Semantic Scaffolds for Pseudocode-to-Code Generation

Ruiqi Zhong, Mitchell Stern, Dan Klein
Online Judges

LeetCode, Hackerrank, Codeforce, etc.

Coding Challenge Description

Code

Input-output Test Cases
Task

Generate

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 declare constant integer numOfAlphabets = 26</td>
<td>const int numOfAlphabets = 26;</td>
</tr>
<tr>
<td>2</td>
<td>int main() {</td>
</tr>
<tr>
<td>3 create string s</td>
<td>string s;</td>
</tr>
<tr>
<td>4 let Count = 0 be an integer</td>
<td>int Count = 0;</td>
</tr>
<tr>
<td>5 read s</td>
<td>cin &gt;&gt; s;</td>
</tr>
<tr>
<td>6 Set Ch to be a</td>
<td>char Ch = 'a';</td>
</tr>
<tr>
<td>7 for i = 0 to i less than the length of s</td>
<td>for (int i = 0; i &lt; s.length(); i++) {</td>
</tr>
<tr>
<td>8 set Count to Count + min(abs(s[i] - Ch), numOfAlphabets - abs(s[i] - Ch))</td>
<td>Count += min(abs(s[i] - Ch), numOfAlphabets - abs(s[i] - Ch));</td>
</tr>
<tr>
<td>9 set s[i] to Ch</td>
<td>Ch = s[i];</td>
</tr>
<tr>
<td>10</td>
<td>}</td>
</tr>
<tr>
<td>11 print Count</td>
<td>cout &lt;&lt; Count &lt;&lt; endl;</td>
</tr>
<tr>
<td>12</td>
<td>return 0;</td>
</tr>
<tr>
<td>13</td>
<td>}</td>
</tr>
</tbody>
</table>

Code: solutions to the coding challenge.
Pseudocode: human annotated instructions based on the code.
# Language Ambiguities

<table>
<thead>
<tr>
<th>#</th>
<th>Pseudocode</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>declare constant integer numOfAlphabets = 26</td>
<td>const int numOfAlphabets = 26;</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>int main() {</td>
</tr>
<tr>
<td>3</td>
<td>create string s</td>
<td>string s;</td>
</tr>
<tr>
<td>4</td>
<td>let Count = 0 be an integer</td>
<td>int Count = 0;</td>
</tr>
<tr>
<td>5</td>
<td>read s</td>
<td>cin &gt;&gt; s;</td>
</tr>
<tr>
<td>6</td>
<td><strong>Set Ch to be a</strong></td>
<td><strong>char Ch = ’a’;</strong></td>
</tr>
<tr>
<td>7</td>
<td>for i = 0 to i less than the length of s</td>
<td>for (int i = 0; i &lt; s.length(); i++) {</td>
</tr>
<tr>
<td>8</td>
<td>set Count to Count + min(abs(s[i] - Ch), numOfAlphabets - abs(s[i] - Ch))</td>
<td>Count += min(abs(s[i] - Ch), numOfAlphabets - abs(s[i] - Ch));</td>
</tr>
<tr>
<td>9</td>
<td>set s[i] to Ch</td>
<td>Ch = s[i];</td>
</tr>
<tr>
<td>10</td>
<td>print Count</td>
<td>cout &lt;&lt; Count &lt;&lt; endl;</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>return 0;</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Several Possibilities

char Ch = ‘a’;
char Ch = a;
Ch = ‘a’;
Ch = a;
### Pseudocode

1. declare constant integer numOfAlphabets = 26
2. create string s
3. let Count = 0 be an integer
4. read s
5. Set Ch to be a
6. for i = 0 to i less than the length of s
7. set Count to Count + min(abs(s[i] - Ch),
   numOfAlphabets - abs(s[i] - Ch))
8. set s[i] to Ch
9. print Count

### Top-k Code Candidate

```cpp
const int numOfAlphabets = 26;
int main() {
    string s;
    int Count = 0;
    cin >> s;
    char Ch = 'a';
    for (int i = 0; i < s.length(); i++) {
        Count += min(abs(s[i] - Ch),
                      numOfAlphabets - abs(s[i] - Ch));
        Ch = s[i];
    }
    cout << Count << endl;
    return 0;
}
```

**Evaluation:** execute the top-k (k=1, 10, 100, 1000) program candidates on the inputs and check outputs. Inputs/outputs come from the online judges.

