

Programming by Example

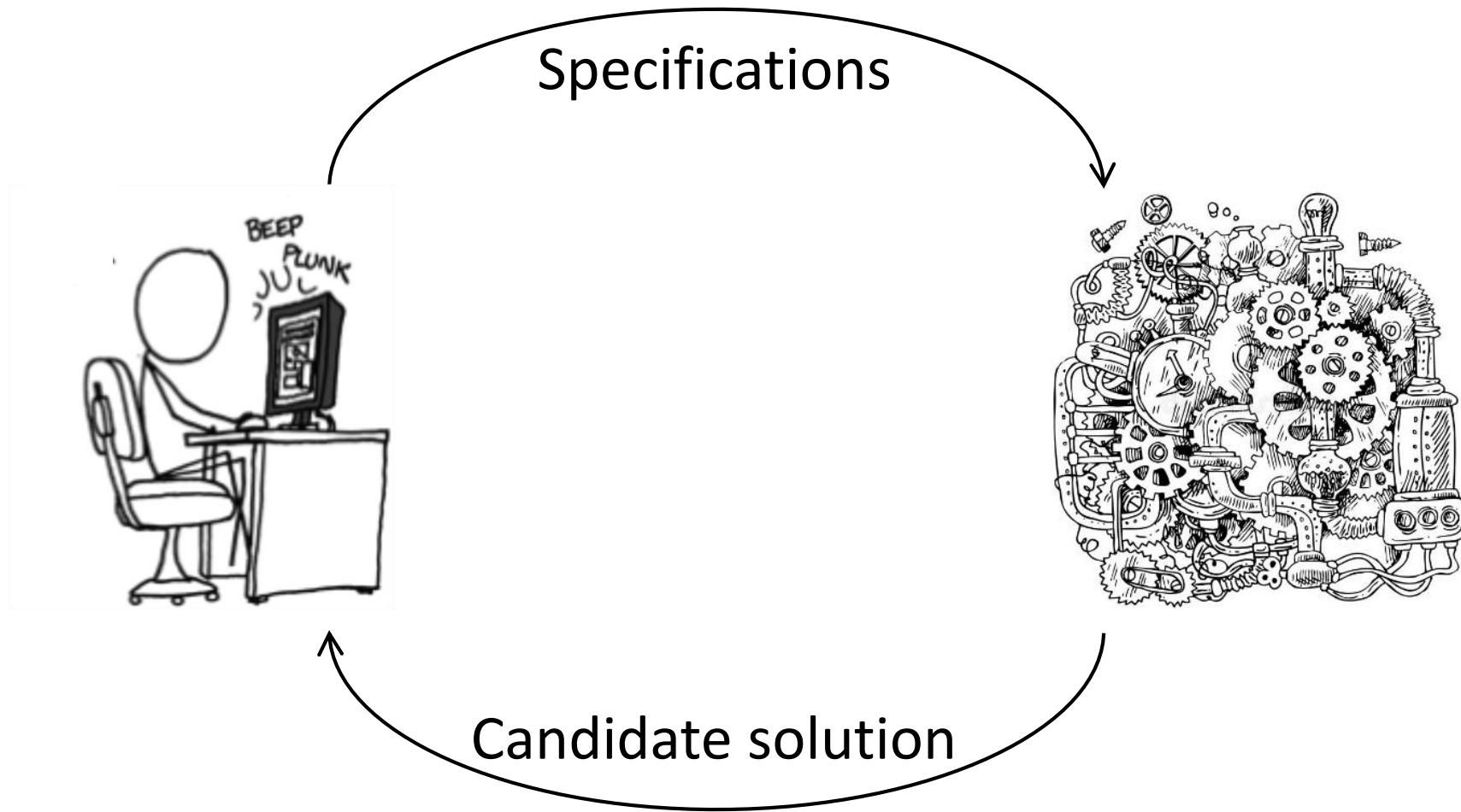
Hila Peleg

Sharon Shoham

Eran Yahav

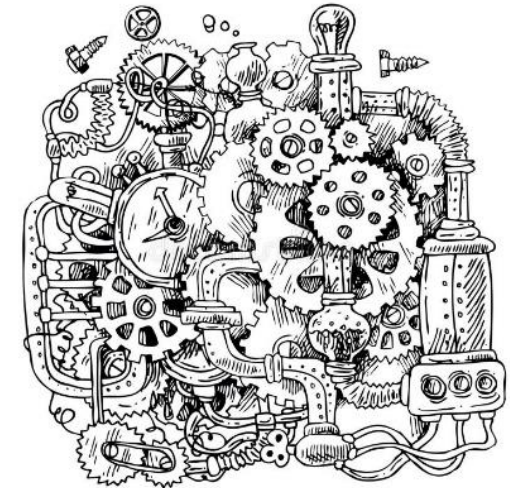
The research leading to these results has received funding from the European Union's - Seventh Framework Programme (FP7) under grant agreement n° 615688 – ERC- COG-PRIME.

Program Synthesis



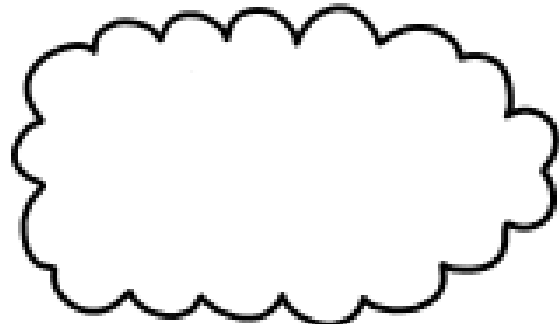
Programming by Example

$$\mathcal{E} = \{ \text{"a**bd**fibfcfde**bd**"} \mapsto \text{"bd"} \}$$



$$p \text{ s.t. } \forall (l, \omega) \in \mathcal{E}. \llbracket p \rrbracket (l) = \omega$$

Many examples are inherently ambiguous



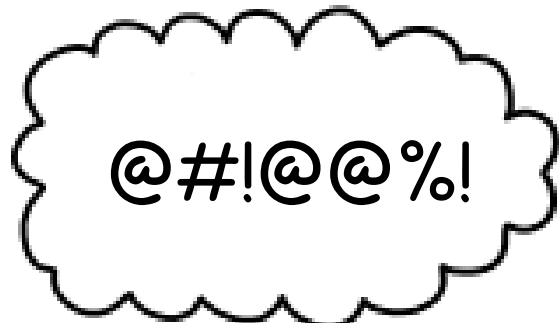
Input:

"a**bd**fibfcfde**bd**fde**bd**ihgfkj fde**bd**"

Output: "bd"

bd	-	4
de	-	3
eb	-	3
df	-	2
hg	-	1

Many examples are inherently ambiguous



Input:

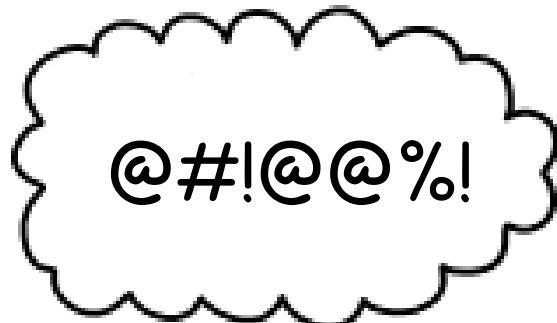
"a**bd**fibfcfde**bd**fde**bd**ihgfkj fde**bd**"

Output: "bd"

```
input.takeRight(2)
```

bd	-	4
de	-	3
eb	-	3
df	-	2
hg	-	1

Many examples are inherently ambiguous



@#!@@%!



Input:

"a**bd**fibfcfde**bd**fde**bd**ihgfkj fde**bd**"

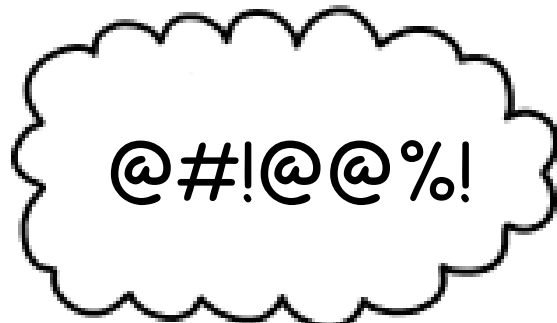
Output: "bd"

```
input.takeRight(2)
```

bd	-	4
de	-	3
eb	-	3
df	-	2
hg	-	1

- PBE aims for consistency with examples
- Examples don't convey intent uniquely

Many examples are inherently ambiguous



@#!@@%!



