Algorithm 1 Easy First Composition

```
Input: data $X = [x_1, x_2, ....x_n]$
while True do
    if $len(X) == 1$ then
        return $X[0]$
    end if
    if $len(X) == 2$ then
        return $cell(X[0], X[1])$
    end if
    $Children_L, Children_R \leftarrow X[: len(X) - 1], X[1 :]$
    $Parents \leftarrow [cell(child_L, child_R)$ for $child_L, child_R$ in $zip(Children_L, Children_R]$
    $Scores \leftarrow [scorer(parent)$ for $parent$ in $Parents]$
    $index \leftarrow argmax(Scores)$
    $X[index] \leftarrow Parents[index]$
    Delete $X[index + 1]$
end while
```

A PSEUDOCODES

We present the pseudocode of the easy first composition in Algorithm 1 and the pseudocode of BT-cell in Algorithm 2. Note that the algorithms are written as they are for the sake of illustration: in practice, many of the nested loops are made parallel through batched operations in GPU.

A.1 BEAM TREE CELL ALGORITHM

Here, we briefly describe the algorithm of BT-cell (Algorithm 2) in words. In BT-Cell, instead of maintaining a single sequence per sample, we maintain some k (initially 1) number of sequences and their corresponding scores (initialized to 0). k is a hyperparameter defining the beam size. Each sequence (henceforth, interchangeably referred to as "beam") is a hypothesis representing a particular sequence of choices of parents. Thus, each beam represents a different path of composition (for visualization see Figure 1). At any moment the score represents the log-probability for its corresponding beam. Now, we describe the steps in each iteration in the recursion of BT-Cell. **Step 1:** similar to gumbel-tree models, we create all candidate parent compositions for each of the k beams. **Step 2:** we score the candidates with the $score$ function (defined in §2.2). **Step 3:** we choose top-k highest scoring candidates. We treat the top-k choices as mutually exclusive. Thus, each of the k beams encounters k branching choices, and are updated into k distinct beams (similar to before, the children are replaced by the chosen parent). Thus, we get $k \times k$ beams. **Step 4:** we update the beam scores. The sub-steps involved in the update are described next. **Step 4.1:** we apply a log-softmax to the scores of the latest candidates to put the scores into the log-probability space. **Step 4.2:** we add the log-softmaxed scores of the latest chosen candidate to the existing beam score for the corresponding beam where the candidate is chosen. As a result, we will have $k \times k$ beam scores. **Step 5:** we truncate the $k \times k$ beams and beam scores into k beams and their corresponding k scores to prevent exponential increase of the number of beams. For that, we again simply use a top-k operator to keep only the highest scored beams.

At the end of the recursion, instead of a single item representing the sequence-encoding, we will have k beams of items with their k scores. At this point, to get a single item, we do a weighted summation with the softmaxed scores as the weights as described in §3.

B GATED RECURSIVE CELL (GRC)

The Gated Recursive Cell (GRC) was originally introduced by Shen et al. (2019a) drawing inspiration from the Transformer's feed-forward networks. In our implementation, we use the same variant of GRC as was used in Chowdhury & Caragea (2021) where a GELU Hendrycks & Gimpel (2016)

Algorithm 2 Beam Tree Cell

```
**Input:** data $X = [x_1, x_2, ....x_n], k$ (beam size)
$BeamX \leftarrow [X]$
$BeamScores \leftarrow [0]$
while True do
  if $len(BeamX[0]) == 1$ then
    $BeamX \leftarrow [beam[0]$ for $beam$ in $BeamX]$
    break
  end if
  if $len(BeamX[0]) == 2$ then
    $BeamX \leftarrow [cell(beam[0], beam[1])$ for $beam$ in $BeamX]$
    break
  end if
  $NewBeamX \leftarrow []$
  $NewBeamScores \leftarrow []$

  for $Beam$,$BeamScore$ in $zip(BeamX, BeamScores)$ do
    $Parents \leftarrow [cell(beam[i], beam[i+1])$ for $i$ in $range(0, len(beam) - 1)]$
    $Scores \leftarrow log \circ softmax([scorer(parent)$ for $parent$ in $Parents])$
    $Indices \leftarrow topk(Scores, k)$

    for $i$ in $range(K)$ do
      $newBeam \leftarrow deepcopy(Beam)$
      $newBeam[Indices[i]] \leftarrow Parents[Indices[i]]$
      Delete $newBeam[Indices[i] + 1]$
      $NewBeamX.append(newBeam)$
      $newScore \leftarrow BeamScore + Scores[indices[i]]$
      $newBeamScores.append(newScore)$
    end for
  end for
  $Indices \leftarrow topk(newBeamScores, k)$
  $BeamScores \leftarrow [newBeamScores[i]$ for $i$ in Indices]
  $BeamX \leftarrow [newBeamX[i]$ for $i$ in Indices]
end while
$BeamScores \leftarrow Softmax(BeamScores)$
Return $sum([score * X$ for $score, X$ in $zip(BeanScores, BeamX)])$
```

activation function was used. We present the equations of GRC here:

$$\begin{bmatrix} z_i \\ h_i \\ c_i \\ u_i \end{bmatrix} = W_2 \text{ GeLU} \left(W_1^{Cell} \begin{bmatrix} child_{left} \\ child_{right} \end{bmatrix} + b_1 \right) + b_2 \tag{6}$$

$$o_i = LN(\sigma(z_i) \odot child_{left} + \sigma(h_i) \odot child_{right} + \sigma(c_i) \odot u_i) \tag{7}$$

σ is $sigmoid$; o_i is the parent composition $\in \mathbb{R}^{d_h \times 1}$; $child_{left}, child_{right} \in \mathbb{R}^{d_h \times 1}$; $W_1^{cell} \in \mathbb{R}^{d_{cell} \times 2 \cdot d_h}$; $b_1 \in \mathbb{R}^{d_{cell} \times 1}$; $W_2 \in \mathbb{R}^{d_h \times d_{cell}}$; $b_1 \in \mathbb{R}^{d_h \times 1}$. We use this same GRC function for any recursive model (including our implementation of Ordered Memory) that constitutes GRC.

