Primer to Lambeq

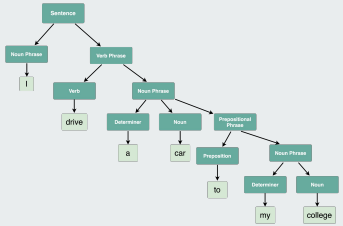
Jae Park

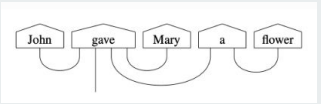
Source: https://cqcl.github.io/lambeq/index.html

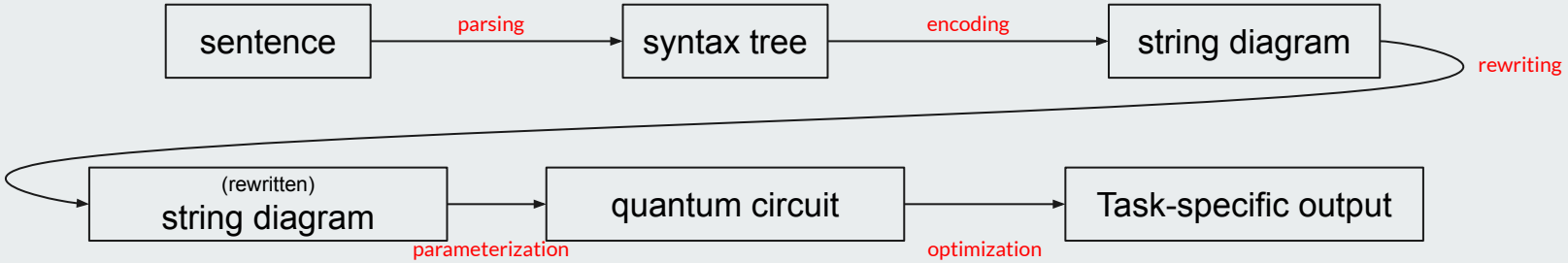
What is Lambeq?

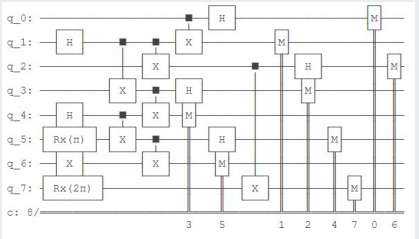
Lambeq is an open-source, modular, extensible high-level Python library for experimental Quantum Natural Language Processing (QNLP), created by Cambridge Quantum's QNLP team. At a high level, the library allows the conversion of **any sentence** to a **quantum circuit**, based on a given compositional model and certain parameterisation and choices of ansätze, and facilitates training for both quantum and classical NLP experiments.

Pipeline

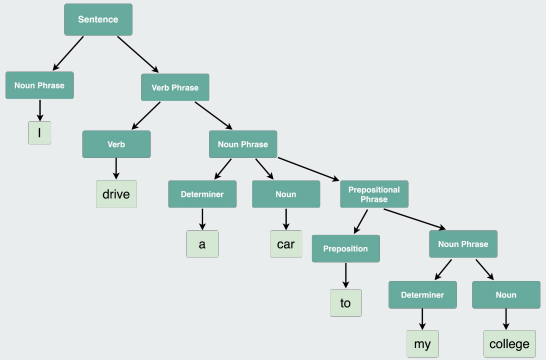

Sentence
Noun Phrase
Verb Phrase
I
Verb
Noun Phrase
drive
Determiner
Noun
Prepositional Phrase
a
car
Preposition
Noun Phrase
to
Determiner
Noun
my
college


John
gave
Mary
a
flower


sentence
parsing
syntax tree
encoding
string diagram
rewriting
(rewritten)
string diagram
parameterization
quantum circuit
optimization
Task-specific output


q_0:
q_1:
q_2:
q_3:
q_4:
q_5:
q_6:
q_7:
c: 8/
Rx(π)
Rx(2π)
3 5 1 2 4 7 0 6

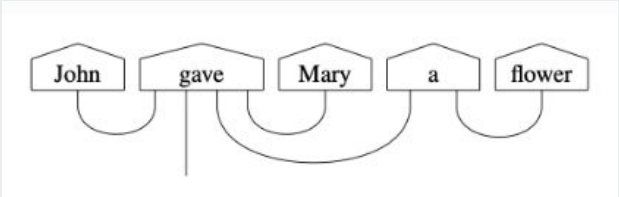
1. Sentence
2. **Syntax Tree (Parse Tree)**
3. String Diagram
4. Quantum Circuit/Tensor Net
5. Output



A syntax tree for the sentence is obtained by calling a statistical Combinatory Categorial Grammar (CCG). Lambeq is equipped with a detailed API that greatly simplifies this process and ships with support for several state-of-the-art parsers. (*Default provided*: BobCatParser)

CCG: A grammar formalism inspired by combinatory logic. It defines a number of combinators (application, composition, and type-raising being the most common) that operate on syntactically-typed lexical items, by means of natural deduction style proofs. CCG is categorised as a *mildly context-sensitive* grammar, standing in between context-free and context-sensitive hierarchy and providing a nice trade-off between expressive power and computational complexity.

