A Prompt completion algorithm

Algorithm 2 describes the prompt completion algorithm introduced in Section 2.2. It implicitly considers a single action, which takes the next sequence element.

Algorithm 2 – Prompt completion

Input: Grounded schema $\{T, \mathcal{C}, E^{\text{rb}}\}$ with rebound CSCG emission matrix E^{rb}, delimiter token $x_{\emptyset}$, prompt $x^{(\text{prompt})} = (x_1, \ldots, x_m)$

Output: A completed prompt $x^{(\text{prompt completed})} = (x_1, \ldots, x_m, x_{m+1}, \ldots, x_{m+p} = x_{\emptyset})$

1: Run max-product for MAP inference and return $z^{\text{MAP}} = (z_1, \ldots, z_m) = \text{argmax}_z \, p(z|x^{(\text{prompt})})$.

2: Set $\ell = 0$. While $x_{m+\ell} \neq x_{\emptyset}$, increment $\ell \leftarrow \ell + 1$ and sample the next most likely observation: $z_{m+\ell} \in \text{argmax}_j \, T_{z_{m+\ell-1},\, j}$ and $x_{m+\ell} \in \text{argmax}_j \, E^{\text{rb}}_{z_{m+\ell},\, j}$.

B Rapid binding in CSCGs

Algorithm 3 is a variant of the rebinding Algorithm 1 that does not use EM. Instead, it first searches for "surprising observations": a surprise has a low probability of being emitted by its decoded clone. This decoded clone (and all the clones in its clone set) are then *rapidly bound* to emit the surprise.

Algorithm 3 – Rapid binding algorithm

Input: Grounded schema $\{T, \mathcal{C}, E^0\}$, pseudocount ϵ, surprise probability p_{surprise}, prompt $x^{(\text{prompt})}$

Output: Rapidly bound emission matrix E^{rb}

1: Add the pseudocount ϵ to the initial emission matrix and normalize its rows.

2: Run max-product for MAP inference and return $z^{\text{MAP}} = \text{argmax}_z \, p(z|x^{(\text{prompt})})$.

3: Define the set of surprising observations $\mathcal{S} = \left\{ x_n : \; E^0_{z_n^{\text{MAP}},\, x_n} \leq p_{\text{surprise}} \right\}$.

4: Set $E^{\text{rb}} = E^0$. For each surprising observation x_n, (rapidly) bind z_n^{MAP} and all the clones in the clone slot of z_n^{MAP} to emit x_n.
That is, $\forall \, \tilde{z} : \mathcal{C}(\tilde{z}) = \mathcal{C}(z_n^{\text{MAP}})$ set $E^{\text{rb}}_{\tilde{z},\, x_n} = 1$ and $E^{\text{rb}}_{\tilde{z},\, j} = 0, \; \forall j \neq x_n$.

Initially, for any given clone i, only one entry j of the row E^0_i in the emission matrix is set to 1. After Step 1, we have $E^0_{i,\, j} = \frac{1}{1+N_{\text{obs}}\epsilon}$ and $E^0_{i,\, k} = \frac{\epsilon}{1+N_{\text{obs}}\epsilon}$, $\forall k \neq j$. We can then use $p_{\text{surprise}} = \frac{1}{2(1+N_{\text{obs}}\epsilon)}$.

C Additional materials for the GINC dataset

First, we present two additional plots for the GINC experiment.

