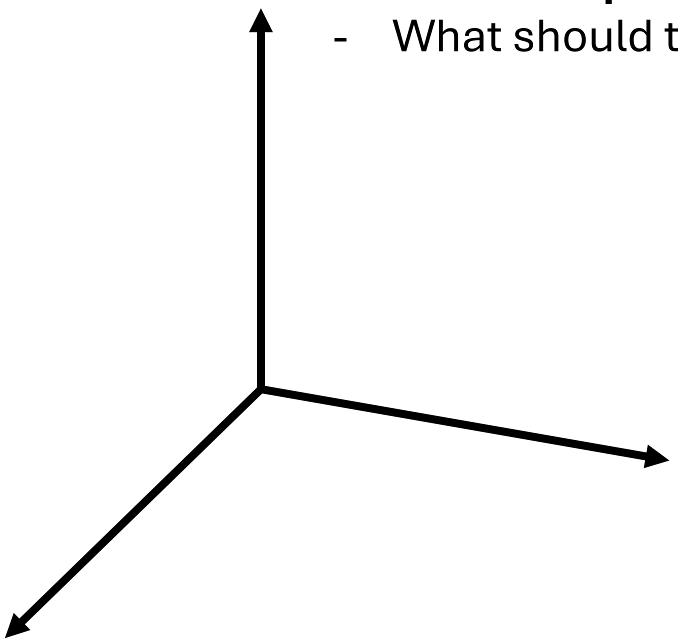
Machine Programming

Lecture 8 – Controlled Decoding and Steering

Ziyang Li

Dimensions in Program Synthesis

Behavioral Specification
- What should the program do?



Synthesis Strategy
- How do we find such a program?

Structural Specification
- What is the space of the programs?

The Course So Far

Behavioral Specification
- What should the program do?

1. Examples
2. Types
3. Functional Specifications
4. Natural Language

Synthesis Strategy
- How do we find such a program?

Enumeration
- Enumerating all programs with a grammar
- Bottom-up vs top-down

Data-Driven Approaches
- Next token prediction, greedy decoding and sampling
- Prompting language models, Iterative Refinement

Structural Specification
- What is the space of the programs?

General Purpose Programming Language
Python / Java / C / Rust / ...

Domain Specific Languages

Today

Behavioral Specification
- What should the program do?

1. Examples
2. Types
3. Functional Specifications
4. Natural Language

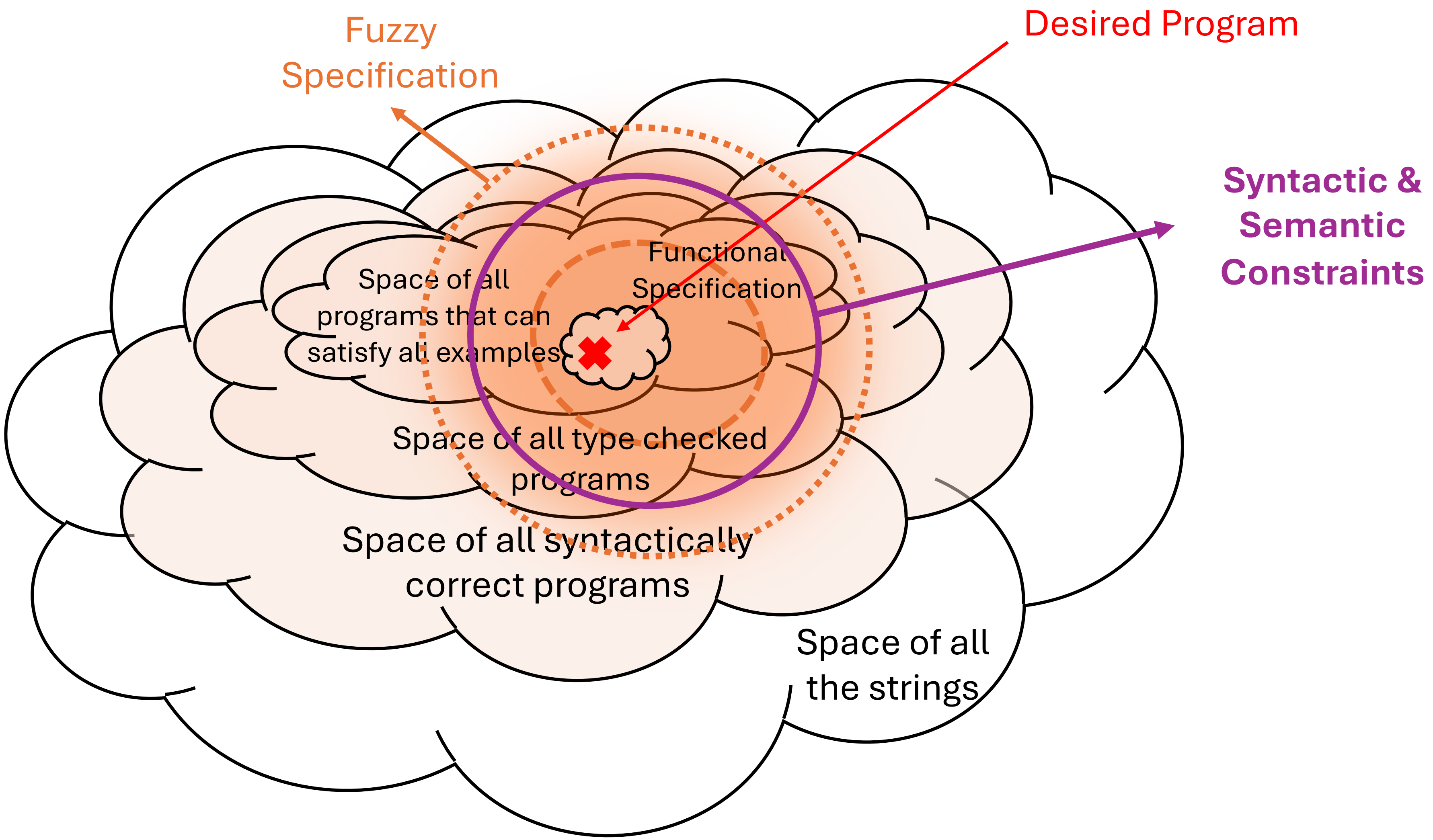
Structural Specification
- What is the space of the programs?

