# Program Language Translation Using a Grammar-Driven Tree-to-Tree Model

Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro Sandoval, Rakia Segev, Eric Weiner, Robert Keller

# Background papers

- Tree-to-tree Neural Networks for Program Translation (Chen, Liu, Song), 2018
- A Syntactic Neural Model for General-Purpose Code Generation (Yin, Neubig) 2017
- Effective Approaches to Attention-based Neural Machine Translation (Luong, Pham, Manning) 2015
- Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks (Tai, Socher, Manning) 2015

# Context

- PL to PL translation
  - Infix to prefix, HLL to IR, Imperative to functional, IDE’s and tooling
- Translation seems solved but lots of special cases
  - Polyglot tools, embedded HTML/JS/CSS, Python + DSLs (Spark, Tensorflow), JSX (JS+ Html templating)
- Syntax-driven parsing is the most important “Solved unsolved problem” in CS

# Idea

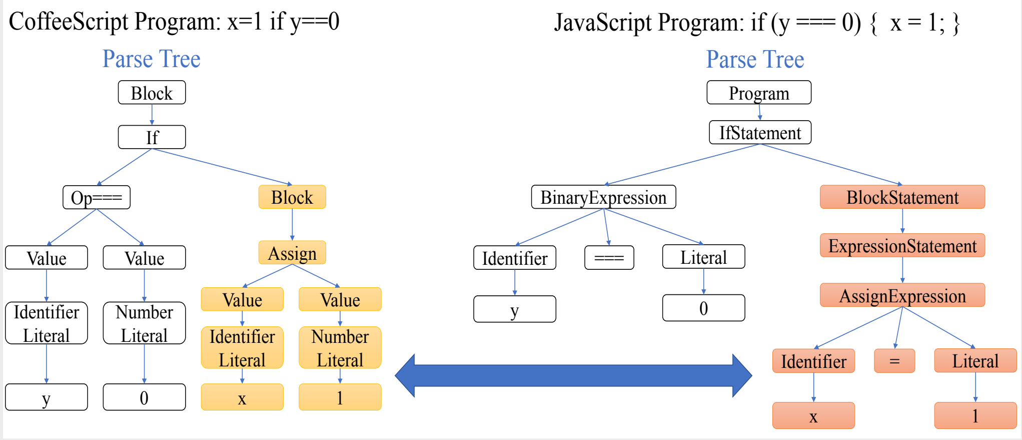
- Neural encoder-decoder architecture could be useful
- Allows for end-to-end data driven training
- Can refine on pure syntax directed translation
  - Combined translation/optimization
  - folding of redundancies
  - Stylistic conventions

# Problem

- Formal languages are not free-form
  - Things appearing at every step of generation are tightly governed by grammar rules
- Existing approaches often generate invalid outputs
- Should incorporate the strong inductive bias of syntactic validity without taking away End-to-End trainability

# Refinement

- Use tree-structured RNN's to deal with the rigid structure of formal languages
- Enforce the syntactic validity
- Should incorporate the strong inductive bias of syntactic validity without taking away End-to-End trainability



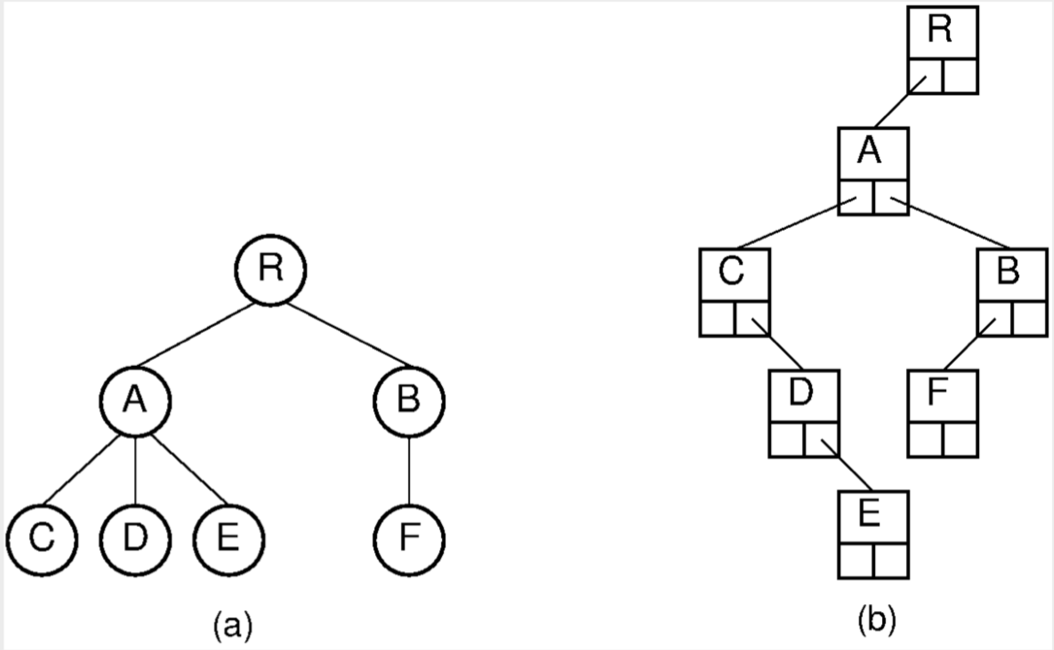
# Combining in tree-LSTM

- Tree-LSTM hidden and cell state combining rule

$$(h, c) = LSTM(([h_L : h_R], [c_L, c_R]), t_s).$$

- Binary constitution-tree LSTM cell
- Each node has too children so combining is straightforward (How?), needs binarized trees

