Supplementary Material for *RobustFill: Neural Program Learning under Noisy I/O*

A. Attention Formulas

The formula $c_i = \mathrm{Attention}(h_{i-1}, x_i, S)$ is as follows:

$$
\begin{aligned}
t_i &= \tanh(W[h_{i-1}; x_i]) \\
d_{ij} &= s_j \cdot t_i \\
\alpha_{ij} &= \frac{e^{d_{ij}}}{\sum_k e^{d_{ik}}} \\
c_i &= \sum_j \alpha_{ij} s_j
\end{aligned}
$$

Where i is the current timestep, h_{i_1} is the previous hidden state, x_i is the current input, $S = s_1, ..., s_N$ are the vectors being attended to, and W is a learned parameter matrix. The interpolated context vector c_i is concatenated into the input and fed into the LSTM. In the case of double attention, the output of the first attention mechanism C_i^A is concatenated to the input of the second attention, i.e.:

$$
t_i^B = \tanh(W[h_{i-1}; x_i; c_i^A])
$$

where the remaining steps are identical.

B. DSL Extended Description

Section 3.2 of the paper provides the grammar of our domain specific language, which both defines the space of possible programs, and allows us to easily sample programs. The formal semantics of this language are defined below in Figure 1. The program takes as input a string v and produces a string as output (result of `Concat` operator).

As an implementational detail, we note that after sampling a program from the grammar, we flatten calls to nesting functions (as defined in Figure 2 of the paper) into a single token. For example, the function `GetToken(t, i)` would be tokenized as a single token $\texttt{GetToken}_{\texttt{t,i}}$ rather than 3 separate tokens. This is possible because for nesting functions, the size of the total parameter space is small. For all other functions, the parameter space is too large for us to flatten function calls without dramatically increasing the vocabulary size, so we treat parameters as separate tokens.

$$
\begin{array}{rcl}
[\![\texttt{Concat}(e_1, e_2, e_3, ...)]\!]_v & = & \texttt{Concat}([\![e_1]\!]_v, [\![e_2]\!]_v, [\![e_3]\!]_v, ...) \\
[\![n_1(n_2)]\!]_v & = & [\![n_1]\!]_{v_1}, \text{ where } v_1 = [\![n_2]\!]_v \\
[\![n(f)]\!]_v & = & [\![n]\!]_{v_1}, \text{ where } v_1 = [\![f]\!]_v \\
[\![\texttt{ConstStr}(c)]\!]_v & = & c \\
[\![\texttt{SubStr}(k_1, k_2)]\!]_v & = & v[p_1..p_2], \text{ where} \\
 & & p_1 = k_1 > 0 \text{ ? } k_1 \ : \ \texttt{len(v)} + k_1 \\
 & & p_2 = k_2 > 0 \text{ ? } k_2 \ : \ \texttt{len(v)} + k_2 \\
[\![\texttt{GetSpan}(r_1, i_1, y_1, r_2, i_2, y_2)]\!]_v & = & v[p_1..p_2] \text{ ,where} \\
 & & p_1 = y_1 \text{(Start or End) of } |i_1|^{\texttt{th}} \text{ match of } r_1 \text{ in } v \text{ from beginning (end if } i_i < 0) \\
 & & p_2 = y_2 \text{(Start or End) of } |i_2|^{\texttt{th}} \text{ match of } r_2 \text{ in } v \text{ from beginning (end if } i_2 < 0) \\
[\![\texttt{GetToken(t,i)}]\!]_{\texttt{v}} & = & |i|^{\texttt{th}} \text{ match of } t \text{ in } v \text{ from beginning (end if } i < 0) \\
[\![\texttt{GetUpto(r)}]\!]_{\texttt{v}} & = & v[0..i], \text{ where } i \text{ is the index of end of first match of } r \text{ in } v \text{ from beginning} \\
[\![\texttt{GetFrom(r)}]\!]_{\texttt{v}} & = & v[j..len(v)], \text{ where j is the end of last match of } r \text{ in } v \text{ from end} \\
[\![\texttt{GetFirst(t,i)}]\!]_{\texttt{v}} & = & \texttt{Concat}(s_1, \cdots, s_i), \text{ where } s_j \text{ denotes the } j^{\texttt{th}} \text{ match of } t \text{ in } v \\
[\![\texttt{GetAll(t)}]\!]_{\texttt{v}} & = & \texttt{Concat}(s_1, \cdots, s_m), \text{ where } s_i \text{ denotes the } i^{\texttt{th}} \text{ match of } t \text{ in } v \text{ and } m \text{ denotes the total matches} \\
[\![\texttt{ToCase(s)}]\!]_{\texttt{v}} & = & \texttt{ToCase}(s, v) \\
[\![\texttt{Trim()}]\!]_{\texttt{v}} & = & \texttt{Trim}(v) \\
[\![\texttt{Replace}(\delta_1, \delta_2)]\!]_{\texttt{v}} & = & \texttt{Replace}(v, \delta_1, \delta_2)
\end{array}
$$