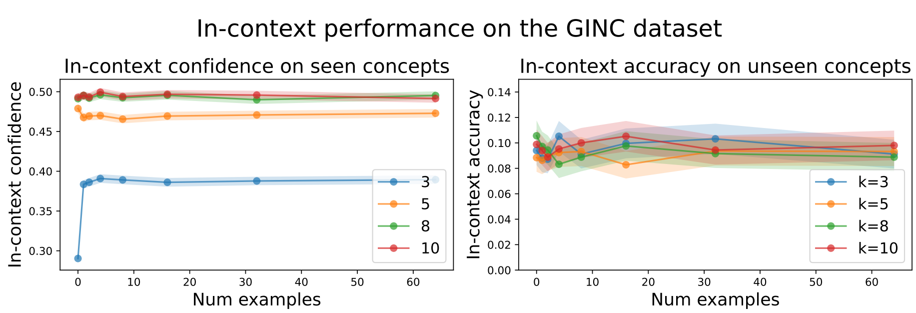

Figure 8: [Left] In-context confidence for the CSCG with 50 clones on the GINC test dataset, defined as the averaged probability of the predictions. For larger values of k, CSCG correctly infers the context of the aliased observations and is more confident in his predictions. [Right] Similar to the transformer and LSTM reported in [50], CSCG fails to extrapolate and has a low in-context accuracy when the test prompts are sampled from five novel concepts, unseen during training.

Second, we present the table of results associated with Fig. 3 for the CSCGs with 10 and 50 clones.

Context length	No. of examples	CSCG with 10 clones	CSCG with 50 clones
3	0	0.509 (0.020)	0.534 (0.020)
	1	0.351 (0.019)	0.445 (0.019)
	2	0.366 (0.019)	0.453 (0.020)
	4	0.356 (0.019)	0.468 (0.020)
	8	0.360 (0.019)	0.454 (0.020)
	16	0.354 (0.019)	0.460 (0.020)
	32	0.354 (0.019)	0.441 (0.0219)
	64	0.369 (0.019)	0.468 (0.020)
5	0	0.682 (0.018)	0.927 (0.010)
	1	0.640 (0.019)	0.927 (0.012)
	2	0.629 (0.019)	0.904 (0.012)
	4	0.654 (0.019)	0.883 (0.013)
	8	0.627 (0.019)	0.894 (0.012)
	16	0.637 (0.019)	0.902 (0.012)
	32	0.634 (0.019)	0.901 (0.012)
	64	0.637 (0.019)	0.899 (0.012)
8	0	0.696 (0.018)	0.969 (0.007)
	1	0.694 (0.018)	0.972 (0.007)
	2	0.686 (0.018)	0.972 (0.006)
	4	0.681 (0.018)	0.978 (0.006)
	8	0.690 (0.018)	0.973 (0.006)
	16	0.686 (0.018)	0.975 (0.006)
	32	0.676 (0.018)	0.968 (0.006)
	64	0.694 (0.018)	0.975 (0.007)
10	0	0.684 (0.018)	0.975 (0.006)
	1	0.705 (0.018)	0.977 (0.006)
	2	0.674 (0.018)	0.971 (0.006)
	4	0.713 (0.018)	0.974 (0.006)
	8	0.690 (0.018)	0.977 (0.006)
	16	0.689 (0.018)	0.977 (0.006)
	32	0.712 (0.018)	0.978 (0.006)
	64	0.690 (0.018)	0.978 (0.006)

Table 1: In-context accuracy for a CSCG with 10 clones and a CSCG 50 clones trained on the GINC dataset, averaged (with 95% confidence intervals) on each each pair (k, n) of context length and number of examples n of the GINC test set.

CSCG performs better on zero-shot prompts than on few-shot prompts: We observe that, for short contexts, CSCG in-context accuracy is higher on zero-shot prompts $n = 0$ than on few-shot prompts $n = 1, 2, \ldots$. We hypothesize that the difference between the training and the prompt distributions creates a gap that lowers few-shot in-context accuracy. The performance gap disappears for larger contexts $k \in \{8, 10\}$ as they "overpower" the train-test distribution divergence. [50] made a similar observation for transformers. However, their performance gap was also observable for larger contexts.

D Additional materials for the LIALT dataset

D.1 Natural language instructions

Tables 2 and 3 present the natural language instructions respectively used for the nine list algorithms and four matrix algorithms of the LIALT dataset. Language instructions are grouped in clusters of five: all five instructions within one cluster describe to the same algorithm. As described in the main text, each demonstration of the LIALT training and first test set uniformly selects one instruction.

