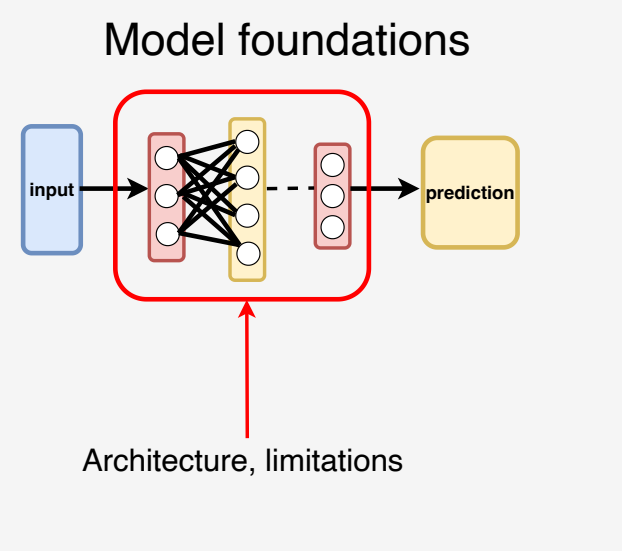
Language Model Programming Lecture 4: Prompting as Programming

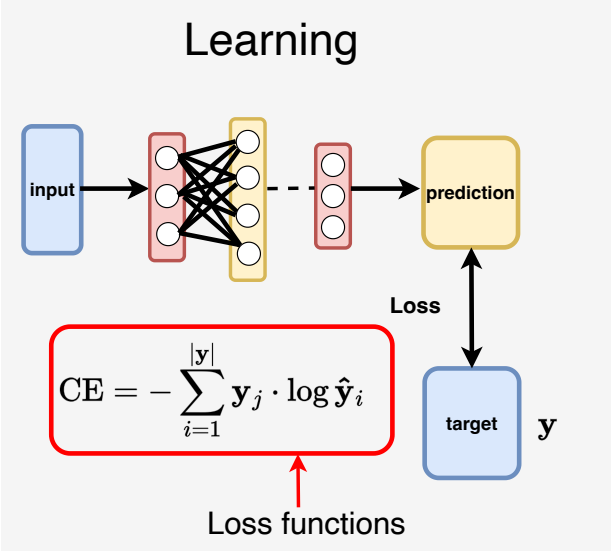
Kyle Richardson, **Gijs Wijnholds**

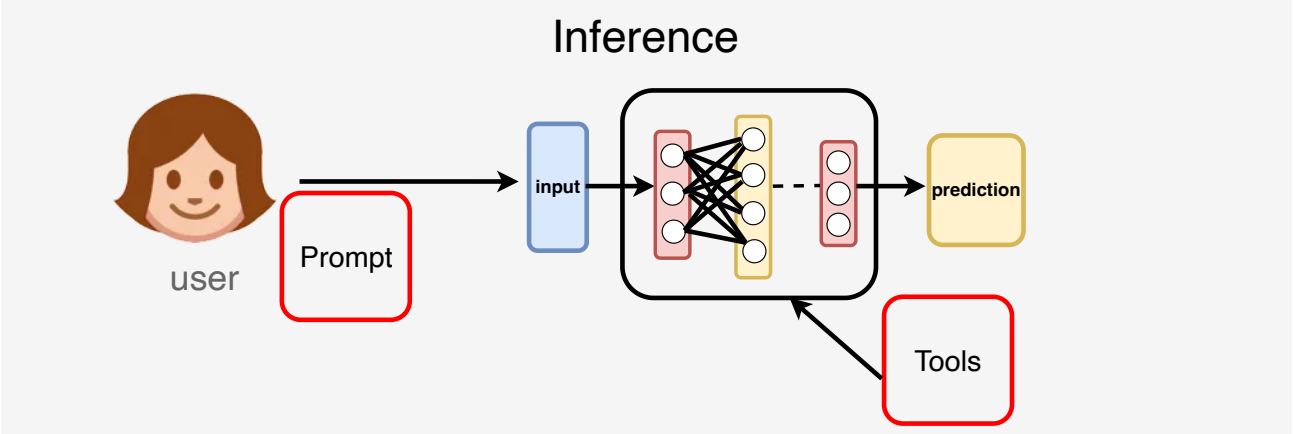
Allen Institute for AI (AI2)
Leiden Institute of Advanced Computer Science