C BSRP-GRC DETAILS

For the decisions about whether to shift or reduce, we use a scorer function similar to that used in Chowdhury & Caragea (2021). Where Chowdhury & Caragea (2021) use the decision function on concatenation of local hidden states (n-gram window), we use the decision function on the concatenation of last two items in the stack and the next item in the queue. The output is a scalar sigmoid activated logit score s. We then treat $log(s)$ as the score for reducing in that step, and $log(1 - s)$

Model	**near-IID**	**Length Gen.**			**Argument Gen.**		**LRA**
(Lengths)	$\leq$ 1000	200-300	500-600	900-1000	100-1000	100-1000	2000
(Arguments)	$\leq$ 5	$\leq$ 5	$\leq$ 5	$\leq$ 5	10	15	10
BT-GRC+SOFT	69	44	37.1	29.4	39.5	38.6	31.6

Table 3: Accuracy of BT-GRC+SOFT on ListOps. We report the median of 3 runs. The model was trained on lengths $\leq$ 100, depth $\leq$ 20, and arguments $\leq$ 5.

as the score for shifting in that step. The scores are manipulated appropriately for edge cases (when there are no next item to shift, or when there are no two items in the stack to reduce). Besides that, we use the familiar beam search strategy over standard shift-reduce parsing. Finally the beams of final states are merged through the weighted summation of the states based on the softmaxed scores of each beam similar to BT-Cell models as described in §3.

D ST-GUMBEL TOP-K ISSUES

Let us use the notation $argmax_k(.)$ to denote the function for selecting the index of the k^{th} highest value element in a vector or a list. Let us say, we have s as a vector such that s_i represents the element at the i^{th} dimension of s. We can now, represent the standard straight-through gumbel softmax Choi et al. (2018) as follows:

$$p = \text{softmax}(s + G) \tag{8}$$

$$(\text{one\_hot}(\text{argmax}_1(p)) - p).detach() + p \tag{9}$$

Here G is the gumbel noise. This strategy allows the forward propagation to have discrete one hot values (one_hot(argmax$_1(p)$)) while backpropagation to propagate through the soft p. However, this creates a discrepancy between forward propagation and backpropagation that leads to biased gradient estimation. This problem can be exacerbated in ST-Gumbel Top-k. In this method, instead of top 1, top k items are selected from p. The straight-through estimation involved in the selection of some r^{th} item among the top k selections ($r \leq k$) can be then expressed as:

$$(\text{one\_hot}(\text{argmax}_r(p)) - p).detach() + p \tag{10}$$

Note that $p_{\text{argmax}_r(p)} \leq p_{\text{argmax}_1(p)}$. Thus the discrepancy between p and one_hot(argmax$_r(p)$) will only increase with increasing r. Moreover while originally in ST-Gumbel, when selecting top-1, p_i can be interpreted (in our context) as the probability that s_i is "top-1" item. But in ST-Gumbel Top-k, when selecting the top r^{th}, item there is no corresponding interpretation - p_i would still indicate probability for s_i being the "top-1" item not the probability of s_i being the "top r^{th}" item. Overall, we attempted ST-Gumbel Top-k more so because it's a simple naive extension of plain top-k but it's not a particularly principled approach. As such, the poor performance is not that surprising. A possible extension could be to transform p to some p^r for every top r^{th} selection such that p_i^r do reflect probability for s_i to be top r^{th} item but that would be non-trivial to do without adding significant overhead (similar to that caused by differentiable sorting - see Appendix E.6).

E ADDITIONAL EXPERIMENTS AND ANALYSIS

E.1 DIFFERENTIABLE SORTING

Besides the three extensions considered for BT-Cell, another strategy can be to simply use differentiable versions of top-k operators or sorting functions (Adams & Zemel, 2011; Grover et al., 2019; Cuturi et al., 2019; Xie et al., 2020; Blondel et al., 2020; Petersen et al., 2021; 2022). However, as we discussed before these methods can lead to issues related to "washing out" of representations due to using soft permutation matrices leading to higher error accumulation (although temperature can be used to partially counteract that (Petersen et al., 2021)). Besides that, using these techniques in a recursive loop can lead to significant overhead and added time complexity. Instead, in this paper, we mainly aim to show that a simple method, namely, softpath can already bring marked improvement compared to plain top-k especially in low beam size settings for listops. We keep a more exhaustive investigation of application of differentiable sorters for beam search as a future work. Nevertheless,

Model	**DG**	**Length Gen.**			**Argument Gen.**		**LRA**
(Lengths)	≤ 100	200-300	500-600	900-1k	100-1k	100-1k	2K
(Arguments)	≤ 5	≤ 5	≤ 5	≤ 5	10	15	≤ 10
(Depths)	8-10	≤ 20	≤ 20	≤ 20	≤ 10	≤ 10	≤ 10
With gold trees							
GoldTreeGRC	99.95	99.95	99.9	99.8	76.95	77.1	74.55
Baselines without gold trees							
CYK-GRC	99.45	99.0	—	—	67.8	35.15	—
Ordered Memory	**99.95**	99.8	99.25	96.4	**79.95**	**77.55**	**77**
CRvNN	99.9	99.4	99.45	98.9	65.7	43.4	65.1
Beam Tree Models with beam size 5 (also without gold trees)							
BT-LSTM	98.9	98.5	98.1	97.7	74.75	40.75	65.05
BT-GRC	**99.95**	**99.95**	**99.95**	**99.9**	75.35	72.05	68.1
BT-GRC + Softpath	99.9	99.6	98.1	97.1	78.1	71.25	75.45
BT-GRC + Gumbelpath	99.9	**99.95**	99.8	99.7	75.2	72.75	71.8
Beam Tree Models with beam size 2 (also without gold trees)							
BT-GRC + Softpath	96.2	95.40	93.8	91.2	64.45	51.95	51.05

Table 4: Accuracy on ListOps-DG. We report the median of 3 runs except in the last block where we report the mean/std of 10 runs as mentioned. Our models were trained on lengths ≤ 100, depth ≤ 6, and arguments ≤ 5. We bold the best results and underline the second-best among models that do not use gold trees.

.

we present some preliminary results here. Xie et al. (2020) used an optimal transport-based differentiable top k method in beam decoding for machine translation. Here, we use their method (SOFT Top-k) and create a new variant of BT-GRC by replacing the softpath with SOFT Top-k. We call this new variant as BT-GRC + SOFT. We run this model in ListOps with the same dataset settings as used in Table 1. We report the results in Table 3. As we can see in Table 3, BT-GRC+SOFT shows very poor performance. This supports our hypothesis that using soft permutation matrix in all recursive iterations may not be ideal because of increase chances of error accumulation and "washing out" through interpolations of what would be distinct path representations. In Appendix E.6, we also demonstrate that using SOFT significantly slows down BT-GRC.

E.2 LISTOPS-DG EXPERIMENT

Dataset Settings: The length generalization experiments in ListOps do not give us an exact perspective in depth generalization[4] capacities. So there is a question of how models will perform in unseen depths. To check for this, we create a new ListOps split which we call "ListOps-DG". For this split, we create $100,000$ training data with arguments ≤ 5, lengths ≤ 100, and depths ≤ 6. We create 2000 development data with arguments ≤ 5, lengths ≤ 100, and depths 7. We create 2000 test data with arguments ≤ 5, lengths ≤ 100, and depths 8-10. In addition, we also still tested on the same length-generalization splits (which now simultaneously have much higher depths too: ≤ 20), argument generalization splits, and LRA. The results are presented in Table 4. We only evaluate the models that were promising ($\geq 90\%$ in near IID settings) in the original ListOps split. We report the median of 3 runs for each model (except in the last block of the table).

