hay64] and Gaifman [Gai65], who showed that they have the same expressive power as context-free grammars. Dependency grammars are very popular in NLP, supported by large-scale parsing tools such as those provided by Spacy [Hon+20]. We take the formalisation of Gaifman [Gai65] as a starting point and show how the dependency relation may be seen as a diagram in a free rigid category.

Let us fix a vocabulary V and a set of symbols B, called categories in [Gai65], with $s \in B$ the sentence symbol.

Definition 1.5.15 (Dependency grammar [Gai65])**.** *A dependency grammar G consists in a set of rules $G \subseteq (B + V) \times B^*$ of the following shapes:*

I $(x, y_1 \dots y_l \star y_{l+1} \dots y_n)$ *where $x, y_i \in B$, indicating that the symbol x may depend on the symbols $y_1 \dots y_l$ on the left and on the symbols $y_{l+1} \dots y_n$ on the right.*

II (w, x) *for $w \in V$ and $x \in B$, indicating that the word w may have type x.*

III (x, s) *indicating that the symbol x may govern a sentence.*

Following Gaifman [Gai65], we define the language $\mathcal{L}(G)$ generated by a dependency grammar G to be the set of strings $u = w_1 w_2 \dots w_n \in V^*$ such that there are symbols $x_1, x_2 \dots x_n \in B$ and a binary *dependency relation* $d \subseteq X \times X$ where $X = \{ x_1, \dots, x_n \}$ satisfying the following conditions:

1. $(w_i, x_i) \in G$ for all $i \leq n$,
2. $(x_i, x_i) \notin RTC(d)$ where $RTC(d)$ is the reflexive transitive closure of d, i.e. the dependency relation is *acyclic*,
3. if $(x_i, x_j) \in d$ an $(x_i, x_k) \in d$ then $x_j = x_k$, i.e. every symbol depends on at most one head, i.e. the dependency relation is single-headed or *monogamous*,
4. if $i \leq j \leq k$ and $(x_i, x_k) \in RTC(d)$ then $(x_i, x_j) \in RTC(d)$, i.e. the dependency relation is *planar*,
5. there is exactly one x_h such that $\forall j\, (x_h, x_j) \notin d$ and $(x_h, s) \in G$, i.e. the relation is *connected* and *rooted* (x_h is called the root and we say that x_h governs the sentence).
6. for every x_i, let $y_1, \dots, y_l \in X$ be the (ordered list of) symbols which depend on x_i from the left and $y_{l+1}, \dots, y_n \in X$ be the (ordered list of) symbols which depend on x_i from the right, then $(x, y_1 \dots y_l \star y_{l+1} \dots y_n) \in G$, i.e. the dependency structure is allowed by the rules of G.

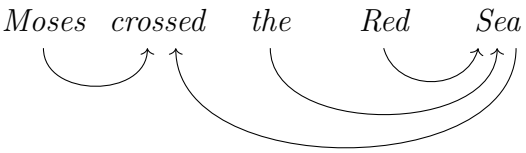
Example 1.5.16. *Consider the dependency grammar with* $V = \{$ *Moses, crossed, the, Red, Sea* $\}$, $B = \{ d, n, a, s, v \}$ *and rules of type I:*

$$(v, n \star n), (a, \star n), (d, \star), (n, \star), (n, ad\star)\,,$$

of type II:

$$(Moses, n), (crossed, v), (the, d), (Red, a), (Sea, n)$$

and a single rule of type III (v, s)*. Then the sentence "She tied a plastic bag" is grammatical as witnessed by the following dependency relation:*



This combinatorial definition of a dependency relation has an algebraic counterpart as a morphism in a free rigid category. Given a dependency grammar G, let $\Delta(G) \subseteq V \times P(B)$ be the pregroup lexicon defined by:

$$(w, y_1^r \dots y_l^r\, x\, y_{l+1}^l \dots y_n^l) \in \Delta(G) \iff (w, x) \in G \wedge (x, y_1 \dots y_l \star y_{l+1} \dots y_n) \in G \tag{1.20}$$

also, let $I(G)$ be rules in G of the form (x, s) where $x \in B$ and s is the sentence symbol.

Proposition 1.5.17. *For any dependency grammar* G*, if a string of words is grammatical* $u \in \mathcal{L}(G)$ *then there exists a morphism* $u \to s$ *in* $\mathbf{RC}(\Delta(G) + I(G))$*.*

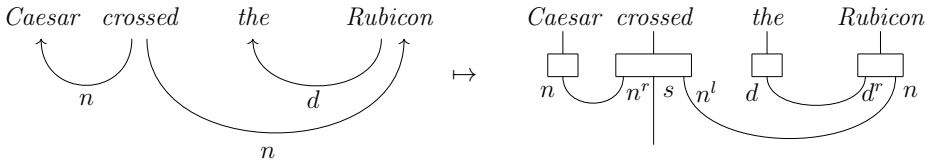
Proof. Fix a dependency grammar G, and suppose $w_1 \dots w_n \in \mathcal{L}(G)$ then there is a list of symbols $X = \{x_1, x_2 \dots x_n\}$ with $x_i \in B$ and a dependency relation $d \subseteq X \times X$ such that the conditions $(1), \dots, (6)$ given above are satisfied. We need to show that d defines a diagram $w_1 \dots w_n \to s$ in $\mathbf{RC}(\Delta(G) + I(G))$. Starting from the type $w_1 w_2 \dots w_n$, conditions (1) and (6) of the dependency relation ensure that there is an assignment of a single lexical entry in $\Delta(G)$ to each w_i. Applying these lexical entries we get a morphism $w_1 w_2 \dots w_n \to T$ in $\mathbf{RC}(\Delta(G))$ where:

$$T = \otimes_{i=1}^{n} (y_{i,1}^r \dots y_{i,l_i}^r\, x_i\, y_{i,l_i+1}^l \dots y_{i,n_i}).$$

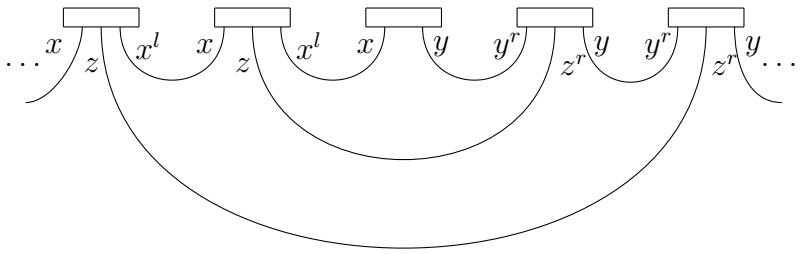
For each pair $(x_i, x_j) \in d$ with $i \leq j$, x_i must appear as some $y_{j,k}$ with $k \leq l_j$ by condition (6). Therefore we can apply a cup connecting x_i and $(y_{j,k})^r$ in T. Similarly for $(x_i, x_j) \in d$ with $j \leq i$, x_i must appear as some $y_{j,k}$ with $k > l_j$ and we can apply a cup connecting $(y_{j,k})^l$ and x_i in T. By monogamy (3) and connectedness (5) of the dependency relation, there is exactly one such pair for each x_i, except for the root x_h. Therefore we can keep applying cups until only x_h is left. By planarity (4) of the dependency relation, these cups don't have to cross, which means that the diagram obtained is a valid morphism $T \to x_h$ in $\mathbf{RC}(\Delta(G))$. Finally condition (5) ensures that there exists an induced step $x_h \to s \in I(G)$. Overall we have built a morphism $w_1 w_2 \dots w_n \to T \to x_h \to s$ in $\mathbf{RC}(\Delta(G) + I(G))$, as required. □

Corollary 1.5.18. *For any dependency grammar G there is a pregroup grammar $\tilde{G} = (V, B, \Delta(G), I(G), s)$ such that $\mathcal{L}(G) \subseteq \mathcal{L}(\tilde{G})$.*

Example 1.5.19. *An example of the translation defined above is the following:*



The proposition above gives a structural reduction from dependency grammars to pregroup grammars, where the dependency relation witnessing the grammaticality of a string u is seen as a pregroup reduction $u \to s$. This leads to a first question: do all the pregroup reduction arise from a dependency relation? In [Pre07], Preller gives a combinatorial description of pregroup reductions, which is strikingly similar to the definition of dependency relation. In particular it features the same conditions for monogamy (3), planarity (4) and connectedness (5). However, pregroup reductions are in general *not* acyclic, as the following example shows:



(1.21)

Therefore we do not expect that any given pregroup grammar reduces to a dependency grammar. The question still remains for pregroup grammars with restricted types of lexicons. Indeed, the cyclic example above uses a lexicon which is not of the shape (1.20). We define operadic pregroups to be pregroup grammars with lexicon of the shape (1.20).

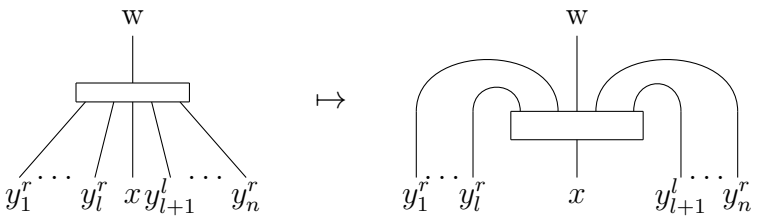
Definition 1.5.20 (Operadic pregroup)**.** *An operadic pregroup is a pregroup grammar $G = (V, B, \Delta, s)$ such that for any lexical entry $(w, t) \in \Delta$ we have $t = y^r x\ z^l$ for some $x \in B$ and $y, z \in B^*$.*

Using Delpeuch's autonomization of monoidal categories [Del19], we can show that reductions in an operadic pregroup always form a tree, justifying the name "operadic" for these structures.

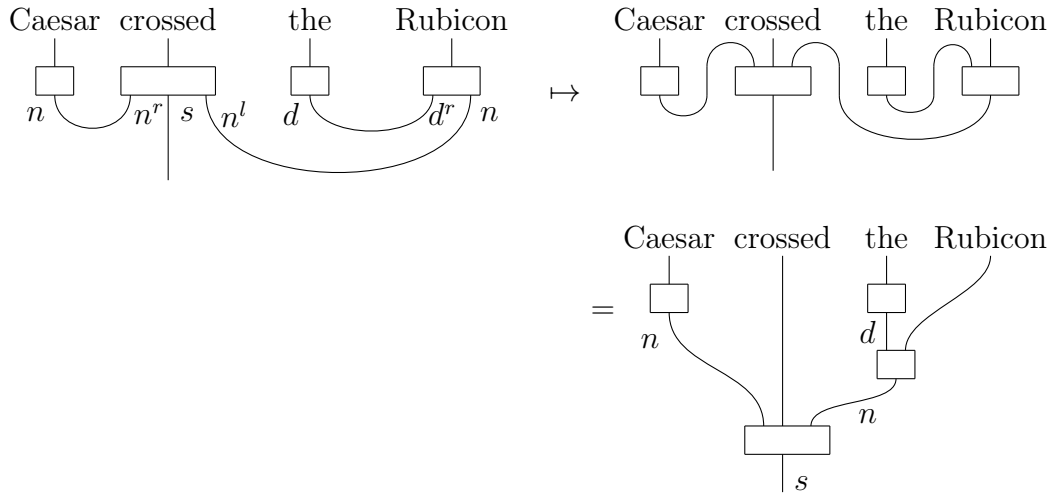
Proposition 1.5.21. *Every operadic pregroup is functorially equivalent to a CFG.*

Proof. It will be sufficient to show that the reductions of an operadic pregroup are trees. Fix an operadic pregroup $\tilde{G} = (V, B, \Delta, I, s)$. We now define a functor $F : \mathbf{RC}(\tilde{G}) \to \mathcal{A}(\mathbf{MC}(G))$ where $\mathcal{A}$ is the free rigid (or autonomous) category construction on monoidal categories as defined by Delpeuch [Del19], and G is a context-free grammar with basic symbols $B + V$ and production rules $y_1 \dots y_l w y_{l+1} \dots y_n \to$

$x \in G$ whenever $(w, y_1^r \dots y_l^r \, x \, y_{l+1}^l \dots y_n^l) \in \tilde{G}$. F is given by the identity on objects $F(x) = x$, and on lexical entries it is defined by:



As shown by Delpeuch the embedding $\mathbf{MC}(G) \to \mathcal{A}(\mathbf{MC}(G))$ is full on the subcategory of morphisms $g : x \to y$ where $x, y \in (B + V)^*$. Given any pregroup reduction $g : u \to s$ in $\mathbf{RC}(\tilde{G})$ we may apply the functor F to get a morphism $F(g) : F(u) \to F(s)$. By definition of F, we have that $F(u) = u$ and $F(s) = s$ are both elements of $(B + V)^*$. By fullness of the embedding, all the cups and caps in $F(g)$ can be removed using the snake equation, i.e. $F(g) \in \mathbf{MC}(G)$. We give an example to illustrate this:



It is easy to see that the induced monoidal diagram has the same connectivity as the pregroup reduction. Since it is a monoidal diagram it must be *acyclic*, and since all the boxes have only one ouput it must be a *tree*, finishing the proof. □

Proposition 1.5.22. *Every dependency grammar is struturally equivalent to an operadic pregroup.*

Proof. We need to show that given a dependency grammar G and a string of words $u \in V^*$, dependency relations for u are in one-to-one correspondence with reductions $u \to s$ for the operadic pregroup $\tilde{G} = (V, B, \Delta(G), I(G), s)$. The proof of Proposition 1.5.17, gives an injection of dependency relations for u to pregroup reductions $u \to s$ for $\tilde{G}$. For the opposite direction note that, by the Lambek switching lemma 1.5.9, any pregroup reduction $u \to s$ can be obtained using lexical entries followed by a diagram made only of cups (contractions). This defines a relation on u satisfying conditions (1) and (3-6) of dependency relations, see [Pre07]. It remains to show that also condition (2) is satisfied, i.e. that reductions in an operadic pregroup are *acyclic*, but this follows from Proposition 1.5.21. □

Theorem 1.5.23. *Every dependency grammar is structurally equivalent to both a pregroup and a context-free grammar.*

Proof. Follows from Propositions 1.5.22 and 1.5.21. □

Overall, we have three equivalent ways of looking at the structures induced by dependency grammars a.k.a operadic pregroups. First, we may see them as *dependency relations* as first defined by Gaifman [Gai65] and reviewed above. Second, we may see them as *pregroup reductions* (i.e. patterns of cups) as proven in Proposition 1.5.17. Third, we may see them as *trees* as shown in Proposition 1.5.21.

On the one hand, this new algebraic perspective will allow us to give functorial semantics to dependency grammars. In 2.5 we interpret them in rigid categories using their characterization as pregroup grammars. In 3.4.3, we interpret them in a category of probabilistic processes (where cups and caps are not allowed) using the characterization of dependency relations as trees. On the other, it allows us to interface DisCoPy with established dependency parsers such as those provided by SpaCy [Hon+20].

1.5.3 rigid.Diagram

The `rigid` module of DisCoPy is often used as a suitable intermediate step between any grammar and tensor-based semantics. For example, the `lambeq` package [Kar+21] provides a method for generating instances of `rigid.Diagram` from strings parsed using a transformer-based CCG parser [**clark2021**]. We describe the implementation of the `rigid` module and construct an interface with SpaCy's dependency parser [Hon+20]. A `rigid.Ob`, or basic type, is defined by a `name` and a winding number `z`. It comes with property methods `.l` and `.r` for taking left and right adjoints by acting on the winding integer `z`.

Listing 1.5.24. Basic types and their iterated adjoints.

```
class Ob(cat.Ob):
    @property
    def z(self):
        """ Winding number """
        return self._z

    @property
    def l(self):
        """ Left adjoint """
        return Ob(self.name, self.z - 1)

    @property
    def r(self):
        """ Right adjoint """
        return Ob(self.name, self.z + 1)

    def __init__(self, name, z=0):
        self._z = z
        super().__init__(name)
```

Types in rigid categories also come with a monoidal product, We implement them by subclassing `monoidal.Ty` and providing the defining methods of `rigid.Ob`, note that taking adjoints reverses the order of objects.

Listing 1.5.25. Pregroup types, i.e. types in free rigid categories.

```
class Ty(monoidal.Ty, Ob):
    @property
    def l(self):
        return Ty(*[x.l for x in self.objects[::-1]])

    @property
    def r(self):
        return Ty(*[x.r for x in self.objects[::-1]])

    @property
    def z(self):
        return self[0].z

    def __init__(self, *t):
        t = [x if isinstance(x, Ob)
             else Ob(x.name) if isinstance(x, cat.Ob)
             else Ob(x) for x in t]
        monoidal.Ty.__init__(self, *t)
        Ob.__init__(self, str(self))
```

Rigid diagrams are monoidal diagrams with special morphisms called `Cup` and `Cap`, satisfying the snake equations (1.18). The requirement that the axioms are satisfied is relaxed to the availability of a polynomial time algorithm for checking equality of morphisms. This is implemented in DisCoPy, with the `rigid.Diagram.normal_fom` method, following the algorithm of Dunn and Vicary [DV19b, Definition 2.12]. Rigid diagrams are also biclosed, i.e. they can be `curry`ed and `uncurry`ed.

Listing 1.5.26. Diagrams in free rigid categories.

```
@monoidal.Diagram.subclass
class Diagram(monoidal.Diagram):
    @staticmethod
    def cups(left, right):
        return cups(left, right)

    @staticmethod
    def caps(left, right):
        return caps(left, right)

    @staticmethod
    def curry(self, n_wires=1, left=False):
        return curry(self, n_wires=n_wires, left=left)

    @staticmethod
    def uncurry(self, n_wires=1, left=False):
        return uncurry(self, n_wires=n_wires, left=left)
```

```
    def normal_form(self, normalizer=None, **params):
        ...


class Box(monoidal.Box, Diagram):
    ...


class Id(monoidal.Id, Diagram):
    ...
```

Note that currying and uncurrying correspond to transposition of wires in the rigid setting. The class comes with its own `Box` *instance which carry a winding number* `_z` *for their transpositions. The currying and uncurrying functions are defined as follows.*

```
from discopy.rigid import Id, Ty, Box, Diagram

def curry(diagram, n_wires=1, left=False):
    if not n_wires > 0:
        return diagram
    if left:
        wires = diagram.dom[:n_wires]
        return Diagram.caps(wires.r, wires) @ Id(diagram.dom[n_wires:])\
            >> Id(wires.r) @ diagram
    wires = diagram.dom[-n_wires:]
    return Id(diagram.dom[:-n_wires]) @ Diagram.caps(wires, wires.l)\
        >> diagram @ Id(wires.l)

def uncurry(diagram, n_wires=1, left=False):
    if not n_wires > 0:
        return diagram
    if left:
        wires = diagram.cod[:n_wires]
        return Id(wires.l) @ diagram\
            >> Diagram.cups(wires.l, wires) @ Id(diagram.cod[n_wires:])
    wires = diagram.cod[-n_wires:]
    return diagram @ Id(wires.r)\
        >> Id(diagram.cod[:-n_wires]) @ Diagram.cups(wires, wires.r)
```

We only showed the main methods available with rigid diagrams. The DisCoPy implementation also comes with classes `Cup` and `Cap` for representing the structural morphisms. This allows to define `rigid.Functor` as a monoidal Functor with a predefined mapping on instances of `Cup` and `Cap`.

Listing 1.5.27. Functors from free rigid categories.

```
class Functor(monoidal.Functor):
    def __init__(self, ob, ar, cod=(Ty, Diagram)):
        super().__init__(ob, ar, cod=cod)

    def __call__(self, diagram):
        if isinstance(diagram, Ty):
            ...
```

```
        if isinstance(diagram, Cup):
            return self.cod[1].cups(
                self(diagram.dom[:1]), self(diagram.dom[1:]))
        if isinstance(diagram, Cap):
            return self.cod[1].caps(
                self(diagram.cod[:1]), self(diagram.cod[1:]))
        if isinstance(diagram, Box):
            ...
        if isinstance(diagram, monoidal.Diagram):
            return super().__call__(diagram)
        raise TypeError()
```

We build an interface with the dependency parser of SpaCy [Hon+20]. From a SpaCy dependency parse we may obtain both an `operad.Tree` and a `rigid.Diagram`.

Listing 1.5.28. Interface between `spacy` and `operad.Tree`

```
from discopy import operad

def find_root(doc):
    for word in doc:
        if word.dep_ == 'ROOT':
            return word

def doc2tree(word):
    if not word.children:
        return operad.Box(word.text, operad.Ob(word.dep_), [])
    root = operad.Box(word.text, operad.Ob(word.dep_),
                      [operad.Ob(child.dep_) for child in word.children])
    return root(*[doc2tree(child) for child in word.children])

def from_spacy(doc):
    root = find_root(doc)
    return doc2tree(root)
```

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("Moses crossed the Red Sea")
assert str(from_spacy(doc)) = 'crossed(Moses, Sea(the, Red))'
```

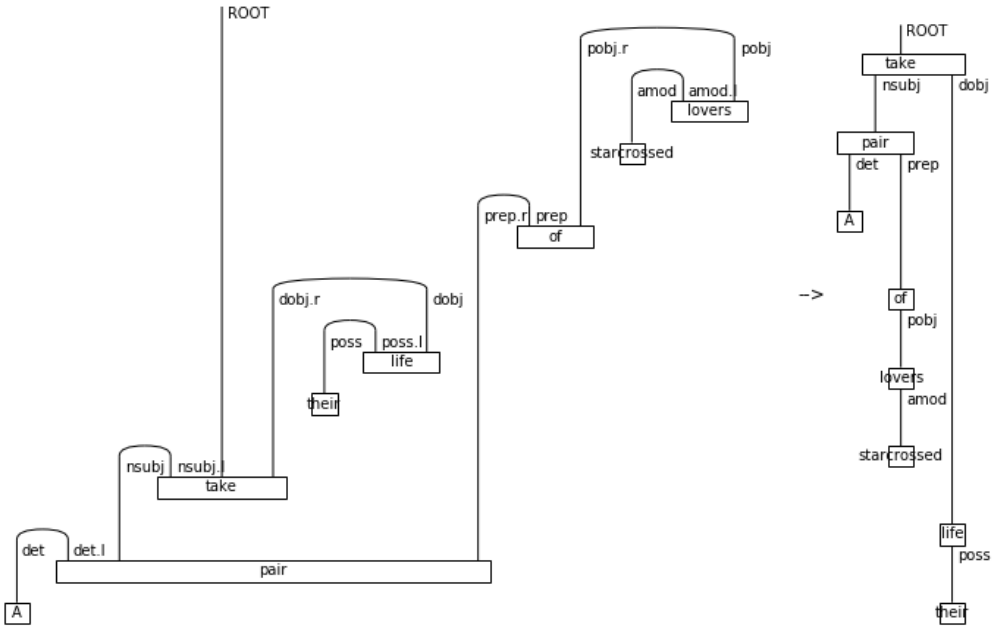
Listing 1.5.29. Interface between `spacy` and `rigid.Diagram`

```
from discopy.rigid import Ty, Id, Box, Diagram, Functor

def doc2rigid(word):
    children = word.children
    if not children:
        return Box(word.text, Ty(word.dep_), Ty())
    left = Ty(*[child.dep_ for child in word.lefts])
    right = Ty(*[child.dep_ for child in word.rights])
    box = Box(word.text, left.l @ Ty(word.dep_) @ right.r, Ty(),
              data=[left, Ty(word.dep_), right])
    top = curry(curry(box, n_wires=len(left), left=True), n_wires=len(right))
```

```
    bot = Id(Ty()).tensor(*[doc2rigid(child) for child in children])
    return top >> bot

def doc2pregroup(doc):
    root = find_root(doc)
    return doc2rigid(root)
```

We can now build pregroup reductions from dependency parses. We check that the outputs of the two interfaces are functorially equivalent.

```
def rewiring(box):
    if not box.data:
        return Box(box.name, box.dom, Ty())
    left, middle, right = box.data[0], box.data[1], box.data[2]
    new_box = Box(box.name, middle, left @ right)
    return uncurry(uncurry(new_box, len(left), left=True), len(right))

F = Functor(ob=lambda ob: Ty(ob.name), ar=rewiring)
assert repr(F(doc2pregroup(doc)).normal_form()) ==\
    repr(operad.tree2diagram(from_spacy(doc)))

drawing.equation(dep2pregroup(doc).normal_form(left=True),
                 operad.tree2diagram(from_spacy(doc)), symbol='->')
```



1.6 Hypergraphs and coreference

Coreference resolution is the task of finding all linguistic expressions, called mentions, which refer to the same entity in a piece of text. It has been a core research topic in NLP [Ela06], including early syntax-based models of pronoun resolution [Hob78], Bayesian and statistical approaches [GHC98; CM15] as well as neural-based models [CM16; Lee+17]. This is still a very active area of research with new state-of-the-art models released every year, and several open-source tools available in the web [Qi+20].

In the previous sections, we studied a range of formal grammars that capture the syntactic structure of sentences. The aim of this section is to cross the sentence boundary and move towards an analysis of text and discourse. Assuming that the resolution process has been completed, we want a suitable syntactic representation of text with coreference. We can obtain it using a piece of structure known as a commutative special Frobenius algebra, or more simply a "spider". These were introduced in linguistics by Sadrzadeh et al. [SCC13; SCC14] as a model for relative pronouns, and have recently been used by Coecke [Coe20] to model the interaction of sentences within text.

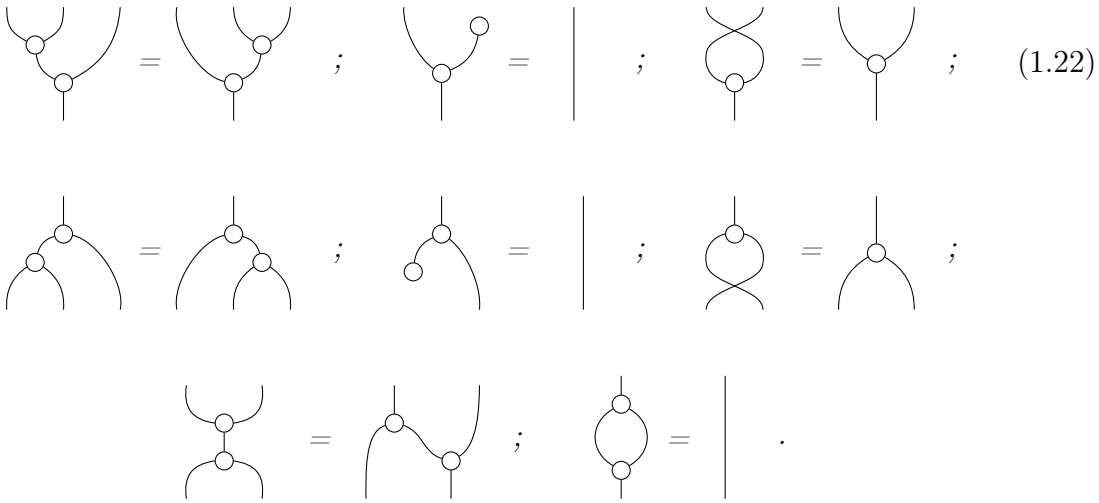
In this section, we introduce pregroup grammars with coreference, a syntactic model which allows to represent the grammatical and referential structure of text diagrammatically. This is similar in spirit to the work of Coecke [Coe20], although our approach preserves the pregroup formalism and adds coreference as extra structure. This makes our model suited for implementation since one can first parse sentences with a pregroup grammar and then link the entities together using coreference resolution tools. We show a proof-of-concept implementation using the `hypergraph` module of DisCoPy.

1.6.1 Hypergraph categories

The term "hypergraph categories" was introduced in 2018 by Fong and Spivak [FS18b], to refer to categories equipped with *Frobenius algebras* on every object. These structures were studied at least since Carboni and Walters [CW87] and have been applied to such diverse fields as databases [BSS18], control theory [BE14], quantum computing [CK17] and linguistics [SCC14]. In a recent line of work [Bon+16; Bon+20; Bon+21], Bonchi, Sobocinski et al. developed a rewrite theory for morphisms in these categories in terms of double-pushout hypergraph rewriting [Bon+20]. This makes apparent the combinatorial nature of hypergraph categories, making them particularly suited to implementation [WZ21]. We will not be interested here in the rewrite theory for these categories, but rather in their power in representing the grammatical and coreferential structure of language. They will also provide us with an intermediate step between syntax and the semantics of Sections 2.4 and 2.5.

Definition 1.6.1 (Hypergraph category)**.** *[FS18b] A hypergraph category is a symmetric monoidal category such that each object a is equipped with a commutative*

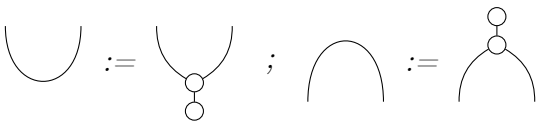
special Frobenius algebra $\texttt{Frob}_a = \{\Delta_a, \epsilon_a, \nabla_a, \eta_a\}$ *satisfying the following axioms:*



(1.22)

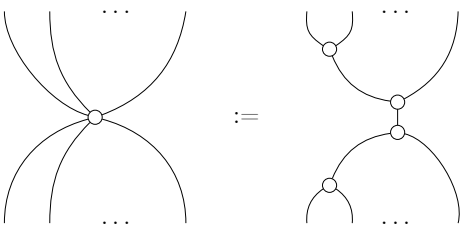
Where the unlabeled wire denotes object a.

Proposition 1.6.2. *Hypergraph categories are self-dual compact-closed with cups and caps given by:*



(1.23)

The axioms of commutative special Frobenius algebras (1.22), may be expressed in a more intuitive way as fusion rules of *spiders*. Spiders are defined as follows:

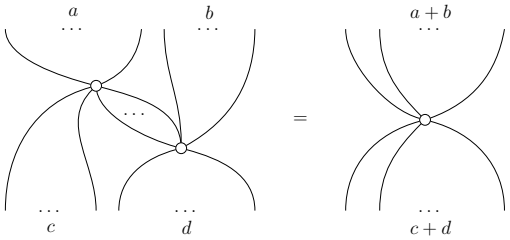


(1.24)

They are the normal form of commutative special Frobenius algebras. More precisely, using the axioms (1.22), it can be shown that any connected diagram built using the Frobenius generators $\texttt{Frob}_a$ can be rewritten into the right-hand side above. This was shown in the context of categorical quantum mechanics [HV19] where Frobenius algebras, corresponding to "observables", play a foundational role. The following result is also proved in [HV19], and used in the context of the ZX calculus [van20], it provides a more concise and intuitive way of reasoning with commutative special Frobenius algebras.

Proposition 1.6.3 (Spider fusion)**.** *[HV19] The axioms of special commutative Frobe-*

nius algebras (1.22), are equivalent to the spider fusion rules:

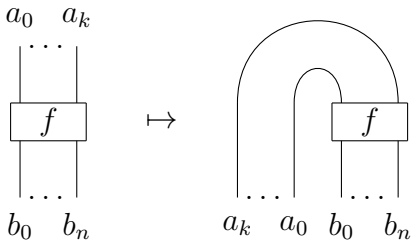


(1.25)

Given a monoidal signature Σ, the free hypergraph category is defined by:

$$\mathbf{Hyp}(\Sigma) = \mathbf{MC}(\Sigma + \{\, \mathtt{Frob}_a \,\}_{a \in \Sigma_0}) / \cong$$

where **MC** is the free monoidal category construction, defined in 1.3 and $\cong$ is the equivalence relation generated by the axioms of commutative special Frobenius algebras (1.22), or equivalently the spider fusion rules (1.25). Without loss of generality, we may assume that the monoidal signature Σ has trivial `dom` function. Formally we have that for any monoidal signature $B^* \xleftarrow{\mathtt{dom}} \Sigma \xrightarrow{\mathtt{cod}} B^*$ there is a monoidal signature of the form $\Sigma' \xrightarrow{\mathtt{cod}} B^*$ such that $\mathbf{Hyp}(\Sigma) \simeq \mathbf{Hyp}(\Sigma')$. The signature Σ' is defined by taking the *name* of every generator in Σ, i.e. turning all the inputs into outputs using caps as follows:

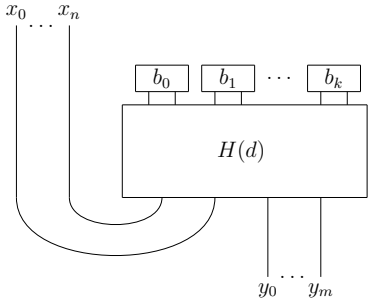


We define a hypergraph signature as a monoidal signature where the generators have only output types.

Definition 1.6.4 (Hypergraph signature)**.** *A hypergraph signature Σ over B is a set of hyperedge symbols Σ together with a map $\sigma : \Sigma \to B^*$.*

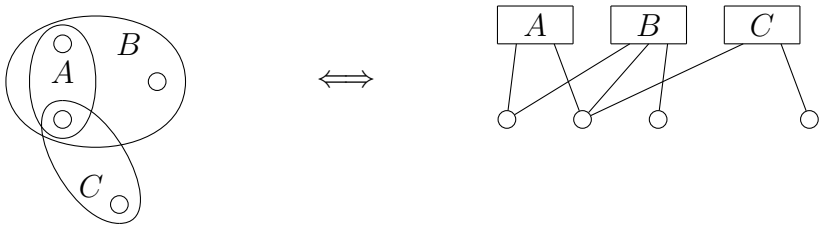
Morphisms in $\mathbf{Hyp}(\Sigma)$ for a hypergraph signature Σ have a normal form given as follows.

Proposition 1.6.5 (Hypergraph normal form)**.** *Let $\Sigma \to B^*$ be a hypergraph signature and $x, y \in B^*$. Any morphism $d : x \to y \in \mathbf{Hyp}(\Sigma)$ is equal to a diagram of the following shape:*



where $H(d) \in \mathbf{Hyp}(\varnothing)$ is a morphism built using only spiders, i.e. generated from $\cup_b \mathtt{Frob}_b$ for $b \in B$.

This normal form justifies the name "hypergraph" for these categories. Indeed we may think of the spiders of a diagram d as *vertices*, and boxes with n ports as *hyperedges* with n vertices. Then the morphism $H(d)$ defined above is the *incidence graph* of d, indicating for each vertex (spider) the hyperedges (boxes) that contain it. Note however, that morphisms in $\mathbf{Hyp}(\Sigma)$ carry more data than a simple hypergraph. First, they carry labels for every hyperedge and every vertex. Second they are "open", i.e. they carry input-output information allowing to compose them. If we consider morphisms $d : 1 \to 1$ in $\mathbf{Hyp}(\Sigma)$, these are in one-to-one correspondence with hypergraphs labeled by Σ, as exemplified in the following picture:



(1.26)

1.6.2 Pregroups with coreference

Starting from a pregroup grammar $G = (V, B, \Delta, I, s)$, we can model coreference by adding memory or *reference types* for each lexical entry, and rules that allow to swap, merge or discard these reference types. Formally, this is done by fixing a set of reference types R and allowing the lexicon to assign reference types alongside pregroup types:

$$\Delta \subseteq V \times P(B) \times R^*$$

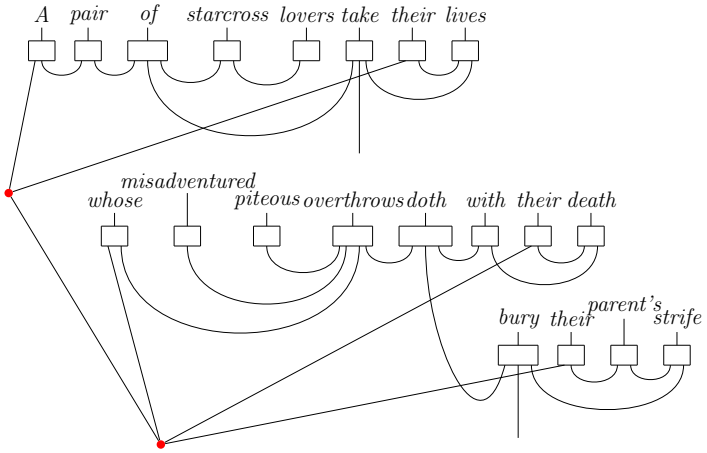
We can then represent the derivations for such a grammar with coreference in one category $\mathbf{Coref}(\Delta, I)$ defined by:

$$\mathbf{Coref}(\Delta, I) = \mathbf{RC}(\Delta + I + \{\, \mathtt{Frob}_r \,\}_{r \in R}, \{\, \mathtt{swap}_{r,x} \,\}_{r \in R,\, x \in P(B)+R})$$

where $\mathtt{Frob}_r$ contains the generators of Frobenius algebras for every reference type $r \in R$ (allowing to initialise, merge, copy and delete them), $\mathtt{swap}_{r,x}$ allows to swap any reference type $r \in R$ to the right of any other type $x \in P(B) + R$, i.e. reference types commute with any other object. Note that we are not quotienting **Coref** by any axioms since we use it only to represent the derivations.

Definition 1.6.6 (Pregroup with coreference)**.** *A pregroup grammar with coreference is a tuple $G = (V, B, R, I, \Delta, s)$ where V is a vocabulary, $B \ni s$ is a set of basic types, R is a set of reference types, $I \subseteq B \times B$ is a set of induced steps, $\Delta \subseteq V \times P(B) \times R^*$ is a lexicon, assigning to every word $w \in V$ a set of types $\Delta(w)$ consisting of a pregroup type in $P(B)$ and a list of reference type in R^*. An utterance $u = w_0 \dots w_n \in V^*$ is k-grammatical in G if there are types $(t_i, r_i) \in \Delta(w_i) \subseteq P(B) \times R^*$ such that there is morphism $t_0 \otimes r_0 \otimes t_1 \otimes r_1 \dots t_n \otimes r_n \to s^k$ in $\mathbf{Coref}(\Delta, I) =: \mathbf{Coref}(G)$ for some $k \in \mathbb{N}$.*

Example 1.6.7. *We denote reference types with red wires. The following is a valid reduction in a pregroup grammar with coreference, obtained from Example 1.5.6 by adding reference types for the words "A", "whose" and "their".*



We can now consider the problem of parsing a pregroup grammar with coreference.

Definition 1.6.8. `CorefParsing`

Input: G, $u \in V^*$, $k \in \mathbb{N}$
Output: $f \in \mathbf{Coref}(G)(u, s^k)$

Note that, seen as a decision problem, `CorefParsing` is equivalent to the parsing problem for pregroups.

Proposition 1.6.9. *The problem* ∃`CorefParsing` *is equivalent to the parsing problem for pregroup grammars.*

Proof. First note that for any pregroup grammar with coreference $G = (V, B, R, I, \Delta, s)$ there is aa corresponding pregroup grammar $\tilde{G} = (V, B, I, \tilde{\Delta}, s^k)$ where $\tilde{\Delta}$ is obtained from Δ by projecting out the R^* component. Suppose there is a reduction $d : u \to s^k$ in $\mathbf{Coref}(G)$, since there are no boxes making pregroup and reference types interact, we can split the diagram d into a pregroup reduction in $\mathbf{RC}(G)$ and a coreference resolution $R^n \to 1$ where n is th number of reference types used. Therefore u is also grammatical in $\tilde{G}$. Now, $\tilde{G}$ reduces functorially to G since we can map the lexical entries in $\tilde{\Delta}$ to the corresponding entry in Δ followed by discarding the reference types with counit spiders. Therefore any parsing for $\tilde{G}$ induces a parsing for $\mathbf{Coref}(G)$, finishing the proof. □

Supposing we can parse pregroups efficiently, `CorefParsing` becomes only interesting as a function problem: which coreference resolution should we choose for a given input utterance u? There is no definite mathematical answer to this question. Depending on the context, the same utterance may be resolved in different ways. Prominent current approaches are neural-based such as the mention-ranking coreference models of Clark and Manning [**clark2016a**]. SpaCy offers a `neuralcoref` package implementing their model and we show how to interface it with the `rigid.Diagram`

class at the end of this section. Being able to represent the coreference resolution in the diagram for a piece of text is particularly useful in semantics, for example it allows to build arbitrary conjunctive queries as one-dimensional strings of words, see 2.4.

The image of the parsing function for pregroups with coreference may indeed be arbitrarily omplicated, in the sense that any hypergraph $d \in \mathbf{Hyp}(\Sigma)$ can be obtained as a k-grammatical reduction, where the hyperedges in Σ are seen as *verbs* and the vertices in B as *nouns*.

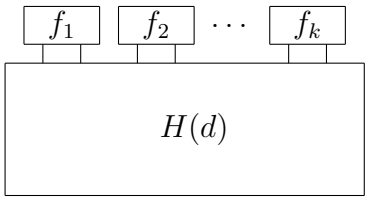
Proposition 1.6.10. *For any finite hypergraph signature $\Sigma \xrightarrow{\sigma} B^*$, there is a pregroup grammar with coreference $G = (V, B, R, \Delta(\sigma), s)$ and a full functor $J : \mathbf{Coref}(\Delta(\sigma)) \to \mathbf{Hyp}(\Sigma)$ with $J(w) = J(s) = 1$ for $w \in V$.*

Proof. We define the vocabulary $V = \Sigma + B$ where the elements of Σ are seen as verbs and the elements of B as nouns. We also set the basic types to be $B + \{ s \}$ and the reference types to be $R = B$. The lexicon $\Delta(\sigma) \subseteq V \times P(B + \{ s \}) \times R^*$ is defined by:

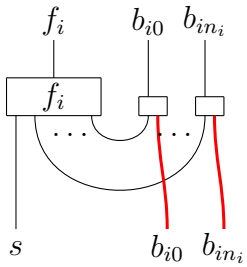
$$\Delta(\sigma) = \{ (w, s\, \sigma(w)^l, \epsilon) \mid w \in \Sigma \subseteq V \} + \{ (w, w, w) | w \in B \subseteq V \}$$

where ϵ denotes the empty list. The functor $J : \mathbf{Coref}(\Delta(\sigma)) \to \mathbf{Hyp}(\Sigma)$ is given on objects by $J(w) = 1$ for $w \in V$, $J(x) = x$ for $x \in B + R$ and $J(s) = 1$ and on the lexical entries by $J(w) = w$ if $w \in \Sigma$ and $J(w) = \mathtt{cup}_w$ if $w \in B$.

In order to prove the proposition, we show that for any $d : 1 \to 1 \in \mathbf{Hyp}(\Sigma)$ there is an utterance $u \in V^*$ with a k-grammatical reduction $g : u \to s^k$ in $\mathbf{Coref}(\Delta(\sigma))$ such that $J(g) = d$, where k is the number of boxes in d. Fix any diagram $d : 1 \to 1 \in \mathbf{Hyp}(\Sigma)$. By proposition 1.6.5, it can be put in normal form:



Where $f_1 \dots f_k$ are the boxes in d and $H(d)$ is a morphism built only from the Frobenius generators $\mathtt{Frob}_b$ for $b \in B$. Let $\sigma(f_i) = b_{i0} \dots b_{in_i}$, then for every $i \in \{ 1, \dots k \}$, we may build a sentence as follows (leaving open the reference wires):



Tensoring these sentences together and connecting them with coreference $H(d)$ we get a morphism $g : u \to s^k$ in $\mathbf{Coref}(\Delta(\sigma))$ where $u = f_1\sigma(f_1) \dots f_k\sigma(f_k)$, and it is easy to check that $J(g) = d$. □

1.6.3 hypergraph.Diagram

The `hypergraph` module of DisCoPy, still under development, is an implementation of diagrams in free hypergraph categories based on [Bon+20; Bon+21]. The main class `hypergraph.Diagram` is well documented and it comes with a method for composition implemented using pushouts. We give a high-level overview of the datastrutures of this module, with a proof-of-concept implementation of dependency grammars with coreference. Before introducing diagrams, recall that the types of free hypergraph categories are *self-dual* rigid types.

```
class Ty(rigid.Ty):
    @property
    def l(self):
        return Ty(*self.objects[::-1])

    r = l
```

We store a `hypergraph.Diagram` via its incidence graph. This is given by a list of `boxes` and a list of integers `wires` of the same length as ports.

```
port_types = list(map(Ty, self.dom)) + sum(
    [list(map(Ty, box.dom @ box.cod)) for box in boxes], [])\
    + list(map(Ty, self.cod))
```

Note that `port_types` are the concatenation of the domain of the diagram, the pairs of domains and codomains of each box, and the codomain od the diagram. We see that `wires` defines a mapping from ports to spiders, which we store as a list of `spider_types`.

```
spider_types = {}
for spider, typ in zip(wires, port_types):
    if spider in spider_types:
        if spider_types[spider] != typ:
            raise AxiomError
    else:
        spider_types[spider] = typ
spider_types = [spider_types[i] for i in sorted(spider_types)]
```

Thus a `hypergraph.Diagram` is initialised by a domain `dom`, a codomain `cod`, a list of boxes `boxes` and a list of wires `wires` as characterised above. The `__init__` method automatically computes the `spider_types`, and in doing so checks that the diagram is well-typed.

Listing 1.6.11. Diagrams in free hypergraph categories

```
class Diagram(cat.Arrow):
    def __init__(self, dom, cod, boxes, wires, spider_types=None):
        ...

    def then(self, other):
        ...

    def tensor(self, other=None, *rest):
        ...
```

```
    __matmul__ = tensor

    @property
    def is_monogamous(self):
        ...

    @property
    def is_bijective(self):
        ...

    @property
    def is_progressive(self):
        ...

    def downgrade(self):
        ...

    @staticmethod
    def upgrade(old):
        return rigid.Functor(
            ob=lambda typ: Ty(typ[0]),
            ar=lambda box: Box(box.name, box.dom, box.cod),
            ob_factory=Ty, ar_factory=Diagram)(old)

    def draw(self, seed=None, k=.25, path=None):
        ...
```

The current `draw` *method is based on a random spring layout algorithm of the hypergraph. One may check whether a diagram is symmetric, compact-closed or traced, using methods* `is_progressive`, `is_bijective` *and* `is_monogamous` *respectively. The* `downgrade` *method turns any* `hypergraph.Diagram` *into a* `rigid.Diagram`. *This is by no means an optimal algorithm. There are indeed many ways to tackle the problem of extraction of rigid diagrams from hypergraph diagrams.*

The classes `hypergraph.Box` and `hypergraph.Id` are defined as usual, by suclassing the corresponding `rigid` classes and `Diagram`. Two special types of morphisms, `Spider` and `Swap`, can be defined directly as subclasses of `Diagram`.

Listing 1.6.12. Swaps and spiders in free hypergraph categories

```
class Swap(Diagram):
    """ Swap diagram. """
    def __init__(self, left, right):
        dom, cod = left @ right, right @ left
        boxes, wires = [], list(range(len(dom)))\
            + list(range(len(left), len(dom))) + list(range(len(left)))
        super().__init__(dom, cod, boxes, wires)


class Spider(Diagram):
    """ Spider diagram. """
    def __init__(self, n_legs_in, n_legs_out, typ):
```

```
        dom, cod = typ ** n_legs_in, typ ** n_legs_out
        boxes, spider_types = [], list(map(Ty, typ))
        wires = (n_legs_in + n_legs_out) * list(range(len(typ)))
        super().__init__(dom, cod, boxes, wires, spider_types)
```

We now show how to build hypergraph diagrams from SpaCy's dependency parser and the coreference information provided by the package `neuralcoref`.

Listing 1.6.13. Coreference and hypergraph diagrams

```
import spacy
import neuralcoref

nlp = spacy.load('en')
neuralcoref.add_to_pipe(nlp)
doc1 = nlp("A pair of starcross lovers take their life")
doc2 = nlp("whose misadventured piteous overthrows doth with \
            their death bury their parent's strife.")
```

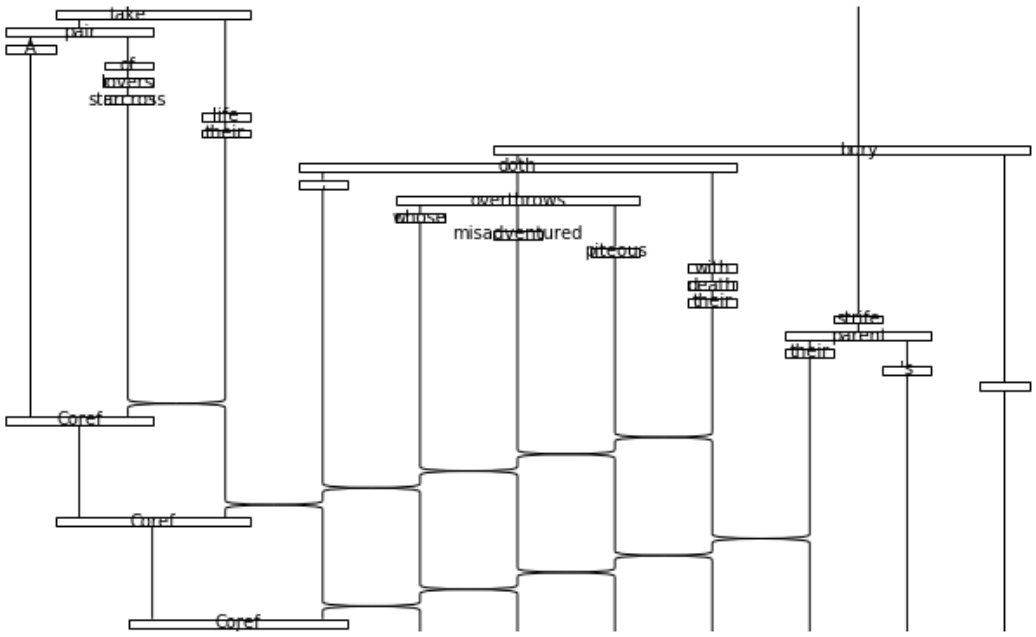
We use the interface with SpaCy from the `operad` *module to extract dependency parses and a* `rigid.Functor` *with codomain* `hypergraph.Diagram` *to turn the dependency trees into hypergraphs.*

```
from discopy.operad import from_spacy, tree2diagram
from discopy.hypergraph import Ty, Id, Box, Diagram
from discopy import rigid

F = rigid.Functor(ob=lambda typ: Ty(typ[0]),
                  ar=lambda box: Box(box.name, box.dom, box.cod),
                  ob_factory=Ty, ar_factory=Diagram)

text = F(tree2diagram(from_spacy(doc1, lexicalised=True)))\
    @ F(tree2diagram(from_spacy(doc2, lexicalised=True)))
assert text.is_monogamous
text.downgrade().draw(figsize=(10, 7))
```

We can now use composition in `hypergraph` *to add coreference boxes that link leaves according to the coreference* `clusters`*.*

```
from discopy.rigid import Ob

ref = lambda x, y: Box('Coref', Ty(x, y), Ty(x))

def coref(diagram, word0, word1):
    pos0 = diagram.cod.objects.index(word0)
    pos1 = diagram.cod.objects.index(word1)
    swaps = Id(Ty(word0)) @ \
        Diagram.swap(diagram.cod[pos0 + 1 or pos1:pos1], Ty(word1))
    coreference = swaps >> \
        ref(word0, word1) @ Id(diagram.cod[pos0 + 1 or pos1:pos1])
    return diagram >> Id(diagram.cod[:pos0]) @ coreference @ \
        Id(diagram.cod[pos1 + 1 or len(diagram.cod):])

def resolve(diagram, clusters):
```

```
    coref_diagram = diagram
    for cluster in clusters:
        main = str(cluster.main)[0]
        for mention in cluster.mentions[1:]:
            coref_diagram = coref(coref_diagram, Ob(main), Ob(str(mention)))
    return coref_diagram

doc = nlp("A pair of starcross lovers take their life, whose misadventured \
    piteous overthrows doth with their death bury their parent's strife.")
clusters = doc._.coref_clusters
resolve(text, clusters).downgrade().draw(figsize=(11, 9), draw_type_labels=False)
```



Note that the only way we have so far of applying functors to a `hypergraph.Diagram` is by first downgrading it to a `rigid.Diagram`. An important direction of future work is the implementation of double pushout rewriting for hypergraph diagrams [Bon+20; Bon+21]. In particular, this would allow to compute free functors directly on the hypergraph representation as they are special instances of rewrites.

Chapter 2

Functors for Semantics

The modern word "semantics" emerged from the linguistic turn of the end of the 19th century along with Peirce's "semiotics" and Saussure's "semiology". It was introduced as "sémantique" by the French linguist Michel Breal, and has its root in the greek word $\sigma\eta\mu\check{\alpha}\sigma\acute{\imath}\bar{\alpha}$ (semasia) which translates to "meaning" or "signification". Semantics is the scientific study of language meaning. It thus presupposes a definition of language as a system of signs, a definition of meaning as a mental or computational process, and an understanding of how signs are mapped onto their meaning. The definition and analysis of these concepts is a problem which has motivated the work of linguists, logicians and computer scientists throughout the 20th century.

In his work on "The Semantic Conception" [Tar36; Tar43], Alfred Tarski proposed to found the science of semantics on the concept of *truth* relative to a *model*. As a mathematician, Tarski focused on the formal language of logic and identified the conditions under which a logical formula φ is true in a model K. The philosophical intuition and mathematical tools developed by Tarski and his collaborators had a great impact on linguistics and computer science. They form the basis of Davidson's [Dav67b; Dav67a] truth-conditional semantics, as well as Montague's "Universal Grammar" [Mon70b; Mon70a; Mon73], which translates natural language sentences into logical formulae. They are also at the heart of the development of relational databases in the 1970s [Cod70; CB76; CM77], where formulas are used as queries for a structured storage of data. These approaches adhere to the principle of *compositionality*, which we may sum up in Frege's words: "The possibility of our understanding sentences which we have never heard before rests evidently on this, that we can construct the sense of a sentence out of individual parts which correspond to words" [Fre14]. In Tarski's approach, compositionality manifests itself in the notion of *satisfaction* for a formula φ in a model K, which is defined by induction over the formal grammar from which φ is constructed.

$$\text{Formulas} \xrightarrow{\text{Model}} \text{Truth}$$

In his 1963 thesis [Law63], Lawvere introduced the concept of *functorial semantics* as a foundation for universal algebra. The idea is to represent syntax as a free category with products and semantics as a structure-preserving functor computing

the meaning of a compound algebraic expression from the semantics of its basic operations. The functorial approach to semantics is naturally compositional, but it generalises Tarski's set-theoretic approach by allowing the semantic category to represent processes in different models of computation. For example, taking semantics in the category of complex vector spaces and linear maps allows to build a structural understanding of quantum information protocols [AC07]. The same principles are used in applications of category theory to probability [Fon13; CJ19], databases [Spi12], chemistry [BP17] and network theory [BCR18]. Of particular interest to us are the Categorical Compositional Distributional (DisCoCat) models of Coecke et al. [CCS08; CCS10] where functors are used to compute the semantics of natural language sentences from the distributional embeddings of their constituent words. As we will see, functors are useful for both constructing new models of meaning and formalising already existing ones.

$$\text{Syntax} \xrightarrow{\text{Functor}} \text{Semantics}$$

In this chapter, we use functorial semantics to characterise the expressivity and complexity of a number of NLP models, including logical, tensor network, quantum and neural network models, while showing how they are implemented in DisCoPy. We start in 2.1 by showing how to implement semantic categories in Python, while introducing the two main concrete categories we are interested in, $\mathbf{Mat}_{\mathbb{S}}$ and **Set**, implemented respectively with the Python classes `Tensor` and `Function`. In 2.2, we show that Montague semantics is captured by functors from a biclosed grammar to the category of sets and functions **Set**, through a lambda calculus for manipulating first-order logic formulas. We then give an implementation of Montague semantics by defining currying and uncurrying methods for `Function`. In 2.3, we show that recurrent and recursive neural network models are functors from regular and monoidal grammars (respectively) to a category **NN** of neural network architectures. We illustrate this by building an interface between DisCoPy and Tensorflow/Keras [Cho+15]. In 2.4, we show that functors from pregroup grammars to the category of relations allow to translate natural language sentences into conjunctive queries for a relational database. In 2.5, we formalise the relationship between DisCoCat and tensor networks and use it to derive complexity results for DisCoCat models. In 2.7, we study the complexity of our recently proposed quantum models for NLP [Coe+20; Mei+20]. We show how tensor-based models are implemented in just a few lines of DisCoPy code, and we use them to solve a knowledge embedding task 2.8.

2.1 Concrete categories in Python

We describe the implementation of the main semantic modules in DisCoPy: `tensor` and `function`. These consists in classes `Tensor` and `Function` whose methods carry out numerical computation. `Diagram` may then be evaluated using `Functor`. We may start by considering the *monoid*, a set with a unit and a product, or equivalently a category with one object. We can implement it as a subclass of `cat.Box` by overriding `init`, `repr`, `then` and `id`. In fact, it is sufficient to provide an additional `tensor` method and we can make `Monoid` a subclass of `monoidal.Box`. Both `then` and `tensor` are interpreted as multiplication, `id` as the unit.

Listing 2.1.1. Delooping of a monoid as `monoidal.Box`.