Input:

"a**bd**fibfcfde**bd**fde**bd**ihgfkj fde**bd**"

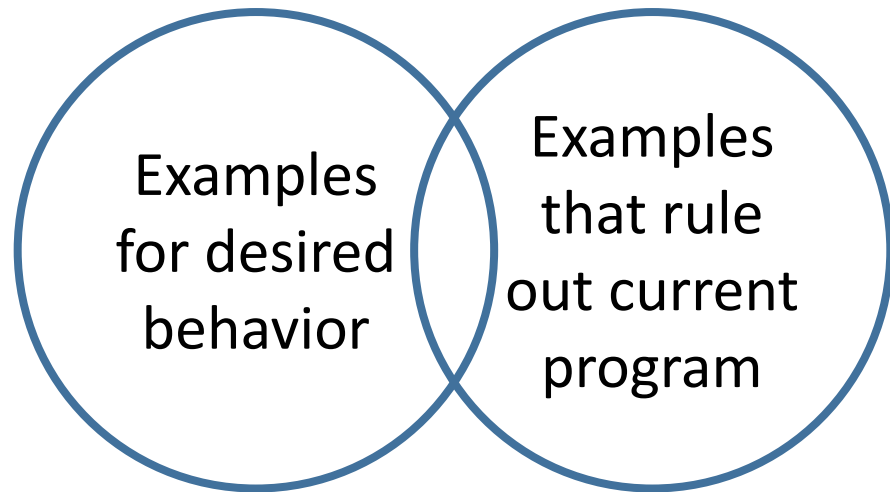
Output: "bd"

```
input.takeRight(2)
```

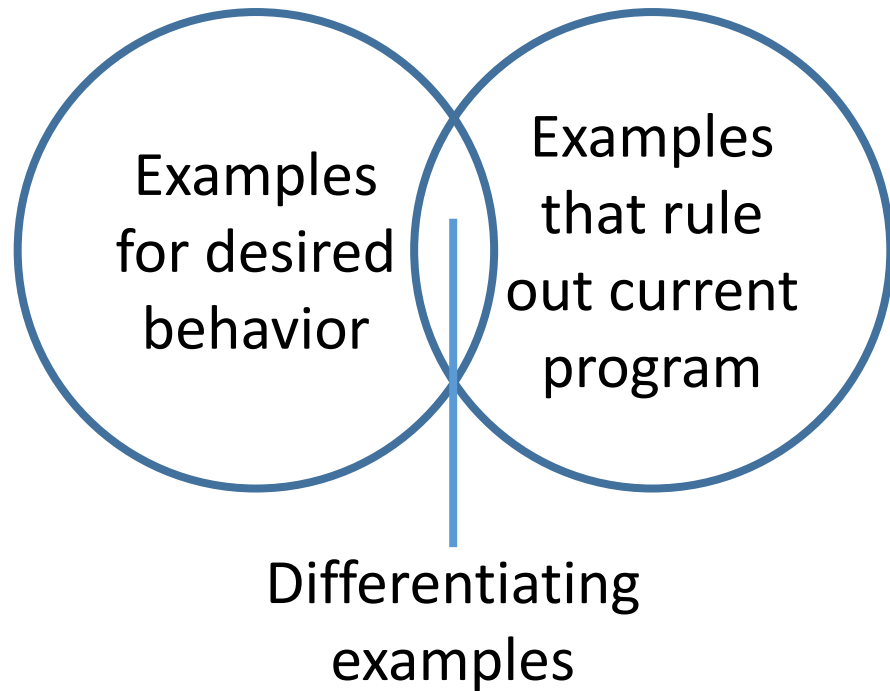
bd	-	4
de	-	3
eb	-	3
df	-	2
hg	-	1

- PBE aims for consistency with examples
- Examples don't convey intent uniquely
- In other words: **overfitting**

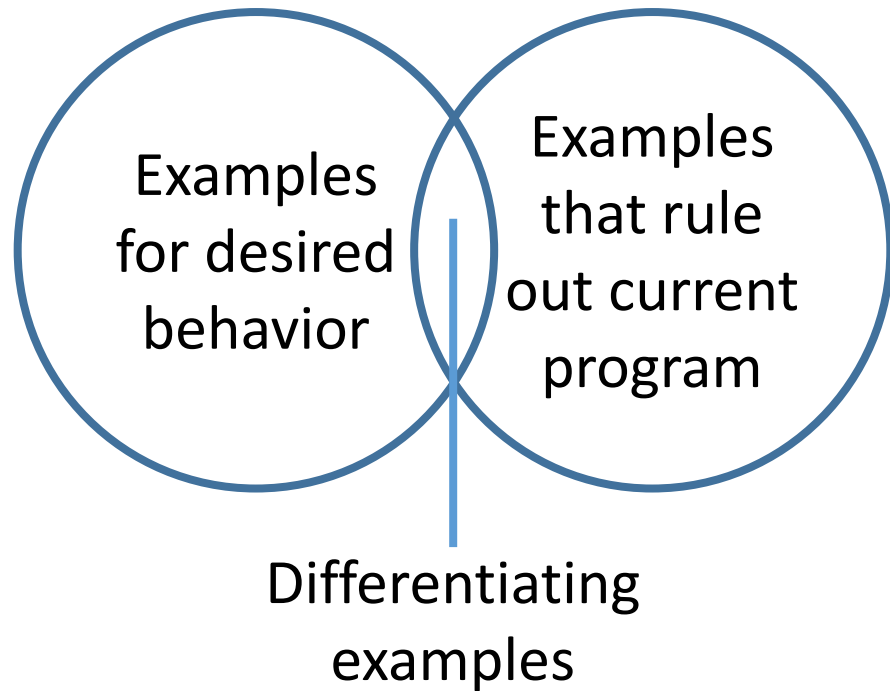
Problem 1: Differentiating examples are hard



Problem 1: Differentiating examples are hard

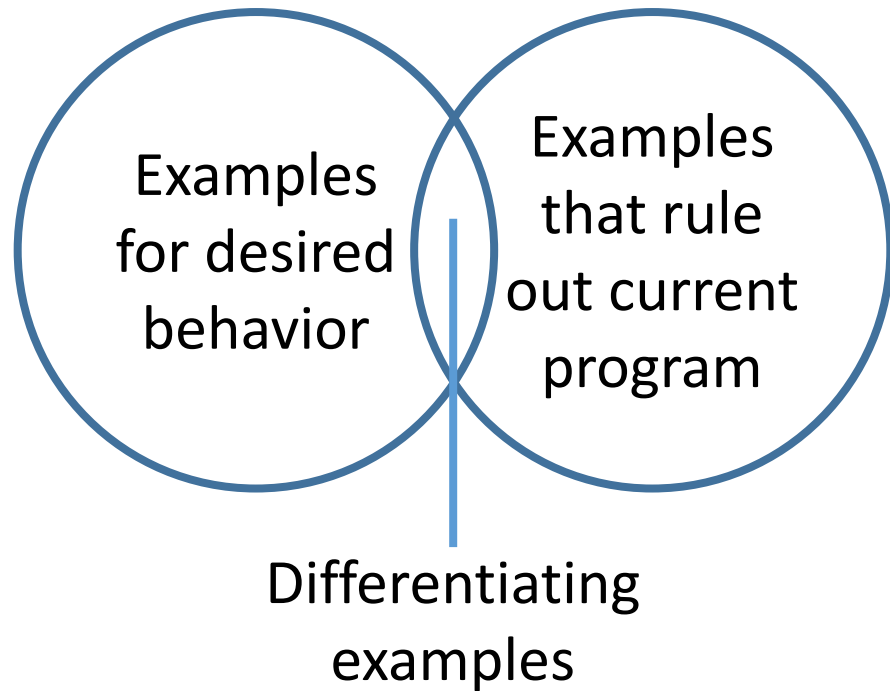


Problem 1: Differentiating examples are hard



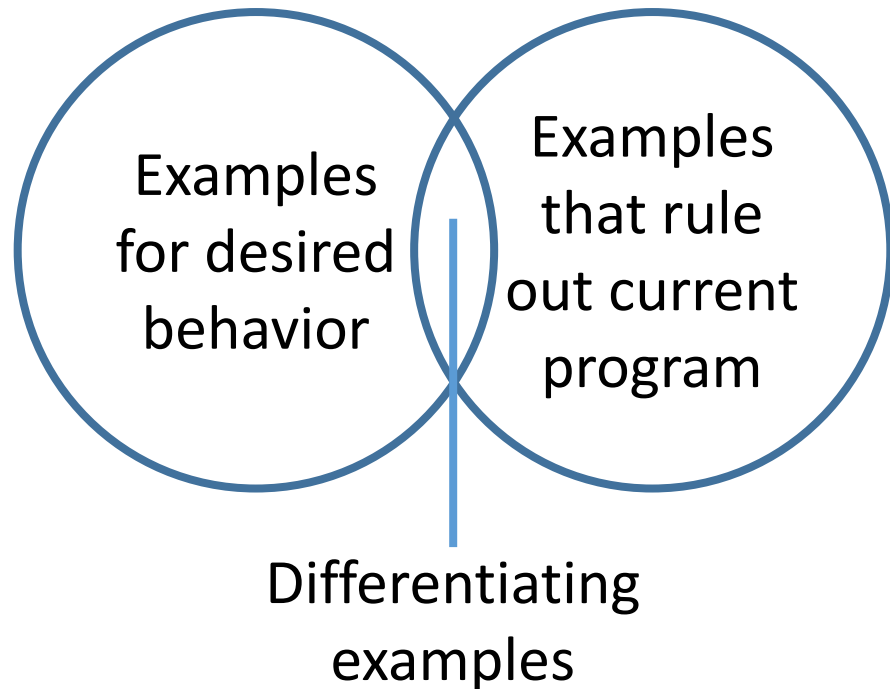
- Example: find the median
- Input: [1,2,3] Output: 2

