Programming with a Differentiable Forth Interpreter

Matko Bošnjak [1] **Tim Rocktäschel** [2] **Jason Naradowsky** [3] **Sebastian Riedel** [1]

Abstract

Given that in practice training data is scarce for all but a small set of problems, a core question is how to incorporate prior knowledge into a model. In this paper, we consider the case of prior *procedural* knowledge for neural networks, such as knowing how a program should traverse a sequence, but not what local actions should be performed at each step. To this end, we present an end-to-end differentiable interpreter for the programming language Forth which enables programmers to write program sketches with slots that can be filled with behaviour trained from program input-output data. We can optimise this behaviour directly through gradient descent techniques on user-specified objectives, and also integrate the program into any larger neural computation graph. We show empirically that our interpreter is able to effectively leverage different levels of prior program structure and learn complex behaviours such as sequence sorting and addition. When connected to outputs of an LSTM and trained jointly, our interpreter achieves state-of-the-art accuracy for end-to-end reasoning about quantities expressed in natural language stories.

1. Introduction

A central goal of Artificial Intelligence is the creation of machines that learn as effectively from human instruction as they do from data. A recent and important step towards this goal is the invention of neural architectures that learn to perform algorithms akin to traditional computers, using primitives such as memory access and stack manipulation (Graves et al., 2014; Joulin & Mikolov, 2015; Grefenstette et al., 2015; Kaiser & Sutskever, 2015; Kurach et al., 2016; Graves et al., 2016). These architectures can be trained through standard gradient descent methods, and enable machines to learn complex behaviour from input-output pairs or program traces. In this context, the role of the human programmer is often limited to providing training data. However, training data is a scarce resource for many tasks. In these cases, the programmer may have

partial procedural background knowledge: one may know the rough structure of the program, or how to implement several subroutines that are likely necessary to solve the task. For example, in programming by demonstration (Lau et al., 2001) or query language programming (Neelakantan et al., 2015a) a user establishes a larger set of conditions on the data, and the model needs to set out the details. In all these scenarios, the question then becomes how to exploit various types of prior knowledge when learning algorithms.

To address the above question we present an approach that enables programmers to inject their procedural background knowledge into a neural network. In this approach, the programmer specifies a program *sketch* (Solar-Lezama et al., 2005) in a traditional programming language. This sketch defines one part of the neural network behaviour. The other part is learned using training data. The core insight that enables this approach is the fact that most programming languages can be formulated in terms of an abstract machine that executes the commands of the language. We implement these machines as neural networks, constraining parts of the networks to follow the sketched behaviour. The resulting neural programs are consistent with our prior knowledge and optimised with respect to the training data.

In this paper, we focus on the programming language Forth (Brodie, 1980), a simple yet powerful stack-based language that facilitates factoring and abstraction. Underlying Forth's semantics is a simple abstract machine. We introduce ∂4, an implementation of this machine that is differentiable with respect to the transition it executes at each time step, as well as distributed input representations. Sketches that users write define underspecified behaviour which can then be trained with backpropagation.

For two neural programming tasks introduced in previous work (Reed & de Freitas, 2015) we present Forth sketches that capture different degrees of prior knowledge. For example, we define only the general recursive structure of a sorting problem. We show that given only input-output pairs, ∂4 can learn to fill the sketch and generalise well to problems of unseen size. In addition, we apply ∂4 to the task of solving word algebra problems. We show that when provided with basic algorithmic scaffolding and trained jointly with an upstream LSTM (Hochreiter & Schmidhuber, 1997), ∂4 is able to learn to read natural

---

[1]Department of Computer Science, University College London, London, UK [2]Department of Computer Science, University of Oxford, Oxford, UK [3]Department of Theoretical and Applied Linguistics, University of Cambridge, Cambridge, UK. Correspondence to: Matko Bošnjak <m.bosnjak@cs.ucl.ac.uk>.

language narratives, extract important numerical quantities, and reason with these, ultimately answering corresponding mathematical questions without the need for explicit intermediate representations used in previous work.

The contributions of our work are as follows: i) We present a neural implementation of a dual stack machine underlying Forth, ii) we introduce Forth sketches for programming with partial procedural background knowledge, iii) we apply Forth sketches as a procedural prior on learning algorithms from data, iv) we introduce program code optimisations based on symbolic execution that can speed up neural execution, and v) using Forth sketches we obtain state-of-the-art for end-to-end reasoning about quantities expressed in natural language narratives.

2. The Forth Abstract Machine

Forth is a simple Turing-complete stack-based programming language (ANSI, 1994; Brodie, 1980). We chose Forth as the host language of our work because i) it is an established, general-purpose high-level language relatively close to machine code, ii) it promotes highly modular programs through use of branching, loops and function calls, thus bringing out a good balance between assembly and higher level languages, and importantly iii) its abstract machine is simple enough for a straightforward creation of its continuous approximation. Forth's underlying abstract machine is represented by a state $S = (D, R, H, c)$, which contains two stacks: a data evaluation pushdown stack D (*data stack*) holds values for manipulation, and a return address pushdown stack R (*return stack*) assists with return pointers and subroutine calls. These are accompanied by a *heap* or random memory access buffer H, and a program counter c.

A Forth program P is a sequence[1] of Forth *words* (i.e. commands) $P = w_1...w_n$. The role of a word varies, encompassing language keywords, primitives, and user-defined subroutines (e.g. DROP discards the top element of the data stack, or DUP duplicates the top element of the data stack).[2] Each word w_i defines a transition function between machine states $w_i : S \to S$. Therefore, a program P itself defines a transition function by simply applying the word at the current program counter to the current state. Although usually considered as a part of the heap H, we consider Forth programs P separately to ease the analysis.

An example of a Bubble sort algorithm implemented in Forth is shown in Listing 1 in everything except lines 3b-4c. The execution starts from line 12 where literals are pushed on the data stack and the SORT is called. Line 10 executes the main loop over the sequence. Lines 2-7

---

[1]Forth is a concatenative language.

[2]In this work, we restrict ourselves to a subset of all Forth words, detailed in Appendix A.

```
1  : BUBBLE ( a1 ... an n-1 -- one pass )
2      DUP IF >R
3a         OVER OVER < IF SWAP THEN
4a         R> SWAP >R 1- BUBBLE R>
3b         { observe D0 D-1 -> permute D-1 D0 R0}
4b         1- BUBBLE R>
3c         { observe D0 D-1 -> choose NOP SWAP }
4c         R> SWAP >R 1- BUBBLE R>
5      ELSE
6         DROP
7      THEN
8  ;
9  : SORT ( a1 .. an n -- sorted  )
10     1- DUP 0 DO >R R@ BUBBLE R> LOOP DROP
11 ;
12 2 4 2 7 4 SORT \ Example call
```

Listing 1: Three code alternatives (white lines are common to all, coloured/lettered lines are alternative-specific): i) Bubble sort in Forth (a lines – green), ii) PERMUTE sketch (b lines – blue), and iii) COMPARE sketch (c lines – yellow).

denote the BUBBLE procedure – comparison of top two stack numbers (line 3a), and the recursive call to itself (line 4a). A detailed description of how this program is executed by the Forth abstract machine is provided in Appendix B. Notice that while Forth provides common control structures such as looping and branching, these can always be reduced to low-level code that uses jumps and conditional jumps (using the words BRANCH and BRANCH0, respectively). Likewise, we can think of sub-routine definitions as labelled code blocks, and their invocation amounts to jumping to the code block with the respective label.

3. $\partial 4$: Differentiable Abstract Machine

When a programmer writes a Forth program, they define a sequence of Forth words, *i.e.*, a sequence of *known* state transition functions. In other words, the programmer knows *exactly* how computation should proceed. To accommodate for cases when the developer's procedural background knowledge is incomplete, we extend Forth to support the definition of a program *sketch*. As is the case with Forth programs, sketches are sequences of transition functions. However, a sketch may contain transition functions whose behaviour is learned from data.

To learn the behaviour of transition functions within a program we would like the machine output to be differentiable with respect to these functions (and possibly representations of inputs to the program). This enables us to choose parameterised transition functions such as neural networks.