August 2024

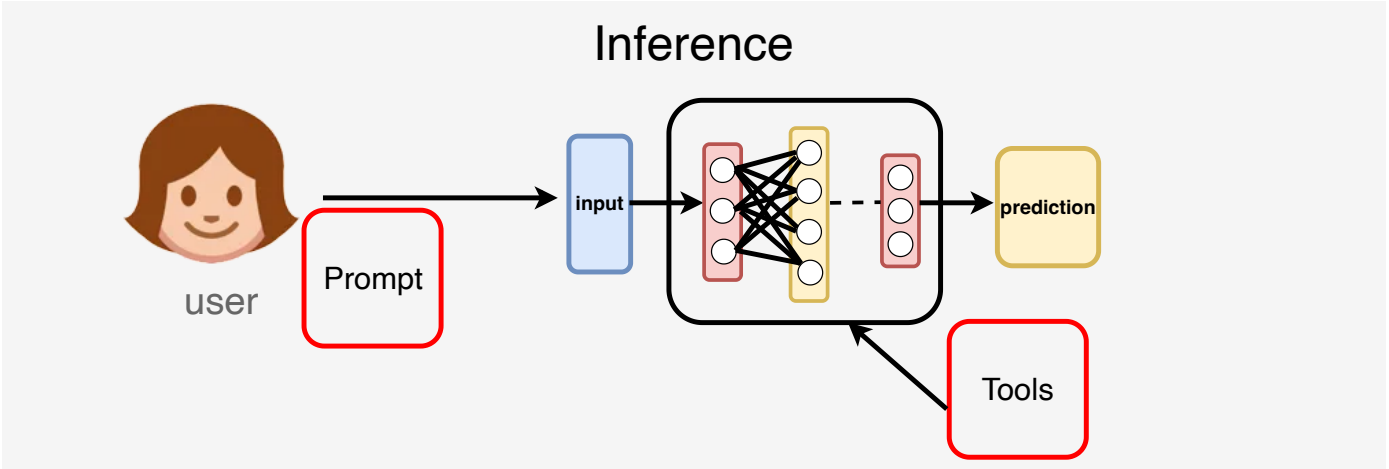
Course Overview







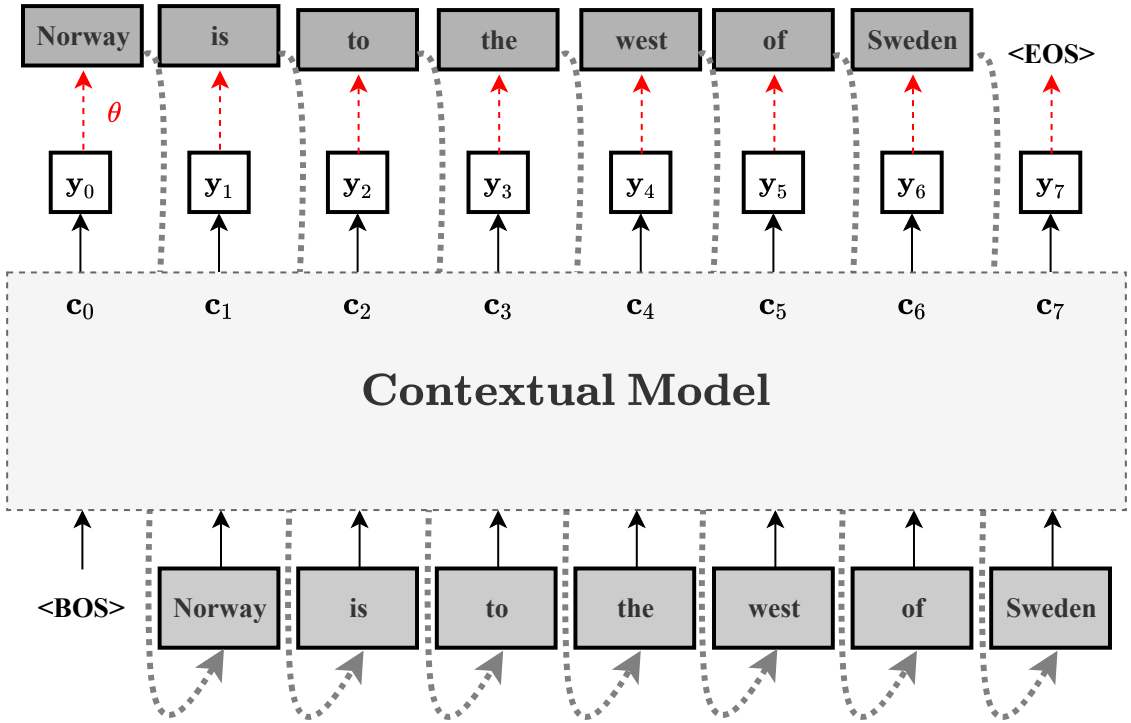
Today's Focus



- How to generate
- Prompting techniques
- Break
- Prompting is programming

Basics of generation with LLMs

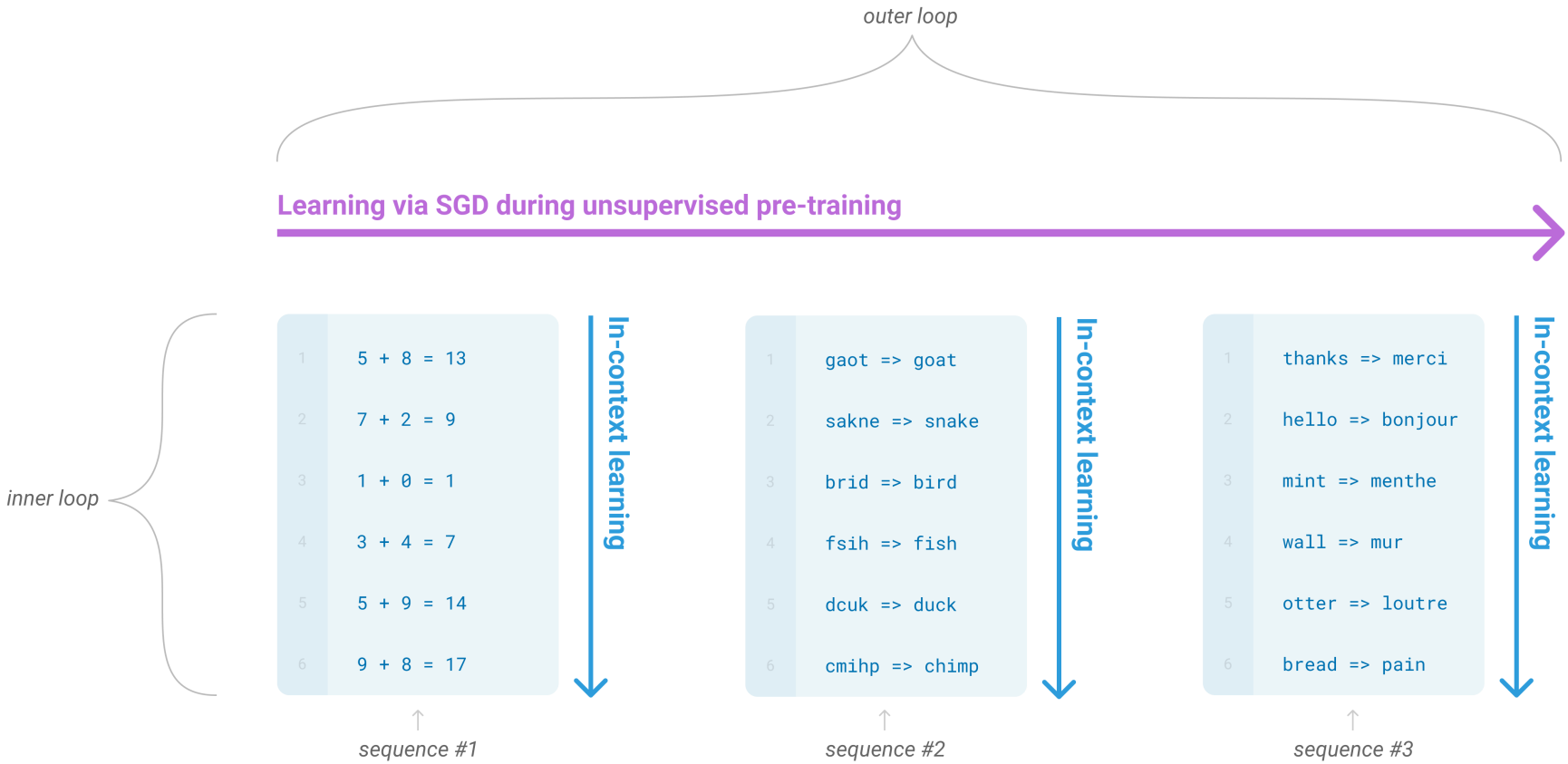
Prompting and decoding



- Learnt LLM probabilities are conditioned on the **prompt**
- The generated output is the combination of:
  - The prompt-conditioned probabilities
  - A **decoding strategy**

In-Context Learning

The power of scale during pretraining, the model learns strong "pattern recognition abilities"



Brown et al. [2020]

Prompting algorithm: simple decoding

Algorithm 1: Simple Generation

Input: LM f, tokenized prompt x
Output: Completion o

$o \leftarrow \epsilon$
while *True* **do**
  $\mathbf{z} \leftarrow \texttt{softmax}(f(x \cdot o))$ `// compute logits`
  $t \leftarrow \mathit{pick}(\mathbf{z})$ `// decoding step`
  if $t = \mathit{EOS}$ **then break**
  $o \leftarrow o \cdot t$
end while
return o

Prompting algorithm: simple decoding

Algorithm 2: Simple Generation

Input: LM f, tokenized prompt x
Output: Completion o

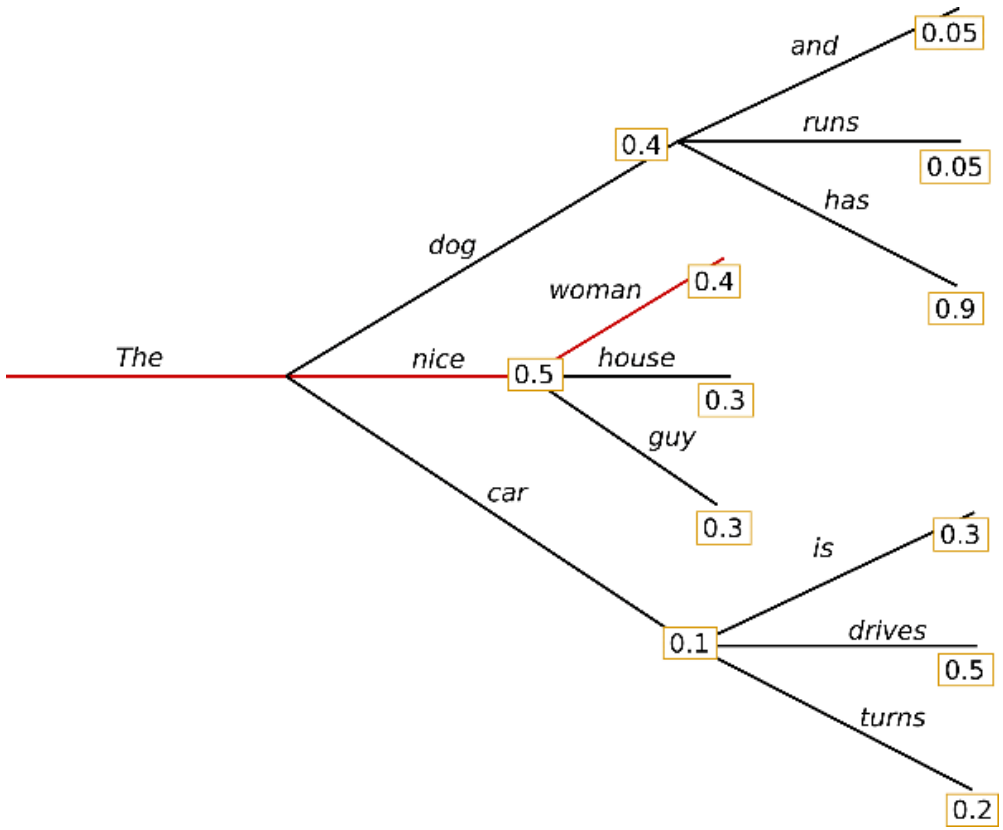
$o \leftarrow \epsilon$
while *True* **do**
  $\mathbf{z} \leftarrow \texttt{softmax}(f(x \cdot o))$ `// compute logits`
  $t \leftarrow \mathit{pick}(\mathbf{z})$ `// decoding step`
  if $t = \mathit{EOS}$ **then break**
  $o \leftarrow o \cdot t$
end while
return o

So, how to pick?

Greedy Search

Strategy take the most probably option at each timestep:

$$w_t = \arg\max_w P(w \mid w_1 \ldots w_{t-1})$$

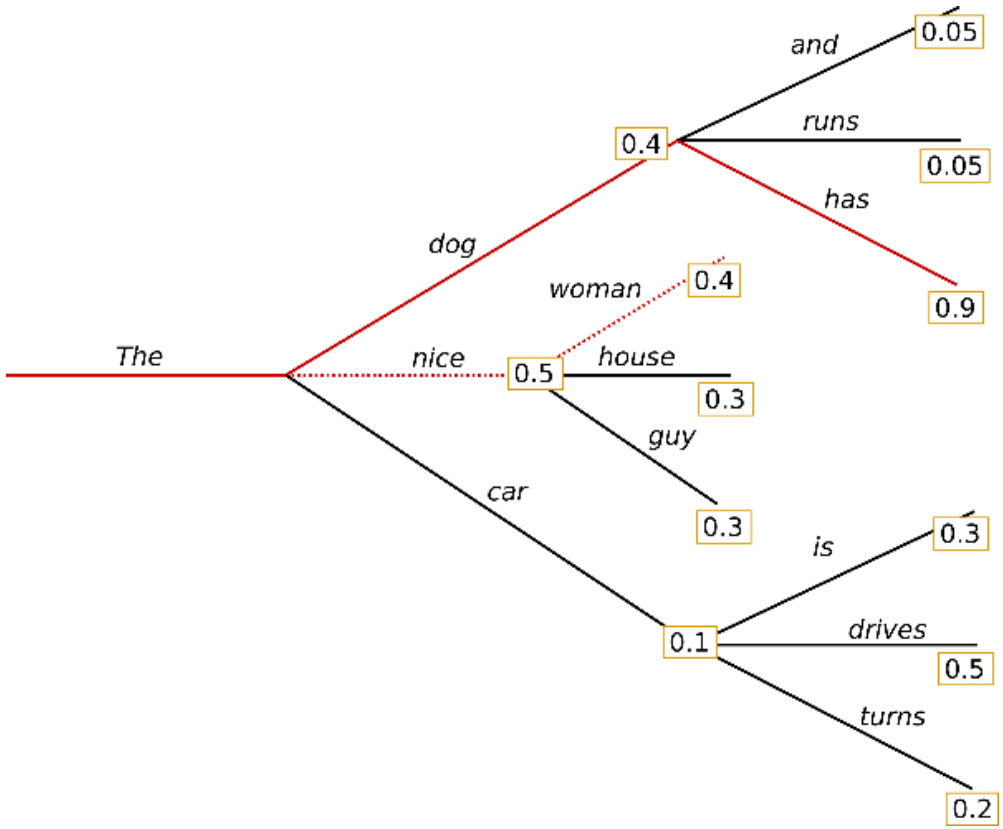


Pro linear in the sequence length

Con local choices don't guarantee the most probable sequence

Beam Search

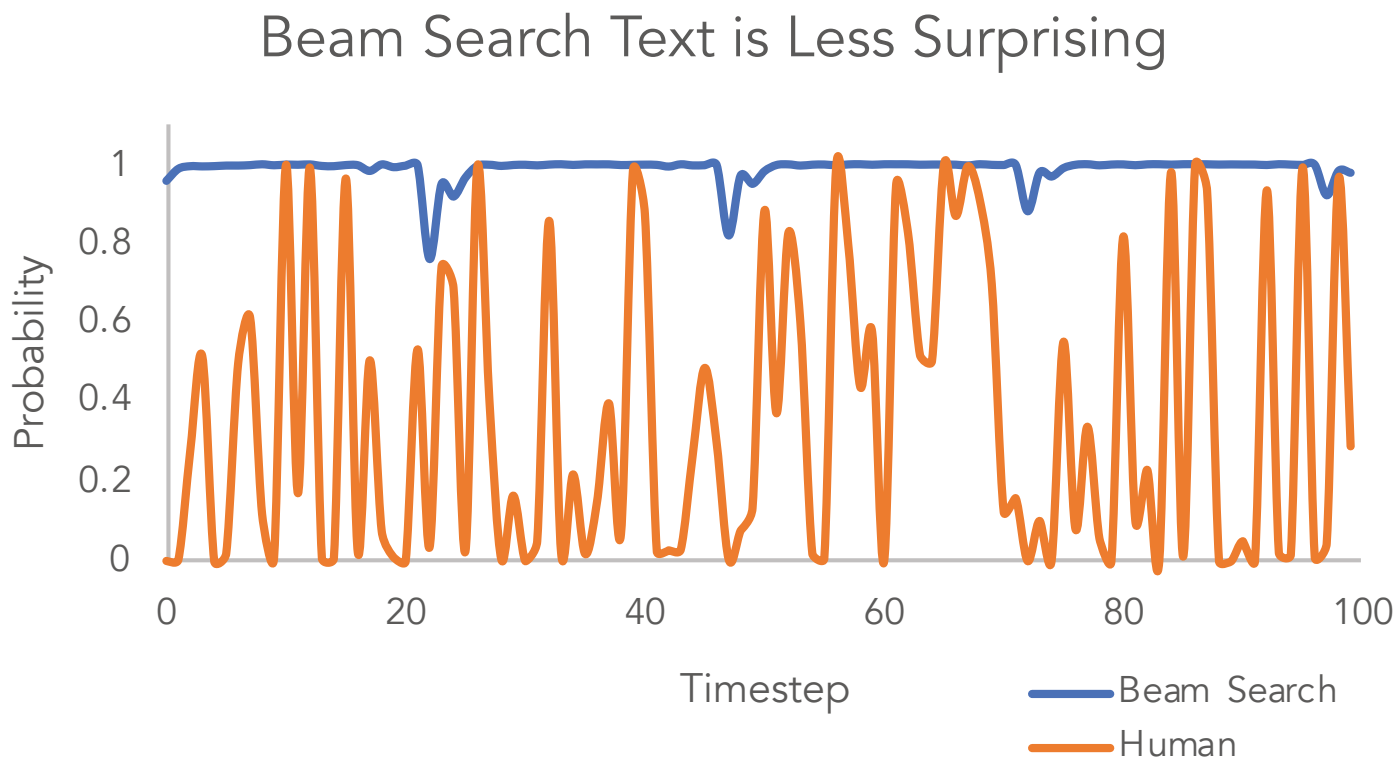
Strategy like greedy search, but keeping track of the k most likely *beams*



