Cambridge Quantum

# λambeq — An Efficient High-Level Python Library for Quantum NLP

*by Dimitri Kartsaklis, Ian Fan, Richie Yeung, Anna Pearson, Robin Lorenz, Alexis Toumi, Giovanni de Felice, Konstantinos Meichanetzidis, Stephen Clark, Bob Coecke*

Cambridge Quantum
DIMITRI KARTSAKLIS
dimitri.kartsaklis@cambridgequantum.com
IAN FAN
ian.fan@cambridgequantum.com
RICHIE YEUNG
richie.yeung@cambridgequantum.com
ANNA PEARSON
anna.pearson@cambridgequantum.com
ROBIN LORENZ
robin.lorenz@cambridgequantum.com
ALEXIS TOUMI
alexis.toumi@cambridgequantum.com
GIOVANNI DE FELICE
giovanni.defelice@@cambridgequantum.com
KONSTANTINOS MEICHANETZIDIS
k.mei@cambridgequantum.com
STEPHEN CLARK
steve.clark@cambridgequantum.com
BOB COECKE
bob.coecke@cambridgequantum.com

Cambridge Quantum
Terrington House
13-15 Hills Road
Cambridge CB2 1NL
United Kingdom

Published by Cambridge Quantum

08 September 2021

# lambeq: An Efficient High-Level Python Library for Quantum NLP

Dimitri Kartsaklis Ian Fan Richie Yeung Anna Pearson
Robin Lorenz Alexis Toumi Giovanni de Felice
Konstantinos Meichanetzidis Stephen Clark Bob Coecke

Cambridge Quantum Computing
*17 Beaumont Street, Oxford, OX1 2NA, UK*

{dimitri.kartsaklis;ian.fan;richie.yeung;anna.pearson;
robin.lorenz;alexis.toumi;giovanni.defelice;
kmei;steve.clark;bob.coecke}@cambridgequantum.com

**Abstract**

We present `lambeq`, the first high-level Python library for Quantum Natural Language Processing (QNLP). The open-source toolkit offers a detailed hierarchy of modules and classes implementing all stages of a pipeline for converting sentences to string diagrams, tensor networks, and quantum circuits ready to be used on a quantum computer. `lambeq` supports syntactic parsing, rewriting and simplification of string diagrams, ansatz creation and manipulation, as well as a number of compositional models for preparing quantum-friendly representations of sentences, employing various degrees of syntax sensitivity. We present the generic architecture and describe the most important modules in detail, demonstrating the usage with illustrative examples. Further, we test the toolkit in practice by using it to perform a number of experiments on simple NLP tasks, implementing both classical and quantum pipelines.

## 1 Introduction

*Quantum Natural Language Processing* (QNLP) is a very young area of research, aimed at the design and implementation of NLP models that exploit certain quantum phenomena such as superposition, entanglement, and interference to perform language-related tasks on quantum hardware. The advent of the first quantum machines, known as *noisy intermediate-scale quantum* (NISQ) computers, has already allowed researchers to make the first small steps towards exploring practical QNLP, by training models and running simple NLP experiments on quantum hardware (Meichanetzidis et al., 2020; Lorenz et al., 2021). Despite the limited capabilities of the current quantum machines, this early work is important in helping us understand better the process, the technicalities, and the unique nature of this new computational paradigm. At this stage, getting more hands-on experience is crucial in closing the gap that exists between theory and practice, and eventually leading to a point where practical real-world QNLP applications will become a reality.

With that goal in mind, we introduce `lambeq`[1][2], an open-source, modular, extensible high-level Python library, which provides the necessary tools for implementing a pipeline for experimental QNLP. At a high level, the library allows the conversion of any sentence to a quantum circuit, based on a given compositional model and certain parameterisation and choices of ansätze. This first version of the

[1] Stylised '$\lambda$ambeq', pronounced "lambek". The name is a tribute to mathematician Joachim Lambek (1922-2014), whose seminal work lay at the intersection of mathematics, logic, and linguistics.

[2] `https://github.com/CQCL/lambeq`


Figure 1: Pipeline implemented by `lambeq`.

library includes three compositional models, each using syntactic information to a different degree: a bag-of-words model with no syntactic information present, a word-sequence model which respects the order of words, and a fully syntax-based model following the compositional distributional framework from Coecke et al. (2010), often dubbed DisCoCat[3]. It should be noted that `lambeq` is extensible, and in practice it can accommodate any compositional model that can encode sentences as *string diagrams*, `lambeq`'s native data structure.[4] Further compositional models are currently being developed and will be released in the near future as part of a new version of the toolkit.

In general, `lambeq` loosely follows the pipeline employed in the first small-scale NLP experiments on quantum hardware by Meichanetzidis et al. (2020) and Lorenz et al. (2021). Figure 1 shows the processing of a sentence in more detail, the main stages of which are the following:

1. Depending on the selected compositional model, a syntax tree for the sentence might be obtained by calling a statistical CCG[5] parser. `lambeq` is equipped with a detailed API that greatly simplifies this process, and is shipped with support for a state-of-the-art parser (Yoshikawa et al., 2017).

2. Internally, the parse tree is converted into a string diagram, following the theory and practice described in Yeung and Kartsaklis (2021). The string diagram can be seen as an abstract representation of the sentence reflecting the relationships between the words as defined by the compositional model of choice, independently of any implementation decisions that take place at a lower level. For storing and manipulating string diagrams, `lambeq` uses as a backend DisCoPy (de Felice et al., 2020), a specialised Python library designed for this purpose.

3. The string diagram can be simplified or otherwise transformed by the application of rewriting rules; these can be used for example to remove specific interactions between words that might be considered redundant for the task at hand, or in order to make the computation more amenable to implementation on a quantum processing unit.

4. Finally, the resulting string diagram can be converted into a concrete quantum circuit (or a tensor network in the case of a "classical" experiment[6]), based on a specific parameterisation scheme and concrete choices of ansätze. `lambeq` features an extensible class hierarchy containing a selection of pre-defined ansätze, appropriate for both classical and quantum experiments.

After Step 4, the output of the pipeline (quantum circuit or tensor network) is ready to be used for training. In the case of a fully quantum pipeline, the quantum circuit will be processed by a quantum compiler and subsequently uploaded onto a quantum computer, while in the classical case the tensor network will be passed to an ML or optimisation library, such as PyTorch[7] or JAX[8]. This first version

[3] DIStributional COmpositional CATegorical.

[4] String diagrams in `lambeq` are close to *tensor networks* (a popular structure in quantum physics), but richer from a representation point of view. See Section 4 for more details.

[5] Combinatory Categorial Grammar (Steedman, 2000).

[6] In this paper we loosely use the term "quantum experiment/pipeline" to refer to an experiment or pipeline that uses quantum circuits on quantum or classical hardware. This is as opposed to a "classical experiment/pipeline", which mainly involves training tensor networks on classical hardware.

[7] `https://pytorch.org`

[8] `https://github.com/google/jax`

of `lambeq` does not include any optimisation or training features of its own, which is one of our future goals (for more details see Section 8).

We demonstrate the usage of the toolkit in practice by performing classical and quantum experiments on the meaning classification dataset introduced by Lorenz et al. (2021). Specifically, we implement:

- a classical experiment, in which the model encodes sentences as tensor networks and is trained using PyTorch;
- a classical simulation of a quantum pipeline, where sentences are encoded as quantum circuits with JAX used as part of the training backend;
- a noiseless shot-based quantum simulation experiment executed on `qiskit`'s[9] Aer simulator using `tket`[10], CQC's quantum compiler.
- a noisy shot-based quantum simulation experiment, again on `qiskit`'s Aer simulator using `tket`.