- Input: “zeus” -> Output: 18
- Input: “map” -> Output: 35
- Input: “ares” -> Output: 34
Task Properties

- **Length**
  - 15 lines per-program on average, 457 lines max.
  - Need to generate line-by-line (instead of learning end-to-end).

- **Semantic Evaluation**
  - Execution correctness (instead of BLEU score/other surface form metrics).
  - “i += 1;” is equivalent to “i = i + 1;”

- **Search**
  - Top-1 solution is not enough.
  - Quality of top-1000 solution might matter.
  - We care about efficiency.
Task Properties

- **Length**
  - 15 lines per-program on average, 457 lines max.
  - Need to generate line-by-line (instead of learning end-to-end).

- **Semantic Evaluation**
  - Execution correctness (instead of BLEU score/other surface form metrics).
  - “i += 1;” is equivalent to “i = i + 1;”

- **Search**
  - Top-1 solution is not enough.
  - Quality of top-1000 solution might matter.
  - We care about efficiency.
Task Properties

- **Length**
  - 15 lines per-program on average, 457 lines max.
  - Need to generate line-by-line (instead of learning end-to-end).

- **Semantic Evaluation**
  - Execution correctness (instead of BLEU score/other surface form metrics).
  - “i += 1;” is equivalent to “i = i + 1;”

- **Search**
  - Top-1 solution is not enough.
  - Quality of top-1000 solution might matter.
  - We care about efficiency.
Task Properties

- **Length**
  - 15 lines per-program on average, 457 lines max.
  - Need to generate line-by-line (instead of learning end-to-end).

- **Semantic Evaluation**
  - Execution correctness (instead of BLEU score/other surface form metrics).
  - “i += 1;” is equivalent to “i = i + 1;”

- **Search**
  - Top-1 solution is not enough.
  - Quality of top-1000 solution might matter.
  - We care about efficiency.

We propose a method to efficiently generate 1000 candidates for long programs and search for a semantically correct solution.
Notation.

- $L$: # of lines.
- $l$: line index.
- $x_l$: the pseudocode input at line $l$.

Translate each line independently.

- Generate 100 code fragment candidates $y_{lc}$ for each $x_l$.
- $y_{lc}$ is assigned a probability score $p_{lc}$.

Generate full program $y$.

- For each line $l$ we select one code fragment $c_l$, then concatenate the code fragments.
- We score a full program by taking the independent scoring for each line.
Notation.
- \( L \): # of lines.
- \( l \): line index.
- \( x_l \): the pseudocode input at line \( l \).

Translate each line independently.
- Generate 100 code fragment candidates \( y_{lc} \) for each \( x_l \).
- \( y_{lc} \) is assigned a probability score \( p_{lc} \).

Generate full program \( y \).
- For each line \( l \) we select one code fragment \( c_l \), then concatenate the code fragments.
- We score a full program by taking the independent scoring for each line.