Pro more likely to find probable solutions

Why greedy/beam search isn't enough

Beam Search vs. Humans humans are much more varied than Beam Search:

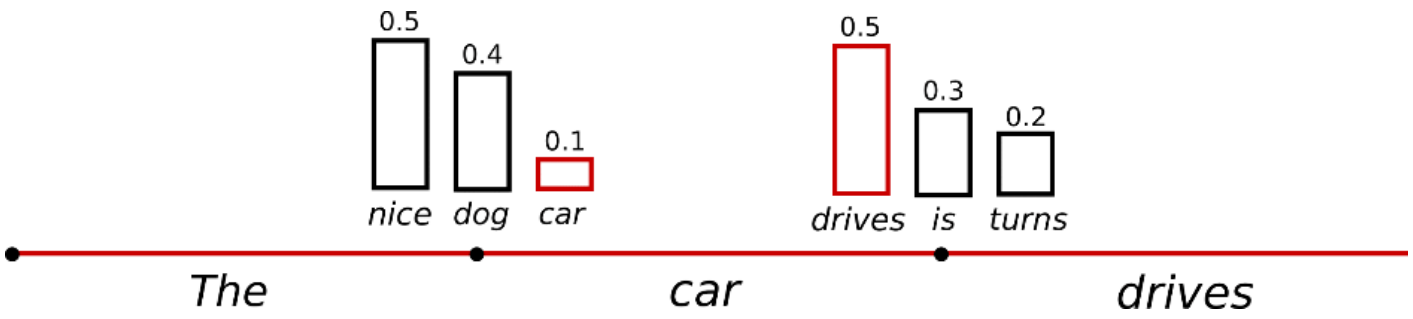


Holtzman et al. [2020]

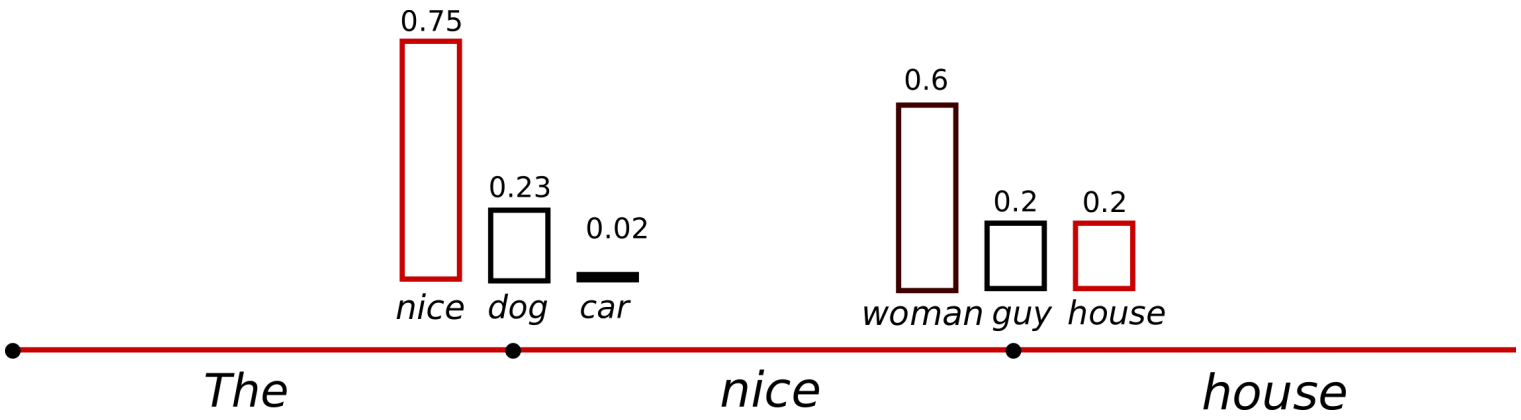
Sampling

Weighted sampling sample word from the conditional probability distribution:

$$w_t \sim P(w \mid w_1 \ldots w_{t-1})$$

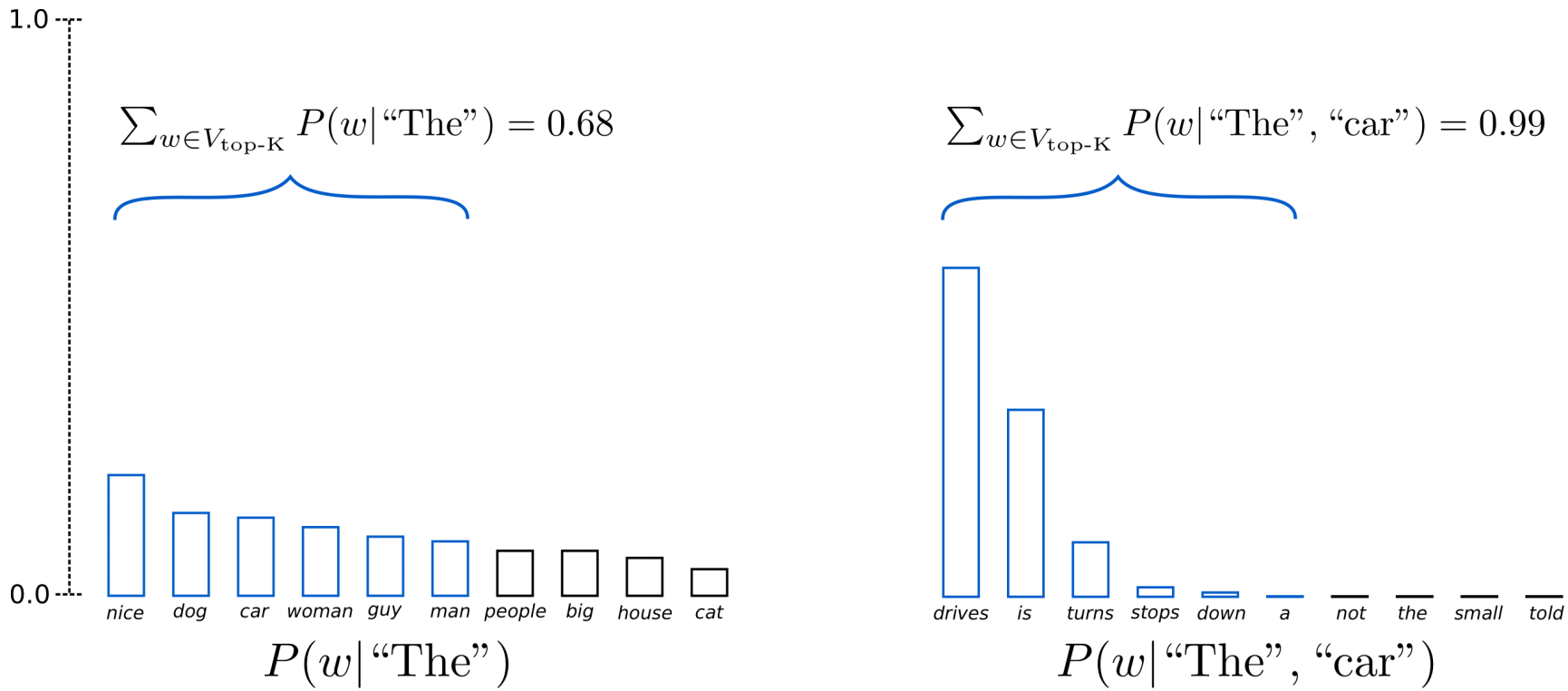


With temperature scaling



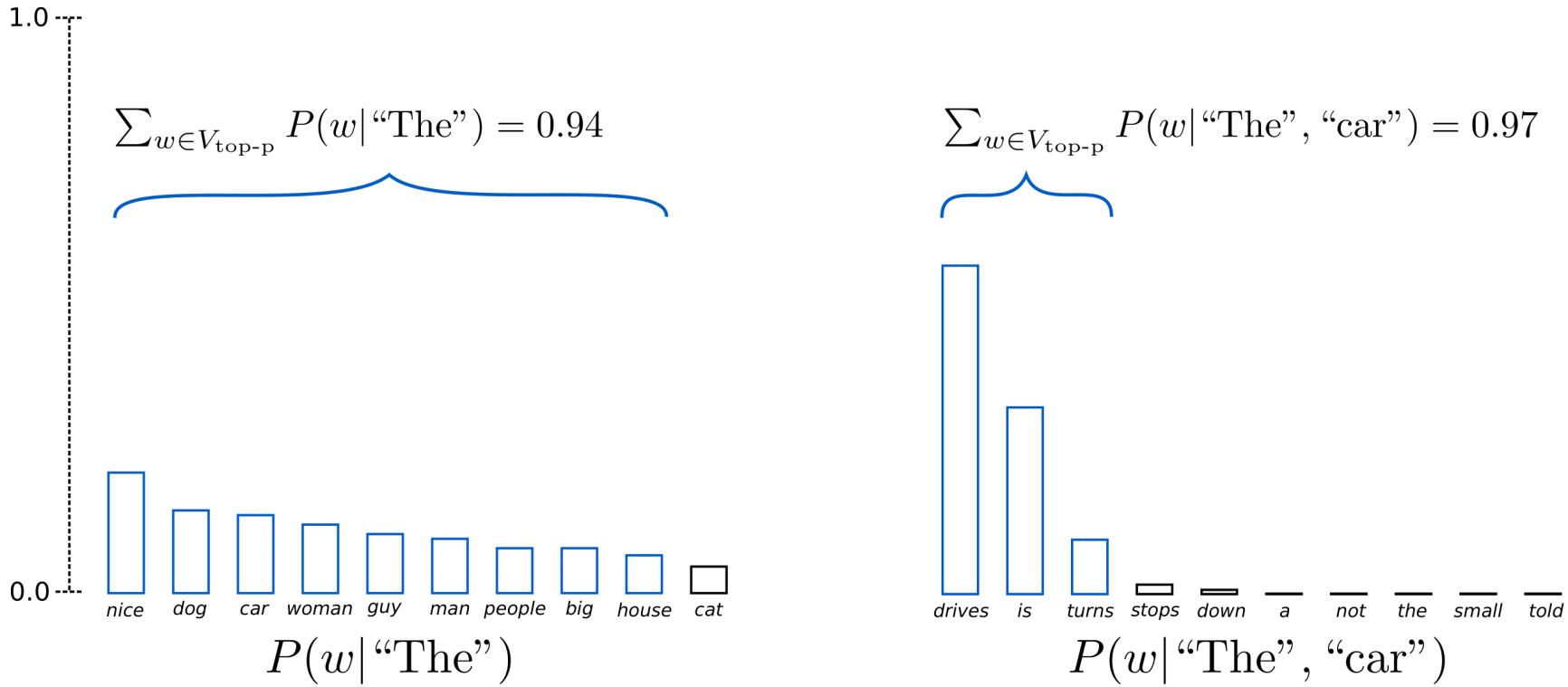
Top-K Sampling

Core idea sample only from the k most likely words:



