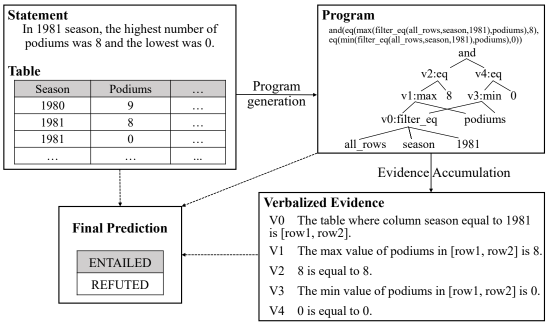
Learning to Generate Programs for Table Fact Verification via Structure-Aware Semantic Parsing

Suixin Ou, Yongmei Liu[*]
Dept. of Computer Science, Sun Yat-sen University, Guangzhou 510006, China
ousx@mail2.sysu.edu.cn, ymliu@mail.sysu.edu.cn

Abstract

Table fact verification aims to check the correctness of textual statements based on given semi-structured data. Most existing methods are devoted to better comprehending logical operations and tables, but they hardly study generating latent programs from statements, with which we can not only retrieve evidences efficiently but also explain reasons behind verifications naturally. However, it is challenging to get correct programs with existing weakly supervised semantic parsers due to the huge search space with lots of spurious programs. In this paper, we address the challenge by leveraging both lexical features and structure features for program generation. Through analyzing the connection between the program tree and the dependency tree, we define a unified concept, operation-oriented tree, to mine structure features, and introduce Structure-Aware Semantic Parsing to integrate structure features into program generation. Moreover, we design a refined objective function with lexical features and violation punishments to further avoid spurious programs. Experimental results show that our proposed method generates programs more accurately than existing semantic parsers, and achieves comparable performance to the SOTA on the large-scale benchmark TABFACT.



In 1981 season, the highest number of podiums was 8 and the lowest was 0.

Season	Podiums	…
1980	9	…
1981	8	…
1981	0	…
…	…	...

```
and(eq(max(filter_eq(all_rows,season,1981),podiums),8),
eq(min(filter_eq(all_rows,season,1981),podiums),0))
```

V0 The table where column season equal to 1981 is [row1, row2].
V1 The max value of podiums in [row1, row2] is 8.
V2 8 is equal to 8.
V3 The min value of podiums in [row1, row2] is 0.
V4 0 is equal to 0.

Figure 1: The pipeline of ProgVGAT (Yang et al., 2020) on TFV. Here the task is, given a table and a statement, to predict whether the table *entails* the statement or *refutes* it. Verbalized evidences are verbal descriptions of the program execution procedure.

1 Introduction

With the rise of misleading information on the Internet, such as fake news, rumors and political deceit, fact-checking has been developed as a means of detecting and filtering false information. *Table fact verification (TFV)* is a specific fact-checking task that requires performing logical operations such as comparison, superlative and aggregation over given tables to verify textual statements.

Programs play an important role in TFV. On one hand, correct programs can provide rationales for model decisions, which make reasoning analysis and failure diagnosis feasible (Zhou et al., 2018). On the other hand, they can be used to fetch the key evidences for verification. Figure 1 gives an example of mainstream methods (Zhong et al., 2020a; Shi et al., 2020b; Yang et al., 2020; Shi et al., 2021) for TFV. It first generates latent programs from statements, then collects evidences from tables by executing the programs over the tables, and finally leverages all information for final predictions. Compared with naive methods(Chen et al., 2020; Zhang et al., 2020a) which simply put statements and linearized tables into language models for verification, the mainstream methods additionally introduce programs to reveal the evidences (e.g., *verbalized evidence V1*) covered by logical operations (e.g., *max([row1, row2]], podiums)*) and to fetch the key information from the table (e.g., *8*). But an incorrect or spurious program may introduce irrelevant or even contradictory evidences. So it is crucial to get correct programs that properly extract evidences from tables, especially when tables are too large to be encoded by neural networks.

Despite being important, program generation remains underexplored for TFV. To the best of our knowledge, only LPA (Chen et al., 2020) works on program generation. It first searches programs

[*] Corresponding author

with human-designed features, then ranks them with a neural network, and finally uses the execution result of the top program as the prediction. However, it exhibits an unacceptable performance which means it generates incorrect programs. The remaining approaches just predict the correctness of statements but never concern about generating correct programs. In TFV, there is still a need to find better solutions for program generation.

Intuitively, we can resort to weakly supervised semantic parsing (Liang et al., 2011) for the program generation, but existing semantic parsers may fail in TFV for the amplified spurious program problem caused by the binary label. Due to the lack of program labels, existing methods will sample label consistent programs for model training. In TFV, any sampled program that outputs a Boolean value has a 50% chance of hitting the correct label; hence there are many label consistent programs, while only a small part of label consistent programs are correct, implying that the rest are all spurious.

In this paper, we carefully examine the syntax structures of statements and find that task-related structure features are the key to address the issue mentioned above. We propose a unified operation-oriented tree constructed in three steps. Firstly, we link entities between the table, trigger dictionary and statement. Secondly, we obtain the original tree using a dependency parser with the linked statement as input. Thirdly, the original tree is pruned and merged to a simplified tree that contains only information related to operations. Such a unified tree can provide distant supervision, assisting our model in generating single operations correctly and generating all operations in the correct order. As a result, we have a higher probability of getting correct programs and evading spurious ones. Then we introduce Structure-Aware Semantic Parsing (SASP) by designing a scoring function based on the proposed tree and fusing the sample distributions computed by the scoring function and neural network. At last, we design a refined objective function with lexical features and violation punishments to avoid spurious programs further.

Experimental results on Tabfact and Logic2Text show that SASP improves the performance of the baseline model significantly, and achieves comparable performance to the State-Of-The-Art method. Our contributions are as follows:

- We propose an operation-oriented tree to provide distant supervision for semantic parsing.
- We propose SASP which leverages both lexical features and structure features for the serious spurious problem in weakly supervised semantic parsing for TFV.
- With the proposed method, we can generate more accurate programs which can not only boost existing mainstream methods for TFV, but also provide explanation for verification.

2 Related Work

Fact Verification Fact verification aims at identifying the truthfulness of online textual statements given different sources of evidences, including document sets (Thorne et al., 2018; Nie et al., 2019; Zhong et al., 2020b; Wan et al., 2021), images (Suhr et al., 2019; Li et al., 2020) and structured tables (Chen et al., 2020; Zhong et al., 2020a; Shi et al., 2020b; Zhang et al., 2020a; Yang et al., 2020; Shi et al., 2021). Despite the sources of evidences used to support the verification vary, the methods for different tasks appear to have the same idea. They first locate the key evidences that will aid in their verification, then fuse the collected key evidences with the original statement to make the final prediction. In this paper, we focus on generating better programs that allow existing methods to get key evidences from tables efficiently, hence benefiting existing methods for TFV.

