Appendix

A. StarCraft environment details

A.1. Instruction grammar

$\langle$*task*$\rangle$ ::= { ‘,’ $\langle$*line-type*$\rangle$ }

$\langle$*line-type*$\rangle$ ::= $\langle$*building*$\rangle$ | $\langle$*unit*$\rangle$

A.2. Generation of instructions

We initially generate the build-tree as a random directed acyclic graph. Next we randomize the building production capabilities by assigning a random building to each unit (that building being the one capable of producing that unit). We generate instructions, unit by unit, randomly selecting each unit from those whose instructions are sufficiently short. For example, if the cap on instruction length is 5, we would exclude any unit that is produced by a building with a prerequisite chain exceeding a depth of 5. If the resulting instruction had length 3, we would repeat this process again but for instructions of length 2. We iterate these steps until the instruction reaches the desired length or no valid instructions are possible.

A.3. Spawning of world objects

Each episode begins with a Nexus building (per the original game) and three Probe workers. We choose the number of other initial buildings at random between 0 and 36 (the number of grids in our 6×6 environment. We choose the starting location of all buildings uniformly at random (although no two buildings are permitted to occupy the same grid). The three Probe workers spawn at the Nexus (per the original game).

B. Minecraft environment details

B.1. Instruction grammar

$\langle$*task*$\rangle$ ::= { ‘,’ $\langle$*line-type*$\rangle$ }

$\langle$*line-type*$\rangle$ ::= $\langle$*subtask*$\rangle$ | $\langle$*while-expression*$\rangle$ | $\langle$*if-expression*$\rangle$

$\langle$*while-expression*$\rangle$ ::= while $\langle$*object*$\rangle$ do $\langle$*subtask-list*$\rangle$

$\langle$*if-expression*$\rangle$ ::= $\langle$*if-block*$\rangle$ $\langle$*else-block*$\rangle$ | $\langle$*if-block*$\rangle$

$\langle$*if-block*$\rangle$ ::= if $\langle$*object*$\rangle$ do $\langle$*subtask-list*$\rangle$

$\langle$*else-block*$\rangle$ ::= else do $\langle$*subtask-list*$\rangle$

$\langle$*subtask-list*$\rangle$ ::= { ‘,’ $\langle$*subtask*$\rangle$ }

$\langle$*subtask*$\rangle$ ::= $\langle$*interaction*$\rangle$ | $\langle$*resource*$\rangle$

$\langle$*interaction*$\rangle$ ::= inspect | pickup | transform

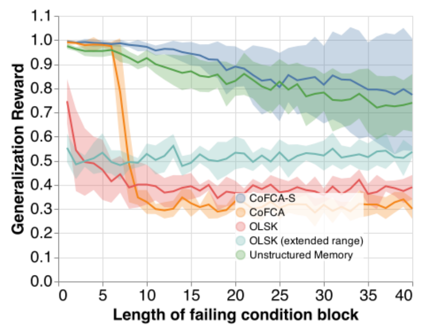
$\langle$*resource*$\rangle$ ::= iron | gold | wood

B.2. Generation of instructions

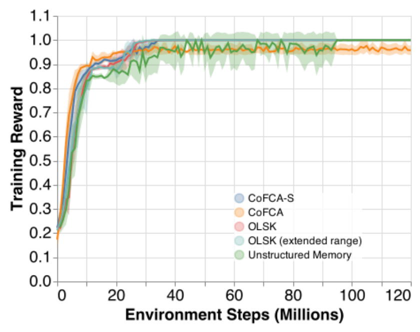
Tasks are generated randomly. Most lines in the task are sampled uniformly at random from {*If*, *While*, *Subtask*}. If the current line is inside an if-clause and the preceding line is a subtask, then {*Else*, *EndIf*} is added to the list of randomly sampled line types. Similar rules apply to while-clauses (we add *EndWhile* to the list) and else-clauses (we add *EndIf*).

