Highlights

NeuroLISP: High-level Symbolic Programming with Attractor Neural Networks

Gregory P. Davis, Garrett E. Katz, Rodolphe J. Gentili, James A. Reggia

- Gated attractor neural networks can learn to represent and interpret programs
- NeuroLISP performs symbolic AI algorithms that are readily implemented in LISP
- High-level programming constructs enrich cognitive control of neural working memory
- Program expressions can be treated as data to model reasoning about behavior

NeuroLISP: High-level Symbolic Programming with Attractor Neural Networks

Gregory P. Davis[a,*], Garrett E. Katz[b], Rodolphe J. Gentili[c], James A. Reggia[a]

[a]*Department of Computer Science, University of Maryland, College Park, MD, USA*
[b]*Department of Elec. Engr. and Comp. Sci., Syracuse University, Syracuse, NY, USA*
[c]*Department of Kinesiology, University of Maryland, College Park, MD, USA*

Abstract

Despite significant improvements in contemporary machine learning, symbolic methods currently outperform artificial neural networks on tasks that involve compositional reasoning, such as goal-directed planning and logical inference. This illustrates a computational explanatory gap between cognitive and neurocomputational algorithms that obscures the neurobiological mechanisms underlying cognition and impedes progress toward human-level artificial intelligence. Because of the strong relationship between cognition and working memory control, we suggest that the cognitive abilities of contemporary neural networks are limited by biologically-implausible working memory systems that rely on persistent activity maintenance and/or temporal nonlocality. Here we present *NeuroLISP*, an attractor neural network that can represent and execute programs written in the LISP programming language. Unlike previous approaches to high-level programming with neural networks, NeuroLISP features a temporally-local working memory based on itinerant attractor dynamics, top-down gating, and fast associative learning, and implements several high-level programming constructs such as compositional data structures, scoped variable binding, and the ability to manipulate and execute programmatic expressions in working memory (i.e., programs can be treated as data). Our computational experiments demonstrate the correctness of the NeuroLISP interpreter, and show that it can learn non-trivial programs that manipulate complex derived data structures (multiway trees), perform compositional string manipulation operations (PCFG SET task), and implement high-level symbolic AI algorithms (first-order unification). We conclude that NeuroLISP is an effective neurocognitive controller that can replace the symbolic components of hybrid models, and serves as a proof of concept for further development of high-level symbolic programming in neural networks.

Keywords: Programmable neural networks, Working memory, Symbolic processing, Cognitive control, Compositionality, Associative learning

1. Introduction

While the capabilities of artificial neural networks have improved significantly in the past several decades, implementing high-level cognitive abilities with neural computations remains a significant challenge. Many of these abilities, such as goal-directed planning and logical inference, are readily captured in symbolic algorithms, while neural networks struggle to learn the robust, generalizable procedures necessary to carry out these tasks. This is often addressed with hybrid systems that combine neural networks with symbolic programs to leverage the unique benefits of both methods (Garcez et al., 2015; Marcus, 2020; Kipf et al., 2019; Andreas et al., 2016; Sun and Naveh, 2004; Chella et al., 2008), such as neural-guided search algorithms (Bunel et al., 2018; Silver et al., 2016; Kalyan

[*]Corresponding author.
Email addresses: `grpdavis@umd.edu` (Gregory P. Davis), `gkatz01@syr.edu` (Garrett E. Katz), `rodolphe@umd.edu` (Rodolphe J. Gentili), `reggia@umd.edu` (James A. Reggia)

et al., 2018). The success of hybrid models supports the idea that neural networks lack the cognitive control provided by symbolic programming. Given that the human nervous system can reliably perform a wide array of high-level cognitive tasks, this lacuna highlights a *computational explanatory gap* between cognitive and neurocomputational algorithms that hinders development of human-level neurocognitive models (Reggia et al., 2017, 2014). What improvements to neural networks might help bridge this gap?

Working memory is a form of short-term memory that is actively manipulated during cognitive processing, and has been extensively studied in both the neural and cognitive sciences (D'Esposito and Postle, 2015; Baddeley, 2002). Because working memory capacity is strongly correlated with measures of general intelligence in humans (Conway et al., 2003; Colom et al., 2004; Jaeggi et al., 2008; Oberauer et al., 2008, 2007), advances in working memory mechanisms may significantly improve the cognitive capabilities of neural networks, and may also have implications for perceptual and motor learning in both humans and cognitive robots (Persiani et al., 2018; Phillips and Noelle, 2005; Montero-Odasso et al., 2012; Sidarta et al., 2018; Shuggi et al., 2017). In addition, working memory includes both static and dynamic aspects that are reminiscent of symbolic programming; it contains richly organized representations (data structures) that are algorithmically manipulated by top-down cognitive control (Oberauer, 2009; Edin et al., 2009; Zanto et al., 2011; D'Esposito and Postle, 2015). This makes working memory a promising system of study for bridging the computational explanatory gap (Reggia et al., 2019, 2020).

Recently, neural attention has greatly improved the performance of neural networks on challenging natural language processing tasks such as machine translation and image captioning (Galassi et al., 2020; Vaswani et al., 2017; Sukhbaatar et al., 2015; Bahdanau et al., 2015; Rush et al., 2015; You et al., 2016). Attention mechanisms allow neural networks to take a more active role in data processing by selectively filtering and routing information via top-down gating. However, even with attention, deep neural networks have a limited ability to learn tasks that require robust compositional reasoning (Hupkes et al., 2020; Lake and Baroni, 2018; Loula et al., 2018), and we propose that this is due in part to biologically-implausible working memory. For example, Transformer networks (Vaswani et al., 2017) perform temporally non-local operations over a history of activation states, while models like Neural Turing Machines and Differentiable Neural Computers (Graves et al., 2014, 2016) include specialized RAM-like circuitry dedicated to maintaining activity patterns in spatially segregated neural populations. In contrast, neuroscientific evidence suggests that human working memory is supported by activity-silent mechanisms such as rapid synaptic plasticity (Manohar et al., 2019; Mongillo et al., 2008; Rose et al., 2016; Stokes, 2015; Barbosa et al., 2019). This form of working memory permits storage of multiple representations in a shared neural substrate, and requires top-down control that is qualitatively different from the filtering/selection provided by neural attention. Because of the strong relationship between working memory control and cognition in general, improvements to neural working memory that achieve temporal locality without specialized memory arrays may provide the necessary tools for learning high-level cognitive behaviors that are currently beyond the reach of contemporary neural models.

Fast associative learning can be used to establish activity patterns as attractor states in recurrent neural networks, making them available for subsequent recall without persistent maintenance (Ba et al., 2016; Hopfield, 1982; Amit, 1992; Pascanu and Jaeger, 2011; Miller, 2016; Hoshino et al., 1997). Such attractor neural networks have a long history in neurocomputation research, but have only recently addressed the compositional structure and procedural top-down control that are characteristic of human working memory. Recently developed "programmable attractor networks" achieve human-like control of working memory on several cognitive tasks (Sylvester and Reggia, 2016; Sylvester et al., 2013), and are capable of storing and manipulating structured representations such as lists, associative arrays, and trees (Davis et al., 2021a). These networks feature temporally-local control of distributed (non-local) representations in working memory that is based on itinerant attractor dynamics, fast associative learning, and top-down gating. Critically, their behavior is directed by learned instruction sequences that are themselves stored as sequential attractors in memory, and they can be flexibly "reprogrammed" to perform new tasks without changes to the underlying neural architecture.

One such network, the "Neural Virtual Machine" (NVM), is a purely neural system with program-independent circuitry that supports the key functionality of conventional computer architectures (Katz et al., 2019). While the NVM achieves universal neural programming and can simulate any Turing machine, its low-level assembly-like language makes it difficult to express high-level programs that are common in symbolic AI. In addition, it features segregated regions for representing programs and data, making it difficult to implement cognitive procedures that involve reasoning about behavior (e.g., planning, imitation, metacognition, etc). These procedures are more readily implemented in high-level languages that treat programs as "first-class citizens" that can be programmatically manipulated, such as

LISP or Scheme.

In this paper, we present *NeuroLISP*, an attractor neural network that can represent and execute programs written in the LISP programming language. NeuroLISP implements the core functionality of a LISP interpreter using only neural computations, and demonstrates how high-level symbolic structures can be reliably constructed and manipulated by sub-symbolic neural processes. As such, our model contributes to bridging the computational explanatory gap, and may inform studies on the neural basis of cognition and consciousness (Reggia et al., 2019). In addition, NeuroLISP serves as a purely-neural replacement for the top-down control provided by symbolic algorithms in hybrid models, and has the potential to carry over the unique advantages of neural computation to high-level cognition, such as adaptive learning, improved generalization abilities, fault tolerance, and seamless integration with low-level neural models of sensory and motor processing.

To our knowledge, this is the first effort to implement a high-level functional programming language in a fixed neural architecture with distributed representations. NeuroLISP is based on the same core principles as the NVM; namely, itinerant attractor dynamics, fast associative learning, and top-down gating. However, it implements several features of high-level symbolic programming that are absent in the NVM and other programmable neural networks, such as native support for compositional data structures, scoped variable binding, and the ability to construct, manipulate, and execute programmatic expressions (i.e., programs can be treated as data). These features facilitate implementation of high-level cognitive processes by improving both the static and dynamic components of working memory.

We present empirical results that demonstrate the breadth of NeuroLISP's capabilities. After verifying the correctness of the implemented interpreter with a suite of handwritten tests, we evaluated the network's memory capacity with basic programs involving list storage and variable binding. Our results show that the network's memory capacity scales linearly with the size of its memory regions. Next, we trained NeuroLISP with a small library of multiway tree processing algorithms, including depth-first traversal and substitution, demonstrating its ability to learn procedures that manipulate complex data structures. Then, we evaluated NeuroLISP using programs with greater relevance to artificial intelligence. Specifically, we tested a library of sequence manipulation functions that solves the PCFG SET task, a benchmark for compositionality in machine learning models (Hupkes et al., 2020), and a first-order unification algorithm that performs symbolic pattern matching, a key component of automated reasoning. With sufficiently sized neural regions, NeuroLISP achieved perfect performance on test cases with significant memory and processing demands. Finally, we evaluated runtime and memory usage to show that the model can be simulated efficiently, and that it scales well on parallel computing hardware. We conclude that NeuroLISP is an effective neurocognitive controller that can replace the symbolic components that provide robust top-down control in hybrid models, and serves as a proof of concept for further development of high-level symbolic programming in neural networks.

2. Methods

LISP is a family of high-level programming languages with an extensive history of use in artificial intelligence (McCarthy et al., 1965; Norvig, 1992). Today, active communities of developers exist for several dialects of LISP, including Common Lisp (Seibel, 2006), Clojure (Hickey, 2008), and Racket (Felleisen et al., 2015). LISP is celebrated for the simplicity and consistency of its syntax and underlying data structures: the contents of memory are made up of "s-expressions" (symbolic expressions), each of which is either an atomic symbol or a pair containing two s-expressions (referred to as a "cons cell"). This recursive definition permits expression of compositional structures such as lists and trees. Notably, s-expressions are used to represent both programs and the data they manipulate, which facilitates programmatic modification and generation of programs (i.e., *programs as data*). LISP also includes operators that allow programs to influence their own evaluation and switch between treatment of s-expressions as programs or data: the "quote" operation prevents evaluation of a sub-expression in the program, and instead returns it directly as data, while the "eval" operation explicitly induces evaluation of an s-expression that was returned as data from evaluation of a program sub-expression. Altogether, the ability to interchange programs and data makes LISP a valuable language for modeling high-level cognitive functions that include reasoning about behavior, such as planning, imitation, and metacognition, which are difficult for neural networks to learn. More generally, high-level symbolic programming provides a number of useful tools for cognitive modeling, such as scoped variable binding and compositional data structures.

NeuroLISP[1] is a purely-neural model that emulates an interpreter for a dialect of LISP that includes the core functionality of Common Lisp, and serves as a proof of concept for further development of high-level symbolic programming in neural networks. The operators supported in NeuroLISP are listed in Table 1 and described in more detail in Appendix A. NeuroLISP represents discrete symbols as distributed patterns of neural activation that

Table 1: Operators supported in NeuroLISP

Lists (Cons Cells)	`cons`, `car`, `cdr`, `cadr`, `list`
Hash Maps (Associative Arrays)	`makehash`, `checkhash`, `gethash`, `sethash`, `remhash`
I/O	`read`, `print`
Function Definition	`defun`, `lambda`, `label`
Variable Binding	`let`, `setq`
Conditional Statements	`cond`, `if`
Logical Statements	`eq`, `atom`, `listp`, `not`, `and`, `or`
Evaluation	`eval`, `quote`
Control	`progn`, `dolist`, `error`, `halt`

are organized into complex data structures by learned associations in neural pathways. The high-level workflow of NeuroLISP is shown in Figure 1. The model is constructed by a one-time user-configurable procedure that involves

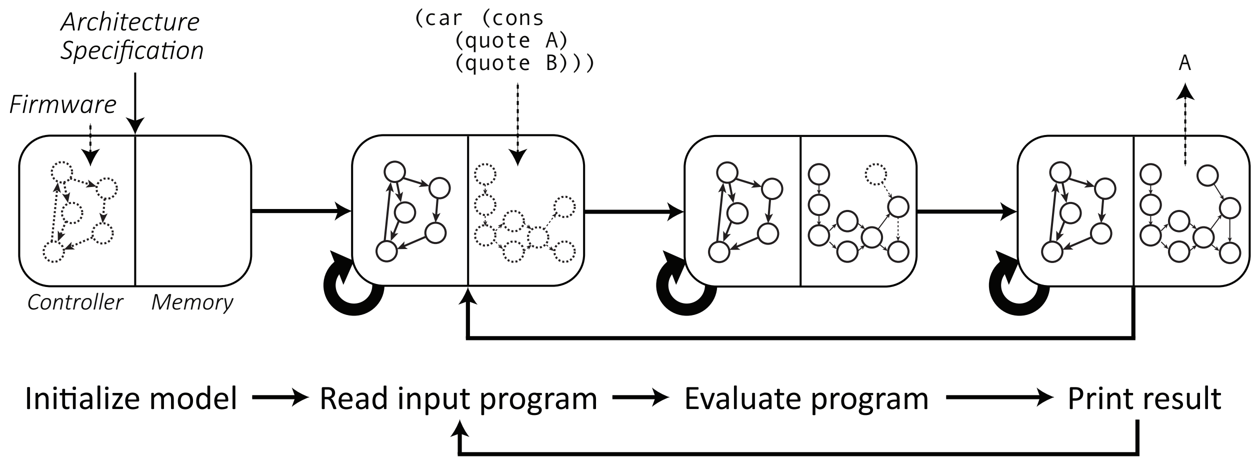

Figure 1: NeuroLISP workflow. Graphs depict learned distributed activity states and transitions in neural memory. First, the model is constructed and initialized by a one-time procedure (left) that constructs the neural components and “flashes” the interpreter firmware (dashed graph in left Controller half of model). Then, the model begins execution in a read-eval-print-loop that begins by parsing a sequence of inputs representing a program to be executed (center left). The program depicted here (top center) constructs a cons cell containing two symbols, `(A B)`, and retrieves the `car` (first) element. The program is fed into the model as a sequence of activation states over time, each representing one symbol in the program. During parsing, the model modifies its memory to create a representation of the program in neural memory (dashed graph in right Memory half of model). Once the program is parsed and learned in memory, it is evaluated, which may involve construction of new memories (dashed circle and arrows, center right). Upon completion, the result (`A`) is printed as output of the model (right) via a sequence of activation states representing the symbols in the output stream. Finally, the model returns to the beginning of the loop to parse the next program. The previously learned programs remain in memory as new programs are learned.

learning the underlying “firmware” of the LISP interpreter using a one-step associative learning rule. Subsequently, the model executes a cycle of activity that 1) reads a sequence of input activity patterns specifying a program to be executed, 2) evaluates the program according to the implemented language by modifying its memory, and 3) prints

[1]https://github.com/vicariousgreg/neurolisp

the result as a sequence of output activity patterns. Programs are read in as temporally-extended sequences of neural inputs, and stored in memory as systems of interconnected attractor states called attractor graphs (Davis et al., 2021a) in a recurrent neural region that represents a shared program/data memory space. During evaluation, new memories are constructed based on the interpreted programmatic expressions, and the final result is printed via sequential activation of neural patterns that represent a stream of output symbols. The neurocomputational procedures involved in parsing, evaluation, and printing are discussed in Section 2.3.

In the following, we first outline the mechanisms that govern model execution and the various types of dynamics that they support (Section 2.1). Then we describe the fundamental data structures of the virtual interpreter, their representation as systems of attractors, and the basic operations that are performed on them via algorithmic control of top-down gating (Section 2.2). This is followed by an explanation of the virtual interpreter, including expression evaluation, comparison operations, input/output, scoped variable bindings, function definitions, and function applications (Section 2.3). Finally, we present experimental methods (Section 2.4) and empirical results (Section 3) that demonstrate that NeuroLISP properly implements the LISP programming language and can successfully execute high-level programs.

2.1. Neural Architecture

NeuroLISP is a multi-region recurrent neural network with gated inter-connections that implements a virtual LISP interpreter. The architecture of the model (shown in Figure 2 and described in more detail in Appendix B) is inspired by the Neural Virtual Machine (Katz et al., 2019) and the stack machine architectures used for early LISP machines (Koopman, 1989). A Controller sub-network (bottom left) controls the flow of information processing through the model over time by translating patterns of activation into temporally-extended gating of model components. These activation patterns represent learned programs that are evaluated by the underlying virtual interpreter, and can be modified during program evaluation (i.e., programs can be treated as data). Discrete symbols are represented by patterns of activation in a lexicon region (*lex*, center of Figure 2) that can be exchanged as input and output with an external environment, allowing the model to be programmed via environmental interactions. Unlike contemporary deep neural networks, NeuroLISP uses a one-step local learning rule that permits rapid modification of associative networks within and between regions. This learning rule is used for both one-time initialization of interpreter functions and online learning during program execution (e.g., when modifying variable bindings, creating new data structures, and modifying runtime/data stacks).

Regions in NeuroLISP represent symbolic information as distributed patterns of activation, and function according to a shared set of rules for activation dynamics and learning. Neurons in a region r receive inputs from a variety of sources, each with a unique gate that determines when it is active during model execution:

$$\mathbf{s}_r(t) = \underbrace{\sum_{q,\ell} g_{r,q[\ell]}(t)\; W_{r,q[\ell]}(t)\; \mathbf{v}_q(t)}_{\text{weighted connectivity}} + \underbrace{g_r^{bias}(t)\; \mathbf{b}_r}_{\text{bias}} + \underbrace{g_r^{noise}(t)\; \mathbf{n}_r(t)}_{\text{noise}} + \underbrace{g_r^{read}(t)\; \mathbf{I}_r(t)}_{\text{external inputs}} + \underbrace{g_r^{saturate}(t)\; \sigma_r^{-1}(\mathbf{v}_r(t))}_{\text{maintenance}} \tag{1}$$

where $\mathbf{s}_r(t)$ is a vector of cumulative synaptic input to region r at time t that is aggregated from several sources:

- weighted inputs from connected regions (solid lines with arrow heads in Figure 2). $W_{r,q[\ell]}$ is a weight matrix for the connection from region q to region r that is active when $g_{r,q[\ell]}(t) = 1$, ℓ is a label that distinguishes between weight matrices that share source and target regions, and $\mathbf{v}_q(t)$ is a vector of neural activity in source region q at time t. When $r = q$, the connection is recurrent (looped arrows in Figure 2).

- bias vector. $\mathbf{b}_r$ is a bias vector for region r that is active when $g_r^{bias}(t) = 1$. The bias term is used by the *gate sequence* region during comparison operations (Section 2.3.2).

- random noise. When $g_r^{noise}(t) = 1$, a vector of random inputs $\mathbf{n}_r(t)$ generates a random activity pattern in region r at time t. The random vector $\mathbf{n}_r(t)$ is produced by a Bernoulli process with probability λ_r. For regions with recurrent dynamics, $\lambda_r = 0.5$ to maintain balance between positive and negative activation levels. For context regions (labeled *ctx* in Figure 2), λ_r is a variable parameter (for details on the implications of this parameter on contextualized attractor dynamics, see Davis et al. (2021a)).

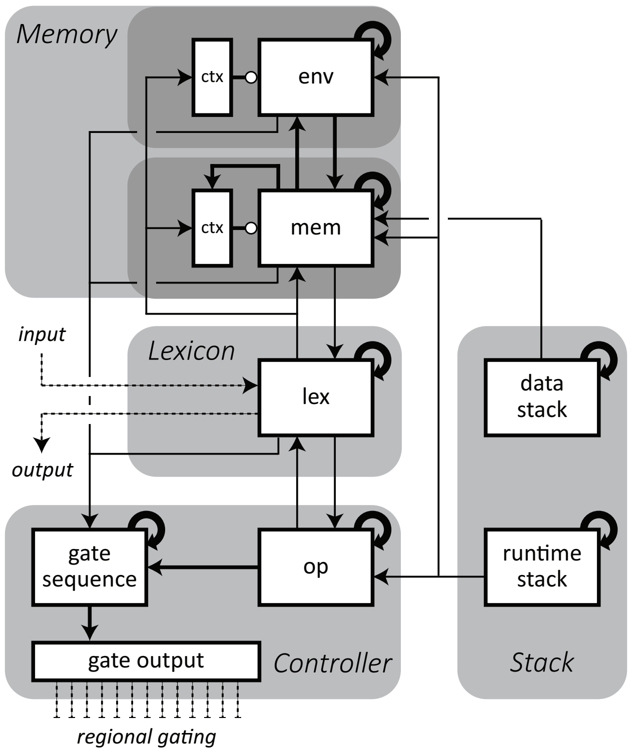

Figure 2: NeuroLISP architecture, inspired by the Neural Virtual Machine (Katz et al., 2019) and the stack machine architecture of traditional LISP machines (Koopman, 1989). The model is made up of several neural regions (boxes) with recurrent and inter-regional connectivity (looped and straight arrows with solid lines). Connections are controlled by neurons in the *gate output* region of the Controller sub-network (bottom left), which determine the components of the model that are active at each timestep (regional gating, dashed lines). Each gate ($g(t)$ with subscripts in Equations 1 - 6) is assigned to a unique neuron in the *gate output* region, and its activation level is used to determine whether the gate is open or closed at each timestep. Activation of the *gate output* region is guided by a cascade of regions with recurrent dynamics (*gate sequence* and *op*) that implement the core functionality of a virtual LISP interpreter. Together, the Controller regions translate learned LISP programs to temporally-extended sequences of regional gating that specify pathways for information processing over time, much like the control unit of a conventional computer architecture. The lexicon region (*lex*, center) serves as a bridge between model components, and its activity patterns represent discrete symbols that may correspond to interpreter functions (i.e., LISP operators) or arbitrary symbols that can be read from or written to an external environment (dashed input/output lines, center left). Data structures, including LISP programs, are represented by systems of attractors in the core memory region (*mem*, center), which store symbolic contents via the pathway from *mem* to *lex*. The remaining components support interpretive functions: the environment region (*env*, top) stores a tree structure containing namespaces of variable bindings that are modified and accessed during program execution, and the Stack sub-network regions (*runtime stack* and *data stack*, right) store stack sequences made up of pointers to various activation states in the model, making them accessible without persistent maintenance.