# Tree Binarization: Left child right sibling

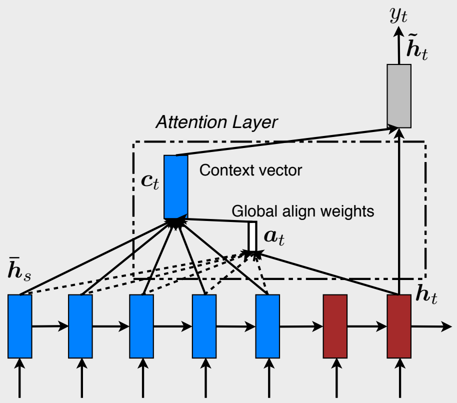



Facilitates simpler tree-LSTM cell architecture compared to N-ary tree

Only used in source tree in this work

# Decoder Architecture & attention

- At end of encoding source tree, create empty target tree and push source root node state to it and put it in a  queue
- Pop nodes from Queue and:
- Use Attention to decide which part of source tree to concentrate On.
  - Calclulate source context embedding $e_s$ by weighted sum inner product of hidden top-of-queue state with all source tree states (soft attention)
  - Calculate target context embedding vector from source context and top-of-queue hidden state
  - Calculate target token by softmax




Attention map: $w_i \propto \exp(h_i^T W_0 h_5')$

Source embedding: $e_s = \sum_{i=1}^{17} w_i h_i = [h_1; \ldots; h_{17}]w$

Combined embedding: $e_t = \tanh(W_1 e_s + W_2 h_5{}')$

Predicting the node: node $=$ argmax $\mathbf{softmax}(W\mathrm{e_t})$

$$e_t = \mathbf{tanh}(W_1[e_s; h])$$

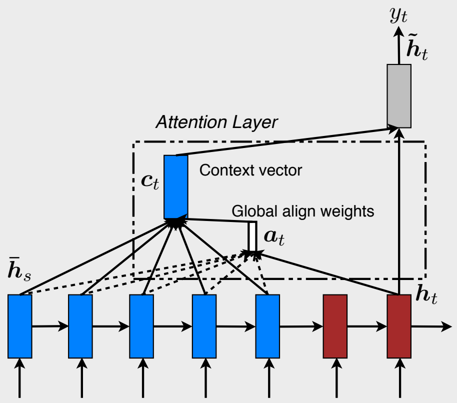
$$t_t = \mathbf{argmax\ softmax}(We_t)$$

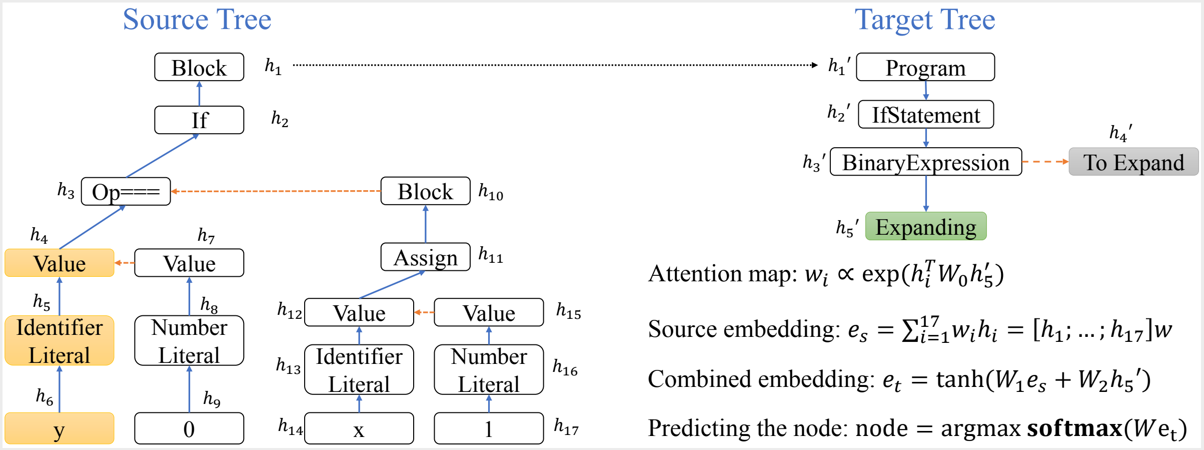
# Decoding Algorithm

- Prior Art:
  - Target tree is binarized as well
  - If the selected token is not <EOS>, use two parameterized LSTM cells $LSTM_L$ and $LSTM_R$ To generate child node cell and hidden states and push onto the queue, for <EOS> do nothing

