Semantic Scaffolds for Pseudocode-to-Code Generation

Berkeley
N L P

Ruiqi Zhong, Mitchell Stern, Dan Klein

Online Judges

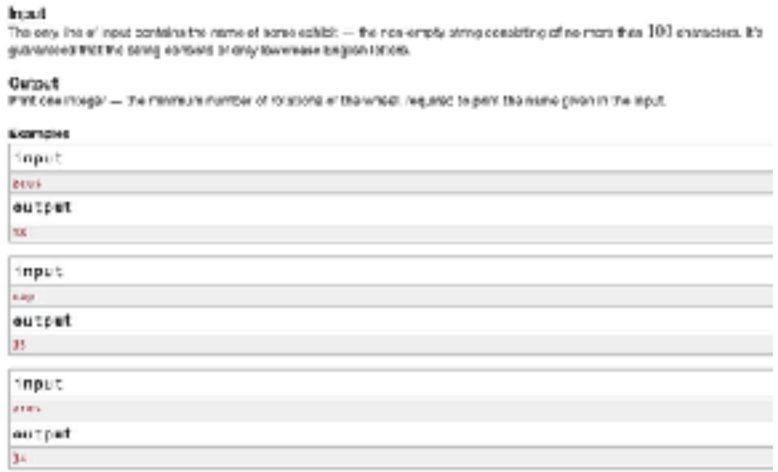
LeetCode, Hackerrank, Codeforce, etc.

Coding Challenge Description

Code

```
#!/bin/python

import math
import os
import random
import re
import sys

# Complete the miniMaxSum function below.
def miniMaxSum(arr):

if __name__ == '__main__':
    arr = map(int, raw_input().rstrip().split())

    miniMaxSum(arr)
```

Input-output Test Cases



Task

Generate

Pseudocode

```
declare constant integer numOfAlphabets = 26

    create string s
    let Count = 0 be an integer
    read s
    Set Ch to be a
    for i = 0 to i less than the length of s
        set Count to Count + min(abs(s[i] - Ch),
            numOfAlphabets - abs(s[i] - Ch))
        set s[i] to Ch

    print Count


```

Code

```
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;
}
```

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

Language Ambiguities

Pseudocode

```
declare constant integer numOfAlphabets = 26

    create string s
    let Count = 0 be an integer
    read s
    Set Ch to be a
    for i = 0 to i less than the length of s
        set Count to Count + min(abs(s[i] - Ch),
            numOfAlphabets - abs(s[i] - Ch))
        set s[i] to Ch

    print Count
```

Code

```
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;
}
```

Several Possibilities

```
char Ch = 'a';
char Ch = a;
Ch = 'a';
Ch = a;
```

Evaluation

Pseudocode

```
declare constant integer numOfAlphabets = 26

    create string s
    let Count = 0 be an integer
    read s
    Set Ch to be a
    for i = 0 to i less than the length of s
        set Count to Count + min(abs(s[i] - Ch),
            numOfAlphabets - abs(s[i] - Ch))
        set s[i] to Ch

    print Count
```