Top-P Nucleus Sampling

Core idea sample from the set of most likely words whose combined probability exceeds p:



Prompting techniques

K-shot learning

Zero-shot

The model predicts the answer given only a natural language description of the task. No gradient updates are performed.

```
1  Translate English to French:        ← task description
2  cheese =>                           ← prompt
```

One-shot

In addition to the task description, the model sees a single example of the task. No gradient updates are performed.

```
1  Translate English to French:        ← task description
2  sea otter => loutre de mer          ← example
3  cheese =>                           ← prompt
```

Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

```
1  Translate English to French:        ← task description
2  sea otter => loutre de mer          ← examples
3  peppermint => menthe poivrée        ←
4  plush girafe => girafe peluche      ←
5  cheese =>                           ← prompt
```

Brown et al. [2020]

Chain of Thought Prompting

Standard Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The answer is 27. ✗

Chain-of-Thought Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9. ✓

Wei et al. [2022]

Chain of Thought Prompting: the Zero Shot case

(a) Few-shot

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: The answer is 11.

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A:

(Output) The answer is 8. ✗

(b) Few-shot-CoT

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A:

(Output) The juggler can juggle 16 balls. Half of the balls are golf balls. So there are 16 / 2 = 8 golf balls. Half of the golf balls are blue. So there are 8 / 2 = 4 blue golf balls. The answer is 4. ✓

(c) Zero-shot

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A: The answer (arabic numerals) is

(Output) 8 ✗

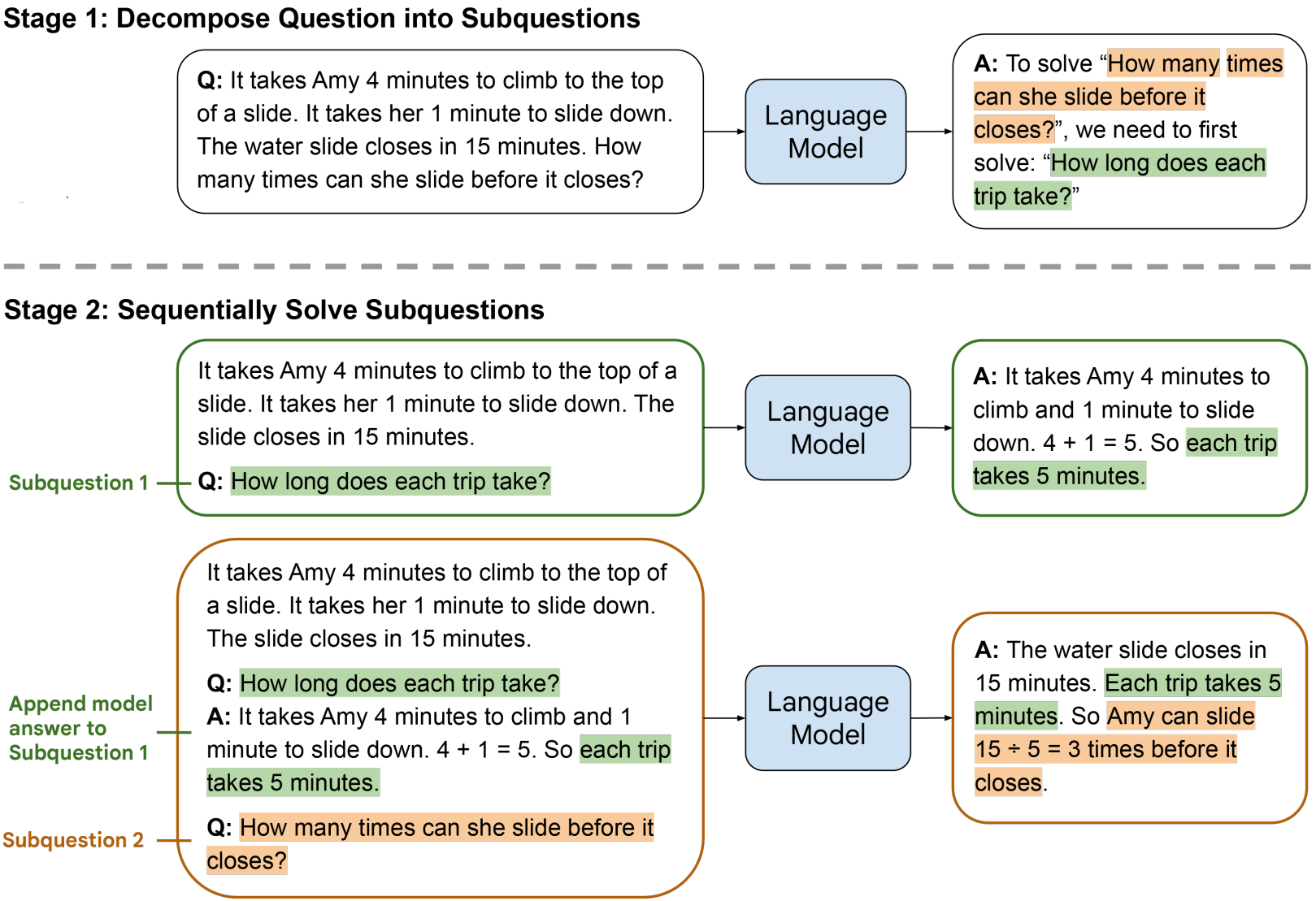
(d) Zero-shot-CoT (Ours)

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A: ***Let's think step by step.***

(Output) There are 16 balls in total. Half of the balls are golf balls. That means that there are 8 golf balls. Half of the golf balls are blue. That means that there are 4 blue golf balls. ✓

Kojima et al. [2022]

Least-to-most Prompting



Zhou et al. [2023]

ReAct prompting

(1) Hotspot QA

Question: Aside from the Apple Remote, what other device can control the program Apple Remote was originally designed to interact with?

(1a) Standard

Answer: iPod ✗

(1b) CoT (Reason Only)

Thought: Let's think step by step. Apple Remote was originally designed to interact with Apple TV. Apple TV can be controlled by iPhone, iPad, and iPod Touch. So the answer is iPhone, iPad, and iPod Touch.

Answer: iPhone, iPad, iPod Touch ✗

(1c) Act-Only

Act 1: Search[Apple Remote]
Obs 1: The Apple Remote is a remote control …

Act 2: Search[Front Row]
Obs 2: Could not find [Front Row]. Similar: …

Act 3: Search[Front Row (software)]
Obs 3: Front Row is a discontinued media center software …

Act 4: Finish[yes] ✗

(1d) ReAct (Reason + Act)

Thought 1: I need to search Apple Remote and find the program it was originally designed to interact with.
Act 1: Search[Apple Remote]
Obs 1: The Apple Remote is a remote control introduced in October 2005 by Apple … originally designed to control the Front Row media center program …

Thought 2: Apple Remote was originally designed to control the Front Row media center program. I need to search Front Row next and find what other device can control it.
Act 2: Search[Front Row]
Obs 2: Could not find [Front Row]. Similar: ['Front Row Seat to Earth', 'Front Row Motorsports',' Front Row (software) ', …]

Thought 3: Front Row is not found. I need to search Front Row (software) .
Act 3: Search[Front Row (software)]
Obs 3: Front Row is a discontinued media center software …

Thought 4: Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.
Act 4: Finish[keyboard function keys] ✓

Yao et al. [2023]

LM-generated program calls

Chain-of-Thought (Wei et al., 2022)

Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 tennis balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The bakers at the Beverly Hills Bakery baked 200 loaves of bread on Monday morning. They sold 93 loaves in the morning and 39 loaves in the afternoon. A grocery store returned 6 unsold loaves. How many loaves of bread did they have left?

Model Output

A: The bakers started with 200 loaves. They sold 93 in the morning and 39 in the afternoon. So they sold 93 + 39 = 132 loaves. The grocery store returned 6 loaves. So they had 200 - 132 - 6 = 62 loaves left.
The answer is 62.

Program-aided Language models (this work)

Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 tennis balls.
`tennis_balls = 5`
2 cans of 3 tennis balls each is
`bought_balls = 2 * 3`
tennis balls. The answer is
`answer = tennis_balls + bought_balls`

Q: The bakers at the Beverly Hills Bakery baked 200 loaves of bread on Monday morning. They sold 93 loaves in the morning and 39 loaves in the afternoon. A grocery store returned 6 unsold loaves. How many loaves of bread did they have left?

Model Output

A: The bakers started with 200 loaves
`loaves_baked = 200`
They sold 93 in the morning and 39 in the afternoon
`loaves_sold_morning = 93`
`loaves_sold_afternoon = 39`
The grocery store returned 6 loaves.
`loaves_returned = 6`
The answer is
```
answer = loaves_baked - loaves_sold_morning
  - loaves_sold_afternoon + loaves_returned
```

```
>>> print(answer)
74
```

Gao et al. [2023], Chen et al.

Interim Prompting Summary

- Including demonstration examples (few-shot reasoning)
- Inciting the model to generate reasoning steps (Chain-of-Thought)
- Decomposing the question and solve with multiple prompts (Least-to-most)
- Multiple steps including reasoning and actions (ReAct)
- Generating programs for reasoning (PAL)

Interim Prompting Summary

- Including demonstration examples (few-shot reasoning)
- Inciting the model to generate reasoning steps (Chain-of-Thought)
- Decomposing the question and solve with multiple prompts (Least-to-most)
- Multiple steps including reasoning and actions (ReAct)
- Generating programs for reasoning (PAL)

Let's embed (L)LMs inside
a programming language 😍

Prompting is programming

A query language for language models: LMQL

```
beam(n=3)
  "A list of good dad jokes. A indicates the "
  "punchline \n"
  "Q: How does a penguin build its house? \n"
  "A: Igloos it together. END \n"
  "Q: Which knight invented King Arthur's Round"
  "Table? \n"
  "A: Sir Cumference. END \n"
  "Q: [JOKE] \n"
  "A: [PUNCHLINE] \n"
from "gpt2-medium"
where
  STOPS_AT(JOKE, "?") and STOPS_AT(PUNCHLINE, "END")
  and len(words(JOKE)) < 20
  and len(characters(PUNCHLINE)) > 10
```

Beurer-Kellner et al. [2023]

What is a programming language for prompting?