B.3. Spawning of world objects

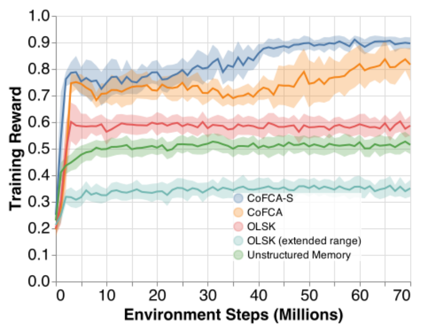
At the beginning of each episode we choose the number n of resources/merchants uniformly at random between 0 and 36 (the number of grids in our 6×6 gri-world). We then sample uniformly at random n times from {*iron*, *gold*, *wood*, *merchant*} to select the candidate population of the gridworld. We then test the feasibility of the environment for the instruction by checking that the requisite resources exist for each subtask that the agent will have to perform. If the environment is deemed infeasible, we perform the aforementioned sampling process again. After 50 resamples, if we have still failed to generate a feasible environment, we generate a new instruction per section B.2. Once we have generated a feasible population, if $n \leq 30$, we place water in a straight line through a random index and at a random horizontal/vertical orientation. Next we place the remaining n resources/merchants and the agent each in unique, random, open grids. At this point, we check that the agent has access to a wood resource (with which to build a bridge), and if not, we remove the water from the map. Finally, we place walls at any open even-indexed tiles (this to ensure that walls do not cut off parts of the gridworld).



(a) Generalization by condition block size. X-axis corresponds to instruction length; Y-axis is mean success per episode.



(b) Cumulative reward on training episodes for StarCraft environment



(c) Cumulative reward on training episodes for Minecraft environment

C. Analysis of long pointer movements in the Minecraft domain

Here we present results assessing the agents' capability to perform larger pointer movements than those learned during training. We trained the agents on instructions from the Minecraft domain with lengths sampled randomly from between 1 and 10 (the same training regimen as in §4.3.1). We evaluated the agent on a special instruction beginning with a failing condition-block (*if* or *while*) that extends to the end of the instruction, followed by a concluding subtask. Thus a successful agent will generally have to jump over the failing condition block to reach the final subtask. We varied the length of the failing condition block between 1 and 40 and noted each agent's performance at each length. These results are shown in Figure 1a. This experiment identifies one of the key failure points of the CoFCA architecture that the Scan mechanism is intended to address. The CoFCA architecture is unlikely to sample pointer movements larger than those it was trained to perform. Concretely, if the agent has never seen a control-flow block larger than n, and $\mathbf{P}$ has always placed zero mass on pointer movements greater than $\pm n$, it is unlikely that it will ever place more than zero mass on those movements, even when longer control-flow blocks require them. This explains the precipitous drop in its performance in Figure 1a as soon as the required jump exceeds the largest that it might have encountered in its training set. We also note the relatively strong performance of the unstructured memory architecture, comparable to CoFCA-S; this shows that the recurrence within the unstructured memory is able to handle longer condition blocks but is unable to deal with multiple control flows accounting for its relatively poor performance in the other generalization experiments. Finally, we note that OLSK (extended range) maintains consistently poor performance irrespective of the condition-block length because, as noted in §4.3.1, this architecture ignores the instruction, having never learned to interpret it in the first place.

D. Discussion of training performance

Figures 1b and 1c display training performance on the StarCraft and Minecraft domain. Training performance for the baselines was lower on the Minecraft domain because it requires more fine-grained control of the pointer. Results do not in any way compensate for the failure buffer discussed in §3.6 and that mechanism therefore depresses the performance of the algorithms. On the Minecraft domain, none of the baselines learned to consistently sequence substasks for longer instructions.

E. Pseudocode / schematics for baselines

This section provides pseudocode and schematics for our baselines. We indicate sections that deviate from the algorithms given in Fig. 1 with red highlighting. In these sections, we retain the variable names given in Section 3. For review:

- $\mathbf{M}$: an encoding of the instructions.
- p_t: the integer pointer into $\mathbf{M}$.
- π: the policy, implemented as a neural network.
- $\mathbf{x}_t$: the observation for the current time-step.
- $\mathbf{P}$: a collection of possible pointer movement distributions.
- ϕ: a neural network that chooses among these distributions.
- c_t: a binary value that permits or prevents movement of p_t.
- ψ: a neural network that determines the value of the gate value.