Problem 1: Differentiating examples are hard



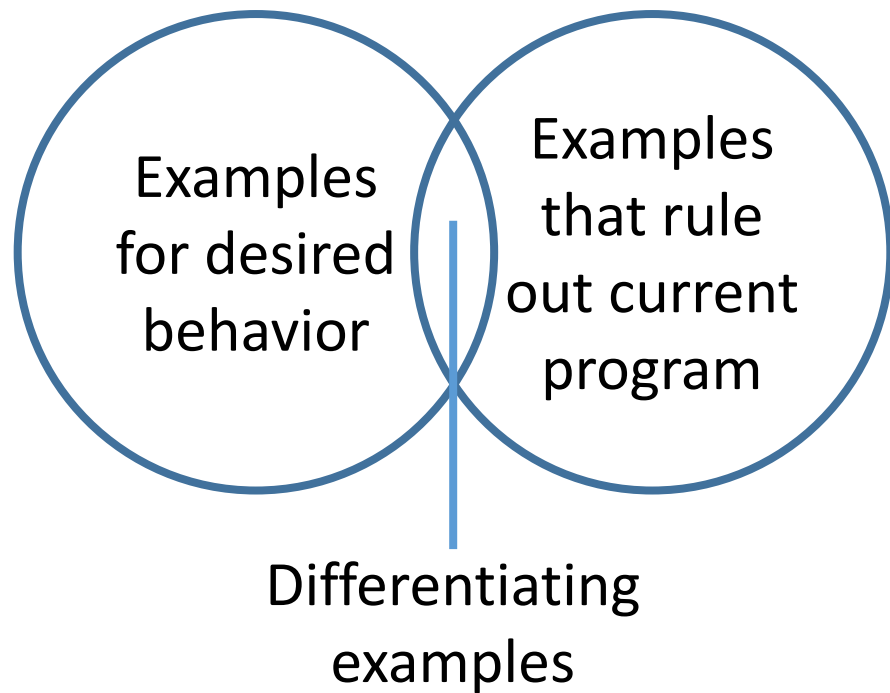
- Example: find the median
- Input: [1,2,3] Output: 2
- Candidate program:
`input[input.length / 2]`

Problem 1: Differentiating examples are hard



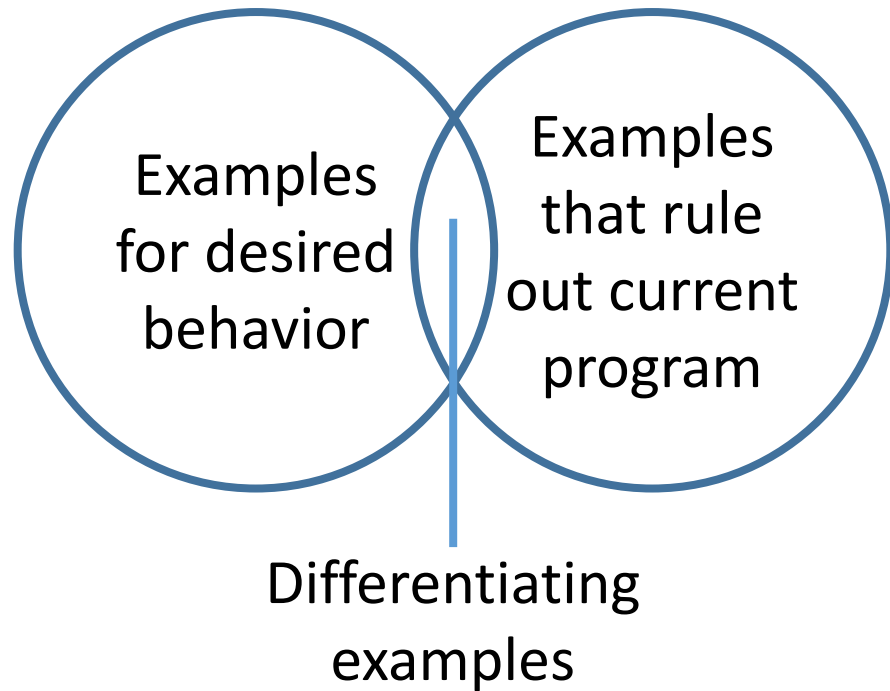
- Example: find the median
- Input: [1,2,3] Output: 2
- Candidate program:
`input[input.length / 2]`
- To create a differentiating example:
 - Figure out why this happened
 - Turn that **back into** an example

Problem 1: Differentiating examples are hard



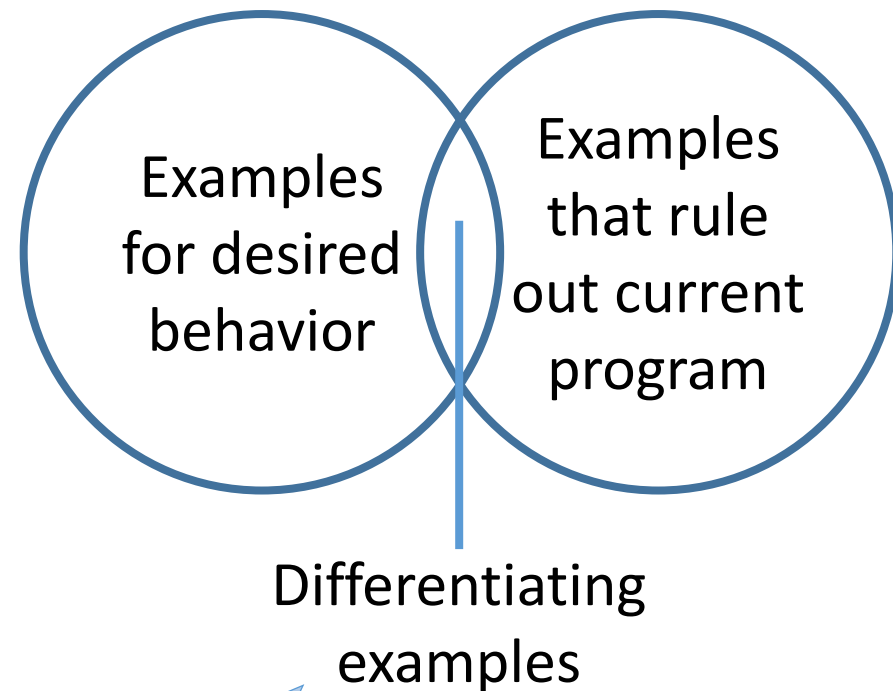
- Example: find the median
- Input: [1,2,3] Output: 2
- Candidate program:
`input[input.length / 2]`
- To create a differentiating example:
 - Figure out why this happened
 - Turn that **back into** an example
- Make sure the median is not in the middle

Problem 1: Differentiating examples are hard



- Example: find the median
- Input: [1,2,3] Output: 2
- Candidate program:
`input[input.length / 2]`
- To create a differentiating example:
 - Figure out why this happened
 - Turn that **back into** an example
- Make sure the median is not in the middle
- Input: [1,3,2] Output: 2

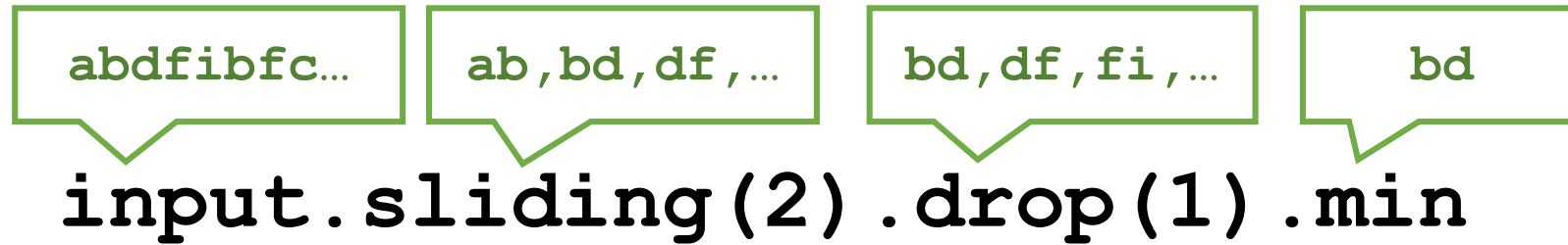
Problem 1: Differentiating examples are hard