- external inputs. When $g_r^{read}(t) = 1$, region r "reads" an input pattern $\mathbf{I}_r(t)$ from the external environment. The dashed line entering the *lex* region in Figure 2 indicates external inputs. The adjacent dashed line labeled *output* indicates gated outputs that are "printed" to the environment when $g_r^{write}(t) = 1$. Unlike other gates in the model, these gates signal to the external environment to interact with the activity in the *lex* region. In principle, input and output processes could be implemented by gated pathways with additional sensory and motor networks that control continuous behavior in a simulated or real environment. Here we focus on cognitive control of such processes, and omit the low-level networks involved with sensorimotor processing.

- activity maintenance. When $g_r^{saturate}(t) = 1$, activity $\mathbf{v}_r(t)$ in region r is cycled back into the region's inputs to maintain it over time. σ_r^{-1} is the inverse of the region's activation function. For simplicity, we assume that $g_r^{saturate}(t) = 1$ whenever all of the above gates are closed (i.e., a region maintains its activation pattern whenever it is not receiving synaptic input).

A region may also receive gated inputs that contextualize its dynamics via multiplicative modulation:

$$\mathbf{x}_r(t) = \prod_q \begin{cases} \mathbf{v}_q(t) > 0, & \text{if } g^{context}_{r,q}(t) = 1 \\ \mathbf{1}, & \text{otherwise} \end{cases} \tag{2}$$

where $\mathbf{x}_r(t)$ is a vector of cumulative multiplicative inputs to region r at time t, $\mathbf{1}$ is a vector all ones, and $\prod$ indicates the Hadamard product of a set of vectors. $\mathbf{x}_r(t)$ is aggregated from the activity state $\mathbf{v}_q(t)$ of each region q that provides multiplicative inputs when the corresponding gate $g^{context}_{r,q}(t) = 1$. These connections are depicted by solid lines with circular heads in Figure 2. When none of these gates are active (or if region r has no contextual inputs), $\mathbf{x}_r(t) = \mathbf{1}$. Note that in NeuroLISP, there is a single dedicated context region for each recurrent region that receives contextual input (*mem* and *env*), but the mathematical model presented here does not impose such constraints.

These two types of inputs are combined and passed into the neural activation function:

$$\mathbf{v}_r(t+1) = \sigma_r\Big(\mathbf{x}_r(t) \odot \mathbf{s}_r(t)\Big) \tag{3}$$

where $\mathbf{v}_r(t)$ is a vector of neural activity in region r at time t, σ_r is the activation function of neurons in region r, and $\odot$ is the Hadamard product. The synaptic inputs $\mathbf{s}_r(t)$ are gated by multiplicative inputs $\mathbf{x}_r(t)$ before being passed into the activation function. When $\mathbf{x}_r(t)$ contains zeroes, the corresponding neurons receive no net input. Note that recurrent regions with contextual dynamics must use a sign-preserving bipolar activation function (e.g., sign/signum or the hyperbolic tangent). This ensures that deactivated neurons function differently from neurons receiving strong negative input.

Learning in the model occurs in two stages, each controlled by a different type of gate. The first stage involves updating a regional eligibility trace to store the current pattern of activation as a target for subsequent learning:

$$\epsilon_r(t+1) = \begin{cases} \sigma_r^{-1}\Big(\mathbf{v}_r(t+1)\Big), & \text{if } g^{\epsilon}_r(t) = 1 \\ \epsilon_r(t), & \text{otherwise} \end{cases} \tag{4}$$

where $\epsilon_r(t)$ is the eligibility trace for region r at time t, $\mathbf{v}_r(t+1)$ is the most recently computed activity pattern in region r for timestep $t+1$ (Equation 3), and σ_r^{-1} is the inverse of the activation function for region r. When $g^{\epsilon}_r(t) = 1$, $\epsilon_r(t)$ is updated such that $\sigma_r\,(\epsilon_r(t+1)) = \mathbf{v}(t+1)$. We refer to $\epsilon_r(t)$ as an eligibility trace, a term borrowed from reinforcement learning, because it temporarily stores an activity state for use in the second stage of learning, in which a pathway-specific weight matrix is updated with the store-erase learning rule (Katz et al., 2019):

$$\Delta W_{r,q[\ell]}(t) = \underbrace{\frac{1}{\|\mathbf{v}_q(t)\|}}_{\text{norm}} \Bigg(\underbrace{\epsilon_r(t) - \Big(\mathbf{x}_r(t) \odot W_{r,q[\ell]}(t)\,\mathbf{v}_q(t)\Big)}_{\text{target delta}}\Bigg) \underbrace{\mathbf{v}_q(t)^{\top}}_{\text{source}} \tag{5}$$

$$W_{r,q[\ell]}(t+1) = W_{r,q[\ell]}(t) + g^{learn}_{r,q[\ell]}(t)\,\Delta W_{r,q[\ell]}(t) \tag{6}$$

where $W_{r,q[\ell]}(t)$ is the weight matrix for connection ℓ from region q to region r at time t. Weight updates are distributed across the weight matrix and are normalized according to the magnitude of the source pattern. When $g^{learn}_{r,q[\ell]}(t) = 1$, weights are updated such that the current inputs will produce the eligibility trace in the future:

$$\mathbf{x}_r(t) \odot W_{r,q[\ell]}(t+1)\,\mathbf{v}_q(t) = \epsilon_r(t)$$

This equality is only guaranteed to hold for the most recently learned association, which may deteriorate as additional associations are learned. This was investigated empirically in Davis et al. (2021a), in which practical memory capacities were established for networks that learn using contextually-gated store-erase learning.

The mathematical model outlined above affords a diverse set of dynamics that depend on top-down control of regional gating over time (e.g., $g_{r,q[\ell]}(t)$, $g^{noise}_r(t)$, etc). In NeuroLISP, these gates are controlled by neurons in the *gate output* region (one neuron per gate) based on learned sequences of activation patterns in the Controller sub-network

(Figure 2). This allows the model to be "programmed" with new computational procedures that specify pathways through which activation flows, much like opcodes in the instruction set of a conventional computer architecture. These procedures make up the core functionality of the implemented language interpreter that are established using one-step learning during model construction, and remain fixed during model execution. Among these procedures are the core functions for constructing and accessing compositional data structures, which are implemented as systems of attractors linked by contextualized transitions (Section 2.2).

Connection gating is used for both inter-regional and recurrent connectivity. Inter-regional gating allows the model to control the spread of activation between neural regions, initializing a target region to a state that is specified by learned associations via weighted inputs from a source region. Recurrent dynamics within a region fall into one of two categories: attractor convergence and sequential transitions. By interleaving these dynamics, a region can be made to iterate through a sequence of attractor states, settling at each attractor before advancing to the next state in the sequence (i.e., itinerant attractor dynamics, Miller (2016); Hoshino et al. (1997)). Inter-regional and recurrent dynamics can be combined into complex behavior that is orchestrated by the gate controller according to learned "programs". For example, inter-regional gating may be used to initialize a region to the start of an attractor sequence that can be traversed with subsequent recurrent gating.

Attractor itinerancy is typically limited in that each state has a single successor state in the sequence. This is overcome by the addition of "contextual" multiplicative gating (Equations 2 and 3), which permits context-dependent recurrent transitions that depend non-linearly on inputs from another region. These inputs differ from weighted inter-regional inputs in that they do not drive the target region toward a particular pattern directly; instead, they contextualize its recurrent dynamics, selecting among multiple learned associations to govern each transition. Thus, when a region executes a transition in this regime, the consequent state depends on both the initial state of that region and the pattern of activation that is used to contextualize the transition. This "functional branching" makes it possible to learn directed graphs of attractors and transitions through which a region may traverse. In previous work, we have shown that attractor graphs can efficiently represent compositional data structures such as linked lists, associative arrays, and trees (Davis et al., 2021a). Such structures can be traversed via temporally-extended top-down control of regional and contextual gating.

Previous programmable attractor networks have relied primarily on itinerant attractor sequences without contextual gating, which restricts the space of possible programming languages that may be implemented. For example, the Neural Virtual Machine implements an assembly-like language, with programs represented as linear sequences of instructions (Katz et al., 2019). This makes it difficult to encode complex programs that are much more easily expressed in higher-level languages as abstract syntax trees, which can be represented directly in neural memory as attractor graphs. In Section 2.2, we show how the "cons cells" of the LISP programming language can be represented by simple attractor graphs and composed into nested expressions to represent complex programs (stored in the *mem* region). These expressions can then be recursively evaluated by other regions with simpler non-contextualized dynamics (Controller regions) that implement an assembly-like language suitable for defining LISP interpreter functions (e.g., evaluation, variable lookups, input/output, etc), as described in Section 2.3.

2.2. Compositional Data Structures

Compositional data structures are implemented in NeuroLISP as systems of attractors (distributed representations) with gated transitions, called attractor graphs. The details of the dynamics underlying attractor graphs can be found in Davis et al. (2021a). Here we describe how they are used to implement cons cells and associative arrays (maps), two fundamental data structures that serve as building blocks for NeuroLISP's memory system (Figure 3). These data structures can be constructed and accessed in neural memory via computationally-efficient gating operations with constant time and linear memory requirements.

2.2.1. Cons Cells

Cons cells are ordered pairs of elements that may be atomic symbols or other cons cells. Atomic symbols are represented as attractor states in the memory (*mem*) region that are associated with a corresponding pattern of activation in the lexicon (*lex*) region (Figure 3a). Cons cells are also represented as *mem* attractor states (Figure 3b), but they differ from atomic symbols in three ways. First, they are associated with a reserved *lex* pattern that identifies the memory state as a cons cell. Second, each cons cell serves as the source state for a unique sequence of transitions

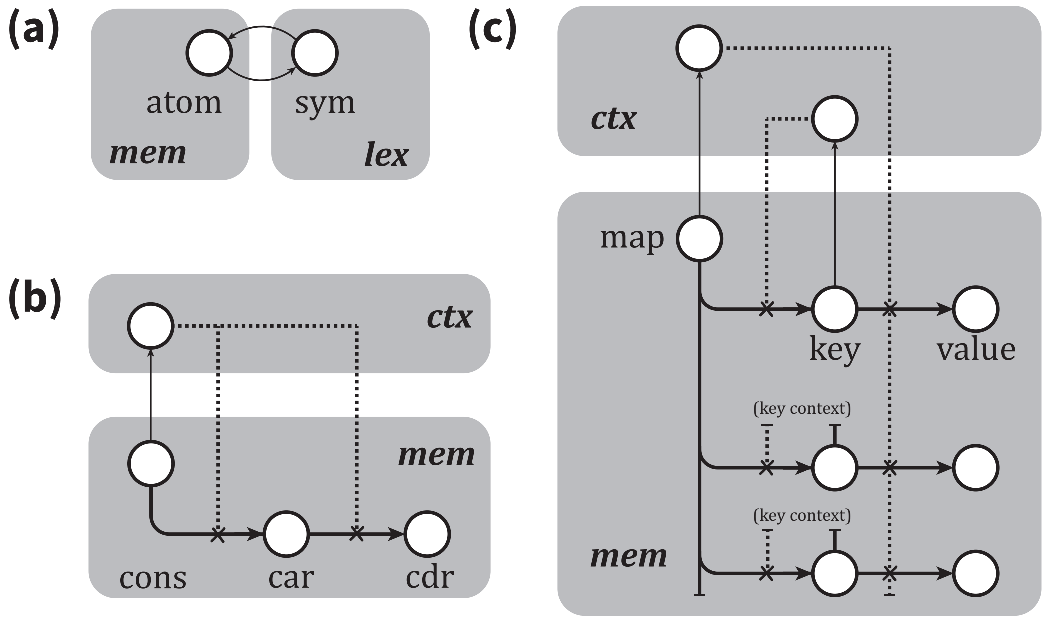

Figure 3: Graphical depiction of attractor graph representations of fundamental data structures (Davis et al., 2021a). Each gray rectangle represents the activity space of a region (*mem*, *lex*, or *ctx*), and each circle represents a unique distributed neural activity pattern in that region. Solid lines indicate learned associations between states, either between or within regions. Dashed lines indicate contextual dependencies for recurrent *mem* transitions, which can only be executed when the corresponding *ctx* pattern is present. (a) Atomic symbols are represented by pairs of states in the *lex* and *mem* region. The *mem* state allows the symbol to participate in compositional structures, while the *lex* state allows the symbol to be read from or written to the environment, and interpreted as a variable name, function name, or LISP operator. (b) Cons cells are represented by a unique activity state (labeled "cons") that serves as the head of a trajectory through the elements contained in the cell (labeled "car" and "cdr"). These transitions are contextualized by a unique *ctx* state (circle within *ctx* rectangle). (c) Associative arrays (maps) are represented similarly, except that there are multiple trajectories from the head state (labeled "map") that run through each key/value pair in the map. Each transition to a key is contextualized by a unique *ctx* state associated with the key (lower right circle in *ctx* rectangle). This permits verification that a key is contained in the map prior to value lookups (see Appendix D). Each key state is associated with the corresponding value in the map by a unique map context state (top right circle in *ctx* rectangle). This allows the same memory state to serve as a key in multiple maps, each with a unique corresponding value. For clarity, we only show the context states for a single key/value pair, and abbreviate the remaining pairs (bottom of *mem* rectangle).

from the cons cell through the two elements of its ordered pair. Third, each cons cell *mem* state is associated with a unique state in the memory context region (*ctx* region adjacent to *mem*) that contextualizes the transitions linking the cons cell with its elements. This organization allows memory states to be contained as elements in several cons cells without duplication, as the transitions linking the elements of a cons cell are contextualized by a unique multiplicative pattern (see Appendix C).

The car and cdr operations retrieve the first and second elements of a cons cell, respectively. These operations can be performed by iterating through the `mem` attractor sequence, starting with the cons cell attractor, and stopping at the desired element. This sequence is contextualized by the unique `ctx` state associated with the cons cell, which must be retrieved prior to iteration. To construct a cons cell, a `mem` attractor sequence must be constructed using several gates, including the noise (Equation 1), eligibility trace (Equation 4), and plasticity gates (Equation 6). This sequence links together a newly created cons attractor with the two elements that will be contained as car and cdr elements. The details of these operations are included in Appendix C.

2.2.2. Associative Arrays (Maps)

Associative arrays, or maps, are collections of key/value pairs. The fundamental operations of maps include addition, modification, and removal of key/value pairs, checking whether a key/value pair exists for a given key, and retrieving that value if it exists. Maps are generally implemented as hash tables in conventional computers, which use a hashing function to transform keys into unique offsets for indexing an array in linear memory. In lieu of such a hashing function, we use neural attractor transitions to uniquely associate keys with values, similarly to the above implementation of cons cells. Several unique features of maps, however, make their underlying implementation more involved than cons cells.

The organization of maps in neural memory is shown in Figure 3c. Each key/value pair in a map corresponds to a pair of attractor states linked by a contextualized transition. Unlike cons cells, the first element of a pair (the key) is provided during operations performed on the map (e.g., lookups). To retrieve the value for a given key, the context state for the map is used to execute a transition from the key's memory state to the value's memory state, much like the cdr element of a cons cell is retrieved from the corresponding car element. However, unlike with cons cells, the corresponding transition from the map to the key state is not contextualized by the map's context state, as a map may contain multiple key/value pairs. Instead, a context state associated with the key memory is used for the transition, making it possible to check whether a key is contained in a map (see Appendix D for details).

2.3. Virtual Interpreter

NeuroLISP implements a virtual interpreter that evaluates programmatic expressions stored as nested cons cells in neural memory (Section 2.2.1). Interpreter functions are orchestrated by sequential activation in the Controller sub-network, which controls model functionality by opening and closing gates on model components over time (Equations 1 - 6). Much like a conventional computer, the Controller specifies pathways through the architecture for information processing based on learned instructions. This includes manipulation of memory, runtime/data stacks, input/output pathways, and environmental variable bindings, but also the functionality of the Controller itself. For example, conditional statements require the Controller to execute different procedures based on the result of operations performed on the contents of memory (e.g., comparisons, logical operations). Interpreter functionality in the Controller (i.e., interpreter firmware) is learned with fast associative learning (Equations 4 - 6) during a one-time initialization procedure.

The design of the Controller and Stack sub-networks is inspired by the Neural Virtual Machine, which implements an assembly-like language and represents programs using temporal sequences of distributed neural activity patterns (Katz et al., 2019). Sequences in the *op* region represent programs in a low-level assembly-like language that implement interpreter functions for the higher-level LISP language. We refer to these low-level programs as op-sequences to distinguish them from LISP programs. The relationship between these two levels is discussed in more detail in Appendix E.

2.3.1. Evaluation

In LISP, *eval* is a core interpreter function that recursively evaluates a LISP expression and returns the result. The *eval* function contains conditional logic that determines how an expression is to be evaluated based on its contents and structure, and invokes other necessary interpreter functions. For example, an atomic expression is interpreted as a variable name to be looked up in the environment, while a list is interpreted as an application of a function or built-in operator.

The *eval* function is implemented in NeuroLISP as a central op-sequence of the Controller that branches off into one of several other op-sequences based on the currently active pattern of activity in *mem*, which represents the LISP expression to be evaluated. This begins a cascade of op-sequence calls that implements the expression's operator via top-down control of gated neural computations, and may include recursive evaluation of sub-expressions. During recursive evaluation, the Controller uses the stack regions to temporarily store `op`, `mem`, and `env` activity states by learning associations in the corresponding pathways (see Figure 2). For example, evaluation of a cons expression (e.g., `(cons 'a 'b)`) begins with recursive evaluation of the sub-expressions for the car and cdr elements, which are stored temporarily on the data stack and retrieved during construction of the cons cell attractor sequence (Section 2.2.1). Further details on recursive evaluation are included in Appendix E.

Compound expressions are stored in memory as lists represented by chains of cons cells. The first element of the list represents either a built-in operation (Table 1) or a sub-expression that can be evaluated to retrieve a function (e.g., a function name or lambda expression). To distinguish between these cases, all patterns in the *lex* region representing symbols for built-in operators are learned as attractors in a recurrent *lex* matrix. These patterns are recognizable via comparison (Section 2.3.2): if the pattern remains stable following recurrent dynamics, it represents a built-in operator. Each built-in operator has a corresponding *op* sequence that implements its operation and can be retrieved via the pathway from *lex* to *op*. When the first element of a compound expression is not a built-in operation, it must evaluate to a function, and may either be a variable naming a function that can be retrieved from the environment (Section 2.3.4), or a cons cell representing a lambda function (Section 2.3.5). In either of these cases, the operator is recursively evaluated, and the parent expression is interpreted as a call to the returned function, with the remaining elements of the list interpreted as expressions for the values of the function's arguments. When evaluating non-compound expressions (i.e., individual symbols), built-in symbols are simply returned, and other symbols are interpreted as variable names, and looked up in the environment.

Two special LISP operators provide programmatic control of evaluation: `quote` and `eval`. The `quote` operator instructs the interpreter to skip evaluation of sub-expressions and return them directly as data. Conversely, the `eval` operator instructs the interpreter to evaluate the value that was returned from evaluation of the sub-expression. These two operators allow seamless interchange of programs and data (i.e., *programs as data*), and make it straightforward to implement programs that generate other programs. The `quote` operator is implemented in NeuroLISP by an op-sequence that simply retrieves the memory state representing the expression's argument (e.g., evaluating `(quote x)` returns the memory state representing the symbol `x`). The op-sequence implementing the `eval` operator recursively evaluates the argument sub-expression, then performs a second round of recursive evaluation on the resulting memory state, and returns the final result.

2.3.2. Conditional Evaluation

Conditional evaluation involves comparisons that are initiated by branching instructions in the *op* region. A comparison is performed on two activity states that occur in the same region (one of *mem*, *lex*, or *env*) at different timesteps. The result of the comparison causes the *gate sequence* region to initiate one of two operations for advancing the *op* region, analogous to the jump operations that occur in conventional computer architectures. If the compared states are within a threshold of similarity, *op* is advanced to a new sequence designated by the branch operation's operand. Otherwise, *op* is simply advanced to the next instruction in the current op-sequence.

Comparisons are performed in two stages. For illustrative purposes, we consider a comparison performed between two *mem* states to determine whether a key is contained in an associative array / map (Section 2.2.2). First, the key memory state is retrieved, and an association is learned in the pathway from *mem* to *gate sequence* that links the key memory state with a designated *gate sequence* state corresponding to the jump gate procedure. Next, the key is used to execute a transition from the map state in *mem*, which yields the key state if the key is contained in the map, or a random state otherwise. The comparison is performed to determine which of these two cases occurred. At this point, two gates are opened in the *gate sequence* region: one that controls the pathway from *mem*, and another that activates a bias input that pushes *gate sequence* toward a designated sequence that corresponds to a false comparison. If the current *mem* state matches the memorized state (i.e., the key memory state), then the net input to *gate sequence* will match the jump sequence state. Otherwise, the net input will be dominated by the bias term. The resulting *gate sequence* activity performs the appropriate operation for advancing *op* according to the result of the comparison: jump to the operand if the result was true, or advance to the next instruction in the current op-sequence if false. The details of the comparison operation are discussed in Appendix F.

2.3.3. Input/Output

As mentioned in Section 2.1, symbols represented by *lex* activity patterns can be read from or printed to the environment via control of special input/output gates. Each of these patterns is also reciprocally associated with a unique activity pattern in *mem*, which allows the symbol to serve as a component of compositional structures (Figure 3a). The bi-directional associations between *mem* and *lex* representations of a symbol make it possible for the model to recognize that a symbol has never been seen before, and to construct a new *mem* state to represent it. This is done by reading the symbol to *lex*, memorizing it, executing an inter-regional transition from *lex* to *mem* and back to *lex*, and comparing the resulting *lex* activity pattern to the memorized pattern (Section 2.3.2). If they match, the symbol has a *mem* representation already. If not, one is created, and the bi-directional associations are created. Thus, NeuroLISP's lexicon is automatically expanded as it encounters new input symbols.

A read operation parses a sequence of symbolic inputs into a data structure in memory. Two special symbols representing open and closed parentheses indicate delimiters of nested expressions, which can be parsed recursively. When the open symbol is encountered, the interpreter enters a loop, recursively parsing each symbol until a close symbol is encountered. Each parsed memory, besides the close symbol, is placed in a chain of cons cells representing a list. An additional quotation symbol streamlines expression quotation; when it is encountered, the result of parsing the next symbol(s) is appended to a list containing the `quote` symbol. A read operation is completed when the expression is closed (or immediately after reading a non-delimiting symbol), and the resulting memory structure is returned to the caller. Conversely, a write operation performs a pre-order traversal of a memory structure, printing an open parenthesis upon entry to a cons cell, a close parenthesis upon exit of a cons cell, and the corresponding symbol when a leaf node is reached.

NeuroLISP begins operation in a read-eval-print-loop, in which an expression to be evaluated is read in as input, constructing a program in memory to be evaluated. The memory state that is returned from evaluation is then printed, and the loop repeats. Thus, once the model is initialized with interpreter functions, it can be programmed via environmental interactions that prompt the model to control its own plasticity, rather than by direct manipulation of the weight matrices in the model.