Figure 1. The semantics of the DSL for string transformations.

C. Synthetic Training Data Generation

Since there are only a few hundred real-world FlashFill benchmarks, we use synthetically generated training data to train our neural models. The key idea in data generation is to uniformly sample programs from the DSL, and then for each sampled program, generate a set of input-output examples that are consistent with it. We now describe the key steps in the data generation process in more detail.

First, programs are sampled randomly from the DSL. We treat the DSL as a probabilistic context free grammar (PCFG) where the probability of expanding to any child node is uniformly random. Even though the top-level concat operator can take an arbitrary number of expressions e, in practice, we limit it to have at most k expressions, where k is randomly sampled from 1 to 10.

Next, the input strings are sampled from the space of all random ASCII strings with lengths between 1 and 100, using some simple heuristics that are extracted from the sampled programs preconditions. For example, if the program contained GetToken(Word, 2) and GetFrom(Space, 4) as sub-expressions, then we would first generate 2 words and 4 spaces, then shuffle these and add other random ASCII characters. In this case, words are defined as random ASCII strings that match the particular regular expression of [A-Za-z]{1,10}. Finally, to generate the output strings, we execute the program on the input strings.

However, the extracted heuristics do not always encapsulate all preconditions exactly, as there are some edge cases that may prevent successful execution. If the program could not be executed on an input string (e.g., say one expression in our sampled program is SubStr(GetToken(word, 2), 1, 10)), but the 2nd word isnt 10 characters long), we simply reject the input string and re-sample until we find one that executes successfully. We find that in practice, the pre-conditions are usually sufficient conditions for efficient generation of viable input strings.

D. Synthetic Evaluation Details

Results on synthetically generated examples are largely omitted from the paper since, in a vacuum, the synthetic dataset can be made arbitrarily easy or difficult via different generation procedures, making summary statistics difficult to interpret. We instead report results on an external real-world dataset to verify that the model has learned function semantics which are at least as expressive as programs observed in real data.

Nevertheless, we include additional details about our experiments on synthetically generated programs for readers interested in the details of our approach. As described in the paper, programs were randomly generated from the DSL by first determining a program length up to a maximum of 10 *expressions*, and then independently sampling each expression. We used a simple set of heuristics to restrict potential inputs to strings which will produce non-empty outputs (e.g. any program which references the third occurrence of a number will cause us to sample strings containing at least three numbers). We rejected any degenerate samples e.g. those resulting in empty outputs, or outputs longer than 100 characters.

Figure 4 shows several random synthetically generated samples.

Figure 2 shows the accuracy of each model on the synthetically generated validation set. Model accuracy on the synthetic validation set is generally consistent with accuracy on the FlashFill dataset, with stronger models on the synthetic dataset also demonstrating stronger performance on the real-world data.

