Journal of the Brazilian Computer Society, 2026, 32:1, doi: 10.5753/jbcs.2026.6101
This work is licensed under a Creative Commons Attribution 4.0 International License.

Towards a Neural Lambda Calculus: Neurosymbolic AI Applied to the Foundations of Functional Programming

João Marcos Flach [**Federal University of Rio Grande do Sul** | ***jmflach@inf.ufrgs.br***]
Álvaro F. Moreira [**Federal University of Rio Grande do Sul** | ***afmoreira@inf.ufrgs.br***]
Luis C. Lamb [**Federal University of Rio Grande do Sul** | ***luislamb@acm.org***]

Institute of Informatics, Federal University of Rio Grande do Sul, Porto Alegre, 91501-970, Rio Grande do Sul, Brazil.

Received: 05 June 2025 • **Accepted:** 15 April 2026 • **Published:** 26 May 2026

Abstract Over the last decade, deep neural network models have become the dominant paradigm in machine learning. The use of artificial neural networks in symbolic learning has seen growing relevance in recent years. To study the capabilities of neural networks in the domain of symbolic AI, researchers have explored the ability of deep neural networks to learn arithmetic operations, perform logical inference, and execute computer programs. However, executing computer programs remains highly challenging for neural networks. As a result, success in this area has been limited, often requiring biased elements in the learning process and restricting the range of executable programs. In this work, we analyze the ability of neural networks to learn how to execute programs as a whole. To do so, we propose a distinct approach. Instead of using an imperative programming language with complex structures, we use the lambda calculus (λ-calculus), a simple and Turing-complete mathematical formalism. This formalism serves as the basis for modern functional programming languages and lies at the heart of computability theory since its initial definition by Alonzo Church. We introduce the an integrated neural learning and lambda-calculus formalization. We also explore the execution of a program in λ-Calculus based on reductions and show that learning to perform these reductions is enough to execute any program.

Keywords: Machine Learning, Lambda Calculus, Neurosymbolic AI, Neural Networks, Transformer Model, Sequence-to-Sequence Models, Computational Models

1 Introduction

In machine learning, a long-standing debate concerns the most effective ways to learn from data [LeCun *et al*., 2015; d'Avila Garcez *et al*., 2009; Marcus and Davis, 2019; Russell and Norvig, 2021]. One perspective, known as symbolic or rule-based models, represents knowledge explicitly through logical rules, ontologies, and compositional structures, enabling interpretable inference and systematic generalization across domains [Besold *et al*., 2022; Garcez *et al*., 2006; Russell and Norvig, 2021]. The other perspective, statistical learning, involves using mathematical or statistical models to automatically extract patterns and relationships from data [LeCun *et al*., 2015; De Raedt *et al*., 2016; Rumelhart and McClelland, 1986].

Deep neural networks have been used successfully in applications such as speech recognition, machine translation, and image classification LeCun *et al*. [2015]; Hinton [2025]. Over the last decade in particular, advances in the field have led to models, most notably large-scale transformer architectures [Brown *et al*., 2020; Vaswani *et al*., 2017] that are reshaping the technological and scientific landscape and enabling scientists to tackle an increasingly broad range of problems, including those with a natural symbolic structure. When neural networks are applied to symbolic problems, the result is a hybrid approach that combines the advantages of both rule-based and statistical approaches. This combination falls into the realm of *Neurosymbolic AI* [d'Avila Garcez *et al*., 2009; Kautz, 2022]. This field combines the statistical nature of machine learning with the logical nature of reasoning in AI [d'Avila Garcez *et al*., 2009], and has attracted the attention of academic and industry researchers [Besold *et al*., 2022; Garcez and Lamb, 2023; Gaur and Sheth, 2024].

Integrating learning and reasoning is motivated by the need to build more robust AI models [Dietterich, 2017] as advocated by Valiant and now a subject of increasing interest in the AI community [Valiant, 2003, 2018; Marcus and Davis, 2019]. There are several approaches to build such systems. Kautz [2022] presents six different forms of neurosymbolic AI, varying in how and where the two approaches are combined. In the present work, we use the form *Neuro: Symbolic* $\rightarrow$ *Neuro*, where we take a symbolic domain (reductions of the λ calculus) and apply it to a neural architecture (the transformer model).

We aim to explore the capacity of *transformer* neural networks [Vaswani *et al*., 2017] to learn rules that perform computations, a task traditionally considered challenging for neural networks. We emphasize, however, that this study represents a first step towards building a *neural lambda calculus*, rather than a comprehensive analysis of the transformer model through the lens of computational complexity and computability theory. The idea of training a machine learning model to perform formal symbolic computations is highly relevant. Most works in this field, however, restrict the class of programs that the model can take as input and consist of teaching the model to understand rules for the evaluation of mathematical expressions [Arabshahi *et al*., 2018; Lample and Charton, 2019].

In contrast, we use the *lambda calculus* (λ-Calculus) as the underlying framework [Barendregt, 1984]. Introduced by Church in the 1930s [Church, 1936], it is a simple, elegant, and Turing-complete formal system based on function abstraction that captures the notion of function definition and function application. It is the base for modern functional programming languages, such as OCaml, Racket, Haskell [Michaelson, 2011], and it has become one of the foundational computational models alongside Turing machines [Cardone and Hindley, 2006].

The λ-Calculus can be seen as a programming language whose programs, called the λ terms, can be subject to reductions that represent and are interpreted as computations performed within the formalism. Applying a single reduction to a term represents a one-step computation in the λ-calculus, while a full computation involves successive single reductions of a term until a *normal form* is obtained (if it has one), i.e., a term that cannot be reduced any further.

Our research focus is on learning to reduce lambda expressions using a neural model. Specifically, we analyze the following two hypotheses.

- *H1: The transformer model can learn to perform one-step computation on lambda calculus.*
- *H2: The transformer model can learn to perform a full computation on lambda calculus.*

Since lambda terms do not have a fixed size, we use a sequence-to-sequence (seq2seq) model, which can handle variable-length inputs and produce outputs, within practical limits. We use the transformer model which has been widely used across a range of applications, including symbolic tasks [Vaswani *et al*., 2017].

2 Related Work

Zaremba and Sutskever [2014] used seq2seq models to learn to evaluate short computer programs using an imperative language with the `Python` syntax. However, their domain is restricted to short programs that can use just some arithmetic operations, variable assignment, if-statements, and non nested loops. Researchers have also analyzed the computational expressivity of transformers under realistic constraints. Peng *et al*. [2024] elucidate certain shortcomings of the transformer architecture, such as the complexity of elementary function composition with respect to function domain sizes. Li *et al*. [2024] explain why chains of thought (CoT) allow transformer architectures to perform serial computation. Liu *et al*. [2023] demonstrated that a transformer with limited depth can implement finite-state automata, and therefore any algorithm with bounded memory.

Other studies have worked towards developing models that learn algorithms or learn to execute computer code [Kaiser and Sutskever, 2015; Graves *et al*., 2014; Trask *et al*., 2018]. However, these works are restricted to some arithmetical operations or sequence computations (copying, duplicating, sorting). Additional work concentrates on acquiring an understanding of program representation. For example, [Maddison and Tarlow, 2014] builds generative models of ”natural source code”, i.e., code written by humans and meant to be understood by other humans, while [Mou *et al*., 2014] applies neural networks to determine program equivalence. [Zhang *et al*., 2023] show that transformers can compute derivations of Horn formulae and empirically demonstrate that transformers usually learn simpler representations that capture statistical artefacts of the dataset rather than performing actual logical reasoning.

The transformer model [Vaswani *et al*., 2017] brought several key advancements and improvements compared to previous seq2seq models prevalent at that time, including improved parallelism, reduced sequential processing requirements, and the ability to handle longer sequences. These properties have contributed to the widespread adoption of the transformer model in various machine learning and AI applications.

Lample and Charton [2019] applied the transformer model to learn symbolic integration of functions, demonstrating that it could outperform existing methods in both accuracy and efficiency. This study highlights the potential of the transformer model as a valuable tool for tasks that require symbolic reasoning and mathematical operations.

The transformer model has also enabled recent developments in conversational AI technologies. A notable example is `ChatGPT` [1], which is based on the GPT-3 model [Brown *et al*., 2020]. These chatbots can answer questions across a wide range of domains [Roose, 2022] and perform basic symbolic reasoning. However, their symbolic reasoning capability is still limited, with models producing incorrect answers to straightforward questions.

In the present work, we shift from the imperative paradigm, used in all prior works, to the functional paradigm, represented by the λ-Calculus. With this, we abstract the idea of learning to compute computer programs to learn to perform reductions on λ terms.