Usually takes a REPL and some time

- Example: find the median
- Input: [1,2,3] Output: 2
- Candidate program:
`input[input.length / 2]`
- To create a differentiating example:
 - Figure out why this happened
 - Turn that **back into** an example
- Make sure the median is not in the middle
- Input: [1,3,2] Output: 2

Problem II: Examples are terrible when you (kind of) know what you want

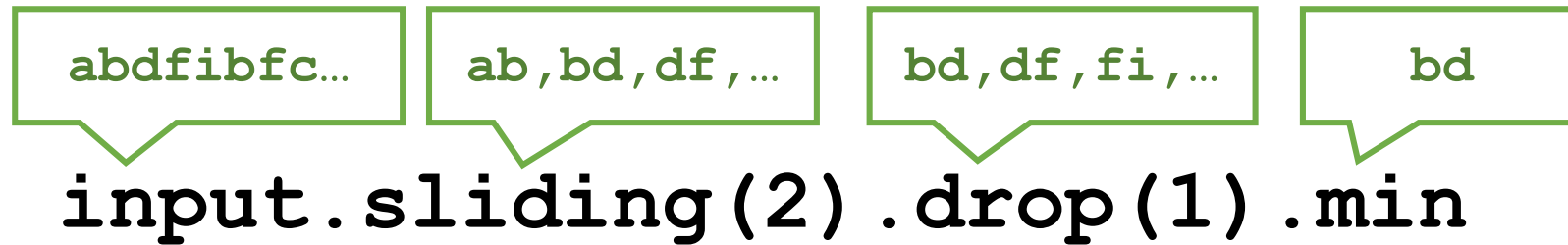


Problem II: Examples are terrible when you (kind of) know what you want

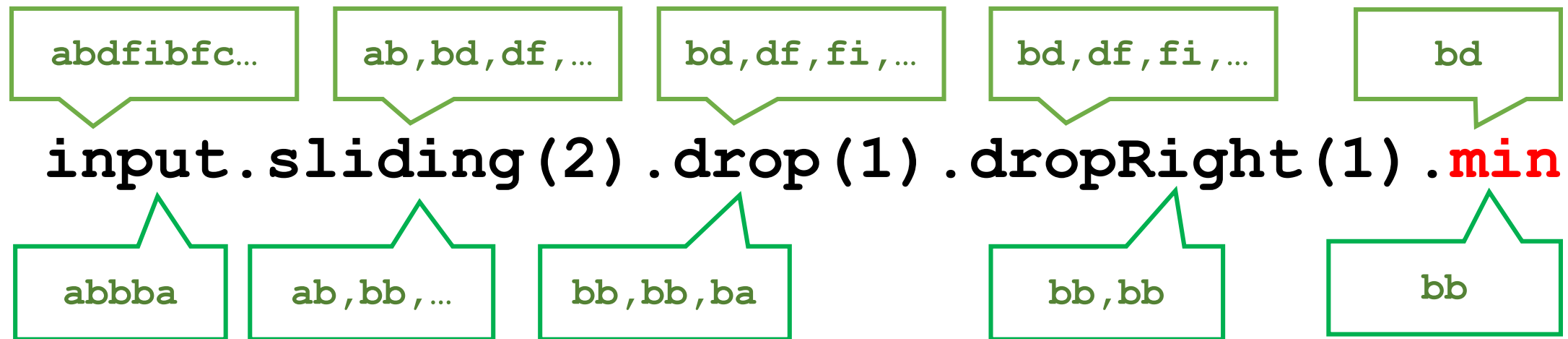


Input: "abbba", Output: "bb"

Problem II: Examples are terrible when you (kind of) know what you want



Input: "abbba", Output: "bb"



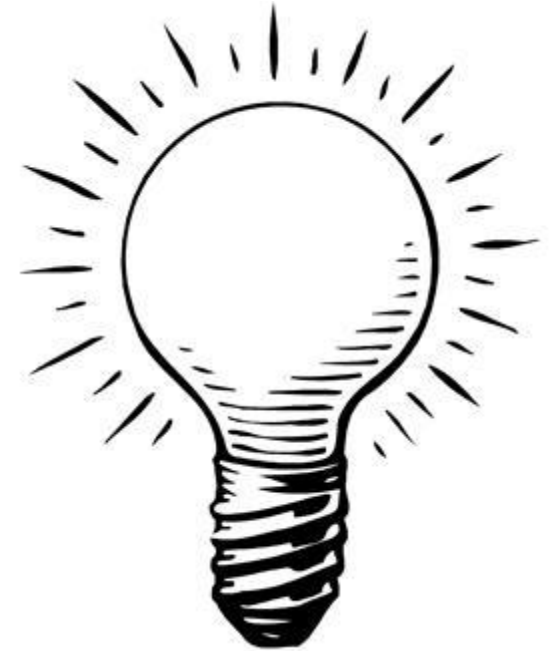
A narrower scope: synthesis for programmers

- Programmers can interact on a lower level



A narrower scope: synthesis for programmers

- Programmers can interact on a lower level
- They understand sub-problems
- They understand the code
- (They can be helped to understand)



A narrower scope: synthesis for programmers

- Programmers can interact on a lower level
- They understand sub-problems
- They understand the code
- (They can be helped to understand)
- They should be given the power



The Granular Interaction Model (GIM)

```
input
//ab, bd, df, ...
.sliding(2)
//bd, df, fi, ...
.drop(1)
//bd
.min
```

- A programmer can talk at the level of the program
- Read debug info
- Reason about subtrees or sequences of methods
- Or intermediate states
- But also examples, if those happen to be easier

The Granular Interaction Model (GIM)

```
input  
//ab, bd, df, ...  
.sliding(2)  
//bd, df, fi, ...  
.drop(1)  
//bd  
.min
```

That looks right

- A programmer can talk at the level of the program
- Read debug info
- Reason about subtrees or sequences of methods
- Or intermediate states
- But also examples, if those happen to be easier

The Granular Interaction Model (GIM)

```
input
```

```
//ab, bd, df, ...
```

```
.sliding(2)
```

```
//bd, df, fi, ...
```

```
.drop(1)
```

```
//bd
```

```
.min
```

That looks right

Those are wrong

- A programmer can talk at the level of the program
- Read debug info
- Reason about subtrees or sequences of methods
- Or intermediate states
- But also examples, if those happen to be easier

Granular operations: an example set

Exclude

`exclude (f.g.h):`
never show programs
of the form
`input...f.g.h...`

Granular operations: an example set

Exclude

`exclude(f.g.h):`
never show programs
of the form
`input...f.g.h...`

Retain

`retain(f.g.h):`
only show programs
of the form
`input...f.g.h...`

Granular operations: an example set

Exclude

`exclude(f.g.h):`
never show programs
of the form
`input...f.g.h...`

Retain

`retain(f.g.h):`
only show programs
of the form
`input...f.g.h...`

Essentially, creates a
procedure

Granular operations: an example set

Exclude

`exclude(f.g.h):`
never show programs
of the form
`input...f.g.h...`

Retain

`retain(f.g.h):`
only show programs
of the form
`input...f.g.h...`

Essentially, creates a
procedure

Affix

`affix(f.g.h):`
only show programs
of the form
`input.f.g.h...`

Back to our example: most frequent bigram

Input: "a**bd**fibfcfde**bd**fde**bd**ihgfkj fde**bd**"

Output: "bd"

```
input.takeRight(2)
```

Back to our example: most frequent bigram

Input: "a**bd**fibfcfde**bd**fde**bd**ihgfkj fde**bd**"

Output: "bd"

```
input.takeRight(2)
```

The user can answer locally:

Back to our example: most frequent bigram

Input: "a**bd**fibfcfde**bd**fde**bd**ihgfkj fde**bd**"

Output: "bd"

```
input.takeRight(2)
```

The user can answer locally:

exclude takeRight (2)

Another step

The synthesizer produces another candidate program

```
input // "abdfibfcfdebdfdebdihgfkjfdebd"  
.drop(1) // "bdfibfcfdebdfdebdihgfkjfdebd"  
.take(2) // "bd"
```


Another step

The synthesizer produces another candidate program

```
input // "abdfibfcfdebfdfdebdihgfkjfdebd"  
.drop(1) // "bdfibfcfdebfdfdebdihgfkjfdebd"  
.take(2) // "bd"
```

Answer: **exclude drop (1) · take (2)**

And another

The synthesizer answers

```
input // "abdfibfcfdebfdebdihgfkjfdebd"  
.zip(input.drop(1)) // List((a,b), (b,d), (d,f), (f,i), ...)   
.take(2) // List((a,b), (b,d))   
.map(p => p._1.toString + p._2) // List("ab", "bd")   
.max // "bd"
```

User provides a compound answer:

And another

The synthesizer answers

```
input // "abdfibfcfdebfdebdihgfkj fdebd"  
.zip(input.drop(1)) // List((a,b), (b,d), (d,f), (f,i), ...) )  
.take(2) // List((a,b), (b,d))  
.map(p => p._1.toString + p._2) // List("ab", "bd")  
.max // "bd"
```

User provides a compound answer:

```
affix zip(input.drop(1))
```

And another

The synthesizer answers

```
input // "abdfibfcfdebfdebdihgfkjfdebd"  
.zip(input.drop(1)) // List((a,b), (b,d), (d,f), (f,i), ...)  
.take(2) // List((a,b), (b,d))  
.map(p => p._1.toString + p._2) // List("ab", "bd")  
.max // "bd"
```

User provides a compound answer:

affix zip(input.drop(1))

exclude take(2)

And another

The synthesizer answers

```
input// "abdfibfcfdebfdebdihgfkjfdebd"  
.zip(input.drop(1))//List((a,b),(b,d),(d,f),(f,i),...)   
.take(2)//List((a,b),(b,d))  
.map(p => p._1.toString + p._2)//List("ab","bd")  
.max// "bd"
```

User provides a compound answer:

affix zip(input.drop(1))

exclude take(2)

And possibly even **retain map(p => p._1.toString + p._2)**

Until finally

```
input // "abdfibfcfdebfdebdihgfkjfdbd"  
.zip(input.drop(1)) // List((a,b), (b,d), (d,f), (f,i), ...  
.map(p => p._1.toString + p._2) // List("ab", "bd", ...  
.groupBy(x => x) // Map("bf" -> List("bf"), "ib" -> List(...  
.map(kv => kv._1 -> kv._2.length) // Map("bf" -> 1, ...  
.maxBy(_. _2) // ("bd", 4)  
._1 // "bd"
```

Evaluating GIM

Synthesizer:

- Scala functional programs
- Precomputed program space
- We record only user time

Three groups:

1. PBE: examples only (11)
2. Syntax: only syntactic operations (10)
3. GIM: both examples and syntactic (11)

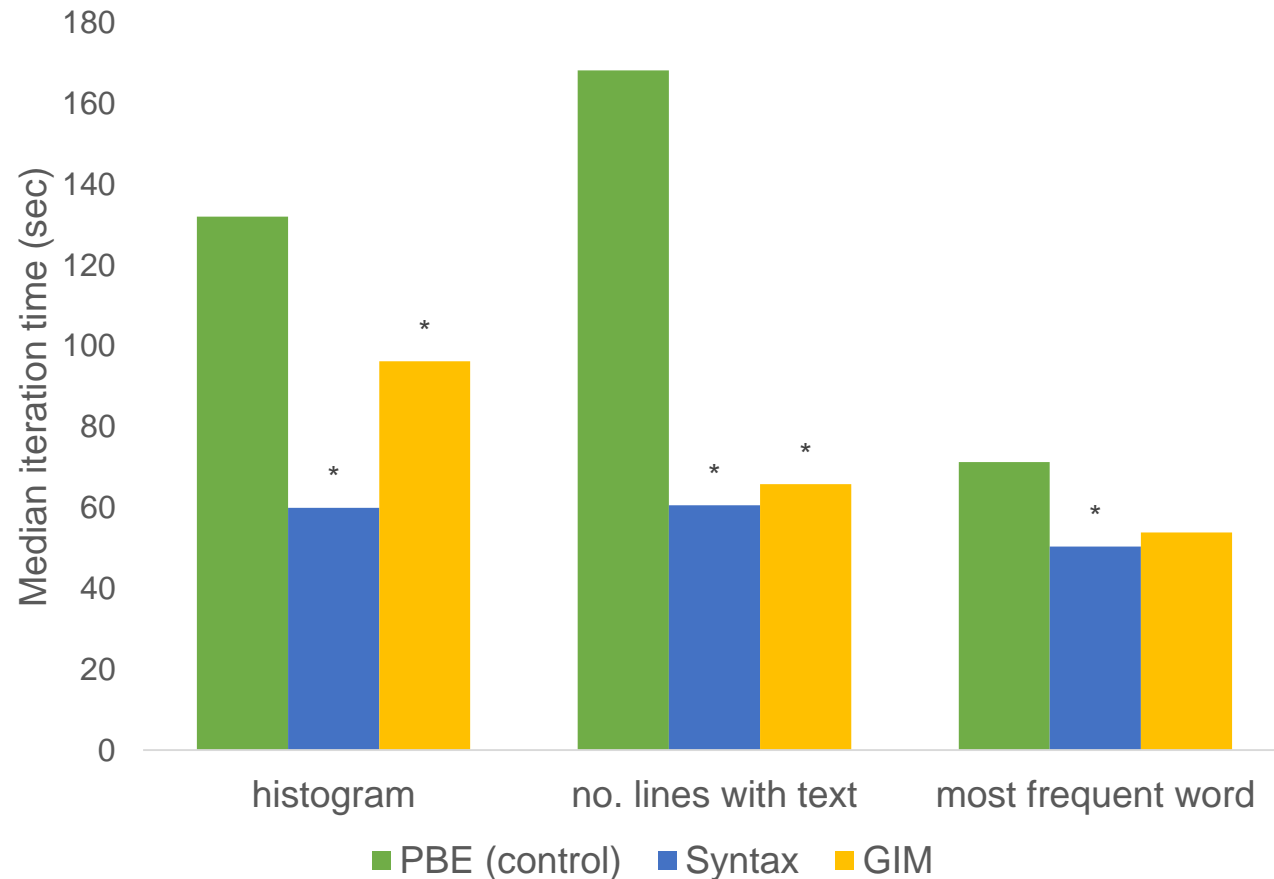
Three problem sets:

1. Most frequent word
2. No. lines with text
3. Histogram

Research questions:

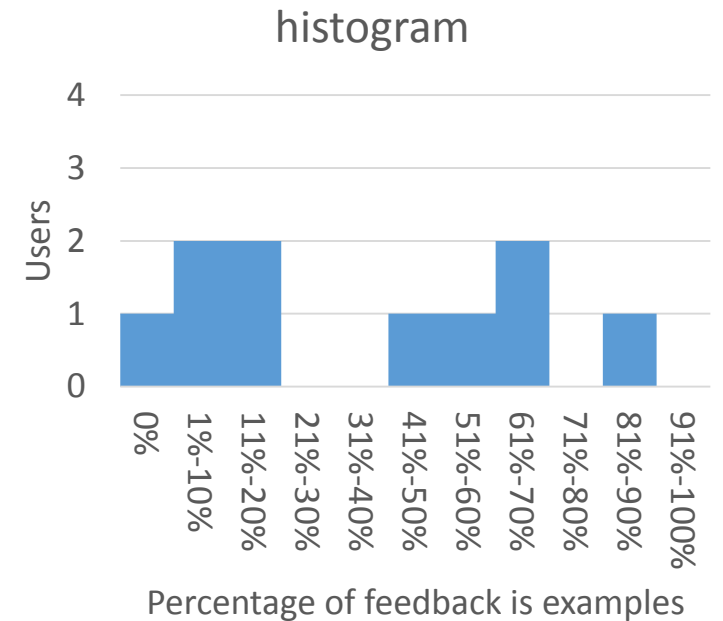
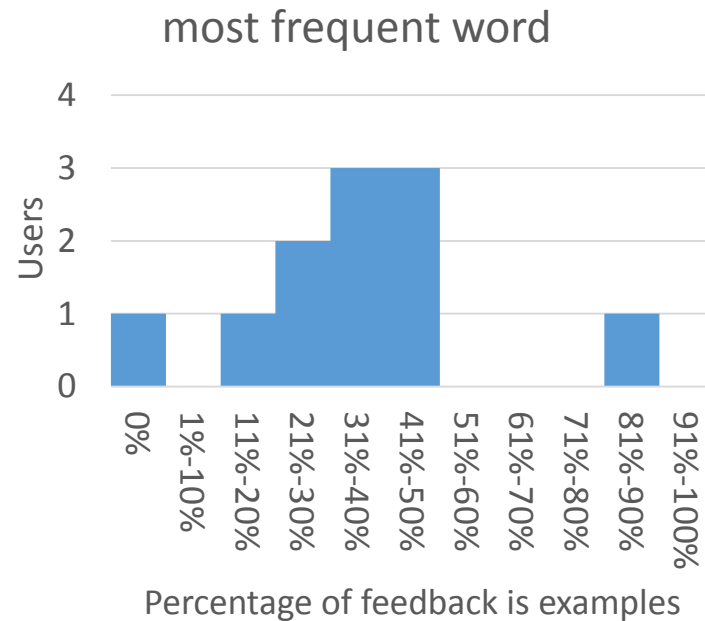
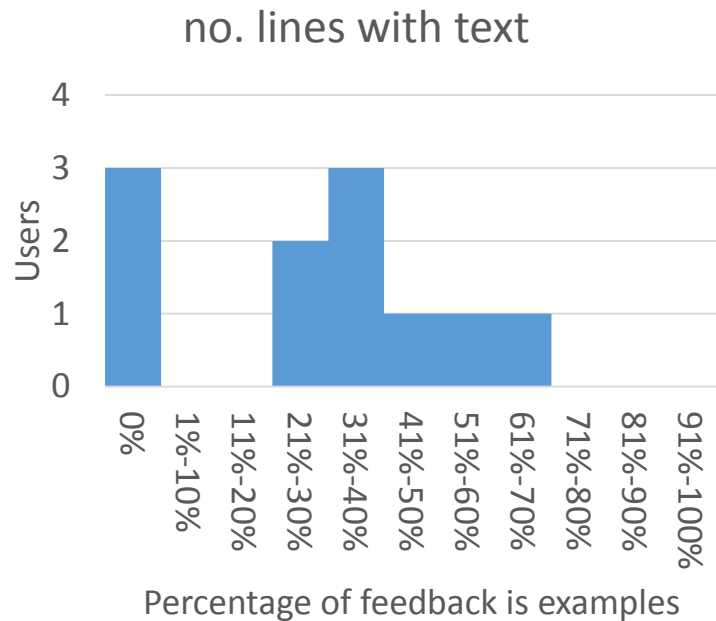
1. Are answers consisting of syntactic predicates easier (faster) to generate than example predicates?
2. Is the total time to solution faster?
3. Do users prefer examples?
4. Are users' answers correct?