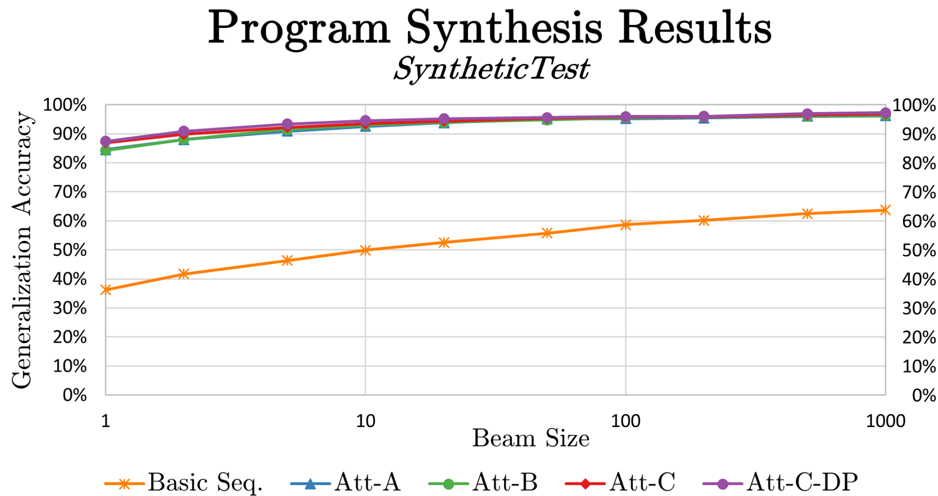

Figure 2. Generalization accuracy for different models on the synthetic validation set

E. Examples of Synthesized Programs

Figure 5 shows several randomly sampled (anonymized) examples from the FlashFill test set, along with their predicted programs outputted by the synthesis model.

Figure 6 shows several examples which were hand-selected to demonstrate interesting limitations of the model. In the case of the first example, the task is to reformat international telephone numbers. Here, the task is underconstrained given the observed input-output examples, because there are many different programs which are consistent with the observed examples. Note that to extract the first two digits, there are many other possible functions which would produce the correct output in the observed examples, some of which would generalize and some which would not: for exampling, getting the second and third characters, getting the first two digits, or getting the first number. In this case, the predicted program extracts the country code by taking the first two digits, a strategy which fails to generalize to examples with different country codes. The third example demonstrates a difficulty of using real world data. Because examples can come from a variety of sources, they may be irregularly formatted. In this case, although the program is consistent with the observed examples, it does not generalize when the second space in the address is removed. In the final example, the synthesis model completely fails, and none of the 100 highest scoring programs from the model were consistent with the observed output examples. The selected program is the closest program scored by character edit distance.

F. Induction Network Architecture

The network architecture used in the program induction setting is described in Section 6.1 of the paper. The network structure is a modification of synthesis Attention-A, using double attention to jointly attend to I^x and O_j, and an additional LSTM to encode I^x. We include a complete diagram below in Figure 3.

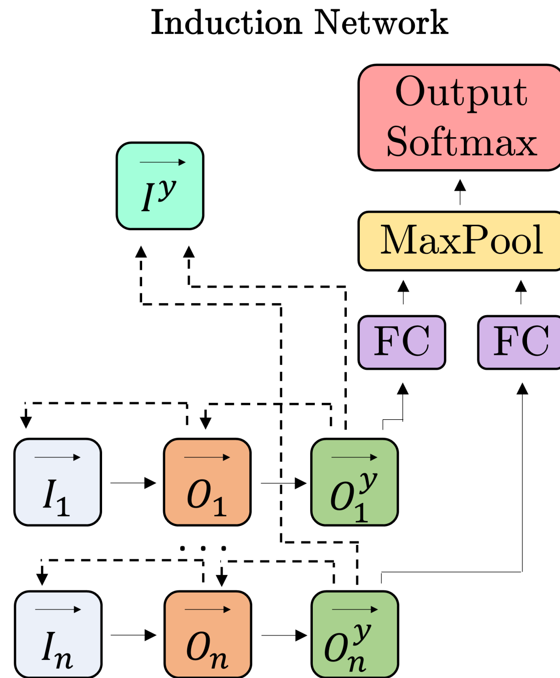

Figure 3. The network architecture used for program induction. A dotted line from x to y means that x attends to y.

<table>
<tr><th colspan="2">Reference program: GetToken_Alphanum_3 | GetFrom_Colon | GetFirst_Char_4</th></tr>
<tr><td>Ud 9:25,JV3 Obb</td><td>2525,JV3 ObbUd92</td></tr>
<tr><td>zLny xmHg 8:43 A44q</td><td>843 A44qzLny</td></tr>
<tr><td>A6 g45P 10:63 Jf</td><td>1063 JfA6g4</td></tr>
<tr><td>cuL.zF.dDX,12:31</td><td>dDX31cuLz</td></tr>
<tr><td>ZiG OE bj3u 7:11</td><td>bj3u11ZiGO</td></tr>
</table>