In the sequel, we will show that to learn one-step computation, we obtain an accuracy of 88.89% on completely random terms λ and 99.73% in terms representing Boolean expressions. For the full computation task, we obtained an accuracy of 97.70% for λ terms that represent Boolean expressions. Considering a string similarity metric, most of our predicted λ terms were above 99% similar to the correct ones.

With these results in hand, we think that the change to the functional paradigm and the use of the transformer model are two promising directions for future research.

3 The Lambda Calculus: A Summary

In this section, we present an overview of the main aspects of the lambda calculus that will be used in the paper.

3.1 Syntax

We start by defining the syntax of lambda terms. In the following, x denotes a member of an infinite and countable set of variables.

[1] available at: `https://openai.com/blog/chatgpt/`

Definition 1 (λ-terms) *The set of λ-terms is defined by the following abstract grammar:*

$$M, N, \ldots ::= \quad x \mid \lambda x.M \mid M\ N$$

Every variable is a λ term; the term $\lambda x.M$ is a function with parameter x and body M; the term $M\ N$ is the application of the term M to the argument N. In a term $\lambda x.M$, the occurrences of a variable x inside M are called *bound occurrences*, and we say that x is a *bound variable*. Variables that are not *bound* are called *free variables*.

3.2 Reductions

The main reduction of the formalism is a binary relation between terms called $\beta - reduction$ that is based on the substitution operation. A substitution takes a λ-term and substitutes a variable in it with another λ-term, similar to what is done in mathematics when a function is applied to an argument.

Definition 2 (*Redex*) *A redex (reducible expression) is any subterm in the format*

$$(\lambda x.M)\ N$$

If a term has no redexes, the term is a normal form. Otherwise, the term is reducible. Now, the β-reduction relation can be defined as:

Definition 3 (One-Step) *β-reduction ($\rightarrow_\beta$) is the smallest relation on lambda terms such that*

$$\frac{M \rightarrow_\beta M'}{M\ N \rightarrow_\beta M'\ N} \qquad \frac{N \rightarrow_\beta N'}{M\ N \rightarrow_\beta M\ N'} \qquad \frac{M \rightarrow_\beta M'}{\lambda x.M \rightarrow_\beta \lambda x.M'}$$

$$(\lambda x.M)\ N \rightarrow_\beta M[x := N]$$

In the definition above $M[x := N]$ is the notation used to represent the term that results from the substitution of all the occurrences of the variable x in the term M by a term N.

The multi-step reduction $\twoheadrightarrow_\beta$ is defined as the reflexive and transitive closure of $\rightarrow_\beta$, as follows:

Definition 4 (Multi-Step) *$\twoheadrightarrow_\beta$ is the smallest relation on lambda terms such that*

$$\frac{}{M \twoheadrightarrow_\beta M} \qquad \frac{M \twoheadrightarrow_\beta N\ N \twoheadrightarrow_\beta P}{M \twoheadrightarrow_\beta P} \qquad \frac{M \rightarrow_\beta N}{M \twoheadrightarrow_\beta N}$$

We say that a term M has a normal form if there is a term N such that $M \twoheadrightarrow_\beta N$ and N is a normal form. Not every term has a normal form. An example is the term $(\lambda x.x\ x)\ (\lambda x.x\ x)$ that β-reduces to itself.

When a term have more than one redex, it is useful to have a strategy to select which redex to reduce at each computation step. The two most common evaluation strategies are: (i) *normal or leftmost order*, where the leftmost, outermost redex of a term is reduced first, and (ii) *applicative or strict order*, where the rightmost, innermost redex of a term is reduced first. An outermost redex is a redex not contained in another redex, and an innermost redex is a redex that does not contain other redexes.

Choosing the evaluation strategy is important to clearly define which redex to reduce through β-reduction. Furthermore, it is not just a matter of personal preference since there is a theorem that says that if a term M has a normal form P, then the *normal order evaluation strategy* will always reach P from M, in a finite number of β reductions.

Therefore, in this work, we always use the *normal order evaluation strategy* when making β reductions on terms to ensure that if the term has a normal form, we can reach it.

Another type of reduction in the lambda calculus is α-reduction, used for the renaming of bound variables when necessary. But since we are following the Barendregt convention [Barendregt, 1984], which states that the names of the bound variables will always be unique, we do not need to consider α-reduction in this work.

3.3 Encodings and Computations

We can use the structure of function abstractions and applications to encode representations for numbers, Booleans, pairs, lists, etc. In this work, we adopt the Church Encoding. Examples of such encoding are given below:

$$\begin{array}{lcl} \texttt{true} & \equiv & \lambda x.\lambda y.x \\ \texttt{false} & \equiv & \lambda x.\lambda y.y \\ \texttt{and} & \equiv & \lambda p.\lambda q.\ (p\ q)\ p \\ \texttt{if} & \equiv & \lambda c.\lambda t.\lambda e.\ (c\ t)\ e \end{array}$$

With these encodings we can define the boolean expression *true* $\wedge$ *false*, for instance, as the lambda term $(\texttt{and true})\ \texttt{false}$ that reduces to its normal form `false` in four small-steps as follows:

$$\begin{array}{cl} & (\texttt{and true})\ \texttt{false} \\ \equiv & ((\lambda p.\lambda q.\ (p\ q)\ p)\ \texttt{true})\ \texttt{false} \\ \rightarrow_\beta & (\lambda q.\ (\texttt{true}\ q)\ \texttt{true})\ \texttt{false} \\ \rightarrow_\beta & (\texttt{true false})\ \texttt{true} \\ \equiv & ((\lambda x.\lambda y.\ x)\ \texttt{false})\ \texttt{true} \\ \rightarrow_\beta & (\lambda y.\ \texttt{false})\ \texttt{true} \\ \rightarrow_\beta & \texttt{false} \\ \equiv & \lambda x.\lambda y.y \end{array}$$

With the lambda term `if` we can define an expression such as *if true then w else z*, for instance, as the lambda term $((\texttt{if true})\ w)\ z$, that reduces to the term w as the following reduction shows:

$$\begin{array}{cl} & ((\texttt{if true})\ w)\ z \\ \equiv & (((\lambda c.\lambda t.\lambda e.\ (c\ t)\ e)\ \texttt{true})\ w)\ z \\ \rightarrow_\beta & (\lambda t.\lambda e.\ (\texttt{true}\ t)\ e)\ w\ z \\ \rightarrow_\beta & \lambda e.\ (\texttt{true}\ w)\ e)\ z \\ \equiv & (\texttt{true}\ w)\ z \\ \equiv & ((\lambda x.\lambda y.x)\ w)\ z \\ \rightarrow_\beta & (\lambda y.w)\ z \\ \rightarrow_\beta & w \end{array}$$

The sequences of one-step reductions above are proofs that $(\texttt{and true})\ \texttt{false} \twoheadrightarrow_\beta \texttt{false}$, and that $((\texttt{if true})\ w)\ z \twoheadrightarrow_\beta w$.

3.4 Prefix Notation

For this work, we only consider λ-terms in the prefix notation for the learning tasks. To write lambda terms in the prefix notation, we need to add the application symbol (@) to the

syntax. The lambda term $N\ M$ is written as $@\ N\ M$ in the prefix notation. We also change the λ symbol for this representation using the uppercase letter "L".

We chose prefix notation because it offers a well-organized structure derived from a tree-like representation. This structure allows for a more straightforward representation of expressions, as it is unambiguous and eliminates the need for parentheses. This makes it easier to process expressions, particularly for the purposes of learning and understanding complex mathematical concepts.

Example 1 *The lambda terms* `true`, `false`, `and`, `if` *in the prefix traditional notation, are:*

$$\begin{array}{lcl} \texttt{true}_{td} & \equiv & \mathrm{L}\ x\ \mathrm{L}\ y\ x \\ \texttt{false}_{td} & \equiv & \mathrm{L}\ x\ \mathrm{L}\ y\ y \\ \texttt{and}_{td} & \equiv & \mathrm{L}\ p\ \mathrm{L}\ q\ @\ @\ p\ q\ p \\ \texttt{if}_{td} & \equiv & \mathrm{L}\ c\ \mathrm{L}\ t\ \mathrm{L}\ e\ @\ @\ c\ t\ e \end{array}$$

With these encodings and notation we have that

$$@\ @\ \texttt{and}_{td}\ \texttt{true}_{td}\ \texttt{false}_{td} \twoheadrightarrow_{\beta} \texttt{false}_{td}$$

and also that

$$@\ @\ @\ \texttt{if}_{td}\ \texttt{true}_{td}\ w\ z \twoheadrightarrow_{\beta}\ w$$

3.5 De Bruijn Indices

The De Bruijn indices are a tool for defining λ-terms without having to name the variables [de Bruijn, 1972]. This approach can be useful, since the terms are agnostic to the variable naming and are simpler in the sense that they are shorter. In practice, they simplify computations as they eliminate the need for α reductions. Basically, a De Bruijn index just replaces the variable names with natural numbers. The abstraction no longer has a variable name, and every variable occurrence is represented by a number, indicating at which abstraction it is bound. These nameless terms are called *De Bruijn terms*, and the numerical variables are called *De Bruijn indices* [Pierce, 2002].