There are also many explainable fact verification works(Kotonya and Toni, 2020a). Attention based methods(Popat et al., 2018; Lu and Li, 2020; Wu et al., 2020) highlight key evidences according to attention weights. Atanasova et al. (2020); Kotonya and Toni (2020b) generate explanations in natural language with text summarization technology. Gad-Elrab et al. (2019); Ahmadi et al. (2020) use horn rules and knowledge graphs to mine explanations. Our work is similar to the third line of works from the perspective of explainability.

Semantic Parsing Due to the expensive cost of annotated programs, weakly supervised semantic parsing (Liang et al., 2011; Berant et al., 2013; Artzi and Zettlemoyer, 2013) has been proposed to learn program generation from sentence-label pairs. Compared with full supervision, weak supervision brings spurious problems: there may be spurious programs that accidentally reach the right answer for the wrong reason, and they will provide wrong supervision for model training. Previous work (Pasupat and Liang, 2016) uses crowd-sourced denotations to prune spurious programs. Liang et al. (2018) use both programs inside and outside the memory buffer to compute the expected return objective in case the neural model is misled by spurious programs inside memory. Dasigi et al. (2019); Misra et al. (2018); Agarwal et al. (2019) rely on lexical features to differentiate between spurious and correct programs. Most recently, Cao et al. (2019); Ye et al. (2019); Shao et al. (2021) exploit the semantic correlations between sentences and programs to rule out spurious programs via jointly learning semantic parser and sentence generator. In this paper, we focus on a more complex problem, learning program generation with (sentence, binary label) pairs, in this field, and take the above approaches a step further by leveraging both lexical features and structure features.

There already exist many works utilizing the structural correlations between a sentence and its programs. Previous works(Reddy et al., 2016; Hu et al., 2018) directly transform the dependency structure of a sentence into a program, which is not satisfactory on complex sentences. In recent years, some works(Wang et al., 2019; Herzig and Berant, 2021; Li et al., 2021) treat structural constraints as latent variables, then parse a sentence into a program under the constraints. However, it is difficult to learn latent variables in a noisy environment. Simultaneously, modeling structural correlations explicitly requres human annotations.(Sun et al., 2020; Shi et al., 2020a). In this paper, we propose a concise and robust method to integrate the structural correlations into semantic parsing.

3 Model

Structure-Aware Semantic Parsing (SASP) centers around the operation-oriented tree to deconstruct some compositionality of statement and generate program correctly. Figure 3 gives an overview of our proposed SASP. In this section, we will first introduce the task formulation, then describe how to construct the operation-oriented tree, and give the way to generate programs following the well-designed tree at last.

3.1 Problem Formulation and Notations

Given a table $T = \{cell_{i,j} | i \leq R, j \leq C\}$ with the table header $H = \{col_j | j \leq C\}$ as evidence, a statement $S = \{w_i | i \leq W\}$ with W words and a true label $y \in Y = \{True, False\}$ where $True$ means T entails S and $False$ means T refutes S, we aim to train a model to do explainable verification. More specifically, we train a model to translate S into an executable program z, then predict a label $\hat{y}_z \in Y$ by accessing the table T with program z such that $\hat{y}_z = y$. Different from most existing methods, which just pay attention to predicting a label $\hat{y} \in Y$ such that $\hat{y} = y$, our model also generates a program as accurate as possible to explain and support the verification.

Incorrect	filter_eq(all_rows, season, 1981); max(v0, podiums); eq(v1, 0); min(v0, podiums); eq(v3, 8); and(v2, v4)

Spurious	filter_eq(all_rows, podiums, 8); max(v0, season); eq(v1, 1981);
	filter_eq(all_rows, podiums, 0); min(v0, season); eq(v4, v1); and(v2, v5)

Correct	filter_eq(all_rows, season, 1981); max(v0, podiums); eq(v1, 8); min(v0, podiums); eq(v3, 0); and(v2, v4)

Figure 2: Different types of programs for the statement in figure 1. Both spurious and correct programs are label consistent as they can be executed to correct label, while only correct programs are semantic consistent as they reflect the underlying meaning of statements.

Program A program z can be seen as a set of executable operations[†] $\{op_i | i \leq M\}$. Considering the program example in figure 1, there are six operations in total, and each operation $op_i = \{op_i.func, ..., op_i.arg_j, ..., op_i.out\}$ has one operator $op_i.func$ (e.g., *filter_eq* in the figure), multiple operands $op_i.arg_j, 0 < j \leq \nu$ relevant to the table T (e.g., *all_rows*, *season* and *1981*) and one output $v_i = op_i.out$ which may be selected as an operand by subsequent operations. When the whole program is executed by an interpreter, it will be parsed into a tree as shown in figure 1 and executed from bottom to up. According to the execution correctness and the semantic consistency, we divide programs from the executable program set Z into three categories, as shown in figure 2.

3.2 Operation-Oriented Dependency Tree

In this part, we first reveal the connection between the program tree and the dependency tree. Then, we design a unified operation-oriented dependency tree for making full use of the connection.

Syntactic structures, the organization of tokens in a sentence and how the contexts among them are interrelated, can be revealed by a dependency tree whose nodes and edges correspond to words and grammatical relations in the sentence. We observe that: (1) the operations related to descendants tend to be executed before those related to ancestors in the dependency tree; (2) the operator and operands

[†] The definition of specific operations are listed in Appendix A.1

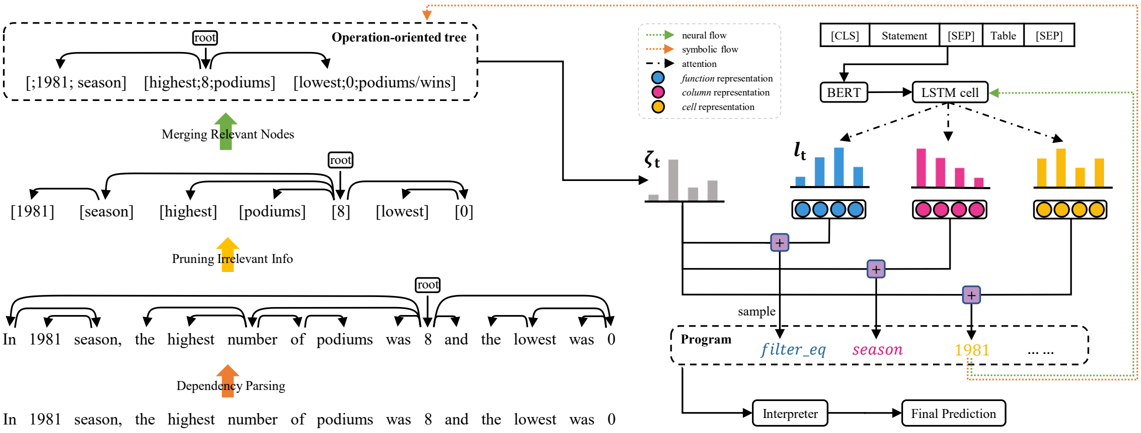


Figure 3: An overview of our proposed approach. The left part illustrates how to construct the operation-oriented tree. The right part depicts how to generate programs with the well-designed tree. The logits computed by LSTM for tokens with *function* type, *column* type and *cell* type are in blue, rose red and yellow respectively. And the scores given by the operation-oriented tree are in grey. They are combined to calculate the final sample distribution.