“find the element at index zero of the list” “print the first element from the list” “return the leading element from the list” “find the head element from the list” ‘retrieve the starting element from the list”	“print the element at index one of the list” “find the second element from the list” “retrieve the second element from the list” “locate the second item from the list” “return the element in second place from the list”
“print the element at index two of the list” “find the third element from the list” “locate the third element from the list” “output the third item from the list” “return the element in third place from the list”	“reverse the list” “mirror the list” “flip the list” “flip the order of the list” “reverse the order of the items in the list”
“duplicate each list item” “replicate every element in the list” “make a copy of each element in the list” “clone each element in the list” “create a second instance of every element in the list”	“rotate the list elements one place forward” “roll the list elements one position to the right” “switch the items of the list one position forward” “advance the list elements one index forward” “move the list elements one position forward”
“rotate the list elements one place backward” “move the list elements one position to the left” “change the items of the list one position backward” “displace the elements of the list one index backward” “roll the list items one position backward”	“print every other member in the list starting with the first member” “find alternate elements in the list beginning with the first element” “print every second item in the list, starting with the first element” “output every second element in the list, starting from the first element” “output odd indexed elements”
“print every other member in the list starting with the second member” “retrieve alternate items in the list starting with the second item” “output odd indexed elements” “retrieve every other entry in the list starting with the second entry” “return every other object in the list starting with the second object”	

Table 2: Natural language instructions for the list algorithms used in the LIALT dataset

“return the matrix diagonal” “collect the diagonal values of the matrix” “retrieve the diagonal elements of the matrix” “return the diagonal entries of the matrix” “fetch the diagonal items of the matrix”	“return the matrix transpose” “retrieve the transpose of the matrix” “get the transposed matrix” “compute the transposed form of the matrix” “derive the transpose matrix”
“roll the columns of the matrix to the right” “rotate the matrix columns to the right” “move the matrix columns to the right” “shift the columns of the matrix to the right” “spin the matrix columns to the right”	“find the matrix element in the second row and second column” ‘find the value in the second row and second column of the matrix” “fetch the matrix element located in row 2 and column 2” ‘print the value at 2 2 in the matrix” “retrieve the matrix element at 2 2”

Table 3: Natural language instructions for the matrix algorithms used in the LIALT dataset

D.2 Learned CSCG model

Our next Figure 9 displays the transition graph of the CSCG trained on the LIALT dataset, displayed by both (a) stacking clones (b) unrolling them using the Kamada-Kawai algorithm [20].