Results: Interestingly, we find that base BT-GRC, CRvNN, and Ordered Memory now does much better in length generalization compared to the original listops split. We think this is because of the increased data (the training data in the original ListOps is ~ 75000 after filtering data of length > 100 whereas here we generated $100,000$ training data). However, while the median of 3 runs in Ordered Memory is decent, we found one run to have very poor length generalization performance. To investigate more deeply if Ordered Memory has a particular stability issue, we ran Ordered Memory for 10 times with diferent seeds, and we find that it frequently fails to learn to generalize over length. As a baseline, we also ran BT-GRC similarly for 10 runs and found it to be much more stable. We report the mean and standard deviation of 10 runs of Ordered Memory and BT-GRC

---

[4] By depth, we simply mean the maximum number of nested operators in a given sequence in case of ListOps

Model	**DG**	**Length Gen.**			**Argument Gen.**		**LRA**
(Lengths)	≤ 100	200-300	500-600	900-1k	100-1k	100-1k	2K
(Arguments)	≤ 5	≤ 5	≤ 5	≤ 5	10	15	≤ 10
(Depths)	8-10	≤ 20	≤ 20	≤ 20	≤ 10	≤ 10	≤ 10
Stability Test: Mean/Std with 10 runs. Beam size 5 for BT-GRC							
Ordered Memory	$99.94_{0.6}$	97.58_{32}	78.785_{197}	61.85_{291}	77.66_{30}	69.03_{107}	67.35_{125}
BT-GRC	$99.84_{1.5}$	$99.58_{5.8}$	98.8_{21}	97.85_{39}	73.82_{57}	66.21_{107}	66.975_{102}

Table 5: Accuracy on ListOps-DG (Stability test). We report the mean and standard deviation of of 10. Our models were trained on lengths ≤ 100, depth ≤ 6, and arguments ≤ 5. Subscript represents standard deviation. As an example, $90_1 = 90 \pm 0.1$

.

Model	**DG1**	**Length Gen.**			**Argument Gen.**		**LRA**
(Lengths)	≤ 50	200-300	500-600	900-1000	100-1000	100-1000	2000
(Arguments)	≤ 5	≤ 5	≤ 5	≤ 5	10	15	≤ 10
(Depths)	8-10	≤ 20	≤ 20	≤ 20	≤ 10	≤ 10	≤ 10
After Training on ListOps-DG1							
NDR (layer 24)	96.7	48.9	32.85	22.1	65.65	64.6	42.6
NDR (layer 48)	91.75	34.60	24.05	19.7	54.65	52.45	39.95
Model	**DG2**	**Length Gen.**			**Argument Gen.**		**LRA**
(Lengths)	≤ 100	200-300	500-600	900-1000	100-1000	100-1000	2000
(Arguments)	≤ 5	≤ 5	≤ 5	≤ 5	10	15	≤ 10
(Depths)	8-10	≤ 20	≤ 20	≤ 20	≤ 10	≤ 10	≤ 10
After Training on ListOps-DG2							
NDR (layer 24)	95.6	44.15	30.7	20.3	67.85	58.05	46
NDR (layer 48)	92.65	38.6	29.15	22.1	73.1	64.4	50.4

Table 6: Accuracy on ListOps-DG1 and ListOps-DG2. We report the max of 3 runs. In ListOps-DG1, NDR was trained on lengths ≤ 50, depth ≤ 6, and arguments ≤ 5. In ListOps-DG1, NDR was trained on lengths ≤ 100, depth ≤ 6, and arguments ≤ 5. Lyaers denote the layers used during inference.

.

in Table 5. As can be seen, the mean of BT-GRC is much higher than that of Ordered Memory in length generalization splits.

E.3 NDR EXPERIMENTS

Dataset Settings: Neural Data Routers (NDR) is a Transformer-based model that was shown to perform well in algorithmic tasks including Listops (Csordás et al., 2022). We tried some experiments with it too. We found NDR to be struggling in the original ListOps splits or the ListOps-DG split. We noticed that in the paper (Csordás et al., 2022), NDR was trained in a much larger sample size ($\sim$ 10 times more data than in ListOps-DG) and also on lower sequence lengths ($\sim$ 50). To better check for the capabilities of NDR, we created two new ListOps split - DG1 and DG2. In DG1, we set the sequence length to 10-50 in training, development, and testing set. We created 1 million data for training, and 2000 data for development and testing. Other parameters (number of arguments, depths etc.) are same as in ListOps-DG split. Split DG2 is the same as ListOps-DG split in terms of data-generation parameters (i.e it includes length sizes ≤ 100) but with much larger sample size for the training split (again, 1 million samples same as DG1). We present the results in Table 6.

Results: We find that even when we focus on the best of 3 runs in the table, although NDR generalizes to slightly higher depths (8-10 from ≤ 6) (as reported in (Csordás et al., 2022)), it still struggles with splits with orders of magnitude higher depths, lengths, and unseen arguments. Following the suggestions of Csordás et al. (2022), we also increase the number of layers during inference (eg. upto 48) to handle higher depth sequences but that did not help substantially. Thus, even after experiencing more data, NDR generalizes worse than Ordered Memory, CRvNN, or BT-GRC. Moreover, NDR requires some prior estimation of the true computation depth of the task for its hyperparameter setup unlike the other latent-tree models.