<table>
<tr><th colspan="2">Reference program: Get_Word_-1(GetSpan(Word, 1, Start, ‘(’, 5, Start)) | GetToken_Number_-5 | GetAll_Proper | SubStr(-24, -14) | GetToken_Alphanum_-2 | EOS</th></tr>
<tr><td>4 Kw ( )SrK (11 (3 CHA xVf)4)8 Qagimg) ( )(vs</td><td>Qagimg4Kw Sr Vf QagimgVf)4)8 QaQagimg</td></tr>
<tr><td>iY))hspA.5 ( )8,ZsLL (nZk.6 (E4w)2(Hpprsqr)2(Z</td><td>Hpgjprsqr8Zs Zk Hpprsqrk.6 (E4w)22</td></tr>
<tr><td>Cqg)) ((1005 (( )VCE hz) (10 Hadj)zg Tqwpaxft-7 5 6</td><td>hz10005Cqg Hadj Tqwpaxft Hadj)zg T5</td></tr>
<tr><td>JvY) (Ihitux)) ((6 SFl (7 XLTD sfs))11,lU7 (6 9</td><td>lU7Jv Ihitux Frl XLTD sfs)6</td></tr>
<tr><td>NjtT(D7QV (4 (yPuY)8.sa ( ))6 aX 4)DXR (@6) Ztje</td><td>DXR4Njt Pu Ztje)6 aX 4)DX6</td></tr>
</table>

<table>
<tr><th colspan="2">Reference program: GetToken_AllCaps_-2(GetSpan(AllCaps, 1, Start, AllCaps, 5, Start)) | EOS</th></tr>
<tr><td>YDXJZ @ZYUD Wc-YKT GTIL BNX</td><td>W</td></tr>
<tr><td>JUGRB.MPKA.MTHV,tEczT-GZJ.MFT</td><td>MTHV</td></tr>
<tr><td>VXO.OMQDK.JC-OAR,HZGH-DJKC</td><td>JC</td></tr>
<tr><td>HCUD-WDOC,RTTRQ-KVETK-whx-DIKDI</td><td>RTTRQ</td></tr>
<tr><td>JFNB.Avj,ODZBT-XHV,KYB @,RHVVW</td><td>ODZBT</td></tr>
</table>

<table>
<tr><th colspan="2">Reference program: SubStr(-20, -8) | GetToken_AllCaps_-3 | SubStr(11, 19) | GetToken_Alphanum_-5 | EOS</th></tr>
<tr><td>DvD 6X xkd6 OZQIN ZZUK,nCF aQR IOHR</td><td>IN ZZUK,nCF aCFv OZQIN ZOZQIN</td></tr>
<tr><td>BHP-euSZ,yy,44-CRCUC,ONFZA.mgOJ.Hwm</td><td>CRCUC,ONFZA.mONFZAy,44-CRCU44</td></tr>
<tr><td>NGM-8nay,xrL.GmOc.PFLH,CMFEX-JPFA,iIcj,329</td><td>,CMFEX-JPFA,iCMFEXrL.GmOc.PPFLH</td></tr>
<tr><td>hU TQFLD Lycb NCPYJ oo FS TUM l6F</td><td>NCPSYJ oo FS FScb NCPYJ NCPYJ</td></tr>
<tr><td>OHHS NNDQ XKQRN KDL 8Ucj dUqh Cpk Kafj</td><td>L 8Ucj dUqh CUXKQRN KDLKDL</td></tr>
</table>

Figure 4. Randomly sampled programs and corresponding input-output examples, drawn from training data. Multi-line examples are all broken into lines on spaces.

Model prediction: GetSpan(‘[’, 1, Start, Number, 1, End) | Const(]) | EOS		
[CPT-101 [CPT-101 [CPT-11] [CPT-1011]	[CPT-101] [CPT-101] [CPT-11] [CPT-1011]	[CPT-101] [CPT-101] [CPT-11] [CPT-1011]
[CPT-1011 [CPT-1012 [CPT-101] [CPT-111] [CPT-1011] [CPT-101]	[CPT-1011] [CPT-1012] [CPT-101] [CPT-111] [CPT-1011] [CPT-101]	[CPT-1011] [CPT-1012] [CPT-101] [CPT-111] [CPT-1011] [CPT-101]