The requirements

- A description of the language (syntax)
- The interface to prompting (semantics)

The components

- Multiple variables in the prompt, that need to be filled
- Declarative constraints that generation needs to be constrained by

From query language to prompting

- Iterative generation of each variable,
- Token masking to implement constraints on decoding

The LMQL Language

LMQL Program

⟨decoder⟩ ⟨query⟩
from ⟨model⟩
[**where** ⟨cond⟩]
[**distribute** ⟨dist⟩]

⟨decoder⟩ ::= **argmax** | **beam**(n=⟨int⟩) | **sample**(n=⟨int⟩)
⟨query⟩ ::= ⟨python_statement⟩+
⟨cond⟩ ::= ⟨cond⟩ **and** ⟨cond⟩ | ⟨cond⟩ **or** ⟨cond⟩ | **not** ⟨cond⟩ | ⟨cond_term⟩
| ⟨cond_term⟩ ⟨cond_op⟩ ⟨cond_term⟩
⟨cond_term⟩ ::= ⟨python_expression⟩
⟨cond_op⟩ ::= < | > | = | **in**
⟨dist⟩ ::= ⟨var⟩ **over** ⟨python_expression⟩

Beurer-Kellner et al. [2023]

The query

Core idea a query is (informally) the body of a Python function

- without inner function declarations,
- where top-level strings are LM prompts

The query

Core idea a query is (informally) the body of a Python function

- without inner function declarations,
- where top-level strings are LM prompts

Prompt variables The prompts have two special components:

- `"[varname]"`: a phrase that needs to be generated (a *hole*)
- `"{varname}"`: a reference to a previously generated phrase

The query

Core idea a query is (informally) the body of a Python function

- without inner function declarations,
- where top-level strings are LM prompts

Prompt variables The prompts have two special components:

- `"[varname]"`: a phrase that needs to be generated (a *hole*)
- `"{varname}"`: a reference to a previously generated phrase

Evaluation of a query we iteratively evaluate top-level strings keeping track of:

- an *interaction trace* (all text generated so far)
- a *scope* (variable assignment)

Evaluation of an LMQL program

Algorithm 1: Evaluation of a top-level string s

Input: string s, trace u, scope σ, language model f

if s *contains* $[\langle$*<varname>*$\rangle]$ **then**
  s_{pre}, varname, $s_{\text{post}} \leftarrow \text{unpack}(s)$
    `// e.g. "a [b] c" → "a ", "b", " c"`
  $u \leftarrow us_{\text{pre}}$ `// append to trace`
  $v \leftarrow decode(f,\text{u})$ `// use the LM for the hole`
  $\sigma[\text{varname}] \leftarrow v$ `// updated scope`
  $u \leftarrow uv$ `// append to trace`
else if s *contains* $\{\langle$*varname*$\rangle\}$ **then**
  varname $\leftarrow \text{unpack}(s)$ `// e.g. "{b}" → "b"`
  $v \leftarrow \sigma[\text{varname}]$ `// retrieve value from scope`
  $s \leftarrow \text{subs}(s, \text{varname}, v)$ `// replace placeholder with value`
  $u \leftarrow us$ `// append to trace`
else
  $u \leftarrow us$ `// append to trace`
end

Beurer-Kellner et al. [2023]

No, really, prompting is programming

```
argmax
  "A list of things not to forget when "
  "travelling:\n"
  things = []
  for i in range(2):
    "- [THING]\n"
    things.append(THING)
  "The most important of these is [ITEM]."
from "EleutherAI/gpt-j-6B"
where
   THING in ["passport",
             "phone",
             "keys", ...] // a longer list
   and len(words(ITEM)) <= 2
```

Beurer-Kellner et al. [2023]

Evaluation example

update	state after update
	$u = \epsilon$ $g = \{\}$

Evaluation example

update	state after update
	$u = \epsilon$ $g = \{\}$
$s \leftarrow$ `"A list of things not to forget when"` $u \leftarrow us$	$u =$ `"A list of things not to forget when"` $g = \{\}$

Evaluation example

update	state after update
	$u = \epsilon$ $g = \{\}$
$s \leftarrow$ `"A␣list␣of␣things␣not␣to␣forget␣when"` $u \leftarrow us$	$u =$`"A␣list␣of␣things␣not␣to␣forget␣when"` $g = \{\}$
$s \leftarrow$ `"travelling:␣\n"` $u \leftarrow us$	$u =$`"A␣list␣of␣things␣not␣to␣forget␣when travelling␣\n"` $g = \{\}$

Evaluation example

update	state after update
	$u = \epsilon$ $g = \{\}$
$s \leftarrow$ `"A␣list␣of␣things␣not␣to␣forget␣when"` $u \leftarrow us$	$u =$ `"A␣list␣of␣things␣not␣to␣forget␣when"` $g = \{\}$
$s \leftarrow$ `"travelling:␣\n"` $u \leftarrow us$	$u =$ `"A␣list␣of␣things␣not␣to␣forget␣when travelling␣\n"` $g = \{\}$
$s \leftarrow$ `"-␣[THING]\n"` $s_{\text{pre}}, \text{varname}, s_{\text{post}} \leftarrow$ `"-␣"`, `THING`, `\n` $u \leftarrow us_{\text{pre}}$ $v \leftarrow$ `"sun␣screen"` $=$ `decode(`f`,` u`)` $u \leftarrow uvs_{\text{post}}$ $g[\text{varname}] \leftarrow v$	$u =$ `"A␣list␣of␣things␣not␣to␣forget␣when travelling␣\n` `-␣sun␣screen\n"` $g = \{i = 0, \text{THING} =$ `"sun␣screen"`, $\text{things} =$ `["sun␣screen"]`$\}$

Evaluation example

update	state after update
	$u = \epsilon$ $g = \{\}$
$s \leftarrow$ `"A␣list␣of␣things␣not␣to␣forget␣when"` $u \leftarrow us$	$u =$ `"A␣list␣of␣things␣not␣to␣forget␣when"` $g = \{\}$
$s \leftarrow$ `"travelling:␣\n"` $u \leftarrow us$	$u =$ `"A␣list␣of␣things␣not␣to␣forget␣when travelling␣\n"` $g = \{\}$
$s \leftarrow$ `"-␣[THING]\n"` $s_{\text{pre}}, \text{varname}, s_{\text{post}} \leftarrow$ `"-␣", THING, \n` $u \leftarrow us_{\text{pre}}$ $v \leftarrow$ `"sun␣screen"` $=$ `decode(`f`, `u`)` $u \leftarrow uvs_{\text{post}}$ $g[\text{varname}] \leftarrow v$	$u =$ `"A␣list␣of␣things␣not␣to␣forget␣when travelling␣\n` `-␣sun␣screen\n"` $g = \{i = 0, \text{THING} =$ `"sun␣screen",` $\text{things} =$ `["sun␣screen"]`$\}$

$s \leftarrow$ `"-␣[THING]\n"`
$s_{\text{pre}}, \text{varname}, s_{\text{post}} \leftarrow$ `"-␣", THING, \n`
$u \leftarrow us_{\text{pre}}$
$v \leftarrow$ `"beach␣towel"` $=$ `decode(`f`, `u`)`
$u \leftarrow uvs_{\text{post}}$
$g[\text{varname}] \leftarrow v$
	$u =$ `"A␣list␣of␣things␣not␣to␣forget␣when travelling␣\n`
	`-␣sun␣screen\n`
	`-␣beach␣towel\n"`
	$g = \{i = 1, \text{THING} =$ `"beach␣towel",`
	$\text{things} =$ `["sun␣screen", "beach␣towel"]`$\}$

Distribution

Core idea grabbing the probability distribution of the LM for the last variable, in a given set

Distribution

Core idea grabbing the probability distribution of the LM for the last variable, in a given set

```
argmax
  "A list of things not to forget when "
  "travelling:\n"
  things = []
  for i in range(2):
    "- [THING]\n"
    things.append(THING)
  "The most important of these is [ITEM]."
from "EleutherAI/gpt-j-6B"
where
   THING in ["passport",
             "phone",
             "keys", ...] // a longer list
distribute ITEM over things
```

⇝

```
A list of things not to forget when travelling:
- sun screen
- beach towel
The most important of these is { sun screen  65%
                               { beach towel  35%
```

Constraints

Constraint syntax

```
⟨cond⟩ ::= ⟨cond⟩ and ⟨cond⟩ | ⟨cond⟩ or ⟨cond⟩ | not ⟨cond⟩ | ⟨cond_term⟩
         | ⟨cond_term⟩ ⟨cond_op⟩ ⟨cond_term⟩
⟨cond_term⟩ ::= ⟨python_expression⟩
⟨cond_op⟩ ::= < | > | = | in
```

*deterministic pure python functions + constants

Built-in functions

$[w_1, \ldots w_k] \leftarrow$ `words`$(\langle \text{var} \rangle)$ //splits $\langle \text{var} \rangle$ into words $w_1, \ldots w_k$

$[s_1, \ldots s_k] \leftarrow$ `sentences`$(\langle \text{var} \rangle)$ //splits $\langle \text{var} \rangle$ into sentences $s_1, \ldots s_k$

$b \leftarrow$ `stop_at`$(\langle \text{var} \rangle, \text{t})$ //indicates if $\langle \text{var} \rangle$ ends in token or string t

Constraints

Constraint syntax

```
⟨cond⟩ ::= ⟨cond⟩ and ⟨cond⟩ | ⟨cond⟩ or ⟨cond⟩ | not ⟨cond⟩ | ⟨cond_term⟩
         | ⟨cond_term⟩ ⟨cond_op⟩ ⟨cond_term⟩
⟨cond_term⟩ ::= ⟨python_expression⟩
⟨cond_op⟩ ::= < | > | = | in
```

*deterministic pure python functions + constants

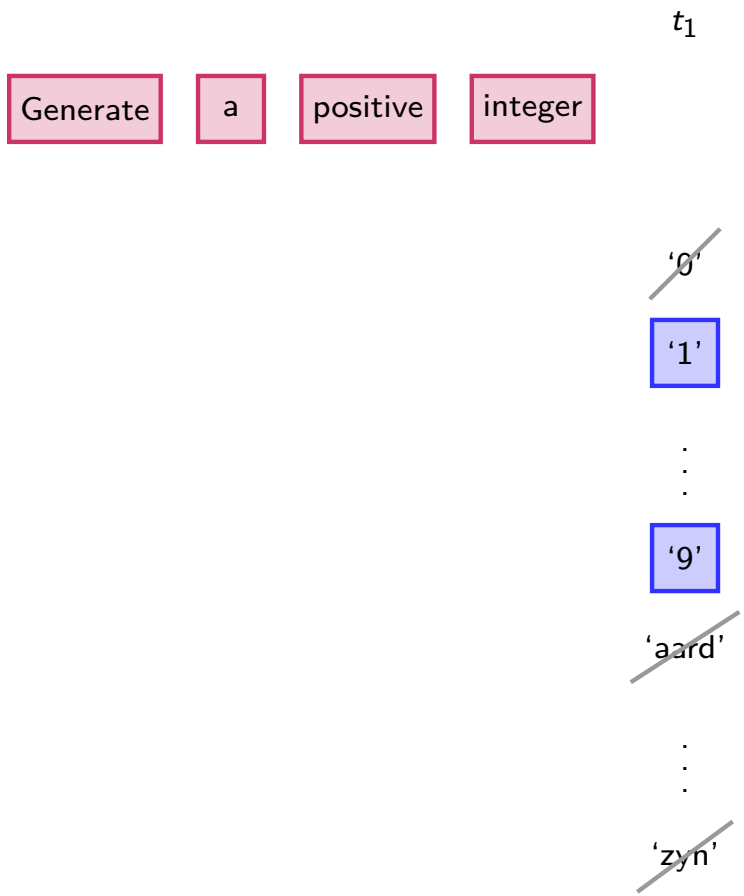
Built-in functions

$[w_1, \ldots w_k] \leftarrow$ `words`$(\langle \text{var} \rangle)$ //splits $\langle \text{var} \rangle$ into words $w_1, \ldots w_k$
$[s_1, \ldots s_k] \leftarrow$ `sentences`$(\langle \text{var} \rangle)$ //splits $\langle \text{var} \rangle$ into sentences $s_1, \ldots s_k$
$b \leftarrow$ `stop_at`$(\langle \text{var} \rangle, \text{t})$ //indicates if $\langle \text{var} \rangle$ ends in token or string t

Constraint validation how to evaluate expressions over to-be-generated strings?