within one operation tend to have shorter distances in the dependency tree; in the correct program compared with the incorrect or spurious one. Use the dependency tree in figure 3 and the program in figure 1 as an example. The operation *filter_eq* related to the child node is executed before the operation *eq(v1, 8)* corresponding to the father node. What's more, the distance of operands in the incorrect operation *filter_eq(all_rows, podiums, 1981)* is 6, while that in *filter_eq(all_rows, season, 1981)*, a correct operation, is just 1.

The observations above suggest that there exist some structural correlations between a statement and its programs. We will present how to make use of them in the next section. Before that, we propose an operation-oriented dependency tree to strengthen the above rules in two steps. First, we prune the original dependency tree to focus exclusively on the operation-related structure. Then, we merge the information around every operation to make information in a single operation more compact. What's more, it is more convenient to define and calculate the distance in a simplified tree.

The left part of figure 3 illustrates how to construct the proposed tree. First of all, we do rule-based entity linking to find potential operators and operands from the statement. For operators detection, we match strings between the statement and the pre-defined trigger words[‡], and give the matched entities a *function* type. As for operands, we divide them into two types, *cell* and *column*, as they are linked to table cells and the table header respectively (e.g., *1981* has a *cell* type and *season* has a *column* type). Then we pass tokens and linked entities with types into a general dependency parser to get a dependency tree τ. Every linked entity node $n = \{n.children, n.type, n.val\}, n \in \tau$ has a list $type$ with one type and a list val with one entity. For every token node, its $type$ list and val list are both empty. After that, for every entity node with a *cell* type value $cell_{i,j}$, we will add *column* and col_j into its $type$ list and val list respectively. At last,

Algorithm 1 Operation-oriented tree construction

Input: Dependency tree τ with root ρ, where every node has a child list $children$, a type list $type$ and a value list val.
Output: Operation-oriented tree $\hat{\tau}$ with root $\hat{\rho}$.

```
function PRUNE($\rho$)
    $\hat{\rho}.children, \hat{\rho}.type, \hat{\rho}.val \leftarrow \{\}, \rho.type, \rho.val$
    for $c \in \rho.children$ do
        $\hat{c} \leftarrow$ PRUNE($c$)
        if MERGE($\hat{\rho}, \hat{c}$) then
            $\hat{\rho}.type \leftarrow \hat{\rho}.type \cup \hat{c}.type$
            $\hat{\rho}.val \leftarrow \hat{\rho}.val \cup \hat{c}.val$
            $\hat{\rho}.children \leftarrow \hat{\rho}.children \cup \hat{c}.children$
        else
            $\hat{\rho}.children \leftarrow \hat{\rho}.children \cup \{\hat{c}\}$
        end if
    end for
    return $\hat{\rho}$
end function

function MERGE($\hat{\rho}, \hat{c}$)
    for $(i, j) \in \{(i, j) | i < |\hat{\rho}.type|, j < |\hat{c}.type|\}$ do
        if $\hat{\rho}.type[i] = \hat{c}.type[j] \wedge \hat{\rho}.val[i] \neq \hat{c}.val[j]$ then
            Return $False$
        end if
    end for
    Return $True$
end function
```

[‡] All pre-defined trigger words are listed in Appendix A.2

we call PRUNE in algorithm 1 using τ as input and get output $\hat{\tau}$. The nodes left in the tree may contain function info corresponding to the logical operations, cell info and column info from tables.

3.3 Structure-Aware Semantic Parsing

In this section, we will introduce SASP, which unifies both structural features and lexical features with one operation-oriented dependency tree.

As shown in the right part of figure 3, we first employ BERT (Devlin et al., 2019) to encode the statement S and the table T following TABERT (Yin et al., 2020). Then we get representations for the statement and entities with different types, which will be fed into the decoder. During decoding, the logits are computed by an LSTM with attention mechanism(Luong et al., 2015):

$$\begin{aligned} h_t &= LSTM(h_{t-1}, x_{t-1}) \\ a_t &= MLP([h_t; Attention(h_t, S)]) \\ l_t &= MatMul(X_t, a_t) \end{aligned} \tag{1}$$

where h_t is the hidden state, x_{t-1} is the token generated previously, X_t is the candidate token list selected from the vocabulary according to the token type at timestep t (e.g., the type for the second token in the program being predicted is *column*), and l_t are the logits for the tth token over X_t.

However, in TFV, it is difficult to find the correct optimization direction with only attention mechanism, especially at the beginning of the training, because of the serious spurious problem. So we bias the logits with our proposed tree additionally. As a result, our model can give the correct program a higher probability, therefore exploring search space efficiently and evading spurious programs.

Algorithm 2 Scoring function with candidate token list X, operation-oriented tree $\hat{\tau}$ and operation being predicted *op* as input, where $\lambda < 1$ is a hyper-parameter.

```
function SCORE($X, \hat{\tau}, op$)
    $Score \leftarrow \{\}$
                                                ▷ operator selection
    if $op = \{\}$ then
        for $x \in X$ do
            $d \leftarrow$ Distance_to_leaf($x, \hat{\tau}$)
            $Score \leftarrow Score \cup \{\lambda^d\}$
        end for
        Return $Score$
    end if
                                                ▷ operand selection
    for $x \in X$ do
        $d \leftarrow 0$
        for $o \in op$ do
            $d \leftarrow d +$ Distance_in_tree($x, o, \hat{\tau}$)
        end for
        $Score \leftarrow Score \cup \{\lambda^{d/|op|}\}$
    end for
    Return $Score$
end function
```

More specifically, we design two scoring mechanisms in line with the two rules found in the previous section. As shown in algorithm 2, given $\lambda < 1$, $score = \lambda^{distance}$ means the closer distances, the higher scores. For operator selection, we calculate the average distance from the candidate $x \in X$ to its leaves in the tree $\hat{\tau}$, and set the distance to be $+\infty$ if it is not in the tree. For example, the candidate operator *max* (triggered by *highest*) has a score of λ^1. In this way, we give operators closer to leaves higher scores, which leads to operations related to descendants being generated before those related to ancestors. For operand selection, we compute the average distance from the candidate $x \in X$ to tokens in the operation *op*. Use the operation in figure 3 as an example, the score of the candidate *1981* is λ^0 when the timestep $t = 3$. In this way, we prioritize the tokens closed to existing information in the operation being generated, so that the distances inside one operation tend to be shorter in the dependency tree. At last, we combine the scores ζ_t given by algorithm 2 and the logits l_t computed by Equation 1 to get the final sample distribution:

$$\begin{aligned} \zeta_t &= Score(X_t, \hat{\tau}, op) \\ P(X_t|S, T, x_{<t}) &= Softmax(l_t + \alpha\zeta_t) \end{aligned} \tag{2}$$

where α is a hyper-parameter, $\hat{\tau}$ is the operation-oriented tree and *op* is the operation being predicted. After we sample $x_t \sim P(X_t|S, T, x_{<t})$, it will be used to update h_t, $\hat{\tau}$ and *op*. We give more details in Appendix A.3.

