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..`$_1$`ELSE..`$_2$`THEN`	$p = (pop_{\mathbf{D}}() = \mathbf{0})$ $p * ..{}_1 + (1-p) * ..{}_2$
`BEGIN..`$_1$`WHILE..`$_2$`REPEAT`	$..{}_1\ jump(c, ..{}_2)$
`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 (white lines and *a* lines, colored green). We provide a description of how the first iteration of this algorithm is executed by the Forth abstract machine:

The program begins at line 12, 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	*line*	comment
1	[]	[]	12	execution start
2	[2 4 2 7 4]	[]	12	pushing sequence [2 4 2 7 4] to D
3	[2 4 2 7 4]	[A_{SORT}]	9	`SORT` (puts return address A_{SORT} to R)

For a sequence of length 4, `SORT` performs a do-loop in line 10 that calls the `BUBBLE` subroutine. 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.

4	[2 4 2 7 3]	[A_{SORT}]	10	`1-`
5	[2 4 2 7 3 3]	[A_{SORT}]	10	`DUP`
6	[2 4 2 7 3 3 0]	[A_{SORT}]	10	`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@`

7	[2 4 2 7 3]	[A_{SORT}]	10	`DO`
8	[2 4 2 7]	[A_{SORT} 3]	10	`>R`
9	[2 4 2 7 3]	[A_{SORT} 3]	10	`R@`

Next, we call `BUBBLE` to perform one iteration of the bubble pass (`BUBBLE` will be called 3 times in total), and consuming 3. Note that this call puts the current program counter onto R as the `BUBBLE` return address A_{BUBBLE}, to be used for the program counter c when exiting the `BUBBLE` subroutine.

Inside the `BUBBLE` subroutine, `DUP` duplicates 3 on R. `IF` consumes the duplicated 3 and interprets it as TRUE (non-zero value). `>R` puts 3 on R.

10	[2 4 2 7 3]	[A_{SORT} 3 A_{BUBBLE}]	1	`BUBBLE` (puts return address A_{BUBBLE} to R)
11	[2 4 2 7 3 3]	[A_{SORT} 3 A_{BUBBLE}]	2	`DUP`
12	[2 4 2 7 3]	[A_{SORT} 3 A_{BUBBLE}]	2	`IF`
13	[2 4 2 7]	[A_{SORT} 3 A_{BUBBLE} 3]	2	`>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 (non-zero value), which it is, so it executes `SWAP`.

14	[2 4 2 7 2 7]	[A_{SORT} 3 A_{BUBBLE} 3]	3	`OVER OVER`
15	[2 4 2 7 1]	[A_{SORT} 3 A_{BUBBLE} 3]	3	`<`
16	[2 4 2 7]	[A_{SORT} 3 A_{BUBBLE} 3]	3	`IF`
17	[2 4 7 2]	[A_{SORT} 3 A_{BUBBLE} 3]	3	`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. Note that R will accumulate the analysed part of the sequence, which will be recursively taken back by the final `R>` in line 4.

18	[2 4 7 2 3]	[A_{SORT} 3 A_{BUBBLE}]	4	`R>`
19	[2 4 7 3 2]	[A_{SORT} 3 A_{BUBBLE}]	4	`SWAP`
20	[2 4 7 3]	[A_{SORT} 3 A_{BUBBLE} 2]	4	`>R`
21	[2 4 7 2]	[A_{SORT} 3 A_{BUBBLE} 2]	4	`1-`
22	[2 4 7 2]	[A_{SORT} 3 A_{BUBBLE} 2]	1	`BUBBLE` ...

When we reach the loop limit we `DROP` (line 10) the length of the sequence and exit `SORT` with the `;` word (line 11) which takes the return address from R. At the end, the stack will 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.

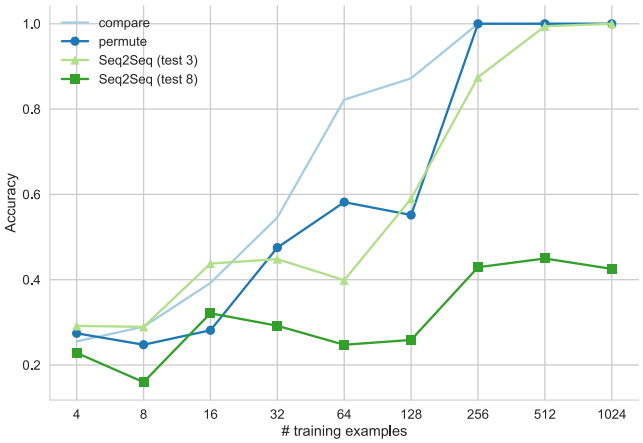


Figure 3: Accuracy of models for varying number of training examples, trained on input sequence of length 3 for the Bubble sort task. Compare, permute, and Seq2Seq (test 8) were tested on sequence lengths 8, and Seq2Seq (test 3) was tested on sequence length 3.

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%.