RQ1+2: Iteration times and total times

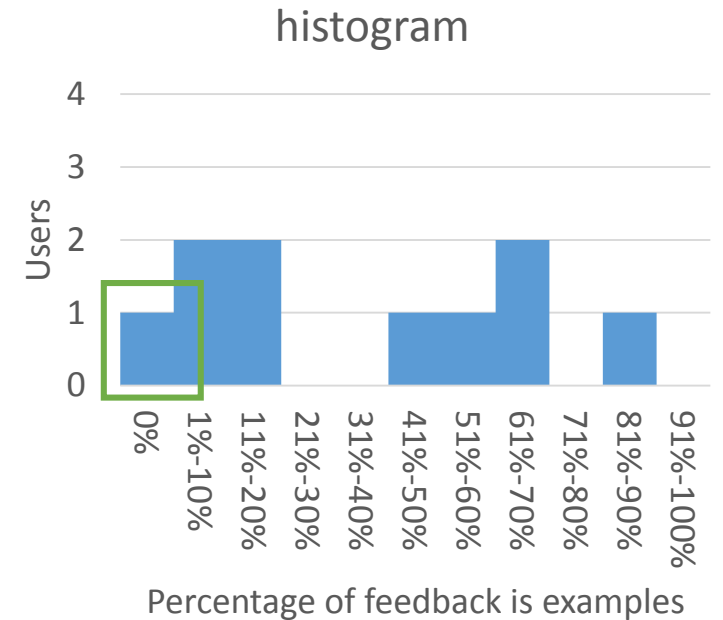
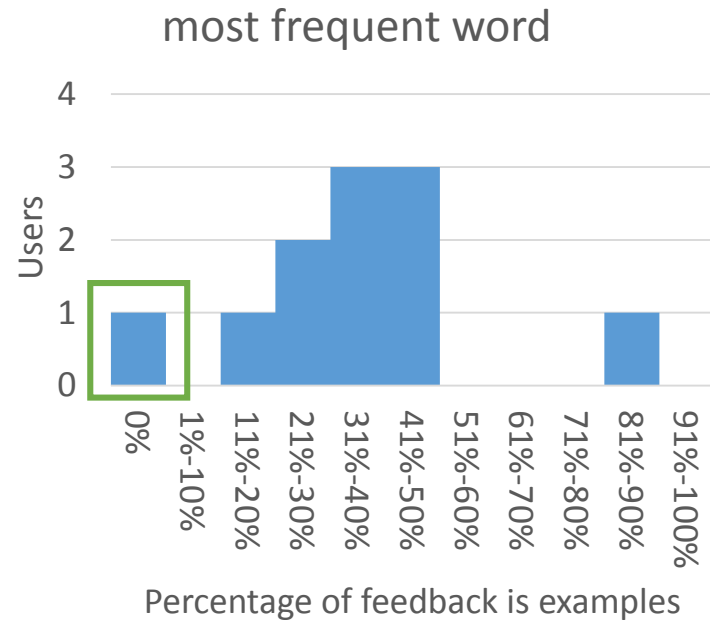
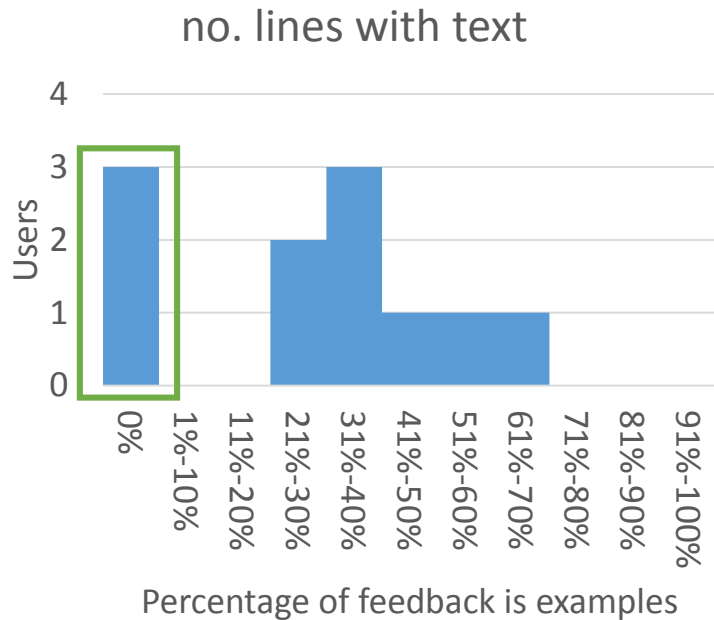


- Syntax-only iterations are fastest
- GIM iterations are almost as fast
- There's no statistically significant difference in total time (RQ2)

RQ3: Users like examples (but not that much)

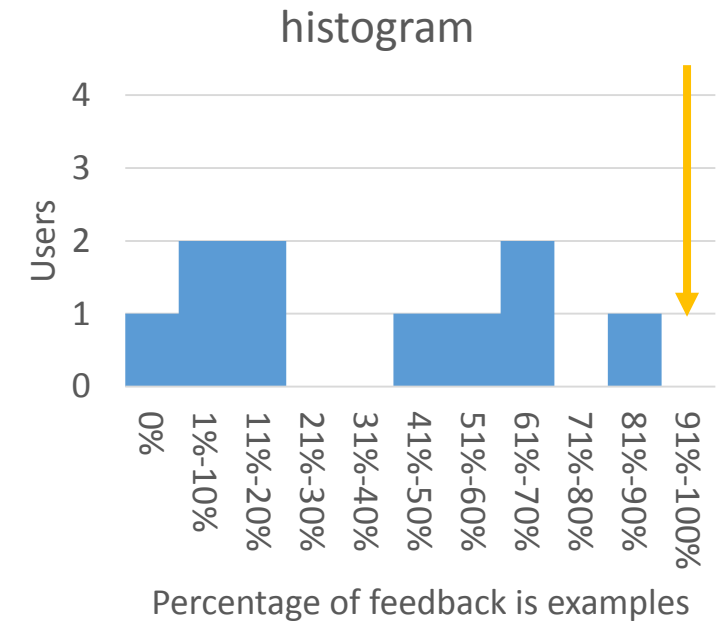
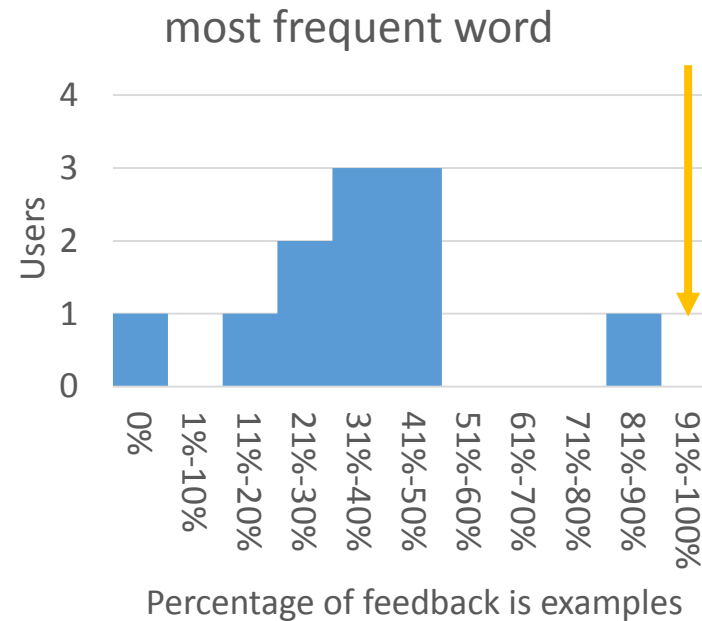
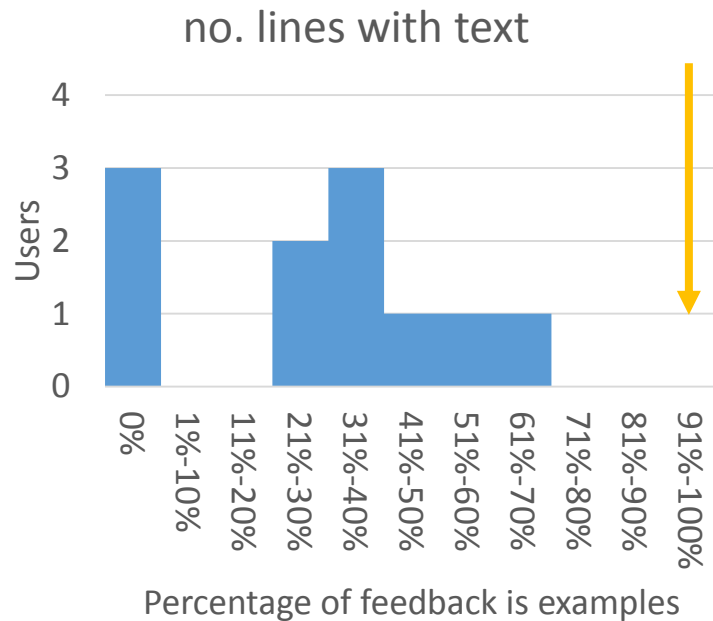