2.3.4. Environment Management

In high-level programming languages, variable bindings are maintained and updated in an *environment* that is accessible during program evaluation. Environments are typically composed of distinct *namespaces* that manage different bindings for the same variable name that are relevant to different execution contexts. For example, if a function *f(x)* calls another function *g(x)*, two distinct bindings are maintained for *x* that may contain different values. When a lookup is performed, the evaluator must retrieve the correct binding based on the current execution context. This can be done *dynamically*, in which the most recent binding is retrieved, or *lexically*, in which the correct binding is determined based on the location of the expression being executed within the program. Lexical scoping is more complex than dynamic scoping because variable lookups are relative to the code being executed, and several bindings for a given variable must be maintained separately. The organization of environments is described in detail in Appendix G. NeuroLISP supports both dynamic and lexical scoping because the procedures for environment organization and access are determined by learned interpreter functions. Lexical scoping was used for the experiments described below.

2.3.5. Function Definitions and Applications

In lexically-scoped languages, function definition involves creation of a *closure* that binds together the body of the function, its argument list, and a namespace containing variable bindings that were accessible at the time of definition. In NeuroLISP, closures are stored as cons cells in the *mem* region (Section 2.2.1) that have special learned associations with namespaces represented in the *env* region (Section 2.3.4), and are associated with a reserved "function" symbol in *lex*. When a function is called, the corresponding namespace is retrieved, and a new namespace is constructed from it to store the bindings of the function's arguments. This namespace is then used for variable lookups during execution of the function body, and it contains both the argument bindings and the extant bindings from when the function was first defined (see Appendix H for details). When dynamic scoping is used, the closure namespace is ignored, and the new namespace is branched off of the caller's namespace.

2.4. Experimental Methods

We performed experiments on NeuroLISP using several different programs: 1) To verify its correctness, we tested a suite of 37 simple handwritten test cases that evaluate the various functions of the implemented language. 2) To determine the relationship between region sizing and memory capacity, we evaluated the network's memory capacity with basic programs involving list storage and variable binding. 3) To demonstrate that NeuroLISP can successfully execute basic LISP programs that manipulate complex data structures, we tested a small library of multiway tree processing functions. 4) To demonstrate that NeuroLISP is capable of compositional processing, we tested a library of sequence manipulation functions that solves the PCFG SET task described in Hupkes et al. (2020). 5) Finally, to show that NeuroLISP can perform high-level procedures that are relevant to traditional symbolic AI, we tested a first-order unification algorithm (Russell and Norvig, 2002), a key component of automated reasoning, type checking, and logic programming. Empirical results from these experiments are presented in Sections 3.1-3.5.

To distinguish between bugs in the interpreter firmware and neurocomputational errors (i.e., corruption of learned neural associations), we implemented a non-neural emulator for the NeuroLISP architecture. The emulator faithfully reproduces the flow of information that occurs through regions in the architecture, but uses explicit symbols and lookup tables in lieu of activation patterns and weight matrices. This allowed us to determine the number of associations in various pathways that would be learned during correct execution of a program, which we refer to as the program's complexity, without the possibility of interference from neurocomputational errors. Because the memory capacity of simple attractor networks is relative to the number of neurons in the network (Amit et al., 1985; Davis et al., 2021a; Katz et al., 2019), we sought to determine the relationship between the size of NeuroLISP regions and the complexity of programs that it can successfully execute. Specifically, we examined the number of generated memory states (attractors in the *mem* region) and the number of variable bindings (associations between *env* and *mem* states), relative to the size of the *mem*, *lex*, and *env* regions. In addition, we examined the impact of the context density parameter $\lambda_{env-ctx}$, which determines how many *env* neurons participate in each variable binding.

Each experiment included several trials in which the NeuroLISP architecture was instantiated with a particular set of model parameters, initialized with the interpreter firmware, and executed with a particular set of inputs encoding a test program. The output of the model was compared to the correct reference output for the trial inputs to determine if the trial was successful or not. Experiments were performed in blocks with shared model parameters and inputs with a shared property that were not necessarily identical (e.g., testing retention of different lists of the same length). Each block contained 20 trials, and results are reported as the percentage of trials in each block that produced correct outputs (each datapoint in the plots in Section 3.1-3.5 corresponds to one block of 20 trials). We systematically varied one model parameter at a time in order to determine its impact on model performance (e.g., how does the size of the *mem* region impact list retention?). The remaining parameters were set in order to avoid degradation of performance (e.g., we used a large *mem* region size when testing the impact of *env* region sizing on variable binding). The details of model parameters used during experiments can be found in Appendix I.

Our computational experiments address the following questions:

- **Correctness**: Does the NeuroLISP firmware correctly implement the language interpreter? Can NeuroLISP successfully execute high-level programs, including multiway tree processing functions, the string manipulation functions of the PCFG SET task, and a first-order unification procedure used in automated reasoning?
- **Memory capacity**: How do the sizes of the *mem* and *lex* regions impact program/data memory storage capacity?
- **Binding capacity**: How does the size of the *env* region and its context density parameter $\lambda_{env-ctx}$ affect a) the number of variable bindings that can be stored in a single namespace, and b) the number of namespaces that can store separate bindings for the same variable name?

We hypothesized that correct performance would require sizing the *mem*, *lex*, and *env* regions according to the complexity of the executed program. Based on results reported in Davis et al. (2021a), we predicted the following:

- **Region sizing**: a linear relationship between a) *mem* region size and program/data memory storage capacity capacity, and b) *env* region size and the number of bindings stored for the same variable name in different namespaces.
- **Context density**: a larger *env* context density $\lambda_{env-ctx}$ would facilitate storage of bindings with the same variable name in different namespaces, but interfere with storage of several variables in one namespace. Thus, a balanced context density would lead to the best performance on complex programs involving multiple bindings across multiple namespaces.

Finally, we investigated the scalability of the model in parallel computing environments with additional experiments that evaluated runtime performance and memory usage. Experiments were performed on a 3.5GHz 10-core Xeon E5-2687W v3 with two NVIDIA RTX 3060 GPUs. The model was implemented in the Python programming language using the `numpy` library for multi-dimensional arrays and the `pyCUDA` library for GPU processing. Model computations that involved matrix operations (e.g., input activation and learning) were performed on GPU(s), while those that did not (e.g., neural activation functions) were performed on the CPU. The results of these experiments are presented in Section 3.6. We investigated several approaches to improving both the runtime and memory efficiency of model simulation without affecting the accuracy of its symbolic behavior, as described below.

A significant advantage of the gated region-and-pathway paradigm is that only a subset of the model (corresponding to active regions and connections) must be computed during each timestep, greatly reducing the computational cost of model execution. This also means that scaling up the number of neurons in a specific region only incurs performance penalties for computations that involve that region and its associated connectivity matrices. For example, the size of the env region does not affect operations that do not involve variable bindings or namespaces, such as parsing inputs and constructing programs in memory. We illustrate this by separately measuring the runtime for program parsing and execution.

Because each timestep of simulation only involves a subset of model computations, the theoretical performance benefits from distributing kernels among multiple compute devices is limited. A more effective strategy would be to split up individual regions, divvying up neurons to separate devices for distributed computation of individual kernels. We do not pursue this possibility here, but note that it is feasible due to the locality of associative learning. Instead, we distribute weight matrices among GPUs to take advantage of their available memory. When the size of the model exceeds available GPU memory, matrices must be shuttled between the host and GPU as they are needed during kernel execution, incurring a significant runtime performance penalty called memory thrashing. Thus, the use of multiple GPUs therefore provides additional memory and allows larger models to be simulated without intractable increases in runtime. In addition, the use of half-precision floating points for weight matrices reduces their memory footprint by 50%, allowing further increases in model size.

When a region is contextually gated, a subset of its neurons do not participate in computations, as their output is guaranteed to be zero and their corresponding weights will not be affected by learning. Thus, a naive implementation of matrix operations involves unnecessary computations when contextual gating is active, as compute threads are allocated to deactivated neurons. The percentage of wasted computations depends on the context density parameter, which is typically less than $\frac{1}{2}$ (i.e., more than half of the computations are wasted). We therefore implemented efficient versions of the matrix operation kernels that perform preprocessing to determine which neurons are active and assign them to compute threads accordingly.

Overall, we hypothesized that runtime would be significantly improved by the use of efficient kernels that avoid unnecessary computations for deactivated neurons. In addition, we hypothesized that runtime would scale roughly with model size until GPU memory is exceeded, at which point runtime would rapidly increase due to memory thrashing. Thus, the use of half-precision weights and distribution of the model across GPUs would increase the maximum model size possible without intractable increases in runtime.

3. Results

3.1. Interpreter Test Suite

We first tested NeuroLISP with 37 simple handwritten test cases that exercised the various components of the implemented language to verify the correctness of the virtual interpreter firmware. These tests include constructing and navigating s-expressions and associative arrays, reading and printing s-expressions, logical and conditional operations, function definitions, and variable bindings in nested namespaces (see Appendix J for details). For each test, NeuroLISP was constructed with *mem*, *lex*, and *env* region sizes of 2048, 2048, and 1024, respectively, and an *env* context density parameter $\lambda_{env} = \frac{1}{4}$. With these parameters, the model successfully passed all tests.

3.2. Memory and Variable Binding Capacity

To examine the relationship between program/data memory capacity and the size of the *mem* and *lex* regions, we performed simple tests involving storage and retrieval of lists containing between 10 and 100 symbols randomly drawn from a set of 10 possible symbols. During each trial, a list of symbols was read in as input using the `read` operation, stored in memory, then printed back as output. First, we systematically varied the size of the *mem* region from 300 to 1500 neurons while keeping the *lex* region size constant at 2048 neurons. Then, we did the opposite, testing variations of the *lex* region size from 300 to 1500 while keeping the *mem* region size constant at 2048. In both cases, $\lambda_{env-ctx}$ was fixed at $\frac{1}{4}$. The results are shown in Figure 4, where each data point indicates the percentage of successful trials (y-axis) involving lists of a specific length (x-axis). Each line indicates results for instantiations of the model with the same parameters. Figure 4a shows that, as predicted, the memory capacity of the model scales roughly linearly with the size of the *mem* region, with larger lists requiring a larger *mem* size to be reliably stored and retrieved. This can be seen by noting the gaps between the lines, and the points at which each line diverges from 100% accuracy, indicating the maximum storage capacity for a given *mem* region size (e.g., 600 neurons suffices for a list of 20 elements, 900 neurons for 50 elements, 1200 neurons for 70 elements, and 1500 neurons for 100 elements). This

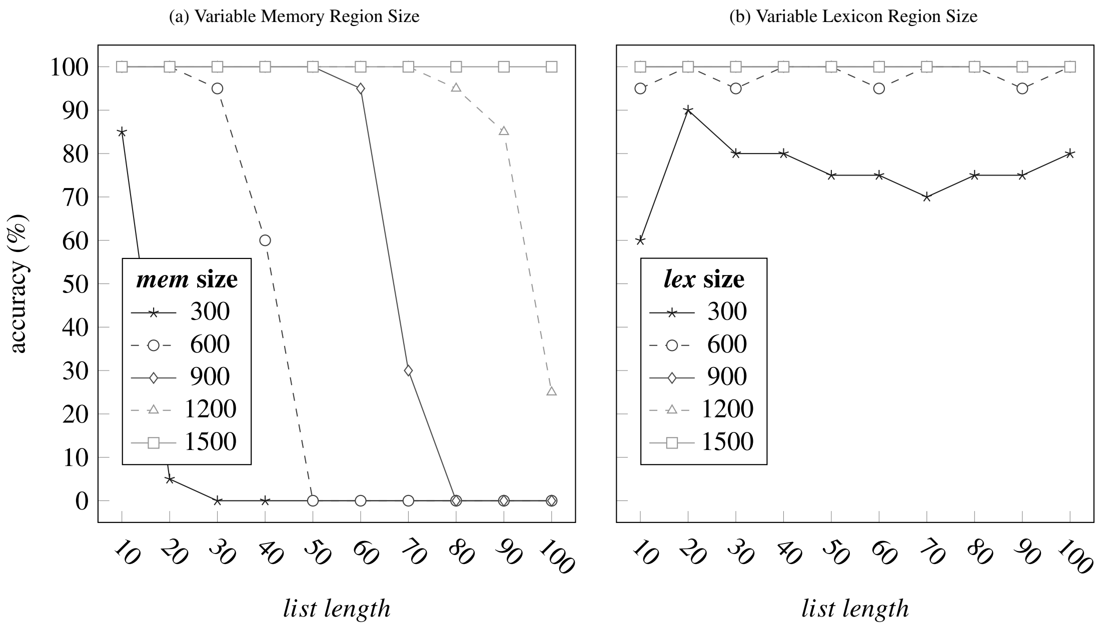

Figure 4: Results for list storage and retrieval with varying *mem* and *lex* region sizes. During each trial, NeuroLISP read in a randomly generated list of symbols of a specified length (x-axis) and stored it in memory, then traversed and printed its contents. A trial was considered successful if the printed list matched the input list. Each datapoint indicates the percentage of successful trials (y-axis) out of 20 for a specified list length (x-axis). (a) With the *lex* region size fixed at 2048 neurons, varying the size of the *mem* region reveals a roughly linear relationship with storage capacity (i.e., successful storage and retrieval of longer lists requires correspondingly larger *mem* region sizes). (b) With the *mem* region size fixed at 2048 neurons, the *lex* region size does not show a linear relationship with storage capacity, but a sufficient *lex* size is necessary for reliable storage. Note that because perfect accuracy is achieved for *lex* sizes of 900 and above, some lines are stacked and are not visible in the plot.

linear relationship is not found in Figure 4b, which nevertheless shows that a sufficiently large *lex* region is necessary to learn the associations between *mem* states and the corresponding *lex* patterns representing stored symbols.

We tested variable binding in two ways that we refer to as *breadth* and *depth* testing. In breadth testing, we used a program that created many bindings with different variable names in the same namespace (Figure 5a). In depth testing, we used a recursive function that creates many bindings with the same variable name in different namespaces (depth refers to depth of recursion; see Figure 5b). We tested these two situations separately because they involve different neurocomputational demands: depth requires learning many attractors with the same context masking pattern, and breadth involves learning many attractors with the same activity pattern, but with different context masks. We hypothesized that $\lambda_{env-ctx}$ would affect these two situations differently. Specifically, a higher $\lambda_{env-ctx}$ would improve depth performance by allowing a greater percentage of the namespace pattern to participate in attractor dynamics, but would reduce breadth performance by increasing the overlap and interference between learned attractors for different variables in the same namespace. This tradeoff was explored in Davis et al. (2021a), in which it was shown that a moderate $\lambda_{env-ctx}$ balanced performance for auto-associative and hetero-associative learning. For both breadth and depth testing, the experiment was repeated for $\lambda_{env-ctx}$ values of $\frac{1}{8}$, $\frac{1}{4}$, and $\frac{1}{2}$.

The results for breadth testing are shown in Figure 6. The size of the *lex* and *mem* regions was fixed at 2048 neurons and 5000 neurons, respectively, while the size of the *env* region was varied from 100 to 600 neurons. For each test, a unique program was generated to create a specific number of variable bindings in the same namespace (using the *setq* operation) before accessing them to print the corresponding values (Figure 5a). Each variable had a unique name, and was bound to a random symbol drawn from a set of 10 possible symbols. As predicted, a higher $\lambda_{env-ctx}$ greatly reduced performance, presumably by increasing interference between learned attractors due to greater overlap.

The results for depth testing are shown in Figure 7. The size of the *lex* and *mem* regions was fixed at 2048 neurons each, while the size of the *env* region was varied from 1000 to 5000 neurons. The test program passed a random list of symbols (drawn from a set of 10 possible symbols) into a recursive function that printed the list in reverse order

(a) Sample Breadth Testing Program

```
(setq v0 'B)
(setq v1 'G)
(setq v2 'F)
(setq v3 'A)
v0
v1
v2
v3
```

(b) Depth Testing Program

```
(progn
  (defun f (x)
    (if x
      (progn
        (f (cdr x))
        (print (car x)))))
  (f (read))
  'NIL)
```

Figure 5: Programs for variable binding capacity testing. (a) Breadth testing involves binding several variables with different names in the same namespace. The sample program shown here binds four variables to random symbols using the `setq` operation, then retrieves them sequentially. NeuroLISP prints the result of evaluating each expression, providing an output sequence to verify correct evaluation. (b) Depth testing involves binding several variables of the same name in different namespaces. The program shown here includes a recursive function that prints a list in reverse. The input list is read in using the `read` operation, and the printed output is compared with the input list to verify correct evaluation.

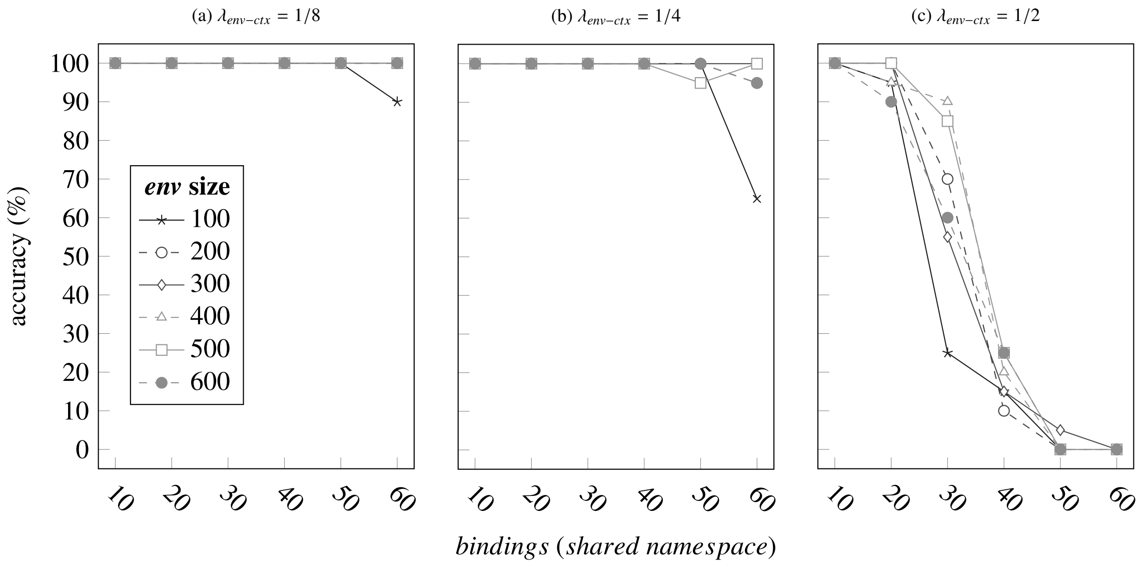

Figure 6: Results for *breadth* testing of variable binding with varying *env* region size and $\lambda_{env-ctx}$. Each test involved binding several variables with unique names in the same namespace using the *setq* operation, then retrieving and printing their values (Figure 5a). The x-axis indicates the number of stored variable bindings, and the y-axis indicates the percentage of successful trials (20 per datapoint). (a) Better performance is achieved with a low $\lambda_{env-ctx}$ of $\frac{1}{8}$, which minimizes the interference between associative learning of distinct variable bindings in the same namespace. (b) Performance deteriorates with a higher $\lambda_{env-ctx}$ of $\frac{1}{4}$. (c) With a high $\lambda_{env-ctx}$ of $\frac{1}{2}$, the model struggles to store large numbers of bindings in the same namespace, even with larger *env* region sizes. Note that some lines in the plots are stacked and are not visible.

(Figure 5b). Importantly, the last symbol in the list was printed by the deepest recursive function call, and variable retrievals were only performed after all bindings were created. The results match our prediction of a linear relationship between *env* region size and the number of bindings that can be successfully stored and retrieved. This can be seen by noting the gaps between the lines in Figure 7a, and the points at which each line diverges from 100% accuracy, indicating the maximum binding capacity for a given *env* region size (e.g., 1000 neurons suffices for 10 bindings, 2000 neurons for 20 bindings, 3000 for 50, 4000 for 60, and 5000 for 80). In addition, the results corroborate our prediction of higher performance with a higher $\lambda_{env-ctx}$.

3.3. Multiway Tree Processing

Having established rough guidelines for sizing the regions of the model, we tested NeuroLISP with a small library of multiway tree processing functions to demonstrate that the model can successfully execute basic LISP programs

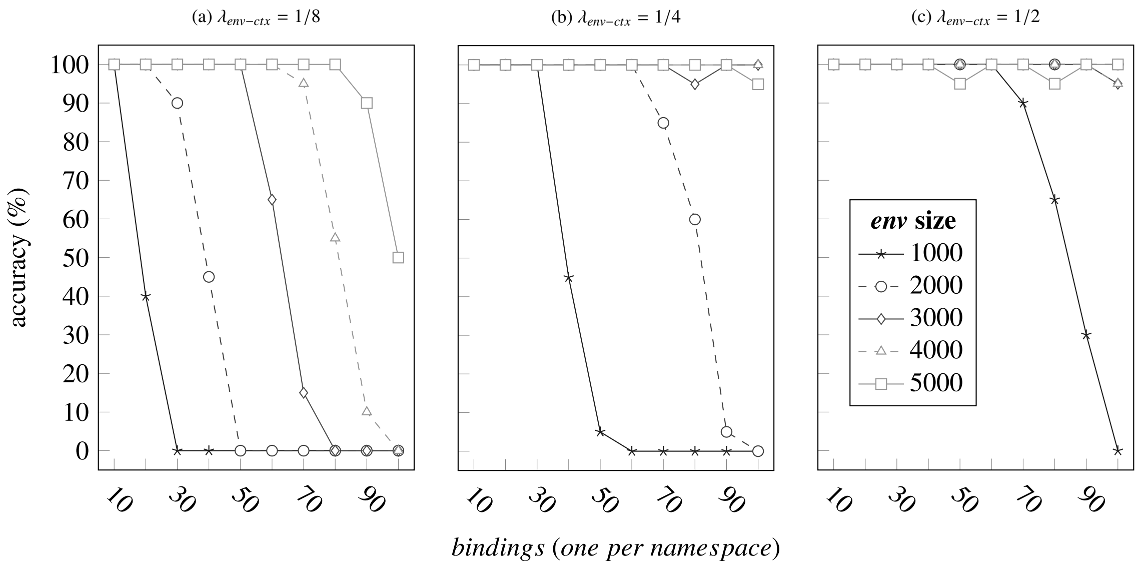

Figure 7: Results for *depth* testing of variable binding with varying *env* region size and $\lambda_{env-ctx}$. Each test involved execution of a recursive function with a single variable, and required storing several bindings with the same variable name in different namespaces (Figure 5b). Because the function uses head recursion, the model was required to store all bindings before retrieving and printing them. The x-axis indicates recursive depth (number of namespaces), and the y-axis indicates the percentage of successful trials (20 per datapoint). (a) With a low $\lambda_{env-ctx}$ of $\frac{1}{8}$, a linear relationship can be seen that resembles that of Figure 4a: more *env* neurons are required to store more bindings. (b) Unlike with breadth testing (Figure 6), a higher $\lambda_{env-ctx}$ of $\frac{1}{4}$ improves binding capacity across namespaces by increasing the number of neurons that participate in *env* region attractor dynamics and improving the discriminability of masked namespace patterns. (c) Increasing $\lambda_{env-ctx}$ to $\frac{1}{2}$ further improves performance. Note that some lines in the plots are stacked and are not visible.

that manipulate complex data structures[2]. A multiway tree is represented as either an atom (leaf node), or a list containing an atom (node label) and one or more multiway trees (children). The implemented functions are listed in Table 2. The full implementation and test cases for the `is-tree?` function are listed in Figure 8 (see Appendix K for remaining functions). Each function was tested with several test cases. For each test, NeuroLISP was constructed with *mem*, *lex*, and *env* region sizes of 6000, 2048, and 1024, respectively, and an *env* context density parameter $\lambda_{env} = \frac{1}{4}$. With these parameters, the model successfully passed all tests.