### Pseudocode \( x_l \)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td><code>create string s</code></td>
</tr>
<tr>
<td>4</td>
<td><code>let Count = 0 be an integer</code></td>
</tr>
<tr>
<td>5</td>
<td><code>read s</code></td>
</tr>
<tr>
<td>6</td>
<td><code>Set Ch to be a</code></td>
</tr>
</tbody>
</table>

Baseline

\[
\begin{align*}
\text{\( p_{61} = 0.70 \), } & \quad \text{\( y_{61} \text{ char Ch = 'a'}; \)} \\
\text{\( p_{62} = 0.15 \), } & \quad \text{\( y_{62} \text{ char Ch = a}; \)} \\
\text{\( p_{63} = 0.05 \), } & \quad \text{\( y_{63} \text{ Ch = 'a'}; \)} \\
\text{\( p_{64} = 0.02 \), } & \quad \text{\( y_{64} \text{ Ch = a}; \)} \\
\text{...} & \\
\text{Others omitted,} & \\
\text{altogether 100 of candidates.} & \\
\end{align*}
\]
Baseline

- **Notation.**
  - $L$: # of lines.
  - $l$: line index.
  - $x_l$: the pseudocode input at line $l$.

- **Translate each line independently.**
  - Generate 100 code fragment candidates $y_{lc}$ for each $x_l$.
  - $y_{lc}$ is assigned a probability score $p_{lc}$.

- **Generate full program $y$.**
  - For each line $l$ we select one code fragment $c_l$, then concatenate the code fragments.
  - We score a full program by taking the independent scoring for each line.

\[
\begin{align*}
  y &= \text{Concat}_{l=1}^{L} y_{lc} \\
  \text{score}(y) &= \prod_{l=1}^{L} p_{lc}
\end{align*}
\]
Select one column (fragment) for each row (line) to form a full program.

<table>
<thead>
<tr>
<th>Line number</th>
<th>Best Candidate</th>
<th>2nd Candidate</th>
<th>3rd Candidate</th>
<th>4th Candidate</th>
<th>[Other]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(y_{1, 1})</td>
<td>(y_{1, 2})</td>
<td>(y_{1, 3})</td>
<td>(y_{1, 4})</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>(y_{2, 1})</td>
<td>(y_{2, 2})</td>
<td>(y_{2, 3})</td>
<td>(y_{2, 4})</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>(y_{3, 1})</td>
<td>(y_{3, 2})</td>
<td>(y_{3, 3})</td>
<td>(y_{3, 4})</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>(y_{4, 1})</td>
<td>(y_{4, 2})</td>
<td>(y_{4, 3})</td>
<td>(y_{4, 4})</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>(y_{5, 1})</td>
<td>(y_{5, 2})</td>
<td>(y_{5, 3})</td>
<td>(y_{5, 4})</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>(y_{6, 1})</td>
<td>(y_{6, 2})</td>
<td>(y_{6, 3})</td>
<td>(y_{6, 4})</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>(y_{7, 1})</td>
<td>(y_{7, 2})</td>
<td>(y_{7, 3})</td>
<td>(y_{7, 4})</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>(y_{8, 1})</td>
<td>(y_{8, 2})</td>
<td>(y_{8, 3})</td>
<td>(y_{8, 4})</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>(y_{9, 1})</td>
<td>(y_{9, 2})</td>
<td>(y_{9, 3})</td>
<td>(y_{9, 4})</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>(y_{10, 1})</td>
<td>(y_{10, 2})</td>
<td>(y_{10, 3})</td>
<td>(y_{10, 4})</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\[
y = \text{Concat}_{l=1}^{L} y_{lc_l} \quad \text{score}(y) = \prod_{l=1}^{L} p_{lc_l}
\]
Baseline Top-1 Combination

Exact top-K solutions can be generated in $O(KL \log K)$ time.