```
from discopy import monoidal
from discopy.monoidal import Ty
from numpy import prod

class Monoid(monoidal.Box):
    def __init__(self, m):
        self.m = m
        super().__init__(m, Ty(), Ty())

    def __repr__(self):
        return "Monoid({})".format(self.m)

    def then(self, other):
        if not isinstance(other, Monoid):
            raise ValueError
        return Monoid(self.m * other.m)

    def tensor(self, other):
        return Monoid(self.m * other.m)

    def __call__(self, *others):
        return Monoid(prod([self.m] + [other.m for other in others]))

    @staticmethod
    def id(x):
        if x != Ty():
            raise ValueError
        return Monoid(1)

assert Monoid(2) @ Monoid.id(Ty()) >> Monoid(5) @ Monoid(0.1) == Monoid(1.0)
assert Monoid(2)(Monoid(1), Monoid(4)) == Monoid(8)
```

Remark 2.1.2. *We define semantic classes as subclasses of* `monoidal.Box` *in order for them to inherit the usual DisCoPy syntax. This can be avoided by explicitly providing the definitions* `__matmul__`*, right and left* `__shift__` *as well as the dataclass methods.*

A weighted context-free grammar (WCFG) is a CFG where every production rule is assigned a *weight*, scores are assigned to derivations by multiplying the weights

of each production rule appearing in the tree. This model is equally expressive as probabilistic CFGs and has been applied to range of parsing and tagging tasks [SJ07]. WCFGs are simply functors from `Tree` into the `Monoid` class! Thus we can define weighted grammars as a subclass of `monoidal.Functor`.

Listing 2.1.3. Weighted grammars as `Functor`.

```
from discopy.monoidal import Functor, Box, Id

class WeightedGrammar(Functor):
    def __init__(self, ar):
        ob = lambda x: Ty()
        super().__init__(ob, ar, ar_factory=Monoid)

weight = lambda box: Monoid(0.5)\
    if (box.dom, box.cod) == (Ty('N'), Ty('A', 'N')) else Monoid(1.0)

WCFG = WeightedGrammar(weight)
A = Box('A', Ty('N'), Ty('A', 'N'))
tree = A >> Id(Ty('A')) @ A
assert WCFG(tree) == Monoid(0.25)
```

We can now generate trees with NLTK and evaluate them in a weighted CFG.

Listing 2.1.4. Weighted context-free grammar.

```
from discopy.operad import from_nltk, tree2diagram
from nltk import CFG
from nltk.parse import RecursiveDescentParser
grammar = CFG.fromstring("""
S -> VP NP
NP -> D N
VP -> N V
N -> A N
V -> 'crossed'
D -> 'the'
N -> 'Moses'
A -> 'Red'
N -> 'Sea'""")

rd = RecursiveDescentParser(grammar)
parse = next(rd.parse('Moses crossed the Red Sea'.split()))
diagram = tree2diagram(from_nltk(parse))
parse2 = next(rd.parse('Moses crossed the Red Red Red Sea'.split()))
diagram2 = tree2diagram(from_nltk(parse2))
assert WCFG(diagram).m > WCFG(diagram2).m
```

Functors into `Monoid` are degenerate examples of a larger class of models called `Tensor` models. An instance of `Monoid` is in fact a `Tensor` with domain and codomain of dimension 1. In `Tensor` models of language, words and production rules are upgraded from just carrying a weight to carrying a tensor. The tensors are multiplied according to the structure of the diagram.

2.1.1 Tensor

Tensors are multidimensional arrays of numbers that can be multiplied along their indices. The `tensor` module of DisCoPy comes with interfaces with `numpy` [Har+20], `tensornetwork` [Rob+19] and `pytorch` [Pas+19] for efficient tensor contraction as well as `simpy` [Meu+17] and `jax` [Bra+18] for computing gradients symbolically and numerically. We describe the implementation of the semantic class `Tensor`. We give a more in-depth example in 2.8, after covering the relevant theory in 2.4 and 2.5.

A semiring is a set $\mathbb{S}$ equipped with two binary operations $+$ and $\cdot$ called addition and multiplication, and two specified elements $0, 1$ such that $(\mathbb{S}, +, 0)$ is a commutative monoid, $(\mathbb{S}, \cdot, 1)$ is a monoid, the multiplication distributes over addition:

$$a \cdot (b + c) = a \cdot b + a \cdot c \qquad (a + b) \cdot c = a \cdot c + b \cdot c$$

and multiplication by 0 annihilates: $a \cdot 0 = 0 = 0 \cdot a$ for all $a, b, c \in \mathbb{S}$. We say that $\mathbb{S}$ is commutative when $a \cdot b = b \cdot a$ for all $a, b \in \mathbb{S}$. Popular examples of semirings are the booleans $\mathbb{B}$, natural numbers $\mathbb{N}$, positive reals $\mathbb{R}^+$, reals $\mathbb{R}$ and complex numbers $\mathbb{C}$.

The axioms of a semiring are the minimal requirements to define matrix multiplication and thus a category $\mathbf{Mat}_\mathbb{S}$ with objects natural numbers $n, m \in \mathbb{N}$ and arrows $n \to m$ given by $n \times m$ matrices with entries in $\mathbb{S}$. Composition is given by matrix multiplication and identities by the identity matrix. For any commutative semiring $\mathbb{S}$ the category $\mathbf{Mat}_\mathbb{S}$ is monoidal with tensor product $\otimes$ given by the kronecker product of matrices. Note that $\mathbb{S}$ must be *commutative* in order for $\mathbf{Mat}_\mathbb{S}$ to be monoidal, since otherwise the interchanger law wouldn't hold. When $\mathbb{S}$ is non-commutative, $\mathbf{Mat}_\mathbb{S}$ is a premonoidal category [PR97].

Example 2.1.5. *The category of finite sets and relations* **FRel** *is isomorphic to* $\mathbf{Mat}_\mathbb{B}$. *The category of finite dimensional real vector spaces and linear maps is isomorphic to* $\mathbf{Mat}_\mathbb{R}$. *The category of finite dimensional Hilbert spaces and linear maps is isomorphic to* $\mathbf{Mat}_\mathbb{C}$.

Matrices $f : 1 \to n_0 \otimes \cdots \otimes n_k$ for objects $n_i \in \mathbb{N}$ are usually called *tensors* with k indices of dimensions n_i. The word tensor emphasizes that this is a k dimensional array of numbers, while matrices are usually thought of as 2 dimensional. Thus $\mathbf{Mat}_\mathbb{S}$ can be thought of as a category of tensors with specified input and output dimensions. This gives rise to more categorical structure. $\mathbf{Mat}_\mathbb{S}$ forms a hypergraph category with Frobenius structure $(\mu, \nu, \delta, \epsilon)$ given by the "generalised Kronecker delta" tensors, defined in Einstein's notation by:

$$\mu^k_{i,j} = \delta^{j,k}_i = \begin{cases} 1 & \text{if } i = j = k \\ 0 & \text{otherwise} \end{cases} \qquad \nu^i = \epsilon_i = 1 \tag{2.1}$$

In particular, $\mathbf{Mat}_\mathbb{S}$ is compact closed with cups and caps given by $\mu\epsilon$ and $\delta\nu$. The transpose f^* of a matrix $f : n \to m$ is obtained by pre and post composing with cups and caps. When the semiring $\mathbb{S}$ is involutive, $\mathbf{Mat}_\mathbb{S}$ has moreover a dagger structure, i.e. an involutive identity on objects contravariant endofunctor (see the nlab). In the

case when $\mathbb{S} = \mathbb{C}$ this is given by taking the conjugate transpose, corresponding to the dagger of quantum mechanics.

The class `Tensor` implements the category of matrices in `numpy` [Har+20], with matrix multiplication as `then` and kronecker product as `tensor`. The categorical structure of $\mathbf{Mat}_{\mathbb{S}}$ translates into methods of the class `Tensor`, as listed below. The types of the category of tensors are given by tuples of dimensions, each entry corresponding to a wire. We can implement them as a subclass of `rigid.Ty` by overriding `__init__`.

Listing 2.1.6. Dimensions, i.e. types of Tensors.

```
class Dim(Ty):
    @staticmethod
    def upgrade(old):
        return Dim(*[x.name for x in old.objects])

    def __init__(self, *dims):
        dims = map(lambda x: x if isinstance(x, monoidal.Ob) else Ob(x), dims)
        dims = list(filter(lambda x: x.name != 1, dims))  # Dim(1) == Dim()
        for dim in dims:
            if not isinstance(dim.name, int):
                raise TypeError(messages.type_err(int, dim.name))
            if dim.name < 1:
                raise ValueError
        super().__init__(*dims)

    def __repr__(self):
        return "Dim({})".format(', '.join(map(repr, self)) or '1')

    def __getitem__(self, key):
        if isinstance(key, slice):
            return super().__getitem__(key)
        return super().__getitem__(key).name

    @property
    def l(self):
        return Dim(*self[::-1])

    @property
    def r(self):
        return Dim(*self[::-1])
```

A `Tensor` is initialised by domain and codomain `Dim` types and an array of shape `dom @ cod`. It comes with methods `then`, `tensor` for composing tensors in sequence and in parallel. These matrix operations can be performed using `numpy`, `jax.numpy` [Bra+18] or `pytorch` [Pas+19] as backend.

Listing 2.1.7. The category of tensors with `Dim` as objects.

```
import numpy as np
```

```
class Tensor(rigid.Box):
    def __init__(self, dom, cod, array):
        self._array = Tensor.np.array(array).reshape(tuple(dom @ cod))
        super().__init__("Tensor", dom, cod)

    @property
    def array(self):
        return self._array

    def then(self, *others):
        if self.cod != other.dom:
            raise AxiomError()
        array = Tensor.np.tensordot(self.array, other.array, len(self.cod))\
            if self.array.shape and other.array.shape\
            else self.array * other.array
        return Tensor(self.dom, other.cod, array)

    def tensor(self, others):
        dom, cod = self.dom @ other.dom, self.cod @ other.cod
        array = Tensor.np.tensordot(self.array, other.array, 0)\
            if self.array.shape and other.array.shape\
            else self.array * other.array
        source = range(len(dom @ cod))
        target = [
            i if i < len(self.dom) or i >= len(self.dom @ self.cod @ other.dom)
            else i - len(self.cod) if i >= len(self.dom @ self.cod)
            else i + len(other.dom) for i in source]
        return Tensor(dom, cod, Tensor.np.moveaxis(array, source, target))

    def map(self, func):
        return Tensor(
            self.dom, self.cod, list(map(func, self.array.flatten())))

    @staticmethod
    def id(dom=Dim(1)):
        from numpy import prod
        return Tensor(dom, dom, Tensor.np.eye(int(prod(dom))))

    @staticmethod
    def cups(left, right):
        ...

    @staticmethod
    def caps(left, right):
        ...

    @staticmethod
    def swap(left, right):
        array = Tensor.id(left @ right).array
        source = range(len(left @ right), 2 * len(left @ right))
        target = [i + len(right) if i < len(left @ right @ left)
                  else i - len(left) for i in source]
        return Tensor(left @ right, right @ left,
                      Tensor.np.moveaxis(array, source, target))
```

```
Tensor.np = np
```

The compact-closed structure of the category of matrices is implemented via static methods `cups` and `caps` and `swap`. We check the axioms of compact closed categories (1.5.12) on a `Dim` object.

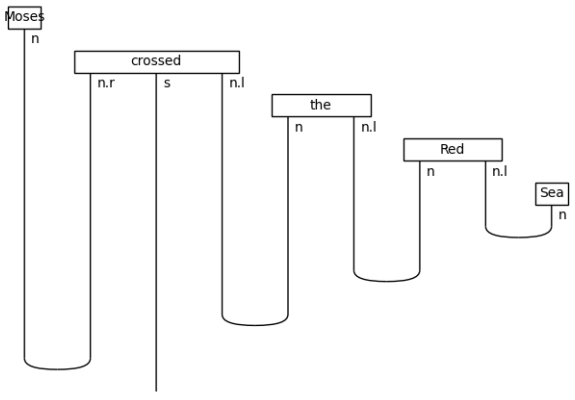
Listing 2.1.8. Axioms of compact closed categories.

```
from discopy import Dim, Tensor
import numpy as np

x = Dim(3, 2)
cup_r, cap_r = Tensor.cups(x, x.r), Tensor.caps(x.r, x)
cup_l, cap_l = Tensor.cups(x.l, x), Tensor.caps(x, x.l)
snake_r = Tensor.id(x) @ cap_r >> cup_r @ Tensor.id(x)
snake_l =  cap_l @ Tensor.id(x) >> Tensor.id(x) @ cup_l
assert np.allclose(snake_l.array, Tensor.id(x).array, snake_r.array)

swap = Tensor.swap(x, x)
assert np.allclose((swap >> swap).array, Tensor.id(x @ x).array)
assert np.allclose((swap @ Tensor.id(x) >> Tensor.id(x) @ swap).array,
                   Tensor.swap(x, x @ x).array)
```

Functors into `Tensor` allow to evaluate any DisCoPy diagram as a tensor network. They are simply `monoidal.Functor`s with codomain `(Dim, Tensor)`, initialised by a mapping `ob` from `Ty` to `Dim` and a mapping `ar` from `monoidal.Box` to `Tensor`. We can use them to give an example of a DisCoCat model [CCS10] in DisCoPy, these are studied in 2.4 and 2.5 and used for a concrete task in 2.8.

Listing 2.1.9. DisCoCat model as `Functor`.

```
from lambeq import BobcatParser
parser = BobcatParser()
diagram = parser.sentence2diagram('Moses crossed the Red Sea')
diagram.draw()
```



```
from discopy.tensor import Dim, Tensor, Functor
import numpy as np

def box_to_array(box):
    if box.name =='Moses':
        return np.array([1, 0])
    if box.name == 'Sea':
        return np.array([0, 1])
    if box.name == 'crossed':
        return np.array([[0, 1], [1, 0]])
    if box.name in ['the', 'Red']:
        return np.eye(2)
    raise NotImplementedError()

ob = lambda x: 2 if x == n else 1
F = Functor(ob, box_to_array)
assert F(diagram).array == [1.0]
```

2.1.2 Function

`Function` is the semantic class that characterises the models studied in 2.2 and 2.3. It consists in an implementation of the category **Set** of sets and functions, or more precisely, the category of Python functions on tuples with `Ty` as objects. We describe the basic methods of `Function`, that arise from the cartesian structure of the category of functions. In 2.2, we also define `curry` and `uncurry` methods for `Function`, accounting for the biclosed structure of this category. Alexis Toumi [Tou22] gives detailed implementation and examples for this class.

Set is a monoidal category with the cartesian product $\times : \mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$ as tensor. It is moreover symmetric, there is a natural transformation $\sigma_{A,B} : A \times B \to B \times A$ satisfying $\sigma_{B,A} \circ \sigma_{A,B} = \mathtt{id}_{A\times B}$ and the axioms 1.16. **Set** is a *cartesian* category.

Definition 2.1.10 (Cartesian category)**.** *A cartesian category* $\mathbf{C}$ *is a symmetric monoidal category such that the product* $\times$ *satisfies the following properties:*

1. *there are projections* $A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$ *for any* $A, B \in \mathbf{C}_0$,
2. *any pair of function* $f : C \to A$ *and* $g : C \to B$ *induces a unique function* $< f, g >: C \to A \times B$ *with* $\pi_1 \circ < f, g >= f$ *and* $\pi_2 \circ < f, g >= g$.

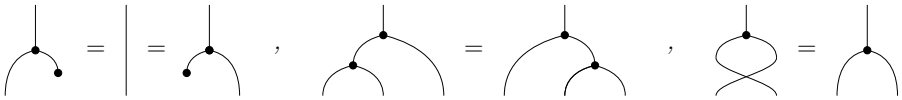
The structure of the category of functions is orthogonal to the structure of tensors. This may be stated formally as in the following proposition, shown in the context of the foundations of quantum mechanics.

Proposition 2.1.11. *[Abr12] A compact-closed category that is also cartesian is trivial, i.e. there is at most one morphism between any two objects.*

The main difference comes from the presence of the diagonal map `copy` in **Set**. This is a useful piece of structure which exists in any cartesian category.

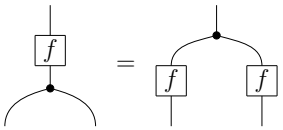
Proposition 2.1.12 (Fox [Fox76]). *In any cartesian category* $\mathbf{C}$ *there is a natural transformation* $\texttt{copy}_A : A \to A \otimes A$ *satisfying the following axioms:*

1. *Commutative comonoid axioms:*



(2.2)

2. *Naturality of copy:*



(2.3)

This proposition may be used to characterise the *free cartesian category* $\mathbf{Cart}(\Sigma)$ from a generating monoidal signature Σ as the free monoidal category with natural comonoids on every object. These were first studied by Lawvere [Law63] who used them to define algebraic theories as functors.

Definition 2.1.13 (Lawvere theory). *A Lawvere theory with signature* Σ *is a product-preserving functor* $F : \mathbf{Cart}(\Sigma) \to \mathbf{Set}$.

We now show how to implement these concepts in Python. A `Function` is initialised by a domain `dom` and a codomain `cod` together with a Python function `inside` which takes tuples of length `len(dom)` to tuples of length `len(cod)`. The class comes with methods `id`, `then` and `tensor` for identities, sequential and parallel composition of functions, as well as a `__call__` method which accesses `inside`.

Listing 2.1.14. The category of Python functions on tuples.

```
class Function(monoidal.Box):
    def __init__(self, inside, dom, cod):
        self.inside = inside
        name = "Function({}, {}, {})".format(inside, dom, cod)
        super().__init__(name, dom, cod)

    def then(self, other):
        inside = lambda *xs: other(*tuple(self(*xs)))
        return Function(inside, self.dom, other.cod)

    def tensor(self, other):
        def inside(*xs):
            left, right = xs[:len(self.dom)], xs[len(self.dom):]
            result = tuple(self(*left)) + tuple(other(*right))
            return (result[0], ) if len(self.cod @ other.cod) == 1 else result
        return Function(inside, self.dom @ other.dom, self.cod @ other.cod)

    def __call__(self, *xs): return self.inside(*xs)

    @staticmethod
    def id(x):
        return Function(lambda *xs: xs, x, x)
```

```
    @staticmethod
    def copy(x):
        return Function(lambda *xs: (*xs, *xs), x, x @ x)

    @staticmethod
    def delete(x):
        return Function(lambda *xs: (), x, Ty())

    @staticmethod
    def swap(x, y):
        return Function(lambda x0, y0: (y0, x0), x @ y, y @ x)
```

We can check the properties of diagonal maps and projections.

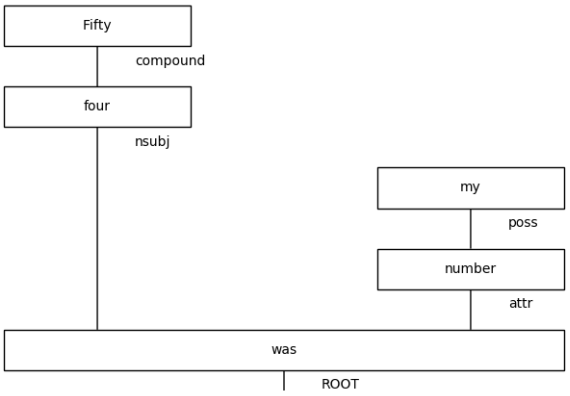
Listing 2.1.15. Axioms of cartesian categories.

```
X = Ty('X')
copy = Function.copy(X)
delete = Function.delete(X)
I = Function.id(X)
swap = Function.swap(X, X)

assert (copy >> copy @ I)(54) == (copy >> I @ copy)(54)
assert (copy >> delete @ I)(46) == (copy >> I @ delete)(46)
assert (copy >> swap)('was my number') == (copy)('was my number')

f = Function(lambda x: (46,) if x == 54 else (54,), X, X)
assert (f >> copy)(54) == (copy >> f @ f)(54)
assert (copy @ copy >> I @ swap @ I)(54, 46) == Function.copy(X @ X)(54, 46)
```

This is all we need in order to interpret diagrams as functions! Indeed, it is sufficient to use an instance of `monoidal.Functor`, with codomain `cod = (Ty, Function)`. We generate a diagram using the interface with SpaCy and we evaluate its semantics with a `Functor`.

Listing 2.1.16. Lawvere theory as a `Functor`

```
from discopy.operad import from_spacy, tree2diagram
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("Fifty four was my number")
diagram = tree2diagram(from_spacy(doc), contravariant=True)
diagram.draw()
```




```
from discopy.monoidal import Ty, Id, Box, Functor

X = Ty('X')
ob = lambda x: X
def box_to_function(box):
    if box.name == 'was':
        return Function(lambda x, y: (x == y, ), X @ X, X)
    if box.name == 'number':
        return Function.id(X)
    if box.name == 'four':
        return Function(lambda x: (4 + x, ), X, X)
    if box.name == 'Fifty':
        return Function(lambda: (50, ), Ty(), X)
    if box.name == 'my':
        return Function(lambda: (54, ), Ty(), X)
    raise NotImplementedError()

F = Functor(ob, box_to_function, ob_factory=Ty, ar_factory=Function)
assert F(diagram)() == (True,)
```

2.2 Montague models

Montague's work appeared in three papers in the 1970s [Mon70b; Mon70a; Mon73]. In the first, he characterises his endeavour in formulating a mathematically precise theory of natural language semantics: "The basic aim of semantics is to characterize the notion of a true sentence (under a given interpretation) and of entailment" [Mon70b].

On the syntactic side, Montague's grammar can be seen as an instance of the categorial grammars studied in 1.4, see e.g. [MR12a]. On the semantic side, he used a blend of lambda calculus and modal logic allowing him to combine the logical meaning of individual words into the meaning of a sentence. This work had an immediate influence on philosophy and linguistics [Par76; HL79; JZ21]. It motivated much of the work on combinatory categorial grammars (see 1.4.4) and has been used in the implementation of semantic parsers turning natural language into database queries [KM14; AFZ14]. From the point of view of large-scale NLP, Montague models suffer from great complexity issues and making them practical comes at the cost of restricting the possibility of (exact) logical reasoning.

It was first proposed by Lambek [Lam88; Lam99b], that Montague semantics should be seen as a functor from categorial grammars to a cartesian closed category. In this section, we make this idea precise, focusing on the first-order logic aspects of Montague's translation and leaving the intensional and modal aspects for future work. We formulate Montague semantics as a pair of functors:

$$G \longrightarrow \mathbf{CCC}(\Gamma_\Sigma) \longrightarrow \mathbf{Set}$$

where G is a categorial grammar, $\mathbf{CCC}(\Gamma_\Sigma)$ is a lambda calculus for typed first-order-logic and $\mathbf{Set}$ is the category of sets and functions. This factorization allows to distinguish between the syntactic translation from sentences to formulae $G \to \mathbf{CCC}(\Gamma_\Sigma)$ and the evaluation of the resulting formulae in a concrete model $\mathbf{CCC}(\Gamma_\Sigma) \to \mathbf{Set}$.

We start in 2.2.1 by introducing the lambda calculus and reviewing its relationship with cartesian closed categories. In 2.2.2, we define a typed lambda calculus $\mathbf{CCC}(\Gamma_\Sigma)$ for manipulating first-order logic formulae, and in 2.2.3 we define Montague models and discuss their complexity. Finally, in 2.2.4 we show how to implement Montague semantics in DisCoPy, by defining `curry` and `uncurry` methods for the `Function` class introduced in 2.1.

2.2.1 Lambda calculus

The lambda calculus was introduced in the 1930s by Alonzo Church as part of his research into the foundations of mathematics [Chu32; Chu36]. It is a model of computation which can be used to simulate any Turing machine [Tur37]. Church also introduced a simply typed version in [Chu40] which yields a weaker model of computation but allows to avoid the infinite recursions of the untyped calculus. There is a well known correspondence — due to Curry and Howard — between the typed lambda calculus and intuitionistic logic, where types are seen as formulae and terms

as proofs. This correspondence was later extended by Lambek [LS86] who showed that the typed lambda calculus has a clear characterisation as the equational theory of *cartesian closed categories*, viewing programs (or proofs) as morphisms.

The rules of the lambda calculus emerge naturally from the structure of the category of sets and functions **Set**. We have seen in 2.1 that **Set** is a monoidal category with the cartesian product $\times : \mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$. Moreover, for any pair of sets A and B, the hom-set $\mathbf{Set}(A, B)$ is itself a set, denoted B^A or $A \to B$. In fact, there is a functor $-^B : \mathbf{Set} \to \mathbf{Set}$ taking a set A to A^B and a function $f : A \to C$ to a function $f^B : A^B \to C^B$ given by $f^B(g) = f \circ g$ for any $g \in A^B$. This functor is the right adjoint of the cartesian product, i.e. there is a natural isomorphism:

$$\mathbf{Set}(A \times B, C) \simeq \mathbf{Set}(A, C^B)$$

This is holds in any cartesian closed category.

Definition 2.2.1 (Cartesian closed category)**.** *A cartesian closed category* $\mathbf{C}$ *is cartesian category equipped with a functor* $-^A$ *for any object* $A \in \mathbf{C}_0$ *which is the right adjoint of the cartesian product* $A \times - \dashv -^A$*. Explicitly, there is a natural isomorphism:*

$$\mathbf{C}(A \times B, C) \simeq \mathbf{C}(A, C^B) \tag{2.4}$$

Proposition 2.2.2. *A cartesian closed category is a cartesian category (Definition 2.1.10) which is also biclosed (Definition 1.4.2).*

Remark 2.2.3. *The categorial biclosed grammars studied in 1.4 map canonically into the lambda calculus, as we will see in 2.2.3.*

A functional signature is a set Γ together with a function $\mathtt{ty} : \Gamma \to TY(B)$ into the set of functional types defined inductively by:

$$TY(B) \ni T, U \ = \ b \in B \mid T \otimes U \mid T \to U \ .$$

we write $x : T$ whenever $\mathtt{ty}(x) = T$ for $x \in \Gamma$. Given a functional signature Γ, we can consider the free cartesian closed category $\mathbf{CCC}(\Gamma)$ generated by Γ. As first shown by Lambek [LS86], morphisms of the free cartesian closed category over Γ can be characterised as the terms of the simply typed lambda calculus generated by Γ.

We define the lambda calculus generated by the basic types B and a functional signature Γ. Its *types* are given by $TY(B)$. We define the set of *terms* by the following inductive definition:

$$TE \ni t, u \ = \ x \mid tu \mid \lambda x.t \mid \ < t, u > \ \mid \pi_1 u \mid \pi_2 u$$

A *typing context* is just a set of pairs of the form $x : T$, i.e. a functional signature Γ. Then a *typing judgement* is a triple:

$$\Gamma \vdash t : T$$

consisting in the assertion that term t has type T in context Γ. A *typed term* is a term t with a typing judgement $\Gamma \vdash t : T$ which is derivable from the following rules of inference:

$$\frac{}{\Gamma, x : T \vdash x : T} \tag{2.5}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash u : U}{\Gamma \vdash < t, u > : T \times U} \qquad \frac{\Gamma \vdash v : T \times U}{\Gamma \vdash \pi_1 v : T} \qquad \frac{\Gamma \vdash v : T \times U}{\Gamma \vdash \pi_2 v : U} \tag{2.6}$$

$$\frac{\Gamma, x : U \vdash t : T}{\Gamma \vdash \lambda x.t : U \to T} \qquad \frac{\Gamma \vdash t : U \to T \qquad \Gamma \vdash u : U}{\Gamma \vdash tu : T} \tag{2.7}$$

We define the typed lambda terms generated by Γ, denoted $\Lambda(\Gamma)$ as the lambda terms that can be typed in context Γ. In order to define equivalence of typed lambda terms we start by defining β-reduction $\to_\beta$ which is the relation on terms generated by the following rules:

$$(\lambda x.t)u \to_\beta t[u/x] \qquad \pi_1 < t, u > \to_\beta t \qquad \pi_2 < t, u > \to_\beta u \tag{2.8}$$

where $t[u/x]$ is the term obtained by substituting u in place of x in t, see e.g. [AT10] for an inductive definition. We define β-conversion $\sim_\beta$ as the symmetric reflexive transitive closure of $\to_\beta$. Next, we define η-conversion $\sim_\eta$ as the symmetric reflexive transitive closure of the relation defined by:

$$t \sim_\eta \lambda x.tx \qquad v \sim_\eta < \pi_1 v, \pi_2 v > \tag{2.9}$$

Finally λ-conversion, denoted $\sim_\lambda$ is the transitive closure of the union $\sim_\beta \cup \sim_\eta$. One may show that for any typed term $\Gamma \vdash t : T$, if $t \to_\beta t'$ then $\Gamma \vdash t' : T$, and similarly for $\sim_\eta$, so that λ-equivalence is well defined on typed terms. Moreover, β-reduction admits *strong normalisation*, i.e. ever reduction sequence is terminating and leads to a normal form without redexes. For any lambda term t we denote its normal form by $\mathtt{nm}(t)$, which of course satisfies $t \sim_\lambda \mathtt{nm}(t)$. However normalising lambda terms is a problem known to be *not elementary recursive* in general [Sta79]! We discuss the consequences of this result for Montague grammar at the end of the section.

We can now state the correspondence between cartesian closed categories and the lambda calculus [LS86]. For a proof, we refer to the lecture notes [AT10] where this equivalence is spelled out in detail alongside the correspondence with intuitionistic logic.

Proposition 2.2.4. *[AT10, Section 1.6.5] The free cartesian closed category over* Γ *is equivalent to the lambda calculus generated by* Γ*.*

$$\mathbf{CCC}(\Gamma) \simeq \Lambda(\Gamma)/\sim_\lambda$$

Note that a morphism $f : x \to y$ in $\mathbf{CCC}(\Gamma)$ may have several equivalent representations as a lambda term in $\Lambda(\Gamma)$. In the remainder of this section, by $f \in \mathbf{CCC}(\Gamma)$ we will mean any such representation, and we will write explicitly $\mathtt{nm}(f)$ when we want its normal form.

2.2.2 Typed first-order logic

Montague used a blend of lambda calculus and logic which allows to compose the logical meaning of individual words into the meaning of a sentence. For example to intepret the sentence "John walks", Montague would assign to "John" the lambda term $J = \lambda x.\text{John}(x)$ and to "walks" the term $W = \lambda\varphi.\exists x \cdot \varphi(x) \wedge \text{walks}(x)$ so that their composition results in the closed logic formula $\exists x \cdot \text{John}(x) \wedge \text{walks}(x)$. Note that x and φ above are symbols of different type, x is a variable and φ a proposition. The aim of this section is to define a typed lambda calculus for manipulating first-order logic formulae — akin to [LF04] and [BGG12] — which will serve as codomain for Montague's mapping.

We start by recalling the basic notions of first-order logic (FOL). A *FOL signature* $\Sigma = \mathcal{C} + \mathcal{F} + \mathcal{R}$ consist in a set of constant symbol $a, b \in \mathcal{C}$, a set of function symbols $f, g \in \mathcal{F}$ and a set of relational symbols $R, S \in \mathcal{R}$ together with a function $\mathtt{ar} : \mathcal{F} + \mathcal{R} \to \mathbb{N}$ assigning an arity to functional and relational symbols. The terms of first-order logic over Σ are generated by the following context-free grammar:

$$FT(\Sigma) \ni t \quad ::= \quad x \mid a \mid f(\vec{x})$$

where $x \in \mathcal{X}$ for some countable set of variables $\mathcal{X}$, $a \in \mathcal{C}$, $f \in \mathcal{F}$ and $\vec{x} \in (\mathcal{X} \cup \mathcal{C})^{\mathtt{ar}(f)}$. The set of first-order logic formulae over Σ is defined by the following context-free grammar:

$$FOL(\Sigma) \ni \varphi \quad ::= \quad \top \mid t \mid x = x' \mid R(\vec{x}) \mid \varphi \wedge \varphi \mid \exists\, x \cdot \varphi \mid \neg\varphi \tag{2.10}$$

where $t, x, x' \in FT(\Sigma)$, $R \in \mathcal{R}$, $\vec{x} \in FT(\Sigma)^{\mathtt{ar}(R)}$ and $\top$ is the truth symbol. Let us denote the variables of φ by $\mathtt{var}(\varphi) \subseteq \mathcal{X}$ and its free variables by $\mathtt{fv}(\varphi) \subseteq \mathtt{var}(\varphi)$. For any formula φ and $\vec{x} \in FT(\Sigma)^*$, we denote by $\varphi(\vec{x})$ the formula obtained by substituting the terms $x_1 \dots x_n$ in place of the free variables of φ where $n = |\mathtt{fv}(\varphi)|$.

We can now build a typed lambda calculus over the set of FOL formulae.

Definition 2.2.5 (Typed first-order logic)**.** *Given a FOL signature $\Sigma = \mathcal{C} + \mathcal{R}$, we define the typed first order logic over Σ as the free cartesian closed category $\mathbf{CCC}(\Gamma_\Sigma)$ where Γ_Σ is the functional signature with basic types:*

$$B = \{\, X, P \,\}_{n \in \mathbb{N}}$$

where X is the type of terms and P is the type of propositions, and entries given by:

$$\Gamma_\Sigma = \{\, \varphi : P \mid \varphi \in FOL(\Sigma) - FT(\Sigma) \,\} + \{\, x : X \mid x \in FT(\Sigma) \,\}$$

In order to recover a FOL formula from a lambda term $f : T$ in $\mathbf{CCC}(\Gamma_\Sigma)$, we need to normalise it.

Proposition 2.2.6. *For any lambda term $f : P$ in $\mathbf{CCC}(\Gamma_\Sigma)$, the normal form is a first order logic formula $\mathtt{nm}(f) \in FOL(\Sigma)$.*

Proof. □

Note that morphisms $\varphi : P$ in $\mathbf{CCC}(\Gamma_\Sigma)$ are the same as first-order logic formulae $\varphi \in FOL(\Sigma)$, since we have adopted the convention that a morphism in $\mathbf{CCC}(\Gamma)$ is the normal form of its representation as a lambda term in $\Lambda(\Gamma)$.

Example 2.2.7. *Take* $\Sigma = \{ \mathit{John}, \mathit{Mary}, \mathit{walks}, \mathit{loves} \}$ *with* $\mathtt{ar}(\mathit{John}) = \mathtt{ar}(\mathit{Mary}) = \mathtt{ar}(\mathit{walks}) = 1$ *and* $\mathtt{ar}(\mathit{loves}) = 2$*. Then the following are examples of well-typed lambda expressions in* $\mathbf{CCC}(\Gamma_\Sigma)$*:*

$$\lambda\varphi.\exists x \cdot \varphi(x) \wedge \mathit{John}(x) \; : \; P \to P \qquad \lambda x \lambda y.\mathit{loves}(x, y) \; : \; X \times X \to P$$

A model for first-order logic formulae is defined as follows.

Definition 2.2.8 (FOL model)**.** *A model* K *over a FOL signature* Σ*, also called* Σ*-model, is given by a set* U *called the* universe *and an interpretation* $K(R) \subseteq U^{\mathtt{ar}(R)}$ *for every relational symbol* $R \in \mathcal{R}$ *and* $K(a) \in U$ *for every constant symbol* $c \in \mathcal{C}$*. We denote by* $\mathcal{M}_\Sigma$ *the set of* Σ*-models.*

Given a model $K \in \mathcal{M}_\Sigma$ with universe U, let

$$\mathtt{eval}(\varphi, K) = \{ \, v \in U^{\mathtt{fv}(\varphi)} \; \mid \; (K, v) \vDash \varphi \, \}$$

where the satisfaction relation ($\vDash$) is defined by induction over (2.10) in the usual way.

Proposition 2.2.9. *Any model* $K \in \mathcal{M}_\Sigma$ *induces a monoidal functor* $F_K : \mathbf{CCC}(\Gamma_\Sigma) \to \mathbf{Set}$ *such that closed lambda terms* $\varphi : F_0$ *are mapped to their truth value in* K*.*

Proof. By the universal property of free cartesian closed categories, it is sufficient to define F_K on generating objects and arrows. On objects we define:

$$F_K(X) = 1 \qquad F_K(P) = \coprod_{n=0}^{N} \mathcal{P}(U^n)$$

where N is the maximum arity of a symbol in Σ and $\mathcal{P}(U^n)$ is the powerset of U^n. On generating arrows $\varphi : P$ and $x : X$ in Γ, F_k is defined by:

$$F_K(\varphi) = \mathtt{eval}(\varphi, K) \in F_K(P) \qquad F_K(x) = 1$$

□

2.2.3 Montague semantics

We model Montague semantics as a pair of functors $G \to \mathbf{CCC}(\Gamma_\Sigma) \to \mathbf{Set}$ for a grammar G. We have already seen that functors $\mathbf{CCC}(\Gamma_\Sigma) \to \mathbf{Set}$ can be built from Σ-models or relational databases. It remains to study functors $G \to \mathbf{CCC}(\Gamma_\Sigma)$, which we call *Montague models*.

Definition 2.2.10 (Montague model)**.** *A Montague model is a monoidal functor $M : G \to \mathbf{CCC}(\Gamma_\Sigma)$ for a biclosed grammar G and a FOL signature Σ such that $M(s) = P$ and $M(w) = 1$ for any $w \in V \subseteq G_0$. The semantics of a grammatical sentence $g : u \to s$ in $\mathbf{BC}(G)$ is the first order logic formula $\mathtt{nm}(M(w)) \in FOL(\Sigma)$ obtained by normalising the lambda term $M(w) : P$ in $\mathbf{CCC}(\Gamma_\Sigma)$.*

Remark 2.2.11. *Note that since $\mathbf{CCC}(\Gamma_\Sigma)$ is biclosed, it is sufficient to define the image of the lexical entries in G to obtain a Montague model. The structural morphisms* app*,* comp *defined in 1.4.3 have a canonical intepretation in any cartesian closed category given by:*

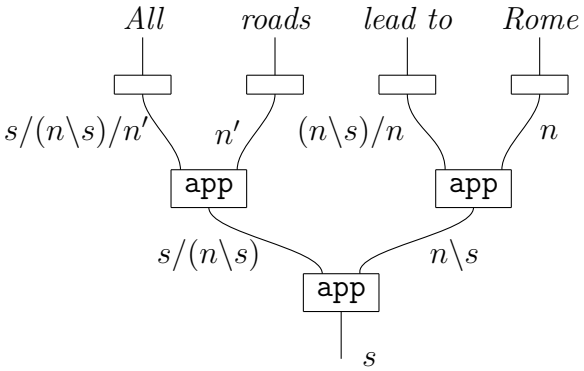
$$\mathtt{app}_{A,B} \mapsto \lambda f.(\pi_2 f)(\pi_1 f) : (A \times (A \to B)) \to B$$

$$\mathtt{comp}_{A,B,C} \mapsto \lambda f \lambda a.(\pi_2 f)(\pi_1 f)a : ((A \to B) \times (B \to C)) \to (A \to C)$$

Example 2.2.12. *Let us consider a Lambek grammar G with basic types $B = \{ n, n', s \}$ and lexicon given by:*

$$\Delta(All) = \{ s/(n\backslash s)/n' \} \quad \Delta(roads) = \{ n' \} \quad \Delta(lead\ to) = \{ (n\backslash s)/n \} \quad \Delta(Rome) = \{ n \}$$

The following is a grammatical sentence in $\mathcal{L}(G)$:



We fix a FOL signature $\Sigma = \mathcal{R} + \mathcal{C}$ with one constant symbol $\mathcal{C} = \{ Rome \}$ and two relational symbols $\mathcal{R} = \{ road, leads\text{-}to \}$ with arity 1 and 2 respectively. We can now define a Montague model M on objects by:

$$M(n) = X \quad M(n') = X \to P \quad M(s) = P$$

and on arrows by:

$$M(All) = \lambda\varphi\lambda\psi.\forall x(\varphi(x) \implies \psi(x)) : (X \to P) \to ((X \to P) \to P)$$

$$M(roads) = \lambda x.road(x) : X \to P \qquad M(Rome) = Rome : X$$

$$M(lead\ to) = \lambda x\lambda y.leads\text{-}to(x, y) : X \to (X \to P)$$

Then the image of the sentence above normalises to the following lambda term:

$$M(All\ roads\ lead\ to\ Rome \to s) = \forall x \cdot (road(x) \implies leads\text{-}to(x, Rome))$$

We are interested in the problem of computing the Montague semantics of sentences generated by a biclosed grammar G.

Definition 2.2.13. $\texttt{LogicalForm}(G)$

Input: $g : u \to s$, Σ, $M : G \to \mathbf{CCC}(\Gamma_\Sigma)$
Output: $\texttt{nm}(M(g)) \in FOL(\Sigma)$

We may split the above problem in two steps. First, given the sentence $g : u \to s$ we are asked to produce the corresponding lambda term $M(g) \in \Lambda(\Gamma_\Sigma)$. Second, we are asked to normalise this lambda term in order to obtain the underlying first-order logic formula in $FOL(\Sigma)$. The first step is easy and may in fact be done in `L` since it is an example of a functorial reduction, see Proposition 1.3.16. The second step may however be exceptionally hard. Indeed, Statman showed that deciding β equivalence between simply-typed lambda terms is not elementary recursive [Sta79]. As a consequence, the normalisation of lambda terms is itself not elementary recursive. We may moreover show that any simply typed lambda term can be obtained in the image of some $M : G \to \mathbf{CCC}(\Gamma_\Sigma)$, yielding the following result.

Proposition 2.2.14. *There are biclosed grammars G such that* $\texttt{LogicalForm}(G)$ *is not elementary recursive.*

Proof. Let G be a biclosed grammar with one generating object a generating arrows $\texttt{swap} : a \otimes a \to a \otimes a$, $\texttt{copy} : a \to a \otimes a$ and $\texttt{discard} : a \to 1$. These generators map canonically in any cartesian category. In fact, there is a full functor $F : \mathbf{BC}(G) \to \mathbf{CCC}$, since all the structural morphisms of cartesian closed categories can be represented in $\mathbf{BC}(G)$. Therefore for any morphism $h \in \mathbf{CCC}$ there is a morphism $f \in \mathbf{BC}(G)$ such that $F(f) = h$. Then for any typed lambda term $g \in \Lambda(\varnothing)$ there is a morphism $f \in \mathbf{BC}(G)$ such that $F(f)$ maps to g under the translation 2.2.4. Therefore normalising g is equivalent to computing $\texttt{nm}(F(f)) = \texttt{LogicalForm}(G)$. Therefore the problem of normalising lambda terms reduces to `LogicalForm`. □

Of course, this proposition is about the worst case scenario, and definitely not about human language. In fact, small to medium scale semantic parsers exist which translate natural language sentences into their logical form using Montague's translation [KM14; AFZ14]. It would be interesting to show that for restricted choices of grammar G (e.g. Lambek or Combinatory grammars), and possibly assuming that the order of the lambda terms assigned to words is bounded (as in [Ter12]), `LogicalForm` becomes tractable.

Even if we are able to extract a logical formula efficiently from natural language, we need to be able to evaluate it in a database in order to compute its truth value. Thus, in order to compute the full-blown montague semantics we need to solve the following problem.

Definition 2.2.15. $\texttt{Montague}(G)$

Input: $g : u \to s$, Σ, $M : G \to \mathbf{CCC}(\Gamma_\Sigma)$, $K \in \mathcal{M}_\Sigma$
Output: $F_K(M(g)) \subseteq U^{\mathtt{fv}(M(g))}$

Note that solving this problem does not necessarily require to solve `LogicalForm`. However, since we are dealing with first-order logic, the problem for a general biclosed grammar is at least as hard as the evaluation of FOL formulae.

Proposition 2.2.16. *There are biclosed grammars G such that* `Montague` *is* `PSPACE`*-hard.*

Proof. It was shown by Vardi that the problem of evaluating $\texttt{eval}(\varphi, K)$ for a first-order logic formula φ and a model K is `PSPACE`-hard [Var82]. Reducing this problem to `Montague` is easy. Take G to be a categorial grammar with a single word w of type s. Given any first-order logic formula φ and model K, we define $M : G \to \mathbf{CCC}(\Gamma_\Sigma)$ by $M(s) = P$ and $M(w) = \varphi$. Then we have that the desired output $F_K(M(w)) = F_K(\varphi) = \texttt{eval}(\varphi, K)$ is the evaluation of φ in K. □

It would be interesting to look at restrictions of this problem, such as fixing the Montague model M, fixing the input sentence $g : u \to s$ and restricting the allowed grammars G. In general however, the proposition above shows that working with full-blown first-order logic is inpractical for large-scale industrial NLP applications. It also shows that models of this kind are the most general and applicable if we can find a compromise between tractability and logical expressivity.

2.2.4 Montague in DisCoPy

We implement Montague models as DisCoPy functors from `biclosed.Diagram` into the class `Function`, as introduced in 1.4 and 2.1 respectively. Before we can do this, we need to upgrade the `Function` class to account for its biclosed structure. The only additional methods we need are `curry` and `uncurry`.

Listing 2.2.17. Curry and UnCurry methods for `Function`.

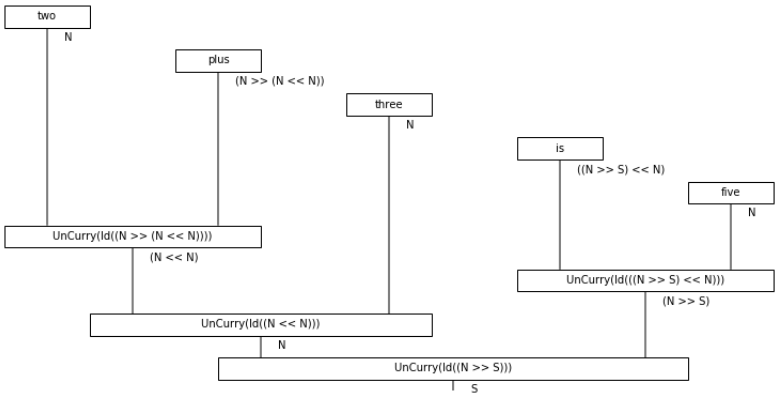
```
class Function:
    ...
    def curry(self, n_wires=1, left=False):
            if not left:
                dom = self.dom[:-n_wires]
                cod = self.cod << self.dom[-n_wires:]
                inside = lambda *xl: (lambda *xr: self.inside(*(xl + xr)),)
                return Function(inside, dom, cod)
            else:
                dom = self.dom[n_wires:]
                cod = self.dom[:n_wires] >> self.cod
                inside = lambda *xl: (lambda *xr: self.inside(*(xl + xr)),)
                return Function(inside, dom, cod)

    def uncurry(self):
        if isinstance(self.cod, Over):
            left, right = self.cod.left, self.cod.right
            cod = left
            dom = self.dom @ right
            inside = lambda *xs: self.inside(*xs[:len(self.dom)])[0](*xs[len(self.dom):])
```

```
            return Function(inside, dom, cod)
        elif isinstance(self.cod, Under):
            left, right = self.cod.left, self.cod.right
            cod = right
            dom = left @ self.dom
            inside = lambda *xs: self.inside(*xs[len(left):])[0](*xs[:len(left)])
            return Function(inside, dom, cod)
        return self
```

We can now map biclosed diagrams into functions using `biclosed.Functor`. We start by initialising a sentence with its categorial grammar parse.

Listing 2.2.18. Example parse from a categorial grammar

```
from discopy.biclosed import Ty, Id, Box, Curry, UnCurry

N, S = Ty('N'), Ty('S')
two, three, five = Box('two', Ty(), N), Box('three', Ty(), N), Box('five', Ty(), N)
plus, is_ = Box('plus', Ty(), N >> (N << N)), Box('is', Ty(), N >> S << N)

FA = lambda a, b: UnCurry(Id(a >> b))
BA = lambda a, b: UnCurry(Id(b << a))

sentence = two @ plus @ three @ is_ @ five
grammar = FA(N, N << N) @ Id(N) @ BA(N, N >> S) >> BA(N, N) @ Id(N >> S) >> FA(N, S)
sentence = sentence >> grammar
sentence.draw()
```



We can now evaluate this sentence in a Montague model defined as a biclosed functor.

Listing 2.2.19. Evaluating a sentence in a Montague model.