(a) Implementation of `is-tree?` Function

```
(defun is-tree? (expr)
  (or (atom expr)
      (and (listp expr)
           (atom (car expr))
           (cdr expr)
           (is-forest? (cdr expr)))))
(defun is-forest? (expr)
  (or (not expr)
      (and (is-tree? (car expr))
           (is-forest? (cdr expr)))))
```

(b) Test Cases for `is-tree?` Function

```
(is-tree? 'a)
(is-tree? '(a b))
(is-tree? '(a (b c)))
(is-tree? '(b d e))
(is-tree? '(a (f g) c (b d e)))
(is-tree? '(x y z))
(is-tree? '(a))

(is-tree? '(a (b c) (d) e))
(is-tree? '((a b c) (d e)))
```

Figure 8: (a) Implementation of the `is-tree?` function, which tests if an expression represents a valid multiway tree. (b) Test cases for `is-tree?`. The first seven test cases evaluate valid trees, while the last two evaluate invalid trees.

[2]The multiway tree data structure representation is thanks to Werner Hett's "Ninety-Nine Prolog Problems", and the processing functions are inspired by binary tree processing functions found in the Appendix to Paul Graham's "ANSI Common Lisp" textbook.

Table 2: Multiway tree processing functions implemented in NeuroLISP.

`(expr-equal?  x y)`	Recursively determines whether two s-expressions are equivalent.
`(tree?  expr)`	Determines whether an expression is a valid multi-way tree. A multiway tree is either an atom, or a list containing an atom and 1 or more multiway trees.
`(copy-tree tree)`	Creates a deep copy of a tree.
`(tree-member elm tree)`	Determines whether an atomic element is contained in a tree as a node label.
`(tree-prefix tree)`	Returns a list containing the node labels of a tree in prefix traversal order. The implementation is memory efficient, and only allocates the memory necessary for the final list.
`(tree-subst new old tree)`	Returns a tree that is equivalent to the input `tree`, except any subtrees matching `old` are replaced with `new`. The implementation is memory efficient: memory is only allocated for ancestors of replaced subtrees.
`(tree-sublis subs tree)`	Performs substitutions on the input `tree` using a list of `old`/`new` pairs (`subs`). As with tree-subst, the implementation minimizes memory allocation.

3.4. PCFG SET Compositionality Task

The PCFG SET task is a sequence processing task designed to evaluate compositional learning in machine learning models (Hupkes et al., 2020). The task involves input sequences that specify nested operations performed on strings of symbols. For example:

`append swap F G H , repeat I J` $\longrightarrow$ `H G F I J I J`

where the left side of the arrow indicates the input sequence, and the right side indicates the expected output sequence. This task is particularly challenging because it requires learning several operations that can be arbitrarily composed into complex expressions. Notably, Hupkes et al. (2020) report empirical results demonstrating that several state-of-the-art artificial neural networks struggle to learn the task, including recurrent, convolution-based, and transformer neural networks. These models learn the task in a data-driven fashion, and are trained using large datasets of generated input/output pairs. In contrast, we trained NeuroLISP with one-step learning on LISP functions that implement each operation of the task (Figure 9a), and show that it can successfully compose these functions to solve input sequences encoding nested operations like the example listed above.

We used the NeuroLISP emulator to determine the complexity of PCFG SET test cases, and drew a sample covering a range of memory demands. Because tests with high memory demands require very large models, we filtered out tests to include only those requiring between 250 and 350 memory states, up to 128 namespaces, and up to 64 runtime/data stack states. From the remaining tests, we created two samples. The first included 20 tests from each bin of required memory states (i.e., 20 tests requiring 250-259 memory states, 20 requiring 260-269 states, etc). The second included tests with varying numbers of variable bindings from 20-120 (i.e., 20 tests requiring 20-29 bindings, 20 requiring 30-39 bindings, etc). Although NeuroLISP is capable of implementing a parsing procedure, for simplicity, we preprocessed each test input sequence into a LISP expression by converting it to prefix form and adding quote operations for element sequences (e.g., `"append swap F G H , repeat I J"` becomes `(append (swap '(F G H)) (repeat '(I J)))`).

The results for the first PCFG SET tests are shown in Figure 10. The size of the *lex* and *env* regions was fixed at 2048 and 1024, respectively, and the $\lambda_{env-ctx}$ was set to $\frac{1}{4}$. The *mem* region size was varied from 3000 to 5500. As expected, tests requiring greater numbers of *mem* states required a larger *mem* region size. With a sufficiently sized *mem* region, the model was able to successfully pass all of the tests. Figure 11 shows the results of the second set of PCFG SET tests with varying numbers of variable bindings. Here, the *lex* and *mem* region sizes are fixed at 2048 and 5500, respectively, while the *env* region size was varied from 100 to 600, and $\lambda_{env-ctx}$ was tested at $\frac{1}{8}$, $\frac{1}{4}$, and $\frac{1}{2}$. The best results were achieved with a moderate $\lambda_{env-ctx}$ of $\frac{1}{4}$ (middle plot), which permitted a surprisingly large number of bindings with a small *env* region size: perfect performance was achieved for up to 120 bindings with only 500 neurons. These results support our hypothesis that a moderate $\lambda_{env-ctx}$ balances between depth and breadth requirements for variable binding, providing a reasonable capacity for many variables bound across many namespaces.

(a) PCFG SET Functions

```
(defun append (x y)
    (if x
        (cons (car x)
            (append (cdr x) y))
        y))
(defun prepend (x y) (append y x))
(defun remove_first (x y) y)
(defun remove_second (x y) x)

(defun last (x) (dolist (e x e)))
(defun copy (x) x)

(defun reverse (pre)
    (let ((post NIL))
        (dolist (x pre post)
            (setq post (cons x post)))))

(defun shift (x)
    (append (cdr x) (list (car x))))

(defun swap-helper (first mid)
    (if (cdr mid)
        (cons (car mid)
            (swap-helper first (cdr mid)))
        (list first)))
(defun swap_first_last (x)
    (cons (last x)
        (swap-helper (car x) (cdr x))))

(defun repeat (x) (append x x))
(defun echo (x) (append x (list (last x))))
```

(b) Unification Functions

```
(defun var? (x)
    (and
        (listp x)
        (eq (car x) 'var)))

(defun match-var (var pat subs)
    (cond
        ((and (var? pat)
            (eq var (cadr pat))) subs)
        ((checkhash var subs)
            (unify (gethash var subs)
                pat subs))
        (true (sethash var pat subs))))

(defun unify (pat1 pat2 subs)
    (cond
        ((not subs) subs)
        ((var? pat1)
            (match-var (cadr pat1)
                pat2 subs))
        ((var? pat2)
            (match-var (cadr pat2)
                pat1 subs))
        ((atom pat1)
            (if (eq pat1 pat2) subs NIL))
        ((atom pat2) NIL)
        (true
            (unify (cdr pat1) (cdr pat2)
                (unify (car pat1)
                    (car pat2) subs)))))
```

Figure 9: LISP functions implementing the PCFG SET sequence manipulation functions (a) and first-order unification algorithm (b). During testing, these expressions were broken into sequences of symbols, each of which was translated into the corresponding neural activation pattern in the *lex* region, and fed into NeuroLISP one at a time (Section 2.3.3). These functions were invoked by additional test code (e.g., an expression encoding a PCFG SET or unification test case) that was also fed into NeuroLISP as sequential activation patterns, stored in neural memory, and executed by the virtual interpreter. Finally, the results were printed as a sequence of neural activation patterns, translated back to symbols, and compared with the ground truth of the corresponding test case to determine if the test was successful.

3.5. First-Order Unification

First-order unification is a symbolic matching process that is an integral component of automated reasoning systems such as theorem provers (Russell and Norvig, 2002). Two expressions containing unbound variables can be unified if there exists a set of substitutions for the variables that makes the expressions equivalent. For example, unifying expressions P and Q below yields the listed set of substitutions:

P: `(f (var x) (g b))` Q: `(f a (g (var y)))` Substitutions: {`x` ⟶ `a`, `y` ⟶ `b`}

where `(var x)` indicates a variable named *x*, and the substitution set contains mappings (`variable` ⟶ `value`) that unify the expressions. Previous work has shown that neural networks can be incorporated as components in automated reasoning systems (Irving et al., 2016; Bansal et al., 2019; Rocktäschel and Riedel, 2017), and that expression-specific neural networks with local representations can perform unification with error-correction learning (Komendantskaya, 2011). Here we show that NeuroLISP can learn to perform first-order unification on arbitrary expressions using a fixed architecture and distributed representations. We trained NeuroLISP with a unification algorithm (Figure 9b) based on that presented in Russell and Norvig (2002), and show that it works on test cases with randomly generated nested expressions.

Unification test cases were produced by randomly generating trees and converting them to s-expressions (see Appendix L for details). We experimentally varied the complexity of these expressions by varying the number of nodes in the randomly generated trees from 6 nodes to 14 nodes. Although the stochastic process introduced variations

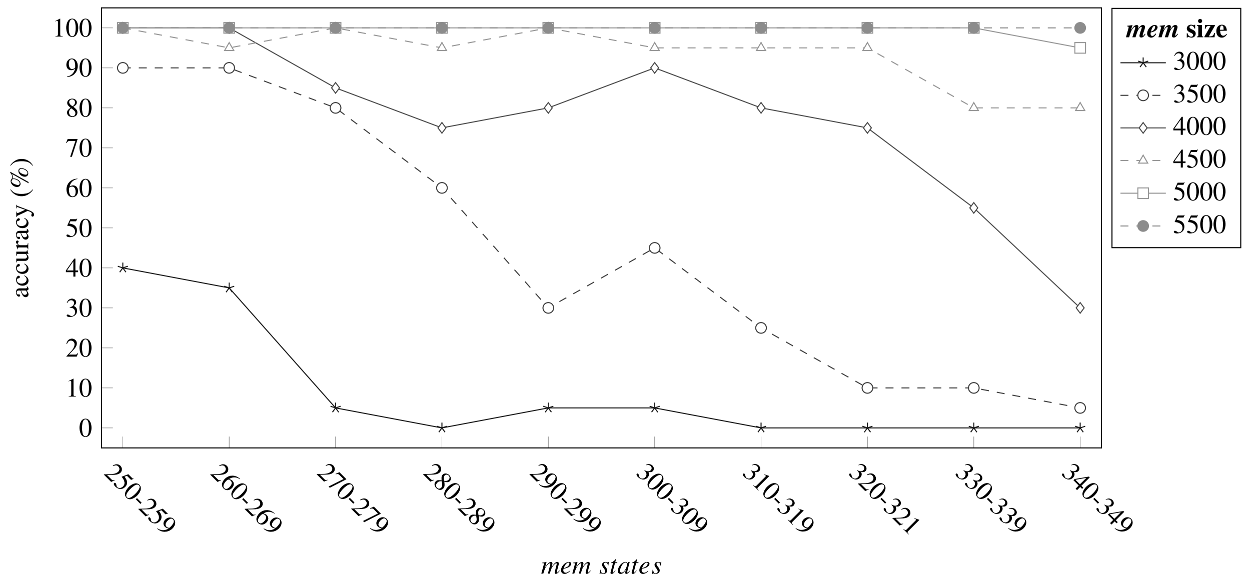

Figure 10: Results for PCFG SET testing with varying *mem* region size. Each test involved reading in an s-expression indicating a composition of symbolic sequence manipulations, executing the indicated functions, and printing the resulting sequence. We labeled and binned each test according to the number of memory states (*mem* attractors) it required based on an emulator for the NeuroLISP architecture, and sampled 20 tests per bin (i.e., 20 tests requiring 250-259 memory states, etc). The x-axis indicates the bin, and the y-axis indicates the percentage of successful trials for tests in each bin. As expected, model performance is contingent upon adequate *mem* region sizing; with a sufficient size, the model achieved perfect performance.

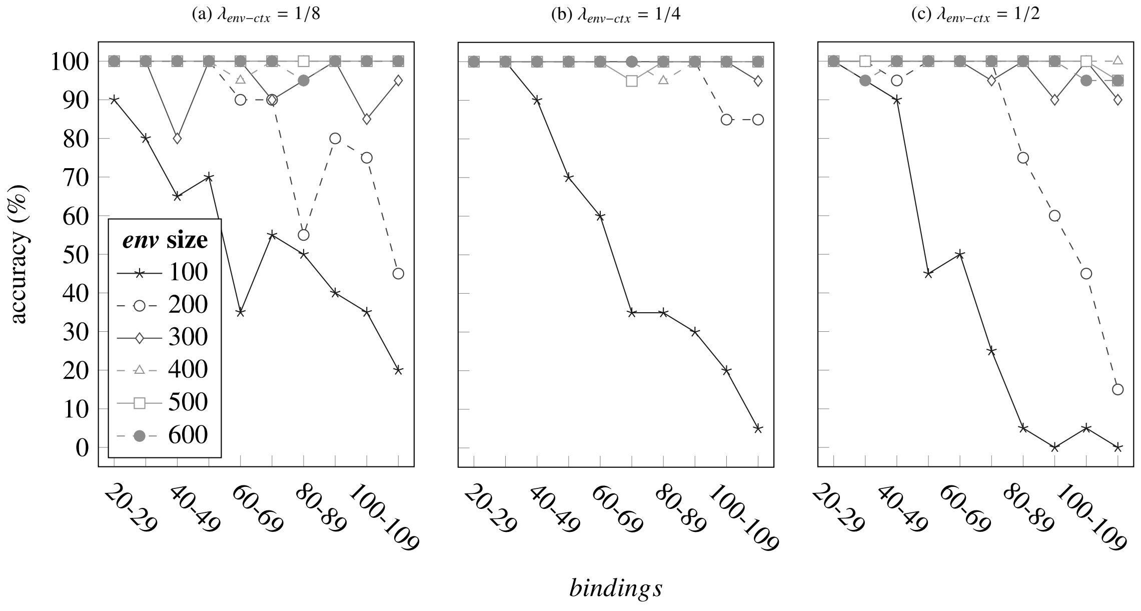

Figure 11: Results for PCFG SET testing with varying *env* region size and *env* context density. As in Figure 10, each test involved reading in an s-expression indicating a composition of symbolic sequence manipulations, executing the indicated functions, and printing the resulting sequence. We labeled and binned each test according to the number of variable bindings it required based on an emulator for the NeuroLISP architecture, and sampled 20 tests per bin (i.e., 20 tests requiring 20-29 variable bindings, etc). The x-axis indicates the bin, and the y-axis indicates the percentage of successful trials for tests in each bin. The best performance is achieved with a moderate context density of $\frac{1}{4}$ that balances the differing demands of storing several variable bindings within (Figure 6) and across (Figure 7) namespaces.

in the size of the final trees due to variable substitutions, the number of starting nodes provides a rough estimate of the complexity of a test case. We generated 20 test cases per initial tree size (expression complexity), and tested the model as above with a) varying *mem* region sizes and b) varying *env* region sizes and $\lambda_{env-ctx}$. In 20% of the test cases, one input expression was mutated to induce a mismatch during unification, and the model was expected to indicate that the expressions could not be unified.

Figure 12 shows the results of unification testing with variable *mem* region sizes. The *lex* and *env* region sizes were fixed at 2048 and 1024 neurons, respectively, and $\lambda_{env-ctx}$ was set to $\frac{1}{4}$. The *mem* region size was varied from 3000 to 4500 neurons. Perfect results were achieved with 4500 neurons. Figure 13 shows the results with variable *env* region sizes (100-600 neurons) and *env* context densities ($\frac{1}{8}$, $\frac{1}{4}$, and $\frac{1}{2}$). Here the *lex* and *mem* region sizes were fixed at 2048 and 5500 neurons. As with the PCFG SET tests, a relatively small *env* size was sufficient for accurate performance on relatively complex test cases. However, the best results were achieved with a smaller $\lambda_{env-ctx}$ of $\frac{1}{8}$. We suspect that this has to do with differences in the PCFG SET and unification programs: the string manipulation functions of the PCFG SET required deeper recursion, and therefore suffered more from smaller $\lambda_{env-ctx}$. These results highlight the trade-off involved with $\lambda_{env-ctx}$: the optimal parameter value depends on the demands of the programs that the model is running.

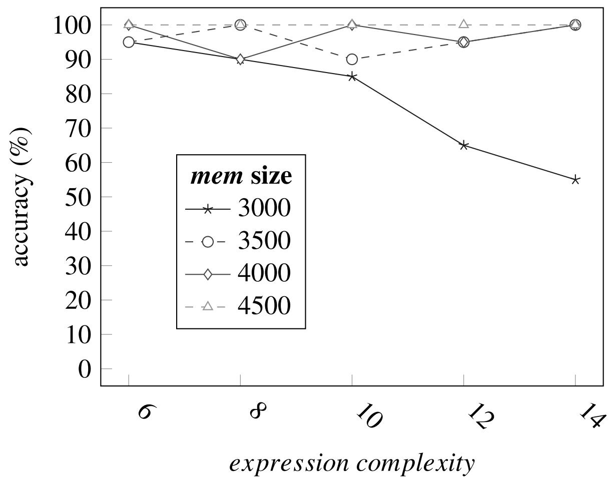

Figure 12: Results for first-order unification testing with varying *mem* region size. Each test involved reading in two s-expressions representing patterns with variables, performing unification on the patterns, and printing the resulting substitutions if the unification was successful. The expressions for each test case were randomly generated using an initial complexity parameter (x-axis; see Appendix L). The y-axis indicates the percentage of successful trials for each complexity parameter setting (20 trials per datapoint). Successful execution required a sufficiently sized *mem* region to meet the memory demands of unifying complex expressions.

3.6. Runtime Performance and Scalability

While the above experiments specifically address model performance in terms of symbolic behavior, here we evaluate how region sizing affects runtime and memory usage, and investigate approaches to improving the efficiency of its implementation. We test the model with varying `mem` and `env` region sizing using a program that includes a simple recursive function (Figure 14). This program is first parsed in its entirety, requiring no modifications to variable bindings or namespaces, and therefore no computations that involve the `env` region and its connectivity. After parsing, the program is executed, which involves both memory access and the utilization of variable bindings and namespaces. We report these two runtimes separately to show that scaling a region only affects the runtime performance of relevant model computations.

Tests were performed using `mem` region sizes varying from 10,000 to 60,000 (`env` size fixed at 10,000) and `env` region sizes varying from 10,000 to 70,000 (`mem` size fixed at 10,000). The `lex` region size was fixed at 2048, and the context density parameters for the `mem` and `env` regions was set to $\frac{1}{4}$. These tests were repeated using different implementation configurations:

- One or two GPUs. For two GPUs, the weight matrices for the model were distributed between GPUs using a greedy algorithm, in which the next largest matrix is assigned to the GPU with the most available memory.

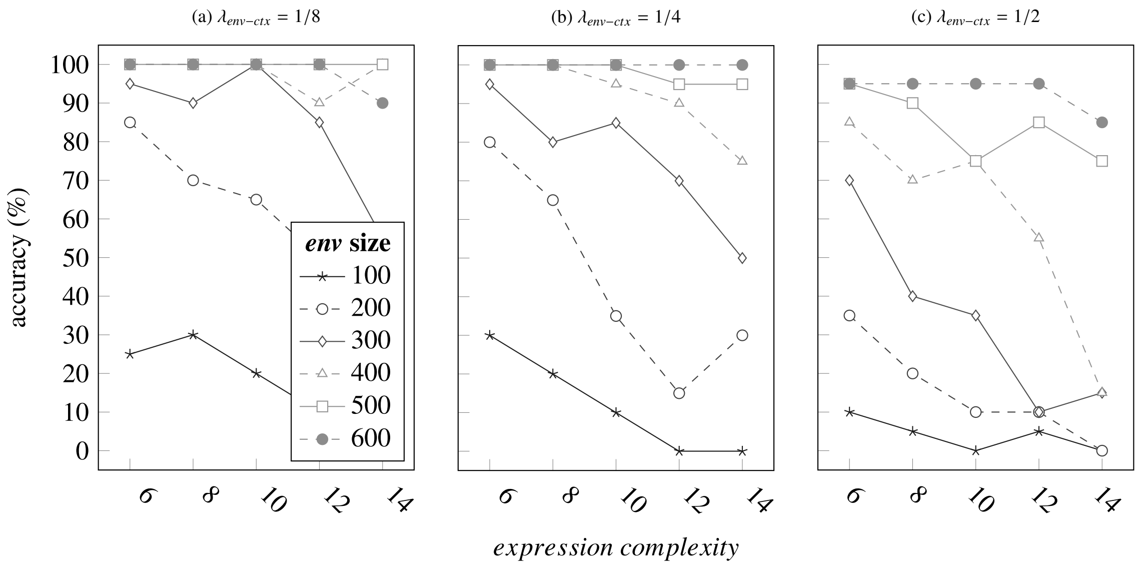

Figure 13: Results for first-order unification testing with varying *env* region size and *env* context density. As in Figure 12, each test involves performing unification on randomly generated expressions and printing substitutions if unification succeeds. The x-axis indicates the complexity of the test expressions (see Appendix L), and the y-axis indicates the percentage of successful trials per complexity parameter setting (20 trials per datapoint). Unlike with PCFG SET testing, slightly better performance is achieved with a low context density of $\frac{1}{8}$. Because low context densities favor larger numbers of bindings within a namespace (Figure 6), we suspect that this is due to differences between the PCFG SET and unification algorithms: the former requires deeper recursion with fewer variables per namespace than the latter.

```
(progn
    (print 'executing)
    (defun f (x)
        (if x (f (cdr x))))
    (f '(a b c d e f g h i j))
    'complete)
```

Figure 14: Program used to evaluate runtime and memory performance. NeuroLISP first parses the entire program and stores it in memory, which requires no modifications to variable bindings or namespaces, and therefore no computations that involve the `env` region. Then, it evaluates the program, executing a recursive function that creates several namespaces and variable bindings. The timing of the printed outputs ("executing" and "complete") indicates the runtimes of parsing and execution, respectively.

Matrix operations for a connection were executed on the GPU holding its weight matrix, and connections were computed one at a a time.

- Single (four byte) or half (two byte) floating-point precision for connection weights. Half-precision weights did not noticeably impact the symbolic behavior of the model, but cut memory usage in half. Thus, larger models could be executed without memory saturation.
- Simple (slow) or efficient (fast) kernels for contextually-gated connection computations. Fast kernels use pre-processing to assign only active neurons and synapses to compute threads, while slow kernels use a naive implementation that does not skip deactivated neurons and synapses.