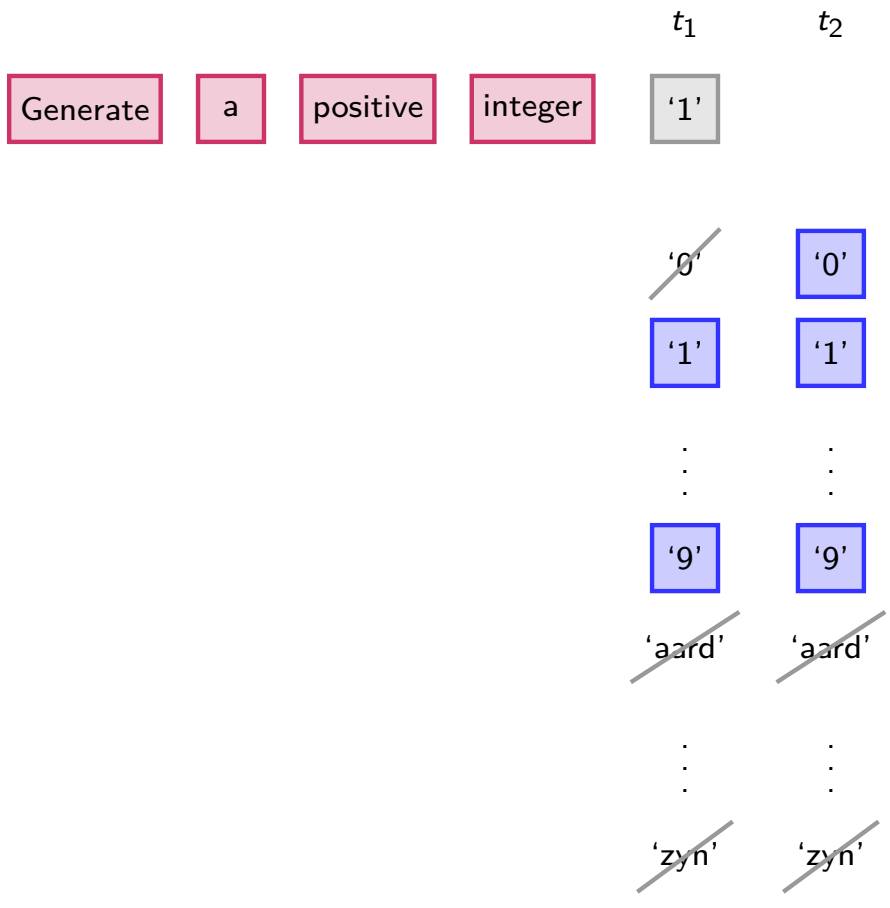
Masked Decoding

Core idea block out part of the vocabulary during decoding



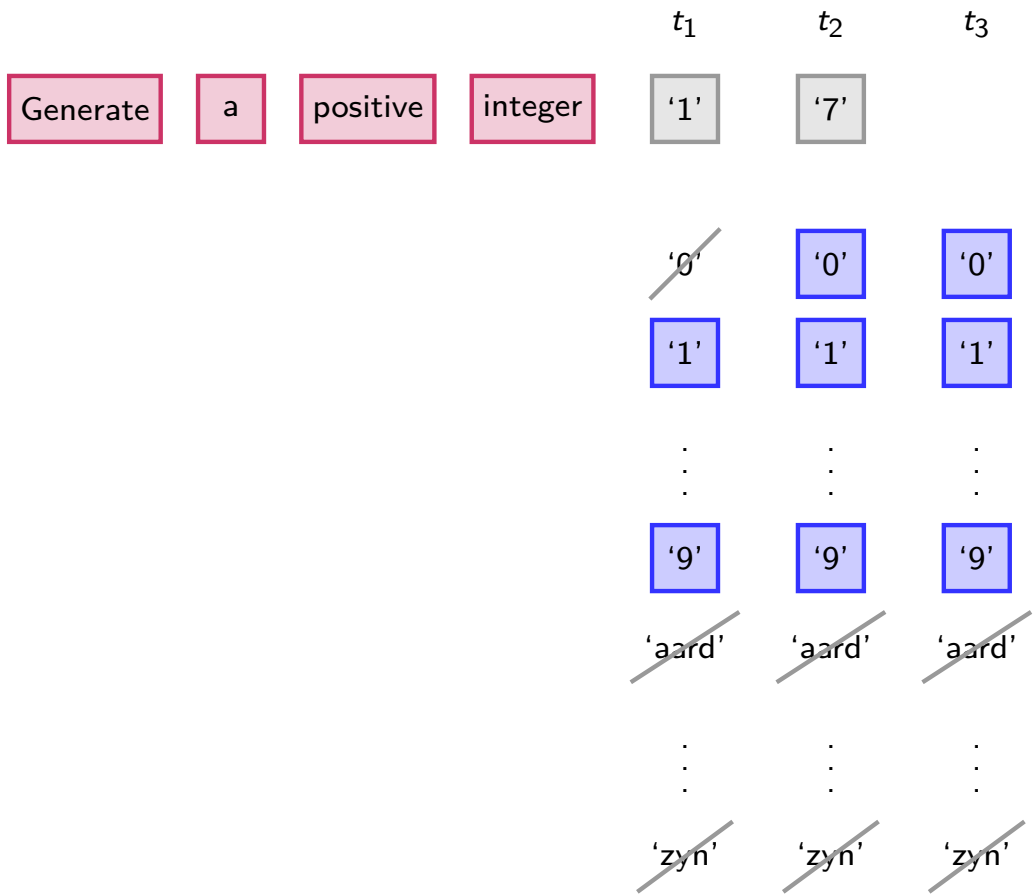
Masked Decoding

Core idea block out part of the vocabulary during decoding



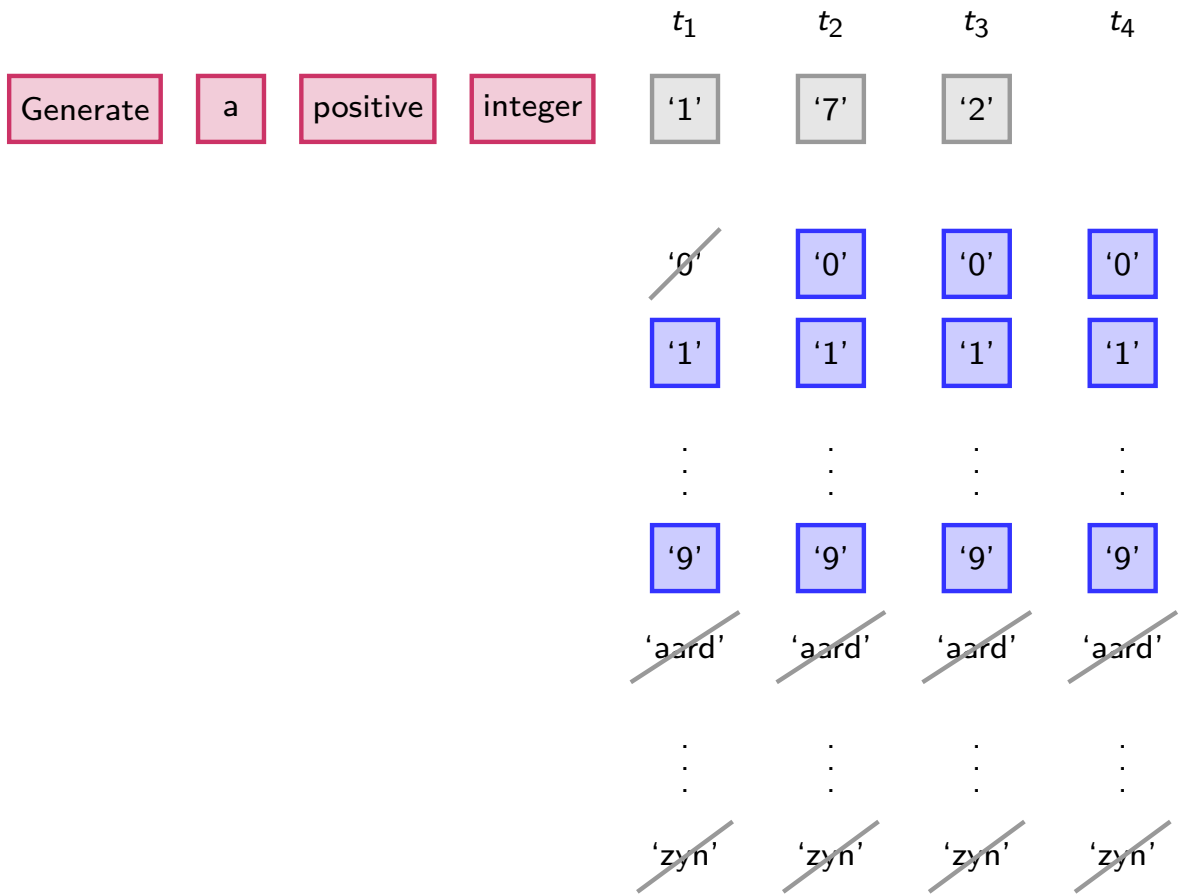
Masked Decoding

Core idea block out part of the vocabulary during decoding



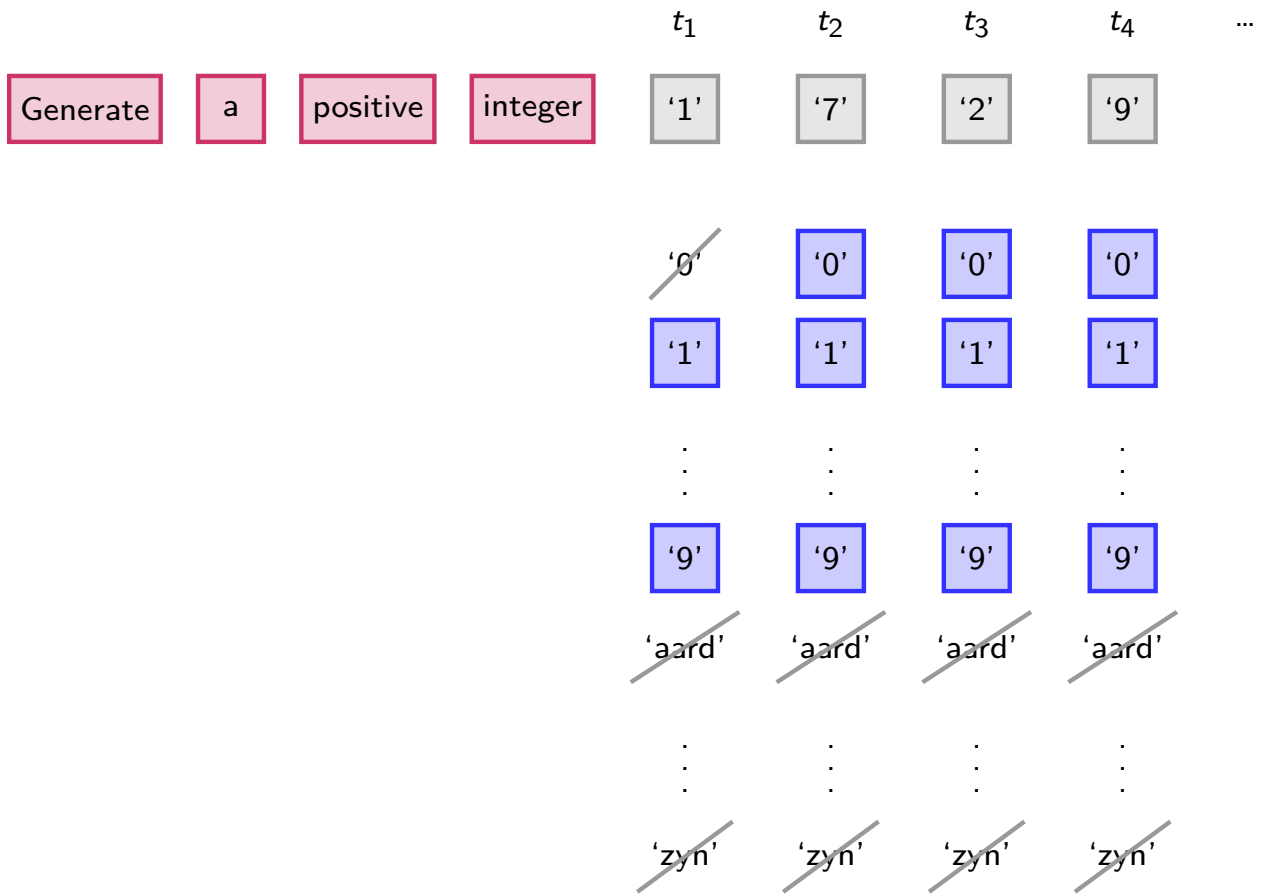
Masked Decoding

Core idea block out part of the vocabulary during decoding



Masked Decoding

Core idea block out part of the vocabulary during decoding



Constrained decoding with masks

Algorithm 2: Decoding

Input: trace u, scope σ, LM f
Output: decoded sequence v

$v \leftarrow \epsilon$
while ***True*** **do**
  $\boldsymbol{m} \leftarrow \text{compute\_mask}(u, \sigma, v)$
  if $\bigwedge_i (m_i = 0)$ **then break**
  $\boldsymbol{z} \leftarrow {}^1/_Z \cdot \boldsymbol{m} \odot \text{softmax}(\boldsymbol{f}(uv))$
  $t \leftarrow \text{pick}(z)$
  if $t = \textit{EOS}$ **then break**
  $v \leftarrow vt$
end

Constrained decoding with masks

Algorithm 2: Decoding

Input: trace u, scope σ, LM f
Output: decoded sequence v

$v \leftarrow \epsilon$
while ***True*** **do**
  $\boldsymbol{m} \leftarrow \text{compute\_mask}(u, \sigma, v)$
  if $\bigwedge_i (m_i = 0)$ **then break**
  $\boldsymbol{z} \leftarrow {}^1/_Z \cdot \boldsymbol{m} \odot \text{softmax}(\boldsymbol{f}(uv))$
  $t \leftarrow \text{pick}(z)$
  if $t = \textit{EOS}$ **then break**
  $v \leftarrow vt$
end

Why not just check after generation?

Beurer-Kellner et al. [2023]

Naive constrained decoding

Algorithm 3: Naive Decoding with Constraints

Input: trace u, scope σ, language model f
Output: decoded sequence v

Function *decode_step*(f, u, v)
  $\boldsymbol{z} \leftarrow \text{softmax}(\boldsymbol{f}(uv))$
  $\boldsymbol{m} \leftarrow \mathbf{1}^{|\mathcal{V}|}$
  do
    $t \leftarrow \text{pick}(1/Z \cdot \boldsymbol{m} \odot \boldsymbol{z})$
    if $t \neq \textit{EOS}$ **then** decode_step(f, u, vt)
    else if $t = \textit{EOS} \wedge \textit{check}(u, vt)$ **then**
      return v
    else $\boldsymbol{m}[t] \leftarrow 0$
  while $\bigvee_i m_i = 1$
decode_step(f, u, ϵ)

Beurer-Kellner et al. [2023]

Incrementalizing constraints