Previous works(Agarwal et al., 2019; Dasigi et al., 2019) measure the relevance between a sentence and a program by their coverage, and use that lexical coverage to augment the reward function. In a similar spirit, we design the reward based on our proposed tree. Our intuition is that different types of tokens play different roles in the operation-oriented tree, and therefore should be treated under varying degrees. And our reward is defined below.

$$R(z) = \begin{cases} \sum\limits_{\kappa \in Type} \sigma_\kappa r_\kappa, & \hat{y}_z = y \\ 0, & otherwise \end{cases} \tag{3}$$

where $Type = \{\text{“}function\text{”}, \text{“}cell\text{”}, \text{“}column\text{”}\}$, $\{r_\kappa | \kappa \in Type\}$ are relevances, $\{\sigma_\kappa | \kappa \in Type\}$

are hyper-parameters, and $\hat{y}_z$ is the label predicted by accessing the table T with the program z. Since all operation-related tokens of a statement are reserved in the operation-oriented tree, we can calculate the relevance between a statement and a program by

$$r_\kappa = \frac{\sum_{n \in \hat{\tau}} \mathbb{1}\{\exists i, n.type[i] = \kappa \wedge n.val[i] \in z\}}{\sum_{n \in \hat{\tau}} \mathbb{1}\{\exists i, n.type[i] = \kappa\}} \tag{4}$$

where $\{n | n \in \hat{\tau}\}$ are nodes of our proposed tree. For further improvement, we modify the generalized update equation in PolicyShaping (Misra et al., 2018) to get Maximum Likelihood Most Violation Reward. The final objective function is:

$$\begin{aligned} J_\theta = \sum_{(S,T) \in D} \Big(\sum_{z \in Z_{set}} R(z) \pi(z|S,T) \\ -\gamma \max_{z' \in Z_{err}} (\pi(z'|S,T)) \Big) \end{aligned} \tag{5}$$

where D contains all S-T pairs, Z_{set} is the set of sampled executable programs, $Z_{err} \subseteq Z_{set}$ is the set of incorrect programs, π is the sample policy, γ is a hyper-parameter and θ contains all the trainable parameters. We think such an update equation more robust than REINFORCE helps the model learn better with many spurious programs in Z_{set}.

4 Experiments

4.1 Experimental Settings

Dataset and Evaluation Metrics We conduct experiments on the large-scale dataset TABFACT (Chen et al., 2020), which aims to study fact verification given semi-structured data as evidence. TABFACT contains 16,573 tables and 118,275 statements which are divided into training (80%), validation (10%) and testing (10%) sets. The testing set is further partitioned into simple and complex sets. The statements in the complex set are more complicated in semantic compositionality than those in the simple set. Because there is no program ground-truth provided in TABFACT, we just use the label accuracy as metric for comparison, which is also called execution accuracy (Ex.Acc).

We also conduct experiments on WikiTableQuestion (WTQ) (Pasupat and Liang, 2015), a commonly used weakly supervised semantic parsing dataset, for further evaluation. And we use the same setting as previous works.

To test our performance on program generation, we use Logic2Text, a dataset that contains around 10,000 correct statement-table-program tuples, to evaluate parse tree matching accuracy (PT.Match) (Kim et al., 2020) for programs generated by our method and other methods that also provide programs. Because there are only "ENTAILED" statements in Logic2Text, we use the model trained on TABFACT to predict programs without tuning.

Implementation Details We use CRF2o (Zhang et al., 2020b) for dependency parsing. For semantic parsing, we use pytorch neural symbolic machine (Liang et al., 2017, 2018; Yin et al., 2020) as our baseline and improve it with the operation-oriented tree. Further, to bootstrap SASP, we use ζ_t in Equation 2 to sample around 10 label consistent programs per example, and load them into memory buffer before training. For BERT parameters, we set the hidden size to 768, and use Adam optimizer with lr 5e-5, warmup step 30k, dropout 0.2. For LSTM parameters, we set hidden size to 200, and use Adam optimizer with lr 3e-3, train step 150k, dropout 0.2. As for hyper-parameters λ, α, σ_{func}, σ_{cell}, σ_{column} and γ, we set them to 0.7, 2, 0.2, 0.4, 0.4 and 0.2 respectively. All experiments were conducted on a workstation with 128 GB of RAM and 2 RTX 3090 GPUs. Our source code is available at: `https://github.com/ousuixin/SASP`.

Compared Systems We compare our model with the following baselines, including six that focus on label prediction and two that pay extra attention to program generation. Among the former five methods, Table-BERT (Chen et al., 2020) and SAT (Zhang et al., 2020a) focus on table linearization, so they use different ways to change 2-dimensional tables into 1-dimensional sequences composed of tokens, and then feed them into BERT for label prediction. LFC (Zhong et al., 2020a), HeterTFV (Shi et al., 2020b), ProgVGAT (Yang et al., 2020) and LERGV(Shi et al., 2021) pay attention to comprehending tables and programs. They use different ways to encode programs (generated by LPA-ranking) and tables for verification, although the programs they use are not precise at all. The latter two methods will generate programs and use program execution results as final predictions, including LPA-ranking (Chen et al., 2020) and MAPO (Liang et al., 2018) with BERT.

4.2 Experimental Results

Performance on TABFACT Table 1 gives the overall performance of all eight baselines and our proposed SASP, from which we can observe that:

Model	Val	Test	Test(Simple)	Test(Complex)
Table-BERT	66.1	65.1	79.1	58.2
SAT	73.3	73.2	85.4	67.2
Tapas*	78.6	78.5	90.5	72.5
LFC	71.8	71.7	85.4	65.1
HeterTFV	72.5	72.3	85.9	65.7
ProgVGAT	74.9	74.4	**88.3**	67.6
LERGV	**75.6**	**75.5**	87.9	**69.5**
MAPO w/ BERT refined-reward	56.6	57.2	60.2	55.8
LPA-Ranking	65.2	65.0	78.4	58.5
SASP	75.0	74.9	87.6	68.8

Table 1: Overall performance (label accuracy) of different methods on TABFACT dataset. We don't compare our model with Tapas(Eisenschlos et al., 2020) directly. Because they focus on the design of pre-trained model and use extra data besides the TABFACT training data.