Raw runtime and memory usage is reported in Figure 15. Program parsing and execution are reported separately in the first and second rows. Runtime scales with the size of the model until GPU memory is saturated, at which point runtime spikes significantly due to memory thrashing. This can be seen in the parsing runtime plot (top left); runtime spikes at different points depending on the number of available GPUs and the floating-point precision used, as these affect the maximum available memory and the total memory used by the model. As expected, parsing runtime was not affected by `env` region sizing (top right), as the corresponding neurons and weights are not needed during parsing.

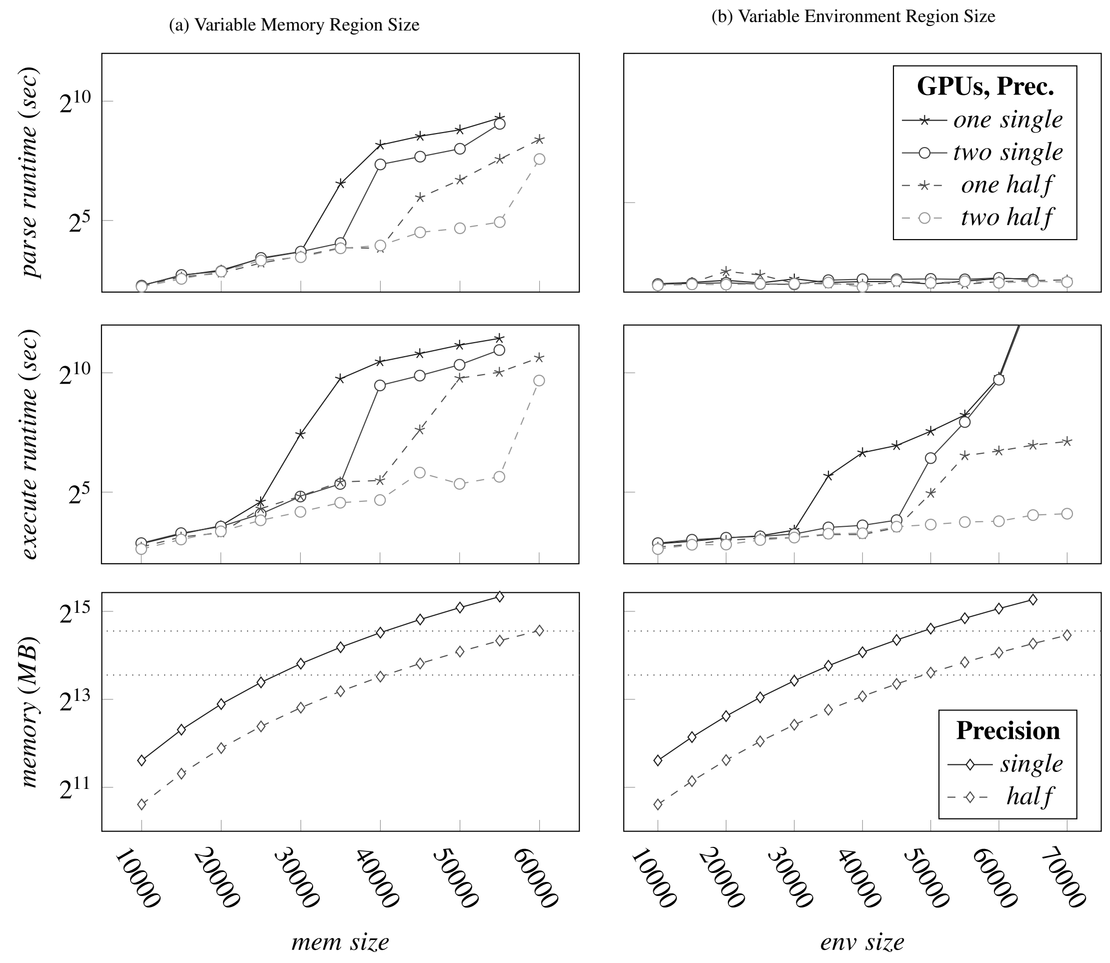

Figure 15: Model runtime and memory usage (y-axes) with increasing `mem` and `env` region sizing (x-axes). Each line indicates performance for a particular combination of GPU count and floating-point precision for connection weights. Solid lines indicate single-precision weights (four bytes per weight) and dashed lines indicate half-precision weights (two bytes per weight). Each precision configuration is tested with both a single and dual GPU setup. The first row shows runtime for program parsing, which does not involve `env` region computations. Thus, as the `env` region is scaled, parsing runtime does not change, as the corresponding weights are not needed in GPU memory. The second row indicates runtime for program execution, which involves both `mem` and `env` region computations. Runtime scales relative to model size until GPU memory is saturated, at which point runtime increases significantly due to memory thrashing (see Figure 16). Total available GPU memory is indicated by dotted horizontal lines in the third row plots (12000MB for one GPU or 24000MB for two GPUs).

Thus, although memory for `env` region connections exceeds the maximum available GPU memory, it is not accessed during parsing, and does not affect runtime. During execution, however, runtime scales with both `mem` and `env` region sizing (middle row). The third row indicates memory usage, which only differs with floating-point precision. The total available GPU memory is indicated by the dotted horizontal lines (12000 megabytes (MB) for one GPU and 24000MB for two GPUs). The points where memory usage crosses these lines correspond to the spikes in runtime in the top two rows; runtime spikes at 12000MB with single GPU configurations and 24000MB with dual GPU configurations.

Figure 16 shows total runtime (parsing and execution combined) relative to memory usage, reported as seconds per MB. Relative runtime slightly decreases as model size increases, likely because the overhead of CPU computations and CUDA kernel dispatch is washed out by increasingly large kernel execution times. The rate of decrease is more significant with `env` region scaling (right plot) because `env` region computations are much less common than `mem` region computations. Once memory is saturated, relative runtime increases sharply in both plots. Although relative performance appears to level off, single-precision performance for high `env` region sizing (right side) indicates that the plateau does not persist; once the model becomes large enough, the relative performance quickly reaches intractable

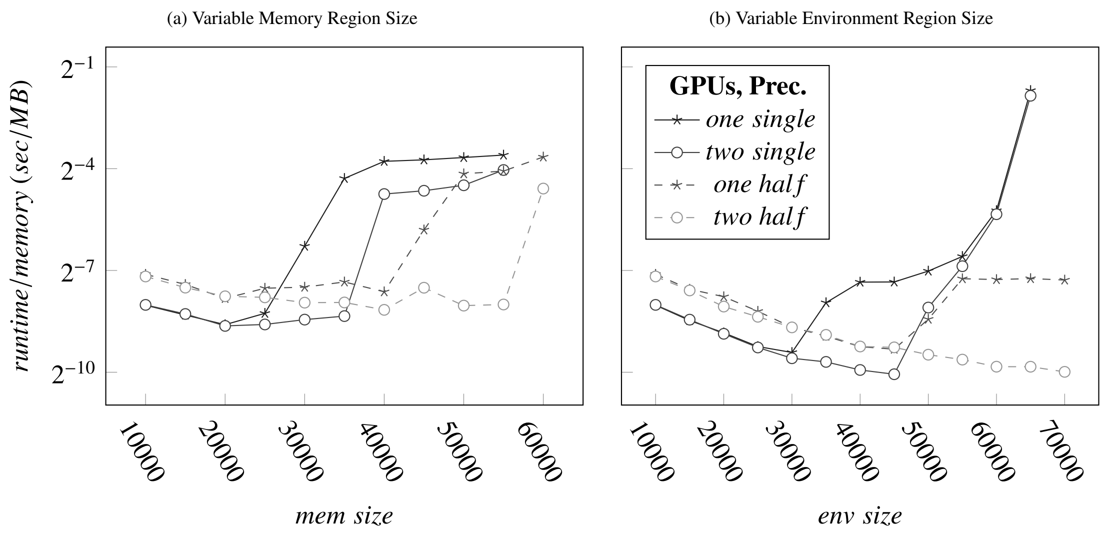

Figure 16: Model runtime relative to memory usage (y-axis) with increasing `mem` and `env` region sizing (x-axes). As in Figure 15, each line indicates performance for a particular combination of GPU count and floating-point precision for connection weights. Here the total runtime, including both program parsing and execution, is reported relative to memory usage (seconds per MB). Relative runtime decrease slightly as the overhead of CPU computations and CUDA kernel dispatch is washed out by increasingly large sizes for connection weight matrices. Once GPU memory is saturated (see third row of Figure 15), performance degrades significantly, and each weight incurs a more significant penalty on runtime.

levels, and the GPUs spend most of their time performing memory transfers.

Finally, Figure 17 show the performance benefits gained by the use of fast kernels for contextually-gated connection computations. Results are reported for single-GPU configurations with either single or half precision weights and either slow or fast kernels. The first row shows the total runtime (top left) and the corresponding speedup gained from using fast kernels over slow kernels (top right). Because not all computations involve contextual gating, the overall performance gain is relatively low, and peaks near 2x for moderately sized models. Thus, we examined computations for the hetero-associative recurrent connection of the `mem` region, which involves contextual gating. The second row shows runtime (middle left) and speedup (middle right) for learning in this connection, which involves both reads and writes for weights. Speedup peaks near 4x with half-precision weights, which corresponds to the fraction of neurons activated by contextual gating. The third row shows runtime (bottom left) and speedup (bottom right) for input activation in this connection, which only involves reads for weights. Here the peak speedup reaches just over 8x for half-precision weights. These results indicate that memory access is a major bottleneck for performance, and that contextual gating provides significant performance increases. This suggests that more widespread use of contextual gating in the model would provide greater increases in performance.

4. Discussion

We have presented NeuroLISP, a multi-region recurrent neural network that implements a virtual interpreter for a dialect of the LISP programming language. The network's architecture is composed of program-independent circuitry that learns both interpreter functions and LISP programs using one-step associative learning, and is capable of flexible reprogramming without architectural changes. To our knowledge, this is the first effort to implement a high-level functional programming language in a fixed neural architecture with distributed representations. NeuroLISP is most closely related to previous programmable attractor neural networks like the Neural Virtual Machine (Katz et al., 2019) and GALIS (Sylvester and Reggia, 2016), which include program-independent circuitry, distributed representations, and local one-step associative learning. However, NeuroLISP includes several novel features that improve its computational capabilities. Most significantly, NeuroLISP implements an interpreter for a high-level programming language (LISP) that supports nested expressions rather than an assembly-like language with sequential instructions. This makes it easy to express complex algorithms like those used in traditional symbolic AI. The network's shared

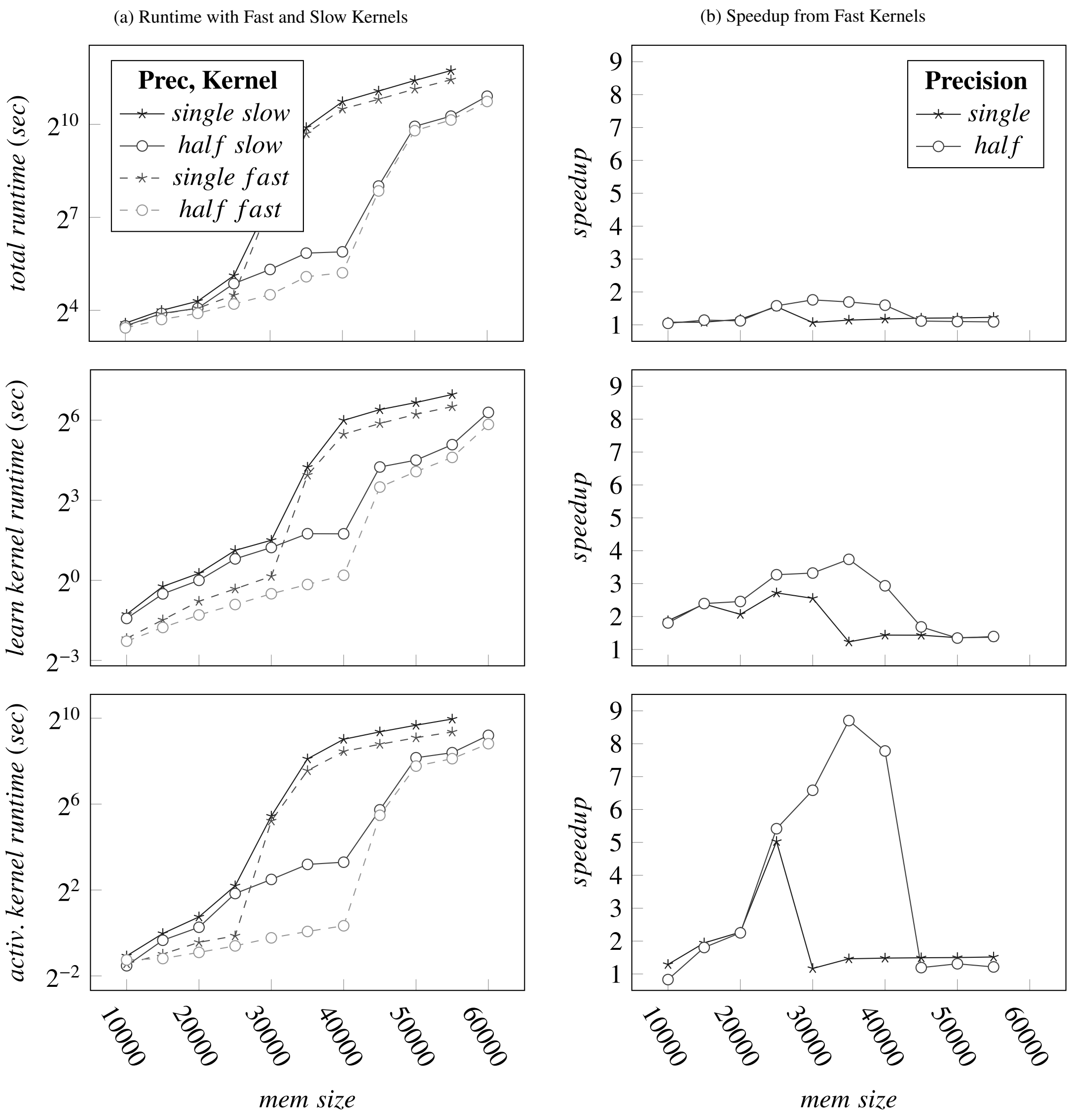

Figure 17: Performance benefits from using efficient matrix computation kernels for contextually-gated connection computations. As in Figures 15 and 16, each line indicates performance for a particular combination of GPU count and floating-point precision for connection weights. Solid lines indicate simple kernels that naively allocate deactivated neurons and weights to compute threads, while dashed lines indicate efficient kernels that perform preprocessing to assign only active neurons and weights (see Section 2.4 for details). Note that the context density parameter for these tests was set to $\frac{1}{4}$, and contextually-gated connection computations therefore involve only $\frac{1}{4}$ of the neurons in the target region, and $\frac{1}{16}$ of the total incomping weights. The first row indicates the overall runtime (top left) along with the relative speedup gained by using efficient kernels (top right). Because only a subset of connection computations involve contextual gating, the overall speedup remains relatively low, peaking near 2x for moderately sized models. The second row indicates the cumulative runtime for learning in the hetero-associative recurrent connection of the `mem` region (middle left) and the corresponding speedup (middle right). Speedup peaks near 4x for moderately sized models. Finally, the third row indicates cumulative runtime for activation in the hetero-associative recurrent connection of the `mem` region (bottom left) and the corresponding speedup (bottom right). Unlike learning, which involves both reading and writing of connection weights, activation only involves reads, and is therefore less intensive. As a result, speedup peaks just above 8x for moderately sized models.

program/data memory region stores compositional data structures as attractor graphs that can be constructed, manipulated, and accessed via top-down gating during program execution (Davis et al., 2021a). Notably, this includes programs themselves; programs can be treated as data, and generated in memory by other programs or by sequential inputs that represent code. Finally, NeuroLISP supports function definitions and variable binding with scoping rules that are determined by the learned interpreter firmware, facilitating program modularity and reuse.

Compared with other programmable neural networks, NeuroLISP is highly flexible and extensible due to its program-independent architecture and fully distributed representations. New programs and interpreter functions can be learned with fast associative learning without adding new neurons/connections or retraining the model on previously learned behaviors. This is in contrast to approaches that involve compiling programs into specialized neural circuits with local representations (Bunel et al., 2016; Neto et al., 2003), storing programs in segregated sub-populations of a RAM-like memory matrix (Bošnjak et al., 2017; Reed and De Freitas, 2015), or performing neural program induction with iterative gradient descent learning (Graves et al., 2016, 2014; Zaremba and Sutskever, 2014). NeuroLISP also features purely-neural mechanisms for procedures that are sometimes performed by non-neural components in hybrid systems, such as call-stack management (Reed and De Freitas, 2015), storage and manipulation of structured memories (Silver et al., 2016; Bunel et al., 2018), and coordinating the flow of information through and between neural circuits (Andreas et al., 2016). All of these mechanisms in NeuroLISP are controlled by learned activity in the Controller regions of the model, and can therefore be reprogrammed in various ways to modify the virtual interpreter without architectural changes.

NeuroLISP also differs from prior models in that its working memory is based on learned attractor dynamics rather than persistent activity patterns, and it achieves temporal locality without specialized RAM-like memory matrices. Models based on neural attention are typically provided with simultaneous access to a sequence of input patterns (temporal non-locality) (Vaswani et al., 2017; Bieber et al., 2020), and/or selectively read and write to a large array of neurons that maintain activity patterns in segregated neural "addresses" (Santoro et al., 2016; Graves et al., 2016; Reed and De Freitas, 2015), neither of which is considered biologically plausible. In contrast, NeuroLISP processes inputs one at a time, stores them in memory using fast associative learning, and retrieves them through top-down control of attractor dynamics. This means that NeuroLISP's working memory does not require copying activity patterns between neural regions, or complex mechanisms for memory allocation, garbage collection, or indexing schemes for data structures (e.g., usage vectors, temporal link matrices; Graves et al. (2016)). Instead, new memories are created by randomly generating activity patterns, establishing them as attractors using one-step auto-associative learning, and linking them directly to other memories using one-step hetero-associative learning (i.e., attractor graphs). The organization of data structures in memory is based on learned algorithmic behaviors contained in the interpreter firmware. Thus, NeuroLISP features improvements to both the static and dynamic aspects of working memory: its memory region natively supports attractor graphs, and its Controller regions learn specific procedures for organizing them into compositional data structures.

Our computational experiments demonstrate the correctness of the NeuroLISP interpreter and show that it can learn to successfully execute several non-trivial programs, including functions that operate on complex derived data structures (multiway trees). The results for the PCFG SET task show that NeuroLISP readily learns string manipulation operations that can be composed into nested expressions, a significant challenge for state-of-the-art neural networks, including recurrent, convolution-based, and transformer networks (Hupkes et al., 2020). We also showed that NeuroLISP can successfully implement first-order unification, a high-level symbolic AI task that is an integral component of automated reasoning systems (Russell and Norvig, 2002). To our knowledge, unification with neural networks has only previously been attempted using expression-specific architectures and local representations (Komendantskaya, 2011), whereas NeuroLISP learns to unify arbitrary expressions using a fixed architecture and distributed representations. In accordance with prior work on the memory capacity of attractor neural networks (Amit et al., 1985; Davis et al., 2021a; Katz et al., 2019), our results show that the model's storage capacity for data structures and variable bindings is linearly dependent upon the size of its memory regions.

Performance testing with parallel processors indicates that simulation runtime scales with memory usage until penalties are introduced by memory thrashing, which occurs when available device memory is exceeded. Because NeuroLISP's architecture is modular, its components can be effectively distributed to different compute devices. In addition, the use of half floating-point precision over single-precision reduces memory usage without affecting symbolic behavior, allowing larger region sizes to be simulated on the same hardware. Finally, contextually-gated connections can be computed more efficiently by allocating only active neurons and synapses to compute threads, and improves runtime efficiency in a way that is similar to architectural modularity. This suggests that more widespread use of contextual gating might lead to further performance gains by reducing the computational load of each simulation timestep.

NeuroLISP is not meant to be a veridical model of the human brain, but several aspects of its architecture and dynamics are inspired by neuroanatomy. Its "region-and-pathway" architecture (Sylvester and Reggia, 2016) is inspired by the organization of the cerebral cortex, and includes several recurrent regions with heterogenous functions. Interactions among these regions are controlled by top-down gating signals that are reciprocally dependent upon regional dynamics. This resembles the functional dynamics of the basal ganglia, which are guided in part by cortical activity and provide top-down control of cortical interactions via modulation of the thalamus (Parent and Hazrati, 1995; Dudman and Gerfen, 2015). In particular, we suggest that the *gate sequence* and *gate output* regions of NeuroLISP's Controller subnetwork represent striatal and pallidal circuitry, while the remaining regions represent various subregions of the prefrontal cortex. Although NeuroLISP does not include sensory or motor circuitry, it could be readily extended to include regions representing sensory and motor cortices, along with corresponding subcortical regions such as the tectum, cerebellum, and motor thalamus.

4.1. Limitations and Future Work

NeuroLISP is limited by its short-term memory capacity and lack of long-term memory. If memory becomes overloaded, learned programs are prone to corruption as memory is updated during program execution and subsequent learning of new programs. This phenomenon, known as *catastrophic forgetting*, is a pervasive issue in artificial neural networks that affects both short-term and long-term memory (Reggia et al., in press; Parisi et al., 2019; Kemker et al., 2018). Although NeuroLISP is subject to catastrophic forgetting, its memory retention may be improved by enrichment of synaptic structure and behavioral strategies. For example, memories may be refreshed via rehearsal (Atkinson et al., 2021) and subject to representational drift to reduce interference between memories (Rule et al., 2019). On a structural level, synapses may be augmented with history-dependent transitions in plasticity, as in cascade models (Fusi et al., 2005), or multiple interacting weights with heterogeneous time constants and learning rates (Benna and Fusi, 2016; Kirkpatrick et al., 2017).

Another limitation of NeuroLISP is that it currently deals exclusively with symbolic processing, and lacks circuitry for low-level perception and action. Prior work has shown that sensory-motor circuits are readily incorporated into programmable neural networks (Davis et al., 2021b; Sylvester and Reggia, 2016), which provide the top-down control typically afforded by non-neural symbolic algorithms in hybrid models. Future work might therefore involve the addition of sensory and motor networks that allow the model to run on robotic hardware that interacts directly with realtime multi-modal environments. The flexibility of the model makes such an extension straightforward, as it only requires additional gating neurons for new regions and pathways, as well as new learned interpreter functions for sensory attention and motor control.

Finally, NeuroLISP currently only learns to execute human-authored programs, and does not learn directly from input/output examples (i.e., program induction). However, its shared program/data memory space makes it possible to learn algorithms for inducing and synthesizing programs directly in neural memory, and more generally suggests future work on alternative learning paradigms such as neuroevolution (Stanley et al., 2019; Such et al., 2017), reinforcement learning (Katz et al., in review, 2020), and imitation learning (Osa et al., 2018; Burke et al., 2019; Katz et al., 2017; Le et al., 2018). These approaches may benefit from the addition of sensory and motor circuitry, and contribute to stabilizing memory and resolving catastrophic forgetting.