To this end, we introduce $\partial 4$, a TensorFlow (Abadi et al., 2015) implementation of a differentiable abstract machine with continuous state representations, differentiable words and sketches. Program execution in $\partial 4$ is modelled by a recurrent neural network (RNN), parameterised by the transition functions at each time step.
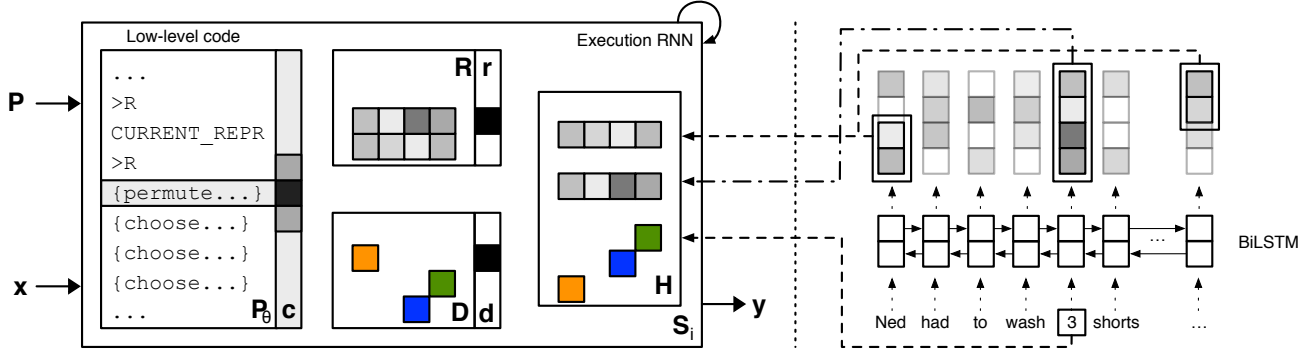
Figure 1: *Left:* Neural Forth Abstract Machine. A forth sketch $\mathbf{P}$ is translated to a low-level code $\mathbf{P}_\theta$, with slots $\{\,\dots\,\}$ substituted by a parametrised neural networks. Slots are learnt from input-output examples $(\mathbf{x},\mathbf{y})$ through the differentiable machine whose state $\mathbf{S}_i$ comprises the low-level code, program counter $\mathbf{c}$, data stack $\mathbf{D}$ (with pointer $\mathbf{d}$), return stack $\mathbf{R}$ (with pointer $\mathbf{r}$), and the heap $\mathbf{H}$. *Right:* BiLSTM trained on Word Algebra Problems. Output vectors corresponding to a representation of the entire problem, as well as context representations of numbers and the numbers themselves are fed into $\mathbf{H}$ to solve tasks. The entire system is end-to-end differentiable.

3.1. Machine State Encoding

We map the symbolic machine state $S = (D, R, H, c)$ to a continuous representation $\mathbf{S} = (\mathcal{D}, \mathcal{R}, \mathbf{H}, \mathbf{c})$ into two differentiable stacks (with pointers), the data stack $\mathcal{D} = (\mathbf{D}, \mathbf{d})$ and the return stack $\mathcal{R} = (\mathbf{R}, \mathbf{r})$, a heap $\mathbf{H}$, and an attention vector $\mathbf{c}$ indicating which word of the sketch $\mathbf{P}_\theta$ is being executed at the current time step. Figure 1 depicts the machine together with its elements. All three memory structures, the data stack, the return stack and the heap, are based on differentiable flat memory buffers $\mathbf{M} \in \{\mathbf{D}, \mathbf{R}, \mathbf{H}\}$, where $\mathbf{D}, \mathbf{R}, \mathbf{H} \in \mathbb{R}^{l \times v}$, for a stack size l and a value size v. Each has a differentiable read operation

$$\text{read}_\mathbf{M}(\mathbf{a}) = \mathbf{a}^T \mathbf{M}$$

and write operation

$$\text{write}_\mathbf{M}(\mathbf{x}, \mathbf{a}) : \mathbf{M} \leftarrow \mathbf{M} - (\mathbf{a}\mathbf{1}^T) \odot \mathbf{M} + \mathbf{x}\mathbf{a}^T$$

akin to the Neural Turing Machine (NTM) memory (Graves et al., 2014), where $\odot$ is the element-wise multiplication, and $\mathbf{a}$ is the address pointer.[3] In addition to the memory buffers $\mathbf{D}$ and $\mathbf{R}$, the data stack and the return stack contain pointers to the current top-of-the-stack (TOS) element $\mathbf{d}, \mathbf{r} \in \mathbb{R}^l$, respectively. This allows us to implement pushing as writing a value $\mathbf{x}$ into $\mathbf{M}$ and incrementing the TOS pointer as:

$$\text{push}_\mathbf{M}(\mathbf{x}) : \text{write}_\mathbf{M}(\mathbf{x}, \mathbf{p}) \qquad (\text{side-effect: } \mathbf{p} \leftarrow \text{inc}(\mathbf{p}))$$

where $\mathbf{p} \in \{\mathbf{d}, \mathbf{r}\}$, $\text{inc}(\mathbf{p}) = \mathbf{p}^T \mathbf{R}^{1+}$, $\text{dec}(\mathbf{p}) = \mathbf{p}^T \mathbf{R}^-$, and $\mathbf{R}^{1+}$ and $\mathbf{R}^{1-}$ are increment and decrement matrices (left and right circular shift matrices).

Popping is realized by multiplying the TOS pointer and the memory buffer, and decreasing the TOS pointer:

$$\text{pop}_\mathbf{M}(\,) = \text{read}_\mathbf{M}(\mathbf{p}) \qquad (\text{side-effect: } \mathbf{p} \leftarrow \text{dec}(\mathbf{p}))$$

Finally, the program counter $\mathbf{c} \in \mathbb{R}^p$ is a vector that, when one-hot, points to a single word in a program of length p, and is equivalent to the c vector of the symbolic state machine.[4] We use $\mathcal{S}$ to denote the space of all continuous representations $\mathbf{S}$.

Neural Forth Words It is straightforward to convert Forth words, defined as functions on discrete machine states, to functions operating on the continuous space $\mathcal{S}$. For example, consider the word DUP, which duplicates the top of the data stack. A differentiable version of DUP first calculates the value $\mathbf{e}$ on the TOS address of $\mathbf{D}$, as $\mathbf{e} = \mathbf{d}^T \mathbf{D}$. It then shifts the stack pointer via $\mathbf{d} \leftarrow \text{inc}(\mathbf{d})$, and writes $\mathbf{e}$ to $\mathbf{D}$ using $\text{write}_\mathbf{D}(\mathbf{e}, \mathbf{d})$. The complete description of implemented Forth Words and their differentiable counterparts can be found in Appendix A.

3.2. Forth Sketches

We define a Forth sketch $\mathbf{P}_\theta$ as a sequence of continuous transition functions $\mathbf{P} = \mathbf{w}_1 \dots \mathbf{w}_n$. Here, $\mathbf{w}_i \in \mathcal{S} \to \mathcal{S}$ either corresponds to a neural Forth word or a trainable transition function (neural networks in our case). We will call these trainable functions *slots*, as they correspond to underspecified "slots" in the program code that need to be filled by learned behaviour.

We allow users to define a slot $\mathbf{w}$ by specifying a pair of a state encoder $\mathbf{w}_{\text{enc}}$ and a decoder $\mathbf{w}_{\text{dec}}$. The encoder

---

[3] The equal widths of $\mathbf{H}$ and $\mathbf{D}$ allow us to directly move vector representations of values between the heap and the stack.

[4] During training $\mathbf{c}$ can become distributed and is considered as attention over the program code.

produces a latent representation $\mathbf{h}$ of the current machine state using a multi-layer perceptron, and the decoder consumes this representation to produce the next machine state. We hence have $\mathbf{w} = \mathbf{w}_{\text{dec}} \circ \mathbf{w}_{\text{enc}}$. To use slots within Forth program code, we introduce a notation that reflects this decomposition. In particular, slots are defined by the syntax `{ encoder -> decoder }` where `encoder` and `decoder` are specifications of the corresponding slot parts as described below.

Encoders   We provide the following options for encoders:

static produces a static representation, independent of the actual machine state.

observe $e_1...e_m$: concatenates the elements $e_1...e_m$ of the machine state. An element can be a stack item `Di` at relative index i, a return stack item `Ri`, etc.

linear N, sigmoid, tanh represent chained transformations, which enable the multilayer perceptron architecture. Linear N projects to N dimensions, and sigmoid and tanh apply same-named functions elementwise.

Decoders   Users can specify the following decoders:

choose $w_1...w_m$: chooses from the Forth words $w_1...w_m$. Takes an input vector $\mathbf{h}$ of length m to produce a weighted combination of machine states $\sum_i^m h_i \mathbf{w_i}(\mathbf{S})$.

manipulate $e_1...e_m$: directly manipulates the machine state elements $e_1 ... e_m$ by writing the appropriately reshaped and softmaxed output of the encoder over the machine state elements with write$_{\mathbf{M}}$.

permute $e_1...e_m$: permutes the machine state elements $e_1...e_m$ via a linear combination of $m!$ state vectors.

3.3. The Execution RNN

We model execution using an RNN which produces a state $\mathbf{S}_{n+1}$ conditioned on a previous state $\mathbf{S}_n$. It does so by first passing the current state to each function $\mathbf{w}_i$ in the program, and then weighing each of the produced next states by the component of the program counter vector $\mathbf{c}_i$ that corresponds to program index i, effectively using $\mathbf{c}$ as an attention vector over code. Formally we have:

$$\mathbf{S}_{n+1} = \text{RNN}(\mathbf{S}_n, \mathbf{P}_\theta) = \sum_{i=1}^{|P|} \mathbf{c}_i \mathbf{w}_i(\mathbf{S}_n)$$

Clearly, this recursion, and its final state, are differentiable with respect to the program code $\mathbf{P}_\theta$, and its inputs. Furthermore, for differentiable Forth programs the final state of this RNN will correspond to the final state of a symbolic execution (when no slots are present, and one-hot values are used).
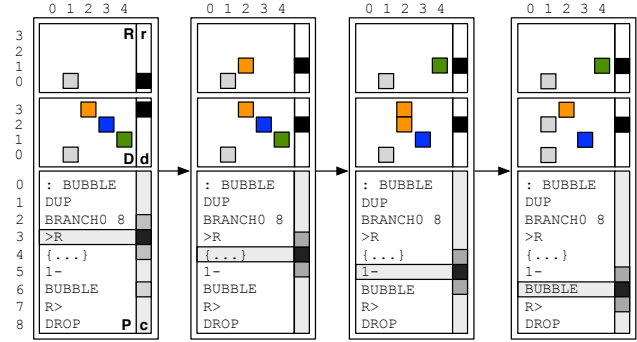

Figure 2: $\partial 4$ segment of the RNN execution of a Forth sketch in blue in Listing 1. The pointers $(\mathbf{d}, \mathbf{r})$ and values (rows of $\mathbf{R}$ and $\mathbf{D}$) are all in one-hot state (colours simply denote values observed, defined by the top scale), while the program counter maintains the uncertainty. Subsequent states are discretised for clarity. Here, the slot $\{\ldots\}$ has learned its optimal behaviour.