```
from discopy.biclosed import Functor

number = lambda y: Function(lambda: (y, ), Ty(), N)
add = Function(lambda x, y: (x + y,), N @ N, N)
equals = Function(lambda x, y: (x == y, ), N @ N, S)
```

```
ob = lambda x: x
ar = {two: number(2), three: number(3), five: number(5),
      is_: equals.curry().curry(left=True),
      plus: add.curry().curry(left=True)}
Montague = Functor(ob, ar, ob_factory=Ty, ar_factory=Function)
assert Montague(sentence)() == (True,)
```

2.3 Neural network models

In the last couple of decades, neural networks have become ubiquitous in natural language processing. They have been used successfully in a wide range of tasks including language modelling [DBM15], machine translation [BCB14; Pop+20], parsing [JM08], question answering [Zho+17] and sentiment analysis [Soc+13a; KGB14].

In this section we show that neural network models can be formalised as functors from a grammar G to the category $\mathbf{Set}_{\mathbb{R}}$ of Euclidean spaces and functions via a category **NN** of neural network architectures. This includes feed-forward, recurrent and recursive neural networks and we discuss how the recent attention mechanisms could be formalised in this diagrammatic framework. At each step, we show how these neural architectures are used to solve concrete tasks such as sentence classification, language modelling, sentiment analysis and machine translation. We keep a level of informality in describing the learning process, an aspect which we will further explore in Chapter 3. We end the section by building an interface between DisCoPy and Tensorflow/Keras neural networks [Cho+15; Mar+15], by defining a class `Network` that allows for composing and tensoring Keras models.

2.3.1 Feed-forward networks

The great advantage of neural networks comes from their ability to simulate any function on Euclidean spaces, a series of results known as universal approximation theorems [TKG03; OK19]. We will thus give them semantics in the category $\mathbf{Set}_{\mathbb{R}}$ where morphisms are functions acting on Euclidean spaces. Note that $\mathbf{Set}_{\mathbb{R}}$ is monoidal with product $\oplus$ defined on objects as the direct sum of Euclidean spaces $\mathbb{R}^n \oplus \mathbb{R}^m = \mathbb{R}^{n+m}$ and on arrows as the cartesian product $f \oplus g(x, y) = (f(x), g(y))$. To our knowledge, $\mathbf{Set}_{\mathbb{R}}$ doesn't have much more structure than this. We will in fact not be able to interpret

In a typical supervised machine learning problem one wants to approximate an unknown function $f : \mathbb{R}^n \to \mathbb{R}^m$ given a dataset of pairs $D = \{(x, f(x)) | x \in \mathbb{R}^n,\ f(x) \in \mathbb{R}^m\}$. Neural networks are parametrized functions built from the following basic processing units:

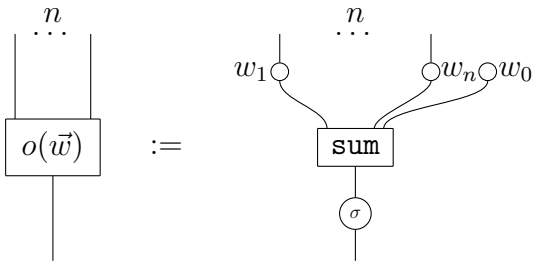
1. sum $\texttt{sum} : n \to 1$ for $n \in \mathbb{N}$,
2. weights $\{w : 1 \to 1\}_{w \in W_0}$,
3. biases $\{r : 0 \to 1\}_{r \in W_1}$
4. and activation $\sigma : 1 \to 1$.

where $W = W_0 + W_1$ is a set of variables. These generate a free cartesian category

$$\mathbf{NN} = \mathbf{Cart}(W + \{\texttt{sum}, \sigma\})$$

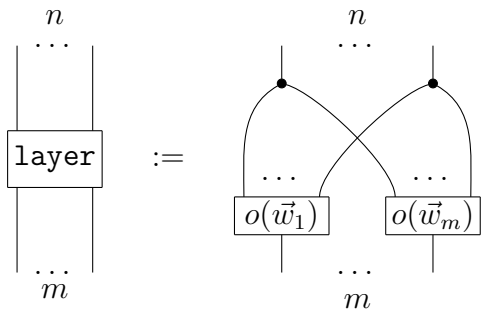
where morphisms are the diagrams of neural network architectures. For example, a neuron with n inputs, bias $w_0 \in W$ and weights $\vec{w} \in W^*$ is given by the following

diagram in **NN**.



(2.11)

"Deep" networks are simply given by composing these neurons in parallel forming a *layer*:



and then stacking layers one on top of the other, as pasta sheets on a lasagna.

Fix a neural network architecture $K : n \to m$ in **NN**. Given a choice of parameters $\theta : W \to \mathbb{R}$, the network K induces a function $I_\theta(K) : \mathbb{R}^n \to \mathbb{R}^m$ called an *implementation*. I_θ is in fact a monoidal functor $\mathbf{NN} \to \mathbf{Set}_{\mathbb{R}}$ defined on objects by $I_\theta(1) = \mathbb{R}$, and on arrows by:

1. $I_\theta(\texttt{sum}) = \{(x_1, \ldots, x_n) \mapsto \sum_{i=1}^n x_i\}$,
2. $I_\theta(w) = \{x \mapsto \theta(w) \cdot x\}$,
3. $I_\theta(r) = \{() \mapsto r\}$
4. $I_\theta(\sigma)$ is a non-linearity such as sigmoid.

For example, the image of the neuron 2.11 is given by the function $I_\theta(o(\vec{w})) : \mathbb{R}^n \to \mathbb{R}$ defined as follows:

$$I_\theta(o(\vec{w}))(\vec{x}) = \sigma(w_0 + \sum_{i=1}^{n} w_i x_i)$$

In order to learn $f : \mathbb{R}^n \to \mathbb{R}^m$, we choose an architecture $K : n \to m$ and a loss function $l : \mathbb{R}^W \to \mathbb{R}$ that evaluates the choice of parameters against the dataset, such as the mean-squared error $l(\theta) = \sum_i (I_\theta(K)(x_i) - f(x_i))^2$. The aim is to minimize the loss function with respect to the parameters so that $I_\theta(K)$ approximates f as closely as possible. The most reliable way of doing this is to compute the gradient of l w.r.t. θ and descending along the gradient. This boils down to computing the gradient of $I_\theta(K)$ w.r.t θ which itself boils down to computing the gradient of each component of the network. Assuming that $\sigma : \mathbb{R} \to \mathbb{R}$ is a smooth function, the image of the functor I_θ lies in $\mathbf{Smooth} \hookrightarrow \mathbf{Set}_{\mathbb{R}}$, the category of smooth functions on Euclidean spaces. The backpropagation algorithm provides an efficient way of updating the parameters

of the network step-by-step while descending along the gradient of l, see [Cru+21] for a recent categorical treatment of differentiation.

Although neural networks are deterministic functions, it is often useful to think of them as probabilistic models. One can turn the output of a neural network into a probability distribution using the `softmax` function as follows. Given a neural network $K : m \to n$ and a choice of parameters $\theta : W \to \mathbb{R}$ we can compose $I_\theta(K) : \mathbb{R}^m \to \mathbb{R}^n$ with a softmax layer, given by:

$$\texttt{softmax}_n(\vec{x})_i = \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}}$$

Then the output is a normalised vector of positive reals of length n, which yields a distribution over $[n]$ the set with n elements, i.e. softmax has the following type:

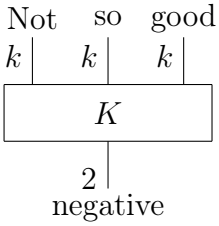
$$\texttt{softmax}_n : \mathbb{R}^n \to \mathcal{D}([n])$$

where $\mathcal{D}(A)$ is the set of probability distributions over A, as defined in 3.1, where softmax is analysed in more detail. In order to simulate a probabilistic process, from a neural network $K : m \to n$, an element of $[n]$ is drawn at random from the induced distribution $\texttt{softmax}_n(I_\theta(K)(x)) \in \mathcal{D}([n])$.

Feed-forward neural networks can be used for *sentence classification*: the general task of assigning labels to pieces of written text. Applications include spam detection, sentiment analysis and document classification. Let $D \subseteq V^* \times X$ be a dataset of pairs (u, x) for u an utterance and x a label. The task is to find a function $f : V^* \to \mathcal{D}(X)$ minimizing a loss function $l(f, D)$ which computes the distance between the predicted labels and the expected ones given by D. For example one may take the mean squared error $l(f, D) = \sum_{(u,x)\in D}(f(u) - |x\rangle)^2$ or the cross entropy loss $-\frac{1}{|D|}\sum_{(u,x)\in D} |x\rangle \cdot \texttt{log}(f(u))$, where $|x\rangle$ is the one-hot encoding of $x \in X$ as a distribution $|x\rangle \in \mathcal{D}(X)$ and $\cdot$ is the inner product. Assuming that every sentence $u \in V^*$ has length at most m and that we have a word embedding $E : V \to \mathbb{R}^k$, we can parametrize the set of functions $\mathbb{R}^{mk} \to \mathbb{R}^{|X|}$ using a neural network $K : mk \to |X|$ and use softmax to get a probability distribution over classes:

$$f(x \mid u) = \texttt{softmax}_k(I_\theta(K)(E^*(u))) \in \mathcal{D}(X)$$

where $u = v_1 v_2 \dots v_m \in V^*$ is an utterance and $E^*(u)$ is the feature vector for u, obtained by concatenating the word embeddings $E^*(u) = \texttt{concat}(E(v_1), E(v_2), \dots, E(v_n))$. In this approach, K plays the role of a black box which takes in a list of words and ouputs a class. For instance in a sentiment analysis task, we can set $X = \{\,\text{positive}, \text{negative}\,\}$ and $m = 3$ and expect that "not so good" is classified as "negative".



A second task we will be interested in is *language modelling*, the task of predicting a word in a text given the previous words. It takes as input a corpus, i.e. a set of strings of words $C \subseteq V^*$, and outputs $f : V^* \to \mathcal{D}(V)$ a function which ouputs a distribution for next word $f(u) \in \mathcal{D}(V)$ given a sequence of previous words $u \in V^*$. Language modelling can be seen as an instance of the classification task, where the corpus C is turned into a dataset of pairs D such that $(u, x) \in D$ whenever ux is a substring of C. Thus language modelling is a *self-supervised* task, i.e. the supervision is not annotated by humans but directly generated from text. Neural networks were first used for language modelling by Bengio et al. [Ben+03], who trained a single (deep) neural network taking a fixed number of words as input and predicting the next, obtaining state of the art results at the time.

For both of these tasks, feed-forward neural networks perform poorly compared to the recurrent architectures which we are about to study. This is because feed-forward networks take no account of the sequential nature of the input.

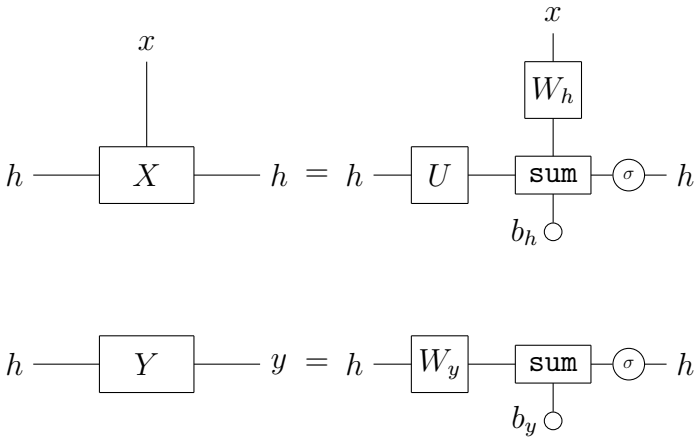
2.3.2 Recurrent networks

The most popular architectures currently used modelling language are *recurrent neural networks* (RNNs). They were introduced in NLP by Mikolov et al. in 2010 [Mik+10] and have since become widely used, see [DBM15] for a survey.

RNNs were introduced by Ellman [Elm90] to process sequence input data. They are defined by the following recursive equations:

$$h_t = \sigma(W_h x_t + U_h h_{t-1} + b_h) \quad y_t = \sigma(W_y h_t + b_y) \tag{2.12}$$

where $t \in \mathbb{N}$ is a time variable, h_t denotes the encoder hidden vector, x_t the input vector, y_t the output, W_h and W_y are matrices of weights and b_h, b_y are bias vectors. We may draw these two components as diagrams in **NN**:

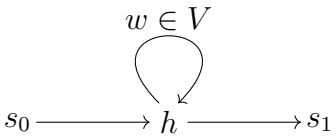


Remark 2.3.1. *The diagrams in the remainder of this section should be read from left/top to bottom/right. One may obtain the corresponding string diagram by bending the left wires to the top and the right wires to the bottom. In the diagram above we have omitted the subscript t since it is determined by the left-to-right reading of the diagram and we use the labels h, x, y for dimensions, i.e. objects of* **NN**.

The networks above are usually called *simple* recurrent networks since they have only one layer. More generally X and Y could be any neural networks of type $X : x \oplus h \to h$ and $Y : h \to y$, often called recurrent network and decoder respectively.

Definition 2.3.2. *A recurrent neural network is a pair of neural networks $X : x \oplus h \to h$ and $Y : h \to y$ in* **NN** *for some dimensions $x, h, y \in \mathbb{N}$.*

The data of RNNs defined above captures precisely the data of a functor from a regular grammar as we proceed to show. Fix a finite vocabulary V and consider the regular grammar RG with three symbols s_0, h, s_1 and transitions $h \xrightarrow{w} h$ for each word $w \in V$.



Note that RG parses any string of words, i.e. the language generated by RG is $\mathcal{L}(RG) = V^*$. The derivations in RG are sequences:

$$s_0 \longrightarrow h \xrightarrow{w_0} h \xrightarrow{w_1} \cdots \xrightarrow{w_k} h \longrightarrow s_1 \tag{2.13}$$

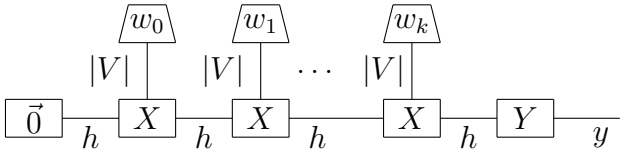
Recurrent neural networks induce functors from this regular grammar to **NN**.

Proposition 2.3.3. *Recurrent neural networks $(X : x \oplus h \to h, Y : h \to y)$ induce functors $RN : RG \to$ **NN** such that $|V| = x$, $RN(h) = h$, $RN(s_1) = y$, $RN(s_0) = 0$ and $RN(s_0 \to h) = \vec{0}$.*

Proof. Given an RNN (X, Y) we can build a functor with $RN(h \xrightarrow{w} h) = (|w\rangle \oplus \mathtt{id}_h) \cdot X$ where $|w\rangle$ is the one-hot encoding of word $w \in V$ as a vector of dimension $n = |V|$, and $RN(h \to s_0) = Y$. □

Remark 2.3.4. *Note that a functor $RN : RG \to$ **NN** induces a finite family of neural networks $RN(h \xrightarrow{w} h)$ indexed by $w \in V$. We do not think that it is always possible to construct a recurrent network $X : |V| \oplus RN(h) \to RN(h)$ such that $RN(h \xrightarrow{w} h) = (|w\rangle \oplus \mathtt{id}_n) \cdot X$, hinting that functors $RN : RG \to$ **NN** are a larger class of processes than RNNs, but we were unable to find a counterexample.*

Given a recurrent neural network $RN : G \to$ **NN**, the image under RN of the derivation 2.13 is the following diagram in **NN**.

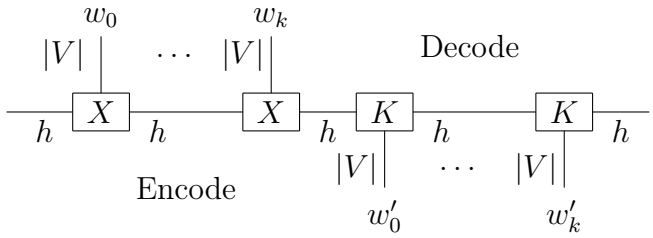


It defines a network that runs through the input words w_i with the recurrent network and uses the decoder to output a prediction of dimension y.

In order to classify sentences in a set of classes X, one may choose $y = |X|$, so that the implementation of the diagram above is a vector of size $|X|$ giving scores for the likelihood that $w_0 w_1 \dots w_k$ belongs to class $c \in X$.

For language modelling, we may set $y = |V|$. Given a string of words $u \in V^*$ with derivation $g_u : s_0 \to s_1 \in \mathbf{C}(RG)$ and an implementation $I_\theta : \mathbf{NN} \to \mathbf{Set}_\mathbb{R}$, we can compute the distribution over the next words $\texttt{softmax}(I_\theta(RN(g_u))) \in \mathcal{D}\,|V|$.

Another task that recurrent neural networks allow to tackle is *machine translation*. This is done by composing a encoder recurrent network $X : |V| \oplus h \to h$ with a decoder recurrent network $K : h \to h \oplus |V|$, as in the following diagram.



(2.14)

One of the drawbacks of standard RNNs is that they don't have memory. For instance in the translation architecture above, the last encoder state is unfolded into all the output decoder states. This often results in a translation that loses accuracy as the sentences grow bigger. *Long-short term memory* networks (LSTMs) were introduced in order to remedy this problem [HS97]. They are a special instance of recurrent neural networks, where the encoder states are split into a *hidden state* and a *cell state*. The cell state allows to store information for longer durations and their architecture was designed to avoid vanishing gradients [GSC00]. This makes them particularly suited to NLP applications, see [OMK19] for example applications.

Moreover, RNNs are well suited for processing data coming in sequences but they fail to capture more structured input data. If we want to process syntax trees and more complex grammatical structures, what we need is a recursive neural network.

2.3.3 Recursive networks

Recursive neural networks (RvNN) generalise RNNs by allowing neural networks to recur over complex structures. They were introduced in the 1990s by Goller and Kuchler [GK96] for the classification of logical terms, and further generalised by Sperduti et al. [SS97; FGS98] who pioneered their applications in chemistry [Bia+00; MSS04]. RvNNs were introduced in NLP by Socher et al. [Soc+13a], who obtained state-of-the-art results in the task of sentiment analysis using tree-shaped RvNNs. They have since been applied to several tasks including word sense disambiguation [CK15] and logical entailment [BPM15].

In [SS97], Sperduti and Starita provide a structured way of mapping labelled directed graphs onto neural networks. The main difficulty in their formalisation appears when the graph in the domain has cycles, in which case they give a procedure for unfolding it into a directed acyclic graph. We review their formalisation for the case of directed acyclic graphs (DAGs). A DAG is given by a set of vertices N and a set of edges $E \subseteq N \times N$ such that the transitive closure of E does not contain loops. In any

DAG (N, E), the parents of a vertex v are defined by $\mathtt{pa}(v) = \{ v' \in N \mid (v', v) \in E \}$ and the children by $\mathtt{ch}(v) = \{ v' \in N \mid (v, v') \in E \}$. A Σ-labelled DAG is a DAG (N, E) with a labelling function $\varphi : N \to \Sigma$.

Suppose we have a set X of Σ-labelled DAGs which we want to classify over k classes, given a dataset of pairs $D \subseteq X \times [k]$. The naive way to do this is to encode every graph in D as a vector of fixed dimension n and learning the parameters of a neural network $K : n \to k$. Sperduti and Starita propose instead to assign a *recursive neuron* to each vertex of the graph, and connecting them according to the topological structure of the graph. Given a labelled DAG (N, E, φ) with an assignment $\theta : E + \Sigma \to \mathbb{R}$ of weights to every edge in E and every label in Σ, the recursive neuron assigned to a vertex v in a DAG (N, E) is the function defined by the following recursive formula:

$$o(v) = \sigma(\theta(\varphi(v)) + \sum_{v' \in \mathtt{pa}(v)} \theta(v', v) o(v')) \tag{2.15}$$

Note that this is a simplification of the formula given by Sperduti et al. where we assume that a single weight is assigned to every edge and label, see [SS97] for details. They further assume that the graph has a *sink*, i.e. a vertex which can be reached from any other vertex in the graph. This allows to consider the output of the neuron assigned to the supertarget as the score of the graph, which they use for classification.

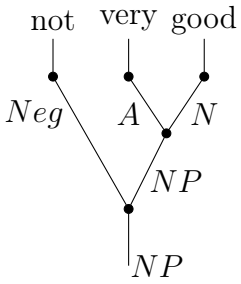
We show that the recursive neuron (2.15) defines a functor from a monoidal grammar to the category of neural networks **NN**. Given any set of Σ-labelled DAGs X, we may construct a monoidal signature $G = \Sigma + E^* \xleftarrow{\varphi + \mathtt{in}} N \xrightarrow{\mathtt{out}} E^*$ where N is the set of all vertices appearing in a graph in X and E is the set of all edges appearing in a graph in X, with $\mathtt{in}, \mathtt{out} : N \to E^*$ listing the input and output edges of vertex v respectively and $\varphi : N \to \Sigma$ is the labelling function. Then the recursive neuron o given above defines a functor $O : \mathbf{MC}(G + \mathtt{swap}) \to \mathbf{Set}_{\mathbb{R}}$ from the free monoidal category generated by G and swaps to $\mathbf{Set}_{\mathbb{R}}$. The image of a vertex $v : l \oplus \vec{e} \to \vec{e'}$ is a function $O(v) : \mathbb{R}^{|\vec{e}|+1} \to \mathbb{R}^{|\vec{e'}|}$ given by: $O(v)(x) = \mathtt{copy}(o(v)(x))$ where $\mathtt{copy} : \mathbb{R} \to \mathbb{R}^{|\vec{e'}|}$ is the diagonal map in $\mathbf{Set}_{\mathbb{R}}$. Note that any DAG $H \in X$ gives rise to a morphism in $\mathbf{MC}(G + \mathtt{swap})$ given by connecting the boxes (vertices) according to the topological structure of the graph, using swaps if necessary. Note that the functor O factors through the category **NN** of neural networks since all the components of Equation 2.15 are generators of **NN**, thus O defines a mapping from DAGs in X to neural networks. Generalising from this to the case where vertices in the graph can be assigned multiple neurons, we define recursive neural networks as follows.

Definition 2.3.5. *A recursive network model is a monoidal functor $F : G \to \mathbf{NN}$ for a monoidal grammar G, such that $F(w) = 1$ for $w \in V \subseteq G_0$. Given a choice of parameters $\theta : W \to \mathbb{R}$, the semantics of a parsed sentence $g : u \to s$ in $\mathbf{MC}(G)$ is given by $I_\theta(F(g)) \in \mathbb{R}^{F(s)}$.*

Remark 2.3.6. *Although it is in general useful to consider non-planar graphs as the input of a recursive neural network, in applications to linguistics one can usually*

assume that the graphs are planar. In order to recover non-planar graphs from the definition above it is sufficient to add a swap to the signature G.

With this definition at hand, we can look at the applications of RvNNs in linguistics. The recursive networks for sentiment analysis of [Soc+13a] are functors from a context-free grammar to neural networks. In this case RvNNs are shown to capture correctly the role of negation in changing the sentiment of a review. This is because the tree structure induced by the context-free grammar captures the part of the phrase which is being negated, as in the following example.

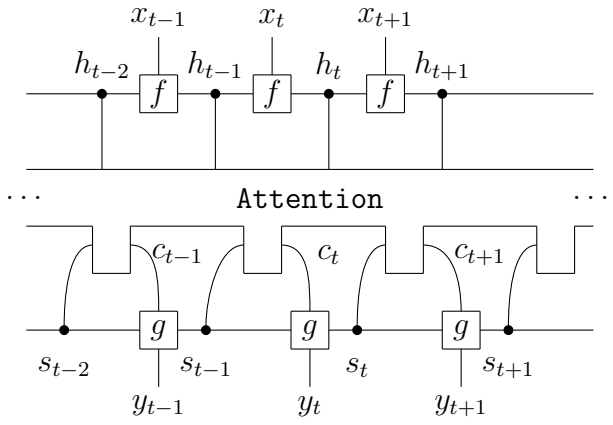


As shown in [Soc+13a, Figure 4] the recurrent network automtically learned to assign a negative sentiment to the phrase above, even if the sub-phrase "very good" is positive. Generalising from this, researchers have shown that recursive networks perform well on natural language entailment tasks [BPM15].

2.3.4 Attention is all you need?

The great advantage of neural networks is also their worst handicap. Since they are able to approximate almost any function, we have no guarantees as to what the resulting function will look like, an issue famously put as the "black box problem". The baffling thing is that neural networks seem to work best when their underlying structure is unconstrained and neurons are arranged in a deep fully connected network. We can see this pattern in the recent progression that neural language models have made: from structured networks to attention.

Attention layers were added to recurrent neural networks by Bahdanau et al. in 2014 [BCB14]. The architecture of Bahdanau et al. is composed of two recurrent neural networks, an encoder f and a decoder g, connected by an attention mechanism as in the following diagram:



(2.16)

where x_i is the ith input word in the domain language, y_i is the ith output word in the target language, the h_is and s_is are the hidden states of the encoder and decoder RNN respectively:

$$h_i = f(x_i, h_{i-1}) \qquad s_i = g(s_{i-1}, y_{i-1}, c_i)$$

and c_i is the context vector calculated from the input $\vec{x}$, the hidden states $\vec{h}$ and the last decoder hidden state s_{i-1} as follows:

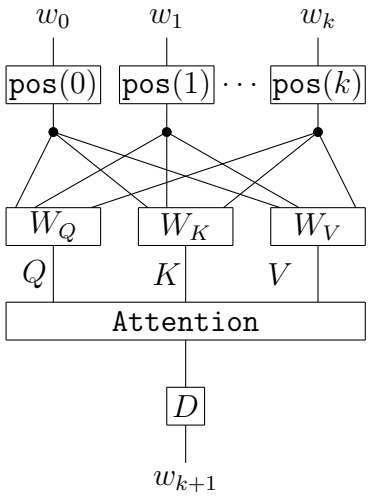
$$c_i = \sum_{j=1}^{n} \alpha_{ij} h_j \tag{2.17}$$

where

$$\alpha_{ij} = (\texttt{softmax}_n(\vec{e_i}))_j \quad (\vec{e_i})_j = a(s_{i-1}, h_j) \tag{2.18}$$

where $n = |\vec{x}|$ and a is a (learned) feed-forward neural network. The coefficients α_{ij} are called *attention weights*, they provide information as to how much input word x_j is relevant for determining the output word y_i. In the picture above, we used a comb notation for `Attention` informally to represent the recursive relation between context vectors c and hidden states h and s, capturing the flow of information in the architecture of Bahdanau et al. This diagram should be read from top-left to bottom-right. At each time step t, the attention mechanism computes the context vector c_t from the last decoder hidden state s_{t-1} and all the encoder hidden states $\vec{h}$. Infinite comb diagrams such as the one above may be formalised as monoidal streams over the category of neural networks [DdR22]. A similar notation is used in Chapter 3. Note that Bahdanau et al. model the encoder f as a bidirectional RNN [SP97] which produces hidden states h_i that depend both on the previous and the following words. For simplicity we have depicted f as a standard RNN.

In 2017, Vaswani et al. published the paper "Attention is all you need" [Vas+17] which introduced *transformers*. They showed that the recurrent structure is not needed to obtain state-of-the-art results in machine translation. Instead, they proposed a model built up from three simple components: positional encoding, attention and feed-forward networks, composed as in the following diagram:



(2.19)

where

1. W_Q, W_K, W_V are (learned) linear functions, and D is a (learned) decoder neural network.

2. the positional encoding $\texttt{pos}$ supplements the word vector w_i of dimension n with another n dimensional vector $\texttt{pos}(i) \in \mathbb{R}^n$ where $\texttt{pos} : \mathbb{N} \to \mathbb{R}^n$ is given by:
$$\texttt{pos}(i)_j = \begin{cases} \texttt{sin}(\frac{j}{m^{2k/n}}) & \text{if } j = 2k \\ \texttt{cos}(\frac{j}{m^{2k/n}}) & \text{if } j = 2k+1 \end{cases}$$
where $m \in \mathbb{N}$ is a hyperparameter which determines the phase of the sinusoidal function (Vaswani et al. choose $m = 1000$ [Vas+17]),

3. and attention is defined by the following formula:
$$\texttt{Attention}(Q, K, V) = \texttt{softmax}(\frac{QK^T}{\sqrt{d_K}})V \tag{2.20}$$
where Q and K are vectors of the same dimension d_K called *query* and *keys* respectively and V is a vector called *values*.

Note that the positional encoding is needed since otherwise the network would have no information about the position of words in a sentence. Using these sinusoidal encodings turns absolute into relative position [Vas+17]. Comparing this definition 2.20 of attention with the one used by Bahdanau et al. [BCB14], we note that in Equations 2.17 and 2.18 the hidden states $\vec{h}$ and $\vec{s}$ play the role of keys K and queries Q respectively, and the values V are again taken to be encoder hidden states $\vec{h}$. The main difference is that instead of a deep neural network (denoted a above) the queries and keys are pre-processed with *linear* operations (W_K, W_Q and W_V above). This architecture is the basis of BERT [Dev+19] and its extensions such as GPT-3 which use billions of parameters to achieve state of the art results in a wide range of NLP tasks.

Reasoning about what happens inside these models and explaining their behaviour with linguistic analysis is hard. Indeed, the same architectures where sentences are replaced by images and words by parts of that image give surprisingly accurate results in the image recognition task [Dos+20], suggesting that linguistic principles are insufficient for analysing these algorithms. However, viewing deep neural networks as end-to-end black-box learners, it becomes interesting to open the box and analyse the output of the intermediate layers with the aim of understanding the different features that the network learns in the process. Along these lines, researchers have found that neural network models automatically encode linguistic features such as grammaticality [Coe+19] and dependency parsing [ACL19]. One possible line of future research would be to study what happens in the linear world of keys, values and queries. One may take the latest forms of attention [Vas+17] as the realisation that accurate predictions can be obtained from a linear process $fracQK^T\sqrt{d_K}$ by using a single $\texttt{softmax}$ activation. We will give a bayesian interpretation of this activation function in Chapter 3.

2.3.5 Neural networks in DisCoPy

We now show how to interface DisCoPy with Tensorflow/Keras neural networks [Cho+15; Mar+15]. In order to do this, we need to define objects, identities, composition and tensor for Keras models.

The objects of our category of neural networks will be instances of `PRO`, a subclass of `monoidal.Ty` initialised by a dimension `n`. A morphism from `PRO(n)` to `PRO(k)` is a neural network with input shape `(n, )` and output shape `(k, )` (we only deal with flat shapes for simplicity). A `neural.Network` is initialised by providing domain and codomain dimensions together with a Keras model of that type. Composition is easily implemented using the call method of Keras models. For `tensor`, we first need to split the domain using `keras.layers.Lambda`, then we act on each subspace independently with `self` and `other`, and finally we concatenate the outputs. Identities are simply Keras models with `outputs = inputs`, and we include a static method for constructing dense layer models.

Listing 2.3.7. The category of Keras models

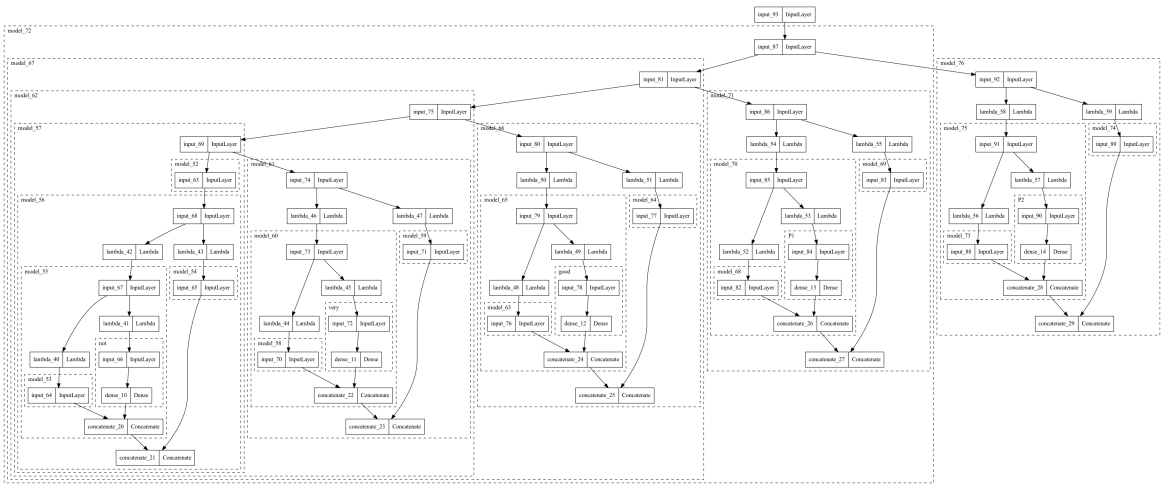
```
from discopy import monoidal, PRO
import tensorflow as tf
from tensorflow import keras

class Network(monoidal.Box):
    def __init__(self, dom, cod, model):
        self.model = model
        super().__init__("Network", dom, cod)

    def then(self, other):
        inputs = keras.Input(shape=(len(self.dom),))
        output = self.model(inputs)
        output = other.model(output)
        composition = keras.Model(inputs=inputs, outputs=output)
        return Network(self.dom, other.cod, composition)

    def tensor(self, other):
        dom = len(self.dom) + len(other.dom)
        cod = len(self.cod) + len(other.cod)
        inputs = keras.Input(shape=(dom,))
        model1 = keras.layers.Lambda(
            lambda x: x[:, :len(self.dom)],)(inputs)
        model2 = keras.layers.Lambda(
            lambda x: x[:, len(self.dom):],)(inputs)
        model1 = self.model(model1)
        model2 = other.model(model2)
        outputs = keras.layers.Concatenate()([model1, model2])
        model = keras.Model(inputs=inputs, outputs=outputs)
        return Network(PRO(dom), PRO(cod), model)

    @staticmethod
    def id(dim):
        inputs = keras.Input(shape=(len(dim),))
        return Network(dim, dim, keras.Model(inputs=inputs, outputs=inputs))
```

```
    @staticmethod
    def dense_model(dom, cod, hidden_layer_dims=[], activation=tf.nn.relu):
        inputs = keras.Input(shape=(dom,))
        model = inputs
        for dim in hidden_layer_dims:
            model = keras.layers.Dense(dim, activation=activation)(model)
        outputs = keras.layers.Dense(cod, activation=activation)(model)
        model = keras.Model(inputs=inputs, outputs=outputs)
        return Network(PRO(dom), PRO(cod), model)
```

As an application, we use a `monoidal.Functor` to construct a Keras model from a context-free grammar parse, illustrating the tree neural networks of [Soc+13a].

Listing 2.3.8. Tree neural networks

```
from discopy.monoidal import Ty, Box, Functor

n, a, np, neg, star = Ty('N'), Ty('A'), Ty('NP'), Ty('Neg'), Ty('*')
not_, very, good = Box('not', star, neg), Box('very', star, a), Box('good', star, n)
P1, P2 = Box('P1', a @ n, np), Box('P2', neg @ np, np)
diagram = not_ @ very @ good >> Id(neg) @ P1 >> P2