Despite its limitations, NeuroLISP provides a promising framework for future research. Its implementation of a high-level programming language with compositional expressions makes it much easier to encode complex behaviors that are difficult to express in low-level assembly languages. Thus, NeuroLISP can replace the non-neural components of hybrid models to create purely-neural systems that integrate high-level cognitive reasoning with the low-level processing that traditional neural networks excel at. Such hybrid models include neural-guided search algorithms as well as hybrid robotic learning systems that use symbolic reasoning to guide neural sensory-motor processing (Katz et al., 2017; Reggia et al., 2018). Conversion to purely-neural modeling facilitates future work on leveraging neural learning to adapt and refine cognitive algorithms based on experience.

4.2. Conclusions

We conclude that high-level programming constructs can be incorporated into neural models to significantly advance their cognitive abilities. NeuroLISP presents a proof of concept that neural networks can implement cognitive algorithms that are typically implemented using symbolic programming techniques, including compositional logical reasoning. Our model is therefore an effective neurocognitive controller that can replace the non-neural components of hybrid models, promoting seamless integration of top-down cognitive control with the strengths of contemporary machine learning. NeuroLISP's working memory system is based on biologically-inspired principles such as temporally-local control of dynamical attractors (distributed representations), fast associative learning, and top-down gating, and is therefore also relevant to interdisciplinary researchers in neuroscience and cognitive science as well as artificial intelligence. Future work should address the control of sensory-motor dynamics in cognitive-robotic systems, as well as experience-based adaptation and refinement of cognitive procedures using methods such as reinforcement learning, program synthesis, and imitation learning.

Acknowledgements

This work was supported by ONR award N00014-19-1-2044.

References

Amit, D.J., 1992. Modeling brain function: The world of attractor neural networks. Cambridge university press.

Amit, D.J., Gutfreund, H., Sompolinsky, H., 1985. Storing infinite numbers of patterns in a spin-glass model of neural networks. Physical Review Letters 55, 1530.

Andreas, J., Rohrbach, M., Darrell, T., Klein, D., 2016. Neural module networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 39–48.

Atkinson, C., McCane, B., Szymanski, L., Robins, A., 2021. Pseudo-rehearsal: Achieving deep reinforcement learning without catastrophic forgetting. Neurocomputing 428, 291–307.

Ba, J., Hinton, G.E., Mnih, V., Leibo, J.Z., Ionescu, C., 2016. Using fast weights to attend to the recent past, in: Advances in Neural Information Processing Systems, pp. 4331–4339.

Baddeley, A.D., 2002. Is working memory still working? European Psychologist 7, 85–97.

Bahdanau, D., Cho, K., Bengio, Y., 2015. Neural machine translation by jointly learning to align and translate, in: 3rd International Conference on Learning Representations, ICLR 2015.

Bansal, K., Loos, S., Rabe, M., Szegedy, C., Wilcox, S., 2019. Holist: An environment for machine learning of higher order logic theorem proving, in: International Conference on Machine Learning, PMLR. pp. 454–463.

Barbosa, J., Stein, H., Martinez, R., Galan, A., Adam, K., Li, S., Valls-Solé, J., Constantinidis, C., Compte, A., 2019. Interplay between persistent activity and activity-silent dynamics in prefrontal cortex during working memory. bioRxiv , 763938.

Benna, M.K., Fusi, S., 2016. Computational principles of synaptic memory consolidation. Nature neuroscience 19, 1697–1706.

Bieber, D., Sutton, C., Larochelle, H., Tarlow, D., 2020. Learning to execute programs with instruction pointer attention graph neural networks. arXiv preprint arXiv:2010.12621 .

Bošnjak, M., Rocktäschel, T., Naradowsky, J., Riedel, S., 2017. Programming with a differentiable forth interpreter, in: International conference on machine learning, PMLR. pp. 547–556.

Bunel, R., Hausknecht, M., Devlin, J., Singh, R., Kohli, P., 2018. Leveraging grammar and reinforcement learning for neural program synthesis. arXiv preprint arXiv:1805.04276 .

Bunel, R.R., Desmaison, A., Mudigonda, P.K., Kohli, P., Torr, P., 2016. Adaptive neural compilation. Advances in Neural Information Processing Systems 29, 1444–1452.

Burke, M., Penkov, S., Ramamoorthy, S., 2019. From explanation to synthesis: Compositional program induction for learning from demonstration. arXiv preprint arXiv:1902.10657 .

Chella, A., Frixione, M., Gaglio, S., 2008. A cognitive architecture for robot self-consciousness. Artificial intelligence in medicine 44, 147–154.

Colom, R., Rebollo, I., Palacios, A., Juan-Espinosa, M., Kyllonen, P.C., 2004. Working memory is (almost) perfectly predicted by g. Intelligence 32, 277–296.

Conway, A.R., Kane, M.J., Engle, R.W., 2003. Working memory capacity and its relation to general intelligence. Trends in Cognitive Sciences 7, 547–552.

Davis, G.P., Katz, G.E., Gentili, R.J., Reggia, J.A., 2021a. Compositional memory in attractor neural networks with one-step learning. Neural Networks 138, 78–97. doi:https://doi.org/10.1016/j.neunet.2021.01.031.

Davis, G.P., Katz, G.E., Soranzo, D., Allen, N., Reinhard, M.J., Gentili, R.J., Costanzo, M.E., Reggia, J.A., 2021b. A neurocomputational model of posttraumatic stress disorder, in: 2021 10th International IEEE/EMBS Conference on Neural Engineering (NER), IEEE. pp. 107–110.

D'Esposito, M., Postle, B.R., 2015. The cognitive neuroscience of working memory. Annual review of psychology 66.

Dudman, J.T., Gerfen, C.R., 2015. The basal ganglia, in: The rat nervous system. Elsevier, pp. 391–440.

Edin, F., Klingberg, T., Johansson, P., McNab, F., Tegnér, J., Compte, A., 2009. Mechanism for top-down control of working memory capacity. Proceedings of the National Academy of Sciences 106, 6802–6807.

Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt, S., 2015. The racket manifesto. 1st Summit on Advances in Programming Languages , 113.

Fusi, S., Drew, P.J., Abbott, L.F., 2005. Cascade models of synaptically stored memories. Neuron 45, 599–611.

Galassi, A., Lippi, M., Torroni, P., 2020. Attention in natural language processing. IEEE Transactions on Neural Networks and Learning Systems , 1–18doi:10.1109/TNNLS.2020.3019893.

Garcez, A.d., Besold, T.R., De Raedt, L., Földiak, P., Hitzler, P., Icard, T., Kühnberger, K.U., Lamb, L.C., Miikkulainen, R., Silver, D.L., 2015. Neural-symbolic learning and reasoning: contributions and challenges, in: 2015 AAAI Spring Symposium Series.

Graves, A., Wayne, G., Danihelka, I., 2014. Neural turing machines. arXiv preprint arXiv:1410.5401 .

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S.G., Grefenstette, E., Ramalho, T., Agapiou, J., et al., 2016. Hybrid computing using a neural network with dynamic external memory. Nature 538, 471–476.

Hickey, R., 2008. The clojure programming language, in: Proceedings of the 2008 symposium on Dynamic languages, pp. 1–1.
Hopfield, J.J., 1982. Neural networks and physical systems with emergent collective computational abilities. Proceedings of the national academy of sciences 79, 2554–2558.
Hoshino, O., Usuba, N., Kashimori, Y., Kambara, T., 1997. Role of itinerancy among attractors as dynamical map in distributed coding scheme. Neural Networks 10, 1375–1390.
Hupkes, D., Dankers, V., Mul, M., Bruni, E., 2020. Compositionality decomposed: How do neural networks generalise? Journal of Artificial Intelligence Research 67, 757–795.
Irving, G., Szegedy, C., Alemi, A.A., Eén, N., Chollet, F., Urban, J., 2016. Deepmath-deep sequence models for premise selection. Advances in Neural Information Processing Systems 29, 2235–2243.
Jaeggi, S.M., Buschkuehl, M., Jonides, J., Perrig, W.J., 2008. Improving fluid intelligence with training on working memory. Proceedings of the National Academy of Sciences 105, 6829–6833.
Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., Gulwani, S., 2018. Neural-guided deductive search for real-time program synthesis from examples. arXiv preprint arXiv:1804.01186 .
Katz, G., Huang, D.W., Hauge, T., Gentili, R., Reggia, J., 2017. A novel parsimonious cause-effect reasoning algorithm for robot imitation and plan recognition. IEEE Transactions on Cognitive and Developmental Systems 10, 177–193.
Katz, G.E., Akshay, ., Davis, G.P., Gentili, R.J., Reggia, J.A., in review. Tunable neural encodings of symbolic robotic manipulation algorithms. Frontiers in neurorobotics .
Katz, G.E., Davis, G.P., Gentili, R.J., Reggia, J.A., 2019. A programmable neural virtual machine based on a fast store-erase learning rule. Neural Networks 119, 10–30.
Katz, G.E., Gupta, K., Reggia, J.A., 2020. Reinforcement-based program induction in a neural virtual machine, in: 2020 International Joint Conference on Neural Networks (IJCNN), IEEE. pp. 1–8.
Kemker, R., McClure, M., Abitino, A., Hayes, T., Kanan, C., 2018. Measuring catastrophic forgetting in neural networks, in: Proceedings of the AAAI Conference on Artificial Intelligence.
Kipf, T., Li, Y., Dai, H., Zambaldi, V., Sanchez-Gonzalez, A., Grefenstette, E., Kohli, P., Battaglia, P., 2019. Compile: Compositional imitation learning and execution, in: International Conference on Machine Learning (ICML), pp. 3418–3428.
Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A.A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al., 2017. Overcoming catastrophic forgetting in neural networks. Proceedings of the national academy of sciences 114, 3521–3526.
Komendantskaya, E., 2011. Unification neural networks: unification by error-correction learning. Logic Journal of the IGPL 19, 821–847.
Koopman, P., 1989. Stack Computers: the new wave. Carnegie Mellon University.
Lake, B., Baroni, M., 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks, in: International Conference on Machine Learning, pp. 2873–2882.
Le, H.M., Jiang, N., Agarwal, A., Dudík, M., Yue, Y., Daumé III, H., 2018. Hierarchical imitation and reinforcement learning. arXiv preprint arXiv:1803.00590 .
Loula, J., Baroni, M., Lake, B., 2018. Rearranging the familiar: Testing compositional generalization in recurrent networks, in: Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, pp. 108–114.
Manohar, S.G., Zokaei, N., Fallon, S.J., Vogels, T., Husain, M., 2019. Neural mechanisms of attending to items in working memory. Neuroscience & Biobehavioral Reviews .
Marcus, G., 2020. The next decade in ai: four steps towards robust artificial intelligence. arXiv preprint arXiv:2002.06177 .
McCarthy, J., Levin, M.I., Abrahams, P.W., Edwards, D.J., Hart, T.P., 1965. LISP 1.5 programmer's manual. MIT press.
Miller, P., 2016. Itinerancy between attractor states in neural systems. Current opinion in neurobiology 40, 14–22.
Mongillo, G., Barak, O., Tsodyks, M., 2008. Synaptic theory of working memory. Science 319, 1543–1546.
Montero-Odasso, M., Verghese, J., Beauchet, O., Hausdorff, J.M., 2012. Gait and cognition: a complementary approach to understanding brain function and the risk of falling. Journal of the American Geriatrics Society 60, 2127–2136.
Neto, J.P., Siegelmann, H.T., Costa, J.F., 2003. Symbolic processing in neural networks. Journal of the Brazilian Computer Society 8, 58–70.
Norvig, P., 1992. Paradigms of artificial intelligence programming: case studies in Common LISP. Morgan Kaufmann.
Oberauer, K., 2009. Design for a working memory. Psychology of learning and motivation 51, 45–100.
Oberauer, K., Süβ, H.M., Wilhelm, O., Wittmann, W.W., 2008. Which working memory functions predict intelligence? Intelligence 36, 641–652.
Oberauer, K., Süß, H., Wilhelm, O., Sander, N., 2007. Individual differences in working memory capacity and reasoning ability, in: Variation in Working Memory. Oxford University Press, pp. 49–75.
Osa, T., Pajarinen, J., Neumann, G., Bagnell, J.A., Abbeel, P., Peters, J., et al., 2018. An algorithmic perspective on imitation learning. Foundations and Trends® in Robotics 7, 1–179.
Parent, A., Hazrati, L.N., 1995. Functional anatomy of the basal ganglia. i. the cortico-basal ganglia-thalamo-cortical loop. Brain research reviews 20, 91–127.
Parisi, G.I., Kemker, R., Part, J.L., Kanan, C., Wermter, S., 2019. Continual lifelong learning with neural networks: A review. Neural Networks 113, 54–71.
Pascanu, R., Jaeger, H., 2011. A neurodynamical model for working memory. Neural networks 24, 199–207.
Persiani, M., Franchi, A.M., Gini, G., 2018. A working memory model improves cognitive control in agents and robots. Cognitive Systems Research 51, 1–13.
Phillips, J.L., Noelle, D.C., 2005. A biologically inspired working memory framework for robots, in: ROMAN 2005. IEEE International Workshop on Robot and Human Interactive Communication, 2005., IEEE. pp. 599–604.
Reed, S., De Freitas, N., 2015. Neural programmer-interpreters. arXiv preprint arXiv:1511.06279 .
Reggia, J.A., Huang, D.W., Katz, G., 2017. Exploring the computational explanatory gap. Philosophies 2, 5.
Reggia, J.A., Katz, G.E., Davis, G.P., 2018. Humanoid cognitive robots that learn by imitating: Implications for consciousness studies. Frontiers in Robotics and AI 5, 1.
Reggia, J.A., Katz, G.E., Davis, G.P., 2019. Modeling working memory to identify computational correlates of consciousness. Open Philosophy

2, 252–269.

Reggia, J.A., Katz, G.E., Davis, G.P., 2020. Artificial conscious intelligence. Journal of Artificial Intelligence and Consciousness 7, 95–107.

Reggia, J.A., Katz, G.E., Davis, G.P., Gentili, R.J., in press. Avoiding catastrophic forgetting with short-term memory during continual learning .

Reggia, J.A., Monner, D., Sylvester, J., 2014. The computational explanatory gap. Journal of Consciousness Studies 21, 153–178.

Rocktäschel, T., Riedel, S., 2017. End-to-end differentiable proving. Advances in Neural Information Processing Systems 30.

Rose, N.S., LaRocque, J.J., Riggall, A.C., Gosseries, O., Starrett, M.J., Meyering, E.E., Postle, B.R., 2016. Reactivation of latent working memories with transcranial magnetic stimulation. Science 354, 1136–1139.

Rule, M.E., O'Leary, T., Harvey, C.D., 2019. Causes and consequences of representational drift. Current opinion in neurobiology 58, 141–147.

Rush, A.M., Chopra, S., Weston, J., 2015. A neural attention model for abstractive sentence summarization, in: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, pp. 379–389.

Russell, S., Norvig, P., 2002. Artificial intelligence: a modern approach.

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., Lillicrap, T., 2016. Meta-learning with memory-augmented neural networks, in: International conference on machine learning, PMLR. pp. 1842–1850.

Seibel, P., 2006. Practical common lisp. Apress.

Shuggi, I.M., Oh, H., Shewokis, P.A., Gentili, R.J., 2017. Mental workload and motor performance dynamics during practice of reaching movements under various levels of task difficulty. Neuroscience 360, 166–179.

Sidarta, A., van Vugt, F.T., Ostry, D.J., 2018. Somatosensory working memory in human reinforcement-based motor learning. Journal of neurophysiology 120, 3275–3286.

Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al., 2016. Mastering the game of go with deep neural networks and tree search. Nature 529, 484–489.

Stanley, K.O., Clune, J., Lehman, J., Miikkulainen, R., 2019. Designing neural networks through neuroevolution. Nature Machine Intelligence 1, 24–35.

Stokes, M.G., 2015. 'activity-silent'working memory in prefrontal cortex: a dynamic coding framework. Trends in Cognitive Sciences 19, 394–405.

Such, F.P., Madhavan, V., Conti, E., Lehman, J., Stanley, K.O., Clune, J., 2017. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. arXiv preprint arXiv:1712.06567 .

Sukhbaatar, S., Weston, J., Fergus, R., et al., 2015. End-to-end memory networks. Advances in neural information processing systems 28, 2440–2448.

Sun, R., Naveh, I., 2004. Simulating organizational decision-making using a cognitively realistic agent model. Journal of Artificial Societies and Social Simulation 7.

Sylvester, J., Reggia, J., 2016. Engineering neural systems for high-level problem solving. Neural Networks 79, 37–52.

Sylvester, J., Reggia, J., Weems, S., Bunting, M., 2013. Controlling working memory with learned instructions. Neural Networks 41, 23–38.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need, in: Advances in neural information processing systems, pp. 5998–6008.

You, Q., Jin, H., Wang, Z., Fang, C., Luo, J., 2016. Image captioning with semantic attention, in: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 4651–4659.

Zanto, T.P., Rubens, M.T., Thangavel, A., Gazzaley, A., 2011. Causal role of the prefrontal cortex in top-down modulation of visual processing and working memory. Nature neuroscience 14, 656–661.

Zaremba, W., Sutskever, I., 2014. Learning to execute. arXiv preprint arXiv:1410.4615 .

Supplementary Material

NeuroLISP: High-level Symbolic Programming with Attractor Neural Networks

Unlettered sections, equations, and figures refer to published journal article, available at:
https://doi.org/10.1016/j.neunet.2021.11.009

Appendix A. NeuroLISP Language Specification

The full list of implemented interpreter functions is listed in Table A.3. Ellipses indicate repetition of arguments. NeuroLISP selectively evaluates sub-expressions according to an expression's operator (first element), and uses the Stack regions to hold intermediate return values for operations with multiple arguments. Note that some functions involve conditional evaluation of argument expressions (e.g., `cond`, `and`, etc). NeuroLISP currently does not support error checking during parsing, user-defined macros, strings, or numerals, but could be programmed to do so by updating the interpreter firmware.

Table A.3: Full specification of the NeuroLISP language operators.

`(cons x y)`	creates a cons cell containing two values
`(car x)`	returns the first value in a cons cell
`(cdr x)`	returns the second value in a cons cell
`(cadr x)`	equivalent to `(car (cdr x))`
`(list x ...)`	creates a list containing the supplied elements (zero or more arguments), represented by a chain of cons cells, terminated by the NIL symbol (empty list)
`(makehash)`	creates a hash map (associative array)
`(checkhash key map)`	checks whether a key is contained in a map (returns `true` or `false`)
`(gethash key map)`	returns the value associated with a key in a map (undefined if key not contained in map)
`(sethash key val map)`	associates a key and value in a map
`(remhash key map)`	removes the key/value pair for a key in a map
`(read)`	reads an input expression and stores it in memory. Parentheses-delineated sequences are recursively parsed as lists, and the quote symbol (') is parsed as an encapsulating quote operation (e.g., `'x` becomes `(quote x)`)
`(print x)`	prints an expression as output. Nested expressions made of cons cells are printed recursively, while functions and hash maps are printed as `#FUNCTION` and `#HASH`
`(defun name (args ...) body)`	defines a function with a name, zero or more arguments, and a body expression
`(lambda (args ...) body)`	creates an anonymous function with zero or more arguments and a body expression
`(label name (args ...) body)`	like lambda, except the function closure contains a binding from the given name to the anonymous function, permitting recursion
`(let ((var val) ...) body)`	binds a series of one or more variable/value pairs, and executes a body expression with the bindings in scope. After completion, the bindings fall out of scope.
`(setq var val ...)`	binds a series of one or more variable/value pairs in the current environment namespace. If bindings for any variable exist in the current scope, they are updated. Otherwise, new bindings are created in the default environment.
`(cond ((clause body) ...)`	conditionally evaluates expressions based on test clauses. Each test clause is evaluated in sequence until one returns `true` or the end of the list is reached. If a clause returns true, its corresponding body expression is evaluated

`(if clause true-body false-body)`	if clause evaluation returns true, the true-body is evaluated. Otherwise, the false-body is evaluated
`(eq x y)`	returns true if `x` and `y` are identical. Does not check for structural equivalence in non-atomic data structures (i.e., different cons cells with the same contents are not considered equal)
`(atom x)`	returns true if `x` is an atomic symbol
`(listp x)`	returns true if `x` is a cons cell
`(not x)`	returns true if `x` is `false` or `NIL`
`(and x y ...)`	returns true if none of the arguments evaluate to `false` or `NIL`. Evaluation is short-circuited if an argument evaluates to `false` or `NIL`
`(or x y ...)`	returns true if all of the arguments evaluate to `false` or `NIL`. Evaluation is short-circuited if any argument evaluates to something other than `false` or `NIL`
`(eval expr)`	evaluates the return value of evaluating an expression
`(quote expr)`	returns an expression without evaluating it
`(progn expr ...)`	evaluates a sequence of expressions and returns the return value of the last expression
`(dolist (var list ret-val) body)`	iterates through a list, binding each element to a variable, and evaluating a body expression with the binding in scope. Upon completion, returns either `NIL` or evaluates an optional `ret-val` expression and returns the result
`(error msg)`	prints an error with an optional message, and halts the interpreter
`(halt)`	halts the interpreter

Appendix B. NeuroLISP Architecture Details

Table B.4 lists the neural regions in NeuroLISP and the region-specific gates utilized by the flashed interpreter firmware. The $g_r^{converge}$ gate (last column) is a special gate that simplifies attractor convergence in the *mem* region. When this gate is active, recurrent dynamics are run repeatedly using the auto-associative matrix until activity converges to a stable attractor, or until a pre-specified number of timesteps has elapsed (10 was used for testing). The meaning of each other gate can be found in Section 2.1 (see Equations 1 - 4).

Table B.4: Neural regions in NeuroLISP (see Figure 2) and their implemented region-specific gates.

Region	g_r^{bias}	g_r^{noise}	g_r^{read}	g_r^{print}	$g_r^{saturate}$	g_r^{ϵ}	$g_r^{converge}$
data stack					✓		
runtime stack					✓		
op					✓	✓	
gate sequence	✓						
gate output							
lex		✓	✓	✓	✓	✓	
mem		✓			✓	✓	✓
mem-ctx		✓			✓	✓	
env		✓			✓	✓	
env-ctx		✓			✓	✓	

Table B.5 lists the connections between and within regions in NeuroLISP and their functional purpose in model execution. Each connection links a source region to a target region, and connections with shared source/target regions are distinguished by unique labels. Some connections (marked in the "Lrn" column) have learning gates that allow them to be updated during model execution ($g^{learn}_{r,q[\ell]}(t)$; see Equation 5). Connections from context regions (marked with a *) are unweighted one-to-one multiplicative connections (see Equation 2). All other connections are weighted all-to-all connections (see Equation 1).

Table B.5: List of connections in NeuroLISP and their functions.