Model	**Number of Operations**					
	8	9	10	11	12	C
With gold trees						
GoldTreeGRC	97.14_{1}	96.5_{2}	$95.29_{2.5}$	$94.21_{9.9}$	$93.67_{7.7}$	$97.41_{1.6}$
Baselines without gold trees						
Transformer*	52	51	51	51	48	51
Universal Transformer*	52	51	51	51	48	51
ON-LSTM*	87	85	81	78	75	60
Self-IRU†	95	93	92	90	88	—
RecurrentGRC	93.04_{6}	$90.43_{4.9}$	88.48_{6}	$86.57_{5.8}$	$80.58_{1.5}$	$83.17_{5.1}$
BalancedTreeGRC	$77.81_{3.6}$	$72.56_{6.6}$	$67.54_{6.4}$	$63.66_{6.6}$	$57.44_{7.6}$	74.45_{10}
RandomTreeGRC	$86.67_{5.1}$	$84.78_{8.2}$	$80.45_{8.2}$	$76.62_{8.5}$	$71.71_{4.7}$	$78.06_{9.7}$
GumbelTreeLSTM	77.03_{12}	$74.62_{6.1}$	$69.55_{1.7}$	$67.94_{8.1}$	59.95_{10}	78.20_{11}
GumbelTreeGRC	93.46_{14}	91.89_{19}	90.33_{22}	88.43_{18}	85.70_{24}	89.34_{29}
CYK-GRC	$96.62_{2.3}$	$96.07_{4.6}$	94.67_{11}	$93.44_{8.8}$	$92.54_{9.3}$	77.08_{27}
BSRP-GRC	89.37_{18}	85.92_{30}	83.06_{35}	80.63_{33}	74.91_{42}	81.88_{31}
CRvNN	$96.9_{3.7}$	$95.99_{2.8}$	$94.51_{2.9}$	$\underline{94.48_{5.6}}$	92.73_{15}	89.79_{58}
Ordered Memory	$\mathbf{97.5_{1.6}}$	$\mathbf{96.74_{1.4}}$	94.95_{2}	$93.9_{2.2}$	$93.36_{6.2}$	$\mathbf{94.88_{7}}$
Beam Tree Models with beam size 5 (also without gold trees)						
BT-LSTM	$93.27_{2.5}$	$92.63_{5.3}$	88.55_{10}	$87.85_{8.2}$	84.56_{12}	73.02_{12}
BT-GRC	96.83_{1}	$95.99_{2.4}$	$95.04_{2.3}$	$94.29_{3.8}$	$93.36_{2.4}$	$\underline{94.17_{14}}$
Gumbel-BT-GRC	95.56_{8}	94.14_{11}	92.77_{16}	91.71_{21}	89.84_{18}	86.55_{35}
BT-GRC + Softpath	$\underline{97.03_{1.4}}$	$\underline{96.49_{1.9}}$	$\underline{95.43_{4.5}}$	$94.21_{6.6}$	$\underline{93.39_{1.5}}$	78.04_{43}
BT-GRC + Gumbelpath	$96.93_{1.2}$	$95.82_{1.5}$	$94.67_{0.6}$	$93.48_{2.9}$	92.58_{9}	83.63_{47}
Beam Tree Models with beam size 2 (also without gold trees)						
BT-GRC	$96.63_{2.2}$	$95.95_{3.7}$	$\mathbf{95.45_{3.1}}$	$93.71_{5.5}$	$93.28_{3.9}$	86.97_{38}
BT-GRC + Softpath	$96.96_{1.1}$	$\underline{96.49_{1.2}}$	$95.38_{1.8}$	$\mathbf{94.64_{3.4}}$	$\mathbf{93.55_{3.3}}$	81.39_{65}
BT-GRC + Gumbelpath	$96.48_{1.5}$	$96.02_{4.7}$	$94.71_{7.2}$	93.94_{7}	$92.46_{1.1}$	90.87_{50}

Table 7: Mean accuracy and standard deviaton on the Logical Inference for ≥ 8 number of operations after training on samples with ≤ 6 operations. We also report results of the systematicity split C. We bold the best results and underline the second-best for all models without gold trees. * indicates that the results were taken from Shen et al. (2019a) and † indicates results from Zhang et al. (2021). Our models were run 3 times on different seeds. Subscript represents standard deviation. As an example, $90_1 = 90 \pm 0.1$

.

E.4 SYNTHETIC LOGICAL INFERENCE RESULTS

Dataset Settings: We also consider the synthetic testbed for detecting logical relations between sequence pairs as provided by Bowman et al. (2015b). Following Tran et al. (2018), we train the models on sequences with ≤ 6 operators and test on data with greater number of operators (here, we check for cases with ≥ 8 operators) to check for capacity to generalize to unseen number of operators. Similar to Shen et al. (2019a); Chowdhury & Caragea (2021), we also train the model on the systematicity split C. In this split we remove any sequence matching the pattern $*(and(not*))*$ from the training set and put them in the test set to check for systematic generalization.

Results: In Table 7, in terms of operation generalization, our proposed BT-Cell models perform similarly to prior SOTA models like Ordered Memory (OM) and CRvNN while approximating GoldTreeGRC for both beam sizes. GRC-based models perform better than comparative LSTM-based models. Unsurprisingly, following discussions in Appendix D, Gumbel-BT-GRC does not perform as well. In terms of systematicity (split C), OM and BT-GRC (with beam size 5) perform similarly (both above 94%) and much better than the other models. Lower beam size extensions hurts systematicity performance for BT-GRC which is not too surprising given we are limiting the beam size. Surprisingly, however, softpath/gumbelpath extensions also hurt systematicity. CYK-GRC shows promise in operator generalization but shows poor systematicity as well. Gumbel-GRC

Model	**M**	**MM**	**Len M**	**Len MM**	**Neg M**	**Neg MM**
RecurrentGRC	71.2_3	71.4_4	49_{25}	49.5_{24}	49.3_6	50.1_6
BalancedTreeGRC	71.1_5	71.4_1	59_8	60.7_5	50.2_4	50.4_6
RandomTreeGRC	72.2_3	72.3_5	61.4_{23}	62.3_{23}	51.7_3	52.7_7
GumbelTreeGRC	71.2_7	71.2_6	57.5_{17}	59.6_{12}	50.5_{20}	51.8_{20}
CRvNN	72.2_4	72.6_5	62_{44}	63.3_{47}	52.8_6	53.8_4
Ordered Memory	72.5_3	$\mathbf{73_2}$	56.5_{33}	57.1_{31}	50.9_7	51.7_{13}
Beam Tree Models with beam size 5						
BT-GRC	71.6_2	72.3_1	64.7_6	66.4_5	$\mathbf{53.7_{37}}$	$\mathbf{54.8_{43}}$
BT-GRC + Softpath	71.7_1	71.9_2	65.6_{13}	66.7_9	53.2_2	54.2_5
BT-GRC + Gumbelpath	72.1_3	71.9_1	66.3_7	66.9_{14}	51.6_{19}	52.2_{21}
Beam Tree Models with beam size 2						
BT-GRC	$\mathbf{72.6_1}$	72.6_2	$\mathbf{66.6_5}$	$\mathbf{68.1_6}$	53.3_{21}	54.4_{24}
BT-GRC + Softpath	71.1_3	71.9_1	63.7_{17}	65.6_{12}	51.8_{19}	53_{12}
BT-GRC + Gumbelpath	67_{45}	67.8_{45}	55.1_{67}	56.2_{67}	47.8_{42}	48.3_{43}

Table 8: Mean accuracy and standard deviaton on MNLI. Our models were run 3 times on different seeds. Subscript represents standard deviation. As an example, $90_1 = 90 \pm 0.1$

.

performs better than fixed tree models (ReccurentGRC or BalancedTreeCell) or RandomTreeCell but still far from SOTA which is not unexpected given its poor results in ListOps.

