Supplementary Materials

A Encoder LSTM Equations

Suppose the input natural language description x consists of n words $\{w_i\}_{i=1}^n$. Let $\mathbf{w}_i$ denote the embedding of w_i. We use two LSTMs to process x in forward and backward order, and get the sequence of hidden states $\{\vec{\mathbf{h}}_i\}_{i=1}^n$ and $\{\overleftarrow{\mathbf{h}}_i\}_{i=1}^n$ in the two directions:

$$\vec{\mathbf{h}}_i = f^{\rightarrow}_{\text{LSTM}}(\mathbf{w}_i, \vec{\mathbf{h}}_{i-1})$$
$$\overleftarrow{\mathbf{h}}_i = f^{\leftarrow}_{\text{LSTM}}(\mathbf{w}_i, \overleftarrow{\mathbf{h}}_{i+1}),$$

where $f^{\rightarrow}_{\text{LSTM}}$ and $f^{\leftarrow}_{\text{LSTM}}$ are standard LSTM update functions. The representation of the i-th word, $\mathbf{h}_i$, is given by concatenating $\vec{\mathbf{h}}_i$ and $\overleftarrow{\mathbf{h}}_i$.

B Inference Algorithm

Given an NL description, we approximate the best AST $\hat{y}$ in Eq. 1 using beam search. The inference procedure is listed in Algorithm 1.

We maintain a beam of size K. The beam is initialized with one hypothesis AST with a single root node (line 2). At each time step, the decoder enumerates over all hypotheses in the beam. For each hypothesis AST, we first find its frontier node n_{f_t} (line 6). If n_{f_t} is a non-terminal node, we collect all syntax rules r with n_{f_t} as the head node to the actions set (line 10). If n_{f_t} is a variable terminal node, we add all terminal tokens in the vocabulary and the input description as candidate actions (line 13). We apply each candidate action on the current hypothesis AST to generate a new hypothesis (line 15). We then rank all newly generated hypotheses and keep the top-K scored ones in the beam. A complete hypothesis AST is generated when it has no frontier node. We then convert the top-scored complete AST into the surface code (lines 18-19).

We remark that our inference algorithm can be implemented efficiently by expanding multiple hypotheses (lines 5-16) simultaneously using mini-batching on GPU.

C Dataset Preprocessing

Infrequent Words We replace word types whose frequency is lower than d with a special `<unk>` token ($d = 5$ for DJANGO, 3 for HS and IFTTT).

Canonicalization We perform simple canonicalization for the DJANGO dataset: (1) We observe that input descriptions often come with quoted string literals (e.g., *verbose_name is a string 'cache entry'*). We therefore replace quoted strings with indexed placeholders using regular expression. After decoding, we run a post-processing step to replace all placeholders with their actual values. (2) For descriptions with cascading variable reference (e.g., *call method self.makekey*), we append after the whole variable name with tokens separated by '.' (e.g., append *self* and *makekey* after *self.makekey*). This gives the pointer network flexibility to copy either partial or whole variable names.

Generate Oracle Action Sequence To train our model, we generate the gold-standard action sequence from reference code. For IFTTT, we simply parse the officially provided ASTs into sequences of APPLYRULE actions. For HS and DJANGO, we first convert the Python code into ASTs using the standard `ast` module. Values inside variable terminal nodes are tokenized by space and camel case (e.g., *ClassName* is tokenized to *Class* and *Name*). We then traverse the AST in pre-order to generate the reference action sequence according to the grammar model.

D Additional Decoding Examples

We provide extra decoding examples from the DJANGO and HS datasets, listed in Table 6 and Table 7, respectively. The model heavily relies on the pointer network to copy variable names and constants from input descriptions. We find the source of errors in DJANGO is more diverse, with most incorrect examples resulting from missing arguments and incorrect words copied by the pointer network. Errors in HS are mostly due to partially or incorrectly implemented effects. Also note that the first example in Table 6 is semantically correct, although it was considered incorrect under our exact-match metric. This suggests more advanced evaluation metric that takes into account the execution results in future studies.

Algorithm 1: Inference Algorithm

Input : NL description x
Output: code snippet c

call Encoder to encode x
$Q = \{y_0(root)\}$ ▷ *Initialize a beam of size K*
for *time step t* **do**
  $Q' = \emptyset$
  foreach *hypothesis $y_t \in Q$* **do**
    $n_{f_t} = \text{FrontierNode}(y_t)$
    $\mathcal{A} = \emptyset$ ▷ *Initialize the set of candidate actions*
    if *n_{f_t} is non-terminal* **then**
      foreach *production rule r with n_{f_t} as the head node* **do**
        $\mathcal{A} = \mathcal{A} \cup \{\text{APPLYRULE}[r]\}$ ▷ APPLYRULE *actions for non-terminal nodes*
    else
      foreach *terminal token v* **do**
        $\mathcal{A} = \mathcal{A} \cup \{\text{GENTOKEN}[v]\}$ ▷ GENTOKEN *actions for variable terminal nodes*
    foreach *action $a_t \in \mathcal{A}$* **do**
      $y'_t = \text{ApplyAction}(y_t, a_t)$
      $Q' = Q' \cup \{y'_t\}$
  $Q =$ top-K scored hypotheses in Q'
$\hat{y} =$ top-scored complete hypothesis AST
convert $\hat{y}$ to surface code c
return c