This notation assumes that the *De Bruijn indice* for a bound variable corresponds to the number of binders (abstractions) under which the variable is. This notation can also be used in conjunction with the prefix notation. We will use it to compare with the traditional notation and see if there is any advantage in using a notation with no variable names for the tasks we are interested in.

Example 2 *The lambda terms* `true`, `false`, `and`, *and* (`and true`) `false` *in the prefix De Bruijn notation are:*

$$\begin{array}{lcl} \texttt{true}_{db} & \equiv & \mathrm{L\ L}\ 2 \\ \texttt{false}_{db} & \equiv & \mathrm{L\ L}\ 1 \\ \texttt{and}_{db} & \equiv & \mathrm{L\ L}\ @\ @\ 2\ 1\ 2 \\ (\texttt{and}_{db}\ \texttt{true}_{db})\ \texttt{false}_{db} & \equiv & @\ @\ \texttt{and}_{db}\ \texttt{true}_{db}\ \texttt{false}_{db} \end{array}$$

4 Neural Networks and Neurosymbolic AI

In this section, we briefly review concepts for the reader who is not an expert in Neural Networks or Symbolic AI, since this work is multidisciplinary. Neurosymbolic AI combines the strengths of both symbolic and connectionist AI [Besold *et al.*, 2022; Garcez and Lamb, 2023; Gaur and Sheth, 2024]. In this paper, the symbolic AI component is the lambda calculus described above. Connectionist AI, on the other hand, represents knowledge as patterns in a network of simple processing units. As in several neurosymbolic works, we aim to combine the two approaches by using a transformer neural network to learn symbolic lambda calculus computations.

The fundamental component of an artificial neural network (ANN) is the neuron, which essentially "activates" when a linear combination of its inputs exceeds a certain threshold. An ANN is a collection of interconnected neurons whose properties are determined by the arrangement of the neurons and their characteristics [LeCun *et al.*, 2015; Russell and Norvig, 2021].

The artificial neurons can be organized into layers. The input data are fed into the first layer, and the output of each neuron in a given layer is used as the input for the next layer until the final layer produces the network output. The connections between neurons are represented by weights that are updated during the training process to minimize the error between the predicted output and the actual output. Neural networks are powerful tools for solving complex real-world problems. However, traditional NNs have fixed-size inputs and outputs. This is a significant constraint, as many crucial problems are better expressed using sequences of unknown lengths, such as speech recognition and machine translation. Thus, a versatile method that can learn to translate sequences into sequences without being restricted to a specific domain would be valuable [Sutskever *et al.*, 2014].

Sequence-to-sequence (seq2seq) models emerged from this need. They are a type of deep learning model that is used for tasks that involve mapping an input sequence to an output sequence of variable length. They have traditionally been applied to various natural language processing tasks, such as machine translation, text summarization, and text generation. The assembly of seq2seq models can be done according to different model architectures such as recurrent neural networks (RNN) [Lipton *et al.*, 2015], Long short-term memory networks (LTSM) [Sutskever *et al.*, 2014], gated recurrent units (GRU) [Cho *et al.*, 2014], and the Transformer [Vaswani *et al.*, 2017]. Given that we can see the tasks we want to accomplish as analogous to machine translation tasks, we have opted to employ the sequence-to-sequence model using the Transformer.

The **Transformer** model is a type of neural network architecture that was introduced in [Vaswani *et al.*, 2017]. It is designed to handle sequential data, such as natural language, and has quickly become one of the most popular models for tasks such as natural language processing, machine translation, text classification, and question answering. The authors showed that their Transformer model could be used at scale in natural language processing and other applications. In particular, the model's architecture and attention mechanism allow the Transformer to capture long-range dependencies in the data, which is particularly useful for processing sequences of variable lengths. Another advantage of the Transformer is its parallelization capacity, which allows it to be trained efficiently on large amounts of data. The Transformer model can be trained in parallel on multiple sequences, which is not

possible with other traditional sequence-to-sequence models.

We chose the Transformer model mainly because to perform the β-reduction over lambda terms, it is necessary to substitute every occurrence of the variable in question with a term, and we believe that the self-attention mechanism can be used to "pay attention" to all these variable occurrences when performing the task.

5 Building Experiments

For each of the hypotheses outlined in the Introduction, we propose a different task for our model to learn. The hypothesis **H1** claims the model can perform a single-step computation in the λ-Calculus, that is, it can take a λ-term and transform it according to the single-step β-reduction rules of Definition 3 following a normal order strategy. We call this the One-Step Beta Reduction (OBR) task. The hypothesis **H2** is that the model can perform multiple reduction steps in the lambda calculus, taking a normalizable λ-term, i.e., a λ-term that has a normal form, and transforming it into its normal form through multiple beta reduction steps, following a normal order strategy. We call the task associated with this hypothesis the Multi-Step Beta Reduction (MBR) task.

The primary focus of our research question aligns with the second hypothesis. However, we chose to begin with a more straightforward hypothesis as a starting point. The first task is considered easier because it requires the execution of a single computational step, which is less complex than performing a full computation. This approach helps us gradually build our understanding and confidence before moving on to the more challenging second hypothesis. To support these hypotheses, we generate several datasets for each of the tasks and use them to train machine learning models. By training the models on these datasets, we will determine if the models can learn and perform the tasks associated with each hypothesis.

5.1 On Training

Since the λ-terms we are using do not have a fixed size, we need our neural model to accept inputs of varying lengths and generate outputs accordingly. We use a sequence-to-sequence model (seq2seq), which allows for inputs and outputs of different sizes. Specifically, we use the Transformer model [Vaswani *et al*., 2017]. This model has been widely used for neurosymbolic learning as illustrated in [Lample and Charton, 2019]. Regarding the hyperparameters, our preliminary experiments suggested that those used by [Lample and Charton, 2019] were adequate for our tasks. If needed, they can be adjusted during the training process. So, the set up hyperparameters are the following:

- Number of encoding layers - 6
- Number of decoding layers - 6
- Embedding layer dimension - 1024
- Number of attention heads - 8
- Optimizer - Adam [Kingma and Ba, 2014]
- Learning rate - 1×10^{-4}
- Loss function - Cross Entropy

5.1.1 Experimental Setting

The experiments were conducted on a server with the following configuration:

- CPU: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
- RAM: 32 GB (2 x 16 Gb) DDR4 @ 2667 MHz
- GPU: Quadro P6000 with 24 Gb
- OS: Ubuntu 18.04.5 LTS

Our initial goal was for each training session to run for 12 to 24 hours. Preliminary results showed that each training consisting of 50 epochs with an epoch size of $50,000$ would take between 12h to 30h to complete. So, we chose this arrangement. This configuration allows the model to process a total of 2.5×10^6 equations, which is 2.5 times the size of the dataset. With this machine, model, and configuration, we can safely have inputs with up to 250 tokens. With more than that, we end up with a memory shortage.

5.2 Lambda Sets and Datasets

To generate the datasets on which the models will train, we first generate *Lambda Sets* (LSs) containing only lambda terms. From these LSs, we generate the datasets needed for the training. We start generating the following three LSs:

- **Random Lambda Set (RLS)**: This LS is generated as randomly and as unbiased as possible. Thus, this LS can have terms that do not have a normal form, and also terms that are codifications of Boolean expressions. With the datasets generated from this LS, we want to assert that the model can learn β-reduction, regardless of whether the input terms represent meaningful codifications or have a normal form.
- **Closed Bool Lambda Set (CBLS)**: This LS has its terms representing closed Boolean expressions. Thus, all the terms in this LS have a normal form. With the datasets generated from this LS, we want to assert that the model can learn the β-reductions from meaningful codifications. In addition, these datasets are useful to validate the models trained with the RLS datasets, i.e., to validate whether the model learned from random terms can perform computations on terms that are meaningful codifications.
- **Open Bool Lambda Set (OBLS)**: The terms in this LS are codifications of open Boolean expressions, that is, with free variables in them. With the datasets generated from this LS, we want to assert that the model can learn β-reduction from terms that are meaningful codifications, even if they have free variables.

For the One-Step Beta Reduction Task (OBR), we generate datasets based on the three LSs proposed. However, for the Multi-Step Beta Reduction task, we do not utilize the Random LS to generate datasets, as the terms produced randomly may not have a normal form leading to infinite loop during the multi-step β-reduction.