<table>
<thead>
<tr>
<th>Line number $l$</th>
<th>Best Candidate</th>
<th>2nd Candidate</th>
<th>3rd Candidate</th>
<th>4th Candidate</th>
<th>[Other]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$y_1, 1$</td>
<td>$y_{1,2}$</td>
<td>$y_{1,3}$</td>
<td>$y_{1,4}$</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>$y_2, 1$</td>
<td>$y_{2,2}$</td>
<td>$y_{2,3}$</td>
<td>$y_{2,4}$</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>$y_3, 1$</td>
<td>$y_{3,2}$</td>
<td>$y_{3,3}$</td>
<td>$y_{3,4}$</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>$y_4, 1$</td>
<td>$y_{4,2}$</td>
<td>$y_{4,3}$</td>
<td>$y_{4,4}$</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>$y_5, 1$</td>
<td>$y_{5,2}$</td>
<td>$y_{5,3}$</td>
<td>$y_{5,4}$</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>$y_6, 1$</td>
<td>$y_{6,2}$</td>
<td>$y_{6,3}$</td>
<td>$y_{6,4}$</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>$y_7, 1$</td>
<td>$y_{7,2}$</td>
<td>$y_{7,3}$</td>
<td>$y_{7,4}$</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>$y_8, 1$</td>
<td>$y_{8,2}$</td>
<td>$y_{8,3}$</td>
<td>$y_{8,4}$</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>$y_9, 1$</td>
<td>$y_{9,2}$</td>
<td>$y_{9,3}$</td>
<td>$y_{9,4}$</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>$y_{10, 1}$</td>
<td>$y_{10,2}$</td>
<td>$y_{10,3}$</td>
<td>$y_{10,4}$</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

$$y = \text{Concat}_{l=1}^{L} y_{lc_l} \quad \text{score}(y) = \prod_{l=1}^{L} p_{lc_l}$$
<table>
<thead>
<tr>
<th>Method</th>
<th>Top 1</th>
<th>Top 10</th>
<th>Top 100</th>
<th>Top 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.0%</td>
<td>8.1%</td>
<td>29.2%</td>
<td>44.3%</td>
</tr>
</tbody>
</table>
Baseline Ignores Dependencies

- **Syntactic dependency and constraints between lines.**
  - Result code needs to be a legal derivation of the Abstract Syntax Tree.
  - Pseudocode does not specify this detail whether to delay scope opening “{ ” to the next line.
  - “Enumerate index i of the array s” can be translated to:
    - ```java
        for (int i = 0; i < s.length(); i++) {
            for (int i = 0; i < s.length(); i++)
        }
    ```

- **Semantic dependency and constraints between lines.**
  - Cannot use an undeclared variable or redeclare a declared variable in the same scope.
  - Pseudocode does not explicitly state variable usage and declaration.
  - “Set Ch to be a”
    - ```java
        char Ch = ‘a’;
        char Ch = a;
        Ch = ‘a’;
    ```

\[ y = \text{Concat}_{l=1}^{L} y_{l+1} \quad \text{score}(y) = \prod_{l=1}^{L} p_{l+1} \]
Baseline Ignores Dependencies

- **Syntactic dependency and constraints between lines.**
  - The full program needs to be a legal derivation of the Abstract Syntax Tree.
  - Pseudocode does not specify whether to delay scope opening “{” to the next line.
  - “Enumerate index i of the array s” can be translated to:
    - for (int i = 0; i < s.length(); i++) {
    - for (int i = 0; i < s.length(); i++)

- **Semantic dependency and constraints between lines.**
  - Cannot use an undeclared variable or redefine a declared variable in the same scope.
  - Pseudocode does not explicitly state variable usage and declaration.
  - “Set Ch to be a”
    - char Ch = ‘a’;
    - char Ch = a;
    - Ch = ‘a’;

\[
y = \text{Concat}_{l=1}^{L} y_{lc_l} \quad \text{score}(y) = \prod_{l=1}^{L} p_{lc_l}
\]
Baseline Ignores Dependencies

- **Syntactic dependency and constraints between lines.**
  - The full program needs to be a legal derivation of AST.
  - Pseudocode does not specify whether to delay scope opening “{” to the next line.
  - “Enumerate index i of the array s” can be translated to:
    - \( \text{for} \ (\text{int} \ i = 0; \ i < \text{s.length()}; \ i++) \) \{ \\
    - \( \text{for} \ (\text{int} \ i = 0; \ i < \text{s.length()}; \ i++) \)