3.4. Program Code Optimisations

The $\partial 4$ RNN requires one-time step per transition. After each time step, the program counter is either incremented, decremented, explicitly set or popped from the stack. In turn, a new machine state is calculated by executing all words in the program and then weighting the result states by the program counter. As this is expensive, it is advisable to avoid full RNN steps wherever possible. We use two strategies to avoid full RNN steps and significantly speed-up $\partial 4$: symbolic execution and interpolation of if-branches.

Symbolic Execution   Whenever we have a sequence of Forth words that contains no branch entry or exit points, we can collapse this sequence into a single transition instead of naively interpreting words one-by-one. We symbolically execute (King, 1976) a sequence of Forth words to calculate a new machine state. We then use the difference between the new and the initial state to derive the transition function of the sequence. For example, the sequence `R> SWAP >R` that swaps top elements of the data and the return stack yields the symbolic state $D = r_1 d_2...d_l.$ and $R = d_1 r_2...r_l.$ Comparing it to the initial state, we derive a single neural transition that only needs to swap the top elements of $\mathbf{D}$ and $\mathbf{R}$.

Interpolation of If-Branches   We cannot apply symbolic execution to code with branching points as the branching behaviour depends on the current machine state, and we cannot resolve it symbolically. However, we can still collapse if-branches that involve no function calls or loops by executing both branches in parallel and weighing their output states by the value of the condition. If the if-branch does contain function calls or loops, we simply fall back to execution of all words weighted by the program counter.

3.5. Training

Our training procedure assumes input-output pairs of machine start and end states $(\mathbf{x}_i, \mathbf{y}_i)$ only. The output $\mathbf{y}_i$ defines a target memory $\mathbf{Y}_i^D$ and a target pointer $\mathbf{y}_i^d$ on the data stack $\mathbf{D}$. Additionally, we have a mask $\mathbf{K}_i$ that indicates which components of the stack should be included in the loss (e.g. we do not care about values above the stack depth). We use $\mathbf{D}_T(\theta, \mathbf{x}_i)$ and $\mathbf{d}_T(\theta, \mathbf{x}_i)$ to denote the final state of $\mathbf{D}$ and $\mathbf{d}$ after T steps of execution RNN and using an initial state $\mathbf{x}_i$. We define the loss function as

$$\mathcal{L}(\theta) = \mathcal{H}(\mathbf{K}_i \odot \mathbf{D}_T(\theta, \mathbf{x}_i), \mathbf{K}_i \odot \mathbf{Y}_i^D)$$
$$+ \mathcal{H}(\mathbf{K}_i \odot \mathbf{d}_T(\theta, \mathbf{x}_i), \mathbf{K}_i \odot \mathbf{y}_i^d)$$

where $\mathcal{H}(\mathbf{x}, \mathbf{y}) = -\mathbf{x} \log \mathbf{y}$ is the cross-entropy loss, and θ are parameters of slots in the program P. We can use backpropagation and any variant of gradient descent to optimise this loss function. Note that at this point it would be possible to include supervision of the intermediate states (trace-level), as done by the Neural Program Interpreter (Reed & de Freitas, 2015).

4. Experiments

We evaluate $\partial 4$ on three tasks. Two of these are simple transduction tasks, sorting and addition as presented in (Reed & de Freitas, 2015), with varying levels of program structure. For each problem, we introduce two sketches.

We also test $\partial 4$ on the more difficult task of answering word algebra problems. We show that not only can $\partial 4$ act as a standalone solver for such problems, bypassing the intermediary task of producing formula templates which must then be executed, but it can also outperform previous work when trained on the same data.

4.1. Experimental Setup

Specific to the transduction tasks, we discretise memory elements during testing. This effectively allows the trained model to generalise to any sequence length if the correct sketch behaviour has been learned. We also compare against a Seq2Seq (Sutskever et al., 2014) baseline. Full details of the experimental setup can be found in Appendix E.

4.2. Sorting

Sorting sequences of digits is a hard task for RNNs, as they fail to generalise to sequences even marginally longer than the ones they have been trained on (Reed & de Freitas, 2015). We investigate several strong priors based on Bubble sort for this transduction task and present two $\partial 4$ sketches in Listing 1 that enable us to learn sorting from only a few hundred training examples (see Appendix C.1 for more detail):

Table 1: Accuracy (Hamming distance) of Permute and Compare sketches in comparison to a Seq2Seq baseline on the sorting problem.

	Test Length 8			Test Length: 64		
Train Length:	2	3	4	2	3	4
Seq2Seq	26.2	29.2	39.1	13.3	13.6	15.9
$\partial 4$ Permute	100.0	100.0	19.82	100.0	100.0	7.81
$\partial 4$ Compare	100.0	100.0	49.22	100.0	100.0	20.65

PERMUTE. A sketch specifying that the top two elements of the stack, and the top of the return stack must be permuted based on the values of the former (line 3b). Both the value comparison and the permutation behaviour must be learned. The core of this sketch is depicted in Listing 1 (b lines), and the sketch is explained in detail in Appendix D.

COMPARE. This sketch provides additional prior procedural knowledge to the model. In contrast to PERMUTE, only the comparison between the top two elements on the stack must be learned (line 3c). The core of this sketch is depicted in Listing 1 (c lines).

In both sketches, the outer loop can be specified in $\partial 4$ (Listing 1, line 10), which repeatedly calls a function BUBBLE. In doing so, it defines sufficient structure so that the behaviour of the network is invariant to the input sequence length.

Results on Bubble sort A quantitative comparison of our models on the Bubble sort task is provided in Table 1. For a given test sequence length, we vary the training set lengths to illustrate the model's ability to generalise to sequences longer than those it observed during training. We find that $\partial 4$ quickly learns the correct sketch behaviour, and it is able to generalise perfectly to sort sequences of 64 elements after observing only sequences of length two and three during training. In comparison, the Seq2Seq baseline falters when attempting similar generalisations, and performs close to chance when tested on longer sequences. Both $\partial 4$ sketches perform flawlessly when trained on short sequence lengths, but under-perform when trained on sequences of length 4 due to arising computational difficulties (COMPARE sketch performs better due to more structure it imposes). We discuss this issue further in Section 5.

4.3. Addition

Next, we applied $\partial 4$ to the problem of learning to add two n-digit numbers. We rely on the standard elementary school addition algorithm, where the goal is to iterate over pairs of aligned digits, calculating the sum of each to yield the resulting sum. The key complication arises when two digits sum to a two-digit number, requiring that the correct extra digit (a *carry*) be carried over to the subsequent column.

```
1  : ADD-DIGITS
     ( a1 b1...an bn carry n -- r1 r2...r_{n+1} )
2    DUP 0 = IF
3      DROP
4    ELSE
5      >R \ put n on R
6a     { observe D0 D-1 D-2 -> tanh -> linear 70
         -> manipulate D-1 D-2 }
7a     DROP
6b     { observe D0 D-1 D-2 -> tanh -> linear 10
         -> choose 0 1 }
7b     { observe D-1 D-2 D-3 -> tanh -> linear 50
         -> choose 0 1 2 3 4 5 6 7 8 9 }
       >R SWAP DROP SWAP DROP SWAP DROP R>
8      R> 1- SWAP >R  \ new_carry n-1
9      ADD-DIGITS \ call add-digits on n-1 subseq.
10     R> \ put remembered results back on the stack
11   THEN
12 ;
```

Listing 2: Manipulate sketch (a lines – green) and the choose sketch (b lines – blue) for Elementary Addition. Input data is used to fill data stack externally

We assume aligned pairs of digits as input, with a carry for the least significant digit (potentially 0), and the length of the respective numbers. The sketches define the high-level operations through recursion, leaving the core addition to be learned from data.

The specified high-level behaviour includes the recursive call template and the halting condition of the recursion (no remaining digits, line 2-3). The underspecified addition operation must take three digits from the previous call, the two digits to sum and a previous carry, and produce a single digit (the sum) and the resultant carry (lines 6a, 6b and 7a, 7b). We introduce two sketches for inducing this behaviour:

MANIPULATE. This sketch provides little prior procedural knowledge as it directly manipulates the $\partial 4$ machine state, filling in a carry and the result digits, based on the top three elements on the data stack (two digits and the carry). Depicted in Listing 2 in green.