dim = 5
ob = lambda x: PRO(dim)
ar = lambda box: Network.dense_model(len(box.dom) * dim, len(box.cod) * dim)
F = Functor(ob, ar, ob_factory=PRO, ar_factory=Network)
keras.utils.plot_model(F(diagram).model)
```



2.4 Relational models

The formal study of *relations* was initiated by De Morgan in the mid 19th century [De 47]. It was greatly developed by Peirce who only published small fragments of his *calculus of relations* [Pei97], although much of it was popularised in the influential work of Schroder [Sch90]. In the first half of the twentieth century, this calculus was often disregarded in favour of Frege's and Russell's approach to logic [Ane12], until it was revived by Tarski [Tar41] who developed it into the rich field of model theory.

With the advent of computer science, the calculus of relations came to be recognized as a convenient framework for storing and accessing data, leading to the development of *relational databases* in the 1970s [Cod70]. SQL queries were introduced by Chamberlin and Boyce in 1976 [CB76] and they are still used for accessing databases today. *Conjunctive queries* are an important subclass of SQL, corresponding to the Select-Where-From fragment. They were introduced by Chandra and Merlin [CM77] who showed that their evaluation in a relational database is an NP-complete problem, spawning a large field of studies in the complexity of constraint satisfaction problems [DKV02]. The blend of algebra and logic offered by the theory of relations is particularly suited to a categorical formalisation and it has motivated the work of Carboni and Walters [CW87] as well as Brady and Trimble [BT00] and Bonchi et al. [BPS17; BSS18] among others.

In this section, we start by reviewing the theory of relational databases and conjunctive queries from a categorical perspective. We then introduce relational models by transposing the definitions into a linguistic setting. This allows us to transfer results from database theory to linguistics and define $\texttt{NP}-\texttt{complete}$ *entailment* and *question answering* problems. The correspondence of terminology between databases, category theory and linguistics is summarized in the table below.

Databases	Algebra	Linguitics
Relational database	Cartesian bicategory	Relational model
attributes	objects	basic types
schema	signature	lexicon
query	morphism	sentence
instance	functor	model
containment	preorder enrichment	entailment

2.4.1 Databases and queries

We introduce the basic notions of relational databases, starting with an example.

Example 2.4.1.

reader	*book*	*writer*
Spinoza	*De Causa*	*Bruno*
Shakespeare	*World of Wordes*	*Florio*
Florio	*De Causa*	*Bruno*
Leibniz	*Tractatus*	*Spinoza*

Consider the structure of the table above, which we denote by ρ. There is a set of attributes $A = \{$ *reader, book, writer* $\}$ *which name the columns of the table and a set of* data values D_a *for each attribute* $a \in A$*,*

$$D_r = D_w = \{\textit{Spinoza, Shakespeare, Leibniz, Bruno, Florio}\}$$

$$D_b = \{\textit{De Causa, World of Wordes, Tractatus}\}$$

A row of the table is a tuple $t \in \prod_{a \in A} D_a$*, which assigns a particular value* t_a *to each attribute* $a \in A$*, e.g.* (*Leibniz, Tractatus, Spinoza*)*. The table then consists in a set of tuples, i.e. a* relation $\rho \subseteq \prod_{a \in A} D_a$*.*

A relational database is a collection of tables (relations), organised by a *schema*. Given a set of attributes A, a schema Σ is a set of relational symbols, together with a domain function $\texttt{dom} : \Sigma \to A^*$. The schema serves to specify the set of names Σ for the tables in a database together with the type of their columns. For example we may have $\rho \in \Sigma$ for the table above with $\texttt{dom}(\rho) = (\text{reader}, \text{book}, \text{writer})$. We have already encountered this type of structure in 1.6.1, where we used the term *hypergraph signature* instead of schema.

Definition 2.4.2. *A relational database K with schema Σ is an assignment of each attribute $a \in A$ to a corresponding set of data values D_a, and an assignment of each symbol $R \in \Sigma$ to a relation $K(R) \subseteq \prod_{a \in \texttt{dom}(R)} D_a$.*

Instead of working directly with relational databases, it is often convenient to work with a simpler notion known as a *relational structure*. The schema is replaced by a *relational signature*, which is a set of symbols Σ equipped with an *arity* function $\texttt{ar} : \Sigma \to \mathbb{N}$.

Definition 2.4.3 (Relational structure)**.** *A relational structure K over a signature Σ, also called a Σ-structure, is given by a set U called the* universe *and an interpretation $K(R) \subseteq U^{\texttt{ar}(R)}$ for every symbol $R \in \Sigma$. We denote by $\mathcal{M}_\Sigma$ the set of Σ-structures with* finite *universe $U(K)$.*

Given two Σ-structures K, K', a homomorphism $f : K \to K'$ is a function $f : U(K) \to U(K')$ such that $\forall\ R \in \Sigma\ \forall\ \vec{x} \in U^{\texttt{ar}(R)}\ \cdot\ \vec{x} \in K(R) \implies f(\vec{x}) \in K'(R)$.

Remark 2.4.4. *Note that relational structures are the same as relational databases with only one attribute. Attributes $a \in A$ can be recovered by encoding them as predicates $a \in \Sigma$ of arity 1 and one may take the universe to be the union of the sets of data values $U = \cup_{a \in A} D_a$.*

We consider the problem of finding a homomorphism between relational structures.

Definition 2.4.5. `Homomorphism`

Input: $K, K' \in \mathcal{M}_\Sigma$
Output: $f : K \to K'$

Proposition 2.4.6. *[GJ90]* `Homomorphism` *is* $\texttt{NP} - \texttt{complete}$.

Proof. Membership may be shown to follow from Fagin's theorem: homomorphisms are defined by an existential second-order logic formula. Hardness follows by reduction from graph homomorphism: take $\Sigma = \{ \bullet \}$ and $\mathtt{ar}(\bullet) = 2$ then a Σ-structure is a graph. □

The most prominent query language for relational databases is SQL [CB76]. *Conjunctive queries* form a subset of SQL (corresponding to the Select-Where-From fragment) with a convenient mathematical formulation. We define conjunctive queries and the corresponding `Evaluation` and `Containment` problems. Let $\mathcal{X}$ be a (countable) set of variables, Σ a relational signature and consider the logical formulae generated by the following context-free grammar:

$$\varphi \ ::= \ \top \mid x = x' \mid \varphi \wedge \varphi \mid \exists\, x \cdot \varphi \mid R(\vec{x})$$

where $x, x' \in \mathcal{X}$, $R \in \Sigma$ and $\vec{x} \in \mathcal{X}^{\mathtt{ar}(R)}$. Let us denote the variables of φ by $\mathtt{var}(\varphi) \subseteq \mathcal{X}$, its free variables by $\mathtt{fv}(\varphi) \subseteq \mathtt{var}(\varphi)$ and its atomic formulae by $\mathtt{atoms}(\varphi) \subseteq \coprod_{R \in \Sigma} \mathrm{var}(\varphi)^{\mathtt{ar}(R)}$, i.e. an atomic formula is given by $R(x_1, \ldots, x_{\mathtt{ar}(R)})$ for some variables $x_i \in \mathcal{X}$.

This fragment is called *regular logic* in the category-theory literature [FS18a]. It yields conjuntive queries via the *prenex normal form*.

Definition 2.4.7. *Conjunctive queries $\varphi \in \mathcal{Q}_\Sigma$ are the prenex normal form $\varphi = \exists\, x_0 \cdots \exists\, x_k \cdot \varphi'$ of regular logic formulae, for the bound variables $\{ x_0, \ldots, x_k \} = \mathtt{var}(\varphi) \setminus \mathtt{fv}(\varphi)$ and $\varphi' = \bigwedge \mathtt{atoms}(\varphi)$. We denote by :*

$$\mathcal{Q}_\Sigma(k) = \{ \varphi \in \mathcal{Q}_\Sigma \mid \mathtt{fv}(\varphi) = k \}$$

the set of conjunctive queries with k free variables.

Given a structure $K \in \mathcal{M}_\Sigma$, let $\mathtt{eval}(\varphi, K) = \{ v \in U(K)^{\mathtt{fv}(\varphi)} \mid (K, v) \vDash \varphi \}$ where the satisfaction relation ($\vDash$) is defined in the usual way.

Definition 2.4.8. `Evaluation`

Input: $\varphi \in \mathcal{Q}_\Sigma, \quad K \in \mathcal{M}_\Sigma$
Output: $\mathtt{eval}(\varphi, K) \subseteq U(K)^{\mathtt{fv}(\varphi)}$

Definition 2.4.9. `Containment`

Input: $\varphi, \varphi' \in \mathcal{Q}_\Sigma$
Output: $\varphi \subseteq \varphi' \ \equiv \ \forall\ K \in \mathcal{M}_\Sigma \ \cdot \ \mathtt{eval}(\varphi, K) \subseteq \mathtt{eval}(\varphi', K)$

Example 2.4.10. *Following from 2.4.1 let $U := D_w \cup D_b$ and fix the schema $\Sigma = \{$read, wrote$\} \cup U$ with $\mathtt{ar}(w) = 1$ for $w \in U$ and $\mathtt{ar}($read$) = \mathtt{ar}($wrote$) = 2$. Consider the relational structure $K \in \mathcal{Q}_\Sigma$ with universe U and $K(w) = \{ w \} \subseteq U$ for $w \in \Sigma - \{$ read, wrote $\}$ and $K($read$) \subseteq U \times U$ given by the first two columns of table ρ, $K($wrote$) \subseteq U \times U$ given by the second two columns of ρ. The following is a conjunctive query with no free variables:*

$$\varphi = \exists y, z \ \cdot \ \mathit{read}(x, y) \wedge \mathit{Bruno}(y) \wedge \mathit{wrote}(x, z)$$

Since φ has one free variables, $\mathtt{eval}(\varphi, K) \subseteq U$. *With K as defined above there are three valuations $v : \mathtt{var}(\varphi) = \{ x, y, z \} \to U$ such that $(v, K) \vDash \varphi$, yielding* $\mathtt{eval}(\varphi, K) = \{$ *Spinoza, Florio, Galileo* $\} \subseteq U$.

Definition 2.4.11 (Canonical structure)**.** *Given a query $\varphi \in \mathcal{Q}_\Sigma$, the canonical structure $CM(\varphi) \in \mathcal{M}_\Sigma$ is given by $U(CM(\varphi)) = \mathtt{var}(\varphi)$ and $CM(\varphi)(R) = \{ \vec{x} \in \mathtt{var}(\varphi)^{\mathtt{ar}(R)} \mid R(\vec{x}) \in \mathtt{atoms}$ for $R \in \Sigma$.*

This result was used by Chandra and Merlin to reduce from `Homomorphism` to both `Evaluation` and `Containment`.

Theorem 2.4.12 (Chandra-Merlin [CM77])**.** *The problems* `Evaluation` *and* `Containment` *are logspace equivalent to* `Homomorphism`*, hence* $\mathtt{NP} - \mathtt{complete}$.

Proof. Given a query $\varphi \in \mathcal{Q}_\Sigma$ and a structure $K \in \mathcal{M}_\Sigma$, query evaluation $\mathtt{eval}(\varphi, K)$ is given by the set of homomorphisms $CM(\varphi) \to K$. Given $\varphi, \varphi' \in \mathcal{M}_\Sigma$, we have $\varphi \subseteq \varphi'$ iff there is a homomorphism $f : CM(\varphi) \to CM(\varphi')$ such that $f(\mathtt{fv}(\varphi)) = \mathtt{fv}(\varphi')$. Given a structure $K \in \mathcal{M}_\Sigma$, we construct $\varphi \in \mathcal{Q}_\Sigma$ with $\mathtt{fv}(\varphi) = \varnothing$, $\mathtt{var}(\varphi) = U(K)$ and $\mathtt{atoms}(\varphi) = K$. □

2.4.2 The category of relations

Let us now consider the structure of the category of relations. A relation $R : A \to B$ is a subset $R \subseteq A \times B$ or equivalently a predicate $R : A \times B \to \mathbb{B}$, we write aRb for the logical statement $R(a, b) = 1$. Given $S : B \to C$, the composition $R; S : A \to C$ is defined as follows:

$$aR; Sc \iff \exists b \in B \cdot aRb \wedge bSc\,.$$

Under this composition relations form a category denoted **Rel**.

We can also construct the category of relations by considering the powerset monad $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ defined on objects by $\mathcal{P}(X) = \{ S \subseteq X \}$ and on arrows by $f : X \to Y$ by $\mathcal{P}(f) : \mathcal{P}(X) \to \mathcal{P}(Y) : S \mapsto f(S)$. A relation $R : A \to B$ is the same as a function $R : A \to \mathcal{P}(B)$ and in fact $\mathbf{Rel} = \mathbf{Kl}(\mathcal{P})$ is the Kleisli category of the powerset monad.

Any function $f : A \to B$ induces a relation $I(f) = \{ (x, f(x)) \mid x \in A \} \subseteq A \times B$, sometimes called the graph of f. The tensor product of relations $R : A \to B$ and $T : C \to D$ is denoted $R \otimes T : A \times C \to B \times D$ and defined by:

$$(a, c)R \otimes T(b, d) \iff aRb \wedge cTd\,.$$

Equipped with this tensor product and unit the one-element set 1, **Rel** forms a symmetric monoidal category, with symmetry lifted from **Set**. We can thus use the graphical language of monoidal categories for reasoning with relations.

Note that each object $A \in \mathbf{Rel}$ is self-dual, as witnessed by the morphisms $\mathtt{cup}_A : A \times A \to 1$ and $\mathtt{cap}_A : 1 \to A \times A$, defined by:

$$(a, a')\mathtt{cup}_A \iff (a = a') \iff \mathtt{cap}_A(a, a')$$

These denoted graphically as cups and caps and satisfy the snake equations 1.18. Thus **Rel** is a compact-closed category. Moreover, every object $A \in \mathbf{Rel}$ comes equipped with morphisms $\Delta : A \to A \times A$ and $\nabla : A \times A \to A$ defined by:

$$a\Delta(a', a'') \iff (a = a' = a'') \iff (a, a')\nabla a'' .$$

Together with the unit $\eta : 1 \to A$ and the counit $\epsilon : A \to 1$, defined by $\eta = A = \epsilon$, the tuple $(\Delta, \epsilon, \nabla, \eta)$ satisfies the axioms of special commutative frobenius algebras, making **Rel** a hypergraph category in the sense of Fong and Spivak [FS18b]. Note that $\mathtt{cup} = \nabla; \epsilon$ and $\mathtt{cap} = \eta; \Delta$, moreover it is easy to show that the snake equations follow from the axioms of special commutative Frobenius algebras. Finally, we can equip the hom-sets $\mathbf{Rel}(A, B)$ with a preorder structure given by:

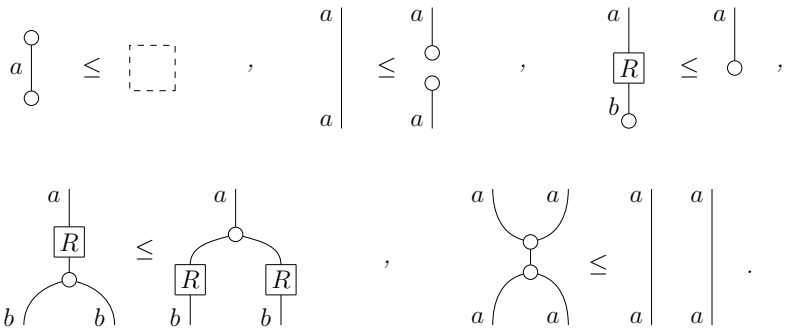
$$R \leq S \iff (aRb \implies aSb) .$$

In category theory, this situation is known as a *preorder enrichment*. Equipped with this preorder enrichment, **Rel** forms a *Cartesian bicategory* in the sense of Carboni and Walters [CW87].

2.4.3 Graphical conjunctive queries

Bonchi, Seeber and Sobocinski [BSS18] introduced graphical conjunctive queries (GCQ), a graphical calculus where query evaluation and containment are captured by the axioms of the *free Cartesian bicategory* $\mathbf{CB}(\Sigma)$ generated by a relational signature Σ. Cartesian bicategories were introduced by Carboni and Walters [CW87] as an axiomatisation of categories of relations, they are hypergraph categories where every hom-set has a partial order structure akin to subset inclusion between relations. We review the correspondence of Bonchi et al. [BSS18], between conjunctive queries and morphisms of free cartesian bicategories. We refer to Appendix 2.4.1 for an introduction to relational datatabases with examples.

Definition 2.4.13 (Cartesian bicategory)**.** *[CW87] A cartesian bicategory* $\mathbf{C}$ *is a hypergraph category enriched in preorders and where the preorder structure interacts with the hypergraph structure as follows:*



for all objects $a, b \in \mathbf{C}_0$ *and morphisms* $R : a \to b$*. A morphism of Cartesian bicategories is a strong monoidal functor which preserves the partial order, the monoid and the comonoid structure.*

We recall the basic notions of relational databases and conjunctive queries. A *relational signature* is a set of symbols Σ equipped with an *arity* function $\mathtt{ar} : \Sigma \to \mathbb{N}$. This is used to define a relational structure as a mathematical abstraction of a databases.

Definition 2.4.14 (Relational structure)**.** *A relational structure K over a signature Σ, also called a Σ-structure, is given by a set U called the* universe *and an interpretation $K(R) \subseteq U^{\mathtt{ar}(R)}$ for every symbol $R \in \Sigma$. We denote by $\mathcal{M}_\Sigma$ the set of Σ-structures with* finite *universe $U(K)$.*

Given two Σ-structures K, K', a homomorphism $f : K \to K'$ is a function $f : U(K) \to U(K')$ such that $\forall\ R \in \Sigma\ \forall\ \vec{x} \in U^{\mathtt{ar}(R)}\ \cdot\ \vec{x} \in K(R) \implies f(\vec{x}) \in K'(R)$.

Let $\mathcal{X}$ be a (countable) set of variables, Σ a relational signature and consider the logical formulae generated by the following context-free grammar:

$$\varphi \ ::= \ \top \mid x = x' \mid \varphi \wedge \varphi \mid \exists\, x \cdot \varphi \mid R(\vec{x})$$

where $x, x' \in \mathcal{X}$, $R \in \Sigma$ and $\vec{x} \in \mathcal{X}^{\mathtt{ar}(R)}$. Let us denote the variables of φ by $\mathtt{var}(\varphi) \subseteq \mathcal{X}$, its free variables by $\mathtt{fv}(\varphi) \subseteq \mathtt{var}(\varphi)$ and its atomic formulae by $\mathtt{atoms}(\varphi) \subseteq \coprod_{R \in \Sigma} \mathrm{var}(\varphi)^{\mathtt{ar}(R)}$, i.e. an atomic formula is given by $R(x_1, \ldots, x_{\mathtt{ar}(R)})$ for some variables $x_i \in \mathcal{X}$. This fragment is called *regular logic* in the category-theory literature [FS18a]. It yields conjuntive queries via the *prenex normal form*.

Definition 2.4.15. *Conjunctive queries $\varphi \in \mathcal{Q}_\Sigma$ are the prenex normal form $\varphi = \exists\ x_0 \cdots \exists\ x_k \cdot \varphi'$ of regular logic formulae, for the bound variables $\{ x_0, \ldots, x_k \} = \mathtt{var}(\varphi) \setminus \mathtt{fv}(\varphi)$ and $\varphi' = \bigwedge \mathtt{atoms}(\varphi)$. We denote by :*

$$\mathcal{Q}_\Sigma(k) = \{ \varphi \in \mathcal{Q}_\Sigma \mid \mathtt{fv}(\varphi) = k \}$$

the set of conjunctive queries with k free variables.

Proposition 2.4.16. *There is a bijective correspondence between relational structures with signature $\sigma : \Sigma \to \mathbb{N} = \{ x \}^*$ and monoidal functors $K : \Sigma \to \mathbf{Rel}$ such that $K(x) = U$.*

Proof. Given a schema $\mathtt{dom} : \Sigma \to A^*$, the data for a monoidal functor $K : \Sigma \to \mathbf{Rel}$ is an assignment of each $a \in A$ to a set of data-values $D_a = K(a)$ and of each symbol $R \in \Sigma$ to a relation $K(R) \subseteq \prod_{a \in \mathtt{dom}(R)} D_a$. This is precisely the data of a relational database. Relational structures are a sub-example with $A = \{ x \}$. □

Bonchi, Seeber and Sobocinski show that queries can be represented as diagrams in the free cartesian bicategory, and that this translation is semantics preserving. Let $\mathbf{CB}(\Sigma)$ be the free Cartesian bicategory generated by one object x and arrows $\{ R : 1 \to x^{\mathtt{ar}}(R) \}_{R \in \Sigma}$, see [BSS18, def. 21].

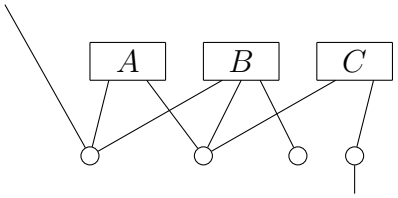
Proposition 2.4.17. *([BSS18, prop. 9, 10]) There is a two-way translation between formulas and diagrams:*

$$\Theta : \mathcal{Q}_\Sigma \leftrightarrows \mathbf{CB}(\Sigma) : \Lambda$$

which preserves the semantics, i.e. such that for all $\varphi, \varphi' \in \mathcal{Q}_\Sigma$ we have $\varphi \subseteq \varphi' \iff \Theta(\varphi) \leq \Theta(\varphi')$, and for all arrows $d, d' \in \mathbf{CB}(\Sigma)$, $d \leq d' \iff \Lambda(d) \subseteq \Lambda(d')$.

Proof. The translation is defined by induction from the syntax of regular logic formulae to that of GCQ diagrams and back. Note that given $\varphi \in \mathcal{Q}_\Sigma$ with $|\mathtt{fv}(\varphi)| = n$, we have $\Theta(\varphi) \in \mathbf{CB}(\Sigma)(0, n)$ and similarly we have $\mathtt{fv}(\Lambda(d)) = m + n$ for $d \in \mathbf{CB}(\Sigma)(m, n)$, i.e. open wires correspond to free variables. □

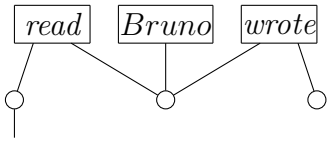
Example 2.4.18. *The translation works as follows. Given a morphism in* $\mathbf{CB}(\Sigma)$*, normalized according to 1.6.5, the spiders are interpreted as variables and the boxes as relational symbols. For example, assuming* $A, B, C \in \Sigma$*, the following morphism* $f : x \to x \in \mathbf{CB}(\Sigma)$



is mapped to the query:

$$\Lambda(f) = \exists x_1, x_2 \cdot A(x_0, x_1) \wedge B(x_0, x_1, x_2) \wedge C(x_1, x_3)$$

The query from Example 2.4.10, is mapped by Θ *to the diagram:*



Proposition 2.4.19. *Let* $[\mathbf{CB}(\Sigma), \mathbf{Rel}]$ *denote the set of morphisms of Cartesian bicategories, there are bijective correspondences between closed diagrams in* $\mathbf{CB}(\Sigma)$*, formulas in* $\mathcal{Q}_\Sigma$ *with no free variables and models with signature* Σ*.*

$$\mathbf{CB}(\Sigma)(0,0) \overset{(1)}{\simeq} \{\, \varphi \in \mathcal{Q}_\Sigma \mid \mathtt{fv}(\varphi) = \varnothing \,\} \overset{(2)}{\simeq} \mathcal{M}_\Sigma \overset{(3)}{\simeq} [\mathbf{CB}(\Sigma), \mathbf{Rel}]$$

Proof. (1) follows from theorem 2.4.17, (2) from theorem 2.4.12 and (3) follows from proposition 2.4.16 since any monoidal functor $\Sigma \to \mathbf{Rel}$ induces a morphism of Cartesian bicategories $\mathbf{CB}(\Sigma) \to \mathbf{Rel}$. □

2.4.4 Relational models

We have seen that relational databases are functors $K : \Sigma \to \mathbf{Rel}$ from a relational signature Σ. It is natural to to generalise this notion by considering functors $G \to \mathbf{Rel}$ where G is a formal grammar. Any of the formal grammars studied in Chapter 1 may be used to build a relational model. However, it is natural to pick a grammar G that we can easily interpret in **Rel**. In other words, we are interested in grammars which have common structure and properties with the category of relations. Recall that **Rel** is compact-closed with the diagonal and its transpose as cups and caps. This makes rigid grammars particularly suited for relational semantics, since we can interpret cups and caps using the compact closed structure of **Rel**.

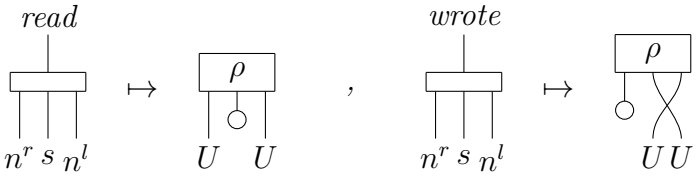
Definition 2.4.20 (Relational model)**.** *A relational model is a rigid monoidal functor* $F : G \to \mathbf{Rel}$ *where* G *is a rigid grammar.*

We illustrate relational models with an example.

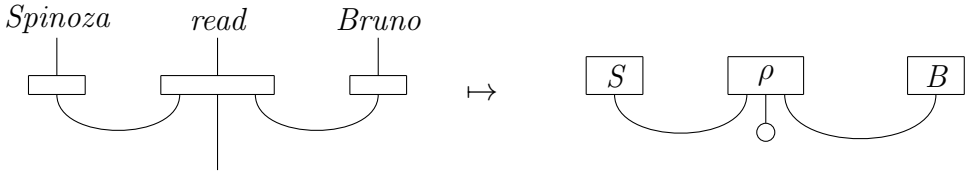
Example 2.4.21 (Truth values)**.** *Let us fix the vocabulary* $V = U + \{\, read, wrote \,\}$*, where* $U = D_w \cup D_b \cup D_r$ *is the set of data values from Example 2.4.1. Consider the pregroup grammar defined by the following lexicon:*

$$\Delta(x) = \{\, n \,\}\ , \quad \Delta(read) = \Delta(wrote) = \{\, n^r s n^l \,\}$$

for all $x \in U \subseteq V$*. We build a functor* $F : \Delta \to \mathbf{Rel}$*, defined on objects by* $F(n) = U$*, and* $F(w) = 1 = F(s)$ *for all* $w \in V$*, on proper nouns by* $F(x \to n) = \{\, x \,\} \subseteq U$ *for* $x \in U \subseteq V$ *and on verbs as follows:*



where $\rho : 1 \to U \otimes U \otimes U \in \mathbf{Rel}$ *is the table (relation) from Example 2.4.1. Interpreting cups and caps in* $\mathbf{RC}(\Delta)$ *with their counterparts in* **Rel***, we can evaluate the semantics of the sentence* g : *Spinoza read Bruno* $\to s$*:*



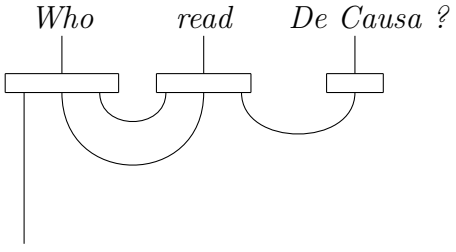
obtaining a morphisms $F(g) : 1 \to 1 \in \mathbf{Rel}$*, which is simply a* truth value *given by the evaluation of the following query:*

$$F(g) = \top \iff \exists x, y \in U \cdot F(Spinoza)(x) \wedge F(read)(x, y) \wedge F(Bruno)(y)$$

which is true with our definition of F*.*

From this example, we see that relational models can be used to give a truth theoretic semantics by interpreting the sentence type s as the unit of the tensor in **Rel**. We can also use these models to answer questions.

Example 2.4.22 (Questions)**.** *Add a question type* $q \in B$ *and the question word* $Who \in V$ *to the pregroup grammar above with* $\Delta(Who) = q\, s^l\, n$*. Then there is a grammatical question* g_q : *Who read De Causa* $\to s$ *in* $\mathbf{RC}(\Delta)$ *given by the following diagram:*



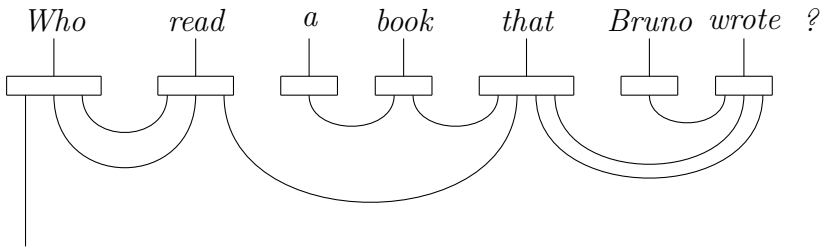
Let $F(q) = U$ *and* $F(\mathit{Who} \to q\, s^l\, n) = \mathtt{cap}_U \subseteq U \otimes U$. *Then evaluating the question above in* F *yields* $F(g_q) = \{\, \mathit{Spinoza},\ \mathit{Florio} \,\} \subseteq U$.

As shown by Sadrzadeh et al. [SCC14], we may give semantics to the relative pronoun "that" using the Frobenius algebra in **Rel**.

Example 2.4.23 (Relative pronouns)**.** *Add the following lexical entries:*

$$\Delta(\mathit{that}) = \{\, n^r\, n\, s^l\, n^{ll}, n^r\, n\, s^l\, n \,\}\ , \quad \Delta(a) = \{\, d \,\}\ , \quad \Delta(\mathit{book}) = \{\, d^r n \,\}\ .$$

. Then there is a grammatical question g'_q : *Who read a book that Bruno wrote* $\to q$ *in* $\mathbf{RC}(\Delta)$ *given by the following diagram:*



Let $F(d) = 1$, $F(a \to d) = \top$, $F(\mathit{that} \to n^r\, n\, s^l\, n) = \nu \cdot \delta \cdot (\delta \otimes \mathtt{id}_U) : 1 \to U^3$ *(the spider with 3 outputs and 0 inputs) and* $F(\mathit{book} \to d^r\, n) = D_b \subseteq U$. *Then evaluating the question above in* F *yields* $F(g'_q) = \{\, \mathit{Spinoza},\ \mathit{Florio},\ \mathit{Galileo} \,\} \subseteq U$, *as expected.*

The first linguistic problem that we consider is the task of computing the semantics of a sentence $u \in \mathcal{L}(G)$ for a pregroup grammar G in a given relational model $F : G \to \mathbf{Rel}$. Throughout this section and the next, we assume that G is a rigid grammar.

Definition 2.4.24. $\mathtt{RelSemantics}(G)$

Input: $g \in \mathbf{RC}(G)(u, s)$, $F : G \to \mathbf{Rel}$
Output: $F(g)$

Since **Rel** is a cartesian bicategory, any monoidal functor from G to **Rel** must factor through a free cartesian bicategory.

Lemma 2.4.25. *Let* $G = (V, B, \Delta, s)$ *be a pregroup grammar. Any relational model* $F : \mathbf{RC}(G) \to \mathbf{Rel}$ *factors through a free cartesian bicategory* $\mathbf{RC}(G) \to \mathbf{CB}(\Sigma) \to \mathbf{Rel}$, *where* Σ *is obtained from* Δ *via the map* $P(B) \to B^*$ *which forgets the adjoint structure.*

Proof. This follows from the universal property of the free cartesian bicategory. □

This lemma allows to reduce the problem of computing the semantics of sentences in **Rel** to `Evaluation`, thus proving its membership in `NP`.

Proposition 2.4.26. *There is a logspace reduction from* $\mathtt{RelSemantics}(G)$ *to conjunctive query* `Evaluation`, *hence* $\mathtt{RelSemantics} \in \mathtt{NP}$.

Proof. The factorisation $K \circ L = F$ of lemma 2.4.25 and the translation Λ of theorem 2.4.17 are in logspace, they give a query $\varphi = \Lambda(L(r)) \in \mathcal{Q}_\Delta$ such that $\texttt{eval}(\varphi, K) = F(r)$. □

The queries that arise from a pregroup grammar are a particular subclass of conjunctive queries. This leads to the question: what is the complexity of `Evaluation` for this class of conjunctive queries? We conjecture that these queries have bounded treewidth, i.e. that they satisfy the tractability condition for the CSP dichotomy theorem [Bul17].

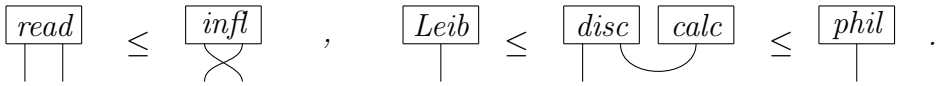
Conjecture 2.4.27. *For any pregroup grammar G, $\texttt{RelSemantics}(G)$ is poly-time computable in the size of $(u, s) \in List(V) \times P(B)$ and in the size of the functor F.*

2.4.5 Entailment and question answering

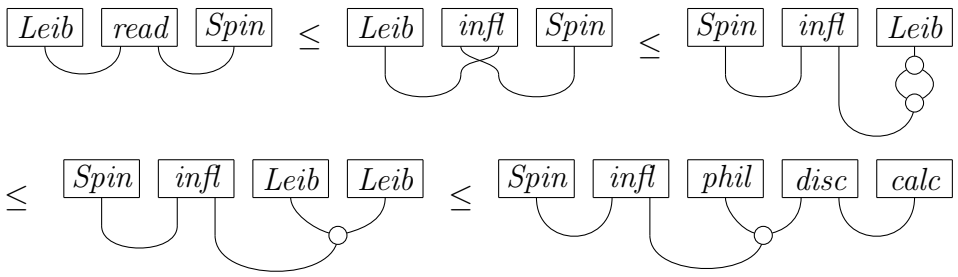
We have seen that relational models induce functors $L : G \to \mathbf{CB}(\Sigma)$, turning sentences into conjunctive queries. Thus we can test whether a sentence $u \in \mathcal{L}(G)$ entails a second sentence $u' \in \mathcal{L}(G)$ by checking containment of the corresponding queries. More generally, we may consider models in a *finitely presented* cartesian bicategory $\mathbf{C}$, i.e. a cartesian bicategory equipped with a finite set of *existential rules* of the form $\forall\, x_0 \cdots \forall\, x_k \cdot \varphi \to \varphi'$ for $\varphi, \varphi' \in \mathbf{CB}(\Sigma)$ with $\texttt{fv}(\varphi) = \texttt{fv}(\varphi') = \{\, x_0, \ldots, x_k \,\}$. These are also called tuple-generating dependencies in database theory [Tho13]. They will allow us to model more interesting forms of entailment in natural language.

Definition 2.4.28 (CB model)**.** *A CB model for a rigid grammar G is a monoidal functor $L : G \to \mathbf{C}$ where $\mathbf{C}$ is a finitely presented Cartesian bicategory.*

Example 2.4.29. *Take $\mathbf{C}$ to be the Cartesian bicategory generated by the signature $\Sigma = \{$ Leib, Spi, infl, calc, phil, ... $\}$ as 1-arrows with codomain given by the function $\texttt{ar} : \Sigma \to \mathbb{N}$ and the following set of 2-arrows:*



The composition of 2-arrows in $\mathbf{C}$ allows us to compute entailment, e.g.:



where the second and third inequations follow from the axioms of definition 2.4.13, the first and last from the generating 2-arrows.

Starting from the pregroup grammar G defined in 2.4.4, and adding the following lexical entries:

$$\Delta(\textit{influenced}) = \Delta(\textit{discovered}) = \{\, n^r s n^l \,\}, \Delta(\textit{calculus}) = \{\, n \,\}, \Delta(\textit{philosopher}) = \{\, d^r n \,\}.$$

We may construct a functor $L : G \to \mathbf{CB}(\Sigma)$ given on objects by $L(w) = L(s) = 1$ and $L(n) = x$, and on arrows by sending every lexical entry to the corresponding symbol in Σ except for the question word "who" which is interpreted as a cap and the functional word "that" which is interpreted as a spider with three outputs. Then one may check that the image of the sentence "Spinoza influenced a philosopher that discovered calculus" is grammatical in G and that the corresponding pregroup reduction is mapped via L to the last diagram in the derivation above.

Definition 2.4.30. `Entailment`

Input: $r \in \mathbf{RC}(G)(u, s), \quad r' \in \mathbf{RC}(G)(u', s), \quad L : \mathbf{RC}(G) \to \mathbf{C}$
Output: $L(r) \leq L(r')$

Proposition 2.4.31. `Entailment` *is undecidable for finitely presented Cartesian bicategories. When $\mathbf{C}$ is freely generated (i.e. it has no existential rules), the problem reduces to conjunctive query* `Containment`*.*

Proof. Entailment of conjunctive queries under existential rules is undecidable, see [BM02]. When $\mathbf{C} = \mathbf{CB}(\Sigma)$ is freely generated by a relational signature Σ, i.e. with no existential rules, theorem 2.4.17 yields a logspace reduction to `Containment`: `Entailment` $\in$ `NP`. □

We now consider the following computational problem: given a natural language corpus and a question, does the corpus contain an answer? We show how to translate a corpus into a relational database so that question answering reduces to query evaluation.

In order to translate a corpus into a relational database, it is not sufficient to parse every sentence independently, since the resulting queries will have disjoint sets of variables. The extra data that we need is a *coreference resolution*, which allows to link the common entities mentioned in these sentences. In 1.6.2, we defined a notion of pregroup grammar with coreference G which allows to represent a corpus of k sentences as one big diagram $C \in \mathbf{Coref}(G)$, assuming that both the pregroup parsing and the coreference resolution have been performed. In order to interpret a pregroup grammar with coreference $G = (V, B, \Delta, I, R, s)$ in a cartesian bicategory $\mathbf{C}$, it is sufficient to fix a CB model from the pregroup grammar (V, B, Δ, I, s) into $\mathbf{C}$ and choose an image for the reference types in R. Then the coreference resolution is interpreted using the Frobenius algebra in $\mathbf{C}$.

Fix a pregroup grammar with coreference G and a CB model $L : \mathbf{Coref}(G) \to \mathbf{CB}(\Sigma)$ with $L(s) = 0$, i.e. grammatical sentences are mapped to closed formulae. We assume that $L(q) = L(a)$ for q and a the question and answer types respectively, i.e. both are mapped to queries with the same number of free variable. Lexical items such as "influence" and "Leibniz" are mapped to their own symbol in the relational signature

Σ, whereas functional words such as relative pronouns are sent to the Frobenius algebra of $\mathbf{CB}(\Sigma)$, see 2.4.4 or [SCC13].

We describe a procedure for answering a question given a corpus. Suppose we are given a corpus $u \in V^*$ with k sentences. Parsing and resolving the coreference yields a morphism $C \in \mathbf{Coref}(G)(u, s^k)$. Using L we obtain a relational database from C given by the canonical structure induced by the corresponding query $K(C) = CM(L(C))$, we denote by $E := \mathtt{var}(L(C))$ the universe of this relational structure. Given a parsed question $g : v \to q \in \mathbf{Coref}(G)$, we can answer the question g by evaluating it in the model $K(C)$.

Definition 2.4.32. `QuestionAnswering`

Input: $C \in \mathbf{Coref}(G)(u, s^k), g \in \mathbf{Coref}(G)(v, q)$
Output: $\mathtt{Evaluation}(K, L(g)) \subseteq E^{\mathtt{fv}(L(g))}$ *where* $K = CM(L(C))$

Proposition 2.4.33. `QuestionAnswering` *is* $\mathtt{NP} - \mathtt{complete}$.

Proof. Membership follows immediately by reduction to `Evaluation`. Hardness follows by reduction from `Evaluation`. Indeed fix any relational structure K and query φ, using Proposition 1.6.10, we may build a corpus $C \in \mathbf{Coref}(G)$ and a question $g \in \mathbf{Coref}(G)$ such that $L(C) = K$ and $L(g) = \varphi$. □

Note that this is an asymptotic result. In practice, the questions we may ask are small compared to the size of the corpus. This would make the problem tractable since `Evaluation` is only `NP`-complete in the combined size of database and query, but it becomes polytime computable when the query is fixed [Tho13].

2.5 Tensor network models

Tensors arose in the work of Ricci and Levi-Civita in the end of the 19th century [RL00]. They were adopted by Einstein [Ein16], who used repeated indices to denote their compositions, and applied to quantum mechanics by Heisenberg [KH25] to describe the possible states of quantum systems.

In the 1970s, Penrose introduced a diagrammatic notation for manipulating tensor expressions [Pen71]: wires represent vector spaces, nodes represent multi-linear maps between them. This work was one of the main motivations behind Joyal and Street's graphical calculus for monoidal categories [JS91], which was later adopted in the development of Categorical Quantum Mechanics (CQM) [AC08]. The same notation is widely used in the Tensor Networks (TN) community [Eis13; BMT15; DEB19]. Until recently, CQM and TN remained separated fields because they were interested in different aspects of these graphical networks. On the one hand, rewriting and axiomatics. On the other, fast methods for tensor contraction and complexity theoretic guarantees. These two lines of research are now seeing a fruitful exchange of ideas as category theorists become more applied and vice-versa. For instance, rewriting strategies developed in the context of categorical quantum mechanics can be used to speed-up quantum computations [Kv20], or solve satisfiability and counting problems [dKM20; TM21].

The tools and methods developed by these communities are finding many applications in artificial intelligence. Tensor networks are widely used in machine learning, supported by efficient contraction tools such as Google's TensorNetwork library [EHL19], and are beginning to be applied to natural language processing [PV17; Zha+19]. Distributional Compositional models of meaning [CCS08; CCS10] (DisCoCat) arise naturally from Categorical Quantum Mechanics [AC08; CK17], In a nutshell, CQM provides us with graphical calculi to reason about tensor networks, and DisCoCat provides a way of mapping natural language to these calculi so that semantics is computed by tensor contraction.

In this section, we establish a formal connection between tensor networks and functorial models. This allows us to transfer complexity and tractability results from the TN literature to the DisCoCat models of meaning. In particular, we show that DisCoCat models based on dependency grammars can be computed in polynomial time. We end by discussing an extension of these tensor-based models where bubbles are used to represent non-linear operations on tensor networks.

2.5.1 Tensor networks

In this section, we review the basic notions of tensor networks. We take as a starting point the definition of tensor networks used in the complexity theory literature on TNs [AL10; OGo19; GK20].

Let us denote an undirected graph by (V, E) where V is a finite set of vertices and $E \subseteq \{\{u, v\} \mid u, v \in V\}$ is a set of undirected edges. The incidence set of a vertex v is $I(v) = \{e \in E \mid v \in e\}$ and the degree of v is the number of incident edges $\mathtt{deg}(v) = |I(v)|$. An order n tensor T of shape $(d_1, \ldots, d_n)$ with $d_i \in \mathbb{N}$ is a function

$T : [d_1] \otimes \cdots \otimes [d_n] \to \mathbb{S}$ where $[d_i] = \{ 1, \ldots, d_i \}$ is the ordered set with d_i elements and $\mathbb{S}$ is a semiring of numbers (e.g. $\mathbb{B}, \mathbb{N}, \mathbb{R}, \mathbb{C}$), see 2.1.

Definition 2.5.1 (Tensor network)**.** *A tensor network (V, E, T) over a semiring $\mathbb{S}$ is a undirected graph (V, E) with edge weights $\texttt{dim} : E \to \mathbb{N}$ and a set of tensors $T = \{ T_v \}_{v \in V}$ such that T_v is a tensor of order $\texttt{deg}(v)$ and shape $(\texttt{dim}(e_0), \ldots, \texttt{dim}(e_{\texttt{deg}(v)}))$ for $e_i \in I(v) \subseteq E$. Each edge $e \in E$ corresponds to an index $i \in [\texttt{dim}(e)]$ along which the adjacent tensors are to be contracted.*

The contration of two tensors $T_0 : [m] \otimes [d] \to \mathbb{S}$ and $T_1 : [d] \otimes [l] \to \mathbb{S}$ along their common dimension $[d]$ is a tensor $T_0 \cdot T_1 : [k] \otimes [l] \to \mathbb{S}$ with:

$$T_0 \cdot T_1(i, j) = \sum_{k \in [d]} T_0(i, k) T_1(k, l)$$

If $T_0 : [m] \to \mathbb{S}$ and $T_1 : [l] \to \mathbb{S}$ do not have a shared dimension then we still denote by $T_0 \cdot T_1 : [m] \otimes [l] \to \mathbb{S}$, the outer (or tensor) product of T_0 and T_1 given by $T_0 \cdot T_1(i, j) = T_0(i) T_1(j)$. Note that this is sufficient to define the product $T_0 \cdot T_1$ for tensors of arbitary shape since $(d_0, \ldots, d_n)$ shaped tensors are in one-to-one correspondence with order 1 tensors of shape $(d_0 d_1 \ldots d_n)$.

We are interested in the *value* $\texttt{contract}(V, E, T)$ of a tensor network which is the number in $\mathbb{S}$ obtained by contracting the tensors in (V, E, T) along their shared edges. We may define this by first looking at the order of contractions, called bubbling in [ALM07; AL10].

Definition 2.5.2 (Contraction order)**.** *[OGo19] A contraction order π for (V, E, T) is a total order of its vertices, i.e. a bijection $\pi : [n] \to V$ where $n = |V|$.*

Given a contraction order π for (V, E, T) we get an algorithm for computing the value of (V, E, T), given by $\texttt{contract}(V, E, T) = A_n$ where:

$$A_0 = 1 \in \mathbb{S} \qquad A_i = A_{i-1} \cdot T_{\pi(i)}$$

One may check that the value A_n obtained is independent of the choice of contraction order π, although the time and space required to compute the value may very well depend on the order of contractions [OGo19]. We can now define the general problem of contracting a tensor network.

Definition 2.5.3. $\texttt{Contraction}(\mathbb{S})$

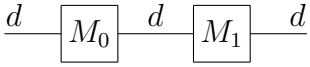
Input: *A tensor network (V, E, T) over $\mathbb{S}$*
Output: $\texttt{contract}(V, E, T)$

Proposition 2.5.4. $\texttt{Contraction}(\mathbb{S})$ *is $\#\texttt{P}$-complete for $\mathbb{S} = \mathbb{N}, \mathbb{R}^+, \mathbb{R}, \mathbb{C}$, it is $\texttt{NP}$-complete for $\mathbb{S} = \mathbb{B}$.*

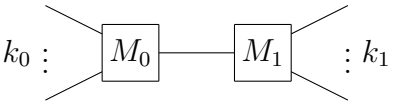
Proof. $\#\texttt{P}$-hardness was proved in [BMT15] by reduction from $\#\texttt{SAT}$. $\texttt{NP}$-completeness when $\mathbb{S} = \mathbb{B}$ follows by equivalence with conjunctive query evaluation. □

Even though contracting tensor networks is a hard problem in general, it becomes tractable, in several restricted cases of interest.

Let us look a bit more closely at the process of contraction. Recall that given two $n \otimes n$ matrices M_0, M_1 the time complexity of matrix multiplication is in general n^3. This is the simplest instance of a tensor contraction which can be depicted graphically as follows:

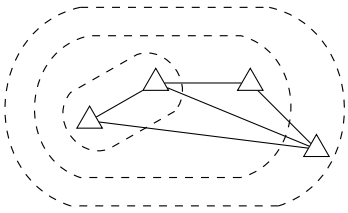


The standard way to compute this contraction is with two for loops over the outgoing wires for each basis vector in the middle wire, resulting in the cubic complexity. There are Strassen-like algorithms with a better asymptotic runtime, but we do not consider these here. Now suppose M_0 and M_1 are tensors with indices of dimension n connected by one edge, with k_0 and k_1 outgoing edges respectively:



Then in order to contract the middle wire, we need $k_0 + k_1$ for loops for each basis vector in the middle wire, resulting in the complexity $n^{k_0+k_1+1}$. In particular if we have 0 edges connecting the tensors, i.e. we are taking the outer product, then the time complexity is $O(n^{k_0+k_1})$. Similarly, if we have k_2 parallel edges connecting the tensors, the time complexity is $O(n^{k_0+k_1+k_2})$. We can speed up the computation by *slicing* [GK20], i.e. by parallelizing the computation over $k_2 \cdot n$ GPUs, where each computes the for loops for a basis vector of the middle wires, reducing the time complexity to $n^{k_0+k_1}$. This is for instance enabled by Google's Tensor Network library [EHL19].

For a general tensor network (V, E, T), we have seen that the contraction strategy can be represented by a contraction order $\pi : [|V|] \to V$, this may be viewed as a *bubbling* on the graph (V, E):



At each contraction step i, we are contracting the tensor A_{i-1} enclosed by the bubble with $T_{\pi(i)}$. From the considerations above we know that the time complexity will depend on the number of edges crossing the bubble at each time step. For instance in the bubbling above, there are at most 3 edges crossing a bubble at each time step, We can capture this complexity using the notion of *bubblewidth* [ALM07], also known as the cutwidth [OGo19], of a tensor network.

Definition 2.5.5 (Bubblewidth)**.** *The bubblewidth of a contraction order $\pi : [n] \to V$ for a graph (V, E) is given by:*

$$\texttt{bubblewidth}(\pi) = \texttt{max}_{j \in [n]} \, |\{\{ \, \pi(j), \pi(k) \, \} \in E \, | \, i \leq j < k\}|)$$

The bubblewidth of a tensor network (V, E) is the minimum bubblewidth of a contraction order π for (V, E):

$$BW(V, E) = \mathtt{min}_{\pi:[n]\to V}(\mathtt{bubblewidth}(\pi))$$

Interestingly, the bubblewidth of a graph can be used to bound from above the much better studied notions of treewidth and pathwidth. Let us denote by $PW(V, E)$ and $TW(V, E)$ the path width and tree width of a graph (V, E).

Proposition 2.5.6 (Aharonov et al. [ALM07])**.**

$$TW(V, E) \leq PW(V, E) \leq 2BW(V, E)$$

The following proposition is a weaker statement then the results of O'Gorman [OGo19], that we can prove using the simple considerations about contraction time given above.

Proposition 2.5.7. *Any tensor network (V, E, T) can be contracted in $O(n^{BW(d)+a})$ time where $n = \mathtt{max}_{e\in E}(\mathtt{dim}(e))$ is the maximum dimension of the edges, $BW(d)$ is the bubble width of d and $a = \mathtt{max}_{v\in V}(\mathtt{deg}(v))$ is the maximum order of the tensors in T.*

Proof. At each contraction step i, we are contracting the tensor A_{i-1} enclosed by the bubble with $T_{\pi(i)}$. From the considerations above we know that the time complexity of such a contraction is at most $O(n^{w_i+\mathtt{deg}(\pi(i))})$ where w_i is the number of edges crossing the i-th bubble and $\mathtt{deg}(\pi(i))$ is the degree of the i-th vertex in the contraction order π. The overall time complexity is the sum of these terms, which is at most of the order $O(n^{BW(d)+a})$. □

Therefore tensor networks of bounded bubblewidth may be evaluated efficiently.

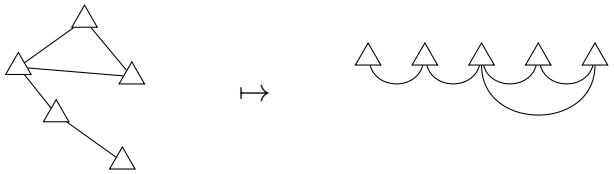
Corollary 2.5.8. $\mathtt{Contraction}(\mathbb{S})$ *can be computed in poly-time if the input tensor networks have bounded bubblewidth and bounded vertex degree.*

2.5.2 Tensor functors

We now show that tensor networks may be reformulated in categorical language as diagrams in a free compact-closed category equipped with a functor into the category of matrices over a commutative semiring $\mathbf{Mat}_{\mathbb{S}}$, defined in 2.1. From this perspective, the contraction order π provides a way of turning the compact-closed diagram into a premonoidal one, where the order of contraction is the linear order of the diagram.

Proposition 2.5.9. *There is a semantics-preserving translation between tensor networks (V, E, T) over $\mathbb{S}$ and pairs (F, d) of a diagram $d : 1 \to 1 \in \mathbf{CC}(\Sigma)$ in a free compact closed category and a monoidal functor $F : \Sigma \to \mathbf{Mat}_{\mathbb{S}}$. This translation is semantics-preserving in the sense that $\mathtt{contract}(V, E, T) = F(d)$.*

Proof. Given a tensor network (V, E, T) we build the signature $\Sigma = V \xrightarrow{\texttt{cod}} E^*$ where $\texttt{cod}(v) = I(v)$ is (any ordering of) the incidence set of vertex v, i.e. vertices correspond to boxes and edges to generating objects. The set of tensors $\{ T_v \}_{v \in V}$ induces a monoidal functor $F : \Sigma \to \mathbf{Mat}_{\mathbb{S}}$ given on objects $e \in E$ by $F(e) = \texttt{dim}(e)$ and on boxes $v \in V$ by $F(v_i) = T_i : 1 \to \texttt{dim}(e_0) \otimes \cdots \otimes \texttt{dim}(e_{\texttt{deg(v)}})$ for $e_i \in \texttt{cod}(v_i) = I(v)$. We build a diagram $d : 1 \to 1 \in \mathbf{CC}(\Sigma)$ by first tensoring all the boxes in Σ and then composing it with a morphism made up only of cups and swaps, where there is a cap connecting an output port of box $v \in \Sigma$ with an output port of $v' \in \Sigma$, whenever $\{ v, v' \} \in E$.



One can check that $F(d) = \texttt{contract}(V, E, T)$ since composing $T_0 \otimes T_1 : 1 \to m \otimes n \otimes n \otimes l$ with $\texttt{id}_m \otimes \texttt{cup}_n \otimes \texttt{id}_l$ in $\mathbf{Mat}_{\mathbb{S}}$ corresponds to contracting the tensors T_0 and T_1 along their common dimension n. For the other direction, it is easy to see that any closed diagram in $\mathbf{CC}(\Sigma)$ induces a graph with vertices Σ and edges given by the struture of the diagram. Then a functor $F : \Sigma \to \mathbf{Mat}_{\mathbb{S}}$ yields precisely the data of the tensors $\{ T_v \}_{v \in \Sigma}$ where $T_v = F(v)$. □

It is often useful to allow a special type of vertex in the tensor network. These are called *COPY tensors* in the tensor network literature [BMT15; OGo19; GK20], where they are used for optimizing tensor contraction. They also appear throughout categorical quantum mechanics [CK17] — and most prominently in the ZX calculus [van20] — where they are called *spiders*. We have seen in 1.6.1 that diagrams with spiders are morphisms in a free hypergraph category $\mathbf{Hyp}(\Sigma)$. Since $\mathbf{Hyp}(\Sigma) \simeq \mathbf{CC}(\Sigma + \texttt{Frob})/\cong$ (see 1.6.1), Proposition 2.5.9 can be used to show that there is a semantics-preserving translation between tensor networks with COPY tensors and pairs (F, d) of a diagram $d \in \mathbf{Hyp}(\Sigma)$ and a functor $F : \Sigma \to \mathbf{Mat}_{\mathbb{S}}$.

Example 2.5.10 (Conjunctive queries as TNs)**.** *Conjunctive queries are tensor networks over* $\mathbb{B}$*. Tensor networks also subsume probabilistic graphical models by taking the underlying category to be* $\mathbf{Prob}$ *or* $\mathbf{Mat}_{\mathbb{R}^+}$ *[Gla+19].*

We now consider the problem of evaluating closed diagrams $d : 1 \to 1 \in \mathbf{CC}(\Sigma)$ using a tensor functor $F : \Sigma \to \mathbf{Mat}_{\mathbb{S}}$

Definition 2.5.11. $\texttt{FunctorEval}(\mathbb{S})$

Input: Σ *a monoidal signature,* $d : 1 \to 1 \in \mathbf{CC}(\Sigma)$*,* $F : \Sigma \to \mathbf{Mat}_{\mathbb{S}}$*,*
Output: $F(d)$

Proposition 2.5.12. $\texttt{FunctorEval}(\mathbb{S})$ *is equivalent to* $\texttt{Contraction}(\mathbb{S})$

Proof. This follows from Proposition 2.5.9. □

The notion of contraction order has an interesting categorical counterpart. It gives a way of turning a compact-closed diagram into a premonoidal one, such that the width of the diagram is the bubblewidth of the contraction order.

Proposition 2.5.13. *There is a semantics-preserving translation between tensor networks (V, E, T) over $\mathbb{S}$ with a contraction order $\pi : [|V|] \to V$ and pairs (F, d) of a diagram $d : 1 \to 1 \in \mathbf{PMC}(\Sigma + \mathtt{swap})$ in a free premonoidal category with swaps and a monoidal functor $F : \Sigma \to \mathbf{Mat}_{\mathbb{S}}$. This translation is semantics-preserving in the sense that $\mathtt{contract}(V, E, T) = F(d)$. Moreover, we have that*

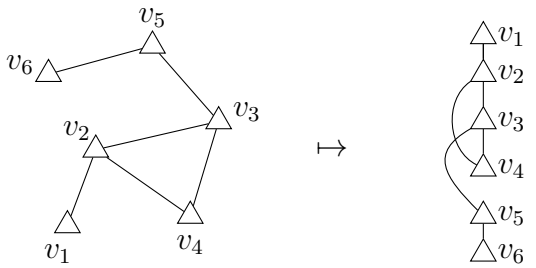
$$\mathtt{bubblewidth}(\pi) = \mathtt{width}(d)$$

with $\mathtt{width}(d)$ as defined in 1.3.4.

Proof. We can use Proposition 2.5.9 to turn (V, E, T) into a compact closed diagram $d : 1 \to 1 \in \mathbf{CC}(\Sigma)$ over a signature Σ with only output types for each box. Given a contraction order $\pi : [n] \to V$, we modify the signature Σ by transposing a output port $e = \{ v, v' \} \in E$ of $v \in V$ into an input whenever $\pi^{-1}(v) > \pi^{-1}(v')$ forming a signature Σ_π Then we can construct a premonoidal diagram

$$\pi(d) = \circ_{i=1}^{k}(\mathtt{perm}_i \otimes b_i) \in \mathbf{PMC}(\Sigma_\pi + \mathtt{swap})$$

where $\mathtt{perm}_i \in \mathbf{PMC}(\mathtt{swap})$ is a morphism containing only swaps and identities. This is done by ordering the boxes according to π and pushing all the wires to the left, as in the following example:



(2.22)

Note that no cups or caps need to be added since the input/output types of the boxes in the signature have been changed accordingly. The bubblewidth of π is then precisely the width of the resulting premonoidal diagram. □

We may define the bubblewidth of a diagram $d \in \mathbf{CC}(\Sigma)$ as the bubblewidth of the corresponding graph via Proposition 2.5.9. Also we define the dimension of a functor $F : \Sigma \to \mathbf{Mat}_{\mathbb{S}}$ as follows:

$$\mathtt{dim}(F) = \mathtt{max}_{x \in \Sigma_0}(F(x))$$

We may now derive the consequences of Proposition 2.5.8 in this categorical context.

Proposition 2.5.14. $\mathtt{FunctorEval}(\mathbb{S})$ *can be computed in polynomial time if the input diagrams d have bounded bubblewidth and the input functors F have bounded dimension.*

Proof. This follows from the conjunction of Proposition 2.5.8 and the reduction from `FunctorEval` to `Contraction` of Proposition 2.5.12. □

2.5.3 DisCoCat and bounded memory