Top-k Code Candidate

```
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.

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.

	l	Pseudocode x_l
	3	create string s
	4	let Count = 0 be an integer
	5	read s
x_6	6	Set Ch to be a

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.

	l	Pseudocode x_l
	3	create string s
	4	let Count = 0 be an integer
	5	read s
x_6	6	Set Ch to be a

p_{61} = 0.70, y_{61} `char Ch = 'a';`
p_{62} = 0.15, y_{62} `char Ch = a;`
p_{63} = 0.05, y_{63} `Ch = 'a';`
p_{64} = 0.02, y_{64} `Ch = a;`
. . .
Others omitted,
altogether 100 of candidates.

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.

$$y = Concat_{l=1}^{L} y_{lc_l} \qquad score(y) = \prod_{l=1}^{L} p_{lc_l}$$

	l	Pseudocode x_l
	3	create string s
	4	let Count = 0 be an integer
	5	read s
x_6	6	Set Ch to be a

p_{61} = 0.70, y_{61} `char Ch = 'a';`
p_{62} = 0.15, y_{62} `char Ch = a;`
p_{63} = 0.05, y_{63} `Ch = 'a';`
p_{64} = 0.02, y_{64} `Ch = a;`

. . .

Others omitted,
altogether 100 of candidates.

An Abstract View

Select one column (fragment) for each row (line) to form a full program.

Line number l	Best Candidate	2nd Candidate	3rd Candidate	4th Candidate	[Other]
1	$y_{1,1}$	$y_{1,2}$	$y_{1,3}$	$y_{1,4}$	...
2	$y_{2,1}$	$y_{2,2}$	$y_{2,3}$	$y_{2,4}$	...
3	$y_{3,1}$	$y_{3,2}$	$y_{3,3}$	$y_{3,4}$	...
4	$y_{4,1}$	$y_{4,2}$	$y_{4,3}$	$y_{4,4}$	...
5	$y_{5,1}$	$y_{5,2}$	$y_{5,3}$	$y_{5,4}$	...
6	$y_{6,1}$	$y_{6,2}$	$y_{6,3}$	$y_{6,4}$	...
7	$y_{7,1}$	$y_{7,2}$	$y_{7,3}$	$y_{7,4}$	...
8	$y_{8,1}$	$y_{8,2}$	$y_{8,3}$	$y_{8,4}$	...
9	$y_{9,1}$	$y_{9,2}$	$y_{9,3}$	$y_{9,4}$	...
10	$y_{10,1}$	$y_{10,2}$	$y_{10,3}$	$y_{10,4}$	...
. . .	. . .	. . .	. . .	...	...

$$y = Concat_{l=1}^{L} y_{lc_l} \qquad score(y) = \prod_{l=1}^{L} p_{lc_l}$$

Baseline Top-1 Combination

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

Line number l	Best Candidate	2nd Candidate	3rd Candidate	4th Candidate	[Other]
1	$y_{1,1}$	$y_{1,2}$	$y_{1,3}$	$y_{1,4}$	...
2	$y_{2,1}$	$y_{2,2}$	$y_{2,3}$	$y_{2,4}$	...
3	$y_{3,1}$	$y_{3,2}$	$y_{3,3}$	$y_{3,4}$	...
4	$y_{4,1}$	$y_{4,2}$	$y_{4,3}$	$y_{4,4}$	...
5	$y_{5,1}$	$y_{5,2}$	$y_{5,3}$	$y_{5,4}$	...
6	$y_{6,1}$	$y_{6,2}$	$y_{6,3}$	$y_{6,4}$	...
7	$y_{7,1}$	$y_{7,2}$	$y_{7,3}$	$y_{7,4}$	...
8	$y_{8,1}$	$y_{8,2}$	$y_{8,3}$	$y_{8,4}$	...
9	$y_{9,1}$	$y_{9,2}$	$y_{9,3}$	$y_{9,4}$	...
10	$y_{10,1}$	$y_{10,2}$	$y_{10,3}$	$y_{10,4}$	...
. . .	. . .	. . .	. . .	...	...

$$y = Concat_{l=1}^{L} y_{lc_l} \qquad score(y) = \prod_{l=1}^{L} p_{lc_l}$$

Performance

Method	Top 1	Top 10	Top 100	Top 1000
Baseline	0.0%	8.1%	29.2%	44.3%

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:
    - `for (int i = 0; i < s.length(); i++) {`
    - `for (int i = 0; i < s.length(); i++)`

$$y = Concat_{l=1}^{L} y_{lc_l} \qquad score(y) = \prod_{l=1}^{L} p_{lc_l}$$

- 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’;`

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++)`

$$y = Concat_{l=1}^{L} y_{lc_l} \qquad score(y) = \prod_{l=1}^{L} p_{lc_l}$$

- 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’;`

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++)

$$y = Concat_{l=1}^{L} y_{lc_l} \qquad score(y) = \prod_{l=1}^{L} p_{lc_l}$$