- **Semantic dependency and constraints between lines.**
  - Cannot use an undeclared variable or redeclare a declared variable in the same scope.
  - Pseudocode does not explicitly state variable usage and declaration.
  - “Set Ch to be a”
    - char Ch = ‘a’;
    - char Ch = a;
    - Ch = ‘a’;

\[ y = \text{Concat}_{l=1}^{L} y_{lc_l} \quad \text{score}(y) = \prod_{l=1}^{L} p_{lc_l} \]
int main () {
    int n, ans = 1;
    for (int i = 1; i <= n / 2; i++) {
        . . .
    }
}

[return type] [function name] ( ) {
    [statement] ;
    for [for conditions] {
        . . .
    }
}
We use an incremental parser to check whether the high level symbol combination is a possible derivation of the AST grammar.
Baseline Ignores Dependencies

- **Syntactic dependency and constraints between lines.**
  - The full program needs to be a legal derivation of AST.
  - Pseudocode does not specify whether to delay scope opening “{” to the next line.
  - “Enumerate index i of the array s” can be translated to:
    - for (int i = 0; i < s.length(); i++) {
    - for (int i = 0; i < s.length(); i++)

- **Semantic dependency and constraints between lines.**
  - Cannot use an undeclared variable or redeclare a declared variable in the same scope.
  - Pseudocode does not explicitly state variable usage and declaration.
  - “Set Ch to be a”
    - char Ch = ‘a’;
    - char Ch = a;
    - Ch = ‘a’;

\[ y = \text{Concat}_{l=1}^{L} y_{lc_i} \quad \text{score}(y) = \prod_{l=1}^{L} p_{lc_i} \]
Baseline Ignores Dependencies

- **Syntactic dependency and constraints between lines.**
  - The full program needs to be a legal derivation of AST.
  - Pseudocode does not specify whether to delay scope opening “{” to the next line.
  - “Enumerate index i of the array s” can be translated to:
    - for (int i = 0; i < s.length(); i++) {
    - for (int i = 0; i < s.length(); i++)

- **Semantic dependency and constraints between lines.**
  - Cannot use an undeclared variable or redeclare a declared variable in the same scope.
  - Pseudocode does not explicitly state variable usage and declaration.
  - “Set Ch to be a”
    - char Ch = ‘a’;
    - char Ch = a;
    - Ch = ‘a’;
# Semantic Constraints

## Parse each fragment

<table>
<thead>
<tr>
<th>Fragments</th>
<th>Variables Used and Declared</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>int main () {}</strong></td>
<td>declared: main used:</td>
</tr>
<tr>
<td>int n, ans = 1;</td>
<td>declared: n, ans used:</td>
</tr>
<tr>
<td>for (int i = 1; i &lt;= n / 2; i++) {}</td>
<td>declared: i used: n</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Semantic Constraints

Parse each fragment

<table>
<thead>
<tr>
<th>Fragments</th>
<th>Variables Used and Declared</th>
</tr>
</thead>
<tbody>
<tr>
<td>int main () {</td>
<td>declared: main used:</td>
</tr>
<tr>
<td>int n, ans = 1;</td>
<td>declared: n, ans used:</td>
</tr>
<tr>
<td>for (int i = 1; i &lt;= n / 2; i++) {</td>
<td>declared: i used: n</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

We keep a variable table to check whether any fragment re-declares a declared variable, or uses an undeclared variable.
1. To form one program candidate, we select one column for each row; the score is given by the equation below.
2. We want to find the top-K scoring candidates that satisfy the syntactic and semantic constraints.
3. We can efficiently check whether the first $l$ fragment combination is valid under these constraints.