DisCoCat models were introduced by Coecke et al. in 2008 [CCS08; CCS10]. In the original formalism, the authors considered functors from a pregroup grammar to the category of finite dimensional real vector spaces and linear maps (i.e. $\mathbf{Mat}_{\mathbb{R}}$). In this work, we follow [Coe+18; dMT20] in treating DisCoCat models as functors into categories of matrices over any semiring. These in particular subsume the relational models studied in 2.4 by taking $\mathbb{S} = \mathbb{B}$. As shown in Section 2.1, $\mathbf{Mat}_{\mathbb{S}}$ is a compact-closed category for any commutative semiring. Therefore, the most suitable choice of syntax for this semantics are *rigid grammars*, for which we have a canonical way of interpreting the cups and caps. Thus, in this section, the grammars we consider are either pregroup grammars 1.5.1, or dependency grammars 1.5.2, or pregroup grammar with crossed dependencies 1.5.1 or with coreference 1.6.2.

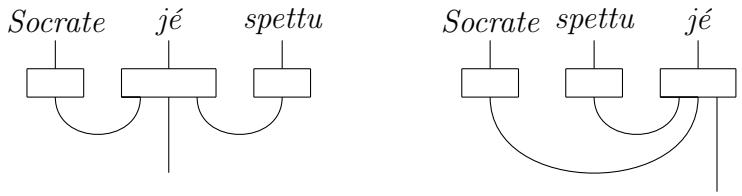
Definition 2.5.15. *A DisCoCat model is a monoidal functor $F : G \to \mathbf{Mat}_{\mathbb{S}}$ for a rigid grammar G and a commutative semiring $\mathbb{S}$. The semantics of a sentence $g : u \to s \in \mathbf{RC}(G)$ is given by its image $F(g)$. We assume that $F(w) = 1$ for $w \in V$ so that*

Distributional models $F : G \to \mathbf{Mat}_{\mathbb{R}}$ can be constructed by counting co-occurences of words in a corpus [GS11]. The image of the noun type $n \in B$ is a vector space where the inner product computes noun-phrase similarity [SCC13]. When applied to question answering tasks, distributional models can be used to compute the distance between a question and its answer [Coe+18].

Example 2.5.16. *As an example we may take the pregroup lexicon:*

$$\Delta(\textit{Socrate}) = \Delta(\textit{spettu}) = \{\, n \,\} \quad \Delta(\textit{jé}) = \{\, n^r s n^l, n^r n^r s \,\}$$

And we may define a DisCoCat model $F : \Delta \to \mathbf{Mat}_{\mathbb{R}}$ with $F(n) = 2$, $F(s) = 1$ and $F(\textit{Socrate}, n) = (1, 0) : 1 \to 2$, $F(\textit{spettu}, n) = (-1, 0)$ and $F(\textit{jé}, n^r s n^l) = F(\textit{jé}, n^r n^r s) = \mathtt{cap}_2 : 1 \to 2 \otimes 2$ in $\mathbf{Mat}_{\mathbb{R}}$. Then the following grammatical sentences:



evaluate to a scalar in $\mathbb{R}$ given by the matrix multiplication:

$$F(\text{“Socrate jé spettu”}) = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ 0 \end{bmatrix} = -1 = F(\text{“Socrate spettu jé”})$$

We are interested in the complexity of the following semantics problem.

Definition 2.5.17. $\mathtt{TensorSemantics}(G, \mathbb{S})$

Input: $g \in \mathbf{RC}(G)(u, s)$, $F : G \to \mathbf{Mat}_{\mathbb{S}}$
Output: $F(g)$

Given a sentence diagram $g \in \mathcal{L}(G)$, the pair (g, F) forms a tensor network. Computing the semantics of g in F amounts to contracting this tensor network. Therefore $\texttt{TensorSemantics}(G, \mathbb{S}) \in \#\texttt{P}$. For a general rigid grammar, this problem is $\#\texttt{P}$-hard, since it subsumes the $\texttt{FunctorEval}(\mathbb{S})(\Sigma)$ problem. When G is a pregroup grammar with coreference, we can construct any tensor network as a list of sentences connected by the coreference, making the $\texttt{TensorSemantics}(G, \mathbb{S})$ problem $\#\texttt{P} - \texttt{complete}$.

Proposition 2.5.18. $\texttt{TensorSemantics}(G, \mathbb{S})$ *where G is a pregroup grammar with coreference is equivalent to* $\texttt{Contraction}(\mathbb{S})$.

Proof. One direction follows by reduction to $\texttt{FunctorEval}(\mathbb{S})$ since any model $F : G \to \mathbf{Mat}_{\mathbb{S}}$ factors through a free compact-closed category. For the other direction, fix any pair (F, d) of a diagram $d : x \to y \in \mathbf{CC}(\Sigma)$ and a functor $F : \Sigma \to \mathbf{Mat}_{\mathbb{S}}$. Using Proposition 1.6.10 we can build a corpus $C : u \to s^k \in \mathbf{Coref}(G)$, where the connectivity of the tensor network is encoded a coreference resolution, so that the canonical functor $\mathbf{Coref}(G) \to \mathbf{CC}(\Sigma)$ maps C to d. □

From a linguistic perspective, a contraction order for a grammatical sentence $g : u \to s$ gives a reading order for the sentence and the bubblewidth is the maximum number of tokens (or basic types) that the reader should hold in memory in order to parse the sentence. Of course, in natural language there is a natural reading order from left to right which induces a "canonical" bubbling of g. In light of the famous psychology experiments of Miller [Mil56], we expect that the short-term memory required to parse a sentence is bounded, and more precisely that $BW(g) = 7 \pm 2$. For pregroup diagrams generated by a dependency grammar, it is easy to show that the bubblewidth is bounded.

Proposition 2.5.19. *The diagrams in $\mathbf{RC}(G)$ generated by a dependency grammar G have bubblewidth bounded by the maximum arity of a rule in G.*

Proof. Dependency relations are acyclic, and the bubblewidth for an acyclic graph is smaller than the maximum vertex degree of the graph, which is equal to the maximum arity of a rule in G, i.e. the maximum nuber of dependents for a symbol plus one. □

Together with Proposition 2.5.14, we deduce that the problem of computing the tensor semantics of sentences generated by a dependency grammar is tractable.

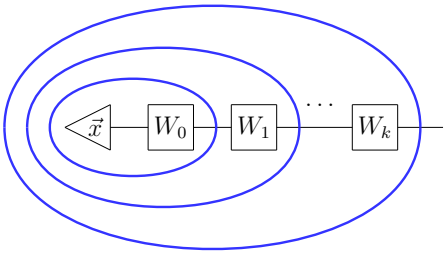
Corollary 2.5.20. *If G is a dependency grammar then* $\texttt{TensorSemantics}(G)$ *can be computed in polynomial time.*

For a general pregroup grammar we also expect the generated diagrams to have bounded bubblewidth, even though they are not acyclic. For instance, the cyclic pregroup reductions given in 1.21 have constant bubblewidth 4, which is obtained by chosing a contraction order from the middle to the sides, even though the naive contraction order of 1.21 from left to right has unbounded bubblewidth. We end with a cojecture, since we were unable to show pregroup reductions have bounded bubblewidth in general.

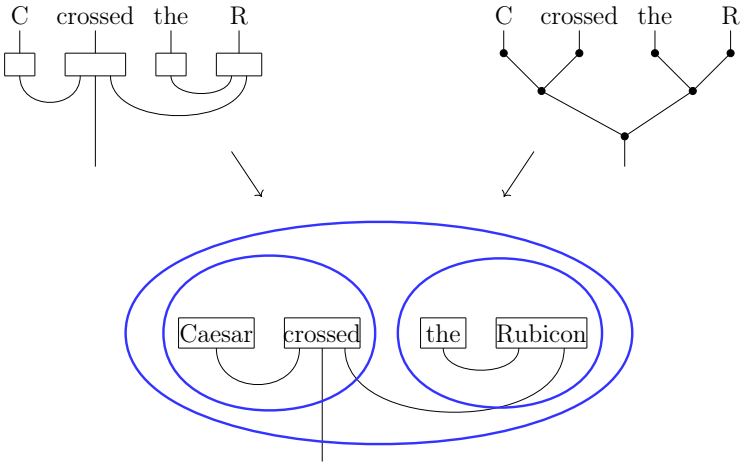
Conjecture 2.5.21. *Diagrams generated by a pregroup grammar G have bounded bubble width and thus* $\texttt{TensorSemantics}(G, \mathbb{S})$ *can be computed in polynomial time.*

2.5.4 Bubbles

We end this section by noting that a simple extension of tensor network models allows to recover the full expressive power of neural networks. We have seen that contraction strategies for tensor networks can be represented by a pattern of bubbles on the diagram. These bubbles did not have any semantic interpretation and they were just used as brackets, specifying the order of contraction. We could however give them semantics, by interpreting bubbles as operators on tensors. As an example, consider the following diagram, where each box W_i is interpreted as a matrix:



Suppose that each bubble acts on the tensor it encloses by applying a non-linearity σ to every entry. Then we see that this diagram specifies a neural network of depth k and width n where n is the maximum dimension of the wires. With this representation of neural networks, we have more control over the structure of the network.



Suppose we have a pregroup grammar G_0 and context-free grammar G_1 over the same vocabulary V and let $u \in \mathcal{L}(G_0) \cap \mathcal{L}(G_1)$ with two parses $g_0 : u \to s$ in $\mathbf{RC}(G_0)$ and $g_1 : u \to s$ in $\mathbf{O}(G_1)$. Take the skeleton of the context-free parse g_1, i.e. the tree without labels. This induces a pattern of bubbles, as described in Example 1.2.4 on Peirce's alpha. Fix a tensor network model $G_0 \to \mathbf{Mat}_{\mathbb{S}}$ for the pregroup grammar with $F(w) = 1$ for $w \in V$ and choose an activation function $\sigma : \mathbb{S} \to \mathbb{S}$. Then σ defines a unary operator on homsets $\mathcal{S}_\sigma : \prod_{x,y\in\mathbb{N}} \mathbf{Mat}_{\mathbb{S}}(x, y) \to \prod_{x,y\in\mathbb{N}} \mathbf{Mat}_{\mathbb{S}}(x, y)$ called a *bubble*, see [HS20] or [TYd21] for formal definitions. We can combine the pregroup reduction with the pattern of bubbles as in the example above. We may

then compute the semantics of u as follows. Starting from the inner-most bubble, we contract the tensor network enclosed by it and apply the activation σ to every entry in the resulting tensor. This gives a new bubbled network where the inner-most bubble has been contracted into a box. And we repeat this process. The procedure described above gives a way of controlling the non-linearities applied to tensor networks from the grammatical structure of the sentence. While this idea is not formalised yet, it points to a higher diagrammatic formalism in which diagrams with bubbles are used to control tensor computations. This diagrammatic notation can be traced back to Peirce and was recently used in [HS20] to model negation in first-order logic and in [TYd21] to model differentiation of quantum circuits. We will also use it in the next section to represent the non-linearity of knowledge-graph embeddings and in Chapter 3 to represent the softmax activation function.

2.6 Knowledge graph embeddings

Recent years have seen the rapid growth of web-scale knowledge bases such as Freebase [Bol+08], DBpedia [Leh+14] and Google's Knowledge Vault [Don+14]. These resources of structured knowledge enable a wide range of applications in NLP, including semantic parsing [Ber+13], entity linking [HOD12] and question answering [BCW14]. Knowledge base *embeddings* have received a lot of attention in the statistical relational learning literature [Wan+17]. They approximate a knowledge base continuously given only partial access to it, simplifying the querying process and allowing to predict missing entries and relations — a task known as *knowledge base completion*.

Knowledge graph embedding are of particular interest to us since they provide a link between the Boolean world of relations and the continuous world of tensors. They will thus provide us with a connection between the relational models of Section 2.4 and the tensor network models of Section 2.5. We start by defining the basic notions of knowledge graph embeddings. Then we review three factorization models for KG embedding, focusing on their expressivity and their time and space complexity. This will lead us progressively from the category of relations to the category of matrices over the complex numbers with its convenient factorization properties.

2.6.1 Embeddings

Most large-scale knowledge bases encode information according to the Resource Description Framework (RDF), where data is stored as triples $(\text{head}, \text{relation}, \text{tail})$ (e.g. $(\text{Obama}, \text{BornIn}, \text{US})$). Thus a *knowledge graph* is just a set of triples $K \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$ where $\mathcal{E}$ is a set of entities and $\mathcal{R}$ a set of relations. This form of knowledge representation can be seen as an instance of relational databases where all the relations are *binary*.

Proposition 2.6.1. *Knowledge graphs are in one-to-one correspondence with functors* $\Sigma \to \mathbf{Rel}$ *where* $\Sigma = \mathcal{R}$ *is a relational signature containing only symbols of arity two.*

Proof. This is easy to see, given a knowledge graph $K \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$, we can build a functor $F : \mathcal{R} \to \mathbf{Rel}$ defined on objects by $F(1) = \mathcal{E}$ and on arrows by $F(r) = \{\, (e_0, e_1) | (e_0, r, e_1) \in K \,\} \subseteq \mathcal{E} \times \mathcal{E}$. Similarly any functor $F : \Sigma \to \mathbf{Rel}$ defines a knowledge graph $K = \{\, (\pi_1 F(r), r, \pi_2 F(r)) \,|\, r \in \Sigma \,\} \subseteq F(1) \times \Sigma \times F(1)$. □

Higher-arity relations can be encoded into the graph through a process known as *reification*. To reify a k-ary relation R, we form k new binary relations R_i and a new entity e_R so that $R(e_1, \ldots, e_k)$ is true iff $\forall i$ we have $R_i(e_R, e_i)$. Most of the literature on embeddings focuses on knowledge graphs, and the results which we present in this section follow this assumption. However, some problems with reification have been pointed out in the literature, and current research is aiming at extending the methods to knowledge *hypergraphs* [Fat+20], a direction which we envisage also for the present work.

Embedding a knowledge graph consists in the following learning problem. Starting from a knowledge graph $K : \mathcal{E} \times \mathcal{R} \times \mathcal{E} \to \mathbb{B}$, the idea is to approximate K by a scoring function $X : \mathcal{E} \times \mathcal{R} \times \mathcal{E} \to \mathbb{R}$ such that $\|\sigma(X) - K\|$ is minimized where $\sigma : \mathbb{R} \to \mathbb{B}$ is any *activation* function.

The most popular activation functions for knowledge graph embeddings are approximations of the `sign` function which takes a real number to its sign ± 1, where -1 is interpreted as the Boolean 0 (false) and 1 is interpreted as the Boolean 1 (true). In machine learning applications, one needs a differentiable version of `sign` such as `tanh` or the sigmoid function. In this section, we are mostly interested in the expressive power of knowledge graph embeddings, and thus we define "exact" embeddings as follows.

Definition 2.6.2 (Embedding)**.** *An* exact *embedding for a knowledge graph* $K \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$*, is a tensor* $X : \mathcal{E} \times \mathcal{R} \times \mathcal{E} \to \mathbb{R}$ *such that* $\mathtt{sign}(X) = K$*.*

Even though `sign` is not a homomorphism of semirings between $\mathbb{R}$ and $\mathbb{B}$, it allows to define a notion of sign rank which has interesting links to measures of learnability such as the VC-dimension [AMY16].

Definition 2.6.3 (Sign rank)**.** *Given* $R \in \mathbf{Mat}_{\mathbb{B}}$ *the sign rank of* R*, denoted* $\mathtt{rank}_{\pm}(R)$ *is given by:*

$$\mathtt{rank}_{\pm}(R) = \mathtt{min}\{\mathtt{rank}(X) \,|\, \mathtt{sign}(X) = R\,,\, X \in \mathbf{Mat}_{\mathbb{R}}\}$$

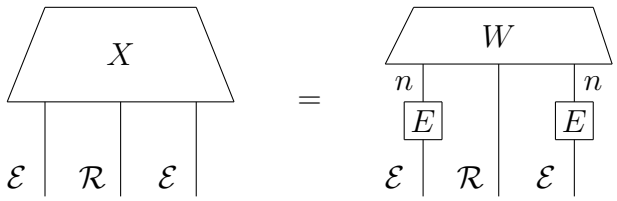
The sign rank of a relation R is often much lower than its rank. In fact, we don't know any relation $R : n \to n$ with $\mathtt{rank}_{\pm}(R) > \sqrt{n}$. The identity Boolean matrix $n \to n$ has sign rank 3 for any n [AMY16]. Therefore any permutation of the rows and columns of an identity matrix has sign rank 3, an example is the relation "is married to". For the factorization models studied in this section, this means that the dimension of the embedding is potentially much smaller than then the number of entities in the knowledge graph.

Several ways have been proposed for modeling the scoring function X, e.g. using translations [Bor+13] or neural networks [Soc+13b]. We are mostly interested in *factorization* models where $X : \mathcal{E} \times \mathcal{R} \times \mathcal{E} \to \mathbb{R}$ is treated as a tensor with three outputs, i.e. a state in $\mathbf{Mat}_{\mathbb{R}}$. Assuming that X admits a factorization into smaller matrices allows to reduce the search space for the embedding while decreasing the time required to predict missing entries. The space complexity of the embedding is the amount of memory required to store X. The time complexity of an embedding is the time required to compute $\sigma(X\,|s, v, o\rangle)$ given a triple. These measures are particularly relevant when it comes to embedding integrated large-scale knowledge graphs. The problem then boils down to finding a good *ansatz* for such a factorization, i.e. an assumed factorization shape that reduces the time and space complexity of the embedding.

2.6.2 Rescal

The first factorization model for knowledge graph embedding — known as `Rescal` — was proposed by Nickel et al. in 2011 [NTK11]. It models the real tensor X with the

following ansatz:

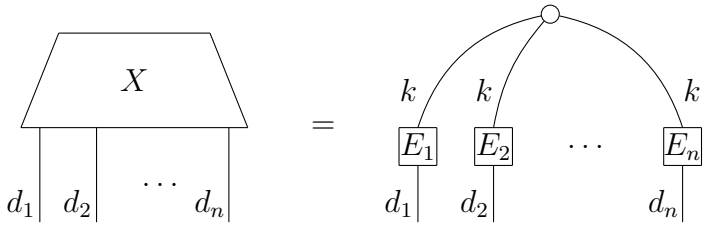


(2.23)

where $E : |\mathcal{E}| \to n$ is the embedding for entities, $W : n \otimes |\mathcal{R}| \otimes n \to 1$ is a real tensor, and n is a hyper-parameter determining the dimension of the embedding.

The well-known notion of rank can be used to get bounds on the dimension of an embedding model. It is formalised diagrammatically as follows.

Definition 2.6.4 (Rank)**.** *The rank of a tensor $X : 1 \to \otimes_{i=1}^{n} d_i$ in $\mathbf{Mat}_{\mathbb{S}}$ is the smallest dimension k such that X factors as $X = \Delta_k^n ; \otimes_{i=1}^{n} E_i$ where $E_i : k \to d_i$ and Δ_k^n is the spider with no inputs and n outputs of dimension k.*

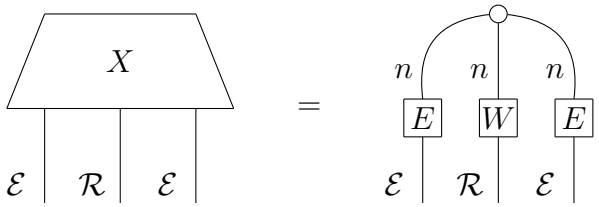


The lowest the rank, the lowest the search space for a factorization. In fact if X is factorized as in 2.23 then $\texttt{rank}(X) = \texttt{rank}(W) \leq |\mathcal{R}|\, n^2$. The time and space complexity of the embedding are thus both quadratic in the dimension of the embedding n.

Since any three-legged real tensor X can be factorized as in 2.23 for some n, `Rescal` is a very expressive model and performs well. However, the quadratic complexity is a limitation which can be avoided by looking for different ansatze.

2.6.3 DistMult

The idea of `DistMult` [Yan+15] is to factorize the tensor X via *joint orthogonal diagonalization*, i.e. the entities are embedded in a vector space of dimension n and relations are mapped to diagonal matrices on $\mathbb{R}^n$. Note that the data required to specify a diagonal matrix $n \to n$ is just a vector $v : 1 \to n$ which induces a diagonal matrix $\Delta(v)$ by composition with the frobenius algebra $\Delta : n \to n \otimes n$ in $\mathbf{Mat}_{\mathbb{R}}$. `DistMult` starts from the assumption — or ansatz — that X can is factorized as follows:



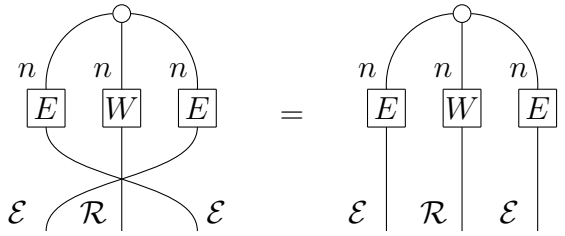
(2.24)

○

where $E : |\mathcal{E}| \to n$ and $W : |\mathcal{R}| \to n$ and $\Delta : n \otimes n \otimes n \to 1$ is the spider with three legs. Note that $\texttt{rank}_{\pm}(K) \leq \texttt{rank}(X) = \texttt{rank}(\Delta) = n$, so that both the time and space complexity of `DistMult` are linear $O(n)$. `DistMult` obtained the state of the art on `Embedding` when it was released. It performs especially well at modeling *symmetric* relations such as the transitive verb "met" which satisfies "x met y" iff "y met x". It is not a coincidence that `DistMult` is good in these cases since a standard linear algebra result says that a matrix $Y : n \to n$ can be orthogonally diagonalized if and only if Y is a symmetric matrix. For tensors of order three we have the following genralization, with the consequence that `DistMult` models *only* symmetric relations.

Proposition 2.6.5. *$X : 1 \to |\mathcal{E}| \times |\mathcal{R}| \times |\mathcal{E}|$ is jointly orthogonally diagonalizable if and only if it is a family of symmetric matrices indexed by R.*

Proof.



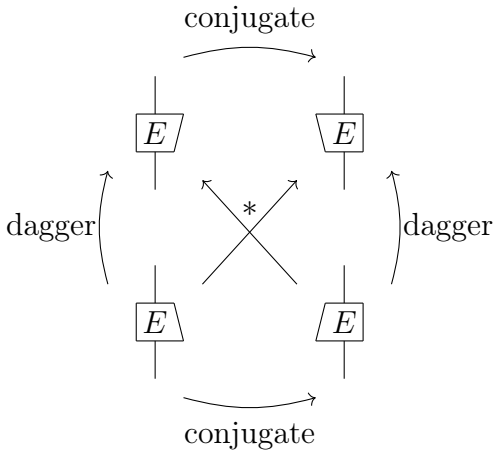
□

This called for a more expressive factorization method, allowing to represent *asymmetric* relations such as "love" for which we cannot assume that "x loves y" implies "y loves x".

2.6.4 ComplEx

Trouillon et al. [Tro+17] provide a factorization of any real tensor X into complex-valued matrices, that allows to model symmetric and asymmetric relations equally well. Their model, called `Complex` allows to embed any knowledge graph K in a low-dimensional complex vector space. We give a diagrammatic treatment of their results, working in the category $\mathbf{Mat}_{\mathbb{C}}$, allowing us to improve their results. The difference between working with real-valued and complex-valued matrices is that the latter have additional structure on morphisms given by conjugation. This is pictured, at the diagrammatic level, by an asymmetry on the boxes which allows to represent the operations of adjoint, transpose and conjugation as rotations or reflexions of the diagram. This graphical gadget was introduced by Coecke and Kissinger in the

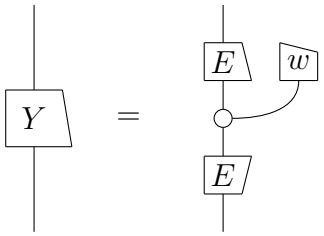
context of quantum mechanics [CK17], we summarize it in the following picture:



(2.25)

The key thing that Trouillon et al. note is that any real square matrix is the real-part of a diagonalizable complex matrix, a consequence of Von Neumann's *spectral theorem.*

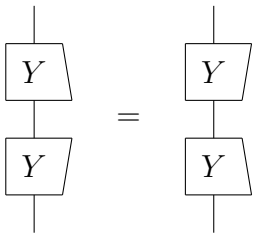
Definition 2.6.6. *$Y : n \to n$ in $\mathbf{Mat}_{\mathbb{C}}$ is unitarily diagonalizable if it factorizes as:*



(2.26)

where $w : 1 \to n$ is a state and $E : n \to n$ is a unitary.

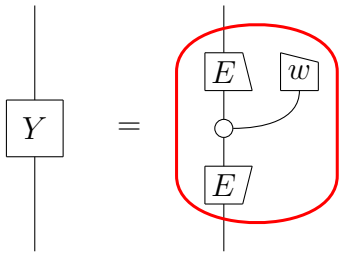
Definition 2.6.7. *Y is normal if it commutes with its adjoint: $YY^\dagger = Y^\dagger Y$.*



Theorem 2.6.8 (Von Neumann [Von29])**.** *Y is diagonalizable if and only if Y is normal.*

Proposition 2.6.9 (Trouillon [Tro+17])**.** *Suppose $Y : n \to n$ in $\mathbf{Mat}_{\mathbb{R}}$ is a real square matrix, then $Z = Y + iY^T$ is a normal matrix and $Re(Z) = Y$. Therefore there is a unitary E and a diagonal complex matrix W such that $Y = Re(EWE^\dagger)$.*

Graphically we have:



(2.27)

where the red bubble indicates the real-part operator.

Note that $\texttt{rank}(A+B) \leq \texttt{rank}(A) + \texttt{rank}(B)$ which implies that $\texttt{rank}(Z) = \texttt{rank}(Y + iY^T) \leq 2\texttt{rank}(Y)$.

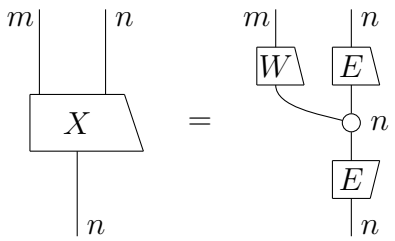
Corollary 2.6.10. *Suppose $Y : n \to n$ in $\mathbf{Mat}_{\mathbb{R}}$ and $\texttt{rank}(X) = k$, then there is $E : n \to 2k$ and $W : 2k \to 2k$ diagonal in $\mathbf{Mat}_{\mathbb{C}}$ such that $Y = Re(EWE^{\dagger})$*

Given a binary relation $R : |\mathcal{E}| \to |\mathcal{E}|$, the sign-rank gives a bound on the dimension of the embedding.

Corollary 2.6.11. *Suppose $R : |\mathcal{E}| \to |\mathcal{E}|$ in $\mathbf{Mat}_{\mathbb{B}}$ and $\texttt{rank}_{\pm}(R) = k$, then there is $E : |\mathcal{E}| \to 2k$ and $W : 2k \to 2k$ diagonal in $\mathbf{Mat}_{\mathbb{C}}$ such that $R = \texttt{sign}(Re(EWE^{\dagger}))$*

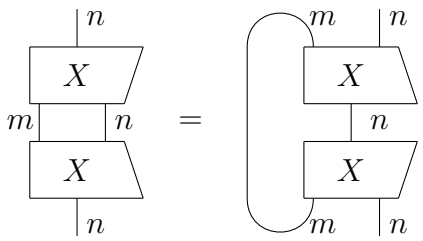
The above reasoning works for a single binary relation R. However, given knowledge graph $K \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$ and applying the reasoning above we will get $|\mathcal{R}|$ different embeddings $E_r : |\mathcal{E}| \to n_r$, one for each relation $r \in \mathcal{R}$, thus obtaining only the following factorization: $K = \texttt{sign}(\sum_{r \in \mathcal{R}} \Delta \circ (E_r \otimes W(r) \otimes E_r)$. Which means that the overall complexity of the embedding is $O(|\mathcal{R}|\, n)$ where $n = \texttt{max}_{r \in \mathcal{R}}(n_r)$. The problem becomes: can we find a single embedding $E : |\mathcal{E}| \to n$ such that all relations are mapped to diagonal matrices over the same n? In their paper, Trouillon et al. sketch a direction for this simplification but end up conjecturing that such a factorization does *not* exist. We answer their conjecture negatively, by proving that for any real tensor X of order 3, the complex tensor $Z = X + iX^T$ is jointly unitarily diagonalizable.

Definition 2.6.12. *$X : n \otimes m \to n$ in $\mathbf{Mat}_{\mathbb{C}}$ is simultaneously unitarily diagonalizable if there is a unitary $E : n \to n$ and $W : m \to n$ such that:*



(2.28)

Definition 2.6.13. *$X : n \otimes m \to n$ is a commuting family of normal matrices if:*



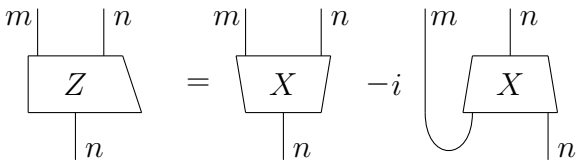
(2.29)

The following is an extension of Von Neumann's theorem to the multi-relational setting.

Theorem 2.6.14. *[HJ12] $X : n \otimes m \to n$ in $\mathbf{Mat}_{\mathbb{C}}$ is a commuting family of normal matrices if and only if it is simultaneously unitarily diagonalizable.*

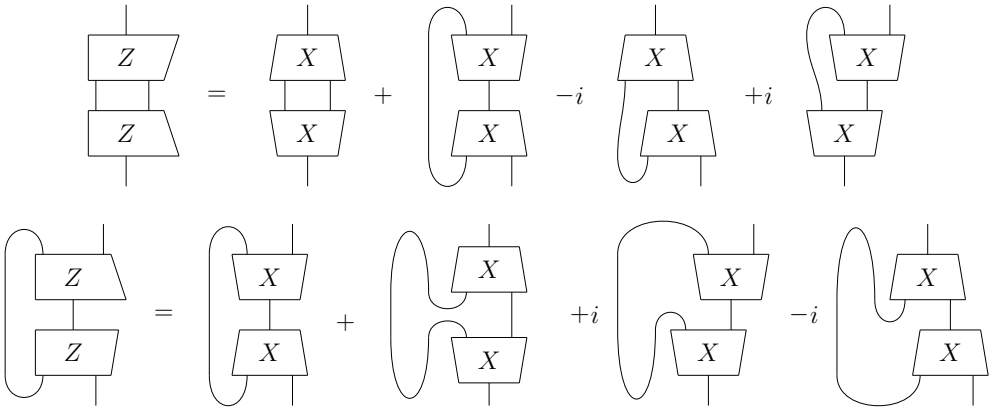
Our contribution is the following result.

Proposition 2.6.15. *For any real tensor $X : n \otimes m \to n$ the complex tensor Z defined by:*



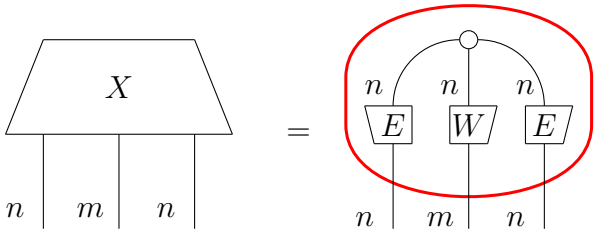
is a commuting family of normal matrices.

Proof. This follows by checking that the two expressions below are equal, using only the snake equation:



□

Corollary 2.6.16. *For any real tensor $X : 1 \to n \otimes m \otimes n$ there is a unitary $E : n \to n$ and $W : n \to m$ in $\mathbf{Mat}_{\mathbb{C}}$ such that:*



(2.30)

where the bubble denotes the real-part operator.

Proposition 2.6.17. *For any knowledge graph $K \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$ of sign rank $k = \texttt{rank}_{\pm}(K)$, there is $E : |\mathcal{E}| \to 2k$ and $W : |\mathcal{R}| \to 2k$ in $\mathbf{Mat}_{\mathbb{C}}$ such that*

$$K = \texttt{sign}(Re(\Delta \circ (E \otimes W \otimes E^*))).$$

This results in an improvement of the bound on the size of the factorization by a factor of $|\mathcal{R}|$. Although this improvement is only incremental, the diagrammatic tools used here open the path for a generalisation of the factorization result to higher-order tensors which would allow to model higher-arity relations. This may require to move into quaternion valued vector spaces where (at least) ternary symmetries can be encoded.

Moreover, as we will show in Section 2.7, quantum computers allow to speed up the multiplication of complex valued tensors. An implementation of `ComplEx` on a quantum computer was proposed by [Ma+19]. The extent to which quantum computers can be used to speed up knowledge graph embeddings is an interesting direction of future work.

2.7 Quantum models

Quantum computing is an emerging model of computation which promises speed-up on certain tasks as compared to classical computers. With the steady growth of quantum hardware, we are approaching a time when quantum computers perform tasks that cannot be done on classical hardware with reasonable resources. Quantum Natural Language Processing (QNLP) is a recently proposed model which aims to meet the data-intensive requirements of NLP algorithms with the computational power of quantum hardware [ZC16; Coe+20; Mei+20]. These models arise naturally from the categorical approaches to linguistics [CCS10] and quantum mechanics [AC08]. They can in fact be seen as instances of the tensor network models studied in 2.5. We have tested them on noisy intermediate-scale quantum computers [Mei+20; Lor+21], obtaining promising results on small-scale datasets of around 100 sentences. However, the crucial use of post-selection in these models, is a limit to their scalability as the number of sentences grows.

In this section, we study the complexity of the quantum models based on a mapping from sentences to pure quantum circuits [Mei+20; Lor+21]. Building on the work of Arad and Landau [AL10] on the complexity of tensor network contraction, we show that the *additive* approximation (with scale $\Delta = 1$) of quantum models is a complete problem for `BQP`, the class of problems which can be solved in polynomial time by a quantum computer with a bounded probability of error. Note that this approximation scheme has severe limits when the amplitude we want to approximate is small compared to the scale Δ. Thus the results may be seen as a negative statement about the first generation of QNLP models. However, specific types of linguistic structure (such as trees) may allow to reduce the post-selection and thus the approximation scale. Moreover, this puts QNLP on par with well-known `BQP`-complete problems such as approximate counting [Bor+09] and the evaluation of topological invariants [FLW00; FKW02] to set a standard for the next generations of QNLP models.

The development of DisCoPy was driven and motivated by the implementation of these QNLP models. The `quantum` module of DisCoPy is described in [TdY22], it features interfaces with the `tensor` module for classical simulation, with PyZX [Kv19] for optimization and with tket [Siv+20] for compilation and evaluation on quantum hardware. In order to run quantum machine learning routines, we also developed diagrammatic tools for automatic differentiation [TYd21]. The pipeline for performing QNLP experiments with DisCoPy has been packaged in Lambeq [Kar+21] which provides state-of-the-art categorial parsers and optimised classes for training QNLP models.

2.7.1 Quantum circuits

In this section, we give a brief introduction to the basic ingredients of quantum computing, and define the categories **QCirc** and **PostQCirc** of quantum circuits and their post-selected counterparts. A proper introduction to the field is beyond the scope of this thesis and we point the reader to [CK17] and [van20] for diagrammatic treatments.

The basic unit of information in a quantum computer is the *qubit*, whose *state space* is the Bloch sphere, i.e. the set of vectors $\psi = \alpha\,|0\rangle + \beta\,|1\rangle \in \mathbb{C}^2$ with norm $\|\psi\| = |\alpha|^2 + |\beta|^2 = 1$. Quantum computing is inherently probabilistic, we never observe the coefficients α and β directly, we only observe the probabilities that outcomes 0 or 1 occur. These probabilities are given by the *Born rule*:

$$P(i) = |\langle i|\psi\rangle|^2$$

with $i \in \{\,0, 1\,\}$ and where $\langle i|\psi\rangle$ is the inner product of $|i\rangle$ and $|\psi\rangle$ in $\mathbb{C}^2$, also called *amplitude*. Note that the requirement that ψ be of norm 1 ensures that these probabilities sum to 1. The *joint state* of n qubits is given by the tensor product $\psi_1 \otimes \cdots \otimes \psi_n$ and thus lives in $\mathbb{C}^{2n}$, a Hilbert space of dimension 2^n. The evolution of a quantum system composed of n qubits is described by a *unitary* linear map U acting on the space $\mathbb{C}^{2n}$, i.e. a linear map satisfying $UU^\dagger = \mathtt{id} = U^\dagger U$. Where $\dagger$ (dagger) denotes the conjugate transpose. Note that the requirement that U be unitary ensures that $\|U\psi\| = \|\psi\|$ and so U sends quantum states to quantum states.

The unitary map U is usually built up as a *circuit* from some set of basic *gates*. Depending on the generating set of gates, only certain unitaries can be built. We say that the set of gates is *universal*, when any unitary can be obtained using a finite sequence of gates from this set. The following is an example of a universal gate-set:

$$\mathtt{Gates} = \{\,CX,\ H,\ Rz(\alpha),\ \mathtt{swap}\,\}_{\alpha\in[0,2\pi]} \tag{2.31}$$

where CX is the controlled X gate acting on 2 qubits and defined on the basis of $\mathbf{C}^4$ by:

$$CX(|00\rangle) = |00\rangle \quad CX(|01\rangle) = |01\rangle \quad CX(|10\rangle) = |11\rangle \quad CX(|11\rangle) = |10\rangle$$

$Rz(\alpha)$ is the Z phase, acting on 1 qubit as follows:

$$Rz(\alpha)(|0\rangle) = |0\rangle \qquad Rz(\alpha)(|1\rangle) = e^{i\alpha}\,|1\rangle\ .$$

H is the Hadamard gate, acting on 1 qubit as follows:

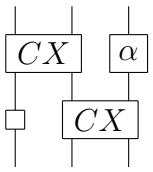
$$H(|0\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \qquad H(|1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\,.$$

and the `swap` gate acts on 2 qubits and is defined as usual.

We define the category of quantum circuits $\mathbf{QCirc} = \mathbf{MC}(\mathtt{Gates})$ as a free monoidal category with objects natural numbers n and arrows generated by the gates in (2.31).

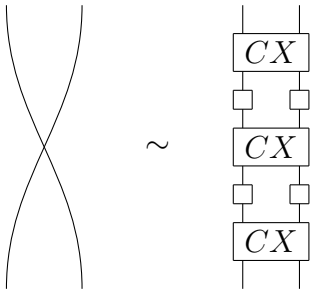
Definition 2.7.1 (Quantum circuit)**.** *A quantum circuit is a diagram* $c : n \to n \in \mathbf{MC}(\mathtt{Gates}) = \mathbf{QCirc}$ *where* n *is the number of qubits, it maps to a corresponding unitary* $U_c : (\mathbb{C}^2)^{\otimes n} \to (\mathbb{C}^2)^{\otimes n} \in \mathbf{Mat}_{\mathbb{C}}$.

Example 2.7.2. *An example of quantum circuit is the following:*

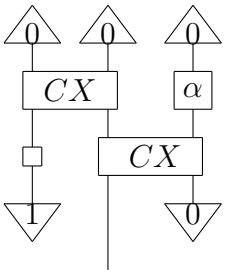


(2.32)

where we denote the $Rz(\alpha)$ gate using the symbol α and the Hadamard gate using a small white box. Note that different quantum circuits c and c' may correspond to the same unitary, we write $c \sim c'$ when this is the case. For instance the swap gate is equivalent to the following composition:



When performing a quantum computation on n qubits, the quantum system is usually prepared in an all-zeros states $|\mathbf{0}\rangle = |0\rangle^{\otimes n}$, then a quantum circuit c is applied to it and measurements are performed on the resulting state $U_c |\mathbf{0}\rangle$, yielding a probability distribution over bitstrings of length n. We can then post-select on certain outcomes, or predicates over the resulting bitstring, by throwing away the measurements that do not satisfy this predicate. Diagrammatically, state preparations and post-selections are drawn using the generalised Dirac notation [CK17] as in the following post-selected circuit:



(2.33)

The category of post-selected quantum circuits is obtained from **QCirc** by allowing state preparations $|0\rangle, |1\rangle$, and post-selections $\langle 0|, \langle 1|$.

$$\mathbf{PostQCirc} = \mathbf{MC}(\texttt{Gates} + \{\, \langle i|, |i\rangle \,\}_{i\in 0,1} + \{\, a : 0 \to 0 \,\}_{a\in\mathbb{R}})$$

Note that we also allow arbitrary scalars $a \in \mathbb{R}$, seen as boxes $0 \to 0$ in **PostQCirc** to rescale the results of measurements, this is needed in order to interpret pregroup grammars in **PostQCirc**, see 2.7.2. Post-selected quantum circuits map functorially into linear maps:

$$I : \mathbf{PostQCirc} \to \mathbf{Mat}_{\mathbb{C}}$$

by sending each gate in **Gates** to the corresponding unitary, state preparations and post-selections to the corresponding states and effects in $\mathbf{Mat}_{\mathbb{C}}$. This makes post-selected quantum circuits instances of tensor networks over $\mathbb{C}$.

Proposition 2.7.3. *For any morphism $d : 1 \to 1$ in* **PostQCirc**, *there exists a quantum circuit $c \in$ **QCirc** such that $I(d) = a \cdot \langle \mathbf{0} | U_c | \mathbf{0} \rangle$ where $a = \prod_i U(a_i)$ is the product of the scalars appearing in d.*

Proof. We start by removing the scalars a_i from the diagram d and multiplying them into $a = \prod_i U(a_i)$. Then we pull all the kets to the top of the diagram and all the bras to the bottom, using `swap` if necessary. □

2.7.2 Quantum models

Quantum models are functors from a syntactic category to the category of post-selected quantum circuits **PostQCirc**. These were introduced in recent papers [Mei+21; Coe+20], and experiments were performed on IBM quantum computers [Mei+20; Lor+21].

Definition 2.7.4 (Quantum model)**.** *A quantum (circuit) model is a functor $F : G \to$ **PostQCirc** for a grammar G, the semantics of a sentence $g \in \mathcal{L}(G)$ is given by a distribution over bitstrings $b \in \{0, 1\}^{F(s)}$ obtained as follows:*

$$P(b) = |\langle \mathbf{b} | F(g) | \mathbf{0} \rangle|^2$$

Quantum models define the following computational problem:

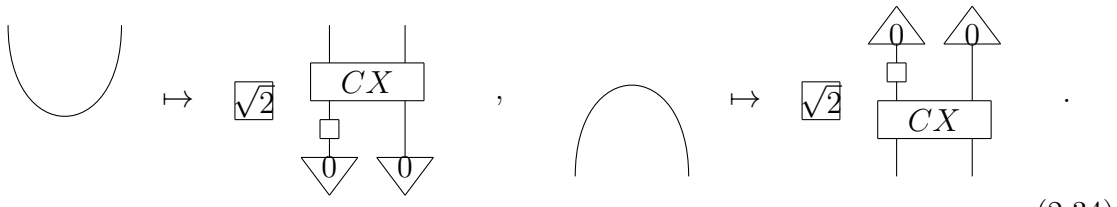
Definition 2.7.5. `QSemantics`

Input: *G a grammar, $g : u \to s \in \mathbf{MC}(G)$, $F : G \to$ **PostQCirc**, $b \in \{0, 1\}^{F(s)}$*

Output: $|\langle \mathbf{b} | I(F(g)) | \mathbf{0} \rangle|^2$

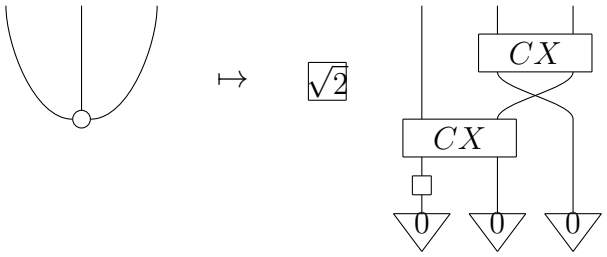
Remark 2.7.6. *Note that quantum language models do* not *assume that the semantics of words be a unitary matrix. Indeed a word may be interpreted as a unitary with some outputs post-selected. However, mapping words to unitaries is justified in many cases of interest in linguistics. For instance, this implies that there is no loss of information about "ball" when an adjective such as "big" is applied to it.*

Any of the grammars studied in Chapter 1 may be intepreted with a quantum model. For monoidal grammars (including regular and context-free grammars), this is simply done by interpreting every box as a post-selected quantum circuit. For rigid grammars, we need to choose an interpretation for the cups and caps. Since the underlying category $\mathbf{Mat}_{\mathbb{C}}$ is compact-closed, there is a canonical way of interpreting them using a CX gate and postselection, and re-scaling the result by a factor of $\sqrt{2}$:



(2.34)

In order to interpret cross dependencies we can use the `swap` gate, see Example 2.7.2. Finally, in order to interpret the Frobenius algebras in a pregroup grammar with coreference, we use the following mapping:



One can check the the image under I of the circuit above maps to the Frobenius algebra in $\mathbf{Mat}_{\mathbb{C}}$.

In practice, quantum models are learned from data by choosing a parametrized quantum circuit in **PostQCirc** for each production rule and then tuning the parameters (i.e. the phases α in $Rz(\alpha)$) appearing in this circuit, see [Mei+20; Lor+21] for details on ansatze and training.

2.7.3 Additive approximations

Since quantum computers are inherently probabilistic, there is no deterministic way of computing a function $F : \{0,1\}^* \to \mathbb{C}$ encoded in the amplitudes of a quantum state. Rather what we obtain is an *approximation* of F. In many cases, the best we can hope for is an *additive* approximation, which garantees to generate a value within the range $[F(x) - \Delta(x)\epsilon, F(x) + \Delta(x)\epsilon]$ where $\Delta : \{0,1\}^* \to \mathbb{R}^+$ is an approximation scale. This approximation scheme has been found suitable to describe the performance of quantum algorithms for contracting tensor network [AL10], counting approximately [Bor+09], and computing topological invariants [FLW00] including the Jones polynomial [FKW02].

Definition 2.7.7 (Additive approximation)**.** *[AL10] A function $F : \{0,1\}^* \to \mathbb{C}$ has an additive approximation V with an approximation scale $\Delta : \{0,1\}^* \to \mathbb{R}^+$ if there exists a probabilistic algorithm that given any instance $x \in \{0,1\}^*$ and $\epsilon > 0$, produces a complex number $V(x)$ such that*

$$P(|V(x) - F(x)| \geq \epsilon\Delta(x)) \leq \frac{1}{4} \tag{2.35}$$

in a running time that is polynomial in $|x|$ and ϵ^{-1}.

Remark 2.7.8. *The error signal ϵ is usually inversely proportional to a polynomial in the run-time t of the algorithm, i.e. we have $\epsilon = \frac{1}{\mathtt{poly}(t)}$.*

An algorithm V satisfying the definiton above is a solution of the following problem defined for any function $F : \{0,1\}^* \to \mathbb{C}$ and approximation scale $\Delta : \{0,1\}^* \to \mathbb{R}^+$.

Definition 2.7.9. $\mathtt{Approx}(F, \Delta)$

Input: $x \in \{0,1\}^*$, $\epsilon > 0$
Output: $v \in \mathbb{C}$ *such that* $P(|v - F(x)| \geq \epsilon\Delta(x)) \leq \frac{1}{4}$

Compare this to the definition of a multiplicative approximation, also known as a fully polynomial randomised approximation scheme [JSV04].

Definition 2.7.10 (Multiplicative approximation)**.** *A function $F : \{0,1\}^* \to \mathbb{C}$ has a multiplicative approximation V if there exists a probabilistic algorithm that given any instance $x \in \{0,1\}^*$ and $\epsilon > 0$, produces a complex number $V(x)$ such that*

$$P(|V(x) - F(x)| \geq \epsilon\,|F(x)|) \leq \frac{1}{4} \tag{2.36}$$

in a running time that is polynomial in $|x|$ and ϵ^{-1}.

Remark 2.7.11. *This approximation is called multiplicative because $V(x)$ is guaranteed to be within a multiplicative factor $(1 \pm \epsilon)$ of the optimal value $F(x)$. For instance the inequality in (2.36) implies that $|F(x)|\,(1-\epsilon) \leq |V(x)| \leq |F(x)|\,(1+\epsilon))$ with probability bigger than $\frac{3}{4}$.*

Note that any multiplicative approximation is also additive with approximation scale $\Delta(x) = |F(x)|$. However, the converse does not hold. An additive approximation scheme allows for the approximation scale $\Delta(x)$ to be exponentially larger than the size of the output $|F(x)|$, making the approximation (2.35) quite loose since the error parameter ϵ may be bigger than the output signal $V(x)$.

2.7.4 Approximating quantum models

In this section, we show that the problem of additively approximating the evaluation of sentences in a quantum model is in `BQP` and that it is `BQP`-complete in special cases of interest. The argument is based on Arad and Landau's work [AL10] on the quantum approximation of tensor networks. We start by reviewing their results and end by demonstrating the consequences for quantum language models.

Consider the problem of approximating the contraction of tensor networks $T(V, E)$ using a quantum computer. Arad and Landau [AL10] show that this problem can be solved in polynomial time, up to an additive accuracy with a scale Δ that is related to the norms of the swallowing operators.

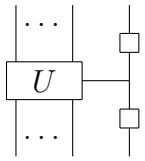
Proposition 2.7.12 (Arad and Landau [AL10])**.** *Let $T(V, E)$ be a tensor network over $\mathbb{C}$ of dimension q and maximal node degree a, let $\pi : [|V|] \to V$ be a contraction order for T and let $\{A_i\}_{i \in \{1,\ldots,k\}}$ be the corresponding set of swallowing operators. Then for any $\epsilon > 0$ there exists a quantum algorithm that runs in $k \cdot \epsilon^{-2} \cdot \texttt{poly}(q^a)$ time and outputs a complex number r such that:*

$$P(|\texttt{value}(T) - r| \geq \epsilon\Delta) \leq \frac{1}{4}$$

with

$$\Delta(T) = \prod_{i=1}^{k} \|A_i\|$$

Proof. Given a tensor network with a contration order π, the swallowing operators A_i are linear maps. For each of them, we can construct a unitary U_i acting on a larger space such that post-selecting on some of its outputs yields A_i. Composing these we obtain a unitary $U_c = U_1 \cdot U_2 \dots U_{|V|}$ represented by a poly-size quantum circuit $c \in \mathbf{QCirc}$ such that $\texttt{value}(T) = \langle \mathbf{0} | U_c | \mathbf{0} \rangle$. In order to compute an approximation of this quantity we can use an H-test, defined by the following circuit:



(2.37)

where the white boxes are hadamard gates and the subdiagram in the middle denotes the controlled unitary U. It can be shown that the probability r of measuring 0 on the ancillary qubit is equal to $Re(\langle \mathbf{0} | U_c | \mathbf{0} \rangle)$. A slightly modified version of the H-test computes $Im(\langle \mathbf{0} | U_c | \mathbf{0} \rangle)$. Arad and Landau [AL10] show that this process can be done in polynomial time and that the result of measuring the ancillary qubit gives an additive approximation of $\texttt{value}(T)$ with approximation scale $\Delta(T) = \prod_{i=1}^{k} \|F(d_i)\|$. □

Corollary 2.7.13. *The problem* $\texttt{Approx}(\texttt{Contraction}(\mathbb{C}), \Delta)$ *with* Δ *as defined above is in* BQP.

Remark 2.7.14. *From the Cauchy-Schwartz inequality we have that:*

$$|T| \leq \prod_{i=1}^{k} \|A_i\| = \Delta(T) \,.$$

In fact we have no guarantee that the approximation scale $\Delta(T)$ *is not exponentially larger than* $|F(d)|$*. This is a severe limitation, since the approximations we get from the procedure defined above can have an error larger than the value we are trying to approximate.*

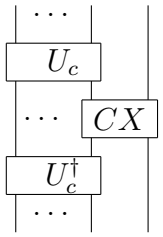
We now consider the problem of approximating the amplitude of a post-selected quantum circuit $|\langle \mathbf{0} | U_c | \mathbf{0} \rangle|^2$. Note that this is an instance of the problem studied in the previous paragraph, since post-selected quantum circuits are instances of tensor networks, so that this problem belong to the class BQP. For this subclass of tensor networks the approximation scale $\Delta(c)$ can be shown to be constant and equal to 1, with the consequence — shown by Arad and Landau [AL10] — that the additive approximation of post-selected quantum circuits is $\texttt{BQP} - \texttt{hard}$.

In order to show BQP-hardness of a problem F, it is sufficient to show that an oracle which computes F can be used to perform universal quantum computation with bounded error. More precisely, for any quantum circuit $u \in \mathbf{QCirc}$ denote by p_0 the probability of obtaining outcome 0 when measuring the last qubit of c. To perform universal quantum computation, it is sufficient to distinguish between the cases where $p_0 < \frac{1}{3}$ and $p_0 > \frac{2}{3}$ for any quantum circuit c. Thus a problem F is

BQP-hard if for any circuit c, there is a poly-time algorithm V using F as an oracle that returns YES when $p_0 < \frac{1}{3}$ with probability bigger than $\frac{3}{4}$ and NO when $p_0 > \frac{2}{3}$ with probability bigger than $\frac{3}{4}$. .

Proposition 2.7.15. *[AL10] The problem of finding an additive approximation of* $|\langle \mathbf{0}| U_c |\mathbf{0}\rangle|^2$ *with scale* $\Delta = 1$ *where* U_c *is the unitary induced by a quantum circuit* $c \in \mathbf{QCirc}$ *is* BQP*-complete.*

Proof. Membership follows by reduction to $\texttt{Approx}(\texttt{Contraction}(\mathbb{S}), \Delta)$, since post-selected quantum circuits are instances of tensor networks. To show hardenss, fix any quantum circuit c on n qubits and denote by p_0 the probability of obtaining outcome 0 on the last qubit of c. We can construct the quantum circuit:



(2.38)

it is easy to check that $\langle \mathbf{0}| U_q |\mathbf{0}\rangle = p_0$. Since q is a circuit, there is a natural contraction order given by the order of the gates, moreover the corresponding swallowing operators U_{q_i} are unitary and thus $\|U_{q_i}\| = 1$, also $\|U(|\mathbf{0}\rangle)\| = \|U(\langle \mathbf{0}|)\| = 1$ and therefore the approximation scale is constant $\Delta(q) = 1$. Suppose we have an oracle V that computes an approximation of $\langle \mathbf{0}| U_q |\mathbf{0}\rangle$ with constant scale $\Delta(q) = 1$. Then for any circuit $c \in \mathbf{QCirc}$, we construct the circuit q and apply V to get a complex number $V(q)$ such that:

$$P(|V(q) - p_0| \geq \epsilon) \leq \frac{1}{4} \implies P(p_0 - \epsilon \leq |V(q)| \leq p_0 + \epsilon) \geq \frac{3}{4}$$

Setting $\epsilon < \frac{1}{6}$, we see that if $|V(q)| < \frac{1}{6}$ then $p_0 < \frac{1}{3}$ with probability $\geq \frac{3}{4}$ and similarly $|V(q)| > \frac{5}{6}$ implies $p_0 > \frac{1}{3}$ with probability $\geq \frac{3}{4}$. Note that this would not be possible if we didn't know that the approximation scale $\Delta(q)$ is bounded. Thus V can be used to distinguish between the cases where $p_0 < \frac{1}{3}$ and $p_0 > \frac{2}{3}$ with probability of success greater than $\frac{3}{4}$. Therefore the oracle V can be used to perform universal quantum computation with bounded error. And thus V is BQP-hard. □

Corollary 2.7.16. $\texttt{Approx}(\texttt{FunctorEval}(\mathbb{C})(I), \Delta = 1)$ *where* $I : \mathbf{PostQCirc} \to \mathbf{Mat}_{\mathbb{C}}$ *is the functor defined in paragraph 2.7.1 is a* BQP*-complete problem.*

Proof. Membership follows by reduction to $\texttt{Approx}(\texttt{Contraction}(\mathbb{C}), \Delta)$ since post-selected quantum circuits are instances of tensor networks. Since $\mathbf{QCirc} \hookrightarrow \mathbf{PostQCirc}$ the problem of Proposition 2.7.15 reduces to $\texttt{Approx}(\texttt{FunctorEval}(I), \Delta = 1)$, thus showing hardness. □

Note that the value $v = |\langle \mathbf{0}| U_c |\mathbf{0}\rangle|$ may be exponentially small in the size of the circuit c, so that the approximation scale $\Delta = 1$ is still not optimal. In this case we

would need exponentially many samples from the quantum computer approximate v up to multiplicative accuracy.

We end by showing BQP-completeness for the problem of approximating the semantics of sentences in a quantum model with approximation scale $\Delta = 1$.

Definition 2.7.17. $\texttt{QASemantics} = \texttt{Approx}(\texttt{QSemantics}, \Delta = 1)$

Input: G *a monoidal grammar,* $g : u \to s \in \mathbf{MC}(G)$, $F : G \to \mathbf{PostQCirc}$, $b \in \{0, 1\}^{F(s)}$, $\epsilon > 0$

Output: $v \in \mathbb{C}$ *such that* $P(|v - \langle \mathbf{b}| I(F(g)) |\mathbf{0}\rangle| \geq \epsilon) \leq \frac{1}{4}$

Proposition 2.7.18. *There are pregroup grammars G, such that the problem* $\texttt{QASemantics}(G)$ *is* BQP*-complete.*

Proof. Membership follows by reduction to the problem of Proposition 2.7.15, since the semantics of a grammatical reduction $g : u \to s$ in $\mathbf{RC}(G)$ is given by the evaluation of $|\langle \mathbf{b}| I(F(g)) |\mathbf{0}\rangle|^2$ which corresponds to evaluating $|\langle \mathbf{0}| U_c |\mathbf{0}\rangle|^2$ where c is defined by Proposition 2.7.3. To show hardness, let G have only one word w of type s, fix any unitary U, and define $F : G \to \mathbf{PostQCirc}$ by $F(s) = 0$ and $F(w) = \langle \mathbf{0}| U |\mathbf{0}\rangle$, then evaluating the semantics of w reduces to the problem of Proposition 2.7.15 and is thus $\texttt{BQP} - \texttt{hard}$. □

Note that we were able to show completeness using the fact that the functor F is in the input of the problem. It is an open problem to show that $\mathbf{QASemantics}(G)(F)$ is BQP-complete for fixed choices of grammar G and functors F. In order to show this, one may need to assume that G is a pregroup grammar with coreference and show that there is a functor F such that any post-selected quantum circuit can be built up using fixed size circuits (corresponding to words) connected by GHZ states (corresponding to the spiders encoding the coreference), adapting the argument from Proposition 2.5.12.

Moreover, as discussed above, this approximation scheme is limited by the fact that the approximation scale Δ may be too big when the output $\langle \mathbf{b}| I(F(g)) |\mathbf{0}\rangle$ is exponentially small. This is particularly significant at the beginning of training, when F is initialised as a random mapping to circuits, see [Mei+20]. This seems to be an inherent problem caused by the use of post-selection in the model, although methods to reduce the post-selection have been proposed, e.g. the snake removal scheme from [Mei+21].

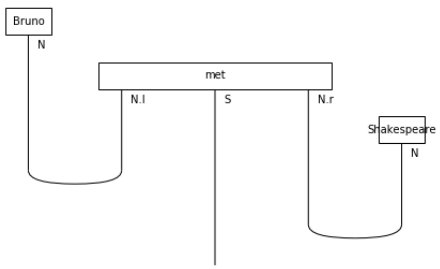
One avenue to overcome this limitation, is to consider a different typ of models, defined as functors $G \to \mathbf{CPM}(\mathbf{QCirc})$ where $\mathbf{CPM}(\mathbf{QCirc})$ is the category of completely positive maps induced by quantum circuits as defined in [HV19]. In this category, post-selection is not allowed since every map is causal. The problem with these models however is that we cannot interpret the cups and caps of rigid grammars. We may still be able to interpret monoidal grammars, as well as acyclic pregroup reductions such as those induced by a dependency grammar. Exploring the complexity of these models, and testing them on quantum hardware, is left for future work.

2.8 DisCoPy in action

We now give an example of how DisCoPy can be used to solve a concrete task. We define a relational model (2.4) and then learn a smaller representation of the data as a tensor model (2.5). Since the sentences we will deal with are all of the form subject-verb-object, this means we will perform a knowledge ambedding task in the sense of 2.6. We start by defining a simple pregroup grammar with 3 nouns and 2 verbs.

Listing 2.8.1. Subject-verb-object language.