In addition to the LS mentioned above, we create an additional LS for each task, which we refer to as a **Mixed Lambda Set (MLS)**. For OBR, the MLS comprises terms coming from RLS, CBLS, and OBLS in the same proportion. For the MBR,

the MLS is formed by terms from CBLS and OBLS, which are in the same proportion. With these mixed LSs, we want to assert that the model can learn from a domain that considers several kinds of terms.

From each of the four lambda sets presented above, we generate three datasets (all in the prefix notation), one for each of the following schemes for variable-naming:

- **De Bruijn**: this variable naming convention, presented in Section 3.5 is a way of representing λ-terms without naming the variables. It results in shorter terms and presents a factor that can be beneficial for our model.
- **Traditional**: Datasets following this convention are generated from the datasets in De Bruijn notation by replacing the De Bruijn indices by traditional variable names, such as a, b, c, x, y, z , etc, following alphabetical order. We utilize this convention because we want to compare the results of the learning process using the De Bruijn convention with those using the traditional convention
- **Random Vars**: This convention also uses the traditional convention for variable naming. However, for this version, we take the De Bruijn datasets and replace the De Bruijn indices by traditional variable names chosen randomly. We utilize this approach because we want to check if the way we name the variables matters for the models' accuracy.

In summary, for the OBR task, we ultimately have a total of 12 datasets, as illustrated in Figure 1, and for the MBR task, we have nine datasets, as shown in Figure 2.

All the datasets are simple text files, with each line in the format

`BETA M N`

For the OBR task, the pair of terms in `BETA M N` is interpreted as: λ term N results of a one-step reduction of λ term M following the normal order strategy. For the MBR task, `BETA M N` is interpreted as: λ term N is the normal form that results from a multi-step reduction of λ term M, following the normal order strategy.

Example 3 *The following are examples of entries in a closed bool dataset with the traditional variable naming convention, for the MBR tasks:*

$$\texttt{BETA} \quad @ @ \, \texttt{and}_{td} \, \texttt{true}_{td} \, \texttt{false}_{td} \quad \texttt{false}_{td}$$
$$\texttt{BETA} \quad @ @ \, \texttt{and}_{td} \, \texttt{true}_{td} \, \texttt{true}_{td} \quad \texttt{true}_{td}$$

Example 4 *Examples of entries in an open bool dataset with the traditional variable naming convention, for the MBR tasks:*

$$\texttt{BETA} \quad @ @ @ \, \texttt{if}_{td} \, \texttt{true}_{td} \, w \, z \quad w$$
$$\texttt{BETA} \quad @ @ @ \, \texttt{if}_{td} \, \texttt{false}_{td} \, w \, z \quad z$$

It should be noted that the same CBLS and OBLS are utilized by both the OBR and the MBR tasks, meaning that the initial λ-terms for the datasets that employ these same LS are consistent across both tasks. These datasets provide us with a broad set of test cases to evaluate the performance of our models.

Each dataset contains around one million examples (pairs of terms `BETA M N`), and we further divide each dataset into training, validation, and test sets. Adhering to the methodology outlined by [Lample and Charton, 2019], we allocate approximately ten thousand examples for both the validation and the test sets. This division of the datasets into training, validation, and test sets allows us to effectively train our models, tune their parameters, and evaluate their performance on independent data. Using a large number of examples in each dataset and following established best practices, we want to ensure that our results are robust and representative of the underlying task.

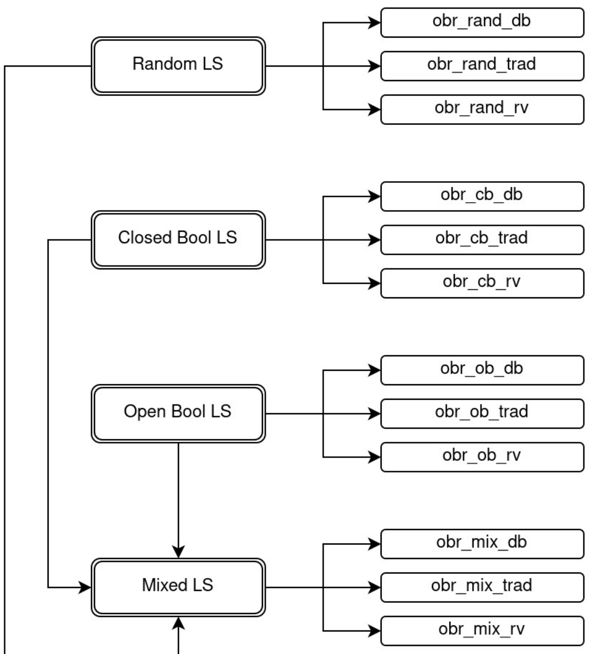

Figure 1. Scheme of how all the datasets for the OBR tasks are generated. It starts with the three Lambda Sets (RLS, CBLS, and OBLS) and ends with all 12 datasets available for the OBR task.

5.2.1 Term Sizes

As mentioned in Section 5.1.1, the maximum number of tokens that our configuration allows is 250. With this restriction, we were able to generate the term sizes listed in Table 1. Although we calculated these sizes using the datasets with terms in the traditional variable-naming convention, we expect similar results for the other datasets (with the exception of the De Bruijn convention, which should produce smaller term sizes).

Table 1. Table showing the minimum, maximum, and average sizes of the input λ-terms for each dataset. The datasets considered use traditional variable naming.

Task	Dataset	min	max	avg
OBR	random	5	249	127.2 ± 64.99
	closed bool	9	193	97.6 ± 26.76
	open bool	5	181	66.46 ± 21.73
	mixed	5	249	86.93 ± 46.56
MBR	closed bool	9	193	97.55 ± 26.75
	open bool	5	181	66.46 ± 21.72
	mixed	5	181	77.96 ± 28.02

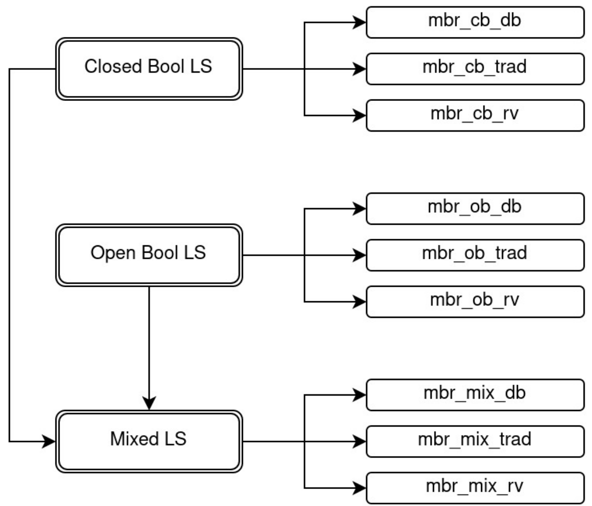

Figure 2. Scheme of how all the datasets for the MBR tasks are generated. It starts with the two Lambda Sets (CBLS, and OBLS), and ends with all nine datasets that are available for the MBR task.

5.2.2 Number of Reductions

For certain Lambda Sets, we iteratively generated the reductions until a normal form was reached for each term. Thus, another important metric we consider is the number of reductions each term had to undergo to reach its normal form. This number can be seen as the number of computational steps necessary to evaluate a given term. Table 2 shows the average number of reductions that the terms of each Lambda Set have undergone to generate their respective datasets for the OBR and MBR tasks.

Table 2. Table showing the minimum, maximum, and average number of reductions that the terms of each Lambda Set have undergone. The terms in the Mixed Lambda Set considered here come only from the closed bool and open bool Lambda Sets.

Lambda Set	min	max	avg
closed bool	3	100	18.8 ± 12.22
open bool	1	100	18.88 ± 10.42
mixed	2	100	18.82 ± 11.32

5.3 Code and Implementation

In this work, we used two distinct pieces of code. The first piece of code, adapted from [Lample and Charton, 2019], was responsible for generating what we call *intermediate* λ-terms. These intermediate lambda terms follow the syntax given by the grammar in Definition 1. This piece of code also handled the learning process.

The second piece of code is responsible for generating, from the intermediate lambda terms, the lambda terms in the formats required for the lambda sets, as described in Section 5.2. It also implements a simulator that performs reductions of these lambda terms. From the simulation results, it generates the datasets with pairs `BETA M N` where the lambda term N is the result of the lambda term M reduction.

We observe that after generating the datasets, they must first be cleaned by (i) deleting repeated pairs, and (ii) because we want all pairs to represent β-reductions, we also delete all pairs `BETA M N` with the first component M in normal form.