The rest of the paper is structured as follows: Section 2 puts this work in context by providing a summary of related research; Section 3 explains `lambeq`'s design goals; Section 4 discusses string diagrams, `lambeq`'s method for representing sentences; Section 5 describes the two external components on which `lambeq` is based, namely the statistical parser and DisCoPy; Section 6 outlines the software architecture and discusses the main modules and classes; Section 7 describes the experiments, and discusses the results; and finally, Section 8 summarises our future goals with regard to subsequent versions of the toolkit.

## 2 Background

Bringing the promise of a substantial leap forward in computational power, quantum computing has generated rapidly growing interest in recent years, with scientists and engineers forming an active and dedicated research community. Promising results and applications in quantum computing can be found in a wide range of topics, such as cryptography (Pirandola et al., 2020), chemistry (Cao et al., 2019), and biomedicine (Cao et al., 2018). One of the areas that has attracted a lot of attention is quantum machine learning (Schuld and Petruccione, 2018), with some work on quantum neural networks; see for example (Gupta and Zia, 2001; Beer et al., 2020).

On language-related tasks, Bausch et al. (2021) employ Grover search to achieve superpolynomial speedups for parsing, while Wiebe et al. (2019) present a Fock-space representation of language, which, together with a Harmony optimisation method, allows them to solve NLP problems on a quantum computer. In Gallego and Orus (2017), parse trees are interpreted as information coarse-graining tensor networks where it is also proposed that they can be instantiated as quantum circuits. Ramesh and Vinay (2003) provide quantum speedups for string-matching. There is also work on quantum-inspired classical models incorporating features of quantum theory, such as its inherent probabilistic nature (Basile and Tamburini, 2017) or the existence of many-body entangled states (Chen et al., 2020).

From an experimental NLP point of view, Meichanetzidis et al. (2020) provided the first proof of concept that practical QNLP is in principle possible in the NISQ era, by training a classifier on a small dataset of 16 sentences. In follow-up work, Lorenz et al. (2021) scaled up the training on datasets of 100-130 sentences, demonstrating convergence of the models and statistically significant results over a random baseline, thus conducting the first complete small-scale NLP experiments on quantum hardware.

The idea of representing language with string diagrams and monoidal categories first appeared in Coecke et al. (2010), in the context of DisCoCat – a compositional model of natural language whose

[9] `https://qiskit.org`

[10] `https://github.com/CQCL/pytket`

mathematical foundations provided a connection to quantum mechanics. In later work, this connection was exploited so that aspects of the DisCoCat model were leveraged to obtain a quadratic speedup (Zeng and Coecke, 2016). Further theoretical work (Meichanetzidis et al., 2021) lays the foundations for implementations on NISQ devices.

Finally, while `lambeq` is the first programming toolkit specifically aimed at QNLP research, there are already a few products available for generic quantum machine learning development. Examples include TensorFlow Quantum (Broughton et al., 2020), Google's library for developing hybrid quantum-classical ML models, and Pennylane (Bergholm et al., 2018), a cross-platform Python library for quantum machine learning and automatic differentiation developed by Xanadu.

# 3 Design goals

In this section we briefly discuss the main design goals of the toolkit. `lambeq` is written in Python, which due to its fast prototyping abilities, extensibility, and the support it receives from a large and dedicated community, is currently perhaps the most popular programming language in academia and scientific computing. This choice allows `lambeq` to greatly benefit from Python's open-source ecosystem and the various freely available scientific and numerical libraries such as `numpy`, `scipy`, PyTorch, JAX and many more.

The toolkit is open-source, published under the Apache 2.0 licence[11]. This allows code transparency, better communication with users and easier reporting of issues. More importantly, members of the expanding QNLP community will be able to make their own contributions and write their own extensions. By open-sourcing `lambeq`, we are aiming at active participation of QNLP researchers as part of a focused effort for extending and improving the available functionality.

One of the main design choices for the toolkit was for it to be highly *modular*. We keep the coupling between the various modules as low as possible, so that each of them could be used independently of the others, providing flexibility and extensibility. *Extensibility*, in particular, is another major design goal, and is supported by detailed object-oriented design and a flexible class hierarchy, which allows easy addition of basic components such as compositional models, ansätze, and rewrite rules.

Finally, the toolkit was designed with *interoperability* in mind. For example, adding a new parser boils down to encapsulating the appropriate calls into a single wrapper class, for which a detailed API is provided. Further, exporting trees and diagrams into JSON format is inherently supported by both `lambeq` and DisCoPy, simplifying communication with other applications. With regard to quantum hardware, DisCoPy features conversion to `tket` format, a compiler for optimising and manipulating platform-agnostic quantum circuits. For experiments on classical hardware, users have the ability to export the diagrams in Google's tensor network format[12], and use one of the PyTorch or TensorFlow backends for manipulation and training.

# 4 String diagrams

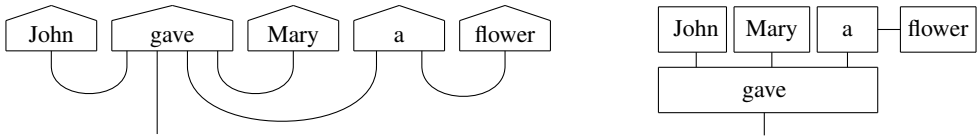
## 4.1 Motivation and connection to tensor networks

"Programming" a quantum computer requires from developers the ability to manipulate *quantum gates* (which can be seen as the "atomic" units of computation in this paradigm) in order to create quantum circuits, which can be further grouped into higher-order constructions. Working at such a low level compares to writing assembly in a classical computer, and is extremely hard for humans – especially on NLP tasks which contain many levels of abstractions.

[11] `https://www.apache.org/licenses/LICENSE-2.0`

[12] `https://github.com/google/TensorNetwork`

In order to simplify NLP design on quantum hardware, `lambeq` represents sentences as string diagrams (Figure 2a). This choice stems from the fact that a string diagram expresses computations in a *monoidal category*[13], an abstraction well-suited to model the way a quantum computer works and processes data. In fact, monoidal categories have previously been used to recast the entire framework of quantum mechanics at a higher conceptual level, forming the field that is now known as *categorical quantum mechanics* (Abramsky and Coecke, 2004). From a more practical point of view, a string diagram can be seen as an enriched *tensor network* (Orús, 2014; Pestun and Vlassopoulos, 2017) (Figure 2b), a mathematical structure with many applications in quantum physics. Compared to tensor networks, string diagrams have some additional convenient properties, for example they respect the order of words, and allow easy rewriting/modification of their structure (see for example Section 6.2).




(a) (b)

Figure 2: String diagram and corresponding tensor network.

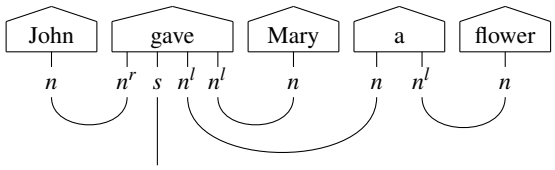
String diagrams and tensor networks constitute an ideal abstract representation of the compositional relations between the words in a sentence, in the sense that they remain close to quantum circuits, yet are independent of any low-level decisions (such as choice of quantum gates and construction of circuits representing words and sentences) that might vary depending on design choices and the type of quantum hardware that the experiment is running on. More information regarding the diagrammatic representation of quantum processes can be found in Coecke and Kissinger (2017).

## 4.2 Pregroup grammars

`lambeq`'s string diagrams are equipped with types, which show the interactions between the words in a sentence according to the *pregroup grammar* formalism (Lambek, 2008). In a pregroup grammar, each type $p$ has a left ($p^l$) and a right ($p^r$) adjoint, for which the following hold:

$$p^l \cdot p \to 1 \to p \cdot p^l \qquad\qquad p \cdot p^r \to 1 \to p^r \cdot p \tag{1}$$

When annotated with pregroup types, the diagram in Figure 2a takes the following form:




Note that each wire in the sentence is labelled with an atomic type or an adjoint. In the above, $n$ corresponds to a noun or a noun phrase, and $s$ to a sentence. The adjoints $n^r$ and $n^l$ indicate that a noun is expected on the left or the right of the specific word, respectively. Thus, the composite type $n \cdot n^l$ of the determiner "a" means that it is a word that expects a noun on its right in order to return a noun phrase.