E.5 NATURAL LANGUAGE INFERENCE EXPERIMENTS

Dataset Settings: We ran our models on MNLI (Williams et al., 2018) which is a natural language inference task. We tested our models on the development set of MNLI and use a randomly sampled subset of $10,000$ data points from the original training set as the development set. Our training setup is different from Chowdhury & Caragea (2021) and other prior latent tree models which combines SNLI (Bowman et al., 2015a) and MNLI training sets (we don't add SNLI data.). We filter sequences ≥ 150 from the training set for efficiency. We also test our models in various stress tests (Naik et al., 2018). We report the results in Table 8. M denotes matched development set (used as test set) of MNLI. MM denotes mismatched development set (used as test set) of MNLI. LenM denotes length matched stress set from (Naik et al., 2018). LenMM denotes length mismatched stress set from (Naik et al., 2018). NegM denotes negation matched stress set from (Naik et al., 2018). NegMM denotes negation mismatched stress set from (Naik et al., 2018). Length matched/mismatched stress sets add to the length of the premise by adding tautologies. Negation matched/mismatched stress sets add tautologies containing "not" terms which can bias the model to falsely predict contradictions.

Results: Results in Table 8 show BT-GRC variants are not particularly better than the other models in the standard matched/mismatched sets. However, discounting Gumbelpath (which tends to break down), even the weakest BT-GRC variants outperform all other models on length matched/mismatched stress test. BT-GRC (with beam 2 or 5) or BT-GRC with Softpath and beam 5 also tend to do marginally better than other models in negation matched/mismatched test sets. Overall most BT-Cell variants show better robustness to stress tests. We ignore testing Gumbel-BT-GRC because it generally is a bad performer.

Overall, given both the performance here and at IMDB, gumbel-based models (including gumbelpath) may be better avoided in general (Also consider that GumbelTreeGRC, here, performs worse than RandomTreeGRC).

E.6 EFFICIENCY ANALYSIS

Settings: In Table 9, we compare the empirical performance of various models in terms of time and memory. We ran each models on ListOps splits of different sequence lengths (200-250, 500-600, and 900-1000). Each split contains 100 samples. We ran each model with the batch size of 1. Other hyperparameters are same as those used for ListOps. Note that we are showing time and memory consumption during training (not inference). All models are ran in an Nvidia RTX A6000 GPU.

	Sequence Lengths					
Model	$\mathbf{200-250}$		$\mathbf{500-600}$		$\mathbf{900-1000}$	
	Time	Memory	Time	Memory	Time	Memory
RecurrentGRC	0.2 min	0.02 GB	0.5 min	0.02 GB	1.3 min	0.03 GB
BalancedTreeGRC	0.3 min	0.02 GB	1.2 min	0.03 GB	2.1 min	0.04 GB
RandomTreeGRC	0.4 min	0.33 GB	1.4 min	1.86 GB	3.0 min	5.18 GB
GumbelTreeGRC	0.5 min	0.35 GB	2.1 min	1.95 GB	3.5 min	5.45 GB
CYK-GRC	9.3 min	32.4 GB	OOM	OOM	OOM	OOM
BSRP-GRC	2.3 min	0.06 GB	6.1 min	0.19 GB	10.5 min	0.42 GB
Ordered Memory	8.0 min	0.09 GB	20.6 min	0.21 GB	38.2 min	0.35 GB
CRvNN	1.5 min	1.57 GB	4.3 min	12.2 GB	8.0 min	42.79 GB
Beam Tree Models with beam size 5						
BT-GRC	1.1 min	1.71 GB	2.6 min	9.82 GB	5.1 min	27.27 GB
BT-GRC + Softpath	1.4 min	2.74 GB	4.0 min	15.5 GB	7.1 min	42.95 GB
BT-GRC + SOFT	5.1 min	2.67 GB	12.6 min	15.4 GB	23.1 min	42.78 GB
Beam Tree Models with beam size 2						
BT-GRC	1.1 min	0.68 GB	2.6 min	3.92 GB	5.1 min	10.90 GB
BT-GRC + Softpath	1.4 min	0.88 GB	4.0 min	5.03 GB	7.1 min	14.01 GB

Table 9: Empirical time and memory consumption for various models. Ran on 100 ListOps data of different sequence lengths wiht batch size 1

.

Discussions:

Assume n denotes the sequence length, d denotes the hidden state dimensions, k denotes the beam size, and m denotes the number of memory slots (for Ordered Memory).

RecurrentGRC: Recurrent models performs reasonably well. While it need to go through n sequential steps but so do most other models. Moreover, computation in each iteration of the sequential loop is relatively simple - it is just the application of the recursive cell function. The space complexity is also very little because of lack of parallel processing accross the sequence length (it has to only store the hidden state memory O(d)). Thus the memory consumption stays nearly constant with increasing sequence length.

BalancedTreeGRC: BalancedTreeGRC is also moderately efficient. It can slightly increase memory consumption because of more parallelized processing. It composes multiple children in every step. However, since it cuts off half of the sequence at every step, its outer sequential loop is only in the order of $O(logn)$. Thus, its memory consumption does not increases as much compared to GumbelTreeGRC or RandomTreeGRC. While the total sequential steps is much less for BalancedTreeGRC, it seems that the added overhead from more parallelized processing per iteration still makes it slightly slower than RecurrentGRC in practice (still it is faster than any other models). However, there may be room for better code optimization for BalancedTreeCell.

RandomTreeCell and GumbelTreeCell: Both are similar in terms of complexity. Both chooses one composition at a time requiring them to still go through n seuqntial steps in the outer loop unlike BalancedTreeGRC. However, unlike RecurrentGRC, in each loop, it has to also parallely compute all possible parent compositions - this leads to higher space complexity for each iteration $O(nd^2)$ that also scales up with increasing sequence length n and also higher computational overhead per iteration in the loop. Thus, they run slower than RecurrentGRC and takes much more memory as well. Nevertheless, in comparison to other latent-tree models these are still relatively fast, simple, and lightweight.

CYK-GRC: CYK-GRC is the worst offender of all in terms of computational efficiency. Note also that GRC can itself be relatively expensive cell function because of high dimensional feedforward networks – adding up overhead compared to classical CYK models. CYK-GRC takes a chart-filling approach in a dynamic programming style. The number of cells in the filled chart is $n^2/2$. Each cell in the original CYK algorithm can again have multiple options, however, Maillard et al. (2019) simplifies this by keeping only one option per cell through attention-based pooling. Even after that, each cell still has to contain a d dimensional hidden state. This leads to atleast $o(n^2d)$ space complexity. Moreover, it still has to take n sequential steps in the outer loop to recursively fill up the chart. At the same time each sequential step is also more expensive than any of the other models. For filling up any cell it has to compute all valid possible ways to combine pre-computed cell and then perform an attention pooling. It has to, thus, apply GRC multiple times to all possible ways of composition from prior chart cells. This step can be, however, parallelized (and we do parallelize it). Nevertheless, applying GRC in parellel to all possible left-right children pair (from prior computed chart cells) can still add to memmory consuption and temporal overheads. This reflects in both extremely high memory consumption and also the slowest speed among all else.