5.4 Accuracy and Similarity

The accuracy of the model in predicting the data in the test set is calculated and recorded after each epoch so that one can evaluate model performance. For each of the models trained, we display a graph showing the evolution of the model's accuracy (y-axis) over the epochs (x-axis). The accuracy of the model determines whether the predicted string matches the expected output. However, accuracy may not be the only relevant metric for evaluating the performance of a model in text generation or other similar tasks. In some cases, it may be useful to measure the similarity between the predicted and expected strings, even if they are not identical. So, additionally, we used a string similarity metric to compare how close the predicted string is to the expected one. For this, we used a common string similarity metric, the Levenshtein distance, which measures the number of changes (insertions, deletions, or substitutions) needed to transform one string into the other [Levenshtein, 1966]. This metric provides the absolute difference between the two strings, so we divide this distance by the maximum distance possible between the two strings (which is the length of the longer string) to generate a percentage of dissimilarity. Then, we subtract this value from 1 to get a percentage of similarity between the two strings. So, we also provide a string similarity value for each trained model. The formula used is as follows:

$$\mathsf{str_sim}\,(s_1, s_2) = \; 1 - \frac{\mathsf{lev_dist}\,(s_1, s_2)}{\mathsf{max}\,(\mathsf{len}(s_1), \mathsf{len}(s_2))}$$

As part of our analysis, we also assessed the capacity of the models trained with each dataset to evaluate the other datasets. We achieve this by performing additional evaluations with each of the already trained models. For this, we use a model trained with one dataset to evaluate the other datasets that use the same notation and are designed for the same task. For example, we take the model that was trained on the *obr_rand_trad* dataset and evaluate how it performs on the *obr_cb_trad*, *obr_ob_trad* and *obr_mix_trad* datasets with respect to accuracy and similarity.

6 Experimental Results

Some trainings experienced an oscillation in accuracy, indicating that the initial learning rate (1×10^{-4}) was too high. So, we had to adjust the learning rate for these trainings. We initially used the same value for all trainings, but decreased it based on the degree of accuracy oscillation. Table 3 shows the final learning rates for each training performed. Although the learning rate was adjusted for different trainings, we kept it consistent for the three conventions in each dataset for comparison purposes. It is important to note that we did not thoroughly search for the optimal learning rate; instead, we selected a parameter that resulted in satisfactory and converging accuracy.

Table 3. Values for the learning rate hyperparameter chosen for each of the tasks and lambda sets trained. The values started with 1×10^{-4}, and it was lowered as the trained showed an unacceptable oscillation, indicating the learning would not converge.

Task	Lambda Set	Learning Rate
One-Step Beta Reduction	random	1×10^{-4}
	closed bool	6×10^{-5}
	open bool	8×10^{-5}
	mixed	1×10^{-4}
Multi-Step Beta Reduction	closed bool	3×10^{-5}
	open bool	5×10^{-5}
	mixed	5×10^{-5}

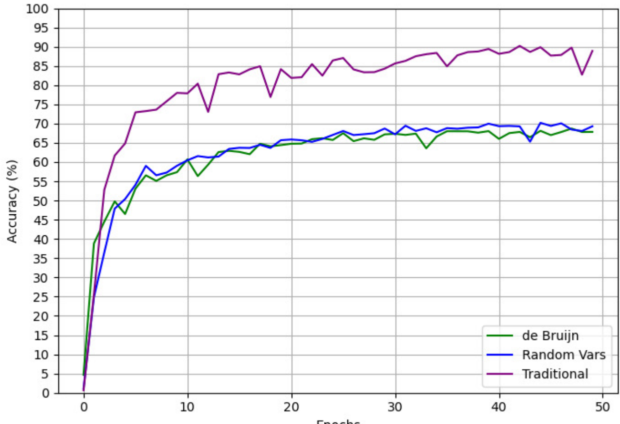

Figure 3. Graph displaying the progression for the training of the One-Step Beta Reduction task, for the random dataset, over the three different conventions.

6.1 Training Results

This section presents graphs that illustrate the training results for every model trained. The graphs display the model's accuracy for the test dataset of the respective training dataset as it evolves over the training epochs. Each graph presents the results for all three variable naming conventions utilized in this study: the traditional convention, the random vars convention, and the De Bruijn convention.

For the OBR task, the training for the random datasets can be seen in Figure 3, the closed bool datasets in Figure 4, the open bool datasets in Figure 5, and the mixed datasets in Figure 6. For the MBR task, the training for the closed bool datasets can be seen in Figure 7, the open bool datasets in Figure 8, and the mix datasets in Figure 9.

In addition to the graphs, tables 4 and 5 show the final accuracies, i.e. the accuracy of the last epoch for all the models trained for both the OBR and MBR tasks. The table also shows the average percentage of similarity of the strings, calculated using the Levenshtein distance shown in Section 5.4.

6.2 Evaluation Across Datasets

In this section, we show the results obtained by some additional evaluations with the already trained models for the OBR and MBR tasks. We use a model trained with one dataset to evaluate the other datasets that use the same convention and are designed for the same task. For example, we take the model that was trained on the *obr_cb_db* dataset and eval-

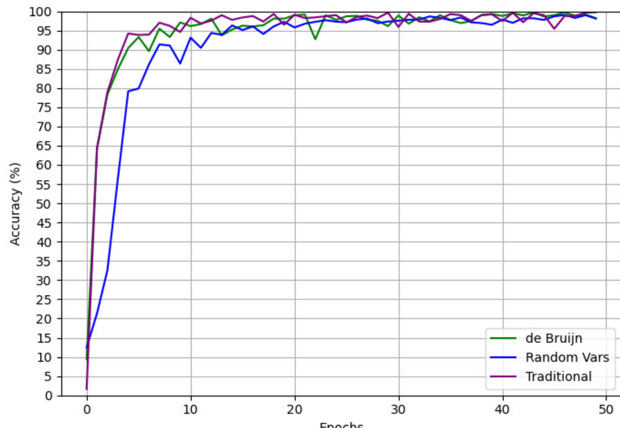

Figure 4. Graph displaying the progression for the training of the One-Step Beta Reduction task, for the closed bool dataset, over the three different conventions.

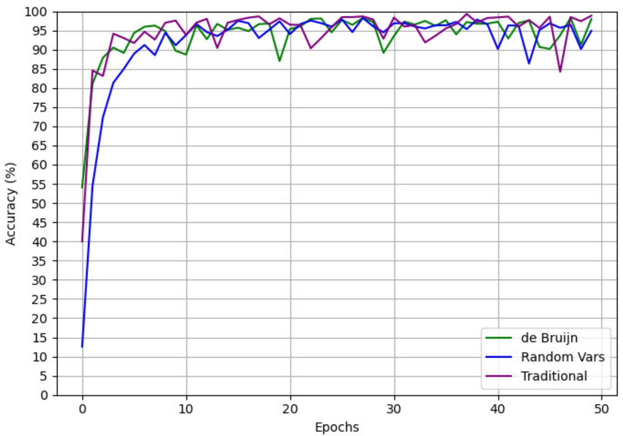

Figure 5. Graph displaying the progression for the training of the One-Step Beta Reduction task, for the open bool dataset, over the three different conventions.

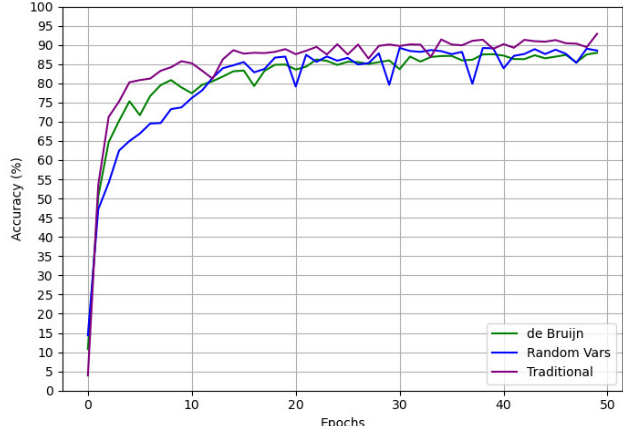

Figure 6. Graph displaying the progression for the training of the One-Step Beta Reduction task, for the mixed dataset, over the three different conventions.

Table 4. Accuracy and the average string similarity for the evaluation of the models trained for the OBR task.

Lambda Set	Convention	ACC (%)	STR SIM (%)
random	trad	88.89	99.83
	random vars	69.30	99.51
	De Bruijn	67.84	99.34
closed bool	trad	99.73	99.998
	random vars	98.10	99.98
	De Bruijn	98.16	99.99
open bool	trad	98.82	99.99
	random vars	94.88	99.95
	De Bruijn	97.94	99.97
mixed	trad	92.88	99.89
	random vars	88.52	99.77
	De Bruijn	87.93	99.73

Table 5. Accuracy and the average string similarity for the evaluation of the models trained for the MBR task.