- 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';

Syntactic Constraints

Parse each fragment

Fragments

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

“High level symbols”

[*return type*] [*function name*] ( ) {

  [*statement*] ;

  for [*for conditions*] {

  . . .

Syntactic Constraints

Parse each fragment

Fragments

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

"High level symbols"

[*return type*] [*function name*] ( ) {

  [*statement*] ;

  for [*for conditions*] {

  . . .

[*function*]

[*return type*] [*function name*] ([*arguments*]*) { [*statement*]* }

[*statement*]

for [*for condition*] { [*statement*]* }

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++)

$$y = Concat_{l=1}^{L} y_{lc_l} \qquad score(y) = \prod_{l=1}^{L} p_{lc_l}$$

- 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’;

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++)`

$$y = Concat_{l=1}^{L} y_{lc_l} \qquad score(y) = \prod_{l=1}^{L} p_{lc_l}$$

- 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

Fragments	Variables Used and Declared	
`int main () {`	`declared: main`	`used:`
`int n, ans = 1;`	`declared: n, ans`	`used:`
`for (int i = 1; i <= n / 2; i++) {`	`declared: i`	`used: n`
`…`		

Semantic Constraints

Parse each fragment

Fragments	Variables Used and Declared
`int main () {`	`declared: main    used:`
`    int n, ans = 1;`	`declared: n, ans  used:`
`    for (int i = 1; i <= n / 2; i++) {`	`declared: i       used: n`
`    …`	

We keep a variable table to check whether any fragment re-declares a declared variable, or uses an undeclared variable.

Beam Search

Line number l	Best Candidate	2nd Candidate	3rd Candidate	4th Candidate	[Other]
1	$y_{1,1}$	$y_{1,2}$	$y_{1,3}$	$y_{1,4}$	...
2	$y_{2,1}$	$y_{2,2}$	$y_{2,3}$	$y_{2,4}$	...
3	$y_{3,1}$	$y_{3,2}$	$y_{3,3}$	$y_{3,4}$	...
4	$y_{4,1}$	$y_{4,2}$	$y_{4,3}$	$y_{4,4}$	...
5	$y_{5,1}$	$y_{5,2}$	$y_{5,3}$	$y_{5,4}$	...
6	$y_{6,1}$	$y_{6,2}$	$y_{6,3}$	$y_{6,4}$	...
7	$y_{7,1}$	$y_{7,2}$	$y_{7,3}$	$y_{7,4}$	...
8	$y_{8,1}$	$y_{8,2}$	$y_{8,3}$	$y_{8,4}$	...
9	$y_{9,1}$	$y_{9,2}$	$y_{9,3}$	$y_{9,4}$	...
10	$y_{10,1}$	$y_{10,2}$	$y_{10,3}$	$y_{10,4}$	...
...	...	...	...	...	...

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.

$$y = Concat_{l=1}^{L} y_{lc_l} \qquad score(y) = \prod_{l=1}^{L} p_{lc_l}$$

Performance

Method	Top 1	Top 10	Top 100	Top 1000
Baseline	0.0%	8.1%	29.2%	44.3%
Beam Syntactic	42.4%↑	51.3%↑	58.2%↑	N/A
Beam Semantic	45.6%↑	54.9%↑	61.9%↑	N/A

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

Performance

Method	Top 1	Top 10	Top 100	Top 1000
Baseline	0.0%	8.1%	29.2%	44.3%
Beam Syntactic	42.4%↑	51.3%↑	58.2%↑	N/A
Beam Semantic	45.6%↑	54.9%↑	61.9%↑	N/A

Time complexity of beam search grows **quadratically** with beam width (>= 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.

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.

Configurations and Scaffold

Parse each fragment →

Fragments	**Syntactic Configurations ϕ**
`int main () {`	`[`*`return type`*`] [`*`function name`*`] ( ) {`
`    int n, ans = 1;`	`[`*`statement`*`] ;`
`    for (int i = 1; i <= n / 2; i++) {`	`for [`*`for conditions`*`] {`