Model prediction: Replace_Space_Comma(GetSpan(Proper, 1, Start, Proper, 4, End) | Const(.) | GetToken_Proper_-1 | EOS		
Jacob Ethan James Alexander Michael	Jacob,Ethan,James,Alexander.- Michael	Jacob,Ethan,James,Alexander.- Michael
Elijah Daniel Aiden Matthew Lucas	Elijah,Daniel,Aiden,Matthew.- Lucas	Elijah,Daniel,Aiden,Matthew.- Lucas
Jackson Oliver Jayden Chris Kevin	Jackson,Oliver,Jayden,Chris.- Kevin	Jackson,Oliver,Jayden,Chris.- Kevin
Earth Fire Wind Water Sun	Earth,Fire,Wind,Water.Sun	Earth,Fire,Wind,Water.Sun
Tom Mickey Minnie Donald Daffy	Tom,Mickey,Minnie,Donald.Daffy	Tom,Mickey,Minnie,Donald.Daffy
Jacob Mickey Minnie Donald Daffy	Jacob,Mickey,Minnie,Donald.- Daffy	Jacob,Mickey,Minnie,Donald.- Daffy
Gabriel Ethan James Alexander Michael	Gabriel,Ethan,James,Alexander- .Michael	Gabriel,Ethan,James,Alexander.- Michael
Rahul Daniel Aiden Matthew Lucas	Rahul,Daniel,Aiden,Matthew.- Lucas	Rahul,Daniel,Aiden,Matthew.- Lucas
Steph Oliver Jayden Chris Kevin	Steph,Oliver,Jayden,Chris.Kevin	Steph,Oliver,Jayden,Chris.Kevin
Pluto Fire Wind Water Sun	Pluto,Fire,Wind,Water.Sun	Pluto,Fire,Wind,Water.Sun

Model prediction: GetAll_Proper | EOS		
Emma Anders Olivia Berglun Madison Ashworth Ava Truillo	Emma Anders Olivia Berglun Madison Ashworth Ava Truillo	Emma Anders Olivia Berglun Madison Ashworth Ava Truillo
Isabella Mia Emma Stevens Chris Charles Liam Lewis Abigail Jones	Isabella Mia Emma Stevens Chris Charles Liam Lewis Abigail Jones	Isabella Mia Emma Stevens Chris Charles Liam Lewis Abigail Jones

Figure 5. Random samples from the FlashFill test set. The first two columns are `InStr` and `OutStr` respectively, and the third column is the execution result of the predicted program. Example strings which do not fit on a single line are broken on spaces, or hyphenated when necessary. All line-ending hyphens are inserted for readability, and are not part of the example.

Model prediction: GetToken_Proper_1 | Const(.) | GetToken_Char_1(GetToken_Proper_-1) | Const(@) | EOS		
Mason Smith	Mason.S@	Mason.S@
Lucas Janckle	Lucas.J@	Lucas.J@
Emily Jacobnette	Emily.B@	Emily.B@
Charlotte Ford	Charlotte.F@	Charlotte.F@
Harper Underwood	Harper.U@	Harper.U@
Emma Stevens	Emma.S@	Emma.S@
Chris Charles	Chris.C@	Chris.C@
Liam Lewis	Liam.L@	Liam.L@
Olivia Berglun	Olivia.B@	Olivia.B@
Abigail Jones	Abigail.J@	Abigail.J@

Figure 5. Random samples from the FlashFill test set. The first two columns are `InStr` and `OutStr` respectively, and the third column is the execution result of the predicted program. Example strings which do not fit on a single line are broken on spaces, or hyphenated when necessary. All line-ending hyphens are inserted for readability, and are not part of the example.

Model prediction: GetFirst_Digit_2 | Const(.) | GetToken_Number_2 | Const(.) | GetToken_Number_3 | Const(.) | GetToken_Alpha_-1 | EOS		
+32-2-704-33	32.2.704.33	32.2.704.33
+44-118-909-3574	44.118.909.3574	44.118.909.3574
+90-212-326 5264	90.212.326.5264	90.212.326.5264
+44 118 909 3843	44.118.909.3843	44.118.909.3843
+386 1 5800 839	386.1.5800.839	38.1.5800.839
+1 617 225 2121	1.617.225.2121	16.617.225.2121
+91-2-704-33	91.2.704.33	91.2.704.33
+44-101-909-3574	44.101.909.3574	44.101.909.3574
+90-212-326 2586	90.212.326.2586	90.212.326.2586
+44 118 212 3843	44.118.212.3843	44.118.212.3843