CHOOSE. Incorporating additional prior information, CHOOSE exactly specifies the results of the computation, namely the output of the first slot (line 6b) is the carry, and the output of the second one (line 7b) is the result digit, both conditioned on the two digits and the carry on the data stack. Depicted in Listing 2 in blue.

The rest of the sketch code reduces the problem size by one and returns the solution by popping it from the return stack.

Quantitative Evaluation on Addition In a set of experiments analogous to those in our evaluation on Bubble sort, we demonstrate the performance of $\partial 4$ on the addition task by examining test set sequence lengths of 8 and 64 while varying the lengths of the training set instances (Table 2). The Seq2Seq model again fails to generalise

Table 2: Accuracy (Hamming distance) of Choose and Manipulate sketches in comparison to a Seq2Seq baseline on the addition problem. Note that lengths corresponds to the length of the input sequence (two times the number of digits of both numbers).

	Test Length 8			Test Length 64		
Train Length:	2	4	8	2	4	8
Seq2Seq	37.9	57.8	99.8	15.0	13.5	13.3
$\partial 4$ Choose	100.0	100.0	100.0	100.0	100.0	100.0
$\partial 4$ Manipulate	98.58	100.0	100.0	99.49	100.0	100.0

to longer sequences than those observed during training. In comparison, both the CHOOSE sketch and the less structured MANIPULATE sketch learn the correct sketch behaviour and generalise to all test sequence lengths (with an exception of MANIPULATE which required more data to train perfectly). In additional experiments, we were able to successfully train both the CHOOSE and the MANIPULATE sketches from sequences of input length 24, and we tested them up to the sequence length of 128, confirming their perfect training and generalisation capabilities.

4.4. Word Algebra Problems

Word algebra problems (WAPs) are often used to assess the numerical reasoning abilities of schoolchildren. Questions are short narratives which focus on numerical quantities, culminating with a question. For example:

A florist had 50 roses. If she sold 15 of them and then later picked 21 more, how many roses would she have?

Answering such questions requires both the understanding of language and of algebra — one must know which numeric operations correspond to which phrase and how to execute these operations.

Previous work cast WAPs as a transduction task by mapping a question to a template of a mathematical formula, thus requiring manuall labelled formulas. For instance, one formula that can be used to correctly answer the question in the example above is `(50 - 15) + 21 = 56`. In previous work, local classifiers (Roy & Roth, 2015; Roy et al., 2015), hand-crafted grammars (Koncel-Kedziorski et al., 2015), and recurrent neural models (Bouchard et al., 2016) have been used to perform this task. Predicted formula templates may be marginalised during training (Kushman et al., 2014), or evaluated directly to produce an answer.

In contrast to these approaches, $\partial 4$ is able to learn both, a soft mapping from text to algebraic operations *and their execution*, without the need for manually labelled equations and no explicit symbolic representation of a formula.

Model description Our model is a fully end-to-end differentiable structure, consisting of a $\partial 4$ interpreter, a

```
   \ first copy data from H: vectors to R and numbers to D
1  { observe R0 R-1 R-2 R-3 -> permute D0 D-1 D-2 }
2  { observe R0 R-1 R-2 R-3 -> choose + - * / }
3  { observe R0 R-1 R-2 R-3 -> choose SWAP NOP }
4  { observe R0 R-1 R-2 R-3 -> choose + - * / }
   \ lastly, empty out the return stack
```

Listing 3: Core of the Word Algebra Problem sketch. The full sketch can be found in the Appendix.

sketch, and a Bidirectional LSTM (BiLSTM) reader.

The BiLSTM reader reads the text of the problem and produces a vector representation (word vectors) for each word, concatenated from the forward and the backward pass of the BiLSTM network. We use the resulting word vectors corresponding only to numbers in the text, numerical values of those numbers (encoded as one-hot vectors), and a vector representation of the whole problem (concatenation of the last and the first vector of the opposite passes) to initialise the $\partial 4$ heap **H**. This is done in an end-to-end fashion, enabling gradient propagation through the BiLSTM to the vector representations. The process is depicted in Figure 1.

The sketch, depicted in Listing 3 dictates the differentiable computation.[5] First, it copies values from the heap **H** – word vectors to the return stack **R**, and numbers (as one-hot vectors) on the data stack **D**. Second, it contains four slots that define the space of all possible operations of four operators on three operands, all conditioned on the vector representations on the return stack. These slots are i) permutation of the elements on the data stack, ii) choosing the first operator, iii) possibly swapping the intermediate result and the last operand, and iv) the choice of the second operator. The final set of commands simply empties out the return stack **R**. These slots define the space of possible operations, however, the model needs to learn how to utilise these operations in order to calculate the correct result.

Results   We evaluate the model on the Common Core (CC) dataset, introduced by Roy & Roth (2015). CC is notable for having the most diverse set of equation patterns, consisting of four operators $(+, -, \times, \div)$, with up to three operands.

We compare against three baseline systems: (1) a local classifier with hand-crafted features (Roy & Roth, 2015), (2) a Seq2Seq baseline, and (3) the same model with a data generation component (GeNeRe) Bouchard et al. (2016). All baselines are trained to predict the best equation, which is executed outside of the model to obtain the answer. In contrast, $\partial 4$ is trained end-to-end from input-output pairs and predicts the answer directly without the need for an intermediate symbolic representation of a formula.

Results are shown in Table 3. All RNN-based methods

---
[5]Due to space constraints, we present the core of the sketch here. For the full sketch, please refer to Listing 4 in the Appendix.

Table 3: Accuracies of models on the CC dataset. Asterisk denotes results obtained from Bouchard et al. (2016). Note that GeNeRe makes use of additional data

Model	Accuracy (%)
Template Mapping	
Roy & Roth (2015)	55.5
Seq2Seq* (Bouchard et al., 2016)	95.0
GeNeRe* (Bouchard et al., 2016)	98.5
Fully End-to-End	
$\partial 4$	96.0

(bottom three) outperform the classifier-based approach. Our method slightly outperforms a Seq2Seq baseline, achieving the highest reported result on this dataset without data augmentation.

5. Discussion

$\partial 4$ bridges the gap between a traditional programming language and a modern machine learning architecture. However, as we have seen in our evaluation experiments, faithfully simulating the underlying abstract machine architecture introduces its own unique set of challenges.

One such challenge is the additional complexity of performing even simple tasks when they are viewed in terms of operations on the underlying machine state. As illustrated in Table 1, $\partial 4$ sketches can be effectively trained from small training sets (see Appendix C.1), and generalise perfectly to sequences of any length. However, difficulty arises when training from sequences of modest lengths. Even when dealing with relatively short training length sequences, and with the program code optimisations employed, the underlying machine can unroll into a problematically large number states. For problems whose machine execution is quadratic, like the sorting task (which at input sequences of length 4 has 120 machine states), we observe significant instabilities during training from backpropagating through such long RNN sequences, and consequent failures to train. In comparison, the addition problem was easier to train due to a comparatively shorter underlying execution RNNs.

The higher degree of prior knowledge provided played an important role in successful learning. For example, the COMPARE sketch, which provides more structure, achieves higher accuracies when trained on longer sequences. Similarly, employing softmax on the directly manipulated memory elements enabled perfect training for the MANIPULATE sketch for addition. Furthermore, it is encouraging to see that $\partial 4$ can be trained jointly with an upstream LSTM to provide strong procedural prior knowledge for solving a real-world NLP task.

6. Related Work

Program Synthesis The idea of program synthesis is as old as Artificial Intelligence, and has a long history in computer science (Manna & Waldinger, 1971). Whereas a large body of work has focused on using genetic programming (Koza, 1992) to induce programs from the given input-output specification (Nordin, 1997), there are also various Inductive Programming approaches (Kitzelmann, 2009) aimed at inducing programs from incomplete specifications of the code to be implemented (Albarghouthi et al., 2013; Solar-Lezama et al., 2006). We tackle the same problem of sketching, but in our case, we fill the sketches with neural networks able to learn the slot behaviour.

Probabilistic and Bayesian Programming Our work is closely related to probabilistic programming languages such as Church (Goodman et al., 2008). They allow users to inject random choice primitives into programs as a way to define generative distributions over possible execution traces. In a sense, the random choice primitives in such languages correspond to the slots in our sketches. A core difference lies in the way we train the behaviour of slots: instead of calculating their posteriors using probabilistic inference, we estimate their parameters using backpropagation and gradient descent. This is similar in-kind to `TerpreT`'s FMGD algorithm (Gaunt et al., 2016), which is employed for code induction via backpropagation. In comparison, our model which optimises slots of neural networks surrounded by continuous approximations of code, enables the combination of procedural behaviour and neural networks. In addition, the underlying programming and probabilistic paradigm in these programming languages is often functional and declarative, whereas our approach focuses on a procedural and discriminative view. By using an end-to-end differentiable architecture, it is easy to seamlessly connect our sketches to further neural input and output modules, such as an LSTM that feeds into the machine heap.