The transition from pregroups to vector space semantics is achieved by a mapping that sends atomic types to vector spaces ($n$ to $N$ and $s$ to $S$) and composite types to tensor product spaces (e.g. $n^r \cdot s \cdot n^l \cdot n^l$ to $N \otimes S \otimes N \otimes N$) (Kartsaklis et al., 2016). Therefore, each word can be seen as a specific state in the

[13]For an introduction to monoidal categories, see Baez and Stay (2010); Coecke and Paquette (2011).

corresponding space defined by its grammatical type, i.e. a tensor, the order of which is determined by the number of wires emanating from the corresponding box. The cups (∪) denote tensor contractions. A concrete instantiation of the diagram requires the assignment of dimensions (which in the quantum case amounts to fixing the number of qubits) for each vector space corresponding to an atomic type. More details about parameterisation can be found in Section 6.3.

# 5 Main components

At the highest possible level, `lambeq` can be loosely seen as a case of *middleware*, sitting in between a statistical parser and DisCoPy, as shown in Figure 3. In practice, though, its purpose greatly exceeds the strict definition of middleware as a mere means of communication between applications, since it includes extensive functionality of its own, designed to support language-related tasks. Furthermore, DisCoPy supports the entire range of `lambeq`'s components with low-level functionality and does not merely provide an output format, as might be implied in the figure. The following sections describe in detail the role of the two main components.


Figure 3: `lambeq` as middleware.

## 5.1 CCG parser

`lambeq`'s string diagrams are based on a pregroup grammar (Section 4.2) to keep track of the types and the interactions between the words in a sentence. When a detailed syntactic derivation is required (as in the case of DisCoCat), a syntax tree needs to be provided by a statistical parser. However, since the pregroup grammar formalism is not particularly well-known in the NLP community, there is currently no wide-coverage pregroup parser that can automatically provide the syntactic derivations. To address this problem, Yeung and Kartsaklis (2021) recently provided a functorial passage from a derivation in the closest alternative grammar formalism, namely Combinatory Categorial Grammar (CCG) (Steedman, 2000), to a string diagram which faithfully encodes the syntactic structure of the sentence in a pregroup-like form. Due to the availability of many robust CCG parsing tools (for example, see Clark and Curran (2007)) this allowed the conversion of large corpora with sentences of arbitrary length and syntactic structure into pregroup and DisCoCat form, solving a long-standing problem.[14]

`lambeq` does not use its own statistical CCG parser, but instead it applies the approach of Yeung and Kartsaklis to implement a detailed interface that allows connection to one of the many external CCG parsing tools that are currently available. By default, `lambeq` is shipped with support for *DepCCG* (Yoshikawa et al., 2017), a state-of-the-art efficient parser which comes with a convenient Python interface.

## 5.2 DisCoPy

While the parser provides `lambeq`'s input, DisCoPy[15] (de Felice et al., 2020) is `lambeq`'s underlying engine, the component where all the low-level processing takes place. At its core, DisCoPy is a

[14] As an example, the DisCoCat version of the book "Alice in Wonderland" can be found at `https://qnlp.cambridgequantum.com/downloads.html`; see also Figure 12.

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

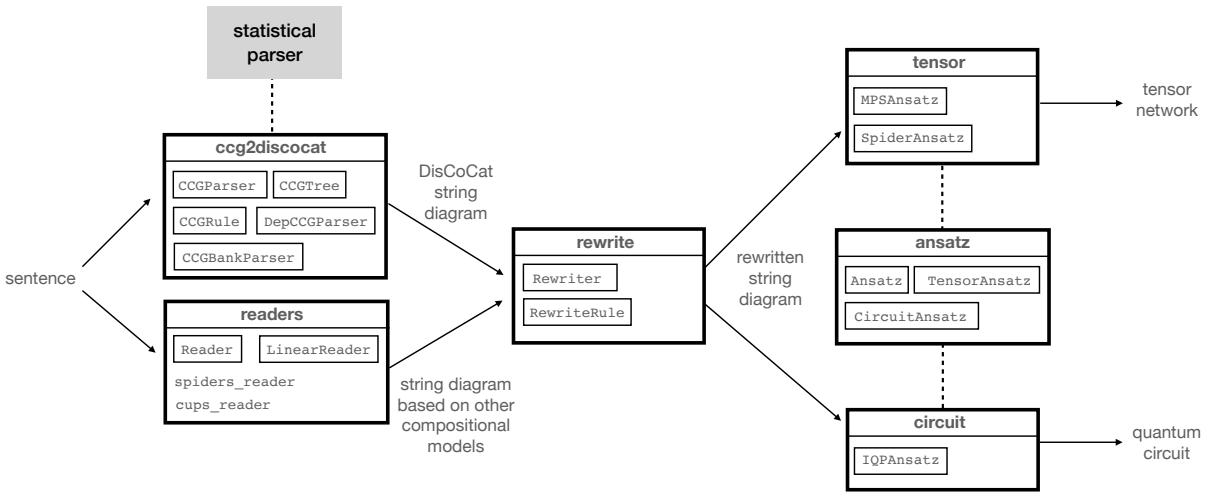



Figure 4: `lambeq`'s architecture and main modules.

Python library that allows computation with monoidal categories. The main data structure is that of a *monoidal diagram*, or string diagram, which is the format that `lambeq` uses internally to encode a sentence. DisCoPy makes this easy, by offering many language-related features, such as support for pregroup grammars and functors for implementing compositional models such as DisCoCat. Furthermore, from a quantum computing perspective, DisCoPy provides abstractions for creating all standard quantum gates and building quantum circuits, which are used by `lambeq` in the final stages of the pipeline in Figure 1.

# 6 Detailed architecture

Both the statistical parser and DisCoPy are integrated seamlessly into a unified architecture (Figure 4), implementing all stages of the generic QNLP pipeline, as was given in Figure 1. In the following sections we describe each in detail, introducing all important `lambeq` modules along the way. More details about `lambeq`'s API can be found in the toolkit's documentation,[16] along with extensive usage examples in the form of Jupyter notebooks.

## 6.1 Sentence input

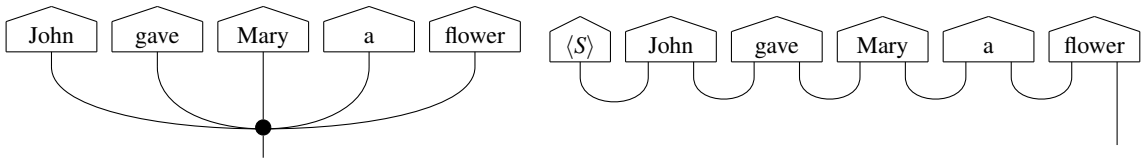
The first part of the process in `lambeq`, given a sentence, is to convert it into a string diagram. In order to obtain a DisCoCat-like output, we first use the `ccg2discocat` module, which, in turn, calls the parser, obtains a CCG derivation for the sentence, and converts it into a string diagram. Example 1 uses the default DepCCG parser in order to produce the DisCoCat diagram of Figure 2a for the sentence "John gave Mary a flower".[17] Other external parsers can be made available to `lambeq` by extending the `CCGParser` class in order to create a wrapper subclass that encapsulates the necessary calls and translates the respective parser's output into `CCGTree` format.

DisCoCat is not the only compositional model that `lambeq` supports. In fact, any compositional scheme that manifests sentences as string diagrams/tensor networks can be added to the toolkit via the module `readers`. For example, the `spiders_reader` instance of the `LinearReader` class represents

[16] `lambeq` documentation is available at `cqcl.github.io/lambeq`. DisCoPy documentation and usage examples can be found at `https://discopy.readthedocs.io/`.