RQ3: Users like examples (but not that much)



Users who used no examples < 30%

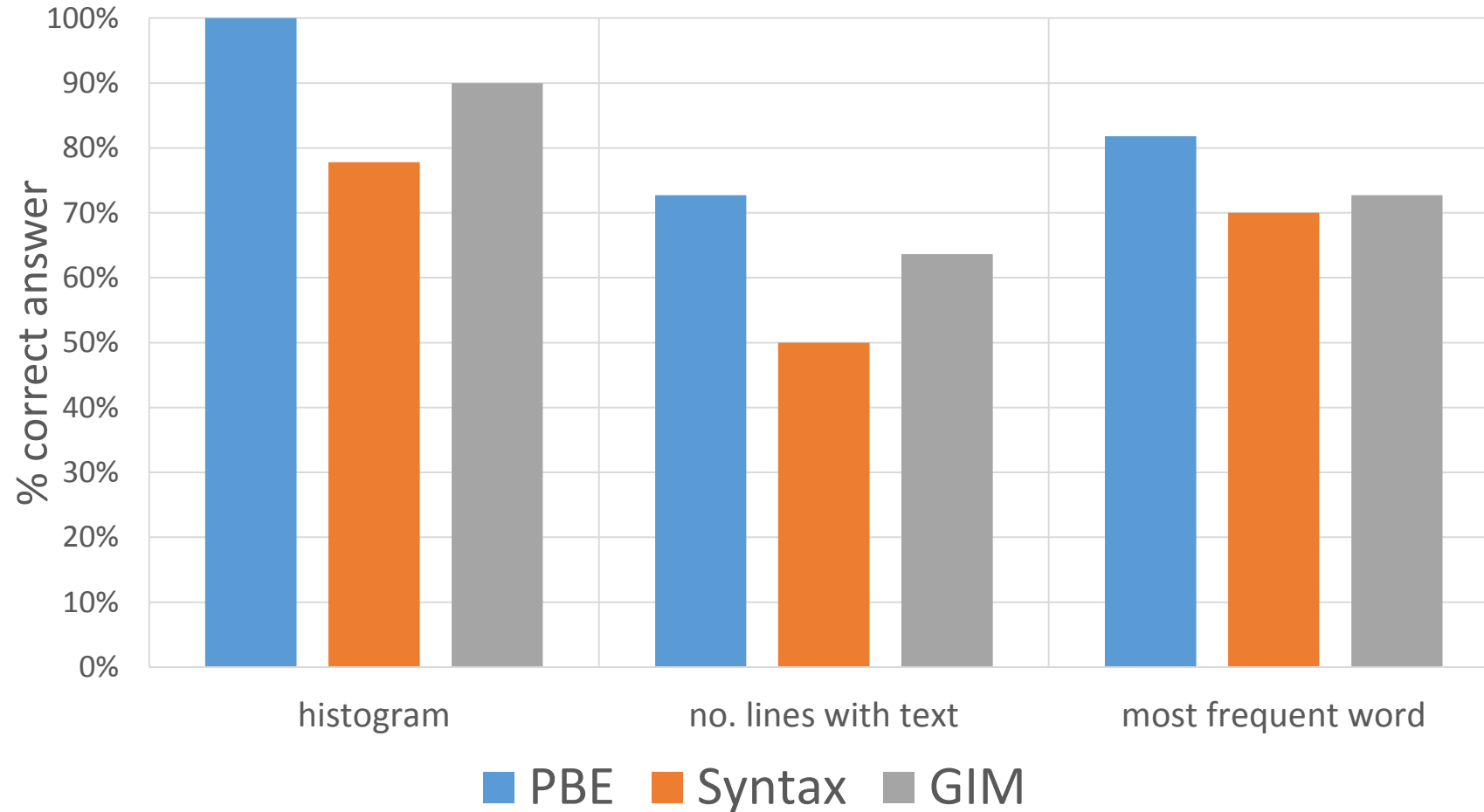
RQ3: Users like examples (but not that much)