BSRP-GRC: BSRP-GRC is essentially a form of stack augmented RNN. Similar to Recurrent-GRC, it's memory consuption is low because of limited parallel processing (although its stack size grow with each iteration which reflects in sharply rising memory with increasing sequence length compared to Recurrent GRC). However, it has added expenses in each recurrent iteration due to additional stack operations and shift/reduce decision computation. In effect this makes the model quite slow although better than some others. BSRPC-GRC taking a shift-reduce parsing strategy also need to increase the sequential steps from n to $2n$. Although that doesn't make an asymptotic difference, it still can further contribute to the empirical slowdown. We use a beam size of 5 for BSRP-GRC here for better comparison against BT-GRC.

Ordered Memory: Ordered Memory (OM) is also a form of stack augmented RNN and thus, have similar memory advantages as RecurrentGRC (although overall memory consumption is relative increased due to storing multiple slots of hidden states). However, the main bottleneck in OM is time. Precisely, OM utilizes a nested loop. The outerloop is the same n times sequential operation as RecurrentGRC, but in the in each step of that loop, OM has to again sequentially apply GRC over the m memory slots. This leads to a $O(nm)$ sequential steps. Thus, we get a heavy hit to time consumption with OM.

CRvNN: The framework of CRvNN is roughly similar to GumbelTreeGRC. However, instead of choosing one parent at a time, it can choose multiple parents at a time, that too in a soft manner. But this comes at a cost. While GumbelTreeGRC can reduce the sequence size with each sequential step, CRvNN has to still maintain the whole sequence in each step with associated "existential probability" for each sequence item. Moreover, computing the composition of contiguous representations also becomes harder. Since in CRvNNs sequence items exists with some probability, what counts as the first existing contiguous item is also probabilistic. In effect, CRvNN needs to create a n^2 attention matrix (similar to Transformers but based on existential probabilities instead of dot product attention) to retrieve neighbor (contiguous) elements to create parent candidates. Furthermore, in each sequential steps, this n^2 attention matrix needs to be applied multiple times - for example, to get local composition scores for sigmoid modulation and also to retrieve some w local items for its convolution-based decision function. That can also add significant overhead (although still quite tame, in terms of temporal overhead, compared to the memory-slot-wise recursion in OM). Thus, in effect, we get increased memory consumption and time compared to GumbelTreeGRC. At the same time, however, efficiency analysis for CRvNN is complicated by the fact that it can dynamically halt early. That is, it does not need to take full n sequential steps unlike any of the other models (besides BalancedTreeGRC). However, this doesn't mean CRvNN is guaranteed to take $O(logn)$ sequentials steps. Ideally, it is intended to take as many sequential steps as is the induced binary tree depth. Underlying binary tree depths are not necessarily always (or even in average) around $O(logn)$. So while it can give an increase performance boost, there is not a clear boost asymptotically. Moreover, we found it challenging to evaluate CRvNN fairly. In the 100 samples dataset, CRvNN with early halting can just induce bad trees (given lack of enough data to be trained well) and halt very fast (unreflective of realistic performance) but increasing number of samples makes the analysis more cumbersome overall for all other models. So we instead show the "worst case" performance of CRvNN. The "worst case" is the case when CRvNN does not halt early; thus, to simulate the "worst case" we disable early halting. The "worst case" is also still practically relevant and needed to be considered to set up the hyperparameters (like batch size) properly so that the model does not run out of memory during training because of the "worst case" induction. With this setup, we find CRvNN to be in-between in terms of performance. We discuss more about it in comparison to BT-Cell models in the next section.

BT-GRC: The time complexity of BT-GRC is $O(n(knd^2 + k^3nd))$. k (beam size) is technically a relative small constant and can be ignored in asymptotic analysis (bringing BT-GRC at a similar complexity level to Gumbel-Tree GRC), but we keep it here for better exposition of the effect of beam size (k). Similar to most other models here, BT-GRC has to take an outer sequential steps of n. In each sequential step, similar to GumbelTreeGRC it has to calculate all parent candidates which can lead to a complexity of $O(nd^2)$, but since now we have to do the same for each of the k beams we have a multiplicative effect: $O(knd^2)$. However, this n^d computation can be done in parallel (all parent candidates are computed in parallel in GumbelTreeGRC). Similarly each beams are independent and can be computed in parallel as well. So in effect the computation cost here is similar to increasing the batch size of GumbelTreeGRC by k. The term k^3nd indicates the selection costs of k vectors of size d from k^2 vectors of size d for a sequence of size n. Again, in practice this can be also parallelized by creating a k^2 permutation matrix and performing a single matrix multiplication in CUDA. Overall, this doesn't add as much cost over parent composition.

As we a priori suspected, empirically, we also verified the increased time/memory cost of BT-GRC in different sequence length to match the effect of increasing the batch size of GumbelTreeGRC by k. Even though, ultimately, k is a constant, in practice, this can still lead to a significant expense. This is where small k (example beam 2) can be valuable. As we can see because most computation through k is parallelized the time is not changed as much in changing from beam size (k) 2 to 5. But the memory consumption can be significantly decreased with lower beam ($k = 2$). Thus, BT-Cell with beam size 2 can be an attractive choice here particularly when it can still perform relatively on par with bigger beam models on synthetic logical inference and natural language tasks. Also with softpath and beam size 2, listops performance is still decent.

Another interesting point to note is that BT-GRC does not rely on any n^2 matrix as CRvNN. Thus, we can see that its memory consumption scales better with increasing length than CRvNN. For example, in sequence length 200-250, CRvNN took slightly less memory than BT-GRC, but in sequence length 900-1000, CRvNN takes nearly twice as much memory compared to BT-GRC.

However, we wouldn't claim that CRvNN is strictly worse than BT-GRC in efficiency departments because CRvNN can be made more efficient by bounding its outer sequential loop and dynamic halting (but these factors are more tricky to fairly analyze).

BT-GRC + Softpath: Although the forward propagation complexity should be nearly the same for Softpath, in practice we find that adding Softpath increases both the memory and time significantly compared to base BT-GRC. We find that this is because of added backpropagation expenses because of more complicated gradient propagation. We verified this by keep the forward network of Softpath variant the same and using Pytorch's $detach()$ function to cut the gradient from Top-K Softpath selection. This change leads to similar empirical efficiency to base BT-GRC. Nevertheless, despite the added costs, Softpath is still comparable to CRvNN and much faster than OM, CYk-GRC, SOFT top-k, and BSRP-GRC. Moreover, Softpath with beam size 2 still remains an attractive option with further lowered memory consumption.