[17] To see the code examples, we refer the reader to the appendix.

a sentence as a "bag-of-words", composing the words using a *spider* (a commutative operation), while `cups_reader` composes words in sequence, from left to right, generating a "tensor train" (Example 2).



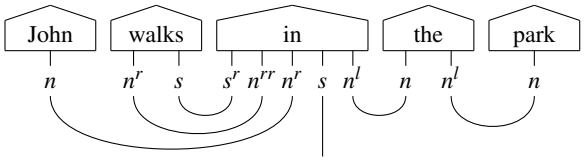

(a) `spiders_reader` (b) `cups_reader`

Writing new compositional models entails the extension of the `Reader` class and the implementation of the method `sentence2diagram` in the subclass using DisCoPy calls.

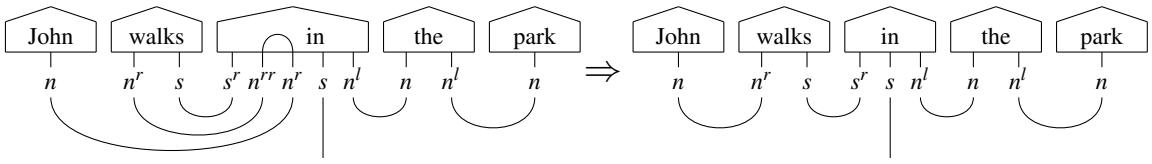
Finally, `lambeq` features a `CCGBankParser` class, which allows conversion of the entire CCGBank corpus (Hockenmaier and Steedman, 2007) into string diagrams. CCGBank consists of 49,000 human-annotated CCG syntax trees, converted from the original Penn Treebank into CCG form. Having a gold standard corpus of string diagrams allows various supervised learning scenarios involving automatic diagram generation. Example 3 shows how to convert Section 00 of CCGBank into string diagrams.

## 6.2 Rewriting

Syntactic derivations in pregroup form can become extremely complicated, which may lead to excessive use of hardware resources and prohibitively long training times. The purpose of the `rewrite` module is to provide a means to the user to address some of these problems, via rewriting rules that simplify the string diagram. As an example, consider the sentence "John walks in the park", the string diagram of which is the following:




Note that the representation of the preposition is a tensor of order 5 in the "classical" case, or a state of 5 quantum systems in the quantum case. Applying the `prepositional_phrase` rewriting rule to the diagram (Example 4) takes advantage of the underlying compact-closed monoidal structure, by using a "cap" ($\cap$) to bridge the discontinued subject noun wire within the preposition tensor:




In the simplified diagram, the order of the preposition tensor is reduced by 2, which at least for a classical experiment is a substantial improvement. This example clearly demonstrates the flexibility of string diagrams compared to simple tensor networks, which was one of the main reasons for choosing them as `lambeq`'s representation format. `lambeq` comes with a number of standard rewrite rules covering auxiliary verbs, connectors, adverbs, determiners and prepositional phrases (Table 1).

| Rule | Description |
|---|---|
| `auxiliary` | Removes auxiliary verbs (such as “do”) by replacing them with caps. |
| `connector` | Removes sentence connectors (such as “that”) by replacing them with caps. |
| `determiner` | Removes determiners (such as “the”) by replacing them with caps. |
| `postadverb`, `preadverb` | Simplifies adverbs by passing through the noun wire transparently using a cap. |
| `prepositional_phrase` | Simplifies the preposition in a prepositional phrase by passing through the noun wire transparently using a cap. |

Table 1: Rewriting rules.

### 6.3 Parameterisation

Up to this point of the pipeline, a sentence is still represented as a string diagram, independent of any low-level decisions such as tensor dimensions or specific quantum gate choices. This abstract form can be turned into a concrete quantum circuit or tensor network by applying ansätze. An *ansatz* can be seen as a map that determines choices such as the number of qubits that every wire of the string diagram is associated with and the concrete parameterised quantum states that correspond to each word. In `lambeq`, ansätze can be added by extending one of the classes `TensorAnsatz` or `CircuitAnsatz`, depending on the type of the experiment. For the quantum case, the library comes equipped with the class `IQPAnsatz`, which turns the string diagram into a standard IQP circuit[18]. For instance, the code in Example 5 produces the circuit in Figure 6 by assigning 1 qubit to the noun type and 1 qubit to the sentence type.

In the case of a classical experiment, parameterising with the `TensorAnsatz` class with $d_n = 4$ for the base dimension of the noun space, and $d_s = 2$ as the dimension of the sentence space, produces the following tensor network:

[18]Instantaneous Quantum Polynomial – a circuit which interleaves layers of Hadamard gates with diagonal unitaries (Havlíček et al., 2019).

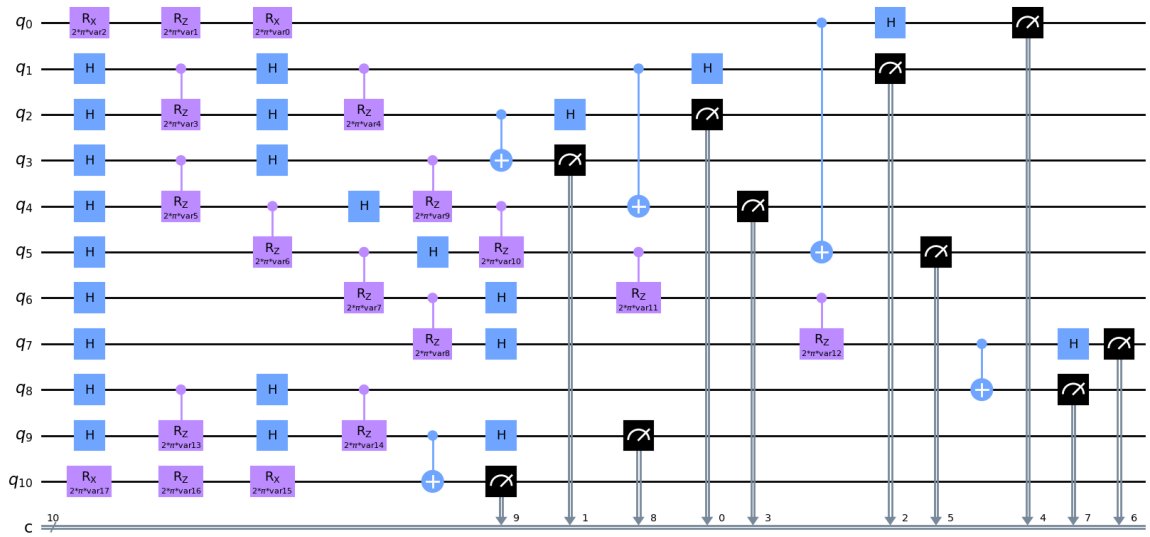
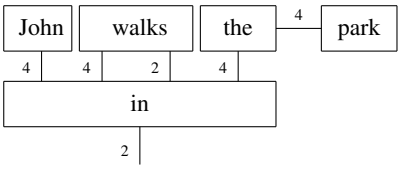
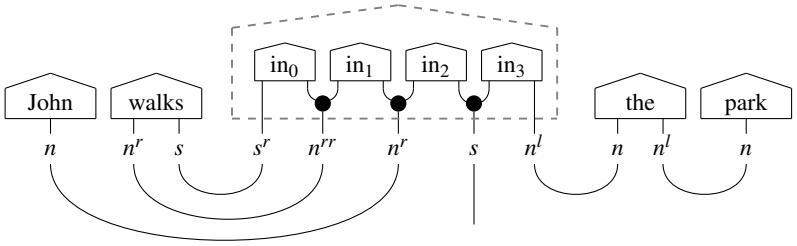


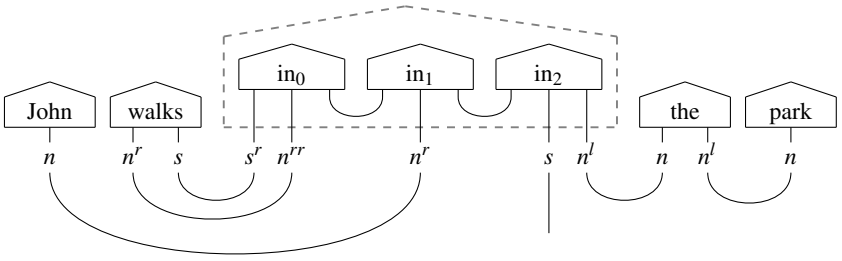

Figure 6: Quantum circuit for the sentence “John walks in the park” in `qiskit` format. Noun and sentence types have been assigned 1 qubit each.