Unstructured Memory

```
1: $\mathbf{M} \leftarrow \text{bag-of-words}_\theta(\mathbf{I})$
2: initialize $\mathbf{h}_0 \in \mathbb{R}^H$
3: $\mathbf{H} \leftarrow \text{BI-GRU}(\mathbf{M})$ {running from first to last index of $\mathbf{I}$}
4: for time step $t$ in episode do
5:     $\mathbf{a}_t \sim \pi(\mathbf{x}_t, \mathbf{M}_{p_t})$
6:     $\mathbf{h}_t \leftarrow \phi(\mathbf{x}_t, \mathbf{H}, \mathbf{a}_t, \mathbf{h}_{t-1})$
7:     $c_t \sim \psi(\mathbf{x}_t, \mathbf{h}_t, \mathbf{a}_t)$
8: end for
```

OLSK

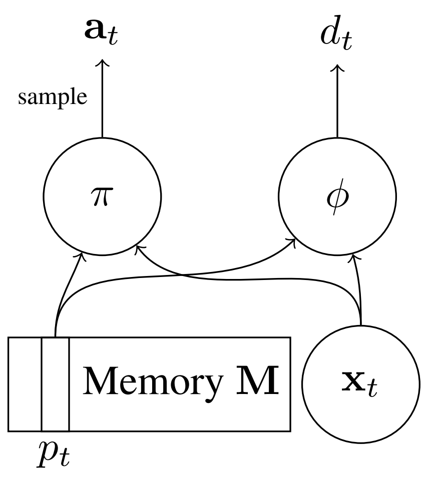
```
1: $p_0 \leftarrow 0$
2: $\mathbf{M} \leftarrow \text{bag-of-words}_\theta(\mathbf{I})$
3: initialize $\mathbf{h}_0 \in \mathbb{R}^H$
4: for time step $t$ in episode do
5:     $\mathbf{a}_t \sim \pi(\mathbf{x}_t, \mathbf{M}_{p_t})$
6:     $\mathbf{u}_t, \mathbf{h}_t \leftarrow \phi(\mathbf{x}_t, \mathbf{M}_{p_t}, \mathbf{a}_t, \mathbf{h}_{t-1})$ {$\mathbf{u}_t \in \mathbb{R}^3$}
7:     $\tilde{\mathbf{u}}_t \leftarrow \text{softmax}(\mathbf{u}_t)$
8:     $d_t \sim \text{Cat}(\tilde{\mathbf{u}}_t)$
9:     $c_t \sim \psi(\mathbf{x}_t, \mathbf{M}_{p_t}, \mathbf{a}_t)$
10:    $p_{t+1} \leftarrow p_t + c_t d_t$
11: end for
```