Requirement we need to compile constraints into a set of invalid tokens (the token mask)

Incrementalizing constraints

Requirement we need to compile constraints into a set of invalid tokens (the token mask)

Partial evaluation to model constraint satisfaction on sequences that are not yet fully generated

Incrementalizing constraints

Requirement we need to compile constraints into a set of invalid tokens (the token mask)

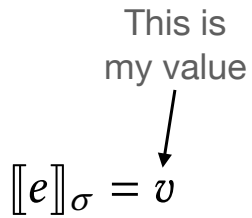
Partial evaluation to model constraint satisfaction on sequences that are not yet fully generated

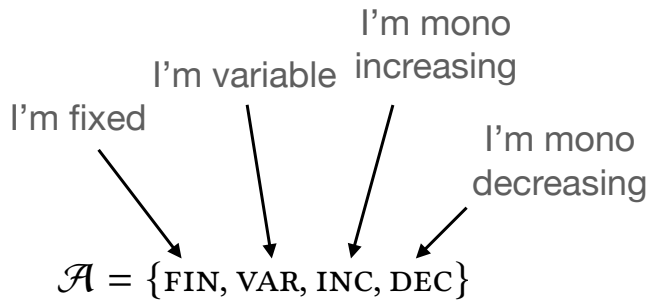
Final and Follow

- Final semantics: does an *expression* evaluate to a fixed value or not?
- Follow semantics: given an *expression* e, an *interaction trace* u and a *token* t, is ut valid?

Final Semantics

Values & Annotators





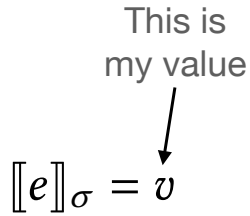
Base case

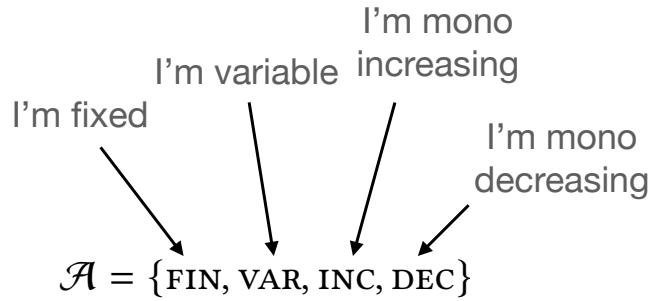
expression	$\textsc{Final}[\cdot\,;\sigma]$
$\langle$const$\rangle$	FIN
python variable $\langle$pyvar$\rangle$	VAR
previous hole $\langle$var$\rangle$	FIN
current var $\langle$var$\rangle$	INC
future hole $\langle$var$\rangle$	INC
`words(`v`)`	$\textsc{Final}[v]$
`sentences(`v`)`	$\textsc{Final}[v]$
`len(`v`)`	$\textsc{Final}[v]$

Beurer-Kellner et al. [2023]

Final Semantics

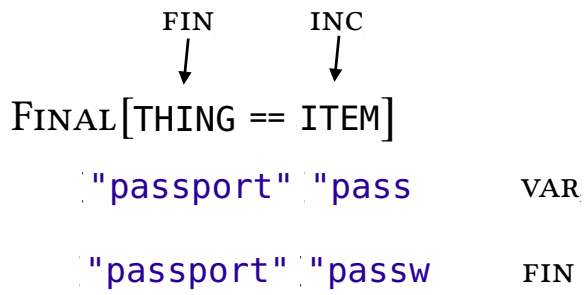
Values & Annotators





Equality & Functions

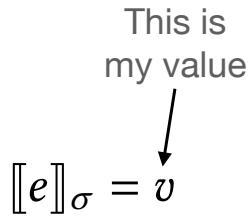
expression	$\textsc{Final}[\cdot\,;\sigma]$
number equality $n == m$	$\begin{cases} \text{FIN} & \text{if } \textsc{Final}[n] = \text{FIN} \\ & \quad \wedge\ \textsc{Final}[m] = \text{FIN} \\ \text{VAR} & \text{else} \end{cases}$
string equality $x == y$	$\begin{cases} \text{FIN} & \text{if } \textsc{Final}[x] = \text{FIN} \\ & \quad \wedge\ \textsc{Final}[y] = \text{FIN} \\ \text{FIN} & \exists i \bullet x[i] \neq y[i] \\ & \quad \wedge\ \textsc{Final}[x] \neq \text{VAR} \\ & \quad \wedge\ \textsc{Final}[y] \neq \text{VAR} \\ \text{VAR} & \text{else} \end{cases}$
function $\texttt{fn}(\tau_1, \ldots, \tau_k)$	$\begin{cases} \text{FIN} & \text{if } \bigwedge_{i=1}^{k} \textsc{Final}[\tau_i] = \text{FIN} \\ \text{VAR} & \text{else} \end{cases}$

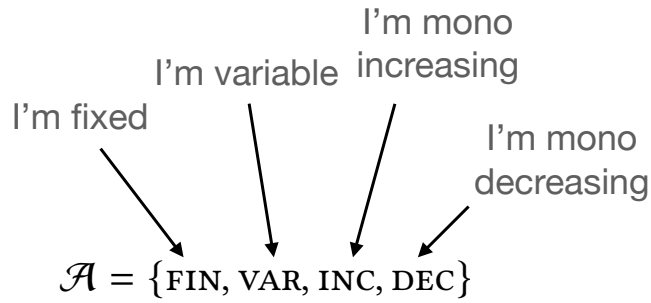


Beurer-Kellner et al. [2023]