```
from discopy.rigid import Ty, Id, Box, Diagram

n, s = Ty('n'), Ty('s')
make_word = lambda name, ty: Box(name, Ty(), ty)
nouns = [make_word(name, n) for name in ['Bruno', 'Florio', 'Shakespeare']]
verbs = [make_word(name, n.l @ s @ n.r) for name in ['met', 'read']]
grammar = Diagram.cups(n, n.l) @ Id(s) @ Diagram.cups(n.r, n)
sentences = [a @ b @ c >> grammar for a in nouns for b in verbs for c in nouns]
sentences[2].draw()
```



We can now build a relational model for this language as a `tensor.Functor`.

Listing 2.8.2. Relational model in DisCoPy.

```
from discopy.tensor import Dim, Tensor, Functor, Spider
import jax.numpy as np
Tensor.np = np

ob = {n: Dim(3), s: Dim(1)}
def mapping(box):
    if box.name == 'Bruno':
        return np.array([1, 0, 0])
    if box.name == 'Florio':
        return np.array([0, 1, 0])
    if box.name == 'Shakespeare':
        return np.array([0, 0, 1])
    if box.name == 'met':
        return np.array([[1, 0, 1], [0, 1, 1], [1, 1, 1]])
    if box.name == 'read':
        return np.array([[1, 0, 0], [0, 1, 1], [0, 1, 1]])
    if box.name == 'who':
```

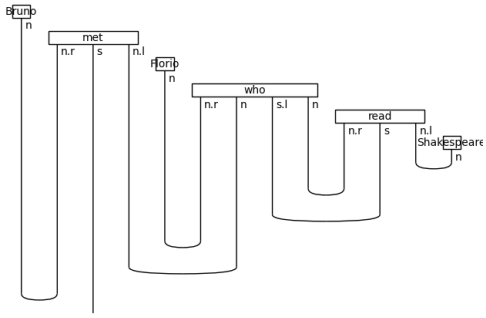
```
        return Spider(0, 3, Dim(3)).array
T = Functor(ob, mapping)
assert T(sentences[2]).array == [0.]
```

We use `float` *numbers for simplicity, but one may use* `dtype=bool` *instead. Note the special intepretation of "who" as a Frobenius spider, see 2.4.4.*

Relational models can be used to evaluate any conjunctive query over words. We can generate a new pregroup reduction using `lambeq` [Kar+21] and evaluate it in `T`.

Listing 2.8.3. Evaluating a conjunctive query in a relational model.

```
from lambeq import BobcatParser
parser = BobcatParser()
diagram = parser.sentence2diagram('Bruno met Florio who read Shakespeare.')
diagram.draw()
assert T(diagram) = [1.0]
```



We now show how to embed the three-dimensional data defined by `T` as a two-dimensional `tensor.Functor` with `float` entries. We start by parametrising two-dimensional tensor functors.

Listing 2.8.4. Parametrising a tensor functor.

```
import numpy

def p_mapping(box, params):
    if box.name == 'Bruno':
        return np.array(params[0])
    if box.name == 'Florio':
        return np.array(params[1])
    if box.name == 'Shakespeare':
        return np.array(params[2])
    if box.name == 'met':
        return np.array([params[3], params[4]])
    if box.name == 'read':
        return np.array([params[5], params[6]])
    if box.name == 'who':
        return Spider(0, 3, Dim(2)).array

ob = {n: Dim(2), s: Dim(1)}
F = lambda params: Functor(ob, lambda box: p_mapping(box, params))
params0 = numpy.random.rand(6, 2)
assert F(params0)(sentences[2]).array != [0.]
```

We obtain a prediction by evaluating `F(params)` on a sentence and taking `sigmoid` to get a number between 0 and 1. We can then define the loss of a functor as the mean squared difference between its predictions and the true labels given by `T`. Of course, other activation and loss functions may be used.

Listing 2.8.5. Defining the loss function for a knowledge embedding task.

```
def sigmoid(x):
    sig = 1 / (1 + np.exp(-x))
    return sig

evaluate = lambda F, sentence: sigmoid(F(sentence).array)

def mean_squared(y_true, y_pred):
    return np.mean((np.array(y_true) - np.array(y_pred)) ** 2)

loss = lambda params: mean_squared(*zip(\
    *[(T(sentence).array, evaluate(F(params), sentence)) for sentence in sentences]))
```

The Jax package [Bra+18] supports automatic differentiation `grad` and just-in-time compilation `jit` for `jax` compatible `numpy` code. Since the code for `Tensor` is compatible, we can directly use Jax to compile a simple update function for the functor's parameters. We run the loop and report the results obtained.

Listing 2.8.6. Learning functors with Jax.