Note that in classical experiments of this kind, the tensors associated with certain words, such as conjunctions, can become extremely large. In some cases, the order of these tensors can be 12 or even higher ($d^{12}$ elements, where $d$ is the base dimension), which makes efficient execution of the experiment impossible. In order to address this problem, `lambeq` includes ansätze for converting tensors into various forms of *matrix product states* (MPSs). Figure 7 shows the output of Example 6, where the original tensors have been split into smaller tensors linked with spiders (`SpidersAnsatz`) (a) or standard contractions (`MPSAnsatz`) (b).




(a) `SpiderAnsatz` (maximum tensor order = 2).




(b) `MPSAnsatz` (maximum tensor order = 3).

Figure 7: String diagrams using matrix product states.

### 6.4 Optimisation/training

Although `lambeq` does not yet include a native machine learning module, it seamlessly collaborates with standard ML and optimisation libraries such as PyTorch and JAX. The documentation of the toolkit includes Jupyter notebooks that demonstrate a variety of classical and quantum pipelines (see also Section 7).

## 7 lambeq in practice

In this section we apply `lambeq` to a simple sentence classification task, demonstrating its usage in four different scenarios: A classical pipeline, where the sentences are encoded as tensor networks; a classical simulation of a quantum pipeline, where sentences are encoded as quantum circuits but the optimisation and training takes place on classical hardware; a shot-based simulation of a quantum pipeline, *without*

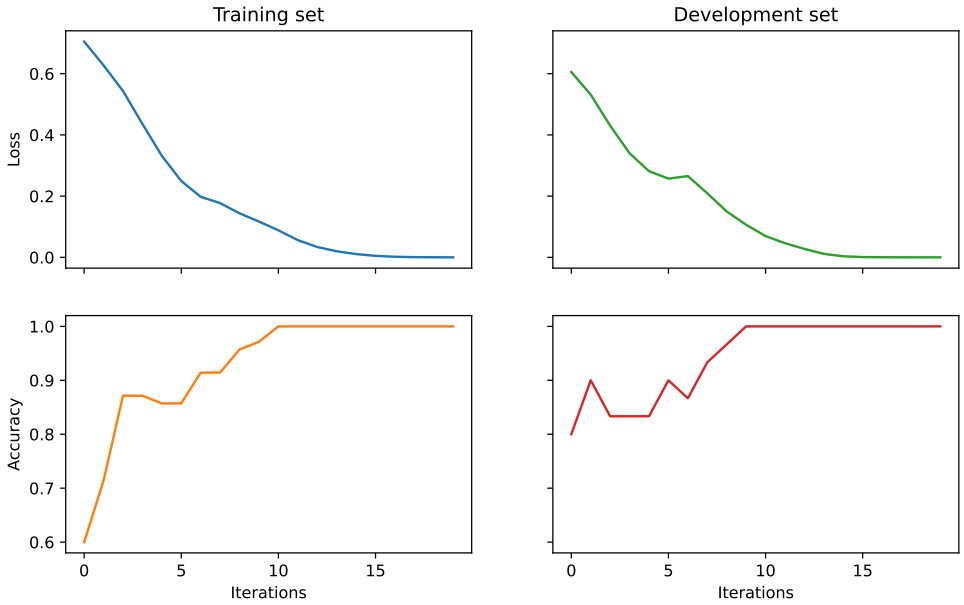



Figure 8: Classical pipeline results.

using a noise model, running on `qiskit`'s Aer simulator; and a noisy shot-based simulation of a quantum pipeline, again on Aer simulator. The code for the experiments can be found in the `examples` folder of the toolkit's documentation in the form of Jupyter notebooks.

For our experiments we use the meaning classification dataset from Lorenz et al. (2021), in which the goal is to classify simple sentences (such as "skillful programmer creates software" and "chef prepares delicious meal") into two categories, food or IT. The dataset consists of 130 sentences created using a simple context-free grammar. We proceed to provide details about the parameterisation and results of the four experiments.

**Classical pipeline** The first step in all the experiments is to convert the sentences into string diagrams with the `ccg2discocat` module. For the classical pipeline, we chose to apply `SpiderAnsatz` (Figure 7a), assigning a dimensionality of 2 to both the noun and sentence spaces. We use the PyTorch backend of Google's tensor network to perform the training, with the Adam optimiser and standard binary cross entropy as the loss. Figure 8 gives plots for the accuracy and loss on the training and development sets. Perfect accuracy is also achieved on the test set.

**Classical simulation of quantum pipeline** In this scenario the string diagrams are converted into quantum circuits using `IQPAnsatz`, with 1 qubit assigned to both the noun and sentence spaces. The number of IQP layers were set to 1. For optimisation we use a gradient-approximation technique, known as *Simultaneous Perturbation Stochastic Approximation* (SPSA) (Spall, 1992). The reason for this choice is that in a variational quantum circuit context like here, proper backpropagation requires some form of "circuit differentiation" that would in turn have to be evaluated on a quantum computer – something very costly from a practical perspective. SPSA provides a less effective but acceptable choice for the purposes of these experiments. The main prediction functions are "just-in-time" compiled with JAX in order to improve speed. Results are shown in Figure 9. Note that this configuration requires about 4 times more iterations than the previous one in order to converge; however this is eventually achieved. Once again, we obtain perfect accuracy on the test set.

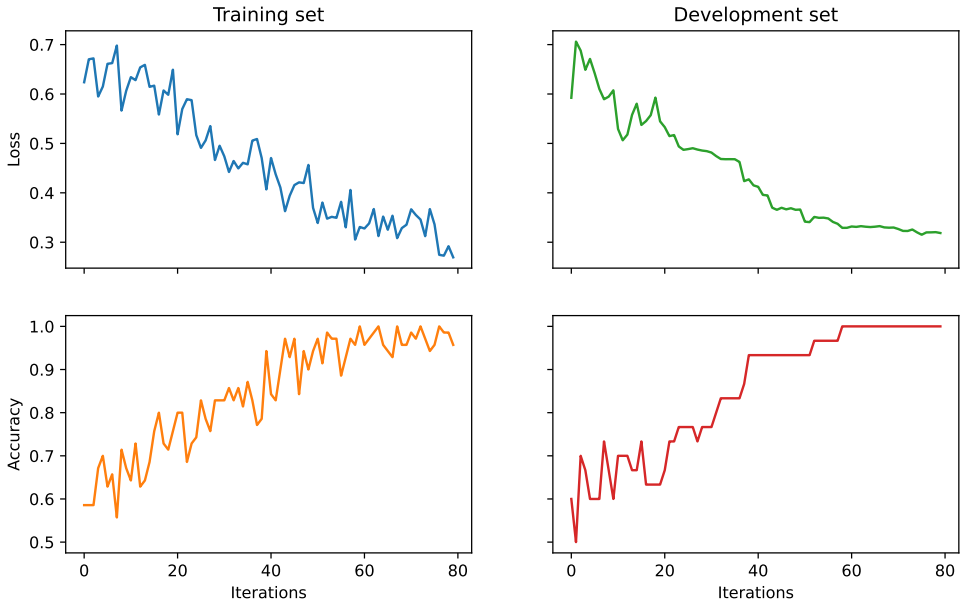



Figure 9: Results for the "classical" simulation of a quantum pipeline.