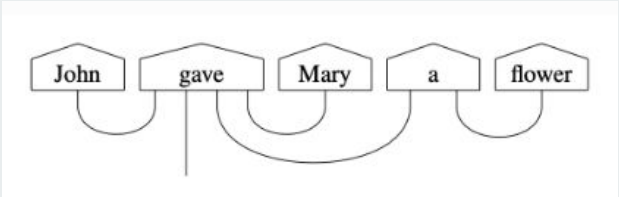
1. Sentence
2. Syntax Tree (Parse Tree)
3. **String Diagram**
4. Quantum Circuit/Tensor Net
5. Output



Internally, the syntax tree is converted into a string diagram (above). This is 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. 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.

Compositional Model: A model that produces semantic representations of sentences by composing together the semantic representations of the words within them. An example of a compositional model is DisCoCat.

1. Sentence
2. Syntax Tree (Parse Tree)
3. **String Diagram**
4. Quantum Circuit/Tensor Net
5. Output



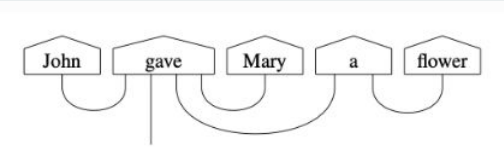
Motivation and connection to tensor network

In order to simplify NLP design on quantum hardware, lambeq represents sentences as string diagrams. This choice stems from the fact that a string diagram expresses computations in a monoidal category, an abstraction well-suited to model the way a quantum computer works and processes data.

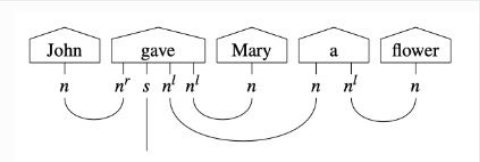
From a more practical point of view, a string diagram can be seen as an enriched tensor network. Compared to tensor networks, string diagrams have some additional convenient properties, for example, they respect the order of words, and allow easy rewriting/modification of their structure.

1. Sentence
2. Syntax Tree (Parse Tree)
3. **String Diagram**
4. Quantum Circuit/Tensor Net
5. Output

string diagram



string diagram annotated with pregroup types



Pregroup grammars

Lambeq's s string diagrams are equipped with types, which show the interactions between the words in a sentence according to the pregroup grammar. In a pregroup grammar, each type p has a left p^l and a right p^r adjoint, for which the following hold:

$$p^l \cdot p \to 1 \to p \cdot p^l \qquad p \cdot p^r \to 1 \to p^r \cdot p$$

Each wire in the sentence is labelled with an adjoint (i.e., atomic type). 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 adjoints 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 $\langle N \otimes S \otimes N \otimes N \rangle$.

Therefore, each word can be seen as a specific state in the 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.

Adjoints: used to represent arguments in composite types. For example, a transitive verb has type $n^r \cdot s \cdot n^l$, meaning it expects a noun argument on both sides in order to return a sentence.

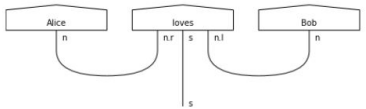
1. Sentence
2. Syntax Tree (Parse Tree)
3. String Diagram
4. **Quantum Circuit/Tensor Net**
5. Output

DisCoPy

While the parser provides lambeq's input, *DisCoPy* is lambeq's underlying engine where all the low-level processing takes place. *DisCoPy* is a 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* employs 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.

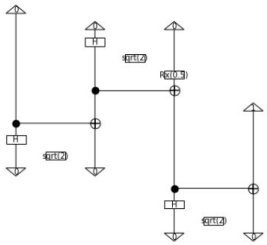
```
from discopy import Diagram, Id, Cup
from discopy.grammar import draw

grammar = Cup(n, n.r) @ Id(s) @ Cup(n.l, n)
parsing = {"{} {} {}.".format(subj, verb, obj): subj @ verb @ obj >> gram
           for subj in [Alice, Bob] for verb in [loves] for obj in [Alic

diagram = parsing['Alice loves Bob.']
print("Diagram for 'Alice loves Bob':")
draw(diagram, draw_type_labels=True)
```

```
Diagram for 'Alice loves Bob':
```




```
from discopy import CircuitFunctor, qubit

ob = {s: 0, n: 1}
ar = lambda params: {
    Alice: Ket(0), Bob: Ket(1),
    loves: verb_ansatz(params['loves'])}

F = lambda params: CircuitFunctor(ob, ar(params))

params0 = {'loves': 0.5}

print("Circuit for 'Alice loves Bob':")
F(params0)(parsing['Alice loves Bob.']).draw(
    aspect='auto', draw_type_labels=False, figsize=(5, 5))
```

```
Circuit for 'Alice loves Bob':
```



Additional In-Depth Resources

https://cqcl.github.io/lambeq/tutorials/sentence-input.html

https://discopy.readthedocs.io/en/main/notebooks/qnlp-experiment.html

https://cqcl.github.io/lambeq/bibliography.html#csc2010