```
from jax import grad, jit
from time import time

step_size = 0.1

@jit
def update(params):
    return params - step_size * grad(loss)(params)

epochs, iterations = 7, 30
params = numpy.random.rand(6, 2)
for epoch in range(epochs):
    start = time()
    for i in range(iterations):
        params = update(params)

    print("Epoch {} ({:.3f} milliseconds)".format(epoch, 1e3 * (time() - start)))
    print("Testing loss: {:.5f}".format(loss(params)))

y_true = [T(sentence).array for sentence in sentences]
y_pred = [0 if evaluate(F(final_params), sentence) < 0.5 else 1
          for sentence in sentences]
print(classification_report(y_true, y_pred))
```

	precision	recall	f1-score	support
0.0	0.75	1.00	0.86	6
1.0	1.00	0.83	0.91	12
accuracy			0.89	18
macro avg	0.88	0.92	0.88	18
weighted avg	0.92	0.89	0.89	18

And voilà! In just a few lines of code we have learnt a compressed 2D tensor representation of the data of a 3D relational model. We have executed a simple statistical relational learning routine [Wan+17], which can in fact be seen as a version of the knowledge graph embedding model `Rescal` [NTK11], see 2.6, we review the theory of knowledge graph embeddings from a diagrammatic perspective, we show the advantages of moving into the complex numbers for embedding relations. In fact, we implemented and tested a similar embedding method using quantum computers [Mei+20; Lor+21] which naturally handle complex tensors, see 2.7. Large-scale diagram parsing is now possible thanks to Lambeq [Kar+21], as we showed above. There remains work to do on optimization and batching for tensor models, which would allow to scale these experiments to real-world corpora.

Chapter 3

Games for Pragmatics

The threefold distinction between syntax, semantics and pragmatics may be traced back to the semiotics of Peirce an his trilogy between sign, object and interpretant. According to Peirce [Pei65], these three aspects would induce three different approaches to semiotics, renewing the medieval *trivium*: formal grammar, logic and formal rhetoric. In his studies [Sta70], Stalnaker gives a finer demarcation between pragmatics and semantics through the concept of *context*: "It is a semantic problem to specify the rules for matching up sentences of a natural language with the propositions that they express. In most cases, however, the rules will not match sentences directly with propositions, but will match sentences with propositions relative to features of the context in which the sentence is used. Those contextual features are part of the subject matter of pragmatics."

In his *Philosophical Investigations* [Wit53], Wittgenstein introduces the concept of *language-game* (*Sprachspiel*) as a basis for his theory of meaning. He never gives a general definition, and instead proceeds by enumeration of examples: "asking, thanking, cursing, greeting, praying". Thus, depending on the language-game in which it is played, the same utterance "Water!" can be interpreted as the answer to a question, a request to a waiter or the chorus of a song. From the point of view of pragmatics, language-games provide a way of capturing the notion of *context*, isolating a particular meaning/use of language within an environment defined by the game.

Since Lewis' work on conventions [Lew69], formal *game theory* has been used to model the variability of the meaning of sentences and their dependence on context [Fra09; MP15; BS18]. These theoretical enquiries have also been supported by psycholinguistic experiments such as those of Frank and Goodman [FG12], where a Bayesian game-theoretic model is used to predict the behaviour of listeners and speakers in matching words with their referents.

In parallel to its use in pragmatics, game theory has been proven significant in designing machine learning tasks. It is at the heart of multi-agent reinforcement learning [TN05], where decision-makers interact in a stochastic environment. It is also used to improve the performance of neural network models, following the seminal work of Goodfellow et al. [Goo+14], and is beginning to be applied to natural language processing tasks such as dialogue generation [Li+16; Li+17], knowledge graph embedding [CW18; XHW18] and word-sense disambiguation [TN19].

The aim of this chapter is to develop formal and diagrammatic tools to model language games and NLP tasks. More precisely, we will employ the theory of *lenses*, which have been developed as a model for the dynamics of data-accessing programs [PGW17] and form the basis of the recent applications of category theory to both game theory [Gha+18; BHZ19] and machine learning [FST19; Cru+21]. The results are still at a preliminary stage, but the diagrammatic representation succeeds in capturing a wide range of pragmatic scenarios.

In Section 3.1, we argue that probabilistic models are best suited for analysing pragmatics and NLP tasks and show how the deterministic models studied in Chapter 2 may be turned into probabilistic ones. In Section 3.2, we introduce lenses and show how they capture the dynamics of probabilistic systems and stochastic environments. In Section 3.3, we show how parametrization may be used to capture agency in these environments and discuss the notions of optimum, best response and equilibrium for multi-agent systems. Throughout, we give examples illustrating how these concepts may be used in both pragmatics and NLP and in Section 3.4, we give a more in-depth analysis of three examples, building on recent proposals for modelling language games with category theory [HL18; de +21].

3.1 Probabilistic models

Understanding language requires resolving large amounts of vagueness and ambiguity as well as inferring interlocutor's intents and beliefs. Uncertainty is a central feature of pragmatic interactions which has led linguists to devise probabilistic models of language use, see [FJ16] for an overview.

Probabilistic modelling is also used throughout machine learning and NLP in particular. For example, the language modelling task is usually formulated as the task of learning the conditional probability $p(x_k \,|\, x_{k-1}, \ldots, x_1)$ to hear the word x_k given that words $x_1, \ldots, x_{k-1}$ have been heard. We have seen in Section 2.3 how neural networks induce probabilistic models using the `softmax` function. The language of probability theory is particularly suited to formulating NLP tasks, and reasoning about them at a high level of abstraction.

Categorical approaches to probability theory and Bayesian reasoning have made much progress in recent years. In their seminal paper [CJ19], Cho and Jacobs identified the essential features of probabilistic resoning using string diagrams, including marginalisation, conditionals, disintegration and Bayesian inversion. Some of their insights derived from Fong's thesis [Fon13], where Bayesian networks are formalised as functors into categories of Markov kernels. Building on Cho and Jacobs, Fritz [Fri20] introduced *Markov categories* as a synthetic framework for probability theory, generalising several results from the dominant measure-theoretic approach to probability into this high-level diagrammatic setting. These algebraic tools are powering interesting developments in applied category theory, including diagrammatic approaches to causal inference [JKZ18] and Bayesian game theory [BHZ19].

In this section, we review the basic notions of probability theory from a categorical perspective and show how they can be used to reason about discriminative and generative NLP models.

3.1.1 Categorical probability

We introduce the basic notions of categorical probability theory, following [CJ19] and [Fri20]. For simplicity, we work in the setting of *discrete* probabilities, although most of the results we use are formulated diagrammatically and are likely to generalise to any Markov category.

Let $\mathcal{D} : \mathbf{Set} \to \mathbf{Set}$ be the discrete distribution monad defined on objects by:

$$\mathcal{D}(X) = \{\, p : X \to \mathbb{R}^+ \,|\, d\text{has finite support and } \sum_{x \in X} p(x) = 1 \,\}$$

and on arrows $f : X \to Y$ by:

$$\mathcal{D}(f) : \mathcal{D}(X) \to \mathcal{D}(Y) : (p : X \to \mathbb{R}^+) \mapsto (y \in Y \mapsto \sum_{x \in f^{-1}(y)} p(x) \in \mathbb{R}^+)$$

We can construct the Kleisli category for the distribution monad $\mathbf{Prob} = \mathbf{Kl}(\mathcal{D})$ with objects sets and arrows discrete conditional distributions $f : X \to \mathcal{D}(Y)$ we

denote by $f(y|x) \in \mathbb{R}^+$ the probability $f(x)(y)$. Composition of $f : X \to \mathcal{D}(Y)$ and $g : Y \to \mathcal{D}(Z)$ is given by:

$$X \xrightarrow{f} \mathcal{D}(Y) \xrightarrow{\mathcal{D}(g)} \mathcal{D}\mathcal{D}(Z) \xrightarrow{\mu_Z} \mathcal{D}(Z)$$

where $\mu_Z : \mathcal{D}\mathcal{D}(Z) \to \mathcal{D}(Z)$ flattens a distribution of distributions by taking sums. Explicitly we have:

$$f \cdot g(z|x) = \sum_y g(z|y) f(y|x) \in \mathbb{R}^+$$

We may think of the objects of **Prob** as *random variables* and the arrows as *conditional distributions*.

The category **Prob** has interesting structure. First of all, it is a symmetric monoidal category with $\times$ as monoidal product. This is not a cartesian product since $\mathcal{D}(X \times Y) \neq \mathcal{D}(X) \times \mathcal{D}(Y)$. The unit of the monoidal product $\times$ is the singleton set 1 which is *terminal* in **Prob**, i.e. for any set X there is only one map $\mathtt{del}_X : X \to \mathcal{D}(1) \simeq 1$ called *discard*. Terminality of the monoidal unit means that if we discard the output of a morphism we might as well have discarded the input, a property often interpreted as *causality* [HV19].

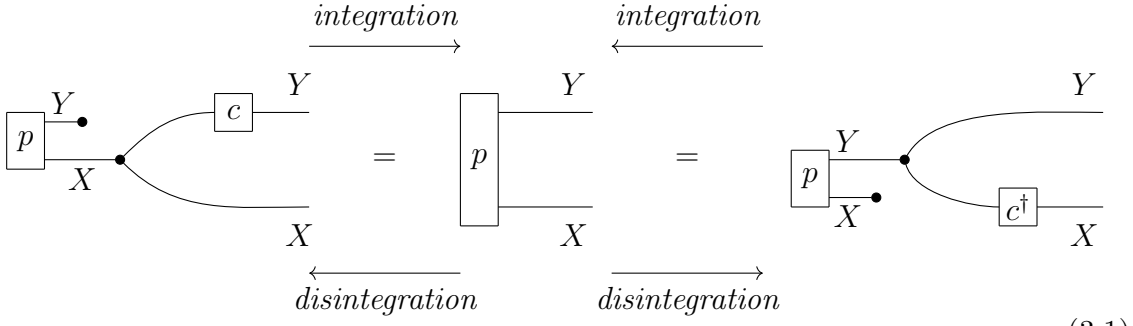
There is a commutative comonoid structure $\mathtt{copy}_X : X \to X \times X$ on each object X with counit $!_X$. Also there is a embedding of **Set** into **Prob** which gives the deterministic maps. These are characterized by the following property

$$\mathtt{copy}_Y \circ f = (f \times f) \circ \mathtt{copy}_X \iff f : X \to Y \text{is deterministic}$$

Morphisms $p : 1 \to X$ in **Prob** are simply distributions $p \in \mathcal{D}(X)$. Given a joint distribution $p : 1 \to X \times Y$ we can take *marginals* of p by composing with the discard map:

In **Prob** the disintegration theorem holds.

Proposition 3.1.1 (Disintegration)**.** *[CJ19] For any joint distribution $p \in \mathcal{D}(X \times Y)$, there are channels $c : X \to \mathcal{D}(Y)$ and $c^\dagger : Y \to \mathcal{D}(X)$ satisfying:*



(3.1)

Proof. The proof is given in [CJ19] in the case of the full subcategory of **Prob** with objects finite sets, i.e. for the category **Stoch** of stochastic matrices. Extending this proof to the infinite discrete case is simple because for any distribution $p \in \mathcal{D}(X \times Y)$ we may construct a stochastic vector over the support of p, which is finite by definition of $\mathcal{D}$. So we can disintegrate p in **Stoch** and then extend it to **Prob** by assigning probability 0 to elements outside of the support. □

Example 3.1.2. *Taking $X = \{1, 2\}$ and $Y = \{A, B\}$ an example of disintegration is the following:*

1	1/8
2	7/8

,

	A	B
1	1	0
2	3/7	4/7

$\leftarrow$

	A	B
1	1/8	0
2	3/8	1/2

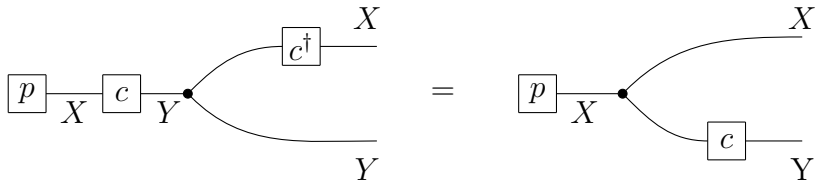
$\rightarrow$

	A	B
1	1/4	0
2	3/4	1

,

A	B
1/2	1/2

Given a *prior* $p \in \mathcal{D}(X)$ and a channel $c : X \to \mathcal{D}(Y)$, we can integrate them to get a joint distribution over X and Y and then disintegrate over Y to get the channel $c^\dagger : Y \to \mathcal{D}(X)$. This process is known as *Bayesian inversion* and $c^\dagger$ is called a Bayesian inverse of c along p. These satisfy the following equation, which can be derived from 3.1.



(3.2)

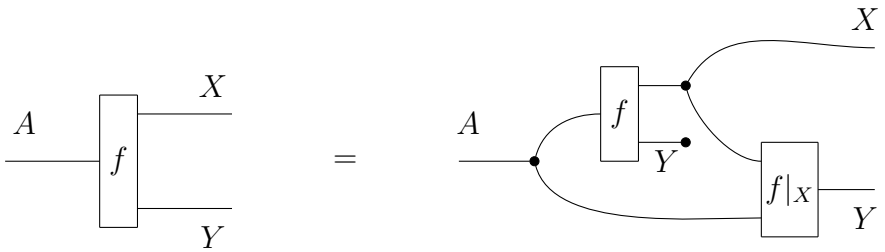
Interpreting this diagrammatic equation in **Prob** we get that:

$$c(y|x)p(x) = c^\dagger(x|y) \sum_x (c(y|x)p(x))$$

known as Bayes law.

The category **Prob** satisfies a slightly stronger notion of disintegration which doesn't only apply to states or joint distributions but to channels directly.

Proposition 3.1.3 (Conditionals)**.** *[Fri20] The category* **Prob** *has conditionals, i.e. for any morphism $f : A \to \mathcal{D}(X \times Y)$ in* **Prob** *there is a morphism $f|_X : A \times X \to \mathcal{D}(Y)$ such that:*



Proof. As for Proposition 3.1.1, this was proved in the case of **Stoch** by Fritz [Fri20] and it is simple to extend the proof to **Prob**. □

3.1.2 Discriminators

The first kind of probabilistic systems that we consider are *discriminators*. We define them in general as probabilistic channels that take sentences in a language $\mathcal{L}(G)$ and produce distributions over a set of features Y. These can be seen as solutions to the general classification task of assigning sentences in $\mathcal{L}(G)$ to classes in Y.

Definition 3.1.4 (Discriminator)**.** *A discriminator for a grammar G in a set of features Y is a probabilistic channel:*

$$c : \mathcal{L}(G) \to \mathcal{D}(Y)$$

Remark 3.1.5. *Throughout this chapter we assume that parsing for the chosen grammar G can be done efficiently. WFor simplicity, we assume that we are given a function:*

$$\texttt{parsing} : \mathcal{L}(G) \to \coprod_{u \in V^*} \mathbf{G}(u, s)$$

where $\mathbf{G}$ is the category of derivations for the grammar G. This could also be made a probabilistic channel with minor modifications to the results of this section.

In the previous chapters, we defined NLP models as functors $F : G \to \mathbf{S}$ where G is a grammar and $\mathbf{S}$ a semantic category. The aim for this section is to show that, in most instances, we can turn these models into probabilstic discriminators. We will do this in two steps. Assuming that parsing can be performed efficiently, it is easy to show that functorial models $F : G \to \mathbf{S}$ induce *encoders*, i.e. deterministic functions $\mathcal{L}(G) \to S$ which assign to every sentence in the language $\mathcal{L}(G) \subseteq V^*$ a compressed semantic representation in the sentence space $S = F(s)$. In order to build a discriminator c from an encoder $f : \mathcal{L}(G) \to S$ the only missing piece of structure is an *activation* function $\sigma : S \to \mathcal{D}(Y)$, mapping semantic states to distributions over classes.

Softmax is a useful activation function which allows to turn real valued vectors and tensors into probability distributions.

$$\texttt{softmax}_X : \mathbb{R}^X \to \mathcal{D}(X)$$

$$\texttt{softmax}(\vec{x})_i = \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}}$$

Thus, when the sentence space is $S = \mathbb{R}^Y$, `softmax` allows to turn encoders $f : X \to \mathbb{R}^Y$ into discriminators $c = \texttt{softmax} \circ f : X \to \mathcal{D}(Y)$. In fact all discriminators arise from encoders by post-composition with `softmax` as the following proposition shows.

Proposition 3.1.6. *Given a channel $c : X \to \mathcal{D}(Y)$ and a prior distribution $p \in \mathcal{D}(Y)$ there is a function $f : X \to \mathbb{R}^Y$ such that*

$$X \xrightarrow{f} \mathbb{R}^Y \xrightarrow{\texttt{softmax}} \mathcal{D}(Y) \;=\; X \xrightarrow{c} \mathcal{D}(Y)$$

Proof. In order to prove the existence of f, we construct the log likelihood function. Given $c : X \to \mathcal{D}(Y)$ there is a Bayesian inverse $c^\dagger : Y \to \mathcal{D}(X)$, then we can define the *likelihood* $l : X \times Y \to \mathbb{R}^+$ by:

$$l(x, y) = c^\dagger(x|y)p(y)$$

and the *log likelihood* is given by

$$f(x) = \texttt{log}_Y(l(x, y)) \in \mathbb{R}^Y$$

where $\texttt{log}_Y : (\mathbb{R}^+)^Y \to \mathbb{R}^Y$ is the logarithm applied to each entry. Then one can prove using Bayes law that:

$$\texttt{softmax}(f) = \texttt{softmax}(\texttt{log}(l(x, y))) = \frac{l(x, y)}{\sum_{y'} l(x, y')} = \frac{c^\dagger(x|y)p(y)}{\sum_{y'} c^\dagger(x|y')} = c(y|x) \quad (3.3)$$

□

Note that many functions f may induce the same channel c in this way. The encoder $f : X \to (\mathbb{R}^+)^Y$ given in the proof is the log of the likelihood function $l : X \times Y \to \mathbb{R}^+$. In these instances, the encoder is well behaved probabilistically, since it satisfies the version 3.3 of Bayes equation. However, in most gradient-based applications of stochastic methods, encoders $X \to (\mathbb{R})^Y$ such as those built from a neural network tend to achieve a better performance.

We already saw in Section 2.3, that `softmax` can be used to turn neural networks into probabilistic models. We rephrase this in the following definition.

Definition 3.1.7. *Given any recursive neural network $F : G \to \mathbf{NN}(W)$ for a monoidal grammar G with $F(s) = n$and $F(v) = 0$ for $v \in V \subseteq G_0$, and parameters $\theta : W \to \mathbb{R}$, we can build a discriminator $\tilde{F} : \mathcal{L}(G) \to \mathcal{D}([n])$ as the following composition:*

$$\tilde{F} = \mathcal{L}(G) \xrightarrow{\texttt{parsing}} \coprod_{u \in V^*} \mathbf{MC}(G)(u, s) \xrightarrow{F} \mathbf{NN}(W)(0, n) \xrightarrow{I_\theta} \mathbb{R}^n \xrightarrow{\texttt{softmax}} \mathcal{D}([n])$$

where $I_\theta : \mathbf{NN}(W) \to \mathbf{Set}_\mathbb{R}$ is the functor defined in2.3.

Softmax induces a *function* $\mathcal{S} : \mathbf{Mat}_\mathbb{R} \to \mathbf{Prob}$ defined on objects by $n \mapsto [n]$ for $n \in \mathbb{N}$ and on arrows by:

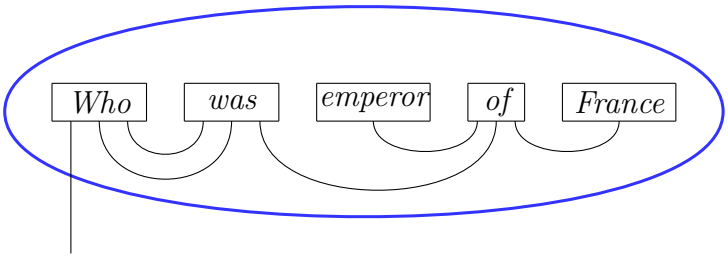
$$n \xrightarrow{M} m \quad \mapsto \quad [n] \xrightarrow{\mathcal{S}(M)} \mathcal{D}([m]) = [n] \xrightarrow{|-\rangle} \mathbb{R}^n \xrightarrow{M\cdot} \mathbb{R}^m \xrightarrow{\texttt{softmax}} \mathcal{D}([m])$$

Where $|-\rangle$ is the *one-hot encoding* which maps each element of $[n]$ to the corresponding basis vector in $\mathbb{R}^n$. Note that the mapping $\mathcal{S}$ is *not functorial*, but it is surjective as can readily be shown from Proposition 3.1.6.

Definition 3.1.8. *Given any tensor network model $F : G \to \mathbf{Mat}_\mathbb{R}$ for a rigid grammar G with $F(s) = n$ and $F(w) = 1$ for $w \in V \subseteq G_0$ we can build a discriminator $\tilde{F} : \mathcal{L}(G) \to \mathcal{D}([n])$ as the following composition:*

$$\tilde{F} = \mathcal{L}(G) \xrightarrow{\texttt{parsing}} \coprod_{u \in V^*} \mathbf{RC}(G)(u, s) \xrightarrow{F} \mathbf{Mat}_\mathbb{R}(1, n) \xrightarrow{\mathcal{S}} \mathcal{D}([n])$$

Example 3.1.9 (Student)**.** *Consider a student who is faced with a question in $\mathcal{L}(G, q)$, and has to guess the answer. This can be seen as a probabilistic channel $\mathcal{L}(G, q) \to \mathcal{D}(A)$ where A is a set of possible answers. We can build this channel using a tensor network model $F : G \to \mathbf{Mat}_{\mathbb{R}}$ with $F(q) = \mathbb{R}^A$ which induces a discriminator $\tilde{F} : \mathcal{L}(G, q) \to \mathcal{D}(A)$, as defined above. The image under $\tilde{F}$ of the grammatical question "Who was the emperor of France" in $\mathcal{L}(G, q)$ is given by the following diagram, representing a distribution in $\mathcal{D}(A)$:*



where the bubble represents the unary operator on hom-sets $\mathcal{S}$.

Definition 3.1.10. *Given any relational model $F : G \to \mathbf{Rel}$ for a rigid grammar G with $F(s) = Y$ and $F(w) = 1$ for $w \in V \subseteq G_0$ we can build a discriminator $\tilde{F} : \mathcal{L}(G) \to \mathcal{D}(Y)$ as follows: $\tilde{F} = \mathcal{L}(G) \xrightarrow{\texttt{parsing}} \coprod_{u \in V^*} \mathbf{RC}(G)(u, s) \xrightarrow{F} \mathbf{Rel}(1, Y) \simeq \mathcal{P}(Y) \xrightarrow{\texttt{uniform}} \mathcal{D}(Y)$ where* `uniform` *takes a subset of Y to the uniform distribution over that subset.*

Example 3.1.11 (Literal listener)**.** *Consider a listener who hears a word in W and needs to choose which object in R the word refers to. This example is based on Bergen et al. [BLG16]. Suppose we have a relation (called lexicon in [BLG16]) $\varphi : W \times R \to \mathbb{B}$, where $\varphi(w, r) = 1$ if w could refer to r and 0 otherwise. We can treat φ as a* likelihood *$\varphi : W \times R \to \mathbb{R}^+$. and model a* literal listener *as a classifier $l : W \to \mathcal{D}(R)$ defined by:*

$$l(w|r) \propto \varphi(w, r)$$

This is the same as taking $l(r) \in \mathcal{D}(W)$ to be the uniform distribution over the words that could refer to r according to φ. Thus the literal listener does not take the context into account. We can extend this model by allowing noun phrases to be uttered, instead of single words, i.e. $W = \mathcal{L}(G, n)$ for some rigid grammar G with noun type $n \in G_0$. Then we can replace the relation $\varphi : W \times R \to \mathbb{B}$ by a relational model *$F : G \to \mathbf{Rel}$ with $F(n) = R$, so that any grammatical noun phrase $g : u \to n$ in $\mathbf{RC}(G)$ is mapped to a subset $F(g) \subseteq R$ of the objects it could refer to. Then the listener is modeled by:*

$$l(g|r) \propto \langle F(g)|r\rangle$$

where $\langle F(g)|r\rangle = 1$ if $r \in F(g)$ and is zero otherwise. This corresponds to taking $l = \tilde{F}$ as defined in the proposition above. As we will see in Example 3.1.13 and Section 3.4.1, we can use a literal listener to define a pragmatic speaker which reasons about the literal interpretation of its words.

3.1.3 Generators

In order to solve NLP tasks, we need *generators* along-side discriminators. These are probabilistic channels which produce a sentence in the language given some semantic information in S. They are used throughout NLP, and most notably in neural language modelling [Ben+03] and machine translation [BCB14].

Definition 3.1.12 (Generator)**.** *A generator for S in G is a probabilistic channel:*

$$c : S \to \mathcal{DL}(G)$$

Bayesian probability theory allows to exploit the symmetry between encoders and decoders. Recall that Bayes law relates a conditional distribution $c : X \to \mathcal{D}(Y)$ with its Bayesian inverse $c^\dagger : Y \to \mathcal{D}(X)$ given a prior $p : 1 \to \mathcal{D}(X)$:

$$c^\dagger(x|y) = \frac{c(y|x)p(x)}{\sum_{x'} c(y|x')p(x')}$$

Thus Bayesian inversion $\dagger$ can be used to build a decoder $c^\dagger : Y \to \mathcal{DL}(G)$ from an encoder $c : \mathcal{L}(G) \to \mathcal{D}(Y)$ given a prior over the language $p \in \mathcal{DL}(G)$, and viceversa.

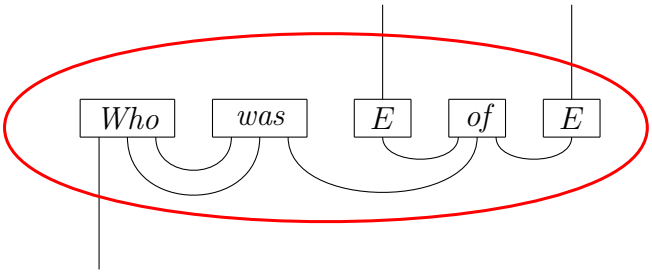
Example 3.1.13 (Speaker)**.** *Consider a speaker who is given an object in R (e.g. a black chess piece), and needs to produce a word in W to refer to it (e.g. "bishop"). We can model it as a generator $s : R \to \mathcal{D}(W)$. Following from Example 3.1.11, assume the speaker has access to a relation $\varphi : R \times W \to \mathbb{B}$. Then she knows what the literal interpretation of her words is, i.e. she can build a channel $l : W \to \mathcal{D}(R)$ corresponding to a literal speaker. In order to perform her task $s = R \to \mathcal{D}(W)$, she can take the Bayesian inverse of the literal listener $s = l^\dagger$. We will see in Section 3.4.1, that this strategy for the teacher yields a Nash equilibrium in a collaborative game with a listener.*

In practice, it is often very expensive to compute the Bayesian inverse of a channel. Expecially when one does not have a finite table of probabilities $X \times Y \to \mathbb{R}^+$ but rather a log-likelihood function $X \to \mathbb{R}^Y$ represented e.g. as a neural network. Thus, in most cases, we must resort to other tools to build generators. Recurrent neural networks are the simplest such tool. Indeed, given a recurrent network $g : n \to n \oplus |V|$ in **NN** we can build a generator $\mathbb{R}^n \to \mathcal{D}(V^*)$, by picking an initial encoder state $x \in \mathbb{R}^n$ and simply iterate the recurrent network by composing it along the encoder space n, see Section 2.3 for examples.

Example 3.1.14 (Translator/Chatbot)**.** *The sequence-to-sequence (Seq2Seq) model of Bahdanau et al. [BCB14], surveyed at the end of Section 2.3 is composed of recurrent encoder and decoder networks connected by an attention mechanism, see (2.16). It was originally used to build a translator $V^* \to \mathcal{D}(V'^*)$ for two vocabularies V and V'. Taking $V' = V$ we can use Seq2Seq to build a chatbot $V^* \to \mathcal{D}(V^*)$. We will use these in 3.3.15.*

From a categorical perspective, we do not understand generators as well as discriminators. If generators arise from functors, discriminators should arise from a dual notion, i.e. a concept of cofunctor, but we were unable to work out what these should be. Recently, Toumi and Koziell-Pipe [TK21] introduced a notion of functorial language model which allows to generate missing words in a grammatical sentence using DisCoCat models. Indeed, if we assume that the grammatical structure is given, then we may generate sentences with that structure using an activation function. We give an example in the case of relational models.

Example 3.1.15 (Teacher)**.** *Consider a teacher who knows an answer (e.g. "Napoleon"), and needs to produce a question with that answer (e.g. "Who was emperor of France?"). Suppose that the teacher has access to a relational model $F : G \to \mathbf{Rel}$ for a grammar G with a question type q and a noun type n with $F(q) = F(n)$ (i.e. answers are nouns). Following [Coe+18; TK21], we may repackage the functor F via an embedding $E : N \to F(n)$ in* **Rel** *where $N = \{\, w \in V \mid (w, n) \in G \,\}$ is the set of nouns, so that $F(w \to n) = E\,|w\rangle$. Fixing a grammatical structure $g : u \to q$, the teacher can generate questions with that structure, by evaluating the structure in her model F and then uniformly choosing which nouns to use in the question. This process is shown in the following diagram:*



where the bubble indicates the `uniform` *operator on hom-sets of* **Rel**. *Read from bottom to top, the diagram above represents a channel $N \to \mathcal{D}(N \times N)$ which induces a channel $N \to \mathcal{D}(\mathcal{L}(G, q))$. However, taking the* `uniform` *function is not a pragmatic choice for the teacher. Indeed, given "Napoleon" as input, the channel above is equally likely to choose the question "Who was emperor of France?" as "Who was citizen of France?". We will see in Section 3.4.2, that the pragmatics of posing questions may be captured by an adversarial scenario in which the teacher aims to ask hard questions to a student who tries to answer them.*

3.2 Bidirectional tools

The concept of *reward* is central in both game theory and machine learning. In the first, it is formalised in terms of a *utility function* and allows to define the optimal strategies and Nash equilibria of games. In the second, where it is captured (pessimistically) using a *loss function*, it defines an objective that the learning system must minimise. Reward is also a central concept in reinforcement learning where one considers probabilistic processes which run through a state-space while collecting rewards [How60; GS14]. In this section we show that the information flow of rewards in a dynamic probabilistic system can be captured in a suitable category of *lenses*. The bidirectionality of lenses allows to represent the action-reaction structure of game-theoretic and machine learning systems.

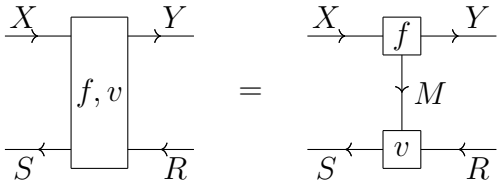
3.2.1 Lenses

Lenses are bidirectional data accessors which were introduced in the context of the view-update problem in database theory [BPV06; JRW12], although they have antecedents in Godel's "Dialectica interpretation". They are widely used in functional programming [PGW17] and have recently received a lot of attention in the applied category theory community, due to their applications to machine learning [FST19; FJ19] and game theory [Gha+18]. There are several variants on the definition of lenses available in the literature. These were largely unified by Riley who introduced *optics* as generalisation of lenses from **Set** to any symmetric monoidal category [Ril18a]. We use the term lens instead of optics for this general notion.

Definition 3.2.1 (Lens)**.** *[Ril18a] Let X, Y, O, S be objects in a symmetric monoidal category $\mathbf{C}$. A* lens *or* optic *$[f, v] : (X, S) \to (Y, O)$ in $\mathbf{C}$ is given by the following data:*

- *an object $M \in \mathbf{C}$,*
- *a forward part $f : X \to M \otimes Y$ called "get",*
- *a backward part $v : M \otimes O \to S$ called "put".*

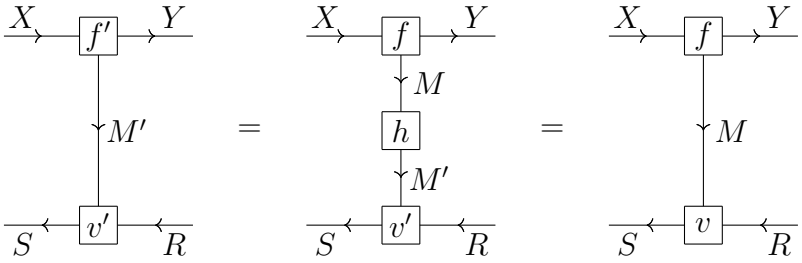
Stripped out of its set-theoretic semantics, a lens is simply seen as a pair of morphisms arranged as in the following two equivalent diagrams.



(3.4)

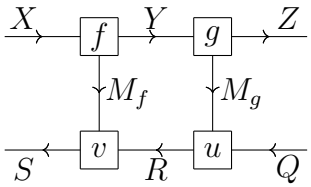
Two lenses $[f, v], [f', v'] : (X, S) \to (Y, R)$ are said to be *equivalent*, written $[f, v] \cong$

$[f', v']$, if there is a morphisms $h : M \to M'$ satisfying



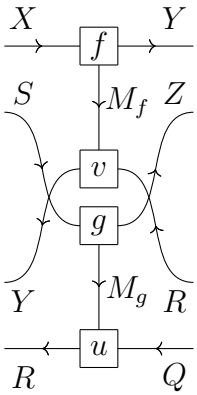
(3.5)

The quotient of the set of lenses by the equivalence relation $\cong$ can be expressed as a coend formula [Ril18b]. Here we omit the coend notation for simplicity. This quotient is needed in order to show that composition of lenses is associative and unital. Indeed, as shown by Riley [Ril18b], equivalence classes of lenses under $\cong$ form a symmetric monoidal category denoted $\mathbf{Lens}(\mathbf{C})$. Sequential composition for $[f, u] : (X, S) \to (Y, O)$ and $[g, v] : (Y, O) \to (Z, Q)$ is given by $[g, v] \circ [f, u] = [(\mathtt{id}_{M_f} \otimes g) \circ f, v \circ (\mathtt{id}_{M_f} \otimes u)]$ with $M_{g \circ f} = M_f \times M_g$, diagrammatically we have:



(3.6)

and tensor product $[f, v] \otimes [g, u]$ is given by:



(3.7)

Moreover, there are cups allowing to turn a covariant wire in the contravariant direction.

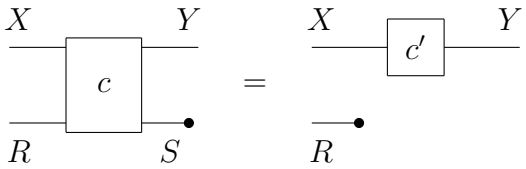
(3.8)

Remark 3.2.2. *This diagrammatic notation is formalised in [Hed17], where categories endowed with this structure are called* teleological. *Note that* $\mathbf{Lens}_{\mathbf{C}}$ *is not compact-closed, i.e. we can only turn wires from the covariant to the contravariant direction. When we draw a vertical wire as in 3.5, we actually mean a cup as in 3.8, this makes the notation more compact.*

We interpret lenses along the lines of compositional game theory [Gha+18]. A lens is a process which makes *observations* in X, produces *actions* or moves in Y, then it gets some *reward* or utility in R and gives back information as to its degree of *satisfaction* or coutility in S. Of course, this interpretation in no way exhausts the possible points of vue on lenses. For instance in [FST19] and [Cru+21], one interprets lenses as smooth functions turning inputs in X into outputs in Y, and then backpropagating the *error* in $R = \Delta Y$ to an error in the input $S = \Delta X$.

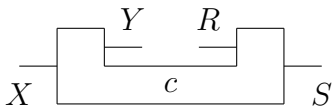
We are particularly interested in lenses over the category **Prob** of conditional probability distributions, which we call *stochastic lenses*. These have been characterised in the context of causal inference [JKZ18], where they are called *combs*, as morphisms of **Prob** satisfying a causality condition.

Definition 3.2.3 (Comb)**.** *[JKZ18] A comb is a stochastic map $c : X \otimes R \to Y \otimes S$ satisfying for some $c' : X \to Y$, the following equation:*



(3.9)

Combs have an intuitive diagrammatic representation from which they take their name.

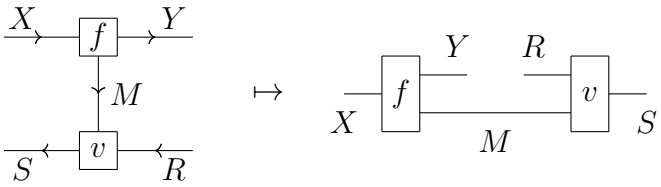


(3.10)

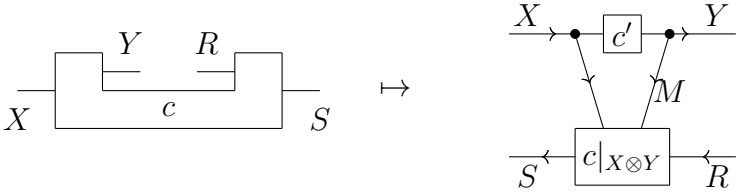
With this diagram in mind, the condition 3.9 reads: "the contribution from input R is only visible via output S "[JKZ18].

Proposition 3.2.4. *In* **Prob***, combs $X \otimes R \to Y \otimes S$ are in one-to-one correspondence with lenses $(X, S) \to (Y, R)$.*

Proof. The translation works as follows:



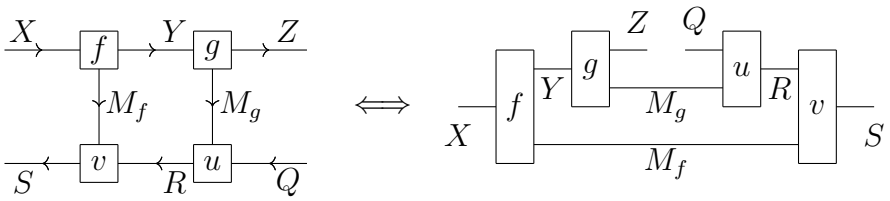
(3.11)



where c' is the morphism defined in 3.2.3 and $c|_{X\otimes Y}$ is the conditional defined by Proposition 3.1.3. Note that $c|_{X\otimes Y}$ is not unique in general. However any such choice

yields equivalent lenses since the category **Prob** is *productive*, see [DdR22, Theorem 7.2] for a slightly more general result. Indeed, this proposition is the base case for the inductive proof of [DdR22, Theorem 7.2]. □

Why should we use lenses $(X, S) \to (Y, R)$ instead of combs? The difference between them is in the composition. Indeed, composing lenses corresponds to nesting combs as in the following diagram:

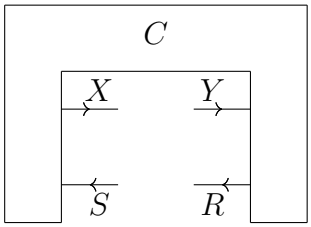


As we will see, the composition of lenses allows to define a notion of feedback for probabilistic systems which correctly captures their dynamics.

3.2.2 Utility functions

We analyse a special type of composition in **Lens**(**Prob**): between a lens and its environment, also called *context* in the open games literature [BHZ19]. This is a comb in **Lens**(**C**) that first produces an initial state in X, then receives a move in Y and produces a *utility* or reward in R.

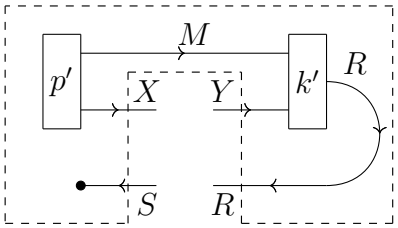
Definition 3.2.5 (Context)**.** *The context for lenses of type* $(X, S) \to (Y, R)$ *over* **C** *is a comb* c *in* **Lens**(**C**) *of the following shape:*



(3.12)

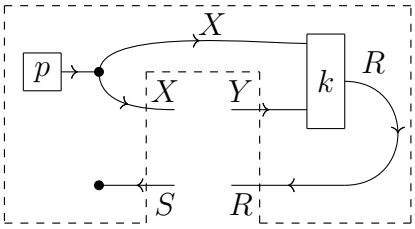
Proposition 3.2.6. *Contexts* C *for lenses of type* $(X, S) \to (Y, R)$ *in* **Prob** *are in one-to-one correspondence with pairs* $[p, k]$ *where* $p \in \mathcal{D}(X)$ *is a distribution over observations called* prior *and* $k : X \times Y \to \mathcal{D}(R)$ *is a channel called* utility function.

Proof. Since the unit of the tensor is terminal in **Prob**, a context C of the type above is given by a pair of morphisms, $[p', k']$ as in the following diagram:



(3.13)

Moreover, by the disintegration theorem, the joint distribution p' above may be factored into a prior distribution p over X and a channel $c : X \to \mathcal{D}(M)$. Therefore the context above is equivalent to to a context induced by a distribution over starting state $p \in \mathcal{D}(X)$ and a utility function $k : X \times Y \to R$ as in the following diagram:



(3.14)

□

As we will see in Section 3.3.2, this notion of utility captures the notion from game theory where utility functions assign a reward in R given the move of the player in Y and the moves of the other players, see also [BHZ19].

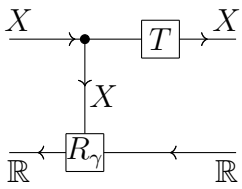
3.2.3 Markov rewards

Another interesting form of composition in **Lens**(**Prob**) arises when considering the notion of a Markov reward process (MRP) [How60]. MRPs are dynamic probabilistic systems which transit through a state space X while collecting rewards in $\mathbb{R}$.

Definition 3.2.7 (MRP)**.** *A Markov reward process with state space X is given by the following data:*

1. *a transition function $T : X \to \mathcal{D}(X)$,*
2. *a payoff function $R : X \to \mathbb{R}$,*
3. *and a discount factor $\gamma \in [0, 1)$.*

This data defines lens $[T, R_\gamma] : (X, \mathbb{R}) \to (X, \mathbb{R})$ in **Prob** drawn as follows:



(3.15)

where $R_\gamma : X \times \mathbb{R} \to \mathcal{D}(\mathbb{R})$ is given by the one-step discounted payoff:

$$R_\gamma(x, r) = R(x) + \gamma r$$

Intuitively, the MRP observes the state $x \in X$ which it is in and collects a reward $R(x)$, then uses the transition T to move to the next state. Given an expected future reward r, the MRP computes the current value given by summing the current reward with the expected future reward discounted by γ. This process is called *Markov*,

because the state at time step $t+1$ only depends on the state a time step t, i.e. x_{t+1} is sampled from the distribution $T(x_t)$.

Thus an MRP is an endomorphism $[T, R_\gamma] : (X, \mathbb{R}) \to (X, \mathbb{R})$ in the category of stochastic lenses. We can compose $[T, R_\gamma]$ with itself n times to get a new lens $[T, R_\gamma]^n$ where the forward part is given by iteration of the transition function and the backward part is given by the n-step discounted payoff:

$$R_\gamma^{n-}(x, r) = \sum_{i=1}^{n-1} \gamma^i R(T^i(x)) + \gamma^n r$$

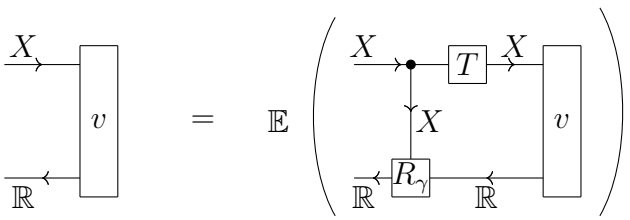
Since $0 \leq \gamma < 1$, this expression converges in the limit as $n \to \infty$ if R and T are deterministic. When R and T are stochastic, one can show that the expectation $\mathbb{E}(R_\gamma^{n-})$ converges to the *value* of the MRP which yields a measure of the total reward expected from this process.

$$\texttt{value}([T, R_\gamma])(x) = \mathbb{E}(\sum_{i=1}^{\infty} \gamma^i R(T^i(x))) = \lim_{n\to\infty} \mathbb{E}(\sum_{i=1}^{n} \gamma^i R(T^i(x))) \tag{3.16}$$

We can represent this as an effect $v = \texttt{value}([T, R_\gamma]) : (X, \mathbb{R}) \to (1, 1)$ in $\mathbf{Lens}(\mathbf{Prob})$, which simply consists in a probabilistic channel $v : X \to \mathbb{R}$. This effect v satifies the following Bellman fixed point equation in $\mathbf{Lens}(\mathbf{Prob})$, characterising it as the iteration of $[T, R_\gamma]$, see [GS14].

$$v(x) = \mathbb{E}(R(x) + \gamma v(T(x)))$$

which we may express diagrammatically as:



(3.17)

where $\mathbb{E}$ denotes the conditional expectation operator, given for any channel $f : X \to \mathcal{D}(\mathbb{R})$ by the function $\mathbb{E}(f) : X \to \mathbb{R}$ defined by $\mathbb{E}(f)(x) = \sum_{y\in\mathbb{R}} yf(y|x)$ (note that there only finitely many non-zero terms in this discrete setting). The value v of the reward process is often estimated by running Monte Carlo methods which iterate the transition function while collecting rewards. Note that *not* all stochastic lenses $(X, \mathbb{R}) \to (X, \mathbb{R})$ have an effect $(X, \mathbb{R}) \to (1, 1)$ with which they satisfy Equation 3.17. In fact it is sufficient to set $T = \texttt{id}_X$, $R(x) = 1$ for all $x \in X$ and $\gamma > 1$ in order to get a counter-example. It would be interesting to characterise the stochastic lenses satisfying 3.17 algebraically, e.g. are they closed under composition? This would in fact be true if the conditional expectation operator $\mathbb{E}$ was functorial. Given the results of [Cha+09] and [AR18], we expect that conditional expectation can be made a functor by restriciting the objects of **Prob**.

3.3 Cybernetics

Parametrization is the process of representing functions $X \to Y$ via a parameter space Π with a map $\Pi \to (X \to Y)$. It is a fundamental tool in both machine learning and game theory, since it allows to define a notion of *agency*, through the choice of parameters. For example, players in a formal game are parametrized over a set of *strategies*: there is a function $X \to Y$, turning observations into moves, for any strategy in Π. In reinforcement learning, the agent is parametrized by a set of *policies*, describing how to turn states into actions. We show that parametrized lenses are suitable for representing these systems and give examples relevant for NLP.

3.3.1 Parametrization

Categories allow to distinguish between two types of parametrization. Let $\mathbf{S}$ be a semantic category with a forgetful functor $\mathbf{S} \hookrightarrow \mathbf{Set}$

Definition 3.3.1 (External parametrization)**.** *An* external *parametrization* (f, Π) *of morphisms* $X \to Y$ *in a category* $\mathbf{S}$*, also called a* family *of morphisms indexed by* Π*, is a function* $f : \Pi \to \mathbf{S}(X, Y)$*. These form a category* $\mathbf{Fam}(\mathbf{S})$ *with composition defined for* $f : \Pi_0 \to \mathbf{S}(X, Y)$ *and* $g : \Pi_1 \to \mathbf{S}(Y, Z)$ *by* $g \circ f : \Pi_0 \times \Pi_1 \to \mathbf{S}(X, Z)$ *with* $g \circ f(\pi_0, \pi_1) = f(\pi_0) \circ g(\pi_1)$*.*

Definition 3.3.2 (Internal parametrization)**.** *An* internal *parametrization* (f, Π) *of morphisms* $X \to Y$ *in a monoidal category* $\mathbf{S}$ *is a morphism* $f : \Pi \otimes X \to Y$ *in* $\mathbf{S}$*. These form a category* $\mathbf{Para}(\mathbf{S})$ *with composition defined for* $f : \Pi_0 \otimes X \to Y$ *and* $g : \Pi_1 \otimes Y \to Z$ *by* $g \circ f : \Pi_0 \otimes \Pi_1 \otimes X \to Z$ *with* $g \circ f = f(\pi_0) \circ g(\pi_1)$*.*

Which parametrization should we prefer, **Fam** or **Para**? It depends on context. Internal parametrization is usually a stricter notion, because it imposes that the parametrization be a morphism of $\mathbf{S}$ and not simply a function. For example, $\mathbf{Para}(\mathbf{Smooth})$ embeds in $\mathbf{Fam}(\mathbf{Smooth})$ but the embedding is not full, i.e. there are external parametrizations defined by non-differentiable functions. In fact, the **Para** construction was introduced in the context of gradient-based learning [Cru+21], where it is very desirable that the parametrization be differentiable. External parametrizations are mostly used in the compositional game theory literature [Gha+18], since they are more flexible and allow to define a notion of *best response* (see 3.3.6). However they are also inextricably linked to **Set**, making them less desirable from a categorical perspective, since it is harder to prove results about $\mathbf{Fam}(\mathbf{S})$ given knowledge of $\mathbf{S}$.

Note that the two notions coincide for the category of sets and functions. Indeed, since **Set** is cartesian closed, we have that:

$$\Pi \to \mathbf{Set}(X, Y) \iff \Pi \to (X \to Y) \iff \Pi \times X \to Y$$

and therefore $\mathbf{Para}(\mathbf{Set}) \simeq \mathbf{Fam}(\mathbf{Set})$. Even though **Prob** is not cartesian closed, these notions again coincide.

Proposition 3.3.3. *Internal and external parametrizations coincide in* **Prob**, *i.e.* $\mathbf{Fam}(\mathbf{Prob}) \simeq \mathbf{Para}(\mathbf{Prob})$.

Proof. This follows by the following derivation in **Set**.

$$\Pi \to \mathbf{Prob}(X, Y) \iff \Pi \to (X \to \mathcal{D}(Y)) \iff \Pi \times X \to \mathcal{D}(Y)$$

□

In fact, the derivation above holds in any Kleisli category for a commutative strong monad.

Proposition 3.3.4. *For any commutative strong monad* $M : \mathbf{Set} \to \mathbf{Set}$, *the Kleisli category* $\mathbf{Kl}(M)$ *is monoidal and we have:*

$$\mathbf{Fam}(\mathbf{Kl}(M)) \simeq \mathbf{Para}(\mathbf{Kl}(M))$$

i.e. the notions of internal and external parametrization for Kleisli categories coincide.

3.3.2 Open games

We now review the theory of open games [Gha+18; BHZ19]. Starting from the definition of lenses, open games are obtained in two steps. First an open game is a family of lenses parametrized by a set of *strategies*: for each strategy the forward part of the lens says how the agent turns observations into moves and the backward part says how it computes a payoff given the outcome of its actions. Second, this family of lenses is equipped with a *best response* function indicating the optimal strategies for the agent in a given context.

Definition 3.3.5 (Family of lenses)**.** *A family of lenses over a symmetric monoidal category* **C** *is a morphism in* $\mathbf{Fam}(\mathbf{Lens}(\mathbf{C}))$. *Explicitly, a family of lenses* $P : (X, S) \to (Y, R)$ *is given by a function* $P : \Pi \to \mathbf{Lens}(\mathbf{C})((X, S), (Y, R))$ *for some set of parameters* Π.

Recall the definition of *context* for lenses given in 3.2.5. Let us use the notation $\mathbb{C}((X, S), (Y, R))$ for the set of contexts of lenses $(X, S) \to (Y, R)$.

Definition 3.3.6 (Best response)**.** *A best response function for a family of lenses* $P : (X, S) \to (Y, R)$ *is a function of the following type:*

$$B : \mathbb{C}((X, S), (Y, R)) \to \mathcal{P}(P)$$

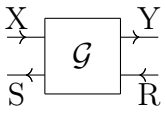
Thus B takes as input a context and outputs a predicate on the set of parameters (or strategies).

Definition 3.3.7 (Open game)**.** *[BHZ19] An open game* $\mathcal{G} : (X, R) \to (Y, O)$ *over a symmetric monoidal category* **C** *is a family of lenses* $\{ [\pi, v_\pi] \}_{\pi \in P_\mathcal{G}}$ *in* **C** *equipped with a best response function* $B_\mathcal{G}$.

Proposition 3.3.8. *[BHZ19] Open games over* **C** *form a symmetric monoidal category denoted* **Game**(**C**).

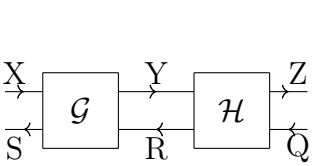
Proof. See [BHZ19] for details on the proof and the composition of best responses in this general case. □

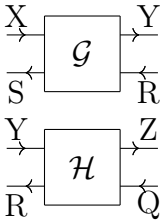
The category **Game**(**C**) admits a graphical calculus developed in [Hed17] which is the same as the one for lenses described in Section 3.2.1.. Each morphism is represented as a box with covariant wires for observations in X and moves in Y, and contravariant wires for rewards in R (called utilities in [Gha+18]) and satisfaction in S (called coutilities in [Gha+18]).



(3.18)

These boxes can be composed in parallel, indicating that the players make a move simultaneously, or in sequence, indicating that the second player can observe the first player's move.





(3.19)

Of particular interest to us is the category of open games over **Prob**. Since we focus on this category for the rest of the chapter we will use the notation **Game** := **Game**(**Prob**). In this category the best response function is easier to specify, since contexts factor as in Proposition 3.2.6.

Proposition 3.3.9. *A context for an open game in* **Game**(**Prob**) $(X, S) \to (Y, R)$ *is given by a stochastic lens* $[p, k] : (1, X) \to (Y, R)$ *or explicitly by a pair of channels* $p : 1 \to \mathcal{D}(M \times X)$ *and* $k : M \times Y \to \mathcal{D}(R)$.

Proof. This follows from Proposition 3.2.6. □

With this simpler notation for contexts as pairs $[p, k]$ we can define the composition of best responses for open games.

Definition 3.3.10. *[BHZ19, Definition 3.18] Given open games* $(X, S) \xrightarrow{\mathcal{G}} (Y, R) \xrightarrow{\mathcal{H}} (Z, O)$ *the composition of their best responses is given by the cartesian product:*

$$B_{\mathcal{G}\cdot\mathcal{H}}([p, k]) = B_{\mathcal{G}}([p, \mathcal{H} \cdot k]) \times B_{\mathcal{H}}([p \cdot \mathcal{G}, k]) \subseteq \Pi_{\mathcal{G}} \times \Pi_{\mathcal{H}}$$

where $\Pi_{\mathcal{G}}$ *and* $\Pi_{\mathcal{H}}$ *are the corresponding sets of strategies.*

We can now define a notion of utility-maximising agent

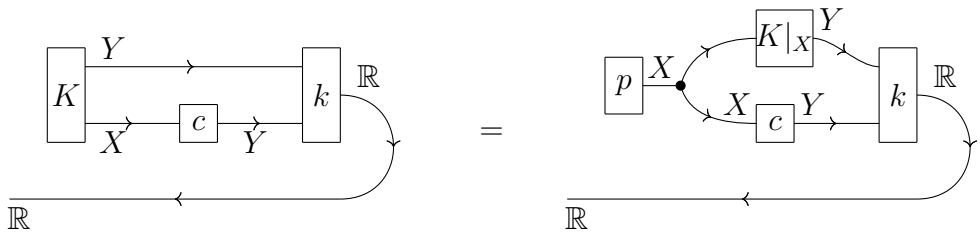
Definition 3.3.11 (Utility-maximising agent)**.** *An (expected) utility maximising agent is a morphism* $c : (X, 1) \to (Y, \mathbb{R})$ *in* **Game** *given by the following data:*

1. $\pi \in \Pi = X \to \mathcal{D}(Y)$ *is the set of strategies.*
2. $f^+_\pi : X \to \mathcal{D}(A) : x \mapsto \pi(x)$
3. $f^-_\pi : \mathbb{R} \to 1$ *is the discard map.*
4. $B : ([p : 1 \to M \times X,\, k : M \times Y \to \mathbb{R}]) = \mathtt{argmax}_{\pi \in P}(\mathbb{E}(p; \mathtt{id}_M \otimes f_\pi; k))$

where $\mathtt{argmax} : \mathbb{R}^\Pi \to \mathcal{P}(\Pi)$.

Note that it is sufficient to specify the input type X and output type Y in order to define a utility-maximising agent $(X, 1) \to (Y, \mathbb{R})$ in **Game**.

Example 3.3.12 (Classifier utility)**.** *We model a classifier $X \to \mathcal{D}(Y)$ as a utility-maximising agent $c : (X, 1) \to (Y, \mathbb{R})$. Assume that Y has a metric $d : Y \times Y \to \mathbb{R}$. Given a dataset of pairs, i.e. a distribution $K \in \mathcal{D}(X \times Y)$, we can define a context $[K, k]$ for c by setting $k = -d$, as shown in the following pair of equivalent diagrams:*

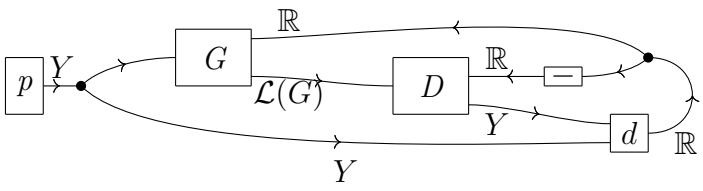


where $(p, K|_X)$ is the disintegration of K along X. This yields a distribution over the real numbers $\mathbb{R}$ corresponding to the utility that classifier c gets in its approximation of $K|_X$. The best response function for c in this context is given by:

$$B([K, k]) = \mathtt{argmax}_c \mathbb{E}(-K; \mathtt{id}_Y \otimes c; d)$$

Maximising this expectation corresponds to minimising the average distance between the true label $y \in Y$ and the predicted label $c(x)$ for (x, y) distributed as in the dataset $K \in \mathcal{D}(X \times Y)$.

Example 3.3.13 (Generator/Discriminator)**.** *Fix a grammar G, a feature set Y with a distance function $d : Y \times Y \to \mathbb{R}$ and a disribution $p \in \mathcal{D}(Y)$. We model a generator $G : Y \to \mathcal{DL}(G)$ and a discriminator $D : \mathcal{L}(G) \to \mathcal{D}(Y)$ as utility-maximising agents and compose them as in the following diagram:*



Where $- : \mathbb{R} \to \mathbb{R}$ is multiplication by -1. This yields an adversarial (or zero-sum) game between the generator and the discriminator. Assuming G and D are utility-maximising agents, the best response function for this closed game is given by:

$$\mathtt{argmax}_G(\mathbb{E}_{y \in p}(d(y, D(G(y))))) \times \mathtt{argmax}_D(-\mathbb{E}_{y \in p}(d(y, D(G(y))))))$$

where we use the notation $\mathbb{E}_{y \in p}$ to indicate the expectation (over the real numbers) where y is distributed according to $p \in \mathcal{D}(Y)$. Thus we can see that the game reaches an equilibrium when the discriminator D can always invert the generator G, and the generator cannot choose any strategy to change this. Implementing G and D as neural networks yields the generative adversarial architecture of Goodfellow et al. [Goo+14] which is used to solve a wide range of NLP tasks, see [WSL19] for a survey. In Section 3.4.2, we will see how this architecture can be used for question answering.

In the remainder of this chapter, we will work interchangeably in the category of families of stochastic lenses **Fam**(**Lens**(**Prob**)) or in **Game**(**Prob**). The only difference between these categories is that the latter has morphisms equipped with a best response function and a predefined way of composing them. As we will see, this definition of best response is not suited to formalising repeated games and the category of families of stochastic lenses gives a more flexible approach.

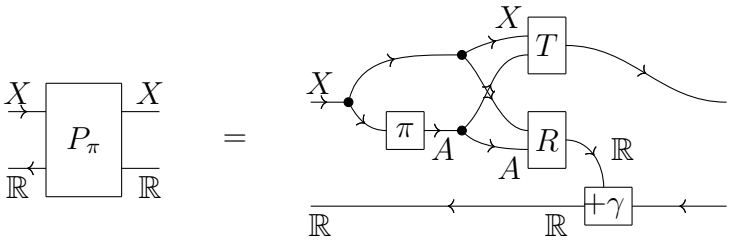
3.3.3 Markov decisions

We now study Markov decision processes (MDP) [How60] as parametrized stochastic lenses. These are central in reinforcement learning, they model a situation in which a single agent makes decisions in a stochastic environment with the aim of maximising its expected long-term reward. An MDP is simply an MRP 3.2.7 parametrised by a set of actions.

Definition 3.3.14 (MDP)**.** *A Markov decision process with states in X and actions in A is given by the following data:*

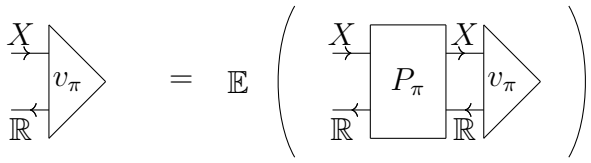
1. *a transition $T : A \times X \to \mathcal{D}(X)$*
2. *a reward function $R : A \times X \to \mathcal{D}(\mathbb{R})$*
3. *a discount factor $\gamma \in [0, 1)$*

A *policy* is a function $\pi : X \to \mathcal{D}(Y)$ which represents a strategy for the agent: it yields a distribution over the actions $\pi(x) \in \mathcal{D}(Y)$ for each state $x \in X$. Given a policy $\pi \in \Pi = X \to \mathcal{D}(Y)$, the MDP induces an MRP which runs through the state space X by choosing actions according to π and collecting rewards. Thus, we can formalise an MDP as a family of MRPs parametrized by the policies Π, i.e. a morphism $P_\pi : (X, \mathbb{R}) \to (X, \mathbb{R})$ in **Fam**(**Lens**(**Prob**)), given by the following composition:



(3.20)

The aim of the agent is to maximise its expected discounted reward. Given a policy π, the MRP P_π induces a value function $v_\pi = \texttt{value}(P_\pi) : X \to \mathbb{R}$, as defined in Section 3.2. Again, we have that the *Bellman expectation equation* holds:



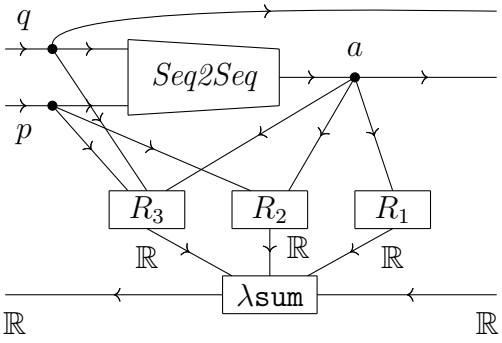
(3.21)

This fixed-point equation gives a way of approximating the value function by iterating the transitions and rewards under a given policy. The aim is then to find the *optimal policy* π^* at starting state x_0 by maximising the corresponding value function v_π:

$$\pi^* = \texttt{argmax}_{\pi \in P}(v_\pi(x_0)).$$

Note that the this optimal policy is not captured by the open games formalism. If we took the MDP P_π to be a morphism in **Game**, i.e. if we equipped it with a best response B as defined in 3.3.6, then the sequential composition $P_\pi \cdot P_\pi$ would have a best response function with target $\Pi \times \Pi$, i.e. the strategy for an MDP with two stages would be a pair of policies for the first and second stage, contradicting the idea that the MDP tries to find the best policy to use at all stages of the game. Working in the more general setting of parametrized stochastic lenses we can still reason about optimal strategies, although it would be interesting to have a compositional characterization of these.

Example 3.3.15 (Dialogue agents)**.** *In [Li+16], Li et al. use reinforcement learning to model dialogue agents. For this they define a Markov decision process with the following components. Fix a set of words V. The states are given by pairs (p, q) where $q \in V^*$ is the current sentence uttered (to which the agent needs to reply) and $p \in V^*$ is the previous sentence uttered by the agent. An action is given by a sequences of words $a \in V^*$. The policy is modeled by a Seq2Seq model with attention based on Bahdanau et al. [BCB14], which they pretrain on a dataset of dialogues. Rewards are computed using three reward functions R_1, R_2 and R_3, where R_1 penalises dull responses a (by comparing them with a manually constructed list), R_2 penalizes semantic similarity between consecutive statements of the agent p and a, and R_3 rewards semantic coherence between p, q and a. These three rewards are combined using a weighted sum $\lambda\texttt{sum}(r_1, r_2, r_3) = \lambda_1 r_1 + \lambda_2 r_2 + \lambda_3 r_3$. We can depict the full architecture as one diagram:*



3.3.4 Repeated games

The definition of MDP given above captures a *single* agent interacting in a fixed environment with no agency. In real-world situations however, the environment consists itself in a collection of agents which also make choices so as to maximise their expected reward.

In order to allow many agents interacting in an environment, we break up the definition of MDPs above, and isolate the part that is making decisions from the *environment* part. We pack the transition T, reward R and discount factor γ into one lens called environment and given by:

$$E = [T, R_\gamma] : (X \times A, \mathbb{R}) \to (X, \mathbb{R})$$

where $R_\gamma : X \times A \times \mathbb{R} \to \mathbb{R}$ is defined by

$$R_\gamma(x, a, r) = R(x, a) + \gamma r$$

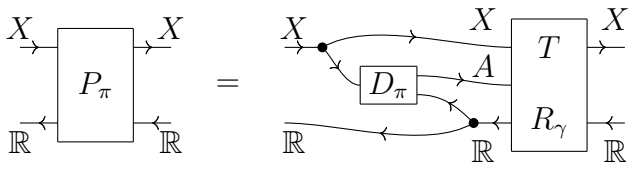
This allows to isolate the decision part, seen as an open game of the following type:

$$D : (X, 1) \to (A, \mathbb{R})$$

which represents an agent turning states in X into actions in A and receiving rewards in $\mathbb{R}$. Explicitly we have the following definition.

Definition 3.3.16 (Decision process)**.** *A decision process D with state space X, action space A and discount factor γ is a utility maximising agent $D : (X, 1) \to (A, \mathbb{R})$*

Composing D with the environment E as in the following diagram, we get back precisely the definition of MDP.



(3.22)

We can now consider a situation in which many decision processes interact in an environment.

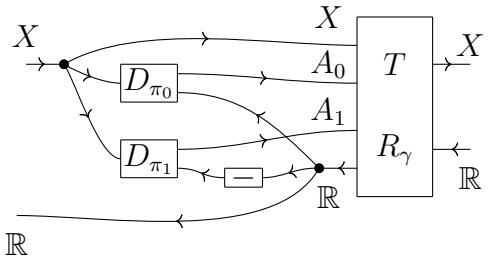
Stochastic games, a.k.a Markov games, were introduced by Shapley in the 1950s [Sha53]. They can be seen as a generalisation of MDPs, even though they appeared before the work of Bellman [Bel57].

Definition 3.3.17 (Stochastic game)**.** *A stochastic game is given by the following data:*

1. *A number k of players, a set of states X and a discount factor γ,*
2. *a set of actions A_i for each player $i \in \{0, \dots, k-1\}$,*
3. *a transition function $T : X \times A_0 \times \cdots \times A_k \to \mathcal{D}(X)$,*

4. a reward function $R : X \times A_0 \times \cdots \times A_k \to \mathbb{R}^k$.

The game is played in stages. At each stage, every player observes the current state in X and chooses an action in A_i, then "Nature" changes the state of the game using transition T and every player is rewarded according to R. It is easy express this repeated game as a parallel composition of decision processes D_i followed by the environment $[T, R_\gamma]$ in **Game**. As an example, we give the diagram for a two-player zero-sum stochastic game.

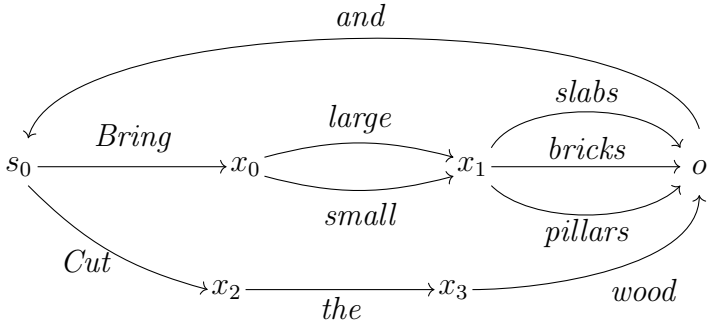


(3.23)

In this case, one can see directly from the diagram, that the zero-sum stochastic game induces an MDP with action set $\Pi_0 \times \Pi_1$ given by pairs of policies from players 0 and 1. A consequence of this is that one can prove the Shapley equations [Ren19], the analogue of the Bellman equation of MDPs, for determining the equilibria of the game.

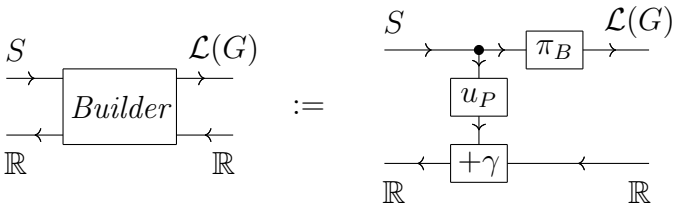
Note that stochastic games only allow players to make moves in parallel, i.e. at every stage the move of player i is independent of the other players' moves. The generality provided by this compositional approach allows to consider repeated stochastic games in which moves are made in sequence, as in the following example.

Example 3.3.18 (Wittgenstein's builders)**.** *[Wit53] Consider a builder and an apprentice at work on the building site. The builder gives orders to his apprentice, who needs to implement them by acting on the state of the building site. In order to model this language game we start by fixing a grammar G for orders, e.g. the regular grammar defined by the following labelled graph:*

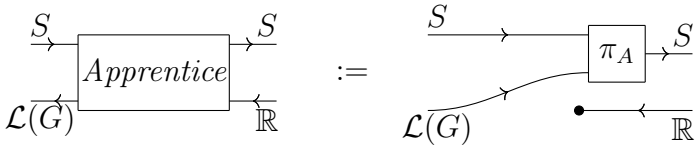


This defines a language for orders $\mathcal{L}(G)$ *given by the paths* $s_0 \to o$ *in the graph above. The builder observes the state of the building site* S *and produces an order in* $\mathcal{L}(G)$*, we can model it as a channel* $\pi_B : S \to \mathcal{D}\mathcal{L}(G)$*. We assume that the builder has a project* P*, i.e. a subset of the possible configurations of the building site that he finds satisfactory* $P \subseteq S$*. We can model these preferences with a utility function*

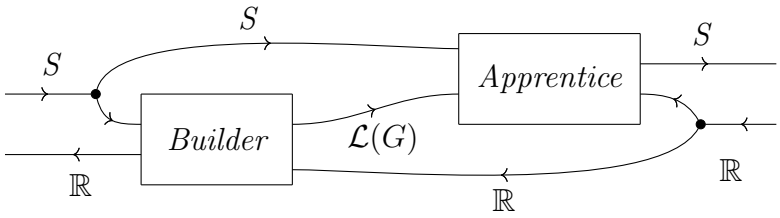
$u_P : S \to \mathbb{R}$, which computes the distance between the current state of the building site and the desired states in P. The builder wants to get the project done as soon as possible and he discounts the expected future rewards by a discount factor $\gamma \in [0, 1)$. This data defines a stochastic lens given by the following composition:



The apprentice receives an order in $\mathcal{L}(G)$ and uses it to modify the state of the building site S. We can model this as a probabilistic channel $\pi_A : \mathcal{L}(G) \times S \to \mathcal{D}(S)$. This data defines the following stochastic lens:



We assume that the apprentice is satisfied when the builder is, i.e. the utilities of the builder are fed directly into the apprentice. This defines a repeated game, with strategy profiles given by pairs (π_B, π_A), given by the following diagram:



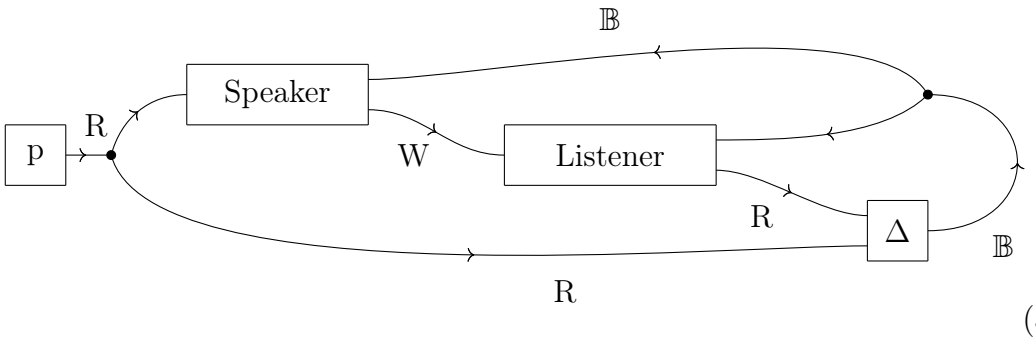
This example can be extended and elaborated in different directions. For example, the policy for the apprentice π_A could be modeled as a functor $F : G \to \mathbf{Set}$ with $F(s_0) = 1$ and $F(o) = S \to \mathcal{D}(S)$ so that grammatical orders in $\mathcal{L}(G)$ are mapped to channels that change the configuration of the building site. Another aspect is the builder's strategy $\pi_B : S \to \mathcal{L}(G, o)$ which could be modeled as a generation process from a probabilistic context-free grammar. Also the choice of state space S is interesting, it could be a discrete minecraft-like space, or a continuous space in which words like "cut" have a more precise meaning.

3.4 Examples

3.4.1 Bayesian pragmatics

In this section we study the *Rational Speech Acts* model of pragmatic reasoning [Fra09; FG12; GS13; BLG16]. The idea, based on Grice's conversational implicatures [Gri67], is to model speaker and listener as rational agents who choose words attempting to be informative in context. Implementing this idea involves the interaction of game theory and Bayesian inference. While this model has been criticised on the ground of attributing excessive rationality to human speakers [Gat+13], it has received support by psycholinguistic experiments on children and adults [FG12] and has been applied successfully to a referential expression generation task [MP15].

Consider a collaborative game between speaker and listener. There are some objects or *referents* in R lying on the table. The *speaker* utters a word in W referring to one of the objects. The *listener* has to guess which object the word refers to. We define this reference game by the following diagram in **Game**(**Prob**).



(3.24)

Where p is a given prior over the referents (encoding the probability that an object $r \in R$ would be referred to) and $\Delta(r, r') = 1$ if $r = r'$ and $\Delta(r, r') = 0$ otherwise. The strategies for the speaker and listener are given by:

$$P_0 = R \to \mathcal{D}(W) \qquad P_1 = W \to \mathcal{D}(R)$$

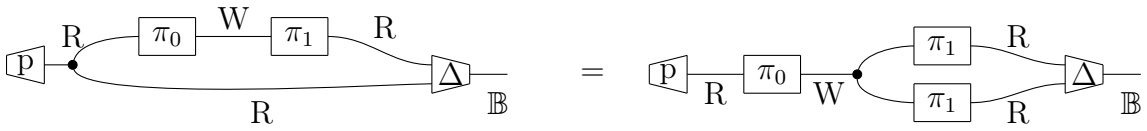
The speaker is modeled by a utility-maximising agent with strategies $\pi_0 : R \to \mathcal{D}(W)$ and best response in context $[p \in \mathcal{D}(R \times W), l : R \times W \to \mathbb{B}]$ given by the $\pi : R \to \mathcal{D}(W)$ in the argmax of $\mathbb{E}(l \circ (\pi \otimes \mathtt{id}_R) \circ p)$. Similarly for the listener with the roles of R and W interchanged. Composing speaker and listener according to (3.24) we obtain a closed game for which the best response is a predicate over the strategy profiles $(\pi_0 : R \to \mathcal{D}(W), \pi_1 : W \to \mathcal{D}(R))$ indicating the subset of Nash equilibria.

$$\mathtt{argmax}_{\pi_0,\pi_1}(\mathbb{E}(\Delta \circ ((\pi_1 \circ \pi_0) \otimes \mathtt{id}_R) \circ \mathtt{copy} \circ p) \tag{3.25}$$

If we assume that the listener uses Bayesian inference to recover the speaker's intended referent, then we are in a Nash equilibrium.

Proposition 3.4.1. *If $\pi_1 : W \to \mathcal{D}(R)$ is a Bayesian inverse of $\pi_0 : R \to \mathcal{D}(W)$ along $p \in \mathcal{D}(R)$ then (π_0, π_1) is a Nash equilibrium for (3.24).*

Proof. Since π_1 is a Bayesian inverse of π_0 along p, Equation (3.2) implies the following equality:



By definition of Δ, it is easy to see that the expectation of the diagram above is 1. Therefore it is in the argmax of (3.25), i.e. it is a Nash equilibrium for (3.24). □

Remark 3.4.2. *We do not believe that all equilibria for this game arise through Bayesian inversion, but were unable to find a counterexample.*

Example 3.4.3. *As an example suppose the referents are shapes $R = \{ Bouba, Kiki \}$ and the words are $W = \{ pointy, round \}$. A literal listener would use the following channel $\pi_0 : W \to \mathcal{D}(R)$ to assign words to their referents.*

	Bouba	*Kiki*
round	1	0
pointy	1/2	1/2

A pragmatic speaker would use the Bayesian inverse of this channel to refer to the letters.

	Bouba	*Kiki*
round	2/3	0
pointy	1/3	1

A pragmatic listener would be inclined to use the Bayesian inverse of the pragmatic speaker's channel.

	Bouba	*Kiki*
round	1	0
pointy	1/4	3/4

And so on. We can see that Bayesian inversion correctly captures the pragmatics of matching words with their referents in this restricted context.

Frank and Goodman [FG12] model the conditional distribution $\pi_0 : R \to \mathcal{D}(W)$ with a likelihood based on an information-theoretic measure known as *surprisal* or *information content.*

$$\pi_0(r)(w) = \frac{|R(w)|}{\sum_{w' \in W(r)} |R(w')|}$$

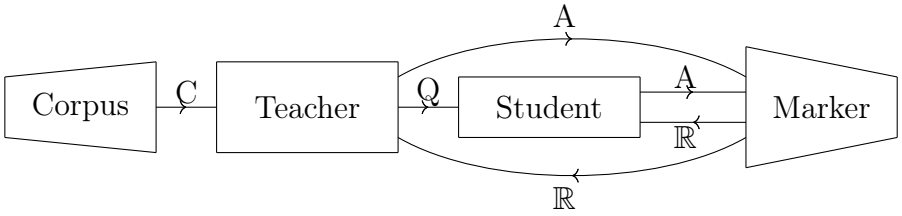
where $R(w)$ is the set of objects to which word w could refer to and $W(r)$ is the set of words that could refer to object r. Then given a prior p over the set of objects, the Bayesian inverse $\pi_1 = \pi_0^\dagger$ can be computed using Bayes rule. In their experiment, the participants were split in three groups: speaker, salience and listeners. They were asked questions to test respectively for the likelihood π_0, the prior p and the posterior predictions π_1 of their model.

Note that the experiment of Frank and Goodman involved three referents (a blue square, a blue circle and a green square) and four words (blue, green, square and circle). The sets $R(w)$ and $W(r)$ were calculated by hand, and computing the Bayesian inverse of π_0 in this case was an easy task. However, as the number of possible referents and words grows, Bayesian inversion quickly becomes computationally intractable without some underlying compositional structure mediating it.

3.4.2 Adversarial question answering

In this section we define a game modelling an interaction between a teacher and a student. The teacher poses questions that the student tries to answer. We assume that the student is incentivised to answer questions correctly, whereas the teacher is incentivised to ask hard questions, resulting in an *adversarial* question answering game (QA). For simplicity, we work in a deterministic setting, i.e. we work in the category of in **Game**(**Set**). We first give a syntactic definition of QA as a diagram, we then instantiate the definition with respect to monoidal grammars and functorial models, which allows us to compute the Nash equilibria for the game.

Let us fix three sets C, Q, A for corpora (i.e. lists of facts), questions and answers respectively. Let U be a set of utilities, which can be taken to be $\mathbb{R}$ or $\mathbb{B}$. A *teacher* $\mathcal{T}\colon \binom{C}{1} \nrightarrow \binom{Q\times A}{U}$ is a utility-maximising player where each strategy represents a function turning facts from the corpus into pairs of questions and answers. A *student* $\mathcal{S}\colon \binom{Q}{1} \nrightarrow \binom{A}{U}$ is a utility-maximising player where each strategy represents a way of turning questions into answers. A *marker* is a strategically trivial open game $\mathcal{M}\colon \binom{A\times A}{U\times U} \nrightarrow \binom{1}{1}$ with trivial play function and a coplay function defined as $\kappa_{\mathcal{M}}(a_{\mathcal{T}}, a_{\mathcal{S}}) = (-d(a_{\mathcal{S}}, a_{\mathcal{T}}), d(a_{\mathcal{S}}, a_{\mathcal{T}}))$ where $d : A \times A \to U$ is a given metric on A. Finally, we model a *corpus* as a strategically trivial open game $f\colon \binom{1}{1} \nrightarrow \binom{C}{1}$ with play function given by $\pi_f(*) = f \in C$. All these open games are composed to obtain a *question answering game* in the following way.



(3.26)

Intuitively, the teacher produces a question from the corpus and gives it to the student who uses his strategy to answer. The marker will receive the correct answer from the teacher together with the answer that the student produced, and output two utilities. The utility of the teacher will be the distance between the student's answer and the correct answer; the utility of the student will be the exact opposite of this quantity. In this sense, question answering is a zero-sum game.

We now instantiate the game defined above with respect to a pregroup grammar $G = (B, V, D, s)$ with a fixed question type $z \in B$. The strategies of the student are given by relational models $\sigma : G \to \mathbf{Rel}$ with $\sigma(z) = 1$, so that given a question $q : u \to z$, $\sigma(q) \in \mathcal{P}(1) = \mathbb{B}$ is the student's answer. In practice, the student may only have a subset of models available to him so we set $\Sigma_{\mathcal{S}} \subseteq \{\sigma : G \to \mathbf{Rel} :\ \sigma(z) = 1\}$.