(1) As a semantic parsing method, our method achieves performance comparable to the State-Of-The-Art method LERGV while maintaining explainability. This is what previous semantic parsers can not do, and shows our superiority in TFV.

(2) Our proposed method works better than Table-BERT and SAT, demonstrating the power of the content snapshot proposed by Tabert in catching key information from a table.

(3) SASP has a lead of 1.2% on the the complex set compared with ProgVGAT, but falls behind on the simple set. There are two reasons for that. On one hand, mainstream methods like ProgVGAT can fix some errors caused by the symbolic interpreter (e.g., executing *eq("USA", "America")* to *False*). While SASP uses the execution result of the generated program as prediction. Due to the limited expression ability, our interpreter can not cover every statement with a correct program, leading to a lower probability of predicting a correct answer. On the other hand, ProgVGAT can not deal with structural mistakes (e.g., replacing *max* with *min* operation) in programs generated by LPA. As a result, ProgVGAT performs worse in complicated semantic environment where LPA has a higher probability of making a structural mistake.

(4) Our method outperforms MAPO and LPA by significant margins, suggesting that SASP can generate programs more accurately.

Performance on WTQ Table 2 shows the experimental results on WTQ. Our model just has comparable performance with our baseline, MAPO w/ BERT. We give two possible reasons below:

Model	Dev	Test
Pasupat and Liang (2015)	37.0	37.1
Dasigi et al. (2019)	43.1	44.3
Agarwal et al. (2019)	43.2	44.1
Wang et al. (2019)	43.7	44.5
MAPO w/ BERT (Yin et al., 2020)	**49.6**	49.4
SASP	49.3	**49.5**

Table 2: Performance (execution accuracy) of different methods on WikiTableQuestion. The first four are all previous works.

(1) As can be seen in figure 1, the program has more than three operations, which is quite common in TFV, while they use at most three operations to answer a question in previous works (Pasupat and Liang, 2015; Zhong et al., 2017; Liang et al., 2018). Because the compositionality of WTQ is lower than TABFACT, our proposed operation-oriented tree can only provide very limited help.

(2) The spurious program problem is further amplified by the binary label in TABFACT. Any program that outputs a Boolean value has a 50% chance of hitting the correct label; hence there are many label consistent programs. While in WTQ, it is not that easy to hit the correct label. Suppose that the vocabulary list has N tokens, but only one token corresponds to the answer. Every executable program in WTQ will output an answer with the string type, so it only has a $\frac{1}{N}$ probability of hitting the correct label. WTQ has much fewer spurious programs, so lexical features are enough to rule out

Model	PT.Match	Ex.Acc
MAPO w/ BERT	13.4	70.1
LPA	15.6	56.7
SASP	**47.9**	**75.9**

Table 3: Performance (matching accuracy and execution accuracy) of different methods on Logic2Text dataset.

spurious programs in WTQ in many cases.

Performance on Logic2Text Results of different semantic parsing methods are shown in table 3. Our model outperforms other methods with a considerable margin on PT.Match metric. This means SASP can generate more correct programs, which makes it behave well in table fact verification.

In program generation for TFV, the search space is too large to be explored completely. To tackle this problem, MAPO w/ refined reward performs systematic search space exploration guided by lexical features in the advanced reward function. It only obtains PT.Match accuracy of 13.4% on Logic2Text. The high Ex.Acc score shows that it just predicts spurious programs executed to "True". For LPA, it first collects all programs under the search space restricted by a lexical feature based algorithm, then ranks these programs with a neural network (BERT). And LPA also has poor behavior in program generation here.

The big gaps (more than 40% in MAPO and LPA) between PT.Match and Ex.Acc accuracy suggest that with only lexical features, there are still many spurious programs being explored. Use the spurious program in figure 2 as an example, it conforms to lexical features by making full use of sentence tokens, and would be a promising candidate in MAPO and LPA. However, such kind of programs will differ from the correct ones in the order of operators or the position of operands, so they can be distinguished from correct programs by structure features. Our method captures both lexical and structure features, therefore evading such spurious programs and biasing generated programs from label consistent towards semantic consistent. The smaller gap (28% in SASP) between PT.Match and Ex.Acc accuracy confirms our analysis above.

4.3 Ablation Study

Effect of Structural Info We further conduct an ablation study to evaluate the necessity of leveraging structure information through rules (1) and (2). For rule (1), which defines the operator selection mechanism, we just drop types and values related to *function* in our proposed tree to see how it influence. For rule (2), which defines the operand selection mechanism, we drop types and values related to *cell* or *column*. If we drop all types from the tree, the algorithm degenerates into MAPO w/ BERT refined-reward violation. The experimental results are given in Table 4. We can see that *function* is the most important type, then is *column* type, followed by *cell* type. And all of the types make significant contributions to the final performance. The results above show that both mechanisms associated with the rule (1) and the rule (2) are crucial for our model because both operator and operand selections are crucial for program generation.

Model	Val	Test
SASP w/o proposed tree	56.6	57.2
SASP w/o function type	59.3	60.1
SASP w/o column type	60.5	61.5
SASP w/o cell type	70.2	71.1
SASP	**75.0**	**74.9**

Table 4: Results (label accuracy) of ablation study that shows the effectiveness of our proposed tree.

Model	Val	Test
SASP w/ binary-reward	60.1	60.2
SASP w/o violation	73.5	73.1
SASP	**75.0**	**74.9**

Table 5: Results (label accuracy) of ablation study that shows well defined reward function and violation punishiment contribute a lot to our method.

Effect of Objective Function To evaluate the impact of the refined objective function in Equation 5, we conduct another ablation study, and the results are shown in table 5.

We change the reward function in Equation 3 with a binary reward function for comparison. The result shows that refined feedback taking lexical features into account plays an essential role in our model. Without the refined reward, some operations may be omitted because the partial programs are already executed to the right label, resulting in a much worse performance.

We also remove the violation punishment to investigate the necessity of a conservative update policy. The result shows that the robust update policy

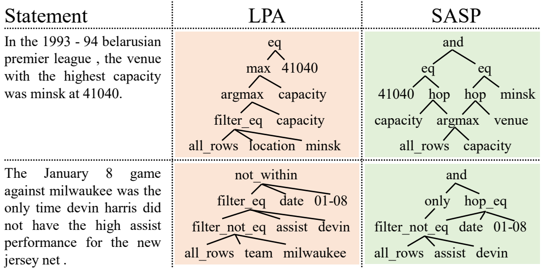


Figure 4: Cases in Logic2Text dataset. We visualize the programs with tree structures.

makes around 1% improvement. The reward function we designed just prioritizes programs that use tokens related to logical operators or tables as much as possible, leading to label inconsistent programs that meet the condition. Giving such programs a punishment complements the refined reward.