Neural approaches Recently, there has been a surge of research in program synthesis, and execution in deep learning, with increasingly elaborate deep models. Many of these models were based on differentiable versions of abstract data structures (Joulin & Mikolov, 2015; Grefenstette et al., 2015; Kurach et al., 2016), and a few abstract machines, such as the NTM (Graves et al., 2014), Differentiable Neural Computers (Graves et al., 2016), and Neural GPUs (Kaiser & Sutskever, 2015). All these models are able to induce algorithmic behaviour from training data. Our work differs in that our differentiable abstract machine allows us to seemingly integrate code and neural networks, and train the neural networks specified by slots via backpropagation. Related to our efforts is also the Autograd (Maclaurin et al., 2015), which enables automatic gradient computation in pure Python code, but does not define nor use differentiable access to its underlying abstract machine.

The work in neural approximations to abstract structures and machines naturally leads to more elaborate machinery able to induce and call code or code-like behaviour. Neelakantan et al. (2015a) learned simple SQL-like behaviour——querying tables from the natural language with simple arithmetic operations. Although sharing similarities on a high level, the primary goal of our model is not induction of (fully expressive) code but its injection. (Andreas et al., 2016) learn to *compose* neural modules to produce the desired behaviour for a visual QA task. Neural Programmer-Interpreters (Reed & de Freitas, 2015) learn to represent and execute programs, operating on different modes of an environment, and are able to incorporate decisions better captured in a neural network than in many lines of code (e.g. using an image as an input). Users inject prior procedural knowledge by training on program traces and hence require *full* procedural knowledge. In contrast, we enable users to use their partial knowledge in sketches.

Neural approaches to language compilation have also been researched, from compiling a language into neural networks (Siegelmann, 1994), over building neural compilers (Gruau et al., 1995) to adaptive compilation (Bunel et al., 2016). However, that line of research did not perceive neural interpreters and compilers as a means of injecting procedural knowledge as we did. To the best of our knowledge, $\partial 4$ is the first working neural implementation of an abstract machine for an actual programming language, and this enables us to inject such priors in a straightforward manner.

7. Conclusion and Future Work

We have presented $\partial 4$, a differentiable abstract machine for the Forth programming language, and showed how it can be used to complement programmers' prior knowledge through the learning of unspecified behaviour in Forth sketches. The $\partial 4$ RNN successfully learns to sort and add, and solve word algebra problems, using only program sketches and program input-output pairs. We believe $\partial 4$, and the larger paradigm it helps establish, will be useful for addressing complex problems where low-level representations of the input are necessary, but higher-level reasoning is difficult to learn and potentially easier to specify.

In future work, we plan to apply $\partial 4$ to other problems in the NLP domain, like machine reading and knowledge base inference. In the long-term, we see the integration of non-differentiable transitions (such as those arising when interacting with a real environment), as an exciting future direction which sits at the intersection of reinforcement learning and probabilistic programming.

References

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S., Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Goodfellow, Ian, Harp, Andrew, Irving, Geoffrey, Isard, Michael, Jia, Yangqing, Jozefowicz, Rafal, Kaiser, Lukasz, Kudlur, Manjunath, Levenberg, Josh, Mané, Dan, Monga, Rajat, Moore, Sherry, Murray, Derek, Olah, Chris, Schuster, Mike, Shlens, Jonathon, Steiner, Benoit, Sutskever, Ilya, Talwar, Kunal, Tucker, Paul, Vanhoucke, Vincent, Vasudevan, Vijay, Viégas, Fernanda, Vinyals, Oriol, Warden, Pete, Wattenberg, Martin, Wicke, Martin, Yu, Yuan, and Zheng, Xiaoqiang. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

Albarghouthi, Aws, Gulwani, Sumit, and Kincaid, Zachary. Recursive program synthesis. In *Computer Aided Verification*, pp. 934–950. Springer, 2013.

Andreas, Jacob, Rohrbach, Marcus, Darrell, Trevor, and Klein, Dan. Neural module networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

ANSI. *Programming Languages - Forth*, 1994. American National Standard for Information Systems, ANSI X3.215-1994.

Bouchard, Guillaume, Stenetorp, Pontus, and Riedel, Sebastian. Learning to generate textual data. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1608–1616, 2016.

Brodie, Leo. *Starting Forth*. Forth Inc., 1980.

Bunel, Rudy, Desmaison, Alban, Kohli, Pushmeet, Torr, Philip HS, and Kumar, M Pawan. Adaptive neural compilation. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*, 2016.

Gaunt, Alexander L, Brockschmidt, Marc, Singh, Rishabh, Kushman, Nate, Kohli, Pushmeet, Taylor, Jonathan, and Tarlow, Daniel. TerpreT: A Probabilistic Programming Language for Program Induction. *arXiv preprint arXiv:1608.04428*, 2016.

Goodman, Noah, Mansinghka, Vikash, Roy, Daniel M, Bonawitz, Keith, and Tenenbaum, Joshua B. Church: a language for generative models. In *Proceedings of the Conference in Uncertainty in Artificial Intelligence (UAI)*, pp. 220–229, 2008.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural Turing Machines. *arXiv preprint arXiv:1410.5401*, 2014.

Graves, Alex, Wayne, Greg, Reynolds, Malcolm, Harley, Tim, Danihelka, Ivo, Grabska-Barwińska, Agnieszka, Colmenarejo, Sergio Gómez, Grefenstette, Edward, Ramalho, Tiago, Agapiou, John, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

Grefenstette, Edward, Hermann, Karl Moritz, Suleyman, Mustafa, and Blunsom, Phil. Learning to Transduce with Unbounded Memory. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*, pp. 1819–1827, 2015.

Gruau, Frédéric, Ratajszczak, Jean-Yves, and Wiber, Gilles. A Neural compiler. *Theoretical Computer Science*, 141 (1):1–52, 1995.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Joulin, Armand and Mikolov, Tomas. Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. In *Proceedings of the Conferences on Neural Information Processing Systems (NIPS)*, pp. 190–198, 2015.

Kaiser, Łukasz and Sutskever, Ilya. Neural GPUs learn algorithms. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

King, James C. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

Kingma, Diederik and Ba, Jimmy. Adam: A Method for Stochastic Optimization. In *Proceedings of the International Conference for Learning Representations (ICLR)*, 2015.

Kitzelmann, Emanuel. Inductive Programming: A Survey of Program Synthesis Techniques. In *International Workshop on Approaches and Applications of Inductive Programming*, pp. 50–73, 2009.

Koncel-Kedziorski, Rik, Hajishirzi, Hannaneh, Sabharwal, Ashish, Etzioni, Oren, and Ang, Siena. Parsing Algebraic Word Problems into Equations. *Transactions of the Association for Computational Linguistics (TACL)*, 3: 585–597, 2015.

Koza, John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, volume 1. MIT press, 1992.

Kurach, Karol, Andrychowicz, Marcin, and Sutskever, Ilya. Neural Random-Access Machines. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

Kushman, Nate, Artzi, Yoav, Zettlemoyer, Luke, and Barzilay, Regina. Learning to Automatically Solve Algebra Word Problems. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 271–281, 2014.

Lau, Tessa, Wolfman, Steven A., Domingos, Pedro, and Weld, Daniel S. Learning repetitive text-editing procedures with smartedit. In *Your Wish is My Command*, pp. 209–226. Morgan Kaufmann Publishers Inc., 2001.

Maclaurin, Dougal, Duvenaud, David, and Adams, Ryan P. Gradient-based Hyperparameter Optimization through Reversible Learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

Manna, Zohar and Waldinger, Richard J. Toward automatic program synthesis. *Communications of the ACM*, 14(3): 151–165, 1971.

Neelakantan, Arvind, Le, Quoc V, and Sutskever, Ilya. Neural Programmer: Inducing latent programs with gradient descent. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015a.

Neelakantan, Arvind, Vilnis, Luke, Le, Quoc V, Sutskever, Ilya, Kaiser, Lukasz, Kurach, Karol, and Martens, James. Adding Gradient Noise Improves Learning for Very Deep Networks. *arXiv preprint arXiv:1511.06807*, 2015b.

Nordin, Peter. *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.

Reed, Scott and de Freitas, Nando. Neural programmer-interpreters. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

Roy, Subhro and Roth, Dan. Solving General Arithmetic Word Problems. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1743–1752, 2015.

Roy, Subhro, Vieira, Tim, and Roth, Dan. Reasoning about quantities in natural language. *Transactions of the Association for Computational Linguistics (TACL)*, 3: 1–13, 2015.

Siegelmann, Hava T. Neural Programming Language. In *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*, pp. 877–882, 1994.

Solar-Lezama, Armando, Rabbah, Rodric, Bodík, Rastislav, and Ebcioğlu, Kemal. Programming by Sketching for Bit-streaming Programs. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pp. 281–294, 2005.

Solar-Lezama, Armando, Tancau, Liviu, Bodik, Rastislav, Seshia, Sanjit, and Saraswat, Vijay. Combinatorial Sketching for Finite Programs. In *ACM Sigplan Notices*, volume 41, pp. 404–415, 2006.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc V. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*, pp. 3104–3112, 2014.