…

Parse each fragment →

Fragments	**Semantic Configurations ϕ**
`int main () {`	`declared: main     used:`
`    int n, ans = 1;`	`declared: n, ans  used:`
`    for (int i = 1; i <= n / 2; i++) {`	`declared: i        used: n`

…

Configurations and Scaffold

Parse each fragment

Fragments

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

Syntactic Configurations ϕ

[*return type*] [*function name*] ( ) {
[*statement*] ;
for [*for conditions*] {

Parse each fragment

Fragments

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

Semantic Configurations ϕ

declared: main	used:
declared: n, ans	used:
declared: i	used: n

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

Combine configuration from each line to form a program scaffold.

Example Scaffolds

Valid Scaffold

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

Invalid Scaffold

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

Use of undeclared variable *n*

Invalid Scaffold

[*return type*] [*function name*] ( ) $\boxed{}$	declared: main	used:
[*statement*] ;	declared: ans	used:
for [*for conditions*] {	declared: i	used: n
. . .	. . .	

Missing "{"

We can efficiently check whether the combination of the first *l* configuration satisfies the constraint.

Scoring Configs and Scaffolds

$p_{1,1}$: `0.8`	$p_{1,1}$: `0.1`	$p_{1,1}$: `0.03`	$p_{1,1}$: `0.01`	. . .
$y_{1,1}$: `i += 1;`	$y_{1,2}$: `int i = 1;`	$y_{1,3}$: `int i;`	$y_{1,4}$: `i -= 1;`	. . .

Marginalize code fragments to assign scores to configurations

p = 0.81, $\phi_{1,1}$

statement; variable used: None; declared *i*.

p = 0.13, $\phi_{1,2}$

statement; variable used: *i*; declared: None.

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

Beam Search for Scaffold

Line number l	Best Config	2nd Config	3rd Config	4th Config	[Other]
1	$\phi_{1,1}$	$\phi_{1,2}$	$\phi_{1,3}$	$\phi_{1,4}$	...
2	$\phi_{2,1}$	$\phi_{2,2}$	$\phi_{2,3}$	$\phi_{2,4}$	...
3	$\phi_{3,1}$	$\phi_{3,2}$	$\phi_{3,3}$	$\phi_{3,4}$	...
4	$\phi_{4,1}$	$\phi_{4,2}$	$\phi_{4,3}$	$\phi_{4,4}$	...
5	$\phi_{5,1}$	$\phi_{5,2}$	$\phi_{5,3}$	$\phi_{5,4}$	...
6	$\phi_{6,1}$	$\phi_{6,2}$	$\phi_{6,3}$	$\phi_{6,4}$	...
7	$\phi_{7,1}$	$\phi_{7,2}$	$\phi_{7,3}$	$\phi_{7,4}$	...
8	$\phi_{8,1}$	$\phi_{8,2}$	$\phi_{8,3}$	$\phi_{8,4}$	...
9	$\phi_{9,1}$	$\phi_{9,2}$	$\phi_{9,3}$	$\phi_{9,4}$	...
10	$\phi_{10,1}$	$\phi_{10,2}$	$\phi_{10,3}$	$\phi_{10,4}$	...
. . .	. . .	. . .	. . .	...	...

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.

Scaffold to Code

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

```
int n, ans;
int n, ans = 0;
```

```
for (int i = 0; i < n; i++) {
for (int i = 0; i <= n - 1; i++) {
for (int i = n; i > 1; i–) {
```

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

	Method	Top 1	Top 10	Top 100	Top 1000
	Baseline	0.0%	8.1%	29.2%	44.3%
	Beam Syntactic	42.4%	51.3%	58.2%	N/A
Beam size 200	Beam Semantic	45.6%	54.9%	61.9%	N/A
Beam size 50	Scaffold Semantic	45.8%↑	55.3%↑	62.8%↑	67.6%

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!

Paper: https://arxiv.org/abs/2005.05927

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

Berkeley

N L P