**Noiseless shot-based simulation of quantum pipeline** For this experiment we repeat the configuration of the classical simulation pipeline, but this time we evaluate the model on `qiskit`'s Aer simulator that is available through `tket`'s Python interface, `pytket`. The simulation is performed without using a noise model. SPSA is again used as the optimisation algorithm. Figure 10 shows that while there is a lot of fluctuation and instability at the early stages of training, the model eventually converges successfully. Accuracy on the test set is again perfect, although it is worth noting that performance can vary considerably between different runs.

**Noisy shot-based simulation of quantum pipeline** Finally, we use again the Aer simulator to run a noisy simulation, which is the closest we can get to an actual run on real quantum hardware. Excluding the noise model, the setting is identical with the one of the noiseless run. In Figure 11 we can see that even more iterations were required for the model to converge, due to the noisy nature of the simulation. However, perfect accuracy was achieved once more on the test set.

## 7.1 Other uses of lambeq

Below are some links to work that has previously been carried out by CQC's QNLP team using the toolkit, before this first official release:

- An early version of the toolkit has been used for the small-scale experiments on quantum hardware by Lorenz et al. (2021), implementing the pipeline of Figure 1.
- A web tool[19] based on `lambeq`'s `ccg2discocat` module was made available in the summer of 2021, allowing the conversion of any sentence into a string diagram or quantum circuit, and supporting various output formats.
- Yeung and Kartsaklis (2021) used `lambeq` in order to create the first large-scale DisCoCat resource including more than 3,000 string diagrams, parsing the book "Alice in Wonderland" into DisCoCat form[20] (Figure 12).

[19] `https://qnlp.cambridgequantum.com/generate.html`
[20] `https://qnlp.cambridgequantum.com/downloads.html`

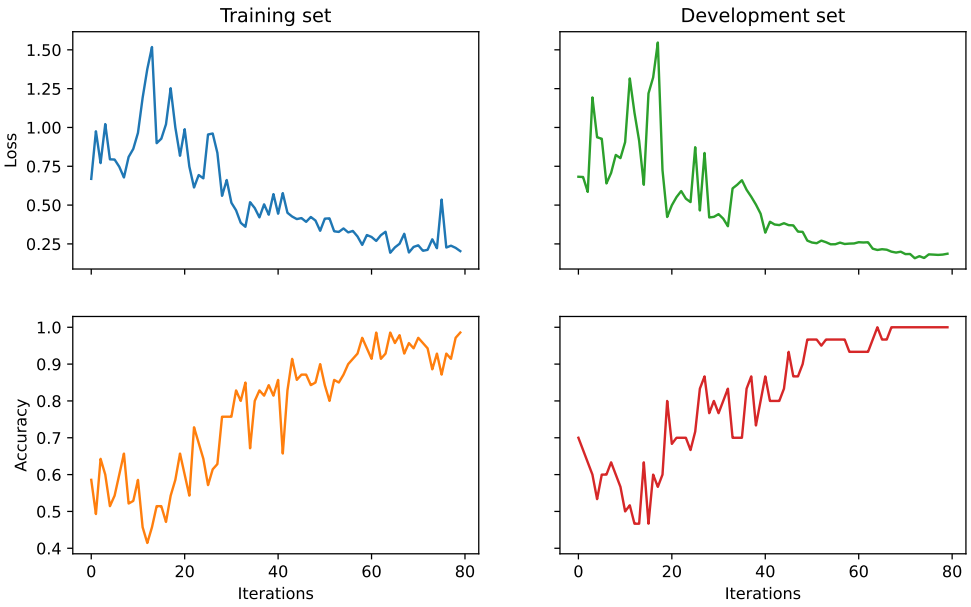



Figure 10: Results for the noiseless shot-based simulation of a quantum pipeline.

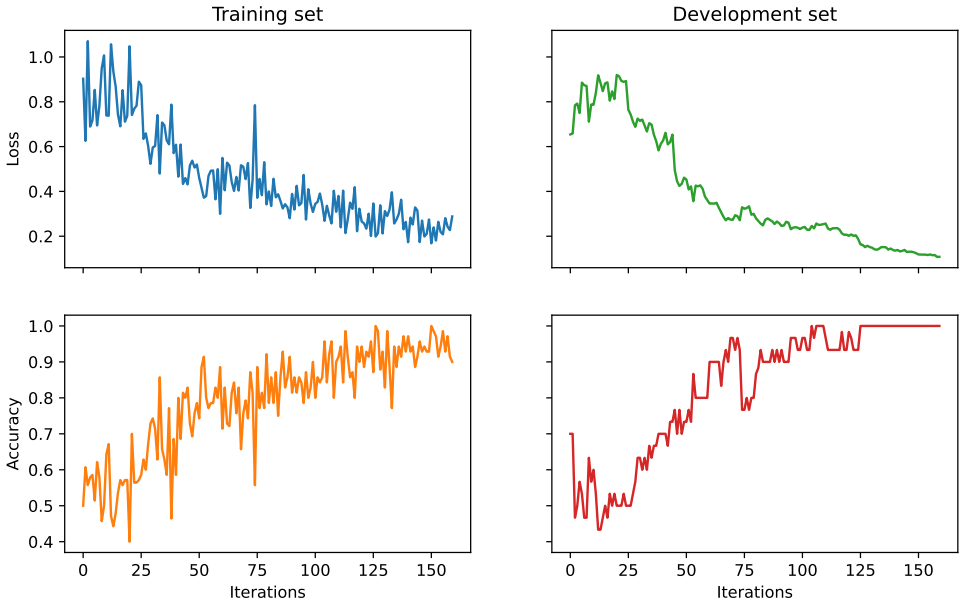



Figure 11: Results for the noisy shot-based simulation of a quantum pipeline.

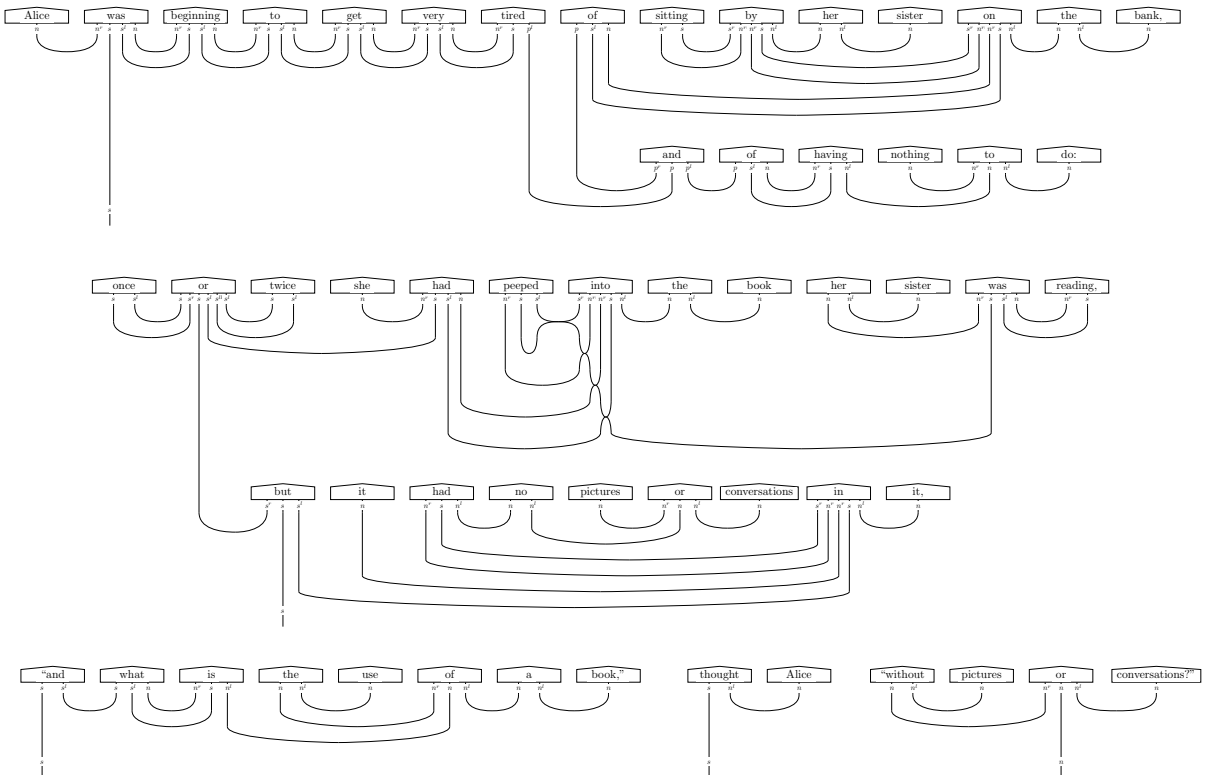



Figure 12: The first few sentences of “Alice in Wonderland” in diagrammatic form. The resource was created using `lambeq`'s `ccg2discocat` module.