Lambda Set	Convention	ACC (%)	STR SIM (%)
closed bool	trad	82.75	97.97
	random vars	76.08	97.06
	De Bruijn	82.20	96.49
open bool	trad	97.70	99.92
	random vars	80.92	98.19
	De Bruijn	97.02	99.77
mixed	trad	97.63	99.89
	random vars	76.58	98.15
	De Bruijn	89.99	98.64

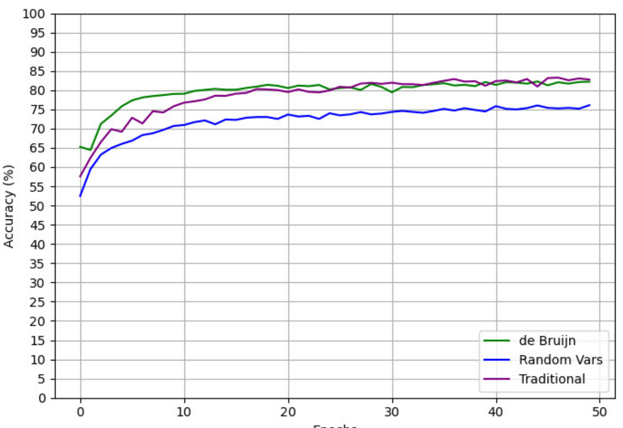

Figure 7. Graph displaying the progression for the training of the Multi-Step Beta Reduction task, for the closed bool dataset, over the three different conventions.

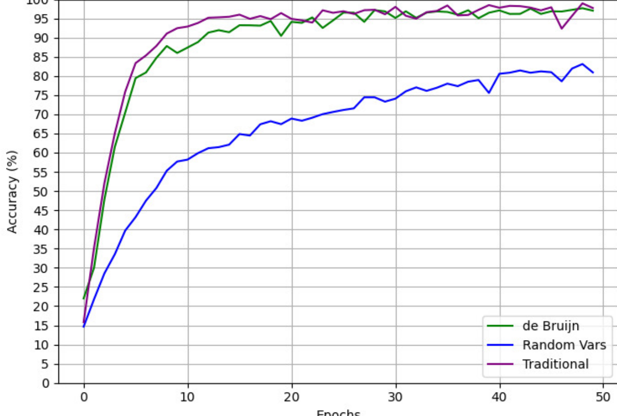

Figure 8. Graph displaying the progression for the training of the Multi-Step Beta Reduction task, for the open bool dataset, over the three different conventions.

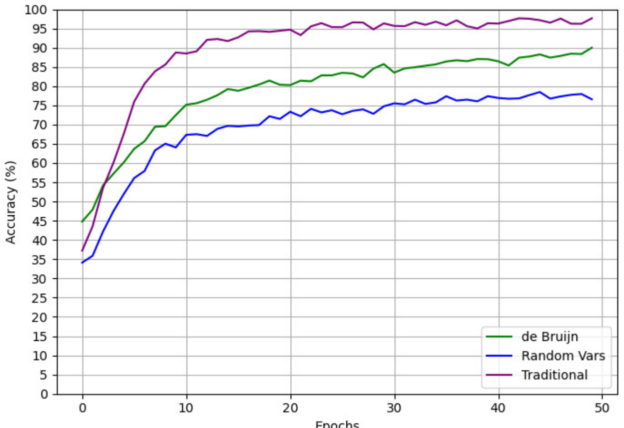

Figure 9. Graph displaying the progression for the training of the Multi-Step Beta Reduction task, for the mixed dataset, over the three different conventions.

uate how it performs on the *obr_rand_db*, *obr_ob_db* and *obr_mix_db* datasets. We use test datasets to perform these evaluations.

Tables 6 and 7 show the values found for the evaluations with the trained models for the OBR and MBR tasks, respectively.

7 Analysis of Results

In this section, we discuss the results of the experiments.

7.1 Training

Next, we discuss the results of the training presented in Section 5. From the results, we are able to determine which datasets and conventions performed better, and we seek to conjecture some hypotheses about what happened in the training.

7.1.1 One-Step Beta Reduction

In the training of the models for the OBR task, it was found that each model achieved an accuracy of at least 67.84%. However, when only the best conventions were considered, each model achieved a minimum accuracy of 88.89%. Furthermore, a model achieved remarkable accuracy 99.73%. These findings, which can be seen in Table 4, highlight the high accuracy and effectiveness of the models, particularly when using optimal conventions, which supports hypothesis H1.

The similarity metric for the strings can also be seen in Table 4 and indicates that, despite incorrectly predicting some terms, the model was able to accurately predict a significant portion of those terms, with all similarities being at least 99.34%. If we take only the best conventions, this number goes up to 99.83%. In addition, some models achieved an outstanding performance of more than 99.99% for this metric. For example, the model trained with the random dataset with the De Bruijn convention got a final accuracy of 67.84%. However, the string similarity metric for the same training was 99.34%. This illustrates how much closer the correct answers were to the incorrect answers obtained by the model.

Upon analyzing the performance of the models in different datasets, it is clear that the closed bool and open bool datasets were easier to learn compared to other datasets, as we can see in Figures 3, 4, 5 and 6. The Boolean datasets achieved good accuracies in a shorter time period and presented similar results among themselves. The random dataset was the hardest to learn, as shown in Fig. 3. We think this is due to the absence of more defined patterns among the terms. The mixed dataset, as expected, fell between random and boolean data sets in terms of difficulty to learn. However, its accuracy surprised us, since it was the most diverse dataset, which means that it learned to perform the OBR task both for random and Boolean terms, with high accuracies (92.88% for the optimal convention, as shown in Table 4).

It should be noted that, although the accuracies for the random and mixed data sets were comparatively lower than those of the Boolean datasets, the graphs illustrating their performance present a growing pattern, as illustrated in Figures 3 and 6. This indicates that further training with more epochs could yield higher accuracies for these datasets.

Analyzing the performance of the models that use different conventions in Table 4, we can observe that the traditional convention consistently outperformed the other two conventions, which exhibited similar levels of performance. However, the only training where the convention really made a difference was in the random dataset, which we saw was the hardest to learn. We suppose that for the other datasets, the difference in convention did not matter because it was so easy for the model to learn that even the "harder" conventions were not a problem.

We also think that the traditional convention performed better overall than the other two conventions. This happens because, for the De Bruijn convention–despite being based on a simpler notation–the beta reduction is more intricate, and consequently, harder to learn. Furthermore, when compared to the random-vars convention, the traditional convention tends to provide the model with more predictable outcomes.

7.1.2 Multi-Step Beta Reduction

For training of the models for the MBR task, every model achieved a minimum accuracy of 76.08%. However, when considering only the best conventions, each model exhibited an accuracy of at least 82.75%. Furthermore, one model achieved a very high 97.70% accuracy. These results, which can be seen in Table 5, emphasize the effectiveness and high accuracy of the models, especially when using the optimal conventions, which support hypothesis H2. The similarity metric for the strings, found in Table 5, again indicates that—even though the model made incorrect predictions for some terms—it accurately predicted a significant portion of those terms, with all similarities no less than 96.49%. Considering only the best conventions, this number increases to 97.97%.

Furthermore, some models performed exceptionally well, achieving up to 99.92% for this metric. Again, the models that did not obtain a good accuracy exhibited very high performance in this metric. For example, the model trained with the mixed dataset, using the random vars convention, obtained an accuracy of 76.58%. However, the string similarity metric for the same training was 98.15%. This shows that for this

Table 6. Accuracy (%) for the evaluation of the models over different datasets, for the task of one-step beta reduction. For each of the three different conventions (trad, random vars, and De Bruijn), the model trained with each dataset (rows) was evaluated with each dataset (columns). The last column indicates the average accuracy of the model over the different datasets.

Convention	Lambda Set	random	closed bool	open bool	mixed	AVERAGE
traditional	random	88.89	63.69	72.66	80.47	76.43
	closed bool	0.00	99.73	7.73	35.92	35.85
	open bool	0.04	80.01	98.82	63.79	60.67
	mixed	72.26	97.42	99.62	92.88	90.55
random vars	random	69.30	22.17	42.27	64.65	49.60
	closed bool	0.05	98.10	18.82	39.26	39.06
	open bool	0.17	77.24	94.88	61.53	58.46
	mixed	65.77	83.90	85.92	88.52	81.03
De Bruijn	random	67.84	47.15	58.35	61.14	58.62
	closed bool	0.00	98.16	10.96	36.87	36.50
	open bool	0.01	77.93	97.94	58.59	58.62
	mixed	65.70	96.39	98.71	87.93	87.18

Table 7. Accuracy (%) for the evaluation of the models over different datasets, for the task of multi-step beta reduction. For each of the three different conventions (trad, random vars, and De Bruijn), the model trained with each dataset (rows) was evaluated with each dataset (columns). The last column indicates the average accuracy of the model over the different datasets.