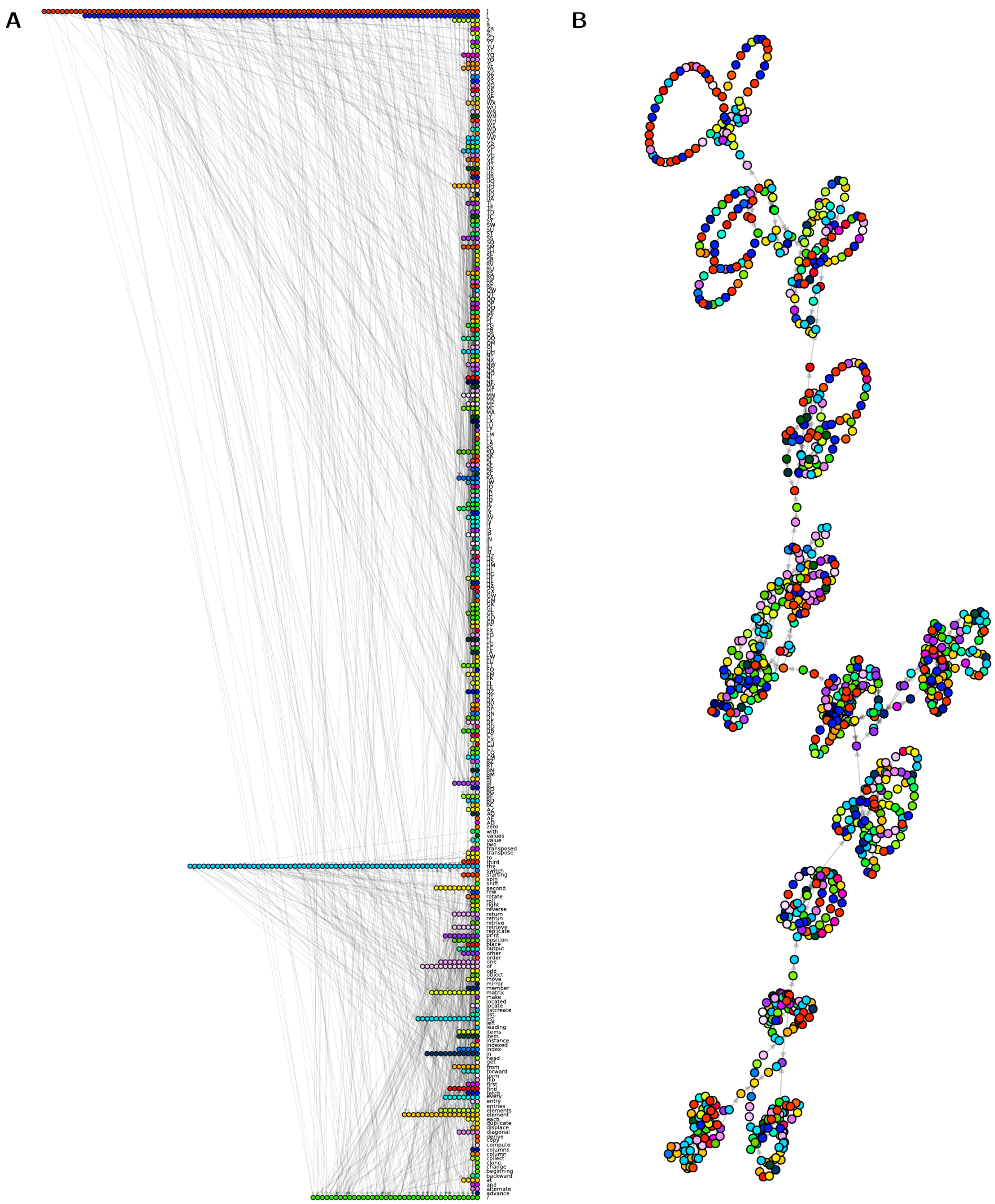

Figure 9: **A.** Transition graph of the learned CSCG model with overallocation ratio 3, visualized with stacked clones. **B.** The same transition graph visualized using the Kamada-Kawai algorithm [20] reveals 13 loosely connected clusters corresponding to the 13 algorithms used in the LIALT dataset.

D.3 Results on the LIALT dataset

We present below the tables of results associated with Fig. 5. Our first Table 4 contains the in-context accuracies averaged on the entire test set.

Overallocation ratio	Instruction-based prompts	Example-based prompts
0.1	0.02 (0.03)	0.01 (0.02)
0.3	0.10 (0.06)	0.15 (0.07)
1.0	0.52 (0.10)	0.49 (0.10)
3.0	0.88 (0.06)	0.93 (0.05)

Table 4: Average in-context accuracy of each CSCG model—with 95% confidence intervals—as a function of CSCG overallocation on both (a) the instruction-based LILAT test set and (b) the example-based LIALT test set.

Our second Table 5 contains the in-context accuracies per tasks, on instructions-based prompts.

	Overallocation ratio			
Task	0.1	0.3	1.0	3.0
`list 1st elem.`	0.00 (0.00)	0.00 (0.00)	0.86 (0.13)	1.00 (0.00)
`list 2nd elem.`	0.33 (0.19)	0.50 (0.20)	0.50 (0.20)	1.00 (0.00)
`list 3rd elem.`	0.00 (0.00)	0.50 (0.20)	0.33 (0.19)	1.00 (0.00)
`list reverse`	0.00 (0.00)	0.00 (0.00)	0.50 (0.18)	0.62 (0.17)
`list repeat twice`	0.00 (0.00)	0.00 (0.00)	1.00 (0.00)	1.00 (0.00)
`list alt. even`	0.00 (0.00)	0.00 (0.00)	0.36 (0.15)	0.91 (0.09)
`list alt. odd`	0.00 (0.00)	0.00 (0.00)	0.83 (0.15)	1.00 (0.00)
`list circ. shift fw.`	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.60 (0.22)
`list circ. shift bw.`	0.00 (0.00)	0.00 (0.00)	0.38 (0.17)	1.00 (0.00)
`matrix diagonal`	0.00 (0.00)	0.00 (0.00)	0.67 (0.14)	1.00 (0.00)
`matrix transpose`	0.00 (0.00)	0.00 (0.00)	0.86 (0.13)	1.00 (0.00)
`matrix roll columns`	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.54 (0.14)
`matrix elem. at idx.`	0.00 (0.00)	0.50 (0.18)	1.00 (0.00)	1.00 (0.00)

Table 5: Average in-context accuracy by task—with standard errors—as a function of CSCG overallocation on instruction-based prompts.