# 8 Conclusions and future work

We introduced `lambeq`, the first high-level open-source Python toolkit for quantum natural language processing. This first release includes abstractions and tools for implementing all the necessary stages of a pipeline that converts sentences into quantum circuits and tensor networks. We briefly outline our plans for the future.

As mentioned before, providing more choices of compositional models and ansätze, for both quantum and classical experimental pipelines, is one of our top priorities and currently work in progress. Support for additional parsers will also be included in one of the future versions of the package. Specifically, `lambeq` will be extended with the transformer-based CCG parser described in Clark (2021), which currently significantly improves the state-of-the-art.

A more ambitious direction we plan to follow is the introduction of a “quantum-friendly” optimisation/training module, providing easy access to machine learning algorithms, objectives, and techniques suitable to quantum hardware. This is a long-term goal that is more likely to be fulfilled in stages. A priority is to integrate into `lambeq` the diagrammatic differentiation features that were recently introduced in DisCoPy (Toumi et al., 2021), taking one step closer to a fully automated quantum machine learning pipeline.

Another longer-term goal is the addition of functionality that supports discourse-related tasks at the paragraph or document level, according to the DisCoCirc model (Coecke, 2021). The idea there is to encode an entire document into a quantum circuit, where nouns are dynamically modified by the interaction with functional words, such as verbs and adjectives, as the text flows from one sentence to another.

Finally, one of the main directions of our team is the active support, maintenance and further contributions to the development of DisCoPy, `lambeq`'s low-level backend.

### Acknowledgements

We would like to thank our colleagues in Cambridge Quantum Computing for their help and support. We are especially grateful to the Oxford QNLP team for all those valuable discussions, insights and exchange of ideas during the past year. The name of the toolkit is a tribute to the great Joachim Lambek, whom Bob Coecke had the pleasure to know personally, and even live in his house for some months as a postdoc in Montreal. In 2004 Bob was presenting categorical quantum mechanics at the McGill category theory seminar; when Lambek saw quantum teleportation with diagrams, he pointed to the screen and said: "This is grammar!". And right he was!

We acknowledge the use of IBM Quantum services for this work. The views expressed are those of the authors, and do not reflect the official policy or position of IBM or the IBM Quantum team.

## References

S. Abramsky and B. Coecke. A Categorical Semantics of Quantum Protocols. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 415–425. IEEE Computer Science Press, 2004. arXiv:quant-ph/0402130.

John Baez and Mike Stay. Physics, Topology, Logic and Computation: A Rosetta Stone. In *New structures for physics*, pages 95–172. Springer, 2010.

Ivano Basile and Fabio Tamburini. Towards Quantum Language Models. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1840–1849, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1196. URL `https://www.aclweb.org/anthology/D17-1196`.

Johannes Bausch, Sathyawageeswar Subramanian, and Stephen Piddock. A Quantum Search Decoder for Natural Language Processing. *Quantum Machine Intelligence*, 3(1), Apr 2021. ISSN 2524-4914. doi: 10.1007/s42484-021-00041-1. URL `http://dx.doi.org/10.1007/s42484-021-00041-1`.

Kerstin Beer, Dmytro Bondarenko, Terry Farrelly, Tobias J. Osborne, Robert Salzmann, Daniel Scheiermann, and Ramona Wolf. Training Deep Quantum Neural Networks. *Nature Communications*, 11, 2020. ISSN 2041–1723. doi: https://doi.org/10.1038/s41467-020-14454-2.

Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, M Sohaib Alam, Shahnawaz Ahmed, Juan Miguel Arrazola, Carsten Blank, Alain Delgado, Soran Jahangiri, et al. Pennylane: Automatic Differentiation of Hybrid Quantum-Classical Computations. *arXiv preprint arXiv:1811.04968*, 2018.

Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J Martinez, Jae Hyeon Yoo, Sergei V Isakov, Philip Massey, Murphy Yuezhen Niu, Ramin Halavati, Evan Peters, et al. Tensorflow Quantum: A Software Framework for Quantum Machine Learning. *arXiv preprint arXiv:2003.02989*, 2020.

Y. Cao, J. Romero, and A. Aspuru-Guzik. Potential of Quantum Computing for Drug Discovery. *IBM Journal of Research and Development*, 62(6):6:1–6:20, 2018. doi: 10.1147/JRD.2018.2888987.

Yudong Cao, Jonathan Romero, Jonathan P. Olson, Matthias Degroote, Peter D. Johnson, Mária Kieferová, Ian D. Kivlichan, Tim Menke, Borja Peropadre, Nicolas P. D. Sawaya, Sukin Sim, Libor Veis, and Alán Aspuru-Guzik. Quantum Chemistry in the Age of Quantum Computing. *Chemical Reviews*, 119(19):10856–10915, August 2019. doi: 10.1021/acs.chemrev.8b00803. URL `https://doi.org/10.1021/acs.chemrev.8b00803`.

Yiwei Chen, Yu Pan, and Daoyi Dong. Quantum Language Model with Entanglement Embedding for Question Answering. *arXiv preprint arXiv:2008.09943*, 2020.

Stephen Clark. Something Old, Something New: Grammar-based CCG Parsing with Transformer Models. *arXiv preprint arXiv:2109.10044v1*, 2021.

Stephen Clark and James R. Curran. Wide-Coverage Efficient Statistical Parsing with CCG and Log-Linear Models. *Computational Linguistics*, 33(4):493–552, 2007. doi: 10.1162/coli.2007.33.4.493. URL `https://aclanthology.org/J07-4004`.

B. Coecke and É. O. Paquette. Categories for the Practicing Physicist. In B. Coecke, editor, *New Structures for Physics*, Lecture Notes in Physics, pages 167–271. Springer, 2011. doi: 10.1007/978-3-642-12821-9. arXiv:0905.3010.

Bob Coecke. The Mathematics of Text Structure. In *Joachim Lambek: The Interplay of Mathematics, Logic, and Linguistics*, pages 181–217. Springer, 2021.

Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017. doi: 10.1017/9781316219317.

Bob Coecke, Mehrnoosh Sadrzadeh, and Stephen Clark. Mathematical Foundations for a Compositional Distributional Model of Meaning. *Linguistic Analysis*, 36:345–384, 2010.

Giovanni de Felice, Alexis Toumi, and Bob Coecke. DisCoPy: Monoidal Categories in Python. In *Proceedings of the 3rd Annual International Applied Category Theory Conference*. EPTCS, 2020.

Angel J Gallego and Roman Orus. Language Design as Information Renormalization. *arXiv preprint arXiv:1708.01525*, 2017.

Sanjay Gupta and R.K.P. Zia. Quantum Neural Networks. *Journal of Computer and System Sciences*, 63(3):355–383, 2001. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.2001.1769. URL `https://www.sciencedirect.com/science/article/pii/S0022000001917696`.