input *for every i in range of integers from 0 to length of result, not included* **pred.** `for i in range(0, len(result)):` ✓	**ref.** `for i in range(len(result)):`
input *call the function blankout with 2 arguments: t.contents and 'B', write the result to out.* **pred.** `out.write(blankout(t.contents, 'B'))` ✓	**ref.** `out.write(blankout(t.contents, 'B'))`
pred. `code_list.append(foreground[v])` ✓	**ref.** `code_list.append(foreground[v])`
input *zip elements of inner_result and inner_args into a list of tuples, for every i_item and i_args in the result* **pred.** `for i_item, i_args in zip(inner_result, inner_args):` ✓	**ref.** `for i_item, i_args in zip(inner_result, inner_args):`
input *activate is a lambda function which returns None for any argument x.* **pred.** `activate = lambda x: None` ✓	**ref.** `activate = lambda x: None`
input *if elt is an instance of Choice or NonCapture classes* **pred.** `if isinstance(elt, Choice):` ✗	**ref.** `if isinstance(elt, (Choice, NonCapture)):`
input *get translation_function attribute of the object t, call the result with an argument eol_message, substitute the result for result.* **pred.** `translation_function = getattr(t, translation_function)` ✗	**ref.** `result = getattr(t, translation_function)(eol_message)`
input *for every s in strings, call the function force_text with an argument s, join the results in a string, return the result.* **pred.** `return ''.join(force_text(s))` ✗	**ref.** `return ''.join(force_text(s) for s in strings)`
input *for every p in parts without the first element* **pred.** `for p in p[1:]:` ✗	**ref.** `for p in parts[1:]:`
input *call the function get_language, split the result by '-', substitute the first element of the result for base_lang.* **pred.** `base_lang = get_language().split()[0]` ✗	**ref.** `base_lang = get_language().split('-')[0]`

Table 6: Predicted examples from DJANGO dataset. Copied contents (copy probability > 0.9) are highlighted

input *<name> Burly Rockjaw Trogg </name> <cost> 5 </cost> <attack> 3 </attack> <defense> 5 </defense> <desc> Whenever your opponent casts a spell, gain 2 Attack. </desc> <rarity> Common </rarity> ...*

pred.
```
class BurlyRockjawTrogg(MinionCard):
    def __init__(self):
        super().__init__('Burly Rockjaw Trogg', 4, CHARACTER_CLASS.ALL, CARD_RARITY.COMMON)
    def create_minion(self, player):
        return Minion(3, 5, effects=[Effect(SpellCast(player=EnemyPlayer()),
            ActionTag(Give(ChangeAttack(2)), SelfSelector()))]) ✓
```

input *<name> Maexxna </name> <cost> 6 </cost> <attack> 2 </attack> <defense> 8 </defense> <desc> Destroy any minion damaged by this minion. </desc> <rarity> Legendary </rarity> ...*

pred.
```
class Maexxna(MinionCard):
    def __init__(self):
        super().__init__('Maexxna', 6, CHARACTER_CLASS.ALL, CARD_RARITY.LEGENDARY,
          minion_type=MINION_TYPE.BEAST)
    def create_minion(self, player):
        return Minion(2, 8, effects=[Effect(DidDamage(), ActionTag(Kill(),
            TargetSelector(IsMinion()))))]) ✓
```

input *<name> Hellfire </name> <cost> 4 </cost> <attack> -1 </attack> <defense> -1 </defense> <desc> Deal 3 damage to ALL characters. </desc> <rarity> Free </rarity> ...*

pred.
```
class Hellfire(SpellCard):
    def __init__(self):
        super().__init__('Hellfire', 4, CHARACTER_CLASS.WARLOCK, CARD_RARITY.FREE)

    def use(self, player, game):
        super().use(player, game)
        for minion in copy.copy(game.other_player.minions):
            minion.damage(player.effective_spell_damage(3), self) ✗
```

ref.
```
class Hellfire(SpellCard):
    def __init__(self):
        super().__init__('Hellfire', 4, CHARACTER_CLASS.WARLOCK, CARD_RARITY.FREE)

    def use(self, player, game):
        super().use(player, game)
        targets = copy.copy(game.other_player.minions)
        targets.extend(game.current_player.minions)
        targets.append(game.other_player.hero)
        targets.append(game.current_player.hero)
        for minion in targets:
            minion.damage(player.effective_spell_damage(3), self)
```

reason Partially implemented effect: only deal 3 damage to opponent's characters

input *<name> Darkscale Healer </name> <cost> 5 </cost> <attack> 4 </attack> <defense> 5 </defense> <desc> Battlecry: Restore 2 Health to all friendly characters. </desc> <rarity> Common </rarity> ...*

pred.
```
class DarkscaleHealer(MinionCard):
    def __init__(self):
        super().__init__('Darkscale Healer', 5, CHARACTER_CLASS.ALL,
            CARD_RARITY.COMMON, battlecry=Battlecry(Damage(2),
            CharacterSelector(players=BothPlayer(), picker=UserPicker())))

    def create_minion(self, player):
        return Minion(4, 5) ✗
```

ref.
```
class DarkscaleHealer(MinionCard):
    def __init__(self):
        super().__init__('Darkscale Healer', 5, CHARACTER_CLASS.ALL,
            CARD_RARITY.COMMON, battlecry=Battlecry(Heal(2), CharacterSelector()))

    def create_minion(self, player):
        return Minion(4, 5)
```

reason Incorrect effect: damage 2 health instead of restoring. Cast effect to all players instead of friendly players only.

Table 7: Predicted card examples from HS dataset. Copied contents (copy probability > 0.9) are highlighted.