Convention	Lambda Set	closed bool	open bool	mixed	AVERAGE
traditional	closed bool	82.75	15.85	49.66	49.42
	open bool	92.21	97.70	96.86	95.59
	mixed	96.20	93.28	97.63	95.70
random vars	closed bool	76.08	24.79	50.98	50.62
	open bool	75.23	80.92	84.25	80.13
	mixed	72.72	50.68	76.58	66.66
De Bruijn	closed bool	82.20	20.20	57.83	53.41
	open bool	90.02	97.02	95.37	94.14
	mixed	88.43	78.64	89.99	85.69

task, the models also got the wrong predictions very close to the correct ones.

For the closed bool dataset in the MBR task, it is important to note that the set of possible terms that the model should predict is small (namely, *true* and *false*). For the traditional convention and, especially, for the random vars convention—the *true* and *false* terms are not always the same term— since there are many alpha-equivalent terms for *true* and *false* using the English alphabet. But in the DB case, there are only 2 distinct terms for the *true* and *false* ("*L L 2*" and "*L L 1*", respectively). Thus, one might expect that the closed bool dataset would be easier to learn since there are only a few possible terms for the model to predict (only two in the DB case), while the open bool, on the other hand, had output terms that differ dramatically from one another.

However, the opposite was actually observed, as we can see in Figures 7 and 8. The closed bool dataset was found to be harder to learn than the open bool dataset, with the model that was trained on it having significantly lower accuracy than the open bool model, which seems counterintuitive. Our hypothesis is that, precisely because the terms were so similar in the closed bool dataset, the model resorted to guessing the output term from a limited set of possibilities, based on some features of the inputs, instead of learning to perform the reductions. But, since this was not possible for the open bool dataset, the model was forced to actually learn to perform the multi-step beta reduction on the input terms. The fact that the closed bool model already starts the training with around 55% accuracy also corroborates our hypothesis that the model is learning to guess from a limited set instead of learning the reductions.

The model trained with the mixed dataset seems to have overcome this issue, as we can see in Figure 9. Considering the traditional convention, the accuracy of the model was similar to the accuracy of the model trained on the open bool dataset, even with half of its terms having come from the closed bool dataset. This actually supports our previous hypothesis, since we think that having more variability on the terms forced the model to learn the reductions instead of only guessing between a small set of possible outcomes.

For trainings on different conventions, Table 5 shows that the random vars convention had the worst accuracy for the three datasets. However, only the models trained on the open bool and on the mixed datasets presented a large gap between different conventions. We suppose that the naming convention did not change the guessing factor on the learning process for the models that were trained on the closed bool. What is interesting is that the De Bruijn convention led to accuracies as good as the traditional convention and significantly better than the random vars convention for the models trained on the open bool and closed bool datasets. This was unexpected since the β reduction in the De Bruijn notation is more intricate than in the traditional notation, which the other two conventions use. For the model trained on the mixed dataset, the order of the different conventions was more aligned with the expected, with the traditional being the best convention,

followed by the other two. However, this result, although expected, was unusual, as the other models did not follow this order.

7.2 Evaluations Across Datasets

In this section, we discuss the results of the evaluations in the datasets presented in Section 6.2. These evaluations can give us some insight into whether the model really learned the reductions or whether it just learned the reductions for that specific set of terms.

7.2.1 One-Step Beta Reduction

Evaluations of this task have yielded promising results, especially for models trained with the mixed dataset. It produced better average accuracies for all models in the OBR tasks, as seen in Table 6. This shows that, as expected, these models were able to better capture the diversity of terms present in the different datasets. The models trained with both Boolean datasets performed poorly for the evaluation with the random dataset, with accuracies close to 0%, as we can see in Table 6. We suppose that this happened because the terms in the random data set are very distinct from the terms in the Boolean datasets. Also, since the opposite did not happen, we think that the random dataset contains terms that are actually harder to learn, as we presumed in the previous section.

As expected, almost every model had better accuracy in the evaluation with the dataset in which it was trained than with the others, as shown in Table 6. But one result that may be seen as counterintuitive is the evaluation of the model trained with the mixed dataset. It had better accuracy for the open bool and closed bool datasets for the traditional and De Bruijn conventions. We again suppose that this happened because the random dataset is the hardest to learn. Thus, since the mixed dataset has 1/3 of its terms from the random dataset, it ends up being harder than the closed bool and open bool datasets. So, it ends up evaluating those two datasets better than the dataset it was trained with, which contains terms from the random dataset.

Another interesting result is that the model trained on the open bool dataset was able to extrapolate and obtain good accuracies for the evaluation of the closed bool dataset, with a minimum accuracy of 77.24%, but the opposite did not happen, with accuracies as low as 7.73%, as we can see in Table 6. Aside from what was mentioned, the different conventions did not present a significant difference between the evaluations.

7.2.2 Multi-Step Beta Reduction

The evaluations for this task have again yielded good results, particularly for the models trained with the open bool dataset. As shown in Table 7, the use of the open bool dataset led to better average accuracies for almost all models in the MBR task.

Table 7 shows that, as expected, the majority of models performed better in the evaluations using the dataset they were trained on, as opposed to the other datasets. However, the model trained with the open bool dataset had accuracies quite close to one another for the three datasets evaluated. In fact, it presented better accuracy for the mixed dataset than for the dataset on which it was trained. We presume that this happened because the model trained with the open bool dataset was able to generalize better than the others.

Again, the models trained with the closed bool did not extrapolate and obtained good accuracies for the other datasets, especially the closed bool, with accuracies as low as 15.85%, as seen in Table 7. But the opposite happened, with the open bool models getting a minimum of 75.23% of accuracy for the closed bool dataset. We think that this happened for the same reason discussed in Section 7.1.2, which is that the model trained on the closed bool dataset just learned to guess the output from a limited set of possible terms, not actually learning the β-reduction. Apart from what was stated, the different conventions did not present any main differences between the evaluations.

8 Conclusions and Further Work

In this work, we carried out an experimental analysis aiming at demonstrating that the Transformer model is capable of capturing the syntactic and semantic features and learning to compute the λ-calculus reductions. The results obtained were positive, with overall good accuracy for both tasks at hand. For the One-Step Beta Reduction, we obtained accuracies up to 99.73%, and string similarity metric of over 99.99%. For the Multi-Step Beta Reduction, we obtained accuracies of up to 97.70%, and string similarity metric exceeding 99.90%. In addition, the models displayed promising generalization performance across different datasets.

Due to limitations of hardware and time, our models were trained for just 50 epochs. Considering that it is a low number compared with substantial trainings of large models and that we did not aim for optimal hyperparameters, we can conjecture that the accuracy of our models can be even higher than what was presented here. Addressing the technological limitations of our work would provide a way for further advancements.

These results illustrate the effectiveness of the model in learning the desired tasks and support the two hypotheses raised in this study and, subsequently, the proposed research question. In addition, these results showed that the Transformer self-attention mechanism is well-suited for capturing the dependencies between variables and functions in the λ-Calculus.

We believe that the methods and experiments presented in this work have yielded some significant avenues for future research. The main contributions resulting from this research can be summarized as follows.

- λ-calculus learning: The outcomes from learning λ-calculus reductions are promising and hold potential implications for future research in the field of AI and computer programs.
- Dataset generation: Since datasets for λ terms and reductions did not exist, we implemented the generation of these datasets from scratch. These datasets and generation methods can be used in future research in the

domain of neurosymbolic approaches to learn and compute λ-calculi.
- Functional programming learning: The results obtained in this study can be taken into account to consider not only imperative, but also functional programming when researching in the field of learning to compute programs.

Our main contribution is the proposal of a novel approach for a neural λ-calculus, illustrated through experiments that demonstrate its potential in AI and programming languages research. Furthermore, although our research was limited to one formalism, the λ-calculus, this opens the possibility of further analyzes of other computational calculi. Future research directions include:
1. Further learning experiments: The current study trained the model for a limited number of epochs. Further research could aim to train the best notation for more epochs to see if performance can be improved.
2. Hyperparameter optimization: The study used a set of predefined hyperparameters for the transformer model. A thorough search for the optimal hyperparameters could be conducted to find the best set of hyperparameters for learning λ- calculus computations.
3. Improved error analysis: The study provided a preliminary error analysis, but more work could aim to conduct a more in-depth error analysis to better understand the types of mistakes the model is making and to identify areas for improvement.
4. Incorporating other formalisms: This study focused on learning lambda calculus, but there are other formalisms, such as Combinatorial Logic and Turing Machines, that could be trained by the model and compared with the current work.
5. Solve typing problems: Learn how to solve some typing problems (well-typedness, type assignment, type checking, and type inhabitation [Pierce, 2002]) that can be uncomputable for some typed λ-calculi.
6. Learn more complex versions of the λ-Calculus: extend the current approach to the typed λ-calculus, with numbers and arithmetical and Boolean operations already embedded.
7. Learn to compute a functional programming language: train the model on a functional programming language based on the λ-Calculus, e.g., Haskell or Lisp [Thompson, 2011].
8. Learn to detect loops: Use the same methods for training to identify whether a λ-term does has no normal form, i.e., if it will enter an infinite loop when applying the reductions.
9. Since the *typed* λ-calculus is related to intuitionistic logics via the Curry-Howard isomorphism Cardone and Hindley [2006], investigating the learnability of typed calculi and the underlying intuitionistic logic [Garcez *et al.*, 2006] is a natural sequence of this work.
10. A further promising direction is to extend the learnability analysis of Nicolau *et al*. [2025], who studied the learnability of Boolean functions on deep neural networks, to the λ-calculus setting explored here. Understanding the learnability of λ-calculus reductions from a formal theoretical perspective would complement the empirical results presented in this work.