Vojtěch Havlíček, Antonio D Córcoles, Kristan Temme, Aram W Harrow, Abhinav Kandala, Jerry M Chow, and Jay M Gambetta. Supervised Learning with Quantum-Enhanced Feature Spaces. *Nature*, 567(7747):209–212, 2019.

Julia Hockenmaier and Mark Steedman. CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396, 2007.

Dimitri Kartsaklis, Mehrnoosh Sadrzadeh, Stephen Pulman, and Bob Coecke. *Reasoning about Meaning in Natural Language with Compact Closed Categories and Frobenius Algebras*, page 199–222. Lecture Notes in Logic. Cambridge University Press, 2016. doi: 10.1017/CBO9781139519687.011.

Joachim Lambek. *From Word to Sentence*. Polimetrica, Milan, 2008.

Robin Lorenz, Anna Pearson, Konstantinos Meichanetzidis, Dimitri Kartsaklis, and Bob Coecke. QNLP in Practice: Running Compositional Models of Meaning on a Quantum Computer. *arXiv preprint arXiv:2102.12846*, 2021.

Konstantinos Meichanetzidis, Alexis Toumi, Giovanni de Felice, and Bob Coecke. Grammar-Aware Question-Answering on Quantum Computers. *arXiv preprint arXiv:2012.03756*, 2020.

Konstantinos Meichanetzidis, Stefano Gogioso, Giovanni de Felice, Nicolò Chiappori, Alexis Toumi, and Bob Coecke. Quantum Natural Language Processing on Near-term Quantum Computers. In Benoît Valiron, Shane Mansfield, Pablo Arrighi, and Prakash Panangaden, editors, Proceedings 17th International Conference on *Quantum Physics and Logic,* Paris, France, June 2 - 6, 2020, volume 340 of *Electronic Proceedings in Theoretical Computer Science*, pages 213–229. Open Publishing Association, 2021. doi: 10.4204/EPTCS.340.11.

Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

Vasily Pestun and Yiannis Vlassopoulos. Tensor Network Language Model. *arXiv preprint arXiv:1710.10248*, 2017.

S. Pirandola, U. L. Andersen, L. Banchi, M. Berta, D. Bunandar, R. Colbeck, D. Englund, T. Gehring, C. Lupo, C. Ottaviani, and et al. Advances in Quantum Cryptography. *Advances in Optics and Photonics*, 12(4):1012, Dec 2020. ISSN 1943-8206. doi: 10.1364/aop.361502. URL `http://dx.doi.org/10.1364/AOP.361502`.

H. Ramesh and V. Vinay. String Matching in O(n+m) Quantum Time. *Journal of Discrete Algorithms*, 1(1):103–110, 2003. ISSN 1570-8667. doi: https://doi.org/10.1016/S1570-8667(03)00010-8. URL `https://www.sciencedirect.com/science/article/pii/S1570866703000108`. Combinatorial Algorithms.

Maria Schuld and Francesco Petruccione. *Supervised Learning with Quantum Computers*, volume 17. Springer, 2018.

James C Spall. Multivariate Stochastic Approximation Using a Simultaneous Perturbation Gradient Approximation. *IEEE Transactions on Automatic Control*, 37(3):332–341, 1992.

Mark Steedman. *The Syntactic Process*. MIT Press, 2000.

Alexis Toumi, Richie Yeung, and Giovanni de Felice. Diagrammatic Differentiation for Quantum Machine Learning. *arXiv preprint arXiv:2103.07960*, 2021.

Nathan Wiebe, Alex Bocharov, Paul Smolensky, Matthias Troyer, and Krysta M Svore. Quantum Language Processing. *arXiv preprint arXiv:1902.05162*, 2019.

Richie Yeung and Dimitri Kartsaklis. A CCG-Based Version of the DisCoCat Framework. In *Proceedings of the 2021 Workshop on Semantic Spaces at the Intersection of NLP, Physics, and Cognitive Science (SemSpace)*, pages 20–31, Groningen, The Netherlands, June 2021. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/2021.semspace-1.3`.

Masashi Yoshikawa, Hiroshi Noji, and Yuji Matsumoto. A* CCG Parsing with a Supertag and Dependency Factored Model. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 277–287. Association for Computational Linguistics, 2017. doi: 10.18653/v1/P17-1026. URL `http://aclweb.org/anthology/P17-1026`.

William Zeng and Bob Coecke. Quantum Algorithms for Compositional Natural Language Processing. *Electronic Proceedings in Theoretical Computer Science*, 221:67–75, Aug 2016. ISSN 2075-2180. doi: 10.4204/eptcs.221.8. URL `http://dx.doi.org/10.4204/EPTCS.221.8`.

# A Appendix

*Example 1.* Using `ccg2discocat`.

```
from lambeq.ccg2discocat import DepCCGParser

# Parse the sentence and convert it into a string diagram
depccg_parser = DepCCGParser()
diagram = depccg_parser.sentence2diagram('John gave Mary a flower')

diagram.draw()
```

*Example 2.* `SpidersReader` and `CupsReader`.

```
from lambeq.reader import cups_reader, spiders_reader

sentence = 'John gave Mary a flower'

# Create string diagrams based on spiders and cups reader
spiders_diagram = spiders_reader.sentence2diagram(sentence)
cups_diagram = cups_reader.sentence2diagram(sentence)

spiders_diagram.draw()
cups_diagram.draw()
```

*Example 3.* `CCGBankParser`

```
from lambeq.ccg2discocat import CCGBankParser

# Replace path below with the root folder of CCGBank in your system.
# The sections must be located in <root>/data/AUTO
parser = CCGBankParser(root='/ccgbank')
diagrams_section_00 = parser.section2diagrams(section_id=0)
```

*Example 4.* Rewriting.

```
from lambeq.ccg2discocat import DepCCGParser
from lambeq.rewrite import Rewriter

# Parse the sentence
diagram = depccg_parser.sentence2diagram('John walks in the park')

# Apply rewrite rule for prepositional phrases
rewriter = Rewriter(['prepositional_phrase'])
rewritten_diagram = rewriter(diagram)

rewritten_diagram.draw()
```

*Example 5.* Generating a quantum circuit.

```
from lambeq.ccg2discocat import DepCCGParser
from lambeq.circuit import IQPAnsatz
from lambeq.core.types import AtomicType

# Define atomic types
N = AtomicType.NOUN
S = AtomicType.SENTENCE

# Get a string diagram
depccg_parser = DepCCGParser()
```

```
diagram = depccg_parser.sentence2diagram('John walks in the park')

# Convert string diagram to quantum circuit
ansatz = IQPAnsatz({N: 1, S: 1}, n_layers=1)
discopy_circuit = ansatz(diagram)
discopy_circuit.draw()

# Convert to tket form
tket_circuit = discopy_circuit.to_tk()
```

*Example 6.* Tensor ansätze.

```
from lambeq import ccg2discocat
from lambeq.tensor import TensorAnsatz, MPSAnsatz, SpiderAnsatz
from lambeq.core.types import AtomicType
from discopy import Dim

# Define atomic types
N = AtomicType.NOUN
S = AtomicType.SENTENCE

# Parse the sentence
depccg_parser = ccg2discocat.DepCCGParser()
diagram = depccg_parser.sentence2diagram('John walks in the park')

# Standard tensor diagram (no splitting of tensors)
std_diagram = TensorAnsatz({N: Dim(4), S: Dim(2)})(diagram)

# MPS tensor diagram
mps_diagram = MPSAnsatz({N: Dim(4), S: Dim(2)}, bond_dim=4)(diagram)

# Spiders diagram
spider_diagram = SpiderAnsatz({N: Dim(4), S: Dim(2)})(diagram)

std_diagram.draw()
mps_diagram.draw()
spider_diagram.draw()
```