Model prediction: GetFirst_Char_1 | Const(.) | GetFirst_Char_1(GetToken_Proper_4) | Const(.) | EOS		
Milk 4, Yoghurt 12, Juice 2 Lassi 5	M.L.	M.L.
Alpha 10 Beta 20 Charlie 40 60 Epsilon	A.E.	A.E.
Sumit 7 Rico 12 Wolfram 15 Rick 19	S.R.	S.R.
Us 38 China 35 Russia 27 India 1	U.I.	U.I.
10 Apple 2 Oranges 13 Bananas 40 Pears	A.P.	1.P.
10 Bpple 2 Oranges 13 Bananas 40 Pears	B.P.	1.P.
Milk 4, Yoghurt 12, Juice 2 Massi 5	M.M.	M.M.
Alpha 10 Beta 20 Charlie 40 60 Delta	A.D.	A.D.
Parul 7 Rico 12 Wolfram 15 Rick 19	P.R.	P.R.
Us 38 China 35 Russia 27 America 1	U.A.	U.A.

Model prediction: Replace_Space_Dash(GetSpan(AlphaNum, 1, Start, Proper, 1, End)) | EOS		
212 2nd Avenue	212-2nd-Avenue	212-2nd-Avenue
124 3rd Avenue	124-3rd-Avenue	124-3rd-Avenue
123 4th Avenue	123-4th-Avenue	123-4th-Avenue
999 5th Avenue	999-5th-Avenue	999-5th-Avenue
123 1st Avenue	123-1st-Avenue	123-1st-Avenue
223 1stAvenue	223-1st-Avenue	223-1stAvenue
112 2nd Avenue	112-2nd-Avenue	112-2nd-Avenue
224 3rd Avenue	224-3rd-Avenue	224-3rd-Avenue
123 5th Avenue	123-5th-Avenue	123-5th-Avenue
99 5th Avenue	99-5th-Avenue	99-5th-Avenue

Figure 6. Selected samples of incorrect model predictions on the Flashfill test set. These include both inconsistent programs, and consistent programs which failed to generalize.

Model prediction: GetToken_Word_1 | Const(-) | GetToken_Proper_1(GetSpan(';', -5, Start, '#', 5, Start)) | GetUpto_Comma Replace_Space_Dash | GetToken_Word_1(GetSpan(Proper, 4, End, '$', 5, End)) \| GetToken_Number_-5 \| GetSpan('#', 5, End, '$', 5, Start) | EOS		
28;#DSI;#139;#ApplicationVirtualization;#148;#BPOS;#138;#Microsoft PowerPoint	DSI-ApplicationVirtualization-BPOS-Microsoft PowerPoint	DSI-Application
102;#Excel;#14;#Meetings;#55;#OneNote;#155;#Word	Excel-Meetings-OneNote-Word	Excel-Meetings
19;#SP Workflow Solutions;#102;#Excel;#194;#Excel Services;#46;#BI	SP Workflow Solutions-Excel-Excel Services-BI	SP Workflow Solutions-Excel
37;#PowerPoint;#141;#Meetings;#55;#OneNote;#155;#Word	PowerPoint-Meetings-OneNote-Word	PowerPoint-Meetings
148;#Access;#102;#Excel;#194;#Excel Services;#46;#BI	Access-Excel-Excel Services-BI	Access-Excel
248;#Bccess;#102;#Excel;#194;#Excel Services;#46;#BI	Bccess-Excel-Excel Services-BI	Bccess-Excel
28;#DCI;#139;#ApplicationVirtualization;#148;#BPOS;#138;#Microsoft PowerPoint	DCI-ApplicationVirtualization-BPOS-Microsoft PowerPoint	DCI-Application
12;#Word;#141;#Meetings;#55;#OneNote;#155;#Word	Word-Meetings-OneNote-Word	Word-Meetings
99;#AP Workflow Solutions;#102;#Excel;#194;#Excel Services;#46;#BI	AP Workflow Solutions-Excel-Excel Services-BI	AP Workflow Solutions-Excel
137;#PowerPoint;#141;#Meetings;#55;#OneNote;#155;#Excel	PowerPoint-Meetings-OneNote-Excel	PowerPoint-Meetings

Figure 6. Selected samples of incorrect model predictions on the Flashfill test set. These include both inconsistent programs, and consistent programs which failed to generalize.