Final Semantics

Values & Annotators





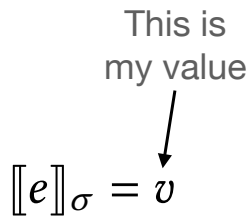
Stop_at & Membership

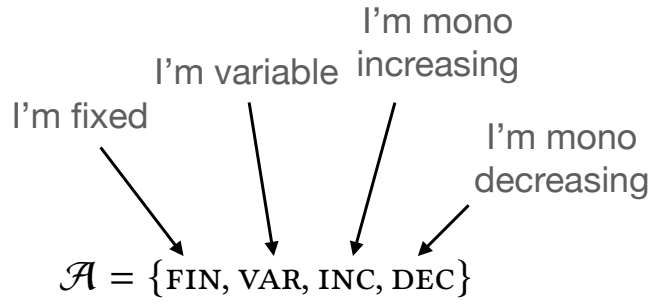
expression	$\textsc{Final}[\cdot\,;\sigma]$
`stop_at(var, s)`	$\begin{cases} \text{FIN} & \text{if } [\![var]\!]_\sigma\texttt{.endswith}(s) \\ & \quad \wedge \textsc{Final}[var] = \text{INC} \\ \text{VAR} & \text{else} \end{cases}$
x `in` s for strings x, s	$\begin{cases} \text{FIN} & \text{if } x \texttt{ in } s \wedge \textsc{Final}[x] = \text{FIN} \\ & \quad \wedge \textsc{Final}[s] = \text{INC} \\ \text{VAR} & \text{else} \end{cases}$
e `in` l for string e, set l	$\begin{cases} \text{FIN} & \text{if } \nexists i \in l \bullet i\texttt{.startswith(e)} \\ & \quad \wedge \textsc{Final}[x] \in \{\text{INC}, \text{FIN}\} \\ & \quad \wedge \textsc{Final}[l] = \text{FIN} \\ \text{VAR} & \text{else} \end{cases}$

Beurer-Kellner et al. [2023]

Final Semantics

Values & Annotators





Inequality & Booleans

expression	$\textsc{Final}[\cdot\,;\sigma]$
$x < y$	$\begin{cases} \text{FIN} & \text{if } x<y \wedge \textsc{Final}[x] \in \{\text{DEC}, \text{FIN}\} \\ & \quad \wedge\ \textsc{Final}[y] \in \{\text{INC}, \text{FIN}\} \\ \text{VAR} & \text{else} \end{cases}$
a and b	$\begin{cases} \text{FIN} & \text{if } \exists v \in \{a,b\} \bullet [\![v]\!]^F_\sigma = \text{FIN}(\bot) \\ \text{FIN} & \text{if } \forall v \in \{a,b\} \bullet [\![v]\!]^F_\sigma = \text{FIN}(\top) \\ \text{VAR} & \text{else} \end{cases}$
a or b	$\begin{cases} \text{FIN} & \text{if } \exists v \in \{a,b\} \bullet [\![v]\!]^F_\sigma = \text{FIN}(\top) \\ \text{FIN} & \text{if } \forall v \in \{a,b\} \bullet [\![v]\!]^F_\sigma = \text{FIN}(\bot) \\ \text{VAR} & \text{else} \end{cases}$
not a	$\textsc{Final}[a]$

Beurer-Kellner et al. [2023]

Follow Maps

Core idea Can we exclude token t for a given expression, and interaction trace u?

Follow Maps

Core idea Can we exclude token t for a given expression, and interaction trace u?

Some example definitions

expression	$\textbf{FOLLOW}[\cdot](u, t)$
$\langle \text{const} \rangle$	$[\![\langle \text{const} \rangle]\!]_\sigma$
python variable $\langle \text{pyvar} \rangle$	$[\![\texttt{pyvar}]\!]_{\sigma[v \leftarrow vt]}$
previous hole $\langle \text{var} \rangle$	$[\![\langle \text{var} \rangle]\!]_\sigma$
current var v	$\begin{cases} \textsc{fin}(v) & \text{if } t = \textsc{eos} \\ \textsc{inc}(vt) & \text{else} \end{cases}$
future hole $\langle \text{var} \rangle$	`None`
`sentences(`v`)`	$\begin{cases} \textsc{fin}(s_1, \ldots, s_k) & \text{if } t = \textsc{eos} \\ \textsc{inc}(s_1, \ldots, s_k, t) & \text{if } s_k\texttt{.endswith(".")} \\ \textsc{inc}(s_1, \ldots, s_k t) & \text{else} \end{cases}$ where $s_1, \ldots, s_k \leftarrow [\![\texttt{sentences(}v\texttt{)}]\!]_\sigma$

Beurer-Kellner et al. [2023]

Follow Maps

Core idea Can we exclude token t for a given expression, and interaction trace u?

Membership

expression	$\textbf{Follow}[\cdot](u,t)$
`x in` l for constant list/set l	$\begin{cases} \textsc{fin}(\top) & \text{if } \texttt{t in l} \\ \textsc{var}(\bot) & \text{if } \exists e \in l \bullet \\ & \quad \texttt{e.startswith(}vt\texttt{)} \\ \textsc{fin}(\bot) & \text{else} \end{cases}$

Hawking

$$\textsc{Follow}[\texttt{TEXT in ["Stephen Hawking"]}](\texttt{"Steph","en"}) = \textsc{var}(\bot)$$

$$\textsc{Follow}[\texttt{TEXT in ["Stephen Hawking"]}](\texttt{"Steph","anie"}) = \textsc{fin}(\bot)$$

$$\textsc{Follow}[\texttt{TEXT in ["Stephen Hawking"]}](\texttt{"Steph"}, t) = \begin{cases} \textsc{fin}(\top) & \text{if } t = \texttt{"en Hawking"} \\ \textsc{fin}(\bot) & \text{else} \end{cases}$$

Beurer-Kellner et al. [2023]

Sound approximation

Token maps using the Follow semantics we can decide on tokens that will definitely invalidate the constraints
Soundness

$$\forall t \in \mathcal{V} \bullet (\textsc{Follow}[e])(u, t) = \textsc{fin}(\bot) \Rightarrow [\![e]\!]_{\sigma[v \leftarrow ut]} = \textsc{fin}(\bot)$$

In other words we are guaranteed to never mask out valid tokens, but will catch as many invalid tokens as possible

Sound approximation

Token maps using the Follow semantics we can decide on tokens that will definitely invalidate the constraints
Soundness

$$\forall t \in \mathcal{V} \bullet (\textsc{Follow}[e])(u, t) = \textsc{fin}(\bot) \Rightarrow [\![e]\!]_{\sigma[v \leftarrow ut]} = \textsc{fin}(\bot)$$

In other words we are guaranteed to never mask out valid tokens, but will catch as many invalid tokens as possible

But how well does it actually work?

Experimental Evaluation

Chain of Thought as an LQML program

```
argmax
    "Pick the odd word out: skirt, dress, pen, jacket.\n"
    "skirt is clothing, dress is clothing, pen is an object, jacket is clothing.\n"
    "So the odd one is pen.\n\n"
    "Pick the odd word out: Spain, France, German, England, Singapore.\n"
    "Spain is a country, France is a country, German is a language, ...\n"
    "So the odd one is German.\n\n"
    "Pick the odd word out: {OPTIONS}\n"
    "[REASONING]"
    "[RESULT]"
from "EleutherAI/gpt-j-6B"
where
    not "\n" in REASONING and not "Pick" in REASONING and
    stops_at(REASONING, "Pick the odd word") and stops_at(REASONING, "\n") and
    stops_at(REASONING, "So the odd one") and stops_at(REASONING, ".") and
    len(WORDS(REASONING)) < 40
 distribute
    RESULT over OPTIONS.split(", ")
```

Beurer-Kellner et al. [2023]

ReAct as an LQML program

```
import wikipedia_utils
sample(no_repeat_ngram_size=3)
    "What is the elevation range for the area that the eastern sector of the Colorado orogeny extends into?"
    "Tho 1: I need to search Colorado orogeny, find the area that the eastern sector of the Colorado ...\n"
    "Act 2: Search 'Colorado orogeny'\n"
    "Obs 2: The Colorado orogeny was an episode of mountain building (an orogeny) ...\n"
    "Tho 3: It does not mention the eastern sector.  So I need to look up eastern sector.\n"
    ...
    "Tho 4: High Plains rise in elevation from around 1,800 to 7,000 ft, so the answer is 1,800 to 7,000 ft."
    "Act 5: Finish '1,800 to 7,000 ft'"
    "Where is Apple Computers headquartered?\n"
    for i in range(1024):
        "[MODE] {i}:"
        if MODE == "Tho":
            "[THOUGHT] "
        elif MODE == "Act":
            " [ACTION] '[SUBJECT]\n"
            if ACTION == "Search":
                result = wikipedia_utils.search(SUBJECT[:-1]) # cutting of the consumed '
                "Obs {i}: {result}\n"
            else:
                break # action must be FINISH
from "gpt2-xl"
where
    MODE in ["Tho", "Act"] and stops_at(THOUGHT, "\n") and
    ACTION in ["Search", "Finish"] and len(words(THOUGHT)) > 2 and
    stops_at(SUBJECT, "'") and not "Tho" in THOUGHT
```

Beurer-Kellner et al. [2023]

Experimental Comparison

	GPT-J-6B[27]				OPT-30B [34]			
	Standard Decoding	**LMQL**	**Δ**	**Est. Cost Savings**	**Standard Decoding**	**LMQL**	**Δ**	**Est. Cost Savings**
Odd One Out								
Accuracy	33.33%	**34.52%**	1.19%		**34.52%**	**34.52%**	0.00%	
Decoder Calls	7.96	**5.96**	-25.11%		7.96	**5.96**	-25.11%	
Model Queries	73.04	**41.51**	-43.16%		73.04	**40.70**	-44.27%	
Billable Tokens	1178.71	**861.32**	-26.93%	0.63¢/query	1173.21	**856.17**	-27.02%	0.63¢/query

Beurer-Kellner et al. [2023]

Conclusion

Prompting is programming we can embed LM calls inside of a programming language

Constraint validation we need to translate constraints into a token masking strategy

Sound approximation The translation in LMQL is sound but not complete, i.e. we don't mask out all invalid tokens

The cliffhanger

But what if we want to generate

The cliffhanger

But what if we want to generate *structured output* 😍?

The cliffhanger

But what if we want to generate *structured output* 😍?

- JSON 😍?

The cliffhanger

But what if we want to generate *structured output* 😍?

- JSON 😍?
- Python 😍😍?

The cliffhanger

But what if we want to generate *structured output* 😍?

- JSON 😍?
- Python 😍😍?
- First-order logic 😍😍😍?

The cliffhanger

But what if we want to generate *structured output* 😍?

- JSON 😍?
- Python 😍😍?
- First-order logic 😍😍😍?

Tomorrow!

Thank you.

References I

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rygGQyrFvH.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

References II

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35: 24824–24837, 2022.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, et al. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*, 2023.