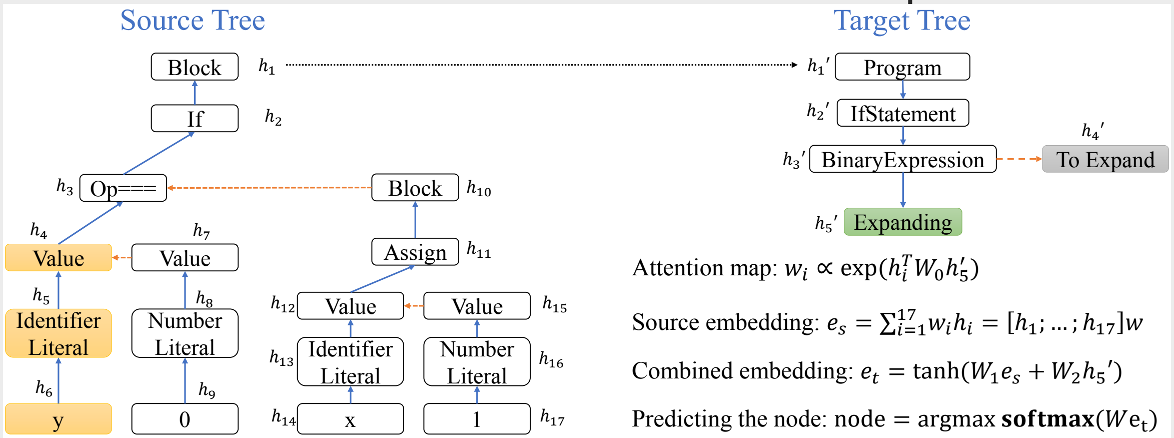
$$(h_i, c_i) = LSTM_i((h, c), [Bt_t; e_t])$$

- Dequeue nodes and repeat until output queue is empty



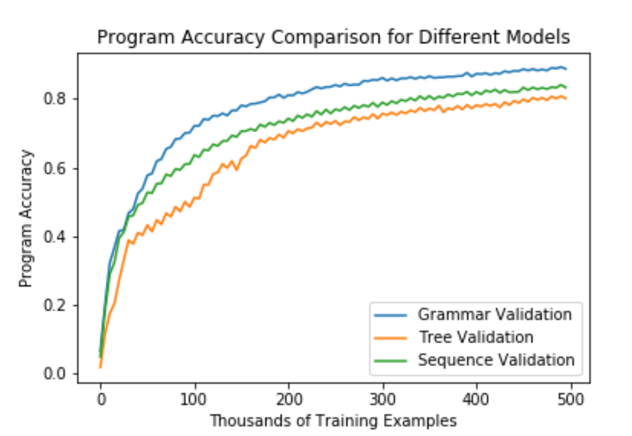
# Decoding Algorithm (This work)




- This work:
  - Target tree is kept N-ary, as enforcing grammar rules on binarized tree is not easy
  - For each source token (Terminal or non-terminal) we know its arity in target grammar, N
  - For each of the N child positions, consider separately the set of tokens that can appear there as a category set (e.g. Literal, Variable, Factor/product for +/- production)
  - Factor out categories across all grammar productions
  - Learn separate weight parameters for each category $W_k$ to generate expanded node tokens and hidden states
  - Use softmax to generate child token of the right category
  - Feed chosen child token to LSTM cell to generate the hidden and cell state of the LSTM

# Results

- 100K synthetic programs in FOR (toy imperative lang) to corresponding programs in Lambda (toy functional lang)
- Comparison with Chen et al. and naïve seq-2-seq
- Don’t reproduce the exact results of Cheng et al. due to dataset differences
- Claim their results show less variation due to random seeds




| Model | Mean Accuracy | $\sigma$ Accuracy |
|---|---|---|
| **Grammar Tree2Tree** | 88.82% | 0.64% |
| Reimplemented Tree2Tree | 80.69% | 7.02% |
| Reimplemented Tree2Seq | 83.59% | 3.95% |
| Chen et al. Tree2Tree (Easy) | 99.76% | N/A |
| Chen et al. Tree2Seq (Easy) | 98.36% | N/A |
| Chen et al. Tree2Tree (Hard) | 97.50% | N/A |
| Chen et al. Tree2Seq (Hard) | 87.84% | N/A |
| **(Long) Grammar Tree2Tree** | 93.70% | N/A |
| (Long) Reimplemented Tree2Tree | 91.89% | N/A |
| (Long) Reimplemented Tree2Seq | 90.60% | N/A |

| Metric | For | Lambda |
|---|---|---|
| Total program count | 100k | 100k |
| Average program length | 22 | 56 |
| Minimum program length | 5 | 13 |
| Maximum program length | 104 | 299 |
| Number of tokens in language | 32 | 33 |