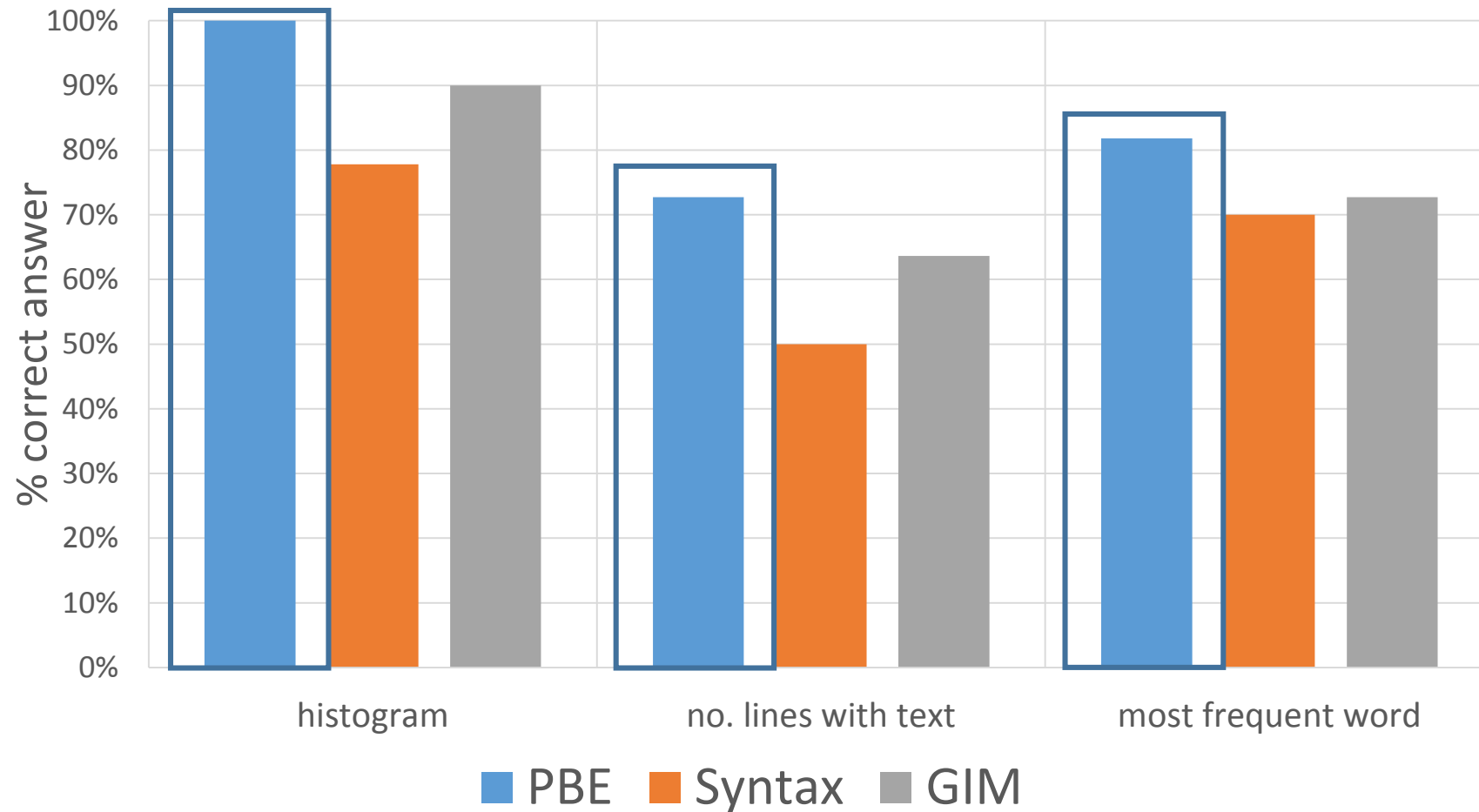
Users who used no examples < 30%

Max examples used < 90%

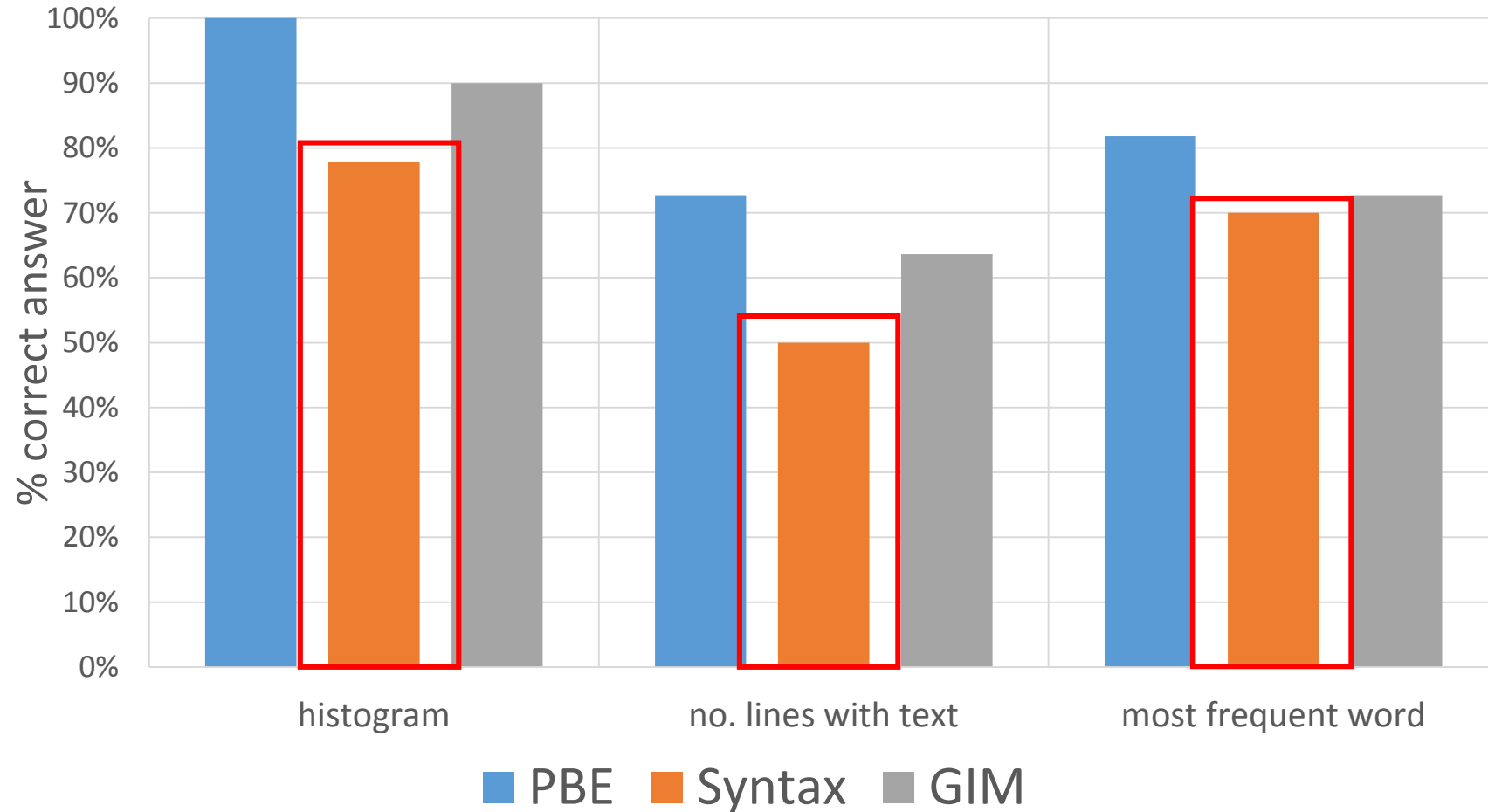
RQ4: More examples -> more correctness



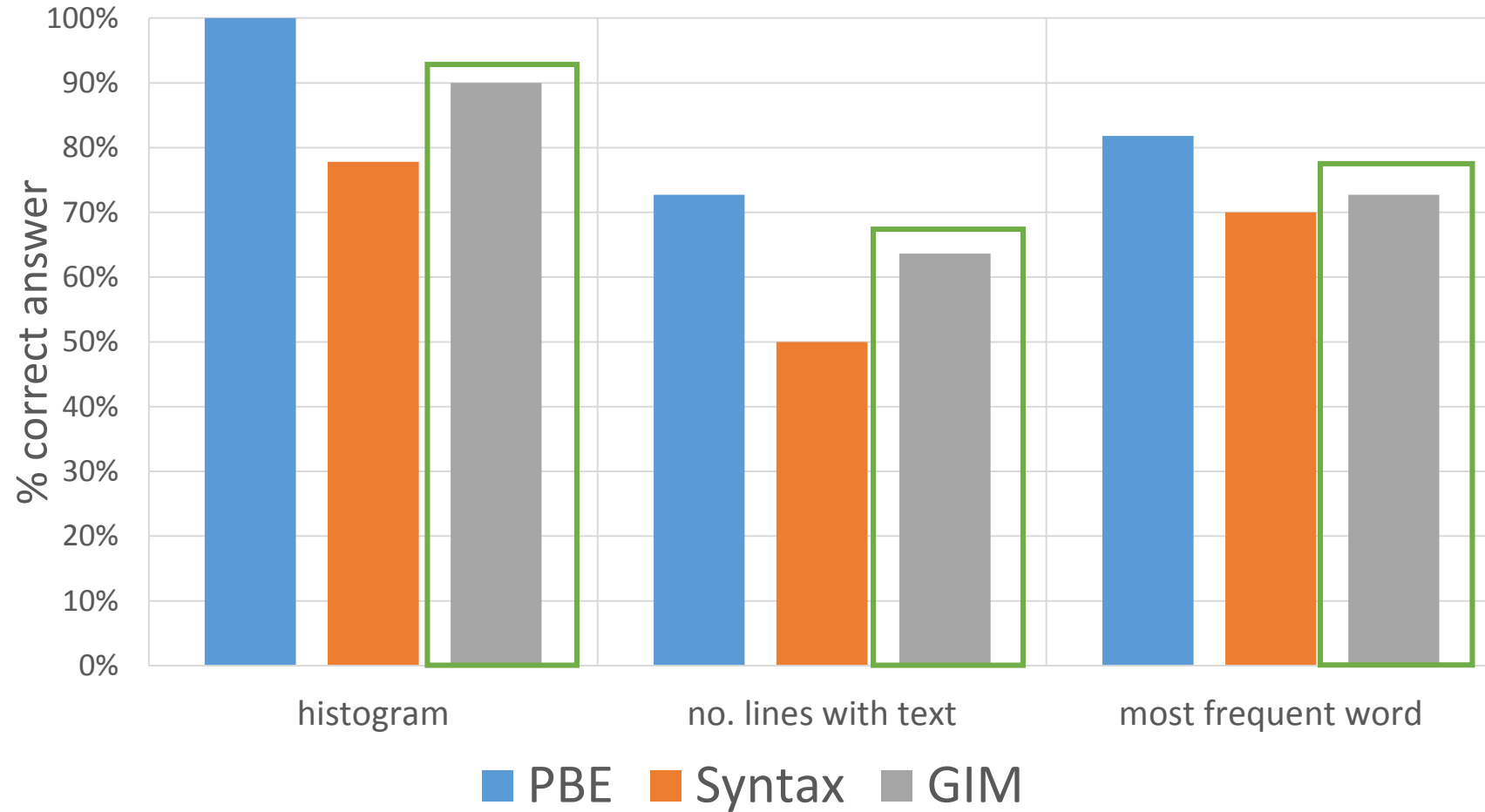
RQ4: More examples -> more correctness



RQ4: More examples -> more correctness



RQ4: More examples -> more correctness



Summary

- Programmers want power and options
- Syntax operations are easier than examples
- And better at getting rid of ambiguity and distracting elements
- Users like examples, but like having other tools
- Let's make synthesizers that cater to programmers!