Appendix

A. Forth Words and their implementation

We implemented a small subset of available Forth words in $\partial 4$. The table of these words, together with their descriptions is given in Table 4, and their implementation is given in Table 5. The commands are roughly divided into 7 groups. These groups, line-separated in the table, are:

Data stack operations {num}, 1+, 1-, DUP, SWAP, OVER, DROP, +, -, *, /
Heap operations @, !
Comparators >, <, =
Return stack operations >R, R>, @R
Control statements IF..ELSE..THEN, BEGIN..WHILE..REPEAT, DO..LOOP
Subroutine control :, {sub}, ;, MACRO
Variable creation VARIABLE, CREATE..ALLOT

Table 4: Forth words and their descriptions. TOS denotes top-of-stack, NOS denotes next-on-stack, DSTACK denotes the data stack, RSTACK denotes the return stack, and HEAP denotes the heap.

Forth Word	Description
{num}	Pushes {num} to DSTACK.
1+	Increments DSTACK TOS by 1.
1-	Decrements DSTACK TOS by 1.
DUP	Duplicates DSTACK TOS.
SWAP	Swaps TOS and NOS.
OVER	Copies NOS and pushes it on the TOS.
DROP	Pops the TOS (non-destructive).
+, -, *, /	Consumes DSTACK NOS and TOS. Returns NOS operator TOS.
@	Fetches the HEAP value from the DSTACK TOS address.
!	Stores DSTACK NOS to the DSTACK TOS address on the HEAP.
>, <, =	Consumes DSTACK NOS and TOS. Returns 1 (TRUE) if NOS $>$ | $<$ | $=$ TOS respectivelly, 0 (FALSE) otherwise.
>R	Pushes DSTACK TOS to RSTACK TOS, removes it from DSTACK.
R>	Pushes RSTACK TOS to DSTACK TOS, removes it from RSTACK.
@R	Copies the RSTACK TOS TO DSTACK TOS.
IF..ELSE..THEN	Consumes DSTACK TOS, if it equals to a non-zero number (TRUE), executes commands between IF and ELSE. Otherwise executes commands between ELSE and THEN.
BEGIN..WHILE..REPEAT	Continually executes commands between WHILE and REPEAT while the code between BEGIN and WHILE evaluates to a non-zero number (TRUE).
DO..LOOP	Consumes NOS and TOS, assumes NOS as a limit, and TOS as a current index. Increases index by 1 until equal to NOS. At every increment, executes commands between DO and LOOP.
:	Denotes the subroutine, followed by a word defining it.
{sub}	Subroutine invocation, puts the program counter PC on RSTACK, sets PC to the subroutine address.
;	Subroutine exit. Consumest TOS from the RSTACK and sets the PC to it.
MACRO	Treats the subroutine as a macro function.
VARIABLE	Creates a variable with a fixed address. Invoking the variable name returns its address.
CREATE..ALLOT	Creates a variable with a fixed address. Do not allocate the next N addresses to any other variable (effectively reserve that portion of heap to the variable)

Table 5: Implementation of Forth words described in Table 4. Note that the variable creation words are implemented as fixed address allocation, and `MACRO` words are implemented with inlining.

Symbol	Explanation
$\mathcal{M}$	Stack, $\mathcal{M} \in \{\mathcal{D},\mathcal{R}\}$
$\mathbf{M}$	Memory buffer, $\mathbf{M} \in \{\mathbf{D},\mathbf{R},\mathbf{H}\}$
$\mathbf{p}$	Pointer, $\mathbf{p} \in \{\mathbf{d},\mathbf{r},\mathbf{c}\}$
$\mathbf{R}^{1\pm}$	Increment and decrement matrices (circular shift)
	$\mathbf{R}^{1\pm}_{ij} = \begin{cases} 1 & i\pm 1 \equiv j (mod\,n)) \\ 0 & otherwise \end{cases}$
$\mathbf{R}^{+},\mathbf{R}^{-},\mathbf{R}^{*},\mathbf{R}^{/}$	Circular arithmetic operation tensors
	$\mathbf{R}^{\{\mathbf{op}\}}_{\mathbf{ijk}} = \begin{cases} 1 & i\{op\}j \equiv k (mod\,n)) \\ 0 & otherwise \end{cases}$
Pointer and value manipulation	**Expression**
Increment $\mathbf{a}$ (or value $\mathbf{x}$)	$inc(\mathbf{a}) = \mathbf{a}^T \mathbf{R}^{1+}$
Decrement $\mathbf{a}$ (or value $\mathbf{x}$)	$dec(\mathbf{a}) = \mathbf{a}^T \mathbf{R}^{1-}$
Algebraic operation application	$\{op\}(\mathbf{a},\mathbf{b}) = \mathbf{a}^T \mathbf{R}^{\{\mathbf{op}\}} \mathbf{b}$
Conditional jump $\mathbf{a}$	$jump(\mathbf{c},\mathbf{a}) : p = (pop_{\mathbf{D}}() = TRUE)$
	$\mathbf{c} \leftarrow p\mathbf{c} + (1-p)a$
$\mathbf{a}^{-1}$	Next on stack, $\mathbf{a} \leftarrow \mathbf{a}^T \mathbf{R}^{1-}$
Buffer manipulation	
READ from $\mathbf{M}$	$read_{\mathbf{M}}(\mathbf{a}) = \mathbf{a}^T \mathbf{M}$
WRITE to $\mathbf{M}$	$write_{\mathbf{M}}(\mathbf{x},\mathbf{a}) : \mathbf{M} \leftarrow \mathbf{M} - \mathbf{a} \otimes \mathbf{1} \cdot \mathbf{M} + \mathbf{x} \otimes \mathbf{a}$
PUSH $\mathbf{x}$ onto $\mathcal{M}$	$push_{\mathbf{M}}(\mathbf{x}) : write_{\mathbf{M}}(\mathbf{x},\mathbf{a})$ [side-effect: $\mathbf{d} \leftarrow inc(\mathbf{d})$]
POP an element from $\mathcal{M}$	$pop_{\mathbf{M}}() = read_{\mathbf{M}}(\mathbf{a})$ [side-effect: $\mathbf{d} \leftarrow dec(\mathbf{d})$]
Forth Word	
Literal $\mathbf{x}$	$push_{\mathbf{D}}(\mathbf{x})$
`1+`	$write_{\mathbf{D}}(inc(read_{\mathbf{D}}(\mathbf{d})),\mathbf{d})$
`1-`	$write_{\mathbf{D}}(dec(read_{\mathbf{D}}(\mathbf{d})),\mathbf{d})$
`DUP`	$push_{\mathbf{D}}(read_{\mathbf{D}}(\mathbf{d}))$
`SWAP`	$x = read_{\mathbf{D}}(\mathbf{d}), y = read_{\mathbf{D}}(\mathbf{d}^{-1})$
	$: write_{\mathbf{D}}(\mathbf{d},y) , write_{\mathbf{D}}(\mathbf{d}^{-1},x)$
`OVER`	$push_{\mathbf{D}}(read_{\mathbf{D}}(\mathbf{d}))$
`DROP`	$pop_{\mathbf{D}}()$
`+, -, *, /`	$write_{\mathbf{D}}(\{op\}(read_{\mathbf{D}}(\mathbf{d}^{-1}),read_{\mathbf{D}}(\mathbf{d})),\mathbf{d})$
`@`	$read_{\mathbf{H}}(\mathbf{d})$
`!`	$write_{\mathbf{H}}(\mathbf{d},\mathbf{d}^{-1})$
`<`	`SWAP >`
`>`	$e_1 = \sum_{i=0}^{n-1} i*\mathbf{d}_i, e_2 = \sum_{i=0}^{n-1} i*\mathbf{d}_i^{-1}$
	$p = \phi_{pwl}(\mathbf{e_1} - \mathbf{e_2})$, where $\phi_{pwl}(x) = min(max(0,x+0.5),1)$
	$p\mathbf{1} + (p-1)\mathbf{0}$
`=`	$p = \phi_{pwl}(\mathbf{d},\mathbf{d}^{-1})$
	$p\mathbf{1} + (p-1)\mathbf{0}$
`>R`	$push_{\mathbf{R}}(\mathbf{d})$
`R>`	$pop_{\mathbf{R}}()$
`@R`	$write_{\mathbf{D}}(\mathbf{d},read_{\mathbf{R}}(\mathbf{r}))$
`IF..₁ELSE..₂THEN`	$p = (pop_{\mathbf{D}}() = \mathbf{0})$
	$p*..1 + (1-p)*..2$
`BEGIN..₁WHILE..₂REPEAT`	$..1\, jump(c,..2)$
`DO..LOOP`	$start = \mathbf{c}, current = inc(pop_{\mathbf{D}}()), limit = pop_{\mathbf{D}}()$
	$p = (current = limit)$
	$jump(p,..),jump(c,start)$

B. Bubble sort algorithm description

An example of a Forth program that implements the Bubble sort algorithm is shown in Listing 1. We provide a description of how the first iteration of this algorithm is executed by the Forth abstract machine:

The program begins at line 11, putting the sequence [2 4 2 7] on the data stack D, followed by the sequence length 4.[6] It then calls the SORT word.

	D	R	c	comment
1	[]	[]	11	execution start
2	[2 4 2 7 4]	[]	8	pushing sequence to D, calling SORT subroutine puts A$_{SORT}$ to R

For a sequence of length 4, SORT performs a do-loop in line 9 that calls the BUBBLE sub-routine 3 times. It does so by decrementing the top of D with the 1− word to 3. Subsequently, 3 is duplicated on D by using DUP, and 0 is pushed onto D.