Target	**Source**	**Label**	**Lrn**	**Function**
data stack				
	data stack	fwd		pushing data stack
	data stack	bwd		popping data stack
runtime stack				
	runtime stack	fwd		pushing runtime stack
	runtime stack	bwd		popping runtime stack
op				
	lex			associating op sequences with symbolic labels
	op			advancing through operation sequences
	runtime stack		✓	returning from an operation sub-call
gate sequence				
	op			associating gate sequences with op sequence states
	gh			advancing through gate sequences
	lex		✓	comparisons on lexicon patterns
	mem		✓	comparisons on memory patterns
	env		✓	comparisons on namespace patterns
gate output				
	gate sequence			retrieving gate patterns from gate sequence states
lex				
	lex			checking if a symbol is built-in
	op			associating symbolic arguments with op sequence states
	mem		✓	retrieving the symbol stored in a memory state
mem				
	mem	auto	✓	memory state attractor convergence
	mem	hetero	✓	memory state hetero-associative transitioning
	*mem-ctx			multiplicative contextual gating of mem dynamics
	lex		✓	retrieving the dedicated memory state for a lexicon symbol
	env		✓	binding memory states to variables in namespaces
	runtime stack		✓	temporary storage during operation sequence execution
	data stack		✓	temporary storage during operation sequence execution
mem-ctx				
	mem		✓	cons cell and associative array key-value associations
	lex		✓	associative array map-key associations
env				
	env	auto	✓	binding-specific namespace attractor convergence
	env	hetero	✓	namespace nesting
	*env-ctx			multiplicative contextual gating of env dynamics
	mem		✓	closure binding
	runtime stack		✓	temporary storage during operation sequence execution
env-ctx				
	lex		✓	variable binding associations

Appendix C. Cons Cell Implementation

A cons cell is stored in neural memory as a trajectory from a unique memory state through the elements stored in the cons cell. Because the trajectory runs through the memory states stored in the cell, constructing a cons cell only requires the addition of a single attractor, and a data structure can be stored in more than one cons cell without being copied. For example, if a memory state is stored as the car element of two cons cells, it has two outgoing transitions linking it to the corresponding cdr elements (shown in Figure C.18). Each of these transitions is differentially accessible because it is contextualized by the unique state of the corresponding cons cell.

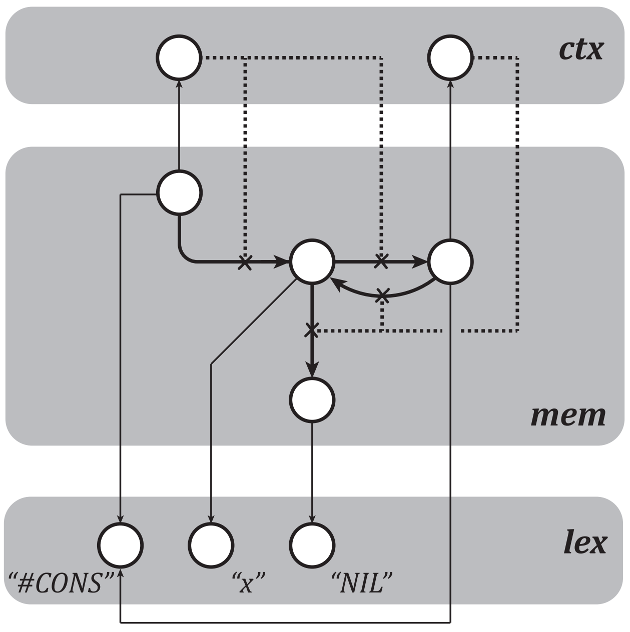

Figure C.18: Graphical depiction of nested cons cells stored in neural memory, illustrating state recycling for lists with repeat elements. Each circle represents a distributed activity pattern, and solid arrows represent learned associations/transitions between these patterns. The represented list contains two copies of the symbol "x", and can be constructed by the expression `(cons 'x (cons 'x NIL))`. Two *mem* states representing cons cells can be identified by their associations with the reserved "#CONS" symbol, represented by a unique activity state in *lex* (bottom left). In addition, each cons cell memory state has an associated context state (top) that contextualizes the transitions linking it with its corresponding car and cdr elements (see Figure 3b). Because both cons cells contain the same car element, the *mem* state associated with the "x" symbol contains two outgoing transitions with unique contexts (center circle in *mem* rectangle). Thus, in the context of the outer cons cell, this state transitions to the inner cons cell, and in the context of the inner cons cell, it transitions to the *mem* state for the null symbol (bottom of *mem* rectangle, associated with "nil" pattern in *lex*).

The car and cdr operations that can be performed on cons cells are implemented as follows. First, the cons cell memory state is activated, and the corresponding context state is retrieved via the pathway from *mem* to *ctx*. This state is then used to contextualize a transition from the cons cell to the car (first) element, which completes the car operation. To complete a cdr operation, an additional transition is then executed from the car element to the cdr (second) element. These operations can be nested: if the car or cdr element is also a cons cell, its context state can be retrieved in order to access one of its elements.

Construction of a cons cell is carried out using several gates, including the noise (Equation 1), eligibility trace (Equation 4), and plasticity gates (Equation 6). The memory states to be linked as car and cdr elements are computed by sub-expressions during program evaluation, and pushed onto the data stack for retrieval during cons cell construction. Once these states are available, the cons cell context state is generated in *ctx*, and the transitions are constructed in reverse order. First, the cdr element is retrieved, masked by the context state (via multiplicative gating; see Equation 2), and stashed in the *mem* eligibility trace ($\epsilon_{mem}(t)$). Then, the car element is retrieved and masked, and the transition from car to cdr element is learned. Next, a new *mem* state is generated to represent the cons cell, and the process is repeated to link the cons cell state to the car state. Finally, the cons cell *mem* state is linked to the generated *ctx* state, making it accessible for subsequent car and cdr operations. A similar process is used for the "list" operation, which is equivalent to nested cons operations (i.e., `(list a b c) == (cons a (cons b (cons c NIL)))`, where `NIL` is a reserved symbol for the null state that serves as a list terminator).

Appendix D. Associative Array Implementation

The organization of associative arrays in memory makes it possible to check whether a key is contained in a map: if the transition from the map yields the key memory state, then the key is contained in the map (Figure D.19a), and the corresponding value can be retrieved via a transition from the key state (Figure D.19b). This can be confirmed with a comparison operation (Section 2.3.2). Adding or updating a key/value pair involves learning the transitions described above (from map to key and key to value), and deleting a key/value pair involves changing the transition from the map state such that it targets a state other than the key state (e.g., the `NIL` state), thereby causing a mismatch that can be detected via comparison. Note that the above operations can be performed with constant-time complexity, as they do not require iteration through key/value pairs in memory. This is possible because of the underlying implementation of maps as attractor graphs with context-dependent transitions, rather than unbranched attractor sequences.

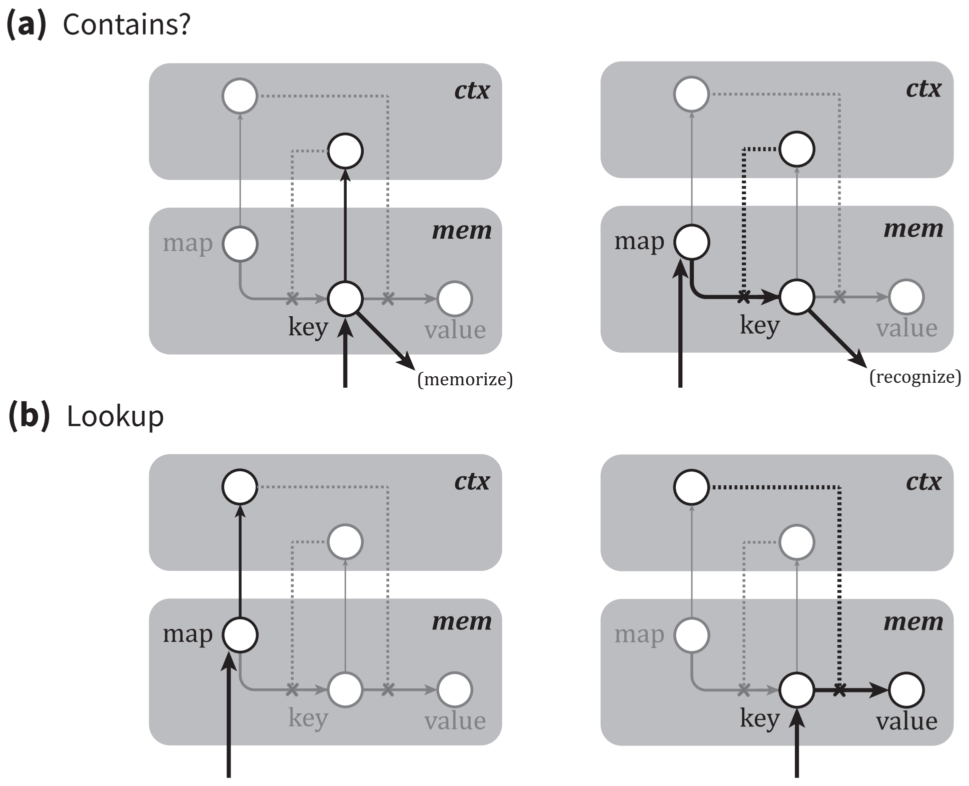

Figure D.19: Graphical depiction of the neural operations for checking for a key and retrieving its value from an associative array / map (see Figure 3c). Circles represent distributed activity patterns. Arrows entering *mem* states from below represent inputs from other regions that are not shown in the image (i.e., *lex* or *data stack*). The depicted operations are agnostic to the source of these inputs. (a) To check if a key is contained in a map, the key memory state (labeled "key") is retrieved and memorized for subsequent comparison (Section 2.3.2), and the corresponding context state in *ctx* is retrieved (bottom right circle in *ctx* rectangle). Next, the map state (labeled "map") is retrieved, and the key context state is used to execute a transition. If the key is contained in the map, this transition will yield the key memory state, which can then be recognized via comparison. Otherwise, a random state will be retrieved, producing a false comparison (i.e., failed recognition). Memorization and recognition occur in the pathway from *mem* to *gate sequence*. (b) Assuming the key is contained in the map, its value can be retrieved as follows. First the map state is retrieved, along with its corresponding context state (top left circle in *ctx* rectangle). Then, the key state is retrieved, and the context state is used to execute a transition to the value state (labeled "value", bottom right of *mem* rectangle).

Appendix E. Organization of Interpreter Memory

The relationship between LISP programs and interpreter operation sequences can be seen in Figure E.20, where a LISP expression is represented by a cons cell in the *mem* region (top), and the low-level assembly op-sequences are represented as sequences of activity in the *op* region (bottom right). Each pattern of activity in *op* represents an assembly instruction that is associated with an opcode and an optional operand via the pathways from *op* to *gate sequence* and *lex*, respectively. The opcode corresponds to a sequence of states in the *gate sequence* region, each of which is associated with a pattern of activity in the *gate output* region that specifies which model components are active or inactive in a given timestep (bottom left of Figure E.20). The operand corresponds to a pattern of activity in the *lex* region that represents a discrete symbol (center right). These symbols serve various functions in the model; for example, they can be printed to the environment as output, used to contextualize environment lookups (Section 2.3.4), or used to retrieve a new *op* sequence during recursive evaluation (i.e., an op-sequence call). Retrieval and usage of the optional operand is directed by the gating sequence specified by the opcode. For example, an instruction that prints a symbol involves opening the pathway from *op* to *lex* to retrieve the instruction operand, and then opening a special *lex* output gate that signals to the environment that an output is ready to be printed (see Section 2.3.3 for further details on input/output operations).

Stack regions are initialized with a bi-directional chain of associated states, each of which serves as a pointer that can be dynamically bound to activity states in other regions (*mem*, *env*, or *op*). Pushing onto a stack involves advancing its activity to the next pointer in the chain, and learning an association between the stack pointer state and an activity state in a target region. The associated activity state can be retrieved as needed during program execution by opening the activity gate from the stack region to the target region, and popped off the stack by advancing the stack region to the previous pointer state in the chain.

During recursive program evaluation, the Controller stores the current memory state and *op* state on the runtime stack, advances to the sub-expression memory state, and jumps to the beginning of the *eval* op-sequence. Upon completion of sub-expression evaluation, the Controller retrieves the *op* state from the stack, returning to the operation that initiated recursive evaluation. At this point, the current memory state represents the return value from evaluating the sub-expression, and the parent expression's memory state is available for retrieval from the runtime stack. If the parent expression contains multiple sub-expressions, the calling op-sequence can then stash the return value on the data stack, retrieve the parent expression, and advance to the next sub-expression for another round of recursive evaluation. Once all of the return values of sub-expressions are stored on the data stack, the parent expression's operation can then be performed. For example, the cons operation described in Section 2.2.1 will retrieve the memory states representing elements of the new cons cell, link them together in memory, and return the generated cons cell state upon completion. The returned cons cell is then passed up for use in the calling expression (e.g., another cons operation may use it as an element in another cons cell).

To evaluate an expression with a built-in operator, the *lex* pattern associated with the car element of the expression is retrieved, the head of the cons cell in *mem* is recovered, and the *lex* pattern is used to retrieve the operator's *op* sequence. Retrieval of the cons cell memory state makes the remainder of the expression available for access during execution of the operation.

Appendix F. Comparison Operations

Formally, comparisons are accomplished in the following manner. Unlike with other weight matrices, learning that occurs on the comparison pathways completely overrides the previous weight matrix, leaving only the most recently learned association for recognizing the memorized pattern. The eligibility trace of the *gate sequence* region is initialized to an activity state $\mathbf{v}_{gs}[true]$, which is the start of the jumping gate sequence that occurs when a comparison is successful. Thus, memorizing an activity pattern in a source region (*src*) at time t for later comparison can be expressed as:

$$\Delta W_{gs,src}(t+1) = \underbrace{\frac{1}{\|\mathbf{v}_{src}(t)\|}}_{\text{norm}} \underbrace{\mathbf{v}_{gs}[true]}_{\text{jump state}} \underbrace{\mathbf{v}_{src}(t)^{\top}}_{\text{source}} - W_{gs,src}(t) \tag{F.1}$$

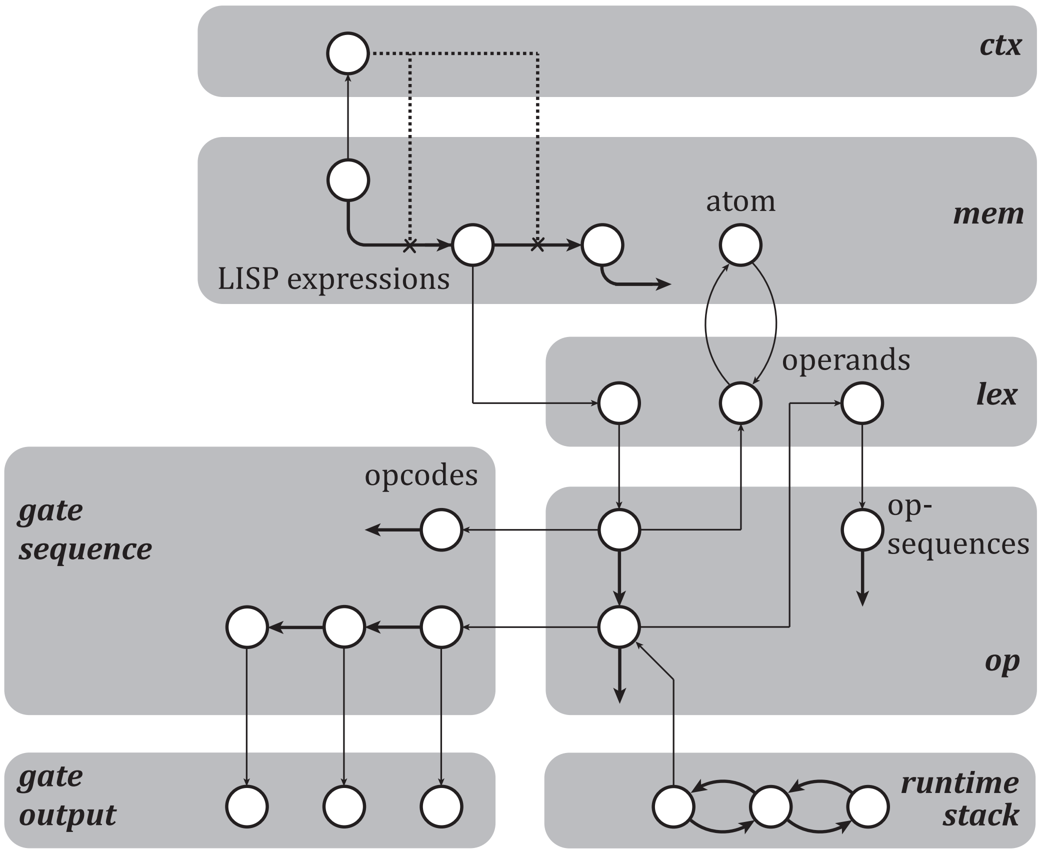

Figure E.20: Relations between states in various regions of the NeuroLISP architecture that implement interpreter functions. Circles represent distributed activity patterns, and arrows represent learned transitions between activity patterns within or between regions. A LISP expression is represented as a cons cell in the *mem* region (see Figure 3b), and its first element represents a LISP operator. This operator can be used to retrieve the corresponding sequence in the *op* region that implements the operation (bottom right *op* rectangle). Each *op* state has an opcode that corresponds to a unique sequence of activity in the *gate sequence* region, which specifies a temporal sequence of gating values via associations with *gate output* states (bottom left). These sequences control the behavior of the relevant model components, including the recurrent dynamics of the *gate sequence* and *op* regions themselves. Some *op* states are associated with an optional operand via the pathway from *op* to *lex* (center right). The corresponding opcode sequence determines what is to be done with this operand; for example, it may be used to retrieve an atomic memory state (top center state labeled "atom" in the *mem* rectangle), or to retrieve a new *op* sequence during recursive evaluation (right side arrow from *lex* state to *op* state). During recursive op-sequence evaluation, states in the *runtime stack* region are temporarily associated with the calling *op* state, which allows the model to return to the calling op-sequence instruction upon completion. For clarity, some associations are omitted from the diagram, including associations between *runtime stack* and *mem* states, as well as associations between some *op*, *gate sequence*, and *gate output* states.

where $W_{gs,src}(t)$ is the weight matrix for the pathway from a source region (*src*) to the gate sequence region (*gs*) at time t, $\mathbf{v}_{src}(t)$ is the *src* activity pattern to memorize, and $\mathbf{v}_{gs}[true]$ is the gate sequence activity state for op-sequence jumping. The updated weight matrix can then be used to determine if another activity pattern in *src* is similar enough to the memorized pattern to yield a successful comparison. This is done by presenting a new input pattern to *gate sequence* while attempting to drive *gate sequence* activity away from the "true" state toward an alternative "false" state that corresponds to a false comparison:

$$\mathbf{v}_{gs}(T+1) = \sigma_{gs}\Big(\underbrace{W_{gs,src}(T)\,\mathbf{v}_{src}(T)}_{\text{recognition}} - \underbrace{\theta\,\mathbf{v}_{gs}[false]}_{\text{bias}}\Big) \tag{F.2}$$

where $\mathbf{v}_{gs}(T+1)$ is the post-comparison activity state in the gate sequence region (*gs*) at time $T+1$, σ_{gs} is the activation function in *gs*, $W_{gs,src}(T)$ is the weight matrix described above, $\mathbf{v}_{src}(T)$ is the activity pattern in *src* that is compared to the memorized pattern, θ is the similarity threshold, and $\mathbf{v}_{gs}[false]$ is the gate sequence activity state for non-jumping operation advancement that occurs when a comparison fails. Equation F.2 is a special form of Equations 1 - 3, where the bias term $\mathbf{b}_{gs} = \theta \, \mathbf{v}_{gs}[false]$. The comparison threshold θ determines how similar the two *src* activation states must be to yield a successful comparison (typically $\theta = 0.95$). Note that when a non-threshold activation function such as the hyperbolic tangent is used, the activity pattern resulting from Equation F.2 will require saturation to match the magnitude of $\mathbf{v}_{gs}[true]$ or $\mathbf{v}_{gs}[false]$.

Appendix G. Environments

The organization of environments in neural memory is graphically depicted in Figure G.21 and described in detail below. Environment namespaces are represented as activity states in the *env* region of the model (top of Figure 2).

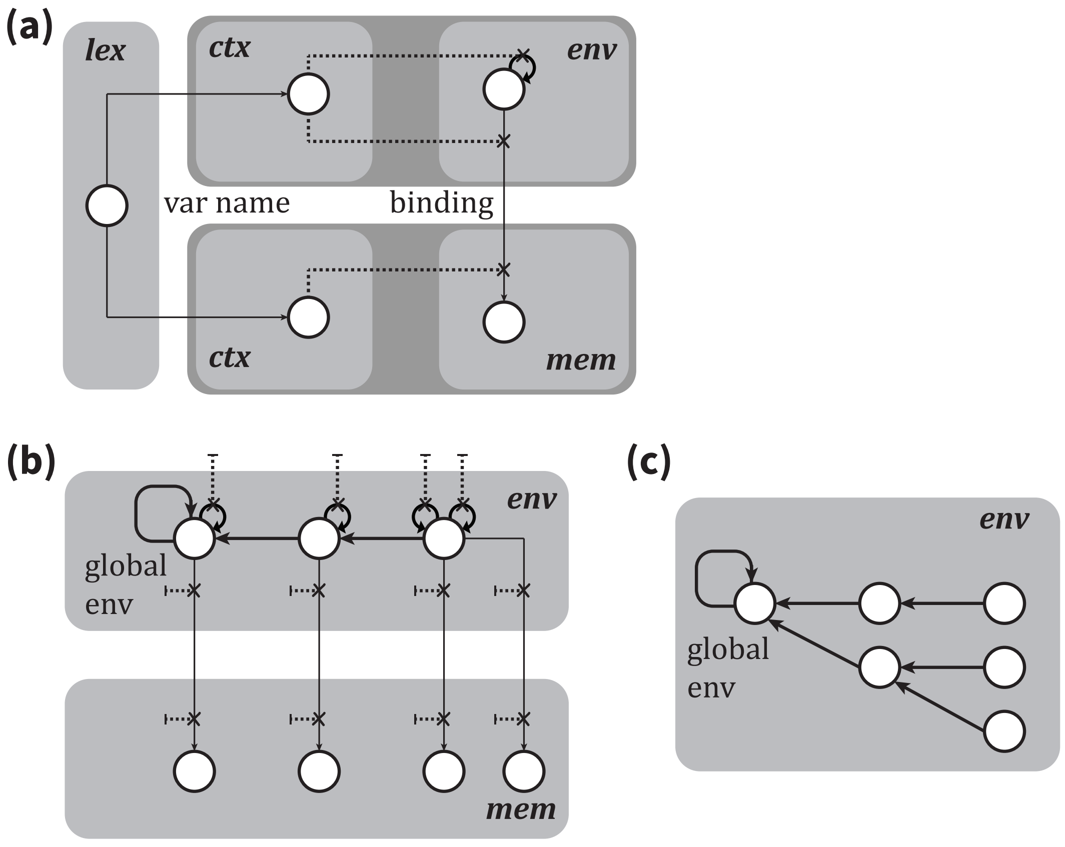

Figure G.21: Graphical depiction of various components of environment management in NeuroLISP. Circles represent distributed activity patterns, and arrows represent learned associations/transitions between activity patterns. (a) Variable bindings are stored as contextualized associations between activity states in the *env* and *mem* regions (arrow labeled "binding", center). The *env* state represents an environmental namespace that may store bindings for several variables. A *lex* activity pattern representing the variable name ("var name", left) is associated with unique patterns in the context regions for *env* and *mem* that are each used to contextualize the corresponding region during variable retrieval (dashed lines connected to "binding" line). In addition, the context state for the *env* region also contextualizes an auto-association of the *env* state with itself (looped arrow in *env* circle, top). This makes it possible to determine if there is a binding for a particular variable in a particular namespace: if the namespace's *env* state is stable under contextualized auto-associative dynamics, a binding exists. This can be determined via a comparison operation (Section 2.3.2). (b) For dynamically scoped variable binding, environment namespaces are chained together into a sequence that terminates with the global environment ("global env", right side). When looking up a variable, the namespaces are inspected in order until a binding is found, or until a transition is executed from the global environment back to itself (i.e., no environments contain a binding for the variable). (c) For lexically scoped variable binding, namespaces are organized into an inverted tree, where different branches are created and maintained for function closures. When a function is defined, a namespace is created and bound to the memory state representing its closure (Section 2.3.5, Figure H.22). When the function is called, this namespace is retrieved and a new namespace is created to store argument bindings (Section 2.3.5).