Finally, our last Table 6 contains the in-context accuracies per tasks, on example-based prompts.

	Overallocation ratio			
Task	0.1	0.3	1.0	3.0
`list 1st elem.`	0.00 (0.00)	0.29 (0.17)	0.57 (0.19)	1.00 (0.00)
`list 2nd elem.`	0.00 (0.00)	0.00 (0.00)	0.17 (0.15)	0.83 (0.15)
`list 3rd elem.`	0.00 (0.00)	0.00 (0.00)	0.33 (0.19)	0.83 (0.15)
`list reverse`	0.00 (0.00)	0.25 (0.15)	0.62 (0.17)	0.75 (0.15)
`list repeat twice`	0.00 (0.00)	0.00 (0.00)	0.67 (0.27)	1.00 (0.00)
`list alt. even`	0.00 (0.00)	0.09 (0.09)	0.45 (0.15)	0.91 (0.09)
`list alt. odd`	0.00 (0.00)	0.17 (0.15)	0.17 (0.15)	1.00 (0.00)
`list circ. shift fw.`	0.00 (0.00)	0.20 (0.18)	0.40 (0.22)	1.00 (0.00)
`list circ. shift bw.`	0.00 (0.00)	0.00 (0.00)	0.38 (0.17)	0.88 (0.12)
`matrix diagonal`	0.00 (0.00)	0.08 (0.08)	0.42 (0.14)	1.00 (0.00)
`matrix transpose`	0.00 (0.00)	0.14 (0.13)	0.57 (0.19)	1.00 (0.00)
`matrix roll columns`	0.08 (0.07)	0.31 (0.13)	0.62 (0.13)	0.92 (0.07)
`matrix elem. at idx.`	0.00 (0.00)	0.25 (0.15)	0.88 (0.12)	1.00 (0.00)

Table 6: Average in-context accuracy by task—with standard errors—as a function of CSCG overallocation on example-based prompts.

D.4 Example failures

Finally, we present a few examples which illustrate the failure modes of our approach. These are primarily a consequence of imperfections in the learned CSCG model.

Each example is presented in the format (prompt, ground truth correct output, actual model response).

1. For these failures, the instruction circuit has been wired to the wrong algorithm circuit (possibly driven by the ambiguity of the forward slash delimiter separating the instruction from the example), resulting in the retrieval of the wrong schema.
   - `output odd indexed elements / [ U V B Q K I ]`
     `[ U B K ] /`
     `[ V Q I ] /`
   - `flip the list / [ S E J ]`
     `[ J E S ] /`
     `[ S S E E J J ]`
   - `reverse the list / [ R T B ]`
     `[ B T R ] /`
     `[ R R T T B B ] /`
   - `mirror the list / [ B A O T ]`
     `[ T O A B ] /`
     `[ B B A A O O T T ] /`
2. For these failures, the schema has been learned incorrectly.
   - `switch the items of the list one position forward / [ L N G X M T ]`
     `[ T L N G X M ] /`
     `[ T L N G X M T ] [ T L N G X M T ] ...`
   - `shift the columns of the matrix to the right / [ [ D Y ] [ V F ] ]`
     `[ [ Y D ] [ F V ] ] /`
     `[ [ get`
   - `/ [ Z J B ] [ Z Z J J B B ] / [ B A E F W L ]`
     `[ B B A A E E F F W W L L ] /`
     `[ B B A E F F W W L L ] /`
   - `/ [ V P X T ] [ P T ] / [ V F J P E W ]`
     `[ F P W ] /`
     `[ F P W ] [ F P W ] ...`