$$y = \text{Concat}_{l=1}^{L} y_{lc_l} \quad \text{score}(y) = \prod_{l=1}^{L} p_{lc_l}$$
## Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Top 1</th>
<th>Top 10</th>
<th>Top 100</th>
<th>Top 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.0%</td>
<td>8.1%</td>
<td>29.2%</td>
<td>44.3%</td>
</tr>
<tr>
<td>Beam Syntactic</td>
<td>42.4%↑</td>
<td>51.3%↑</td>
<td>58.2%↑</td>
<td>N/A</td>
</tr>
<tr>
<td>Beam Semantic</td>
<td>45.6%↑</td>
<td>54.9%↑</td>
<td>61.9%↑</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Adding constraints significantly improve over the baseline; semantic constraints also improve over syntactic constraints.
## Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Top 1</th>
<th>Top 10</th>
<th>Top 100</th>
<th>Top 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.0%</td>
<td>8.1%</td>
<td>29.2%</td>
<td>44.3%</td>
</tr>
<tr>
<td>Beam Syntactic</td>
<td>42.4%†</td>
<td>51.3%†</td>
<td>58.2%†</td>
<td>N/A</td>
</tr>
<tr>
<td>Beam Semantic</td>
<td>45.6%†</td>
<td>54.9%†</td>
<td>61.9%†</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Time complexity of beam search grows **quadratically** with beam width (\(\geq K\)), which becomes intractable if top-1000 candidate is considered.
Motivation for Scaffold

- Baseline assumes independence between lines, which makes top-K generation **fast and exact**.
- Beam search deals with the inherent dependence between lines and **satisfies the constraint**.
- Scaffold search: first search the line dependency structure, then independently generate each line.
Baseline assumes independence between lines, which makes top-K generation fast and exact.

Beam search deals with the inherent dependence between lines and satisfies the constraint.

Scaffold search: first search the line dependency structure, then independently generate each line.
Configurations and Scaffold

Fragments → Parse each fragment → Syntactic Configurations $\phi$

```plaintext
int main () {
    int n, ans = 1;
    for (int i = 1; i <= n / 2; i++) {
        ...
    }
    return ...
}
```

Fragments → Parse each fragment → Semantic Configurations $\phi$

```plaintext
int main () {
    int n, ans = 1;
    for (int i = 1; i <= n / 2; i++) {
        ...
    }
    declared: main used:
    declared: n, ans used:
    declared: i used: n
    ...
```
Configurations and Scaffold

### Syntactic Configurations $\phi$

```c
int main () {
    int n, ans = 1;
    for (int i = 1; i <= n / 2; i++) {
        ...  
    }
}
```

### Semantic Configurations $\phi$

```c
int main () {
    int n, ans = 1;
    for (int i = 1; i <= n / 2; i++) {
        ...  
    }
}
```

Combine configuration from each line to form a program scaffold.
Valid Scaffold

```plaintext
[return type] [function name] ( ) { declared: main used:
[statement] ; declared: n, ans used:
for [for conditions] { declared: i used: n
  ...
...
}
```

Invalid Scaffold

```plaintext
[return type] [function name] ( ) { declared: main used:
[statement] ; declared: ans used:
for [for conditions] { declared: i used: n
  ...
...
}
```

Invalid Scaffold

```plaintext
[return type] [function name] ( ) { declared: main used:
[statement] ; declared: ans used:
for [for conditions] { declared: i used: n
  ...
...
}
```

We can efficiently check whether the combination of the first $l$ configuration satisfies the constraint.
Scoring Configs and Scaffolds

\[
p_{1,1}: 0.8 \quad | \quad p_{1,1}: 0.1 \quad | \quad p_{1,1}: 0.03 \quad | \quad p_{1,1}: 0.01 \quad | \quad \ldots
\]

\[
y_{1,1}: i += 1; \quad | \quad y_{1,2}: \text{int } i = 1; \quad | \quad y_{1,3}: \text{int } i; \quad | \quad y_{1,4}: i -= 1; \quad | \quad \ldots
\]

The score of a scaffold is the product of the probability scores for each configuration that form the scaffold.

\[p = 0.81, \phi_{1,1}\]

statement; variable used: None; declared \(i\).

\[p = 0.13, \phi_{1,2}\]

statement; variable used: \(i\); declared: None.