4.4 Case Study

In figure 4, we provide two cases to demonstrate the effectiveness of our method for program generation. In both cases, our method generates correct programs that are semantic consistent with the statement, while LPA screws them all up. In the first case, *max* is the descendant compared with *minsk* in the dependency tree, so our method uses *max* before *minsk*, while LPA gets the wrong order. This confirms that our method generates programs in the correct order with the operator selection mechanism. In the second case, *devin* has a more close relation to *not* in the dependency tree, so our method chooses *devin* as an operand of *filter_not_eq*, while LPA selects an incorrect operand *milwaukee* for *filter_not_eq*. This confirms that our method generates single operations correctly with the operand selection mechanism.

4.5 Error Analysis

To check the generalizability and limitations of our proposed method, we randomly sampled 200 examples from the validation set of TABFACT, and manually inspected the top one program of the beam search using SASP. We found that SASP generated correct programs for 99 examples, spurious programs for 57 examples and incorrect programs for 44 examples. The proportion of correct programs (49.5%) and spurious programs (28.5%) is similar to that in table 3 (47.9% and 28%). This shows the generalizability of SASP and the rationality of using Logic2Text for PT.Match evaluation. What's more, we classified the causes of 101 spurious or incorrect programs into four main categories.

Unsupported operations cause 30 error examples. For instance, in *"the new york rangers beat the atlanta flames by 2 points"*, the minus operation in a single table cell *"4 - 2"* is not supported by our interpreter. The second category of errors occur when the functions or entities can not be detected and added to dependency tree nodes correctly. Use *"the maroon played 3 teams located in the united states"* as an example, *"the united states"* can not be linked to *"America"* in the given table; hence it will not be added to the operation tree. 31 error examples are caused by this reason. The first two categories can not be handled by our proposed method, and we leave the development of powerful interpreter and robust entity linker for future work.

The third category is structure error, causing 13 error examples. In other words, the order of operators or the position of operands in the predicted program differs from the correct one. The wrong programs in figure 2 are all this kind of error cases. Underutilized information causes 23 error examples. For the statement in figure 1, *"filter_eq(all_rows, season, 1981); max(v0, podiums), eq(v1, 8)"* causes this kind of error.

5 Conclusion

In this paper, we have proposed a novel approach to do explainable verification by structure-aware semantic parsing. Firstly, we define a unified operation-oriented tree by entity linking, dependency parsing and tree pruning. Then, we demonstrate how to integrate our proposed tree into semantic parsing with the operator-related and the operand-related principles. At last, we introduce the refined objective function which could reduce the influence of spurious programs. Experimental results confirm that our proposed method can bias program generation from label consistent towards semantic consistent and achieve acceptable performance on the benchmark dataset TABFACT.

Future work will collect evidences that are more precise and get better verification performance by replacing LPA with SASP in the first stage of mainstream methods.

Acknowledgement

We appreciate the discussion with Weilin Luo, Weinan He and Yeliang Xiu. We acknowledge support from the Natural Science Foundation of China under Grant No. 62076261.

References

Rishabh Agarwal, Chen Liang, Dale Schuurmans, and Mohammad Norouzi. 2019. Learning to generalize from sparse and underspecified rewards. In *International Conference on Machine Learning*, pages 130–140.

Naser Ahmadi, Thi-Thuy-Duyen Truong, Le-Hong-Mai Dao, Stefano Ortona, and Paolo Papotti. 2020. Rulehub: A public corpus of rules for knowledge graphs. *ACM J. Data Inf. Qual.*, 12:21:1–21:22.

Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62.

Pepa Atanasova, Jakob Grue Simonsen, Christina Lioma, and Isabelle Augenstein. 2020. Generating fact checking explanations. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7352–7364.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544.

Ruisheng Cao, Su Zhu, Chen Liu, Jieyu Li, and Kai Yu. 2019. Semantic parsing with dual learning. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 51–64.

Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2020. Tabfact: A large-scale dataset for table-based fact verification. In *8th International Conference on Learning Representations*.

Pradeep Dasigi, Matt Gardner, Shikhar Murty, Luke Zettlemoyer, and Eduard H. Hovy. 2019. Iterative search for weakly supervised semantic parsing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2669–2680.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics, Volume 1: Long and Short Papers*, pages 4171–4186.

Julian Martin Eisenschlos, Syrine Krichene, and Thomas Müller. 2020. Understanding tables with intermediate pre-training. In *Findings of the Association for Computational Linguistics*, Findings of ACL, pages 281–296.

Mohamed H. Gad-Elrab, Daria Stepanova, Jacopo Urbani, and Gerhard Weikum. 2019. Exfakt: A framework for explaining facts over knowledge graphs and text. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 87–95.

Jonathan Herzig and Jonathan Berant. 2021. Span-based semantic parsing for compositional generalization. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, pages 908–921.

Sen Hu, Lei Zou, and Xinbo Zhang. 2018. A state-transition framework to answer complex questions over knowledge base. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2098–2108.

Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural language to SQL: where are we today? *Proceedings of the VLDB Endowment*, 13(10):1737–1750.

Neema Kotonya and Francesca Toni. 2020a. Explainable automated fact-checking: A survey. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 5430–5443.

Neema Kotonya and Francesca Toni. 2020b. Explainable automated fact-checking for public health claims. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pages 7740–7754.

Xiujun Li, Xi Yin, Chunyuan Li, Pengchuan Zhang, Xiaowei Hu, Lei Zhang, Lijuan Wang, Houdong Hu, Li Dong, Furu Wei, et al. 2020. Oscar: Object-semantics aligned pre-training for vision-language tasks. In *European Conference on Computer Vision*, pages 121–137.

Yuntao Li, Bei Chen, Qian Liu, Yan Gao, Jian-Guang Lou, Yan Zhang, and Dongmei Zhang. 2021. Keep the structure: A latent shift-reduce parser for semantic parsing. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, pages 3864–3870.

Chen Liang, Jonathan Berant, Quoc V. Le, Kenneth D. Forbus, and Ni Lao. 2017. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Volume 1: Long Papers*, pages 23–33.

Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V. Le, and Ni Lao. 2018. Memory augmented policy optimization for program synthesis and semantic parsing. In *Annual Conference on Neural Information Processing Systems*, pages 10015–10027.

Percy Liang, Michael Jordan, and Dan Klein. 2011. Learning dependency-based compositional semantics. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, pages 590–599.

Yi-Ju Lu and Cheng-Te Li. 2020. GCAN: graph-aware co-attention networks for explainable fake news detection on social media. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 505–514.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421.

Dipendra Misra, Ming-Wei Chang, Xiaodong He, and Wen-tau Yih. 2018. Policy shaping and generalized update equations for semantic parsing from denotations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2442–2452.