BT-GRC + SOFT: As we already claimed before using differentiable sorting algorithm (SOFT Top-k) (Xie et al., 2020) in a recursive loop can bring significant overhead and slowdown. We show it empirically in Table 9. Replacing Softpath with SOFT Top-k can increase the time taken by around $3\times$ compared to BT-GRC+Softpath.

E.7 PARSE TREE ANALYSIS

In this section, we analyze the induced structures of BT-Cell models. Note, however, although induced structures can provide some insights to the model, we can draw limited conclusions from them. First, if we take a stance similar to Choi et al. (2018) in considering it suitable to allow different kinds of structures to be induced as appropriate for a specific task then it's not clear how structures should be evaluated by themselves (besides just the donwstream task evaluations). Second, the extracted structures may not completely reflect what the models may implicitly induce because the recursive cell can override some of the parser decisions (given how there is evidence that even simple RNNs Bowman et al. (2015b) can implicitly model different tree structures within its hidden states to an extent even when its explicit structure always conform to the left-to-right order of composition). Third, even if the extracted structure perfectly reflects what the model induces, another side of the story is the recursive cell itself and how it utilizes the structure for language understanding. This part of the story can still remain unclear because of the blackbox-nature of neural nets. Nevertheless, extractive structures may still provide some rough idea of the inner workings of BT-Cell variants.

Score	**Parsed Structures**
	BT-GRC (beam size 5)
0.42	((i (did not)) (((like a) (single minute)) ((of this) film)))
0.40	(((i (did not)) ((like a) (single minute))) ((of this) film))
0.20	((i (did not)) (((like a) ((single minute) of)) (this film)))
0.40	((i (shot an)) ((elephant in) (my pajamas)))
0.21	(((i shot) (an elephant)) ((in my) pajamas))
0.19	(((i shot) (an elephant)) (in (my pajamas)))
0.19	((i shot) ((an elephant) ((in my) pajamas)))
0.40	((john saw) ((a man) (with binoculars)))
0.40	(((john saw) (a man)) (with binoculars))
0.20	((john (saw a)) ((man with) binoculars))
0.61	(((roger (dodger is)) (one (of the))) (((most compelling) (variations of)) (this theme)))
0.40	(((roger (dodger is)) ((one (of the)) (most compelling))) ((variations of) (this theme)))
	BT-GRC (beam size 2)
0.50	((i ((did not) like)) (((a single) minute) ((of this) film)))
0.50	((i (((did not) like) (a single))) ((minute of) (this film)))
0.50	((i (shot an)) ((elephant in) (my pajamas)))
0.50	((i ((shot an) elephant)) ((in my) pajamas))
0, 51	((john (saw a)) ((man with) binoculars))
0.49	(john (((saw a) man) (with binoculars)))
1.0	((roger ((dodger is) one)) ((((of the) most) (compelling variations)) ((of this) theme)))

Table 10: Parsed Structures of BT-GRC trained on MNLI. Each block represents different beams.

.

In Table 10, we show the parsed structures of some iconic sentences by BT-GRC after it is trained on MNLI. We report all beams and their corresponding scores. Note, although beam search ensures that the sequence of parsing actions for each beam is unique, different sequences of parsing action can still lead to the same structure. Thus, some beams end up being duplicates. In such cases, for the sake of more concise presentation, we collapse the duplicates into a single beam and add up their corresponding scores. This is why we can note in Table 10 that we sometimes have fewer induced structures than the beam size.

At a rough glance, we can see that the different induced structures roughly correspond to human intuitions. One interesting appeal for beam search is that it can more explicitly account for ambiguous interpretations corresponding to ambiguous structures. For example, *"i shot an elephant in my pajamas"* is ambiguous with respect to whether it is the elephant who is in the shooter's pajamas, or if it is the shooter who is in the pajamas. The induced structure (beam size 5 model in Table 10) *(((i shot) (an elephant)) ((in my) pajamas))* corresponds better to the latter interpretation whereas *((i shot) ((an elephant) ((in my) pajamas)))* corresponds better to the former interpretation (because "an elephant" is first composed with "in my pajamas").

Similar to above, *"john saw a man with binoculars"* is also ambiguous. Its interpretation is ambiguous with respect to whether it is John who is seeing through binoculars, or whether it is the man who just possesses the binoculars. Here, again, we can find (beam size 5 model in Table 10) that the induced structure *(((john saw) (a man)) (with binoculars)* corresponds better to the former interpretation whereas *((john saw) ((a man) (with binoculars)))* corresponds better to the latter. Generally, we find the score distributions to have a high entropy. A future consideration would be whether we should add an auxiliary objective to minimize entropy.

In Table 11, we show the parsed structures of the same sentences by BT-GRC+Softpath after it is trained on MNLI, and in Table 12, we show the same for BT-GRC+Gumbelpath. Most of the points above applies here for Softpath and Gumbelpath as well. Interestingly, Softpath and Gumbelpath seemed to have a relatively lower entropy distribution - that is most evident in beam size 2. We also note that Gumbelpath structures are of a similar quality despite having much lower empirical performance in MNLI.

Score	**Parsed Structures**
BT-GRC + Softpath (beam size 5)	
0.42	(((i did) (not like)) (((a single) minute) ((of this) film)))
0.20	((((i did) not) ((like a) single)) ((minute of) (this film)))
0.19	((((i did) (not like)) ((a single) minute)) ((of this) film))
0.19	(((i (did not)) ((like a) single)) ((minute of) (this film)))
0.41	(((i shot) an) ((elephant in) (my pajamas)))
0.21	(((i shot) (an elephant)) ((in my) pajamas))
0.19	((i (shot an)) ((elephant in) (my pajamas)))
0.19	((((i shot) an) (elephant in)) (my pajamas))
0.21	((john (saw a)) ((man with) binoculars))
0.20	(((john saw) (a man)) (with binoculars))
0.20	((john saw) ((a man) (with binoculars)))
0.19	((john ((saw a) man)) (with binoculars))
0.40	(((roger dodger) (is one)) ((((of the) most) (compelling variations)) ((of this) theme)))
0.21	(((roger (dodger is)) ((one of) the)) (((most compelling) variations) ((of this) theme)))
0.20	((((roger dodger) (is one)) ((of the) most)) ((compelling variations) ((of this) theme)))
0.19	((roger (dodger is)) ((((one of) the) ((most compelling) variations)) ((of this) theme)))
BT-GRC + Softpath (beam size 2)	
0.57	((i ((did not) like)) (((a single) minute) ((of this) film)))
0.43	((i ((did not) like)) (((a single) (minute of)) (this film)))
0.54	((i ((shot an) elephant)) ((in my) pajamas))
0.46	((i (shot an)) ((elephant in) (my pajamas)))
0.55	((john (saw a)) ((man with) binoculars))
0.45	((john ((saw a) man)) (with binoculars))
0.53	((roger ((dodger is) one)) ((((of the) most) (compelling variations)) ((of this) theme)))
0.47	(((roger ((dodger is) one)) ((of the) most)) ((compelling variations) ((of this) theme)))

Table 11: Parsed Structures of BT-GRC + Softpath trained on MNLI. Each block represents different beams.