	D	R	c	
3	[2 4 2 7 3]	[A$_{SORT}$]	9	1-
4	[2 4 2 7 3 3]	[A$_{SORT}$]	9	DUP
6	[2 4 2 7 3 3 0]	[A$_{SORT}$]	9	0

DO consumes the top two stack elements 3 and 0 as the limit and starting point of the loop, leaving the stack D to be [2,4,2,7,3]. We use the return stack R as a temporary variable buffer and push 3 onto it using the word >R. This drops 3 from D, which we copy from R with R@

	D	R	c	
7	[2 4 2 7 3]	[Addr$_{SORT}$]	9	DO
8	[2 4 2 7]	[Addr$_{SORT}$ 3]	9	>R
9	[2 4 2 7 3]	[Addr$_{SORT}$ 3]	9	@R

Next, we call BUBBLE to perform one iteration of the bubble pass, (calling BUBBLE 3 times internally), and consuming 3. Notice that this call puts the current program counter onto R, to be used for the program counter c when exiting BUBBLE.

Inside the BUBBLE subroutine, DUP duplicates 3 on R. IF consumes the duplicated 3 and interprets is as TRUE. >R puts 3 on R.

	D	R	c	
10	[2 4 2 7 3]	[A$_{SORT}$ 3 A$_{BUBBLE}$]	0	calling BUBBLE subroutine puts A$_{BUBBLE}$ to R
11	[2 4 2 7 3 3]	[A$_{SORT}$ 3 A$_{BUBBLE}$]	1	DUP
12	[2 4 2 7 3]	[A$_{SORT}$ 3 A$_{BUBBLE}$]	1	IF
13	[2 4 2 7]	[A$_{SORT}$ 3 A$_{BUBBLE}$ 3]	1	>R

Calling OVER twice duplicates the top two elements of the stack, to test them with <, which tests whether $2 < 7$. IF tests if the result is TRUE (0), which it is, so it executes SWAP.

	D	R	c	
14	[2 4 2 7 2 7]	[A$_{SORT}$ 3 A$_{BUBBLE}$ 3]	2	OVER OVER
15	[2 4 2 7 1]	[A$_{SORT}$ 3 A$_{BUBBLE}$ 3]	2	<
16	[2 4 2 7]	[A$_{SORT}$ 3 A$_{BUBBLE}$ 3]	2	IF
17	[2 4 7 2]	[A$_{SORT}$ 3 A$_{BUBBLE}$ 3]	2	SWAP

To prepare for the next call to BUBBLE we move 3 back from the return stack R to the data stack D via R>, SWAP it with the next element, put it back to R with >R, decrease the TOS with 1− and invoke BUBBLE again. Notice that R will accumulate the analysed part of the sequence, which will be recursively taken back.

	D	R	c	
18	[2 4 7 2 3]	[A$_{SORT}$ 3 A$_{BUBBLE}$]	3	R >
19	[2 4 7 3 2]	[A$_{SORT}$ 3 A$_{BUBBLE}$]	3	SWAP
20	[2 4 7 3]	[A$_{SORT}$ 3 A$_{BUBBLE}$ 2]	3	>R
21	[2 4 7 2]	[A$_{SORT}$ 3 A$_{BUBBLE}$ 2]	3	1-
22	[2 4 7 2]	[A$_{SORT}$ 3 A$_{BUBBLE}$ 2]	0	...BUBBLE

When we reach the loop limit we drop the length of the sequence and exit SORT using the ; word, which takes the return address from R. At the final point, the stack should contain the ordered sequence [7 4 2 2].

---

[6]Note that Forth uses Reverse Polish Notation and that the top of the data stack is 4 in this example.

C. Learning and Run Time Efficiency

C.1. Accuracy per training examples

Sorter When measuring the performance of the model as the number of training *instances* varies, we can observe the benefit of additional prior knowledge to the optimisation process. We find that when stronger prior knowledge is provided (COMPARE), the model quickly maximises the training accuracy. Providing less structure (PERMUTE) results in lower testing accuracy initially, however, both sketches learn the correct behaviour and generalise equally well after seeing 256 training instances. Additionally, it is worth noting that the PERMUTE sketch was not always able to converge into a result of the correct length, and both sketches are not trivial to train.

In comparison, Seq2Seq baseline is able to generalise only to the sequence it was trained on (Seq2Seq trained and tested on sequence length 3). When training it on sequence length 3, and testing it on a much longer sequence length of 8, Seq2Seq baseline is not able to achieve more than 45% accuracy.
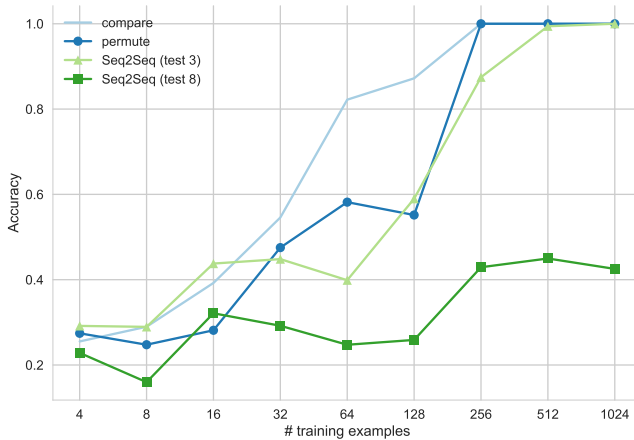

Figure 3: Accuracy of models for varying number of training examples, trained on input sequence of length 3 for the Bubble sort task. Compare, permute, and Seq2Seq (test 8) were tested on sequence lengths 8, and Seq2Seq (test 3) was tested on sequence length 3.

Adder We tested the models to train on datasets of increasing size on the addition task. The results, depicted in Table 4 show that both the choose and the manipulate sketch are able to perfectly generalise from 256 examples, trained on sequence lengths of 8, tested on 16. In comparison, the Seq2Seq baseline achieves 98% when trained on 16384 examples, but only when tested on the input of the same length, 8. If we test Seq2Seq as we tested the sketches, it is unable to achieve more 19.7%.
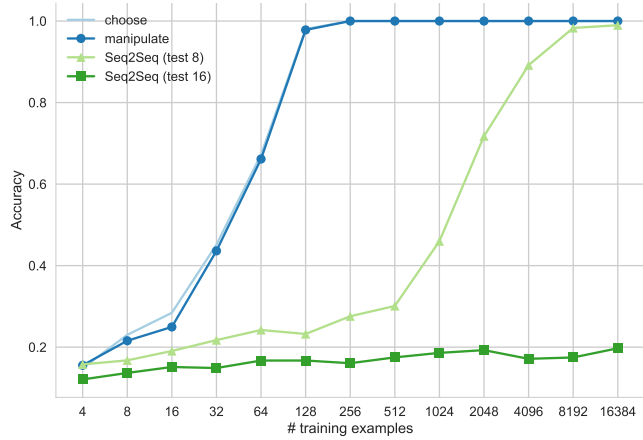

Figure 4: Accuracy of models for varying number of training examples, trained on input sequence of length 8 for the addition task. Manipulate, choose, and Seq2Seq (test 16) were tested on sequence lengths 16, and Seq2Seq (test 8) was tested on sequence length 8.

C.2. Program Code Optimisations

We measure the runtime of Bubble sort on sequences of varying length with and without the optimisations described in Section 3.4. The results of ten repeated runs are shown in Figure 5 and demonstrate large relative improvements for symbolic execution and interpolation of if-branches compared to non-optimised $\partial 4$ code.
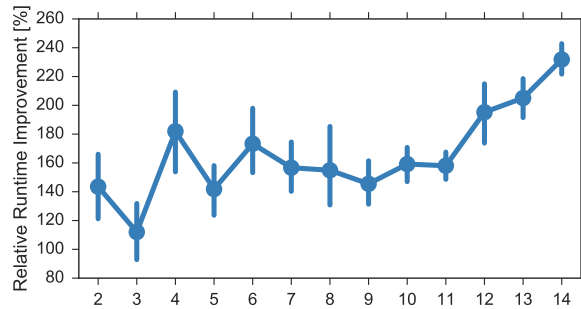

Figure 5: Relative speed improvements of program code optimisations for different input sequence lengths (*bottom*).

D. $\partial 4$ execution of a Bubble sort sketch

Listing 1 (lines 3b and 4b – in blue) defines the BUBBLE word as a sketch capturing several types of prior knowledge. In this section, we describe the PERMUTE sketch. In it, we assume BUBBLE involves a recursive call, that terminates at length 1, and that the next BUBBLE call takes as input some function of the current length and the top two stack elements.

The input to this sketch are the sequence to be sorted and its length decremented by one, $n-1$ (line 1). These inputs

are expected on the data stack. After the length $(n − 1)$ is duplicated for further use with DUP, the machine tests whether it is non-zero (using IF, which consumes the TOS during the check). If $n−1 > 0$, it is stored on the R stack for future use (line 2).

At this point (line 3b) the programmer only knows that a decision must be made based on the top two data stack elements D0 and D-1 (comparison elements), and the top return stack, R0 (length decremented by 1). Here the precise nature of this decision is unknown but is limited to variants of permutation of these elements, the output of which produce the input state to the decrement -1 and the recursive BUBBLE call (line 4b). At the culmination of the call, R0, the output of the learned slot behaviour, is moved onto the data stack using R>, and execution proceeds to the next step.