Yixin Nie, Haonan Chen, and Mohit Bansal. 2019. Combining fact extraction and verification with neural semantic matching networks. In *The Thirty-Third AAAI Conference on Artificial Intelligence*, volume 33, pages 6859–6866.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing, Volume 1: Long Papers*, pages 1470–1480.

Panupong Pasupat and Percy Liang. 2016. Inferring logical forms from denotations. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Volume 1: Long Papers*, pages 23–32.

Kashyap Popat, Subhabrata Mukherjee, Andrew Yates, and Gerhard Weikum. 2018. DeClarE: Debunking fake news and false claims using evidence-aware deep learning. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 22–32.

Siva Reddy, Oscar Täckström, Michael Collins, Tom Kwiatkowski, Dipanjan Das, Mark Steedman, and Mirella Lapata. 2016. Transforming dependency structures to logical forms for semantic parsing. *Trans. Assoc. Comput. Linguistics*, 4:127–140.

Zhihong Shao, Lifeng Shang, Qun Liu, and Minlie Huang. 2021. A mutual information maximization approach for the spurious solution problem in weakly supervised question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, Volume 1: Long Papers*, pages 4111–4124.

Jiaxin Shi, Shulin Cao, Liangming Pan, Yutong Xiang, Lei Hou, Juanzi Li, Hanwang Zhang, and Bin He. 2020a. Kqa pro: A large-scale dataset with interpretable programs and accurate sparqls for complex question answering over knowledge base. *arXiv preprint arXiv:2007.03875*.

Qi Shi, Yu Zhang, Qingyu Yin, and Ting Liu. 2020b. Learn to combine linguistic and symbolic information for table-based fact verification. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 5335–5346.

Qi Shi, Yu Zhang, Qingyu Yin, and Ting Liu. 2021. Logic-level evidence retrieval and graph-based verification network for table-based fact verification. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 175–184.

Alane Suhr, Stephanie Zhou, Ally Zhang, Iris Zhang, Huajun Bai, and Yoav Artzi. 2019. A corpus for reasoning about natural language grounded in photographs. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, Volume 1: Long Papers*, pages 6418–6428.

Yawei Sun, Lingling Zhang, Gong Cheng, and Yuzhong Qu. 2020. SPARQA: skeleton-based semantic parsing for complex questions over knowledge bases. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence*, pages 8952–8959.

James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. 2018. FEVER: a large-scale dataset for fact extraction and verification. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics, Volume 1: Long Papers*, pages 809–819.

Hai Wan, Haicheng Chen, Jianfeng Du, Weilin Luo, and Rongzhen Ye. 2021. A dqn-based approach to finding precise evidences for fact verification. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, Volume 1: Long Papers*, pages 1030–1039.

Bailin Wang, Ivan Titov, and Mirella Lapata. 2019. Learning semantic parsers from denotations with latent structured alignments and abstract programs. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 3772–3783.

Lianwei Wu, Yuan Rao, Yongqiang Zhao, Hao Liang, and Ambreen Nazir. 2020. DTCA: decision tree-based co-attention networks for explainable claim verification. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1024–1035.

Xiaoyu Yang, Feng Nie, Yufei Feng, Quan Liu, Zhigang Chen, and Xiaodan Zhu. 2020. Program enhanced fact verification with verbalization and graph attention network. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pages 7810–7825.

Hai Ye, Wenjie Li, and Lu Wang. 2019. Jointly learning semantic parser and natural language generator via dual information maximization. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2090–2101.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. Tabert: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426.

Hongzhi Zhang, Yingyao Wang, Sirui Wang, Xuezhi Cao, Fuzheng Zhang, and Zhongyuan Wang. 2020a. Table fact verification with structure-aware transformer. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pages 1624–1629.

Yu Zhang, Zhenghua Li, and Min Zhang. 2020b. Efficient second-order TreeCRF for neural dependency parsing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3295–3305.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

Wanjun Zhong, Duyu Tang, Zhangyin Feng, Nan Duan, Ming Zhou, Ming Gong, Linjun Shou, Daxin Jiang, Jiahai Wang, and Jian Yin. 2020a. Logicalfactchecker: Leveraging logical operations for fact checking with graph module network. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6053–6065.

Wanjun Zhong, Jingjing Xu, Duyu Tang, Zenan Xu, Nan Duan, Ming Zhou, Jiahai Wang, and Jian Yin. 2020b. Reasoning over semantic-level graph for fact checking. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6170–6180.

Mantong Zhou, Minlie Huang, and Xiaoyan Zhu. 2018. An interpretable reasoning network for multi-relation question answering. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 2010–2022.

A Appendix

A.1 Pre-difined API

As shown in figure 3 and algorithm 1, when we generate an operation, we first select an operand, then check the pre-defined API to get the type information, and finally select operands under the specific type (according to the type information). Here we list detailed descriptions for the pre-defined API in table 6.

Actually, there are seven different types, including *Function*, *Cell-String*, *Cell-Number*, *Bool*, *Sub-Table*, *Column-String* and *Column-Number*. In figure 3 and algorithm 1, we divide them into three types for a clearer illustration:

function	cell	column
Function	Bool, Sub-Table, Cell-String, Cell-Number	Column-String, Column-Number

In practice, we will select operands according to more detailed type information given by our predefined API.

In addition, we will update cell values and representations by adding the execution result of *op* and the LSTM hidden state h to $Cell$ and C, respectively (Line 12). In practice, we will maintain more detailed symbol lists and representation lists. For example, when the last token of the operation *max(v0, podiums)* is generated, the hidden state of LSTM will be added into the *C-Number* list, while the execution result of *max(v0, podiums)*, *v=8*, will be put into the *Cell-Number* list.

A.2 Pre-difined Trigger Words

In the first step of the operation-oriented tree construction, we match strings between the statement and the pre-defined trigger words to find underlying operators. Here we give details about the pre-defined trigger words in table 7, partly following LPA (Chen et al., 2020).

A.3 Implement Details for Decoding Module

Algorithm 3 gives the complete process of our decoding module. We initialize the program as an empty list with no operations, then enlarge it with operations generated progressively until the neural network outputs a *"stop"* token (Line 13-15). As for operation generation, we first sample an operator from the operator list $Func$, then get the type information of its operands through the pre-defined API, with which we can choose representations under the correct type. After that, we will sample operands from $Cell$ if the operand type is *cell*, and sample them from $Header$ otherwise. Once the generation is finished, the whole expression is added to the program (Line 3-11). All these above are similar to what they do in NSM (Liang et al., 2017). But we redesign the SAMPLE function according to equation 2.

Besides, to maintain our proposed tree $\hat{\tau}$, we will update information by dropping out the used operators and *cell* type operands in *op*. What's more, we will update C, the cell representation list, by adding current hidden state into into C (Line 12). At the same time, *op* and $Cell$ are updated by adding the execution result of *op* ($v = op.out$).