.

We found the structures induced by BT-Cell variants after training on SST5 or IMDB to be more ill-formed. This may indicate that sentiment classification does not provide a strong enough signal for parsing or rather, exact induction of structures are not as necessary (Iyyer et al., 2015). We show the parsings of these models after training on IMDB and SST datasets in a text file included in the supplementary.

F EXTENDED RELATED WORKS

Initially RvNN Pollack (1990); Socher et al. (2010) was used with user-annotated tree-structured data. Some explored use of heuristic trees such as balanced trees for RvNN-like settings (Munkhdalai & Yu, 2017; Shi et al., 2018). In due time, several approaches were introduced for dynamically inducing structures from data for RvNN-style processing. This includes the greedy easy-first framework using children-reconstruction loss (Socher et al., 2011) or gumbel softmax (Choi et al., 2018), RL-based frameworks (Havrylov et al., 2019), CYK-based framework (Le & Zuidema, 2015; Maillard et al., 2019; Drozdov et al., 2019; Hu et al., 2021), shift-reduce parsing or memory-augmented or stack-augmented RNN frameworks (Grefenstette et al., 2015; Bowman et al., 2016; Yogatama et al., 2017; Maillard & Clark, 2018; Shen et al., 2019a; DuSell & Chiang, 2020; 2022), and soft-recursion-based frameworks (Chowdhury & Caragea, 2021; Zhang et al., 2021). Besides RvNNs, other approaches range from adding information-ordering biases to hidden states in RNNs (Shen et al., 2019b) or even adding additional structural or recursive constraints to Transformers (Wang et al., 2019; Nguyen et al., 2020; Fei et al., 2020; Shen et al., 2021; Csordás et al., 2022).

Score	Parsed Structures
BT-GRC + Gumbelpath (beam size 5)	
0.42	(((i (did not)) (like a)) ((single (minute of)) (this film)))
0.38	(((i did) ((not like) (a single))) ((minute (of this)) film))
0.20	(((i did) ((not like) (a single))) ((minute (of this)) film))
0.43	((i (shot an)) ((elephant in) (my pajamas)))
0.20	(((i shot) (an elephant)) ((in my) pajamas))
0.19	(((i shot) (an (elephant in))) (my pajamas))
0.18	(((i (shot an)) (elephant in)) (my pajamas))
0.39	(((john saw) (a man)) (with binoculars))
0.39	((john saw) ((a man) (with binoculars)))
0.21	((john (saw a)) ((man with) binoculars))
0.41	(((roger (dodger is)) (one (of the))) ((most (compelling variations)) ((of this) theme)))
0.21	(((roger dodger) (is one)) (((of the) (most (compelling variations))) ((of this) theme)))
0.20	(((roger dodger) ((is one) (of the))) ((most (compelling variations)) ((of this) theme)))
0.19	(((roger (dodger is)) ((one (of the)) (most (compelling variations)))) ((of this) theme))
BT-GRC + Gumbelpath (beam size 2)	
0.58	(((((i did) (not like)) (a single)) (minute (of this))) film)
0.42	(((((i did) (not like)) (a single)) (minute of)) (this film))
0.64	(((i (shot an)) (elephant (in my))) pajamas)
0.36	(((i (shot an)) (elephant in)) (my pajamas))
0.53	((john (saw a)) (man (with binoculars)))
0.47	(((john (saw a)) (man with)) binoculars)
0.51	(((((roger dodger) (is one)) (of the)) (most (compelling variations))) (of (this theme)))
0.49	((((((roger dodger) (is one)) (of the)) (most (compelling variations))) (of this)) theme)

Table 12: Parsed Structures of BT-GRC + Gumbelpath trained on MNLI. Each block represents different beams.

.

G LIMITATIONS AND FUTURE WORKS

The ideal future direction would be to extend to methods which generalize better in all aspects while maintaining computational efficiency and at the same time having a more flexible architecture for handling more free structures like non-projective trees or directed acyclic graphs and also richer classes of languages (DuSell & Chiang, 2022; Del'etang et al., 2022). Another direction to explore would be to linearize the recursion particularly taking inspiration from state space models (Gu et al., 2022) to make these models more scalable.

H HYPERPARAMETERS

For all recursive/recurrent models, we use a linear layer followed by layer normalization for initial leaf transformation before starting the recursive loop (similar to Shen et al. (2019a); Chowdhury & Caragea (2021)). Overall we use the same boilerplate classifier architecture for classification and the same boilerplate sentnece-pair siamese architecture for logical inference as Chowdhury & Caragea (2021) over our different encoders. In practice, for BT-Cell, we use a stochastic top-k through gumbel perturbation similar to Kool et al. (2019). However, we find deterministic selection to work similarly.

In terms of the optimizer, hidden size, and other hyperparameters besides dropout, we use the same ones as used by (Chowdhury & Caragea, 2021) for all models for corresponding datasets; for number of memory slots and other ordered memory specific parameters we use the same ones as used by (Shen et al., 2019a). For BSRP-Cell we use a beam size of 8 (we also tried with 5 but results were similar or slightly worse). We use a dropout rate of 0.1 for logical inference for all models (tuned on the validation set using grid search among $[0.1, 0.2, 0.3, 0.4]$ with 5 epochs per run using BalancedTreeCell for GRC-based models and GumbelTreeLSTM for LSTM based models). We use dropouts in the same places as used in (Chowdhury & Caragea, 2021). We then use the same chosen dropouts for ListOps. We tune the dropouts for SST in the same way (but with a maximum epoch of 20 per trial) on SST5 using RecurrentGRC for GRC-models, and Gumbel-Tree-LSTM for LSTM models. After tuning, for GRC-based models in SST5, we found a dropout rate of 0.4 for input/output dropout layers, and 0.2 for the dropout layer in the cell function. We found a dropout of 0.3 for LSTM-based models in SST5. and We share the hyperparameters of SST5 with SST2 and IMDB. For MNLI, we used similar settings as Chowdhury & Caragea (2021).

For NDR experiments, we use the same hyperparameters as used for ListOps by Csordás et al. (2022). The hyperparameters will also be available in code.

All codes are run in a single RTX A6000 GPU.