Figure 2 illustrates how portions of this sketch are executed on the $\partial 4$ RNN. The program counter initially resides at >R (line 3 in **P**), as indicated by the vector **c**, next to program **P**. Both data and return stacks are partially filled ($\mathcal{R}$ has 1 element, $\mathcal{D}$ has 4), and we show the content both through horizontal one-hot vectors and their corresponding integer values (colour coded). The vectors **d** and **r** point to the top of both stacks, and are in a one-hot state as well. In this execution trace, the slot at line 4 is already showing optimal behaviour: it remembers the element on the return stack (4) is larger and executes BUBBLE on the remaining sequence with the counter n subtracted by one, to 1.

E. Experimental details

The parameters of each sketch are trained using Adam (Kingma & Ba, 2015), with gradient clipping (set to 1.0) and gradient noise (Neelakantan et al., 2015b). We tuned the learning rate, batch size, and the parameters of the gradient noise in a random search on a development variant of each task.

E.1. Seq2Seq baseline

The Seq2Seq baseline models are single-layer networks with LSTM cells of 50 dimensions.

The training procedure for these models consists of 500 epochs of Adam optimisation, with a batch size of 128, a learning rate of 0.01, and gradient clipping when the L2 norm of the model parameters exceeded 5.0. We vary the size of training and test data (Fig. 3), but observe no indication of the models failing to reach convergence under these training conditions.

E.2. Sorting

The Permute and Compare sketches in Table 1 were trained on a randomly generated train, development and test set containing 256, 32 and 32 instances, respectively. Note that the low number of dev and test instances was due to the computational complexity of the sketch.

The batch size was set to a value between 64 and 16, depending on the problem size, and we used an initial learning rate of 1.0.

E.3. Addition

We trained the addition Choose and Manipulate sketches presented in Table 2 on a randomly generated train, development and test sets of sizes 512, 256, and 1024 respectively. The batch size was set to 16, and we used an initial learning rate of 0.05

E.4. Word Algebra Problem

The Common Core (CC) dataset (Roy & Roth, 2015) is partitioned into a train, dev, and test set containing 300, 100, and 200 questions, respectively. The batch size was set to 50, and we used an initial learning rate of 0.02. The BiLSTM word vectors were initialised randomly to vectors of length 75. The stack width was set to 150 and the stack size to 5.

# F. Qualitative Analysis on BubbleSort of PC traces

In Figure 6 we visualise the program counter traces. The trace follows a single example from start, to middle, and the end of the training process. In the beginning of training, the program counter starts to deviate from the one-hot representation in the first 20 steps (not observed in the figure due to unobservable changes), and after two iterations of SORT, $\partial 4$ fails to correctly determine the next word. After a few training epochs $\partial 4$ learns better permutations which enable the algorithm to take crisp decisions and halt in the correct state.
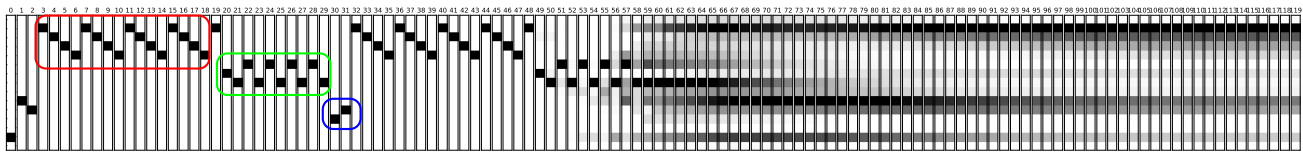
(a) Program Counter trace in early stages of training.
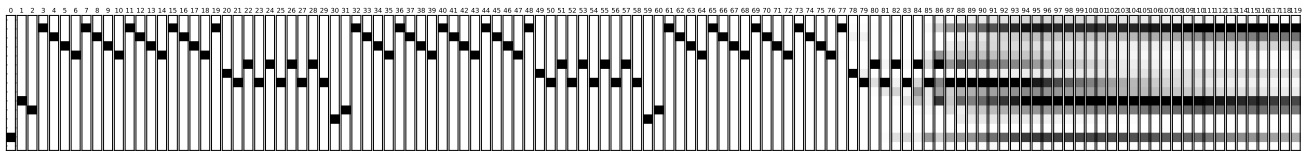
(b) Program Counter trace in the middle of training.
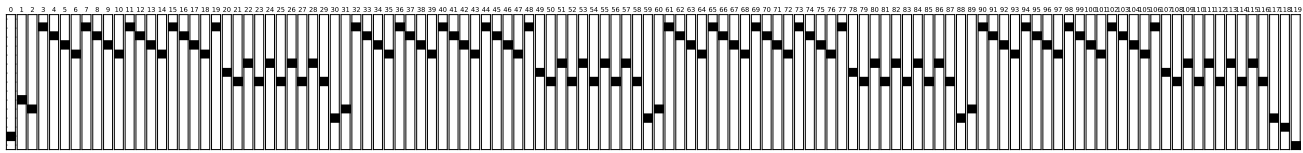
(c) Program Counter trace at the end of training.

Figure 6: Program Counter traces for a single example at different stages of training BubbleSort in Listing 1 (red: successive recursion calls to BUBBLE, green: successive returns from the recursion, and blue: calls to SORT). The last element in the last row is the halting command, which only gets executed after learning the correct slot behaviour.

G. The complete Word Algebra Problem sketch

The Word Algebra Problem (WAP) sketch described in Listing 3 is the core of the model that we use for WAP problems. However, there were additional words before and after the core which took care of copying the data from the heap to data and return stacks, and finally emptying out the return stack.

The full WAP sketch is given in Listing 4. We define a QUESTION variable which will denote the address of the question vector on the heap. Lines 4 and 5 create REPR_BUFFER and NUM_BUFFER variables and denote that they will occupy four sequential memory slots on the heap, where we will store the representation vectors and numbers, respectively. Lines 7 and 8 create variables REPR and NUM which will denote addresses to current representations and numbers on the heap. Lines 10 and 11 store REPR_BUFFER to REPR and NUM_BUFFER to NUM, essentially setting the values of variables REPR and NUM to starting addresses allotted in lines 4 and 5. Lines 14-16 and 19-20 create macro functions STEP_NUM and STEP_REPR which increment the NUM and REPR values on call. These macro functions will be used to iterate through the heap space. Lines 24-25 define macro functions CURRENT_NUM for fetching the current number, and CURRENT_REPR for fetching representation values. Lines 28-32 essentially copy values of numbers from the heap to the data stack by using the CURRENT_NUM and STEP_NUM macros. After that line 35 pushes the question vector, and lines 36-40 push the word representations of numbers on the return stack.

Following that, we define the core operations of the sketch. Line 43 permutes the elements on the data stack (numbers) as a function of the elements on the return stack (vector representations of the question and numbers). Line 45 chooses an operator to execute over the TOS and NOS elements of the data stack (again, conditioned on elements on the return stack). Line 47 executes a possible swap of the two elements on the data stack (the intermediate result and the last operand) conditioned on the return stack. Finally, line 49 chooses the last operator to execute on the data stack, conditioned on the return stack.

The sketch ends with lines 52-55 which empty out the return stack.

```
 1 \ address of the question on H
 2 VARIABLE QUESTION
 3 \ allotting H for representations and numbers
 4 CREATE REPR_BUFFER 4 ALLOT
 5 CREATE NUM_BUFFER 4 ALLOT
 6 \ addresses of the first representation and number
 7 VARIABLE REPR
 8 VARIABLE NUM

10 REPR_BUFFER REPR !
11 NUM_BUFFER NUM !

13 \ macro function for incrementing the pointer to numbers in H
14 MACRO: STEP_NUM
15    NUM @ 1+ NUM !
16 ;

18 \ macro function for incrementing the pointer to representations in H
19 MACRO: STEP_REPR
20    REPR @ 1+ REPR !
21 ;

23 \ macro functions for fetching current numbers and representations
24 MACRO: CURRENT_NUM NUM @ @ ;
25 MACRO: CURRENT_REPR REPR @ @ ;

27 \ copy numbers to D
28 CURRENT_NUM
29 STEP_NUM
30 CURRENT_NUM
31 STEP_NUM
32 CURRENT_NUM

34 \ copy question vector, and representations of numbers to R
35 QUESTION @ >R
36 CURRENT_REPR >R
37 STEP_REPR
38 CURRENT_REPR >R
39 STEP_REPR
40 CURRENT_REPR >R

42 \ permute stack elements, based on the question and number representations
43 { observe R0 R-1 R-2 R-3 -> permute D0 D-1 D-2 }
44 \ choose the first operation
45 { observe R0 R-1 R-2 R-3 -> choose + - * / }
46 \ choose whether to swap intermediate result and the bottom number
47 { observe R0 R-1 R-2 R-3 -> choose SWAP NOP }
48 \ choose the second operation
49 { observe R0 R-1 R-2 R-3 -> choose + - * / }

51 \ empty out R
52 R> DROP
53 R> DROP
54 R> DROP
55 R> DROP
```

Listing 4: The complete Word Algebra Problem sketch