Each variable binding is maintained as a context-dependent association between a namespace and an activity state in the *mem* region that represents a variable's value (arrow labeled "binding" in Figure G.21a). This association is contextualized by activity states in the *ctx* regions (one for *mem* and one for *env*) that are derived from a *lex* pattern that represents the variable name ("var name" on left side of Figure G.21a). Thus, to retrieve a variable, an inter-regional transition is performed from *env* to *mem* in the context of the variable name. As with associative arrays / maps (Section 2.2.2), it is necessary to validate that a namespace contains a binding for a particular variable name before retrieving it. This is done using context-dependent auto-associative learning: namespaces that contain a binding for a variable are stable attractors under recurrent dynamics when the variable name's context pattern is present. Formally, this is expressed as:

$$\mathbf{v}_{env}[binding] = \mathbf{v}_{ctx-env}[var] \odot \mathbf{v}_{env}[namespace]$$
$$\mathbf{v}_{mem}[binding] = \mathbf{v}_{ctx-mem}[var] \odot \mathbf{v}_{mem}[value]$$

$$\sigma_{env}\Big(\mathbf{v}_{ctx-env}[var] \odot (W_{env,env[auto]}\ \mathbf{v}_{env}[binding])\Big) = \mathbf{v}_{env}[binding] \tag{G.1}$$
$$\sigma_{mem}\Big(\mathbf{v}_{ctx-mem}[var] \odot (W_{mem,env}\ \mathbf{v}_{env}[binding])\Big) = \mathbf{v}_{mem}[binding] \tag{G.2}$$

where:

- $\mathbf{v}_{env}[namespace]$ is the activity state in the *env* region representing the namespace.
- $\mathbf{v}_{ctx-env}[var]$ and $\mathbf{v}_{ctx-mem}[var]$ are the variable-specific activity patterns in the *ctx* regions for *env* and *mem*, respectively.
- $\mathbf{v}_{mem}[value]$ is the activity state in the *mem* region representing the value of the binding.
- $\mathbf{v}_{env}[binding]$ and $\mathbf{v}_{mem}[binding]$ are the contextually-masked activity patterns in the *env* and *mem* regions. The binding is stored as an association between these states (Equation G.2).
- σ_{env} and σ_{mem} are the activation functions for neurons in the *env* and *mem* regions, respectively.
- $W_{env,env[auto]}$ is an auto-associative recurrent weight matrix in the *env* region that is updated when new bindings are created.
- $W_{mem,env}$ is an inter-regional weight matrix from the *env* region to the *mem* region that is updated when new bindings are created.

The relation expressed in Equation G.1 makes it possible to determine whether a binding exists for a given variable in a given namespace: the activity states before and after auto-associative dynamics can be compared (Section 2.3.2). If they match, the binding exists, and the relation expressed in Equation G.2 can be used to retrieve the corresponding value in *mem*. Because bindings are stored as context-dependent associations, a namespace may contain bindings for several different variables, much like an associative array can contain multiple key/value pairs.

Namespaces are organized in a nested fashion: when a variable is looked up, and no binding exists for the innermost namespace, the lookup proceeds to the encapsulating namespace. Once the outermost namespace is reached (referred to as the "global environment") and no binding is found, the lookup fails and the program returns an error. For dynamic scoping, namespaces can be maintained in a sequential chain (Figure G.21b), similar to those of the stack regions. Lexical scoping requires a branched organization (Figure G.21c) because function closures maintain the bindings that existed during function definition. Thus, lexical scoping allows access to bindings that would otherwise fall out of a dynamic scope and become inaccessible.

Operations that create new variable bindings evaluate sub-expressions and store the resulting values (*mem* states) on the data stack. Once all the values are computed, a new environment namespace (*env* state) is created to store the new bindings, and is associated with the prior namespace (arrows between *env* states in Figures G.21b and G.21c). Each binding is created by retrieving the variable/argument name (*lex* state), retrieving the corresponding value from the data stack, and learning the associations outlined above to create the binding (Equations G.1 and G.2). These bindings are then available during subsequent evaluation (i.e., the sub-expression of a `let` operation, or the body of a function). Upon completion, the newly created namespace is abandoned by advancing *env* to the prior namespace. However, if a function was defined before completion (i.e., a `defun` within a `let` expression), the abandoned namespace may be retrievable via a closure, as described in the following section.

Appendix H. Closures

Figure H.22 shows the associations making up a closure stored in memory, as well as the environmental bindings created during a function call. Closures for anonymous functions (lambdas) are stored similarly, but do not include a binding for a function name (arrow from "defun env" to "closure" in Figure H.22, top left). When a function is called, a new namespace is created to store argument bindings ("call env"), and is associated with the namespace that was active when the function was defined ("defun environment"). The caller's argument expressions are evaluated and bound to the variables listed in the function definition, which are retrieved from the closure memory structure ("args list"). The body expression is then retrieved and evaluated with the bindings in scope. Upon completion, the resulting *mem* state is returned to the caller, and the caller's namespace is retrieved from the stack (not shown).

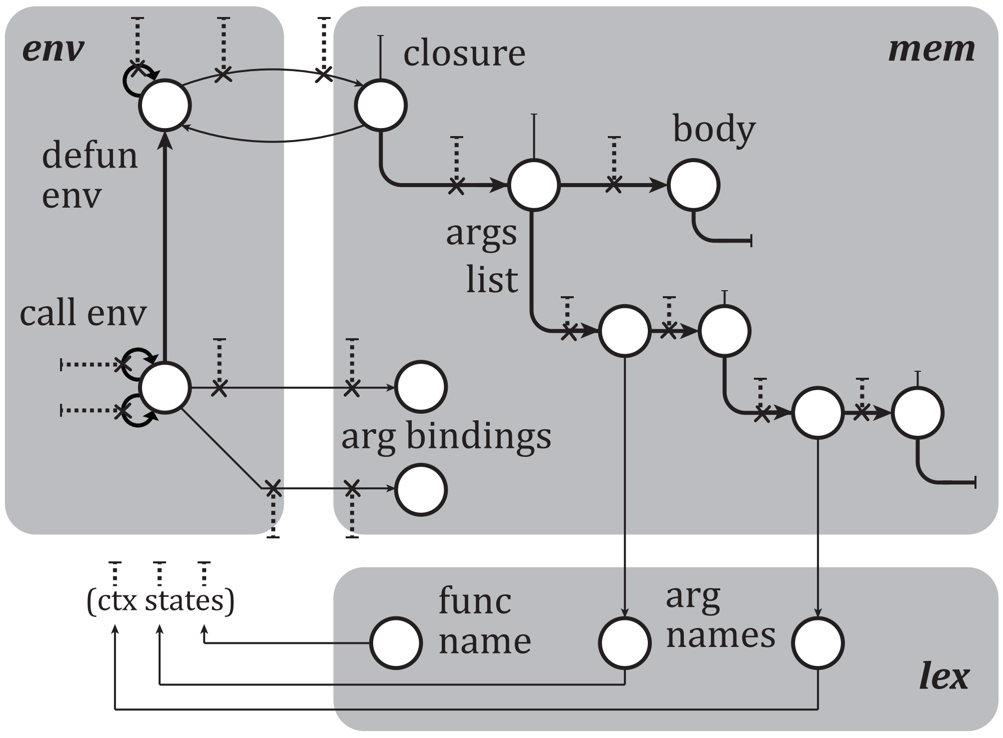

Figure H.22: Graphical depiction of the learned associations that make up closures and argument bindings for function calls. A closure is a cons cell (center top state labeled "closure") containing a list of argument names ("args list", center), and an expression for the body of the function ("body", top right). The closure state is also associated with the environmental namespace that was active when the function was defined ("defun env", top left). In return, the closure is bound to the function name within this environment; a *lex* state representing the function name contextualizes this binding ("func name", bottom). For simplicity, we omit the context states from the diagram, and abbreviate them in the bottom left ("ctx states", see Figure G.21 for details). When the function is called, the interpreter creates a new environment to store argument bindings ("call env", left), evaluates sub-expressions for argument values, and binds them with their associated argument name ("arg bindings", center). Each binding is contextualized by the corresponding *lex* pattern representing the argument name ("arg" names, bottom right), which can be retrieved from the argument list contained in the closure.

Appendix I. Model Parameters for Testing

The model parameters used for the tests outlined in Section 3 are listed in Table I.6. Cells labeled "variable" were experimentally varied (see Sections 2.4 and 3). The sizing of the stack regions was set according to the demands of the test, as determined by the NeuroLISP emulator. Either 256 or 1024 neurons was used (four times the number of orthogonal patterns representing stack frames, for stability purposes). The *op* and *gate sequence* regions were sized according to the number of states necessary for the interpreter firmware sequences (four neurons per orthogonal state pattern). These regions use orthogonal patterns because orthogonality reduces the required region sizes ((Katz et al., 2019)), and because the learned states are only established during one-time initialization. Cells in Table I.6 labeled "flashed" indicate regions that learn solely during the one-time initialization process, which flashes the interpreter firmware. Cells labeled "combo" indicate regions that learn associations in an online fashion during model execution in addition to a small set of flashed associations (e.g., patterns for `NIL`, `true`, and `false` are flashed in the *mem* region, and a pattern is created for the default environment in the *env* region).

Table I.6: Model parameters used for testing. "Size" refers to the number of neurons contained in a region. "Learning Type" refers to the source of learned patterns and associations ("flashed" memories are established during the one-time initialization process, and "combo" refers to a combination of flashed memories and memories learned online during model execution). "Pattern Type" refers to the organization of learned representations: "ortho" patterns are orthogonal vectors established during one-time initialization for high efficiency, "local" patterns are one-hot vectors (used in the gate output region to refer to individual model gates), and "random" patterns are generated with a Bernoulli process. "Activ Func" refers to the activation function used for neurons in the region. "Lambda" refers to the density parameter used for random pattern generation, and determines the probability of generating an individual neural activation value of 1.

Region	Size	Learning Type	Pattern Type	Activ Func	Lambda
runtime stack	256 / 1024	flashed	ortho	sign	N/A
data stack	256 / 1024	flashed	ortho	sign	N/A
op	1536	flashed	ortho	sign	N/A
gate sequence	496	flashed	ortho	sign	N/A
gate output	70	flashed	local	heaviside	N/A
lex	variable	combo	random	sign	0.5
mem	variable	combo	random	sign	0.5
mem-ctx	=size(mem)	combo	random	heaviside	0.25
env	variable	combo	random	sign	0.5
env-ctx	=size(env)	combo	random	heaviside	variable

Appendix J. Interpreter Test Suite

The handwritten test suite described in Section 3.1 is shown in Table J.7.

Table J.7: Basic test suite for the NeuroLISP interpreter. The left column contains test programs, and the right column contains the corresponding expected outputs. Note that because NeuroLISP executes a read-eval-print-loop, the return value of each expression is printed alongside any outputs provided by explicit print commands. Lists (cons cells) are printed as parenthesis-enclosed expressions, while function closures and hash maps (associative arrays) are printed as #FUNCTION and #HASH, respectively.

`(cons (quote A) (cons (quote B) NIL))`	`(A B)`
`(list (quote A) (quote B))`	`(A B)`
`(quote (A B))`	`(A B)`
`(car (cons (quote A) NIL))`	`A`

`(car (cdr (cdr (list`
`  (quote A) (quote B) (quote C)))))`
	`C`

`(car (cdr (car (cdr (quote (A (B C) D))))))`	`C`
`(cadr (quote (A (B C) D)))`	`(B C)`

`(eq ’x ’x)`	`true`
`(eq ’x ’y)`	`false`
`(eq ’x (list ’x))`	`false`
`(atom ’x)`	`true`
`(atom (list ’x))`	`false`
`(listp ’x)`	`false`
`(listp (list ’x))`	`true`
`(print (read)) A`	`A A`
`(print (list (read) (read) (read))) A B C`	`(A B C) (A B C)`
`(progn (print ’foo) (print ’bar) ’baz)`	`foo bar baz`
`(dolist (x ’(A B C) x) (print x))`	`A B C C`
`(eval (quote (print (quote x))))`	`x x`
`(eval (cons ’print (cdr (list ’foo ’(quote x)))))`	`x x`
`(if true ’foo ’bar) (if false ’foo ’bar)`	`foo bar`
`(if (or false (and true true)) ’foo ’bar)`	`foo`
`(cond (false ’a) ((or false false) ’b) ((and true false) ’c) ((not true) ’d) ((eq ’x ’y) ’e) (true ’f))`	`f`
`((lambda (x y) (list x y)) ’foo ’bar)`	`(foo bar)`
`((lambda (f x y) (f x y)) (lambda (x y) (list x y)) ’foo ’bar)`	`(foo bar)`
`((label f (lambda (x) (if x (progn (print (car x)) (f (cdr x)))))) (list ’foo ’bar))`	`foo bar NIL`
`(defun f (x y) (list x y))` `(defun g (x y) (f x y))` `(g ’foo ’bar)`	`#FUNCTION #FUNCTION (foo bar)`
`(let ((x ’foo)) (progn (print x) (let ((x ’bar)) (print x)) x))`	`foo bar foo`
`(let ((x ’foo) (y ’bar)) (list x y))`	`(foo bar)`
`(let ((x ’foo) (y ’bar)) (progn (defun f (x) (print (list x y))) (f ’baz) x))`	`(baz bar) foo`
`(progn (setq x ’foo) x)`	`foo`
`(let ((x ’foo)) (progn (setq x ’bar) x))`	`bar`
`(let ((x ’foo)) (progn (let ((x ’bar)) (setq x ’baz)) x))`	`foo`
`(let ((x ’foo)) (progn (defun f (x) (setq x ’bar)) (f ’baz) x))`	`foo`
`(defun f () (setq x ’foo)) (f) x`	`#FUNCTION foo foo`

```
(let ((hash (makehash))) (progn
  (sethash 'key1 'val1 hash)
  (sethash 'key2 'val2 hash)
  (print (and
    (checkhash 'key1 hash)
    (checkhash 'key2 hash)
    (not (checkhash 'key3 hash))))
  (print (list
    (gethash 'key1 hash)
    (gethash 'key2 hash)))
  (remhash 'key1 hash)
  (print (checkhash 'key1 hash))
  (sethash 'key1 'foo hash)
  (print (checkhash 'key1 hash))
  (print (gethash 'key1 hash))
  hash))
```

```
true (val1 val2) false true foo #HASH
```

```
(progn
  (((lambda (le)
    ((lambda (g) (g g))
      (lambda (h)
        (le (lambda (x) ((h h) x))))))
    (lambda (f)
      (lambda (x) (cond
        (x (progn
          (print x)
          (f (cdr x))
          (print (car x)))
        (true x))))))
    '(a b c))
  'complete)
```

```
(a b c) (b c) (c) c b a complete
```

Appendix K. Multiway Tree Library

The code implementing the multiway tree functions (including helper functions) is listed in Figure K.23. The corresponding test cases are listed in Figure K.24.

```
(defun expr-equal? (x y)
  (cond ((or (atom x) (atom y)) (eq x y))
    ((and (listp x) (listp y))
      (and (expr-equal? (car x) (car y))
           (expr-equal? (cdr x) (cdr y))))
    (true false)))

(defun tree? (expr)
  (or (atom expr)
    (and (listp expr) (atom (car expr))
      (cdr expr) (forest? (cdr expr)))))
(defun forest? (expr)
  (or (not expr)
    (and (tree? (car expr))
      (forest? (cdr expr)))))

(defun copy-tree (tree)
  (if (atom tree) tree
    (cons (car tree)
      (copy-forest (cdr tree)))))
(defun copy-forest (subtrees)
  (if (not subtrees) NIL
    (cons (copy-tree (car subtrees))
      (copy-forest (cdr subtrees)))))

(defun tree-member (elm tree)
  (cond ((atom tree) (eq elm tree))
    (true (or (eq (car tree) elm)
      (forest-member elm (cdr tree))))))
(defun forest-member (elm forest)
  (and forest
    (or (tree-member elm (car forest))
      (forest-member elm (cdr forest)))))

(defun tree-prefix (tree)
  (tree-prefix-helper tree NIL))
(defun tree-prefix-helper (tree seq)
  (if (atom tree)
    (cons tree seq)
    (cons (car tree)
      (forest-prefix-helper
        (cdr tree) seq))))
(defun forest-prefix-helper (subtrees seq)
  (if subtrees
    (tree-prefix-helper
      (car subtrees)
      (forest-prefix-helper
        (cdr subtrees) seq))
      seq))
```

```
(defun tree-subst (new old tree)
  (let
    ((ret (tree-subst-helper new old tree)))
    (if ret ret tree)))
(defun tree-subst-helper (new old tree)
  (cond
    ((expr-equal? tree old) new)
    ((atom tree) NIL)
    (true
      (let ((subtrees (forest-subst-helper
          new old (cdr tree))))
        (if subtrees
          (cons (car tree) subtrees)
          NIL)))))
(defun forest-subst-helper (new old subtrees)
  (if (not subtrees) NIL
    (let ((curr (tree-subst-helper
                  new old (car subtrees)))
          (rest (forest-subst-helper
                  new old (cdr subtrees))))
      (if (or curr rest)
        (cons (if curr curr (car subtrees))
          (if rest rest (cdr subtrees)))
        NIL))))

(defun tree-sublis (subs tree)
  (let ((ret (tree-sublis-helper subs tree)))
    (if ret ret tree)))
(defun tree-sublis-replace (subs tree)
  (if subs
    (if (expr-equal? (car (car subs)) tree)
      (cadr (car subs))
      (tree-sublis-replace (cdr subs) tree))
    NIL))
(defun tree-sublis-helper (subs tree)
  (let ((replacement
          (tree-sublis-replace subs tree)))
    (cond
      (replacement replacement)
      ((atom tree) NIL)
      (true
        (let ((subtrees (forest-sublis-helper
                          subs (cdr tree))))
          (if subtrees
            (cons (car tree) subtrees)
            NIL))))))
(defun forest-sublis-helper (subs subtrees)
  (if (not subtrees) NIL
    (let ((curr (tree-sublis-helper
                  subs (car subtrees)))
          (rest (forest-sublis-helper
                  subs (cdr subtrees))))
      (if (or curr rest)
        (cons
          (if curr curr (car subtrees))
          (if rest rest (cdr subtrees)))
        NIL))))
```

Figure K.23: Library of multiway tree processing functions.

```
(defun test (expr target)
  (if (not (eq (eval expr) target))
    (error (list target 'NOT_EQUAL expr))))

(test '(expr-equal? 'a 'b) false)
(test '(expr-equal? 'a 'a) true)
(test '(expr-equal? '(a (b c)) '(a (b c))) true)
(test '(expr-equal? '(a (b (c))) '(a (b c))) false)

(setq tree1 'a)
(setq tree2 '(a b))
(setq tree3 '(a (b c)))
(setq tree4 '(b d e))
(setq tree5 '(a (f g) c (b d e)))
(setq tree6 '(x y z))
(setq nottree1 '(a))
(setq nottree2 '(a (b c) (d) e))
(setq nottree3 '((a b c) (d e)))

(test '(is-tree? tree1) true)
(test '(is-tree? tree2) true)
(test '(is-tree? tree3) true)
(test '(is-tree? tree4) true)
(test '(is-tree? tree5) true)
(test '(is-tree? nottree1) false)
(test '(is-tree? nottree2) false)
(test '(is-tree? nottree3) false)

(test '(tree-contains? 'a tree1) true)
(test '(tree-contains? 'd tree5) true)
(test '(tree-contains? 'h tree5) false)

(test '(expr-equal? (tree-prefix tree1) '(a)) true)
(test '(expr-equal? (tree-prefix tree2) '(a b)) true)
(test '(expr-equal? (tree-prefix tree3) '(a b c)) true)
(test '(expr-equal? (tree-prefix tree4) '(b d e)) true)
(test '(expr-equal? (tree-prefix tree5) '(a f g c b d e)) true)

(test '(expr-equal? (tree-subst 'z 'a tree1) 'z) true)
(test '(expr-equal? (tree-subst '(z a b) 'a tree1) '(z a b)) true)
(test '(expr-equal? (tree-subst 'z '(b c) tree3) '(a z)) true)
(test '(expr-equal? (tree-subst 'z 'g tree5) '(a (f z) c (b d e))) true)

(setq subs '((a (x y z)) ((b d e) y) (c z)))
(test '(expr-equal? (tree-sublis subs tree1) '(x y z)) true)
(test '(expr-equal? (tree-sublis subs tree5) '(a (f g) z y)) true)
(test '(expr-equal? (tree-sublis subs tree6) tree6) true)

(test '(expr-equal? (copy-tree tree1) tree1) true)
(test '(expr-equal? (copy-tree tree5) tree5) true)
```

Figure K.24: Test cases for multiway tree processing functions.

Appendix L. Unification Test Case Generation

Unification test cases were produced by randomly generating trees and converting them to s-expressions as follows. First, a random tree is generated using the `random_tree` function in the Python `networkx` package[3]. The size of the initial tree is the expression complexity parameter described in Section 3.5 (x-axis of Figures 12 and 13). The tree is rooted by selecting the node with the greatest number of edges, the leaf nodes are labeled with randomly generated symbols, and the tree is copied to produce an identical pair. Then, a set of variable substitutions is generated, each mapping a variable name to a small randomly generated subtree representing the value of that variable. These substitutions are introduced to the pair of trees by randomly selecting a leaf node, and replacing it with a variable name in one tree, and the corresponding value subtree. In 20% of the test cases, a mismatch was introduced to the pair by randomly mutating a node in one tree such that they can no longer be unified. Below is an example of a randomly generated test case with an initial tree size of 10 nodes, leaf symbols drawn from a-j, variable names drawn from V-Z, up to 3 variable substitutions, and a maximum variable value size of up to 5 nodes.

Initial expression:
```
    (((c) (f h)) (b) i)
```
Substitutions:
```
    V ⟶ ((g) j i)
    W ⟶ i
    Y ⟶ (g a)
```
Expression pair with substitutions:
```
    ((((var W)) (f ((g) j i))) (b) (var Y))
    (((i) (f (var V))) (b) (g a))
```
Mismatch mutation (optional):
```
    ((((var W)) (f ((g) j i))) (b) (var Y))
    (((i) (f (var V))) (d) (g a))
```

[3]https://networkx.org/documentation/stable/reference/generated/networkx.generators.trees.random_tree.html