OLSK with extended range

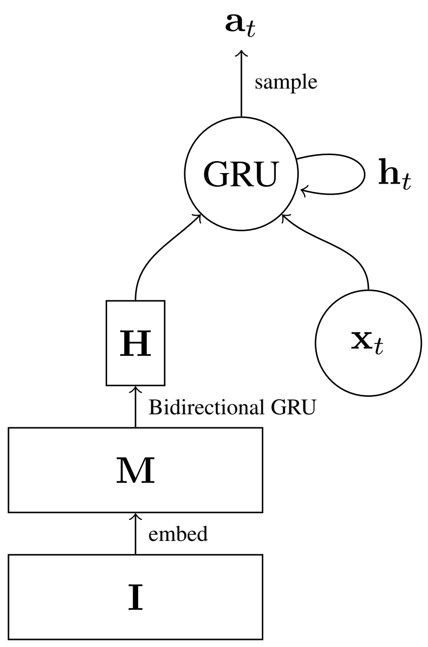
```
1: $p_0 \leftarrow 0$
2: $\mathbf{M} \leftarrow \text{bag-of-words}_\theta(\mathbf{I})$
3: initialize $\mathbf{h}_0 \in \mathbb{R}^H$
4: for time step $t$ in episode do
5:     $\mathbf{a}_t \sim \pi(\mathbf{x}_t, \mathbf{M}_{p_t})$
6:     $\mathbf{u}_t, \mathbf{h}_t \leftarrow \phi(\mathbf{x}_t, \mathbf{M}_{p_t}, \mathbf{a}_t, \mathbf{h}_{t-1})$ {$\mathbf{u}_t \in \mathbb{R}^{2N}$}
7:     $\tilde{\mathbf{u}}_t \leftarrow \text{softmax}(\mathbf{u}_t)$
8:     $d_t \sim \text{Cat}(\tilde{\mathbf{u}}_t)$
9:     $c_t \sim \psi(\mathbf{x}_t, \mathbf{M}_{p_t}, \mathbf{a}_t)$
10:    $p_{t+1} \leftarrow p_t + c_t d_t$
11: end for
```

CoFCA

```
1: $p_0 \leftarrow 0$
2: $\mathbf{M} \leftarrow \text{bag-of-words}_\theta(\mathbf{I})$
3: for time step $t$ in episode do
4:     $\mathbf{H} \leftarrow \text{BI-GRU}(\mathbf{M}, \mathbf{x}_t)$ {Here $\mathbf{H}$ refers to the *last* output of $\text{BI-GRU}(\mathbf{M})$}
5:     $\mathbf{P} \leftarrow \xi(\mathbf{H})$ { $\xi$ is a linear projection}
6:     $\mathbf{a}_t \sim \pi(\mathbf{x}_t, \mathbf{M}_{p_t})$
7:     $\mathbf{u}_t \leftarrow \phi(\mathbf{x}_t, \mathbf{M}_{p_t}, \mathbf{a}_t)$
8:     $\tilde{\mathbf{u}}_t \leftarrow \text{softmax}(\mathbf{u}_t)$
9:     $d_t \sim \text{Cat}(\mathbf{P}\tilde{\mathbf{u}}_t)$
10:    $c_t \sim \psi(\mathbf{x}_t, \mathbf{M}_{p_t}, \mathbf{a}_t)$
11:    $p_{t+1} \leftarrow p_t + c_t d_t$
12: end for
```



(a) Schematic of OLSK / OLSK (extended-range).



(b) Schematic of Unstructured Memory.

Figure 2. Schematics for baselines. Note that the schematic for CoFCA does not differ from CoFCA-S and is therefore omitted.

F. Hyperparameters

F.1. StarCraft

	CoFCA-S	CoFCA	Unstructured Memory	OLSK	OLSK-E
convolution hidden sizes	250	250	150	250	250
convolution kernel sizes	2	2	2	2	2
convolution strides	1	1	1	1	1
ϕ hidden size	250	250	200	200	200
E	200	100	100	150	150
entropy coefficient	0.02	0.02	0.02	0.02	0.02
learning rate	8e-5	7.5e-5	4e-	4e-05	4e-05
L	3	2	NA	NA	NA
time steps per gradient update	35	30	35	30	30
gradient steps per update	6	7	7	7	7

F.2. Minecraft

	CoFCA-S	CoFCA	Unstructured Memory	OLSK	OLSK-E
convolution hidden sizes	32,32	64,32	32,32	64,16	64,16
convolution kernel sizes	2,2	2,2	2,2	2,2	2,2
convolution strides	2,2	2,2	2,2	2,2	2,2
ϕ hidden size	128	128	64	64	64
E	64	32	64	32	32
entropy coefficient	0.015	0.015	0.015	0.015	0.015
learning rate	0.0025	0.0025	0.0025	0.0025	0.0025
L	2	9	NA	NA	NA
time steps per gradient update	25	25	25	25	25
gradient steps per update	2	2	2	2	2