Algorithm 3 Program sampling with statement representation V_s, table cell representation list $C = \{V_c | c \in Cell\}$, table column representation list $H = \{V_h | h \in Header\}$, operator representation list $F = \{V_f | f \in Func\}$, special token list $E = \{V_e | e \in \{continue, stop\}\}$, the pre-defined API, neural network $LSTM$ and the operation-oriented tree $\hat{\tau}$ as input.

```
 1: $z \leftarrow \{\}$
 2: while $True$ do
 3:     $op \leftarrow \{\text{SAMPLE}(F, Func, \hat{\tau}, \{\})\}$
 4:     for $\kappa \in API[op[0]]$ do
 5:         if $\kappa =$ ”$cell$” then
 6:             $op \leftarrow op \cup \{\text{SAMPLE}(C, Cell, \hat{\tau}, op)\}$
 7:         else if $\kappa =$ “$column$” then
 8:             $op \leftarrow op \cup \{\text{SAMPLE}(H, Header, \hat{\tau}, op)\}$
 9:         end if
10:     end for
11:     $z \leftarrow z \cup op$
12:     $\hat{\tau} \leftarrow$ Update_info$(C, Cell, \hat{\tau}, op)$
13:     if $\text{SAMPLE}(E, \{continue, stop\}) = stop$ then
14:         break
15:     end if
16: end while
17: Return $z$
18:
19: function $\text{SAMPLE}(V, X, \hat{\tau}, op)$
20:     $logits \leftarrow \text{Score}_{Att}(V, V_s, h) + \alpha\ \text{SCORE}(X, \hat{\tau}, op)$,
    where $\text{Score}_{Att}$ means attention over context of $V_s$ followed by matrix multiplication and softmax over $V$
21:     $Probs$ =Softmax$(logits)$
22:     $x \leftarrow$ Random_multinomial$(X, Probs)$
23:     $h \leftarrow LSTM(h, x)$
24:     Return $x$
25: end function
26:
27: function $\text{SCORE}(X, \hat{\tau}, op)$
28:     $Score \leftarrow \{\}$
29:     if $op = \{\}$ then
30:         for $x \in X$ do
31:             $d \leftarrow$ Distance_to_leaf$(x, \hat{\tau})$
32:             $Score \leftarrow Score \cup \{\lambda^d\}$
33:         end for
34:         Return $Score$
35:     end if
36:     for $x \in X$ do
37:         $d \leftarrow 0$
38:         for $o \in op$ do
39:             $d \leftarrow d +$ Distance_in_tree$(x, o, \tau)$
40:         end for
41:         $Score \leftarrow Score \cup \{\lambda^{d/|op|}\}$
42:     end for
43:     Return $Score$
44: end function
```

Operator (function)	Operands	Output	Operation description
count	Sub-Table	Cell-Number	Return the number of rows in the given sub-table
is_none	Sub-Table	Bool	Return whether the given sub-table is none
is_not	Bool	Bool	Return false if the input is true, return true otherwise
avg/sum/max/min	Sub-Table, Column-Number	Cell-Number	Return the average/ summation/ max/ min value under the Column-Number column of the given sub-table
argmax/argmin	Sub-Table, Column-Number	Sub-Table	Return the sub-table with the maximum/minimum value under the Column-Number column of the given sub-table
hop	Sub-Table, Column-Number /Column-String	Cell-Number/ Cell-String	Return the Cell value under the given header column
hop_str_contain_not_any/ hop_str_contain_any	Sub-Table, Cell-String, Column-String	Bool	Return whether the given Cell-String value exists under the Column-String column of the given sub-table
hop_eq/ hop_not_eq/hop_less/ hop_less_eq/ hop_greater/ hop_greater_eq	Sub-Table, Cell-Number, Column-Number	Bool	Return whether the value under the Column-Number column of the given sub-table equal/not equal/less/ less equal/greater/greater equal to the given Cell-Number
filter_str_contain_not_any /filter_str_contain_any	Sub-Table, Cell-String, Column-String	Sub-Table	Return the sub-table of the given with the value under the Column-String column equal/not equal to the given Cell-String
filter_eq/filter_not_eq/ filter_less/filter_less_eq/ filter_greater/ filter_greater_eq	Sub-Table, Cell-Number, Column-Number	Sub-Table	Return the sub-table of the given with the value under the Column-Number column equal/not equal/less/less equal/greater/greater equal to the given Cell-Number
diff	Sub-Table, Sub-Table, Column-Number	Cell-Number	Return the difference between numbers in the Column-Number column of the first sub-table and second sub-table
same/row_less/ row_less_eq/row_greater/ row_greater_eq	Sub-Table, Sub-Table, Column-Number	Bool	Return whether the number under Column-Number column of the first sub-table is equal/less/less equal/ greater/greater equal to that of the second sub-table
equal/less/less_eq/ greater/greater_eq	Cell-Number, Cell-Number	Bool	Return whether the first number is equal/less/less equal /greater/greater equal to the second number
mode	Sub-Table, Sub-Table	Bool	Return whether the first sub-table dominates the second sub-table with more than half of rows
all	Sub-Table, Sub-Table	Bool	Return whether the first sub-table takes all rows of the second sub-table
only	Sub-Table	Bool	Return whether the given sub-table only has one row
and/or	Bool, Bool	Bool	Return the Boolean operation results of two inputs

Table 6: Details of the pre-defined API.

Operator (function)	Trigger word list
filter_str_contain_not_any, filter_not_eq	["other than", "not", "no", "never", "n’t"]
is_none	["none", "neither", "not", "no", "never", "n’t"]
is_not	["not", "no", "never", "n’t"]
filter_less_eq, row_less_eq, less_eq,	["at most"]
filter_greater_eq, row_greater_eq, greater	["at least"]
filter_less, less, row_less	["less", "sooner", "faster", "closer", "earlier", "lesser", "smaller", "younger", "worse", "shorter", "fewer", "lower", "behind", "below", "before", "under"]
filter_greater, row_greater, greater	["longer", "taller", "older", "more", "greater", "larger", "slower", "bigger", "better", "higher", "faster", "later", "above", "over", "after"]
same	["same"]
diff	["difference", "gap"]
sum	["total", "sum", "summation"]
avg	["average", "avg", "mean"]
argmax, max	["greatest", "biggest", "tallest", "strongest", "highest", "longest", "largest", "oldest", "most", "fastest", "best", "latest", "top", "first", "max", "maximum"]
argmin, min	["fewest", "closest", "earliest", "smallest", "lowest", "shortest", "poorest", "youngest", "nearest", "least", "slowest", "worst", "latest", "bottom", "last", "minimum"]
mode	["most", "majority", "main", "usually"]
only	["only"]
all	["always", "all", "every", "each"]

Table 7: Details of pre-defined trigger words.