These directions have the potential to advance the development of neurosymbolic models for integrated symbolic learning, as well as functional programming languages. In summary, the development of neural λ-calculi can contribute to a deeper understanding of the underlying computational processes in Artificial Intelligence, Machine Learning, and Computer Science.

Declarations

Funding

This research was supported in part by the CAPES Foundation (Finance Code 001) and the Brazilian Research Council CNPq.

Authors' Contributions

Luis Lamb and João Flach contributed to this work in the conceptualization of the study. João Flach performed the experiments. Álvaro Moreira contributed to the aspects of lambda calculus. All authors contributed to the writing and revision. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

The repository for the code generated during the current study is available at `https://github.com/jmflach/SymbolicLambda`

References

Arabshahi, F., Singh, S., and Anandkumar, A. (2018). Combining symbolic expressions and black-box function evaluations in neural programs. In *6th International Conference on Learning Representations, ICLR 2018*. DOI: 10.48550/arxiv.1801.04342.

Barendregt, H. (1984). *The Lambda Calculus: Its Syntax and Semantics*. North-Holland Linguistic Series. North-Holland. DOI: 10.1016/c2009-0-14341-6.

Besold, T. R., Garcez, A. d., Bader, S., Bowman, H., Domingos, P., Hitzler, P., Kühnberger, K.-U., Lamb, L. C., Lowd, D., Lima, P. M. V., *et al*. (2022). Neural-symbolic Learning and Reasoning: A survey and interpretation. *Neuro-Symbolic Artificial Intelligence: The State of the Art*. DOI: http://dx.doi.org/10.3233/FAIA210348.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., *et al*. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901. DOI: 10.48550/arxiv.2005.14165.

Cardone, F. and Hindley, J. R. (2006). History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5:723–817. Available at:`https://www.researchgate.net/publication/228386842_History_of_lambda-calculus_and_combinatory_logic`.

Cho, K., van Merrienboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. DOI: 10.3115/v1/w14-4012.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363. DOI: 10.2307/2371045.

d'Avila Garcez, A. S., Lamb, L. C., and Gabbay, D. (2009). *Neural-Symbolic Cognitive Reasoning*. Springer, Berlin. DOI: 10.1007/978-3-540-73246-4.

de Bruijn, N. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392. DOI: 10.1016/1385-7258(72)90034-0.

De Raedt, L., Kersting, K., Natarajan, S., and Poole, D. (2016). *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers. DOI: 10.2200/S00692ED1V01Y201601AIM032.

Dietterich, T. G. (2017). Steps toward robust artificial intelligence. *AI Magazine*, 38(3):3–24. DOI: 10.1609/aimag.v38i3.2756.

Garcez, A. d. and Lamb, L. C. (2023). Neurosymbolic AI: The 3rd Wave. *Artificial Intelligence Review*, pages 1–20. DOI: 10.1007/s10462-023-10448-w.

Garcez, A. S. d., Lamb, L. C., and Gabbay, D. M. (2006). Connectionist computations of intuitionistic reasoning. *Theoretical Computer Science*, 358(1):34–55. DOI: 10.1016/j.tcs.2005.11.043.

Gaur, M. and Sheth, A. P. (2024). Building trustworthy neurosymbolic AI systems: Consistency, reliability, explainability, and safety. *AI Mag.*, 45(1):139–155. DOI: 10.1002/AAAI.12149.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural Turing machines. *arXiv preprint arXiv:1410.5401*. DOI: 10.48550/arXiv.1410.5401.

Hinton, G. (2025). Nobel lecture: Boltzmann machines. *Reviews of Modern Physics*, 97(3):030502. DOI: 10.1103/RevModPhys.97.030502.

Kaiser, Ł. and Sutskever, I. (2015). Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*. DOI: 10.48550/arXiv.1511.08228.

Kautz, H. (2022). The Third AI Summer: AAAI Robert Engelmore memorial lecture. *AI Magazine*, 43(1):105–125. DOI: 10.1002/aaai.12036.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. DOI: 10.48550/arXiv.1412.6980.

Lample, G. and Charton, F. (2019). Deep learning for symbolic mathematics. *arXiv preprint arxiv:1912.01412*. DOI: 10.48550/arXiv.1912.01412.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436. DOI: 10.1038/nature14539.

Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710. Available at:`https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf`.

Li, Z., Liu, H., Zhou, D., and Ma, T. (2024). Chain of thought empowers transformers to solve inherently serial problems. In *The Twelfth International Conference on Learning Representations*. DOI: 10.48550/arXiv.2402.12875.

Lipton, Z. C., Berkowitz, J., and Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*. DOI: 10.48550/arxiv.1506.00019.

Liu, B., Ash, J. T., Goel, S., Krishnamurthy, A., and Zhang, C. (2023). Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*. DOI: 10.48550/arXiv.2210.10749.

Maddison, C. and Tarlow, D. (2014). Structured generative models of natural source code. In *International Conference on Machine Learning*, pages 649–657. PMLR. DOI: 10.48550/arxiv.1401.0514.

Marcus, G. and Davis, E. (2019). *Rebooting AI: Building artificial intelligence we can trust*. Vintage. Book.

Michaelson, G. (2011). *An Introduction to Functional Programming Through Lambda Calculus*. Dover books on mathematics. Dover Publications. Book.

Mou, L., Li, G., Liu, Y., Peng, H., Jin, Z., Xu, Y., and Zhang, L. (2014). Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358*. DOI: 10.1007/978-3-319-25159-$2_4$9.

Nicolau, M., Tavares, A. R., Zhang, Z., Avelar, P. H. C., Flach, J. M., Lamb, L. C., and Vardi, M. Y. (2025). Understanding boolean function learnability on deep neural networks: PAC learning meets neurosymbolic models. In *Proceedings of The 19th International Conference on Neurosymbolic Learning and Reasoning NeSy 2025, PMLR*, volume 284, pages 719–735. Available at:`https://proceedings.mlr.press/v284/`.

Peng, B., Narayanan, S., and Papadimitriou, C. (2024). On limitations of the transformer architecture. In *First Conference on Language Modeling*. DOI: 10.48550/arXiv.2402.08164.

Pierce, B. C. (2002). *Types and programming languages*. MIT press. Book.

Roose, K. (2022). The brilliance and weirdness of chatgpt. *The New York Times*. Available at:`https://www.nytimes.com/2022/12/05/technology/chatgpt-ai-twitter.html`.

Rumelhart, D. E. and McClelland, J. L. (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press. Book.

Russell, S. and Norvig, P. (2021). *Artificial Intelligence A Modern Approach*. DOI: 10.5555/773294.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems*, 4:3104–3112. DOI: 10.48550/arXiv.1409.3215.

Thompson, S. (2011). *Haskell: the craft of functional programming*. Addison-Wesley. Book.

Trask, A., Hill, F., Reed, S. E., Rae, J., Dyer, C., and Blunsom, P. (2018). Neural arithmetic logic units. *Advances in neural information processing systems*, 31. DOI: 10.48550/arXiv.1808.00508.

Valiant, L. (2018). What needs to be added to machine learning? In *Proceedings of ACM Turing Celebration Conference - China*, TURC '18, page 6, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3210713.3210716.

Valiant, L. G. (2003). Three problems in computer science. *J. ACM*, 50(1):96–99. DOI: 10.1145/602382.602410.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30. DOI: 10.48550/arXiv.1706.03762.

Zaremba, W. and Sutskever, I. (2014). Learning to execute. *arXiv preprint arXiv:1410.4615*. DOI: 10.48550/arXiv.1410.4615.

Zhang, H., Li, L. H., Meng, T., Chang, K.-W., and Van Den Broeck, G. (2023). On the paradox of learning to reason from data. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, IJCAI '23. DOI: 10.24963/ijcai.2023/375.