We assume that the teacher has the particularly simple role of picking a question-answer pair from a set of possible ones, i.e. we take the corpus C to be a list of question-answer pairs (q, a) for $q : u \to z$ and $a \in A$. For simplicity, we assume q is a yes/no question and a is a boolean answer, i.e. $Q = \mathcal{L}(G, z)$ and $A = \mathbb{B}$. The strategies of the teacher are given by indices $\Sigma_{\mathcal{T}} = \{\, 0, 1, \dots n \,\}$, so that the play function $\pi_{\mathcal{T}} : \Sigma_{\mathcal{T}} \times (Q \times A)^* \to Q \times A$ picks the question-answer pair indicated by the index. The marker will compare the teacher's answer a with the student's answer $\sigma(q) \in \mathbb{B}$ using the metric $d : A \times A \to \mathbb{B} :: (a_0, a_1) \mapsto (a_0 = a_1)$ and output boolean utilities in $U = \mathbb{B}$. Plugging these open games as in (3.26), we can compute the set of equilibria by composing the equilibrium functions of its components.

$$E_{\mathcal{G}} = \{(j, \sigma) \in \Sigma_{\mathcal{T}} \times \Sigma_{\mathcal{S}} :\ j \in \operatorname*{argmax}_{i \in \Sigma_{\mathcal{T}}} a_i \neq \sigma(q_i) \wedge \sigma \in \operatorname*{argmax}_{\sigma \in \Sigma_{\mathcal{S}}} (a_j = \sigma(q_j))\}$$

Therefore, in a Nash equilibrium, the teacher will ask the question that the student, even with his best guess, is going to answer in the worst way. The student, on the other hand, is going to answer as correctly as possible.

We analyse the possible outcomes of this game.

1. There is a pair (q_i, a_i) in C that the student cannot answer correctly, i.e. $\forall \sigma \in \Sigma_{\mathcal{S}} :\ \sigma(q_i) \neq a_i$. Then i is a winning strategy for the teacher and (i, σ) is a Nash equilibrium, for any choice of strategy σ for the student. If no such pair exists, then we fall into one of the following cases.

2. The corpus is consistent — i.e. $\exists \sigma : G \to \mathbf{Rel}$ such that $\forall i \cdot \sigma(q_i) = a_i$ — and the student has access to the model σ that answers all the possible questions correctly. Then, the strategy profile (j, σ) is a Nash equilibrium and a winning strategy for the student for any choice j of the teacher.

3. For any choice i of the teacher, the student has a model σ_i that answers q_i correctly. And viceversa, for any strategy σ of the student there is a choice j of the teacher such that $\sigma(q_j) \neq a_j$. Then the set $E_{\mathcal{G}}$ is empty, there is no Nash equilibrium.

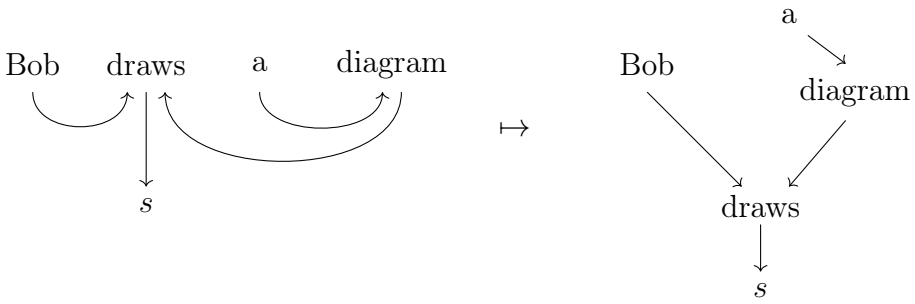
To illustrate the last case, consider a situation where the corpus $C = \{\, (q_0, a_0), (q_1, a_1) \,\}$ has only two elements and the student has only two models $\Sigma_{\mathcal{S}} = \{\, \sigma_0, \sigma_1 \,\}$ such that $\sigma_i(q_i) = a_i$ for $i \in \{\, 0, 1 \,\}$ but $\sigma_0(q_1) \neq a_1$ and $\sigma_1(q_0) \neq a_0$. Then we're in a *matching pennies* scenario, both the teacher and the student have no incentive to choose any one of their strategies and there is no Nash equilibrium. This problem can be ruled out if we allowed the players in the game to have *mixed strategies*, which can be achieved with minor modifications of the open game formalism [Gha+19].

3.4.3 Word sense disambiguation

Word sense disambiguation (WSD) is the task of choosing the correct sense of a word in the context of a sentence. WSD has been argued to be an AI-complete task in the sense that it can be used to simulate other NLP task [Nav09]. In [TN19], Tripodi and

Navigli use methods from evolutionary game theory to obtain state-of-the-art results in the WSD task. The idea is to model WSD as a collaborative game between words where strategies are word senses. In their model, the interactions between words are weighted by how close the words are in a piece of text. In this section, we propose a compositional alternative, where the interaction between words is mediated by the grammatical structure of the sentence they occur in. Concretely, we show that how to build a functor $J : G_W \to \mathbf{Game}$ given a functorial model $F : G_V \to \mathbf{Prob}$ where G_W is a grammar for words and G_V a grammar for word-senses. Given a sentence $u \in \mathcal{L}(G_V)$, the functor J constructs a collaborative game where the Nash equilibrium is given by a choice of sense for each word in u that maximises the score of the sentence u in F.

We work with a *dependency grammar* $G \subseteq (B + V) \times B^*$ where B is a set of basic types and V a set of words, see Definition 1.5.15. Recall from Proposition 1.5.21, that dependency relations are acyclic, i.e. we can always turn the dependency graph into a tree as in the following example:

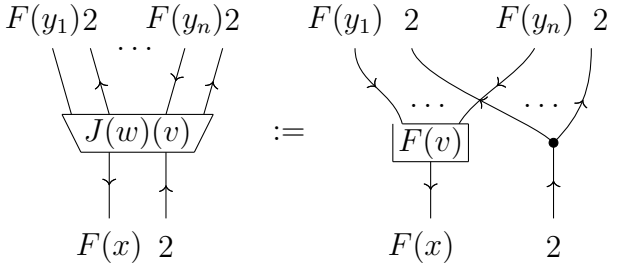


This means that any sentence parsed with a dependency grammar induces a directed acyclic graph of dependencies. We may represent these parses in the free monoidal category $\mathbf{MC}(\tilde{G})$ where $\tilde{G} = B^* \leftarrow V \to B$ is the signature with boxes labelled by words $v \in V$ with inputs given by the symbols that depend on v and a single output given by the symbol on which v depends, as shown in Proposition 1.5.21.

Taking dependency grammars seriously, it is sensible to interpret them directly in the category **Prob** of conditional distributions. Indeed, a functor $F : \tilde{G} \to \mathbf{Prob}$ is defined on objects by a choice of feature set $F(x)$ for every symbol $x \in B$ and on words $v : y_1 y_2 \dots y_n \to x$ by a conditional distribution $F(v) : F(y_1) \times F(y_2) \dots F(y_n) \to \mathcal{D}(F(x))$ indicating the probability that word v has a particular feature given that the words it depends on have particular features. Thus a parsed sentence is sent to a *Bayesian network* where the independency constraints of the network are induced by the dependency structure of the sentence. One may prove formally that functors $\tilde{G} \to \mathbf{Prob}$ induce Bayesian networks using the work of Fong [Fon13].

We are now ready to describe the WSD procedure. Fix a set W of words and V of word-senses with a dependency grammar G_V for senses and a relation $\Sigma \subseteq W \times V$, assigning to each word $w \in W$ the set of its senses $\Sigma(w) \subseteq V$. Composing G_V with Σ, we get a grammar for words G_W. Assume we are given a functorial model $F : G_V \to \mathbf{Prob}$ with $F(s) = 2$, i.e. for any grammatical sentence $g : u \to s$, its image $F(g) \in \mathcal{D}(2) = [0, 1]$ quantifies how likely it is that the sentence is true. We use F to define a functor $J : G_W \to \mathbf{Game}(\mathbf{Prob})$ such that the Nash equilibrium of the image

of any grammatical sentence is an assignement of each word to a sense maximising the overall score of the sentence. On objects J is defined by $J(b) = (F(b), 2)$ for any $b \in B$. Given a word $w : y_1 \dots y_n \to x$ in G_W its image under J is given by an open game $J(w) : (F(y_1) \times F(y_2) \dots F(y_n), 2^n) \to (F(x), 2)$ with strategy set $\Sigma(w)$ (i.e. strategies for word $w \in W$ are given by its senses $v \in \Sigma(w) \subseteq V$) defined for every strategy $v \in \Sigma(w)$ by the following lens:



The best response function in context $[p, k]$ is given by:

$$B_{J(w,t)}(k) = \texttt{argmax}_{v \in \Sigma(w)}(\mathbb{E}(p; J(w)(v); k))$$

Then given a grammatical sentence $g \in \mathbf{MC}(G_W)$ we get a closed game $J(g) : 1 \to 1$ with equilibrium given by:

$$B_{J(g)} = \texttt{argmax}_{v_i \in \Sigma(w_i)}(F(g[w_i := v_i]))$$

where $u[w_i := v_i]$ denotes the sentence obtained by replacing each word $w_i \in W$ with its sense $v_i \in \Sigma(w_i) \subseteq V$. Computing this argmax corresponds precisely to choosing a sense for each word such that the probability that the sentence is true is maximised.

Example 3.4.4. *As an example take the following dependency grammar G with $\tilde{G}$ given by the following morphisms:*

$$\textit{Bob} : 1 \to n\ ,\ \textit{draws} = n \otimes n \to s\ ,\ a : 1 \to d\ ,\ \textit{card} : d \to n\ ,\ \textit{diagram} : d \to n \quad (3.27)$$

The relation $\Sigma \subseteq W \times V$ between words and senses is given by:

$\Sigma(\textit{Bob}) = \{\textit{Bob Coecke}, \textit{Bob Ciaffone}\}$ $\Sigma(\textit{draws}) = \{\textit{draws (pull)}, \textit{draws (picture)}\}$

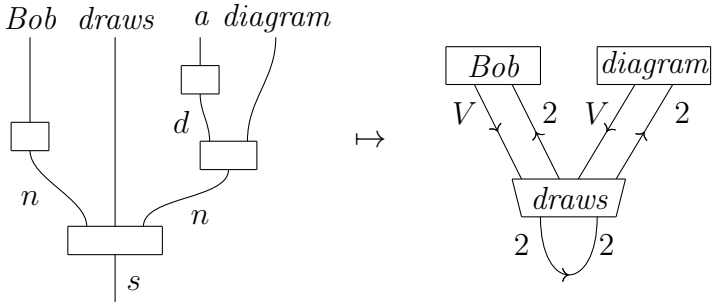
and $\Sigma(x) = \{\, x \,\}$ for $x \in \{\textit{card}, \textit{diagram}, a\}$. The functor $F : G_V \to \mathbf{Prob}$ is defined on objects by $F(d) = 1$ (i.e. determinants are discarded), $F(s) = 2$ and $F(n) = \{\textit{Bob Coecke}, \textit{Bob Ciaffone}, \textit{card}, \textit{diagram}\}$ and on arrows by:

$$F(x)(y) = \begin{cases} 1 & x = y \\ 0 & \textit{otherwise} \end{cases}.$$

for $x \in V \setminus \{\textit{draws (pull)}, \textit{draws (picture)}\}$. The image of the two possible senses of "draw" are given by the following table:

subject	*object*	*draws (picture)*	*draws (pull)*
Bob Coecke	*card*	0.1	0.3
Bob Ciaffone	*card*	0.1	0.9
Bob Coecke	*diagram*	0.9	0.2
Bob Ciaffone	*diagram*	0.1	0.1

Note that a number in $[0,1]$ *is sufficient to specify a distribution in* $\mathcal{D}(2)$*. We get a corresponding functor* $J : G_W \to \mathbf{Game}(\mathbf{Prob})$ *which maps "Bob draws a diagram" as follows:*



Composing the channels according to the structure of the diagram we get a distribution in $\mathcal{D}(2)$ *parametrized over the choice of sense for each word. According to the table above, the expectation of this distribution is maximised when the strategy of the word "Bob" is the sense "Bob Coecke" and the strategy of "draws" is the sense "draws (picture)".*

References

[AD19] AbadiMartín and PlotkinGordon D. "A Simple Differentiable Programming Language". In: *Proceedings of the ACM on Programming Languages* (Dec. 2019). DOI: 10.1145/3371106.

[Abr12] Samson Abramsky. *No-Cloning In Categorical Quantum Mechanics*. Mar. 2012. arXiv: 0910.2401 [quant-ph].

[AC07] Samson Abramsky and Bob Coecke. "A Categorical Semantics of Quantum Protocols". In: *arXiv:quant-ph/0402130* (Mar. 2007). arXiv: quant-ph/0402130.

[AC08] Samson Abramsky and Bob Coecke. "Categorical Quantum Mechanics". In: *arXiv:0808.1023 [quant-ph]* (Aug. 2008). arXiv: 0808.1023 [quant-ph].

[AJ95] Samson Abramsky and Achim Jung. "Domain Theory". In: *Handbook of Logic in Computer Science (Vol. 3): Semantic Structures*. USA: Oxford University Press, Inc., Jan. 1995, pp. 1–168.

[AT10] Samson Abramsky and Nikos Tzevelekos. "Introduction to Categories and Categorical Logic". In: *arXiv:1102.1313 [cs, math]* 813 (2010), pp. 3–94. DOI: 10.1007/978-3-642-12821-9_1. arXiv: 1102.1313 [cs, math].

[AR18] Takanori Adachi and Yoshihiro Ryu. "A Category of Probability Spaces". In: *arXiv:1611.03630 [math]* (Oct. 2018). arXiv: 1611.03630 [math].

[ALM07] Dorit Aharonov, Zeph Landau, and Johann Makowsky. "The Quantum FFT Can Be Classically Simulated". In: *arXiv:quant-ph/0611156* (Mar. 2007). arXiv: quant-ph/0611156.

[Ajd35] K. Ajdukiewiz. "Die Syntaktische Konnexitat". In: *Studia Philosophica* (1935), pp. 1–27.

[ACL19] Afra Alishahi, Grzegorz Chrupała, and Tal Linzen. "Analyzing and Interpreting Neural Networks for NLP: A Report on the First BlackboxNLP Workshop". In: *arXiv:1904.04063 [cs, stat]* (Apr. 2019). arXiv: 1904.04063 [cs, stat].

[AMY16] Noga Alon, Shay Moran, and Amir Yehudayoff. "Sign Rank versus VC Dimension". In: *arXiv:1503.07648 [math]* (July 2016). arXiv: 1503.07648 [math].

[Ane12] Irving H. Anellis. "How Peircean Was the "'Fregean' Revolution" in Logic?" In: *arXiv:1201.0353 [math]* (Jan. 2012). arXiv: 1201.0353 [math].

[AL10] Itai Arad and Zeph Landau. “Quantum Computation and the Evaluation of Tensor Networks”. In: *arXiv:0805.0040 [quant-ph]* (Feb. 2010). arXiv: 0805.0040 [quant-ph].

[AFZ14] Yoav Artzi, Nicholas Fitzgerald, and Luke Zettlemoyer. “Semantic Parsing with Combinatory Categorial Grammars”. en-us. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*. Oct. 2014.

[Ask19] John Ole Askedal. *Peirce and Valency Grammar*. en. De Gruyter Mouton, May 2019. Chap. Signs of Humanity / L’homme et ses signes, pp. 1343–1348.

[BCR18] John C. Baez, Brandon Coya, and Franciscus Rebro. “Props in Network Theory”. In: *arXiv:1707.08321 [math-ph]* (June 2018). arXiv: 1707.08321 [math-ph].

[BE14] John C. Baez and Jason Erbele. “Categories in Control”. In: *arXiv:1405.6881 [quant-ph]* (May 2014). arXiv: 1405.6881 [quant-ph].

[BP17] John C. Baez and Blake S. Pollard. “A Compositional Framework for Reaction Networks”. In: *Reviews in Mathematical Physics* 29.09 (Oct. 2017), p. 1750028. DOI: 10.1142/S0129055X17500283. arXiv: 1704.02051.

[BM02] J. F. Baget and M. L. Mugnier. “Extensions of Simple Conceptual Graphs: The Complexity of Rules and Constraints”. In: *Journal of Artificial Intelligence Research* 16 (June 2002), pp. 425–465. DOI: 10.1613/jair.918. arXiv: 1106.1800.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *arXiv e-prints* 1409 (Sept. 2014), arXiv:1409.0473.

[BKV18] Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. “Globular: An Online Proof Assistant for Higher-Dimensional Rewriting”. In: *Logical Methods in Computer Science ; Volume 14* (2018), Issue 1, 18605974. DOI: 10.23638/LMCS-14(1:8)2018. arXiv: 1612.01093.

[Bar53] Yehoshua Bar-Hillel. “A Quasi-Arithmetical Notation for Syntactic Description”. In: *Language* 29.1 (1953), pp. 47–58. DOI: 10.2307/410452.

[BGG12] Chitta Baral, Marcos Alvarez Gonzalez, and Aaron Gottesman. “The Inverse Lambda Calculus Algorithm for Typed First Order Logic Lambda Calculus and Its Application to Translating English to FOL”. en. In: *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*. Ed. by Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 40–56. DOI: 10.1007/978-3-642-30743-0_4.

[BP83] Jon Barwise and John Perry. *Situations and Attitudes*. MIT Press, 1983.

[Bel57] Richard Bellman. “A Markovian Decision Process”. In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684.

[Ben+03] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. “A Neural Probabilistic Language Model”. In: *The Journal of Machine Learning Research* 3.null (Mar. 2003), pp. 1137–1155.

[BS18] Anton Benz and Jon Stevens. “Game-Theoretic Approaches to Pragmatics”. In: *Annual Review of Linguistics* 4.1 (2018), pp. 173–191. DOI: 10.1146/annurev-linguistics-011817-045641.

[Ber+13] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. “Semantic Parsing on Freebase from Question-Answer Pairs”. In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1533–1544.

[BLG16] Leon Bergen, Roger Levy, and Noah Goodman. “Pragmatic Reasoning through Semantic Inference”. en. In: *Semantics and Pragmatics* 9.0 (May 2016), EARLY ACCESS. DOI: 10.3765/sp.9.20.

[BMT15] Jacob D. Biamonte, Jason Morton, and Jacob W. Turner. “Tensor Network Contractions for #SAT”. In: *Journal of Statistical Physics* 160.5 (Sept. 2015), pp. 1389–1404. DOI: 10.1007/s10955-015-1276-z. arXiv: 1405.7375.

[Bia+00] Anna Maria Bianucci, Alessio Micheli, Alessandro Sperduti, and Antonina Starita. “Application of Cascade Correlation Networks for Structures to Chemistry”. en. In: *Applied Intelligence* 12.1 (Jan. 2000), pp. 117–147. DOI: 10.1023/A:1008368105614.

[BPV06] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. “Relational Lenses: A Language for Updatable Views”. In: *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’06. New York, NY, USA: Association for Computing Machinery, June 2006, pp. 338–347. DOI: 10.1145/1142351.1142399.

[Bol+08] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. “Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge”. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’08. New York, NY, USA: Association for Computing Machinery, June 2008, pp. 1247–1250. DOI: 10.1145/1376616.1376746.

[BHZ19] Joe Bolt, Jules Hedges, and Philipp Zahn. “Bayesian Open Games”. In: *arXiv:1910.03656 [cs, math]* (Oct. 2019). arXiv: 1910.03656 [cs, math].

[Bon+16] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. “Rewriting modulo Symmetric Monoidal Structure”. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (July 2016), pp. 710–719. DOI: 10.1145/2933575.2935316. arXiv: 1602.06771.

[Bon+20] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. “String Diagram Rewrite Theory I: Rewriting with Frobenius Structure”. In: *arXiv:2012.01847 [cs, math]* (Dec. 2020). arXiv: 2012.01847 [cs, math].

[Bon+21] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. “String Diagram Rewrite Theory II: Rewriting with Symmetric Monoidal Structure”. In: *arXiv:2104.14686 [cs, math]* (Apr. 2021). arXiv: 2104.14686 [cs, math].

[BPS17] Filippo Bonchi, Dusko Pavlovic, and Pawel Sobocinski. “Functorial Semantics for Relational Theories”. In: *arXiv:1711.08699 [cs, math]* (Nov. 2017). arXiv: 1711.08699 [cs, math].

[BSS18] Filippo Bonchi, Jens Seeber, and Pawel Sobocinski. “Graphical Conjunctive Queries”. In: *arXiv:1804.07626 [cs]* (Apr. 2018). arXiv: 1804.07626 [cs].

[BSZ14] Filippo Bonchi, Pawe\l Sobociński, and Fabio Zanasi. “A Categorical Semantics of Signal Flow Graphs”. In: *International Conference on Concurrency Theory*. Springer, 2014, pp. 435–450.

[BCW14] Antoine Bordes, Sumit Chopra, and Jason Weston. “Question Answering with Subgraph Embeddings”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 615–620. DOI: 10.3115/v1/D14-1067.

[Bor+13] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. “Translating Embeddings for Modeling Multi-Relational Data”. In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*. NIPS’13. Red Hook, NY, USA: Curran Associates Inc., Dec. 2013, pp. 2787–2795.

[Bor+09] M. Bordewich, M. Freedman, L. Lovász, and D. Welsh. “Approximate Counting and Quantum Computation”. In: *arXiv:0908.2122 [cs]* (Aug. 2009). arXiv: 0908.2122 [cs].

[BPM15] Samuel R. Bowman, Christopher Potts, and Christopher D. Manning. “Recursive Neural Networks Can Learn Logical Semantics”. In: *Proceedings of the 3rd Workshop on Continuous Vector Space Models and Their Compositionality*. Beijing, China: Association for Computational Linguistics, July 2015, pp. 12–21. DOI: 10.18653/v1/W15-4002.

[Bra+18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: http://github.com/google/jax.

[BT00] Geraldine Brady and Todd H. Trimble. “A Categorical Interpretation of C.S. Peirce’s Propositional Logic Alpha”. en. In: *Journal of Pure and Applied Algebra* 149.3 (June 2000), pp. 213–239. DOI: 10.1016/S0022-4049(98)00179-0.

[Bre+82] Joan Bresnan, Ronald M. Kaplan, Stanley Peters, and Annie Zaenen. “Cross-Serial Dependencies in Dutch”. In: *Linguistic Inquiry* 13.4 (1982), pp. 613–635.

[Bro+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner,

Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. “Language Models Are Few-Shot Learners”. In: *arXiv:2005.14165 [cs]* (July 2020). arXiv: 2005.14165 [cs].

[Bul17] Andrei A. Bulatov. “A Dichotomy Theorem for Nonuniform CSPs”. In: *arXiv:1703.03021 [cs]* (2017). DOI: 10.1109/FOCS.2017.37. arXiv: 1703.03021 [cs].

[BM20] Samuele Buro and Isabella Mastroeni. “On the Semantic Equivalence of Language Syntax Formalisms”. en. In: *Theoretical Computer Science* 840 (Nov. 2020), pp. 234–248. DOI: 10.1016/j.tcs.2020.08.022.

[Bus07] Wojciech Buszkowski. “Type Logics and Pregroups”. en. In: *Studia Logica* 87.2 (Dec. 2007), pp. 145–169. DOI: 10.1007/s11225-007-9083-4.

[Bus16] Wojciech Buszkowski. “Syntactic Categories and Types: Ajdukiewicz and Modern Categorial Grammars”. In: 2016. DOI: 10.1163/9789004311763_004.

[BM07] Wojciech Buszkowski and Katarzyna Moroz. “Pregroup Grammars and Context-Free Grammars”. In: 2007.

[CW18] Liwei Cai and William Yang Wang. “KBGAN: Adversarial Learning for Knowledge Graph Embeddings”. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, June 2018, pp. 1470–1480. DOI: 10.18653/v1/N18-1133.

[Can06] R. Ferrer i Cancho. “Why Do Syntactic Links Not Cross?” en. In: *EPL (Europhysics Letters)* 76.6 (Nov. 2006), p. 1228. DOI: 10.1209/epl/i2006-10406-0.

[CW87] A. Carboni and R. F. C. Walters. “Cartesian Bicategories I”. en. In: *Journal of Pure and Applied Algebra* 49.1 (Nov. 1987), pp. 11–32. DOI: 10.1016/0022-4049(87)90121-6.

[CL02] Claudia Casadio and Joachim Lambek. “A Tale of Four Grammars”. en. In: *Studia Logica* 71.3 (Aug. 2002), pp. 315–329. DOI: 10.1023/A:1020564714107.

[CB76] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A Structured English Query Language”. en. In: *Proceedings of the 1976 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control - FIDET ’76*. Not Known: ACM Press, 1976, pp. 249–264. DOI: 10.1145/800296.811515.

[Cha+22] Jireh Yi-Le Chan, Khean Thye Bea, Steven Mun Hong Leow, Seuk Wai Phoong, and Wai Khuen Cheng. “State of the Art: A Review of Sentiment Analysis Based on Sequential Transfer Learning”. In: *Artificial Intelligence Review* (Apr. 2022). DOI: 10.1007/s10462-022-10183-8.

[CM77] Ashok K. Chandra and Philip M. Merlin. “Optimal Implementation of Conjunctive Queries in Relational Data Bases”. en. In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing - STOC ’77*. Boulder, Colorado, United States: ACM Press, 1977, pp. 77–90. DOI: 10.1145/800105.803397.

[Cha+09] Philippe Chaput, Vincent Danos, Prakash Panangaden, and Gordon Plotkin. "Approximating Markov Processes by Averaging". en. In: *Automata, Languages and Programming*. Ed. by Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 127–138. DOI: 10.1007/978-3-642-02930-1_11.

[CK15] Jianpeng Cheng and Dimitri Kartsaklis. "Syntax-Aware Multi-Sense Word Embeddings for Deep Compositional Models of Meaning". In: *arXiv:1508.02354 [cs]* (Aug. 2015). arXiv: 1508.02354 [cs].

[CJ19] Kenta Cho and Bart Jacobs. "Disintegration and Bayesian Inversion via String Diagrams". In: *Mathematical Structures in Computer Science* (Mar. 2019), pp. 1–34. DOI: 10.1017/S0960129518000488. arXiv: 1709.00322.

[Cho+15] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[Cho56] Noam Chomsky. "Three Models for the Description of Language". In: *Journal of Symbolic Logic* 23.1 (1956), pp. 71–72. DOI: 10.2307/2964524.

[Cho57] Noam Chomsky. *Syntactic Structures*. The Hague: Mouton and Co., 1957.

[Chu32] Alonzo Church. "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2 (1932), pp. 346–366. DOI: 10.2307/1968337.

[Chu36] Alonzo Church. "An Unsolvable Problem of Elementary Number Theory". In: *Journal of Symbolic Logic* 1.2 (1936), pp. 73–74. DOI: 10.2307/2268571.

[Chu40] Alonzo Church. "A Formulation of the Simple Theory of Types". In: *The Journal of Symbolic Logic* 5.2 (1940), pp. 56–68. DOI: 10.2307/2266170.

[CM15] Kevin Clark and Christopher D. Manning. "Entity-Centric Coreference Resolution with Model Stacking". In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, July 2015, pp. 1405–1415. DOI: 10.3115/v1/P15-1136.

[CM16] Kevin Clark and Christopher D. Manning. "Deep Reinforcement Learning for Mention-Ranking Coreference Models". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2256–2262. DOI: 10.18653/v1/D16-1245.

[CCS08] Stephen Clark, Bob Coecke, and Mehrnoosh Sadrzadeh. "A Compositional Distributional Model of Meaning". In: *Proceedings of the Second Symposium on Quantum Interaction (QI-2008)* (2008), pp. 133–140.

[CCS10] Stephen Clark, Bob Coecke, and Mehrnoosh Sadrzadeh. "Mathematical Foundations for a Compositional Distributional Model of Meaning". In: *A Festschrift for Jim Lambek*. Ed. by J. van Benthem, M. Moortgat, and W. Buszkowski. Vol. 36. Linguistic Analysis. 2010, pp. 345–384. arXiv: 1003.4394.

[Cod70] E F Codd. “A Relational Model of Data for Large Shared Data Banks”. en. In: 13.6 (1970), p. 11.

[Coe20] Bob Coecke. “The Mathematics of Text Structure”. In: *arXiv:1904.03478 [quant-ph]* (Feb. 2020). arXiv: 1904.03478 [quant-ph].

[Coe+18] Bob Coecke, Giovanni de Felice, Dan Marsden, and Alexis Toumi. “Towards Compositional Distributional Discourse Analysis”. In: *Electronic Proceedings in Theoretical Computer Science* 283 (Nov. 2018), pp. 1–12. DOI: 10.4204/EPTCS.283.1. arXiv: 1811.03277 [cs].

[Coe+20] Bob Coecke, Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. “Foundations for Near-Term Quantum Natural Language Processing”. In: *arXiv:2012.03755 [quant-ph]* (Dec. 2020). arXiv: 2012.03755 [quant-ph].

[Coe+22] Bob Coecke, Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. “How to Make Qubits Speak”. In: *Quantum Computing in the Arts and Humanities: An Introduction to Core Concepts, Theory and Applications*. Ed. by Eduardo Reck Miranda. Cham: Springer International Publishing, 2022, pp. 277–297. DOI: 10.1007/978-3-030-95538-0_8.

[CK17] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge: Cambridge University Press, 2017. DOI: 10.1017/9781316219317.

[Coe+19] Andy Coenen, Emily Reif, Ann Yuan, Been Kim, Adam Pearce, Fernanda Viégas, and Martin Wattenberg. “Visualizing and Measuring the Geometry of BERT”. In: *arXiv:1906.02715 [cs, stat]* (Oct. 2019). arXiv: 1906.02715 [cs, stat].

[Cru+21] G. S. H. Cruttwell, Bruno Gavranović, Neil Ghani, Paul Wilson, and Fabio Zanasi. “Categorical Foundations of Gradient-Based Learning”. In: *arXiv:2103.01931 [cs, math]* (Mar. 2021). arXiv: 2103.01931 [cs, math].

[Cur61] H. B. Curry. “Some Logical Aspects of Grammatical Structure”. In: *Structure Language and Its Mathematical Aspects,* vol. XII. American Mathematical Society, 1961, pp. 56–68.

[DKV02] Víctor Dalmau, Phokion G. Kolaitis, and Moshe Y. Vardi. “Constraint Satisfaction, Bounded Treewidth, and Finite-Variable Logics”. en. In: *Principles and Practice of Constraint Programming - CP 2002*. Ed. by Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Pascal Van Hentenryck. Vol. 2470. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 310–326. DOI: 10.1007/3-540-46135-3_21.

[Dav67a] Donald Davidson. “The Logical Form of Action Sentences”. In: *The Logic of Decision and Action*. Ed. by Nicholas Rescher. University of Pittsburgh Press, 1967, pp. 81–95.

[Dav67b] Donald Davidson. “Truth and Meaning”. In: *Synthese* 17.3 (1967), pp. 304–323.

[dKM20] Niel de Beaudrap, Aleks Kissinger, and Konstantinos Meichanetzidis. “Tensor Network Rewriting Strategies for Satisfiability and Counting”. In: *arXiv:2004.06455 [quant-ph]* (Apr. 2020). arXiv: 2004.06455 [quant-ph].

[de +21] Giovanni de Felice, Elena Di Lavore, Mario Román, and Alexis Toumi. "Functorial Language Games for Question Answering". In: *Electronic Proceedings in Theoretical Computer Science* 333 (Feb. 2021), pp. 311–321. DOI: 10.4204/EPTCS.333.21. arXiv: 2005.09439 [cs].

[dMT20] Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. "Functorial Question Answering". In: *Electronic Proceedings in Theoretical Computer Science* 323 (Sept. 2020), pp. 84–94. DOI: 10.4204/EPTCS.323.6. arXiv: 1905.07408 [cs, math].

[dTC21] Giovanni de Felice, Alexis Toumi, and Bob Coecke. "DisCoPy: Monoidal Categories in Python". In: *Electronic Proceedings in Theoretical Computer Science* 333 (Feb. 2021), pp. 183–197. DOI: 10.4204/EPTCS.333.13. arXiv: 2005.02975 [math].

[De 47] Augustus De Morgan. *Formal Logic: Or, The Calculus of Inference, Necessary and Probable*. en. Taylor and Walton, 1847.

[DBM15] Wim De Mulder, Steven Bethard, and Marie-Francine Moens. "A Survey on the Application of Recurrent Neural Networks to Statistical Language Modeling". en. In: *Computer Speech & Language* 30.1 (Mar. 2015), pp. 61–98. DOI: 10.1016/j.csl.2014.09.005.

[DP05] Sylvain Degeilh and Anne Preller. "Efficiency of Pregroups and the French Noun Phrase". en. In: *Journal of Logic, Language and Information* 14.4 (Oct. 2005), pp. 423–444. DOI: 10.1007/s10849-005-1242-2.

[Del19] Antonin Delpeuch. "Autonomization of Monoidal Categories". In: *arXiv:1411.3827 [cs, math]* (June 2019). arXiv: 1411.3827 [cs, math].

[DV19a] Antonin Delpeuch and Jamie Vicary. "Normalization for Planar String Diagrams and a Quadratic Equivalence Algorithm". In: *arXiv:1804.07832 [cs]* (Sept. 2019). arXiv: 1804.07832 [cs].

[DEB19] Samuel Desrosiers, Glen Evenbly, and Thomas Baker. "Survey of Tensor Networks". In: (2019), F68.006.

[Dev+19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv:1810.04805 [cs]* (May 2019). arXiv: 1810.04805 [cs].

[DdR22] Elena Di Lavore, Giovanni de Felice, and Mario Román. *Monoidal Streams for Dataflow Programming*. Feb. 2022. DOI: 10.48550/arXiv.2202.02061. arXiv: 2202.02061 [cs, math].

[Don+14] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. "Knowledge Vault: A Web-Scale Approach to Probabilistic Knowledge Fusion". In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '14. New York, NY, USA: Association for Computing Machinery, Aug. 2014, pp. 601–610. DOI: 10.1145/2623330.2623623.

[Dos+20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *arXiv:2010.11929 [cs]* (Oct. 2020). arXiv: 2010.11929 [cs].

[DV19b] Lawrence Dunn and Jamie Vicary. "Coherence for Frobenius Pseudomonoids and the Geometry of Linear Proofs". In: *arXiv:1601.05372 [cs]* (2019). DOI: 10.23638/LMCS-15(3:5)2019. arXiv: 1601.05372 [cs].

[Ear70] Jay Earley. "An Efficient Context-Free Parsing Algorithm". In: *Communications of the ACM* 13.2 (Feb. 1970), pp. 94–102. DOI: 10.1145/362007.362035.

[Edu+18] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. "Understanding Back-Translation at Scale". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 489–500. DOI: 10.18653/v1/D18-1045.

[EHL19] Stavros Efthymiou, Jack Hidary, and Stefan Leichenauer. "TensorNetwork for Machine Learning". In: *arXiv:1906.06329 [cond-mat, physics:physics, stat]* (June 2019). arXiv: 1906.06329 [cond-mat, physics:physics, stat].

[Ein16] Albert Einstein. "The Foundation of the General Theory of Relativity". en. In: *Annalen Phys.* 49.7 (1916), pp. 769–822. DOI: 10.1002/andp.200590044.

[Eis13] J. Eisert. "Entanglement and Tensor Network States". In: *arXiv:1308.3318 [cond-mat, physics:quant-ph]* (Sept. 2013). arXiv: 1308.3318 [cond-mat, physics:quant-ph].

[Ela06] Pradheep Elango. *Coreference Resolution: A Survey*. 2006.

[Elm90] Jeffrey L. Elman. "Finding Structure in Time". en. In: *Cognitive Science* 14.2 (1990), pp. 179–211. DOI: 10.1207/s15516709cog1402_1.

[Fat+20] Bahare Fatemi, Perouz Taslakian, David Vazquez, and David Poole. "Knowledge Hypergraphs: Prediction Beyond Binary Relations". In: *arXiv:1906.00137 [cs, stat]* (July 2020). arXiv: 1906.00137 [cs, stat].

[Fon13] Brendan Fong. "Causal Theories: A Categorical Perspective on Bayesian Networks". In: *arXiv:1301.6201 [math]* (Jan. 2013). arXiv: 1301.6201 [math].

[FJ19] Brendan Fong and Michael Johnson. "Lenses and Learners". In: *arXiv:1903.03671 [cs, math]* (May 2019). arXiv: 1903.03671 [cs, math].

[FS18a] Brendan Fong and David I. Spivak. "Graphical Regular Logic". In: *arXiv:1812.05765 [cs, math]* (Dec. 2018). arXiv: 1812.05765 [cs, math].

[FS18b] Brendan Fong and David I. Spivak. "Hypergraph Categories". In: *arXiv:1806.08304 [cs, math]* (June 2018). arXiv: 1806.08304 [cs, math].

[FST19] Brendan Fong, David I. Spivak, and Rémy Tuyéras. "Backprop as Functor: A Compositional Perspective on Supervised Learning". In: *arXiv:1711.10455 [cs, math]* (May 2019). arXiv: 1711.10455 [cs, math].

[Fow08] Timothy A. D. Fowler. "Efficiently Parsing with the Product-Free Lambek Calculus". In: *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*. Manchester, UK: Coling 2008 Organizing Committee, Aug. 2008, pp. 217–224.

[Fox76] Thomas Fox. "Coalgebras and Cartesian Categories". In: *Communications in Algebra* 4.7 (Jan. 1976), pp. 665–667. DOI: 10.1080/00927877608822127.

[FG12] Michael C. Frank and Noah D. Goodman. "Predicting Pragmatic Reasoning in Language Games". en. In: *Science* 336.6084 (May 2012), pp. 998–998. DOI: 10.1126/science.1218633.

[Fra09] Michael Franke. "Signal to Act: Game Theory in Pragmatics". In: (Jan. 2009).

[FJ16] Michael Franke and Gerhard Jäger. "Probabilistic Pragmatics, or Why Bayes' Rule Is Probably Important for Pragmatics". en. In: *Zeitschrift für Sprachwissenschaft* 35.1 (June 2016), pp. 3–44. DOI: 10.1515/zfs-2016-0002.

[FGS98] P. Frasconi, M. Gori, and A. Sperduti. "A General Framework for Adaptive Processing of Data Structures". In: *IEEE Transactions on Neural Networks* 9.5 (Sept. 1998), pp. 768–786. DOI: 10.1109/72.712151.

[FLW00] Michael Freedman, Michael Larsen, and Zhenghan Wang. "A Modular Functor Which Is Universal for Quantum Computation". In: *arXiv:quant-ph/0001108* (Feb. 2000). arXiv: quant-ph/0001108.

[FKW02] Michael H. Freedman, Alexei Kitaev, and Zhenghan Wang. "Simulation of Topological Field Theories by Quantum Computers". In: *Communications in Mathematical Physics* 227.3 (June 2002), pp. 587–603. DOI: 10.1007/s002200200635. arXiv: quant-ph/0001071.

[Fre14] Gottlob Frege. "Letter to Jourdain". In: *The Frege Reader*. Oxford: Blackwell Publishing, 1914, pp. 319–321.

[Fri20] Tobias Fritz. "A Synthetic Approach to Markov Kernels, Conditional Independence and Theorems on Sufficient Statistics". In: *arXiv:1908.07021 [cs, math, stat]* (Mar. 2020). arXiv: 1908.07021 [cs, math, stat].

[Gai65] Haim Gaifman. "Dependency Systems and Phrase-Structure Systems". en. In: *Information and Control* 8.3 (June 1965), pp. 304–337. DOI: 10.1016/S0019-9958(65)90232-9.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[Gat+13] A. Gatt, R. van Gompel, K. van Deemter, and E. J. Krahmer. "Are We Bayesian Referring Expression Generators?" English. In: *Proceedings of the CogSci workshop on the production of referring expressions: bridging the gap between cognitive and computational approaches to reference (PRE-CogSci 2013)* (2013), pp. 1–6.

[GHC98] Niyu Ge, John Hale, and Eugene Charniak. "A Statistical Approach to Anaphora Resolution". In: *Sixth Workshop on Very Large Corpora*. 1998.

[GSC00] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Neural Computation* 12.10 (Oct. 2000), pp. 2451–2471. DOI: 10.1162/089976600300015015.

[Gha+18] Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. "Compositional game theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 2018, pp. 472–481. DOI: 10.3982/ECTA6297.

[Gha+19] Neil Ghani, Clemens Kupke, Alasdair Lambert, and Fredrik Nordvall Forsberg. "Compositional Game Theory with Mixed Strategies: Probabilistic Open Games Using a Distributive Law". English. In: *Applied Category Theory Conference 2019*. July 2019.

[Gla+19] Ivan Glasser, Ryan Sweke, Nicola Pancotti, Jens Eisert, and J. Ignacio Cirac. "Expressive Power of Tensor-Network Factorizations for Probabilistic Modeling, with Applications from Hidden Markov Models to Quantum Machine Learning". In: *arXiv:1907.03741 [cond-mat, physics:quant-ph, stat]* (Nov. 2019). arXiv: 1907.03741 [cond-mat, physics:quant-ph, stat].

[Gog+77] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. "Initial Algebra Semantics and Continuous Algebras". In: *Journal of the ACM* 24.1 (Jan. 1977), pp. 68–95. DOI: 10.1145/321992.321997.

[GK96] C. Goller and A. Kuchler. "Learning Task-Dependent Distributed Representations by Backpropagation through Structure". In: *Proceedings of International Conference on Neural Networks (ICNN'96)*. Vol. 1. June 1996, 347–352 vol.1. DOI: 10.1109/ICNN.1996.548916.

[Goo+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Inc., 2014, pp. 2672–2680.

[GS13] Noah Goodman and Andreas Stuhlmüller. "Knowledge and Implicature: Modeling Language Understanding as Social Cognition". In: *Topics in cognitive science* 5 (Jan. 2013), pp. 173–184. DOI: 10.1111/tops.12007.

[GS14] Alexander Gouberman and Markus Siegle. "Markov Reward Models and Markov Decision Processes in Discrete and Continuous Time: Performance Evaluation and Optimization". en. In: *Stochastic Model Checking. Rigorous Dependability Analysis Using Model Checking Techniques for Stochastic Systems: International Autumn School, ROCKS 2012, Vahrn, Italy, October 22-26, 2012, Advanced Lectures*. Ed. by Anne Remke and Mariëlle Stoelinga. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, pp. 156–241. DOI: 10.1007/978-3-662-45489-3_6.

[GK20] Johnnie Gray and Stefanos Kourtis. "Hyper-Optimized Tensor Network Contraction". In: *arXiv:2002.01935 [cond-mat, physics:physics, physics:quant-ph]* (Feb. 2020). arXiv: 2002.01935 [cond-mat, physics:physics, physics:quant-ph].

[GS11] Edward Grefenstette and Mehrnoosh Sadrzadeh. “Experimental Support for a Categorical Compositional Distributional Model of Meaning”. In: *The 2014 Conference on Empirical Methods on Natural Language Processing*. 2011, pp. 1394–1404. arXiv: 1106.4058.

[Gre65] Sheila A. Greibach. “A New Normal-Form Theorem for Context-Free Phrase Structure Grammars”. In: *Journal of the ACM* 12.1 (Jan. 1965), pp. 42–52. DOI: 10.1145/321250.321254.

[Gri67] Herbert Paul Grice. “Logic and Conversation”. In: *Studies in the Way of Words*. Ed. by Paul Grice. Harvard University Press, 1967, pp. 41–58.

[HOD12] Sherzod Hakimov, Salih Atilay Oto, and Erdogan Dogdu. “Named Entity Recognition and Disambiguation Using Linked Data and Graph-Based Centrality Scoring”. In: *Proceedings of the 4th International Workshop on Semantic Web Information Management*. SWIM ’12. New York, NY, USA: Association for Computing Machinery, May 2012, pp. 1–7. DOI: 10.1145/2237867.2237871.

[HL79] Per-Kristian Halvorsen and William A. Ladusaw. “Montague’s ’Universal Grammar’: An Introduction for the Linguist”. In: *Linguistics and Philosophy* 3.2 (1979), pp. 185–223.

[Har+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[HR76] W.s. Hatcher and T. Rus. “Context-Free Algebras”. In: *Journal of Cybernetics* 6.1-2 (Jan. 1976), pp. 65–77. DOI: 10.1080/01969727608927525.

[Hau35] Felix Hausdorff. *Gesammelte Werke Band III: Deskripte Mengenlehre und Topologie*. de. Ed. by Ulrich Felgner, Horst Herrlich, Mirek Husek, Vladimir Kanovei, Peter Koepke, Gerhard Preuß, Walter Purkert, and Erhard Scholz. Felix Hausdorff - Gesammelte Werke. Berlin Heidelberg: Springer-Verlag, 1935. DOI: 10.1007/978-3-540-76807-4.

[HS20] Nathan Haydon and Pawel Sobocinski. “Compositional Diagrammatic First-Order Logic”. In: *11th International Conference on the Theory and Application of Diagrams (DIAGRAMS 2020)*. 2020. URL: https://www.ioc.ee/~pawel/papers/peirce.pdf.

[Hay64] David G. Hays. “Dependency Theory: A Formalism and Some Observations”. In: *Language* 40.4 (1964), pp. 511–525. DOI: 10.2307/411934.

[Hed17] Jules Hedges. “Coherence for Lenses and Open Games”. In: *arXiv:1704.02230 [cs, math]* (Sept. 2017). arXiv: 1704.02230 [cs, math].

[HL18] Jules Hedges and Martha Lewis. “Towards Functorial Language-Games”. In: *Electronic Proceedings in Theoretical Computer Science* 283 (Nov. 2018), pp. 89–102. DOI: 10.4204/eptcs.283.7.

[HV19] Chris Heunen and Jamie Vicary. *Categories for Quantum Theory: An Introduction*. Oxford University Press, 2019.

[Hob78] Jerry R. Hobbs. “Resolving Pronoun References”. en. In: *Lingua* 44.4 (Apr. 1978), pp. 311–338. DOI: 10.1016/0024-3841(78)90006-2.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[Hon+20] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. *spaCy: Industrial-strength Natural Language Processing in Python*. 2020. DOI: 10.5281/zenodo.1212303.

[HJ12] Roger A Horn and Charles R Johnson. *Matrix Analysis*. Cambridge University Press, 2012.

[How60] Roland Howard. *Dynamic Programming and Markov Processes*. MIT press, Cambridge, 1960.

[Imm87] Neil Immerman. “Languages That Capture Complexity Classes”. In: *SIAM Journal of Computing* 16 (1987), pp. 760–778.

[JKZ18] Bart Jacobs, Aleks Kissinger, and Fabio Zanasi. “Causal Inference by String Diagram Surgery”. In: *arXiv:1811.08338 [cs, math]* (Nov. 2018). arXiv: 1811.08338 [cs, math].

[JZ21] Theo M. V. Janssen and Thomas Ede Zimmermann. “Montague Semantics”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2021. Metaphysics Research Lab, Stanford University, 2021.

[JSV04] Mark Jerrum, Alistair Sinclair, and Eric Vigoda. “A Polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Nonnegative Entries”. In: *Journal of the ACM* 51.4 (July 2004), pp. 671–697. DOI: 10.1145/1008731.1008738.

[JRW12] Michael Johnson, Robert Rosebrugh, and R. J. Wood. “Lenses, Fibrations and Universal Translations†”. en. In: *Mathematical Structures in Computer Science* 22.1 (Feb. 2012), pp. 25–42. DOI: 10.1017/S0960129511000442.

[Jos85] Aravind K. Joshi. “Tree Adjoining Grammars: How Much Context-Sensitivity Is Required to Provide Reasonable Structural Descriptions?” In: *Natural Language Parsing: Psychological, Computational, and Theoretical Perspectives*. Ed. by Arnold M. Zwicky, David R. Dowty, and Lauri Karttunen. Studies in Natural Language Processing. Cambridge: Cambridge University Press, 1985, pp. 206–250. DOI: 10.1017/CBO9780511597855.007.

[JS91] A. Joyal and R. Street. “The Geometry of Tensor Calculus, I”. In: *Advances in Mathematics* 88.1 (1991), pp. 55–112. DOI: 10.1016/0001-8708(91)90003-p.

[Jum+21] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. "Highly Accurate Protein Structure Prediction with AlphaFold". In: *Nature* 596.7873 (Aug. 2021), pp. 583–589. DOI: 10.1038/s41586-021-03819-2.

[JM08] Daniel Jurafsky and James Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Vol. 2. United States: Prentice Hall PTR, Feb. 2008.

[KGB14] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. "A Convolutional Neural Network for Modelling Sentences". In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Baltimore, Maryland: Association for Computational Linguistics, June 2014, pp. 655–665. DOI: 10.3115/v1/P14-1062.

[Kar+21] Dimitri Kartsaklis, Ian Fan, Richie Yeung, Anna Pearson, Robin Lorenz, Alexis Toumi, Giovanni de Felice, Konstantinos Meichanetzidis, Stephen Clark, and Bob Coecke. "Lambeq: An Efficient High-Level Python Library for Quantum NLP". In: *arXiv:2110.04236 [quant-ph]* (Oct. 2021). arXiv: 2110.04236 [quant-ph].

[Kea+16] Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. "Molecular Graph Convolutions: Moving beyond Fingerprints". In: *Journal of Computer-Aided Molecular Design* 30.8 (Aug. 2016), pp. 595–608. DOI: 10.1007/s10822-016-9938-8.

[Kv19] Aleks Kissinger and John van de Wetering. "PyZX: Large Scale Automated Diagrammatic Reasoning". In: *arXiv:1904.04735 [quant-ph]* (Apr. 2019). arXiv: 1904.04735 [quant-ph].

[Kv20] Aleks Kissinger and John van de Wetering. "Reducing T-Count with the ZX-Calculus". In: *Physical Review A* 102.2 (Aug. 2020), p. 022406. DOI: 10.1103/PhysRevA.102.022406. arXiv: 1903.10477.

[KH25] H. A. Kramers and W. Heisenberg. "Über die Streuung von Strahlung durch Atome". de. In: *Zeitschrift für Physik* 31.1 (Feb. 1925), pp. 681–708. DOI: 10.1007/BF02980624.

[KM14] Jayant Krishnamurthy and Tom M. Mitchell. "Joint Syntactic and Semantic Parsing with Combinatory Categorial Grammar". In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Baltimore, Maryland: Association for Computational Linguistics, June 2014, pp. 1188–1198. DOI: 10.3115/v1/P14-1112.

[KSJ18] Marco Kuhlmann, Giorgio Satta, and Peter Jonsson. "On the Complexity of CCG Parsing". In: *Computational Linguistics* 44.3 (Sept. 2018), pp. 447–482. DOI: 10.1162/coli_a_00324.

[Kus06] Boris A. Kushner. “The Constructive Mathematics of A. A. Markov”. In: *The American Mathematical Monthly* 113.6 (2006), pp. 559–566. DOI: 10.2307/27641983.

[Lam86] J. Lambek. “Cartesian Closed Categories and Typed λ-Calculi”. en. In: *Combinators and Functional Programming Languages*. Ed. by Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1986, pp. 136–175. DOI: 10.1007/3-540-17184-3_44.

[Lam88] J. Lambek. “Categorial and Categorical Grammars”. en. In: *Categorial Grammars and Natural Language Structures*. Ed. by Richard T. Oehrle, Emmon Bach, and Deirdre Wheeler. Studies in Linguistics and Philosophy. Dordrecht: Springer Netherlands, 1988, pp. 297–317. DOI: 10.1007/978-94-015-6878-4_11.

[Lam99a] J. Lambek. “Type Grammar Revisited”. en. In: *Logical Aspects of Computational Linguistics*. Ed. by Alain Lecomte, François Lamarche, and Guy Perrier. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1999, pp. 1–27. DOI: 10.1007/3-540-48975-4_1.

[LS86] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Vol. 7. USA: Cambridge University Press, 1986.

[Lam08] Jim Lambek. *From Word to Sentence: A Computational Algebraic Approach to Grammar*. Open Access Publications. Polimetrica, 2008.

[Lam58] Joachim Lambek. “The Mathematics of Sentence Structure”. In: *The American Mathematical Monthly* 65.3 (Mar. 1958), pp. 154–170. DOI: 10.1080/00029890.1958.11989160.

[Lam68] Joachim Lambek. “Deductive Systems and Categories”. en. In: *Mathematical systems theory* 2.4 (Dec. 1968), pp. 287–318. DOI: 10.1007/BF01703261.

[Lam99b] Joachim Lambek. “Deductive Systems and Categories in Linguistics”. en. In: *Logic, Language and Reasoning: Essays in Honour of Dov Gabbay*. Ed. by Hans Jürgen Ohlbach and Uwe Reyle. Trends in Logic. Dordrecht: Springer Netherlands, 1999, pp. 279–294. DOI: 10.1007/978-94-011-4574-9_12.

[LF04] Shalom Lappin and C. Fox. “An Expressive First-Order Logic for Natural Language Semantics”. en. In: *Institute of Philosophy* (2004).

[LHH20] Md Tahmid Rahman Laskar, Jimmy Xiangji Huang, and Enamul Hoque. “Contextualized Embeddings Based Transformer Encoder for Sentence Similarity Modeling in Answer Selection Task”. In: *Proceedings of the 12th Language Resources and Evaluation Conference*. Marseille, France: European Language Resources Association, May 2020, pp. 5505–5514.

[Law63] F: W. Lawvere. “Functorial Semantics of Algebraic Theories”. PhD thesis. Columbia University, 1963.

[Lee+17] Kenton Lee, Luheng He, Mike Lewis, and Luke Zettlemoyer. “End-to-End Neural Coreference Resolution”. In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sept. 2017, pp. 188–197. DOI: 10.18653/v1/D17-1018.

[Leh+14] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, and Christian Bizer. "DBpedia - A Large-Scale, Multilingual Knowledge Base Extracted from Wikipedia". In: *Semantic Web Journal* 6 (Jan. 2014). DOI: 10.3233/SW-140134.

[Lew69] David K. Lewis. *Convention: A Philosophical Study*. Wiley-Blackwell, 1969.

[Li+16] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. "Deep Reinforcement Learning for Dialogue Generation". en. In: *arXiv:1606.01541 [cs]* (Sept. 2016). arXiv: 1606.01541 [cs].

[Li+17] Jiwei Li, Will Monroe, Tianlin Shi, Sébastien Jean, Alan Ritter, and Dan Jurafsky. "Adversarial Learning for Neural Dialogue Generation". In: *arXiv:1701.06547 [cs]* (Sept. 2017). arXiv: 1701.06547 [cs].

[LB02] Edward Loper and Steven Bird. *NLTK: The Natural Language Toolkit*. May 2002. DOI: 10.48550/arXiv.cs/0205028. arXiv: cs/0205028.

[Lor+21] Robin Lorenz, Anna Pearson, Konstantinos Meichanetzidis, Dimitri Kartsaklis, and Bob Coecke. "QNLP in Practice: Running Compositional Models of Meaning on a Quantum Computer". In: *arXiv:2102.12846 [quant-ph]* (Feb. 2021). arXiv: 2102.12846 [quant-ph].

[Ma+19] Yunpu Ma, Volker Tresp, Liming Zhao, and Yuyi Wang. "Variational Quantum Circuit Model for Knowledge Graphs Embedding". In: *arXiv:1903.00556 [quant-ph]* (Feb. 2019). arXiv: 1903.00556 [quant-ph].

[Mac71] S. Mac Lane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[MS99] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[Mar+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[Mei+21] Konstantinos Meichanetzidis, Stefano Gogioso, Giovanni de Felice, Nicolò Chiappori, Alexis Toumi, and Bob Coecke. "Quantum Natural Language Processing on Near-Term Quantum Computers". In: *Electronic Proceedings in Theoretical Computer Science* 340 (Sept. 2021), pp. 213–229. DOI: 10.4204/EPTCS.340.11. arXiv: 2005.04147 [quant-ph].

[Mei+20] Konstantinos Meichanetzidis, Alexis Toumi, Giovanni de Felice, and Bob Coecke. “Grammar-Aware Question-Answering on Quantum Computers”. In: *arXiv:2012.03756 [quant-ph]* (Dec. 2020). arXiv: 2012.03756 [quant-ph].

[Meu+17] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. DOI: 10.7717/peerj-cs.103.

[MSS04] Alessio Micheli, Diego Sona, and Alessandro Sperduti. “Contextual Processing of Structured Data by Recursive Cascade Correlation”. eng. In: *IEEE transactions on neural networks* 15.6 (Nov. 2004), pp. 1396–1410. DOI: 10.1109/TNN.2004.837783.

[Mik+10] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernocký, and Sanjeev Khudanpur. “Recurrent Neural Network Based Language Model”. In: *Proceedings of the 11th Annual Conference of the International Speech Communication Association, INTERSPEECH 2010*. Vol. 2. Jan. 2010, pp. 1045–1048.

[Mil56] George A. Miller. “The Magical Number Seven, plus or Minus Two: Some Limits on Our Capacity for Processing Information”. In: *Psychological Review* 63.2 (1956), pp. 81–97. DOI: 10.1037/h0043158.

[MP15] Will Monroe and Christopher Potts. “Learning in the Rational Speech Acts Model”. In: *arXiv:1510.06807 [cs]* (Oct. 2015). arXiv: 1510.06807 [cs].

[Mon70a] Richard Montague. “English as a Formal Language”. In: *Linguaggi Nella Societa e Nella Tecnica*. Ed. by Bruno Visentini. Edizioni di Communita, 1970, pp. 188–221.

[Mon70b] Richard Montague. “Universal Grammar”. en. In: *Theoria* 36.3 (1970), pp. 373–398. DOI: 10.1111/j.1755-2567.1970.tb00434.x.

[Mon73] Richard Montague. “The Proper Treatment of Quantification in Ordinary English”. In: *Approaches to Natural Language*. Ed. by K. J. J. Hintikka, J. Moravcsic, and P. Suppes. Dordrecht: Reidel, 1973, pp. 221–242.

[Moo88] Michael Moortgat. *Categorial Investigations: Logical and Linguistic Aspects of the Lambek Calculus*. 9. Walter de Gruyter, 1988.

[MR12a] Richard Moot and Christian Retoré. “Lambek Calculus and Montague Grammar”. en. In: *The Logic of Categorial Grammars: A Deductive Account of Natural Language Syntax and Semantics*. Ed. by Richard Moot and Christian Retoré. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 65–99. DOI: 10.1007/978-3-642-31555-8_3.

[MR12b] Richard Moot and Christian Retoré. "The Multimodal Lambek Calculus". en. In: *The Logic of Categorial Grammars: A Deductive Account of Natural Language Syntax and Semantics*. Ed. by Richard Moot and Christian Retoré. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 149–191. DOI: 10.1007/978-3-642-31555-8_5.

[MR12c] Richard Moot and Christian Retoré. "The Non-Associative Lambek Calculus". en. In: *The Logic of Categorial Grammars: A Deductive Account of Natural Language Syntax and Semantics*. Ed. by Richard Moot and Christian Retoré. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 101–147. DOI: 10.1007/978-3-642-31555-8_4.

[Mor11] Katarzyna Moroz. "A Savateev-Style Parsing Algorithm for Pregroup Grammars". en. In: *Formal Grammar*. Ed. by Philippe de Groote, Markus Egg, and Laura Kallmeyer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 133–149. DOI: 10.1007/978-3-642-20169-1_9.

[Nav09] Roberto Navigli. "Word Sense Disambiguation: A Survey". In: *ACM Computing Surveys* 41.2 (Feb. 2009), 10:1–10:69. DOI: 10.1145/1459352.1459355.

[NTK11] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. "A Three-Way Model for Collective Learning on Multi-Relational Data". In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML'11. Madison, WI, USA: Omnipress, June 2011, pp. 809–816.

[OGo19] Bryan O'Gorman. "Parameterization of Tensor Network Contraction". In: *arXiv:1906.00013 [quant-ph]* (2019), 19 pages. DOI: 10.4230/LIPIcs.TQC.2019.10. arXiv: 1906.00013 [quant-ph].

[OBW88] Richard T. Oehrle, E. Bach, and Deirdre Wheeler, eds. *Categorial Grammars and Natural Language Structures*. en. Studies in Linguistics and Philosophy. Springer Netherlands, 1988. DOI: 10.1007/978-94-015-6878-4.

[OK19] Ilsang Ohn and Yongdai Kim. "Smooth Function Approximation by Deep Neural Networks with General Activation Functions". en. In: *Entropy* 21.7 (July 2019), p. 627. DOI: 10.3390/e21070627.

[OMK19] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. "A Survey of the Usages of Deep Learning in Natural Language Processing". In: *arXiv:1807.10854 [cs]* (Dec. 2019). arXiv: 1807.10854 [cs].

[Par76] Barbara Partee. *Montague Grammar*. en. Elsevier, 1976. DOI: 10.1016/C2013-0-11289-5.

[Pas+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL:

http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[Pei65] C. S. Peirce. "On a New List of Categories". In: *Proceedings of the American Academy of Arts and Sciences* 7 (1865), pp. 287–298. DOI: 10.2307/20179567.

[Pei97] Charles S. Peirce. "The Logic of Relatives". In: *The Monist* 7.2 (1897), pp. 161–217.

[Pei06] Charles Santiago Sanders Peirce. "Prolegomena to an Apology for Pragmaticism". In: *The Monist* 16.4 (1906), pp. 492–546.

[Pen71] Roger Penrose. "Applications of Negative Dimensional Tensors". en. 1971.

[Pen93] M. Pentus. "Lambek Grammars Are Context Free". In: *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*. June 1993, pp. 429–433. DOI: 10.1109/LICS.1993.287565.

[PV17] Vasily Pestun and Yiannis Vlassopoulos. "Tensor Network Language Model". In: *arXiv:1710.10248 [cond-mat, stat]* (Oct. 2017). arXiv: 1710.10248 [cond-mat, stat].

[PGW17] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. "Profunctor Optics: Modular Data Accessors". In: *The Art, Science, and Engineering of Programming* 1.2 (Apr. 2017), p. 7. DOI: 10.22152/programming-journal.org/2017/1/7. arXiv: 1703.10857.

[Pop+20] Martin Popel, Marketa Tomkova, Jakub Tomek, Łukasz Kaiser, Jakob Uszkoreit, Ondřej Bojar, and Zdeněk Žabokrtský. "Transforming Machine Translation: A Deep Learning System Reaches News Translation Quality Comparable to Human Professionals". en. In: *Nature Communications* 11.1 (Sept. 2020), p. 4381. DOI: 10.1038/s41467-020-18073-9.

[Pos47] Emil L. Post. "Recursive Unsolvability of a Problem of Thue". EN. In: *Journal of Symbolic Logic* 12.1 (Mar. 1947), pp. 1–11.

[PR97] John Power and Edmund Robinson. "Premonoidal Categories and Notions of Computation". In: *Mathematical Structures in Computer Science* 7.5 (Oct. 1997), pp. 453–468. DOI: 10.1017/S0960129597002375.

[Pre07] Anne Preller. "Linear Processing with Pregroups". en. In: *Studia Logica* 87.2-3 (Dec. 2007), pp. 171–197. DOI: 10.1007/s11225-007-9087-0.

[PL07] Anne Preller and Joachim Lambek. "Free Compact 2-Categories". en. In: *Mathematical Structures in Computer Science* 17.02 (Apr. 2007), p. 309. DOI: 10.1017/S0960129506005901.

[Qi+20] Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. "Stanza: A Python Natural Language Processing Toolkit for Many Human Languages". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. 2020.

[Ren19] Jérôme Renault. "A Tutorial on Zero-Sum Stochastic Games". May 2019.

[Ret05] Christian Retoré. *The Logic of Categorial Grammars: Lecture Notes*. en. Report. INRIA, Sept. 2005, p. 105.

[RL00] M. M. G. Ricci and T. Levi-Civita. “Méthodes de calcul différentiel absolu et leurs applications”. fr. In: *Mathematische Annalen* 54.1 (Mar. 1900), pp. 125–201. DOI: 10.1007/BF01454201.

[Ril18a] Mitchell Riley. “Categories of Optics”. In: *arXiv:1809.00738 [math]* (Sept. 2018). arXiv: 1809.00738 [math].

[Ril18b] Mitchell Riley. “Categories of Optics”. In: *arXiv:1809.00738 [math]* (Sept. 2018). arXiv: 1809.00738 [math].

[Rob+19] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. *TensorNetwork: A Library for Physics and Machine Learning*. May 2019. DOI: 10.48550/arXiv.1905.01330. arXiv: 1905.01330 [cond-mat, physics:hep-th, physics:physics, stat].

[SCC13] Mehrnoosh Sadrzadeh, Stephen Clark, and Bob Coecke. “The Frobenius Anatomy of Word Meanings I: Subject and Object Relative Pronouns”. In: *Journal of Logic and Computation* 23.6 (Dec. 2013), pp. 1293–1317. DOI: 10.1093/logcom/ext044. arXiv: 1404.5278.

[SCC14] Mehrnoosh Sadrzadeh, Stephen Clark, and Bob Coecke. “The Frobenius Anatomy of Word Meanings II: Possessive Relative Pronouns”. In: *Journal of Logic and Computation* abs/1406.4690 (2014), exu027.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. “The Semantic Foundations of Concurrent Constraint Programming”. In: *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’91. New York, NY, USA: Association for Computing Machinery, Jan. 1991, pp. 333–352. DOI: 10.1145/99583.99627.

[Sav12] Yury Savateev. “Product-Free Lambek Calculus Is NP-Complete”. en. In: *Annals of Pure and Applied Logic*. The Symposium on Logical Foundations of Computer Science 2009 163.7 (July 2012), pp. 775–788. DOI: 10.1016/j.apal.2011.09.017.

[Sch90] Schröder. *Vorlesungen über die algebra der logik*. ger. Leipzig, B. G. Teubner, 1890.

[SP97] M. Schuster and K.K. Paliwal. “Bidirectional Recurrent Neural Networks”. In: *IEEE Transactions on Signal Processing* 45.11 (Nov. 1997), pp. 2673–2681. DOI: 10.1109/78.650093.

[Sch98] Hinrich Schütze. “Automatic Word Sense Discrimination”. In: *Computational Linguistics* 24.1 (1998), pp. 97–123.

[Sha53] L. S. Shapley. “Stochastic Games”. en. In: *Proceedings of the National Academy of Sciences* 39.10 (Oct. 1953), pp. 1095–1100. DOI: 10.1073/pnas.39.10.1095.

[STS20] Dan Shiebler, Alexis Toumi, and Mehrnoosh Sadrzadeh. *Incremental Monoidal Grammars*. Jan. 2020. DOI: 10.48550/arXiv.2001.02296. arXiv: 2001.02296 [cs].

[Siv+20] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. “T|ket> : A Retargetable Compiler for NISQ Devices”. en. In: *Quantum Science and Technology* 6.1 (Nov. 2020).

[SJ07] Noah A. Smith and Mark Johnson. “Weighted and Probabilistic Context-Free Grammars Are Equally Expressive”. In: *Computational Linguistics* 33.4 (2007), pp. 477–491. DOI: 10.1162/coli.2007.33.4.477.

[Soc+13a] R. Socher, Alex Perelygin, J. Wu, Jason Chuang, Christopher D. Manning, A. Ng, and Christopher Potts. “Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank”. In: *EMNLP*. 2013.

[Soc+13b] Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng. “Reasoning with Neural Tensor Networks for Knowledge Base Completion”. In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’13. Red Hook, NY, USA: Curran Associates Inc., Dec. 2013, pp. 926–934.

[SW97] Karen Sparck Jones and Peter Willett, eds. *Readings in Information Retrieval*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[SS97] A. Sperduti and A. Starita. “Supervised Neural Networks for the Classification of Structures”. In: *IEEE Transactions on Neural Networks* 8.3 (May 1997), pp. 714–735. DOI: 10.1109/72.572108.

[Spi12] David I. Spivak. “Functorial Data Migration”. en. In: *Information and Computation* 217 (Aug. 2012), pp. 31–51. DOI: 10.1016/j.ic.2012.05.001.

[Sta04] Edward Stabler. “Varieties of Crossing Dependencies: Structure Dependence and Mild Context Sensitivity”. In: *Cognitive Science* 28 (Sept. 2004), pp. 699–720. DOI: 10.1016/j.cogsci.2004.05.002.

[Sta70] Robert C. Stalnaker. “Pragmatics”. In: *Synthese* 22.1-2 (1970), pp. 272–289. DOI: 10.1007/bf00413603.

[Sta79] Richard Statman. “The Typed λ-Calculus Is Not Elementary Recursive”. en. In: *Theoretical Computer Science* 9.1 (July 1979), pp. 73–81. DOI: 10.1016/0304-3975(79)90007-0.

[Ste87] Mark Steedman. “Combinatory Grammars and Parasitic Gaps”. en. In: *Natural Language & Linguistic Theory* 5.3 (Aug. 1987), pp. 403–439. DOI: 10.1007/BF00134555.

[Ste00] Mark Steedman. *The Syntactic Process*. Cambridge, MA, USA: MIT Press, 2000.

[Ste19] Mark Steedman. *14. Combinatory Categorial Grammar*. en. De Gruyter Mouton, May 2019. Chap. Current Approaches to Syntax, pp. 389–420.

[Str07] Lutz Straßurger. “What Is a Logic, and What Is a Proof?” en. In: *Logica Universalis*. Ed. by Jean-Yves Beziau. Basel: Birkhäuser, 2007, pp. 135–152. DOI: 10.1007/978-3-7643-8354-1_8.

[Tar36] Alfred Tarski. “The Concept of Truth in Formalized Languages”. In: *Logic, Semantics, Metamathematics*. Ed. by A. Tarski. Oxford University Press, 1936, pp. 152–278.

[Tar41] Alfred Tarski. “On the Calculus of Relations”. EN. In: *Journal of Symbolic Logic* 6.3 (Sept. 1941), pp. 73–89.

[Tar43] Alfred Tarski. “The Semantic Conception of Truth and the Foundations of Semantics”. In: *Philosophy and Phenomenological Research* 4.3 (1943), pp. 341–376. DOI: 10.2307/2102968.

[Ter12] Kazushige Terui. “Semantic Evaluation, Intersection Types and Complexity of Simply Typed Lambda Calculus”. In: *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*. Ed. by Ashish Tiwari. Vol. 15. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 323–338. DOI: 10.4230/LIPIcs.RTA.2012.323.

[Tes59] Lucien Tesniere. *Elements de Syntaxe Structurale*. Paris/FRA: Klincksiek, 1959.

[Tho13] Michaël Thomazo. “Conjunctive Query Answering Under Existential Rules - Decidability, Complexity, and Algorithms”. en. PhD thesis. Oct. 2013. URL: https://tel.archives-ouvertes.fr/tel-00925722.

[TKG03] Domonkos Tikk, László T. Kóczy, and Tamás D. Gedeon. “A Survey on Universal Approximation and Its Limits in Soft Computing Techniques”. en. In: *International Journal of Approximate Reasoning* 33.2 (June 2003), pp. 185–202. DOI: 10.1016/S0888-613X(03)00021-5.

[Tou22] Alexis Toumi. “Category Theory for Quantum Natural Language Processing”. University of Oxford, 2022.

[TdY22] Alexis Toumi, Giovanni de Felice, and Richie Yeung. *DisCoPy for the Quantum Computer Scientist*. May 2022. DOI: 10.48550/arXiv.2205.05190. arXiv: 2205.05190 [quant-ph].

[TK21] Alexis Toumi and Alex Koziell-Pipe. “Functorial Language Models”. In: *arXiv:2103.14411 [cs, math]* (Mar. 2021). arXiv: 2103.14411 [cs, math].

[TYd21] Alexis Toumi, Richie Yeung, and Giovanni de Felice. “Diagrammatic Differentiation for Quantum Machine Learning”. In: *Electronic Proceedings in Theoretical Computer Science* 343 (Sept. 2021), pp. 132–144. DOI: 10.4204/EPTCS.343.7. arXiv: 2103.07960 [quant-ph].

[TM21] Alex Townsend-Teague and Konstantinos Meichanetzidis. “Classifying Complexity with the ZX-Calculus: Jones Polynomials and Potts Partition Functions”. In: *arXiv:2103.06914 [quant-ph]* (Mar. 2021). arXiv: 2103.06914 [quant-ph].

[TN19] Rocco Tripodi and Roberto Navigli. “Game Theory Meets Embeddings: A Unified Framework for Word Sense Disambiguation”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 88–99. DOI: 10.18653/v1/D19-1009.

[Tro+17] Théo Trouillon, Christopher R. Dance, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. “Knowledge Graph Completion via Complex Tensor Factorization”. In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 4735–4772. eprint: arXiv:1702.06879.

[Tur37] A. M. Turing. “Computability and \Lambda-Definability”. In: *Journal of Symbolic Logic* 2.4 (1937), pp. 153–163. DOI: 10.2307/2268280.

[TP10] Peter D. Turney and Patrick Pantel. “From Frequency to Meaning: Vector Space Models of Semantics”. In: *Journal of Artificial Intelligence Research* 37 (Feb. 2010), pp. 141–188. DOI: 10.1613/jair.2934. arXiv: 1003.1141.

[TN05] Karl Tuyls and Ann Nowé. “Evolutionary Game Theory and Multi-Agent Reinforcement Learning”. en. In: *The Knowledge Engineering Review* 20.1 (Mar. 2005), pp. 63–90. DOI: 10.1017/S026988890500041X.

[VKS19] VákárMatthijs, KammarOhad, and StatonSam. “A Domain Theory for Statistical Probabilistic Programming”. In: *Proceedings of the ACM on Programming Languages* (Jan. 2019). DOI: 10.1145/3290349.

[van87] Johan van Benthem. “Categorial Grammar and Lambda Calculus”. en. In: *Mathematical Logic and Its Applications*. Ed. by Dimiter G. Skordev. Boston, MA: Springer US, 1987, pp. 39–60. DOI: 10.1007/978-1-4613-0897-3_4.

[van20] John van de Wetering. “ZX-Calculus for the Working Quantum Computer Scientist”. In: *arXiv:2012.13966 [quant-ph]* (Dec. 2020). arXiv: 2012.13966 [quant-ph].

[Var82] Moshe Y. Vardi. “The Complexity of Relational Query Languages (Extended Abstract)”. In: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*. STOC ’82. New York, NY, USA: Association for Computing Machinery, May 1982, pp. 137–146. DOI: 10.1145/800070.802186.

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. “Attention Is All You Need”. In: *arXiv:1706.03762 [cs]* (Dec. 2017). arXiv: 1706.03762 [cs].

[VW94] K. Vijay-Shanker and David J. Weir. “The Equivalence Of Four Extensions Of Context-Free Grammars”. In: *Mathematical Systems Theory* 27 (1994), pp. 27–511.

[Von29] J. Von Neumann. “Zur Algebra Der Funktionaloperationen Und Theorie Der Normalen Operatoren”. In: *Mathematische Annalen* 102 (1929), pp. 370–427. DOI: 10.1007/BF01782352.

[Wal89a] R. F. C. Walters. “A Note on Context-Free Languages”. en. In: *Journal of Pure and Applied Algebra* 62.2 (Dec. 1989), pp. 199–203. DOI: 10.1016/0022-4049(89)90151-5.

[Wal89b] R. F. C. Walters. “The Free Category with Products on a Multigraph”. en. In: *Journal of Pure and Applied Algebra* 62.2 (Dec. 1989), pp. 205–210. DOI: 10.1016/0022-4049(89)90152-7.

[Wan+17] Q. Wang, Z. Mao, B. Wang, and L. Guo. “Knowledge Graph Embedding: A Survey of Approaches and Applications”. In: *IEEE Transactions on Knowledge and Data Engineering* 29.12 (Dec. 2017), pp. 2724–2743. DOI: 10.1109/TKDE.2017.2754499.

[WSL19] William Yang Wang, Sameer Singh, and Jiwei Li. "Deep Adversarial Learning for NLP". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 1–5. DOI: 10.18653/v1/N19-5001.

[Wil03] Edwin Williams. *Representation Theory*. en. MIT Press, 2003.

[WZ21] Paul Wilson and Fabio Zanasi. "The Cost of Compositionality: A High-Performance Implementation of String Diagram Composition". In: *arXiv:2105.09257 [cs, math]* (May 2021). arXiv: 2105.09257 [cs, math].

[Wit53] Ludwig Wittgenstein. *Philosophical Investigations*. Oxford: Basil Blackwell, 1953.

[Wu+20] Yongji Wu, Defu Lian, Yiheng Xu, Le Wu, and Enhong Chen. "Graph Convolutional Networks with Markov Random Field Reasoning for Social Spammer Detection". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.01 (Apr. 2020), pp. 1054–1061. DOI: 10.1609/aaai.v34i01.5455.

[XHW18] Wenhan Xiong, Thien Hoang, and William Yang Wang. "DeepPath: A Reinforcement Learning Method for Knowledge Graph Reasoning". In: *arXiv:1707.06690 [cs]* (July 2018). arXiv: 1707.06690 [cs].

[Yan+15] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. "Embedding Entities and Relations for Learning and Inference in Knowledge Bases". In: *arXiv:1412.6575 [cs]* (Aug. 2015). arXiv: 1412.6575 [cs].

[YK21] Richie Yeung and Dimitri Kartsaklis. "A CCG-Based Version of the DisCoCat Framework". In: *arXiv:2105.07720 [cs, math]* (May 2021). arXiv: 2105.07720 [cs, math].

[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)*. Vancouver, Canada: Association for Computational Linguistics, July 2017, pp. 277–287. DOI: 10.18653/v1/P17-1026.

[Yos+19] Masashi Yoshikawa, Hiroshi Noji, Koji Mineshima, and Daisuke Bekki. "Automatic Generation of High Quality CCGbanks for Parser Domain Adaptation". In: *arXiv:1906.01834 [cs]* (June 2019). arXiv: 1906.01834 [cs].

[You67] Daniel H. Younger. "Recognition and Parsing of Context-Free Languages in Time N3". en. In: *Information and Control* 10.2 (Feb. 1967), pp. 189–208. DOI: 10.1016/S0019-9958(67)80007-X.

[ZC16] William Zeng and Bob Coecke. "Quantum Algorithms for Compositional Natural Language Processing". In: *Electronic Proceedings in Theoretical Computer Science* 221 (Aug. 2016), pp. 67–75. DOI: 10.4204/EPTCS.221.8. arXiv: 1608.01406.

[Zha+19] Lipeng Zhang, Peng Zhang, Xindian Ma, Shuqin Gu, Zhan Su, and Dawei Song. "A Generalized Language Model in Tensor Space". In: *arXiv:1901.11167 [cs]* (Jan. 2019). arXiv: 1901.11167 [cs].

[Zho+21] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. *Graph Neural Networks: A Review of Methods and Applications*. Oct. 2021. arXiv: 1812.08434 [cs, stat].

[Zho+17] Qingyu Zhou, Nan Yang, Furu Wei, Chuanqi Tan, Hangbo Bao, and Ming Zhou. "Neural Question Generation from Text: A Preliminary Study". In: *arXiv:1704.01792 [cs]* (Apr. 2017). arXiv: 1704.01792 [cs].