Marginalize code fragments to assign scores to configurations.
# Beam Search for Scaffold

1. To form one scaffold, we select one config for each row; the score is the product of the config scores.
2. We want to find the top-50 scoring scaffolds that satisfy the syntactic and semantic constraints.
3. We can efficiently check whether the first $l$ config combination is valid under these constraints.

<table>
<thead>
<tr>
<th>Line number $l$</th>
<th>Best Config</th>
<th>2nd Config</th>
<th>3rd Config</th>
<th>4th Config</th>
<th>[Other]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\phi_{1, 1}$</td>
<td>$\phi_{1, 2}$</td>
<td>$\phi_{1, 3}$</td>
<td>$\phi_{1, 4}$</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>$\phi_{2, 1}$</td>
<td>$\phi_{2, 2}$</td>
<td>$\phi_{2, 3}$</td>
<td>$\phi_{2, 4}$</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>$\phi_{3, 1}$</td>
<td>$\phi_{3, 2}$</td>
<td>$\phi_{3, 3}$</td>
<td>$\phi_{3, 4}$</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>$\phi_{4, 1}$</td>
<td>$\phi_{4, 2}$</td>
<td>$\phi_{4, 3}$</td>
<td>$\phi_{4, 4}$</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>$\phi_{5, 1}$</td>
<td>$\phi_{5, 2}$</td>
<td>$\phi_{5, 3}$</td>
<td>$\phi_{5, 4}$</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>$\phi_{6, 1}$</td>
<td>$\phi_{6, 2}$</td>
<td>$\phi_{6, 3}$</td>
<td>$\phi_{6, 4}$</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>$\phi_{7, 1}$</td>
<td>$\phi_{7, 2}$</td>
<td>$\phi_{7, 3}$</td>
<td>$\phi_{7, 4}$</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>$\phi_{8, 1}$</td>
<td>$\phi_{8, 2}$</td>
<td>$\phi_{8, 3}$</td>
<td>$\phi_{8, 4}$</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>$\phi_{9, 1}$</td>
<td>$\phi_{9, 2}$</td>
<td>$\phi_{9, 3}$</td>
<td>$\phi_{9, 4}$</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>$\phi_{10, 1}$</td>
<td>$\phi_{10, 2}$</td>
<td>$\phi_{10, 3}$</td>
<td>$\phi_{10, 4}$</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
1. Given a scaffold, we only consider code fragment candidates that agree with the configuration for each line.
2. Any code fragment combination satisfies the constraint, if the scaffold is valid.
3. We can efficiently generate top-K candidates given a scaffold.
## Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Top 1</th>
<th>Top 10</th>
<th>Top 100</th>
<th>Top 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.0%</td>
<td>8.1%</td>
<td>29.2%</td>
<td>44.3%</td>
</tr>
<tr>
<td>Beam Syntactic</td>
<td>42.4%</td>
<td>51.3%</td>
<td>58.2%</td>
<td>N/A</td>
</tr>
<tr>
<td>Beam Semantic</td>
<td>45.6%</td>
<td>54.9%</td>
<td>61.9%</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Scaffold Semantic</strong></td>
<td><strong>45.8%†</strong></td>
<td><strong>55.3%†</strong></td>
<td><strong>62.8%†</strong></td>
<td><strong>67.6%</strong></td>
</tr>
</tbody>
</table>

### Notes:
1. With scaffold search, we obtain better performance with ~16x less compute.
2. Scaffold search allows us to calculate the performance of top-1000 candidates.
Conclusion

- We propose a method to efficiently generate 1000 candidates for long programs and search for a semantically correct solution.
- We need semantic constraints as well to improve performance.
- Scaffold search improves both search qualities and efficiency.
Thank you!


Code: https://github.com/ruiqi-zhong/SemanticScaffold