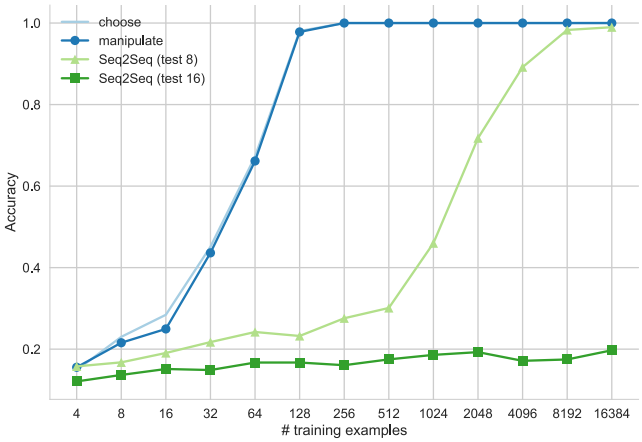


Figure 4: Accuracy of models for varying number of training examples, trained on input sequence of length 8 for the addition task. Manipulate, choose, and Seq2Seq (test 16) were tested on sequence lengths 16, and Seq2Seq (test 8) was tested on sequence length 8.

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.

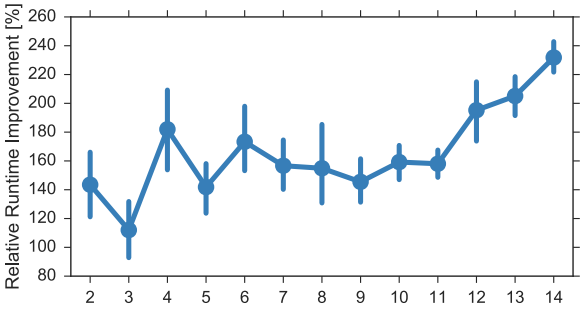


Figure 5: Relative speed improvements of program code optimisations for different input sequence lengths (*bottom*).

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 $\mathbf{P}$), as indicated by the vector $\mathbf{c}$, next to program $\mathbf{P}$. 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 $\mathbf{d}$ and $\mathbf{r}$ 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

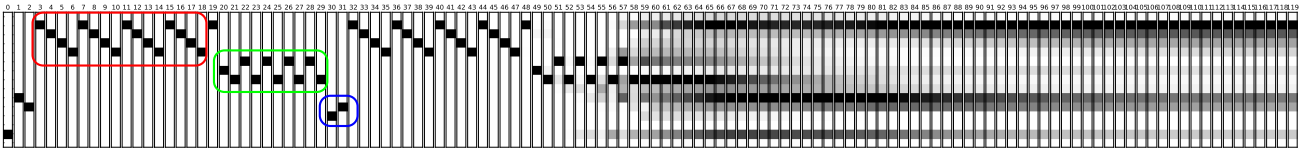
We trained the addition Choose and Manipulate sketches presented in Table 2 on a randomly generated train, development and test sets of sizes 512, 256, and 1024 respectively. The batch size was set to 16, and we used an initial learning rate of 0.05

E.4. Word Algebra Problem

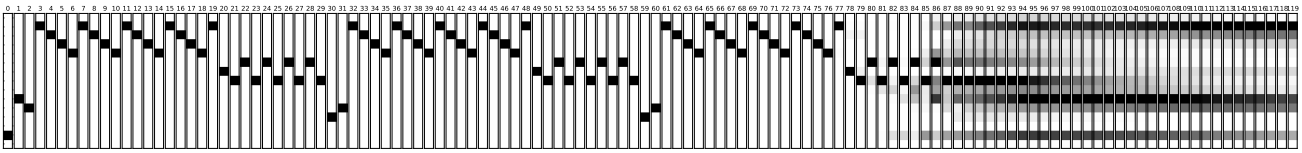
The Common Core (CC) dataset (Roy & Roth, 2015) is partitioned into a train, dev, and test set containing 300, 100, and 200 questions, respectively. The batch size was set to 50, and we used an initial learning rate of 0.02. The BiLSTM word vectors were initialised randomly to vectors of length 75. The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

F. Qualitative Analysis on BubbleSort of PC traces

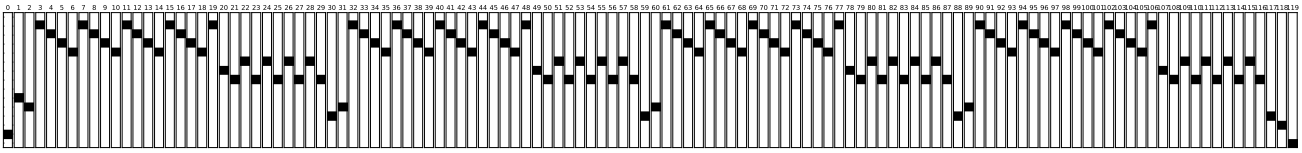
In Figure 6 we visualise the program counter traces. The trace follows a single example from start, to middle, and the end of the training process. In the beginning of training, the program counter starts to deviate from the one-hot representation in the first 20 steps (not observed in the figure due 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.

```
\ address of the question on H
VARIABLE QUESTION
\ allotting H for representations and numbers
CREATE REPR_BUFFER 4 ALLOT
CREATE NUM_BUFFER 4 ALLOT
\ addresses of the first representation and number
VARIABLE REPR
VARIABLE NUM

REPR_BUFFER REPR !
NUM_BUFFER NUM !

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

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

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

\ copy numbers to D
CURRENT_NUM
STEP_NUM
CURRENT_NUM
STEP_NUM
CURRENT_NUM

\ copy question vector, and representations of numbers to R
QUESTION @ >R
CURRENT_REPR >R
STEP_REPR
CURRENT_REPR >R
STEP_REPR
CURRENT_REPR >R

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

\ empty out R
R> DROP
R> DROP
R> DROP
R> DROP
```

Listing 4: The complete Word Algebra Problem sketch