General Purpose Programming Language
Python / Java / C / Rust / …

Synthesis Strategy
- How do we find such a program?

Controlled Decoding
- Grammar constraints, programmatic constraints
- Syntactic and semantic potentials, monitor feedback
- Programmatic and Structural steering

High Level Picture



High Level Picture

- We studied foundations of programming languages and synthesis
  - Syntax (regular tree grammar), Semantics (denotational, operational)
  - Enumeration (top-down, bottom-up) that searches within grammar
- We studied basic usage of language models
  - Purely neural: next token prediction → sequential decoding
  - Prompting and iterative refinement: elicit better programs from LLM
- Question:
  - Can we inject more **structure** during **neural** generation process

Review: Syntax in Regular Tree Grammar

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

Review: Syntax in Regular Tree Grammar

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

→

```
(Program) P ::= L | N
   (List) L ::= input
              | []
              | [N]
              | L ++ L
 (Number) N ::= len(L)
              | min(L)
              | N + N
              | 0 | 1 | 2 | …
```

Review: Syntax in Context Free Grammar

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

→

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```

From Program to Abstract Syntax Tree

Concrete Program

```
min(input ++ [0])
```

Concrete Grammar (Context Free Grammar)

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```

Abstract Grammar (Regular Tree Grammar)

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

From Program to Abstract Syntax Tree

Concrete Program

`min(input ++ [0])`

Token Sequence

`['min', '(', 'input', '++', '[', '0', ']']`

Concrete Grammar (Context Free Grammar)

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```

Abstract Grammar (Regular Tree Grammar)

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

From Program to Abstract Syntax Tree

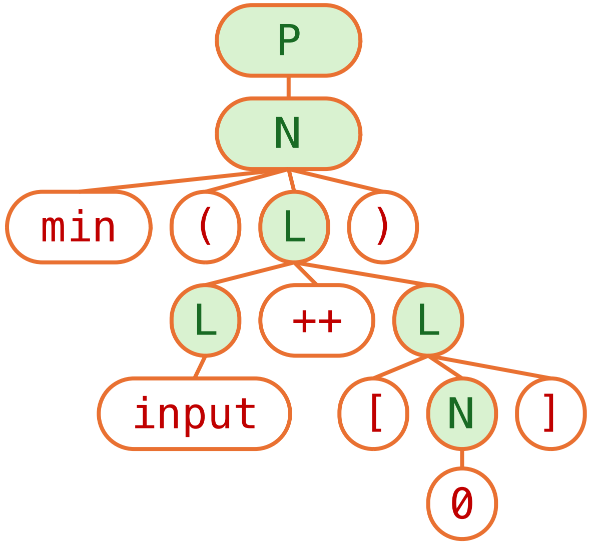
Concrete Program

`min(input ++ [0])`

Token Sequence

`[‘min’, ‘(’, ‘input’, ‘++’, ‘[’, ‘0’, ‘]’]`

Concrete Syntax Tree



Concrete Grammar (Context Free Grammar)

```
(Program) P <- L | N
   (List) L <- ‘input’
             | ‘[‘ ‘]’
             | ‘[‘ N ‘]’
             | L ‘++’ L
 (Number) N <- ‘len’ ‘(‘ L ‘)’
             | ‘min’ ‘(‘ L ‘)’
             | N ‘+’ N
             | ‘0’ | ‘1’ | ‘2’ | …
```

Abstract Grammar (Regular Tree Grammar)

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

From Program to Abstract Syntax Tree

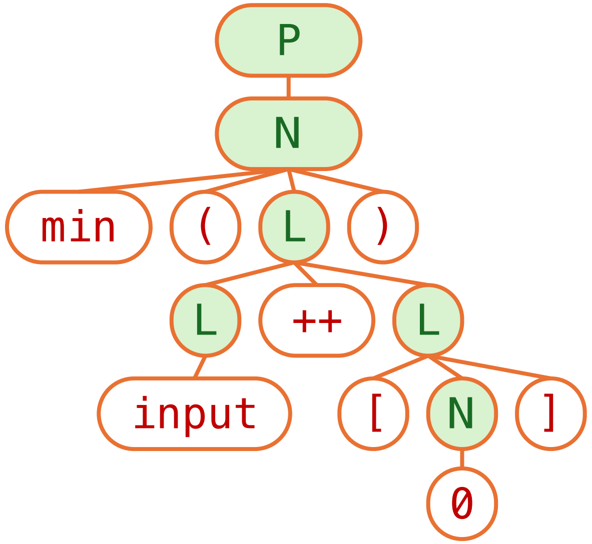
Concrete Program

`min(input ++ [0])`

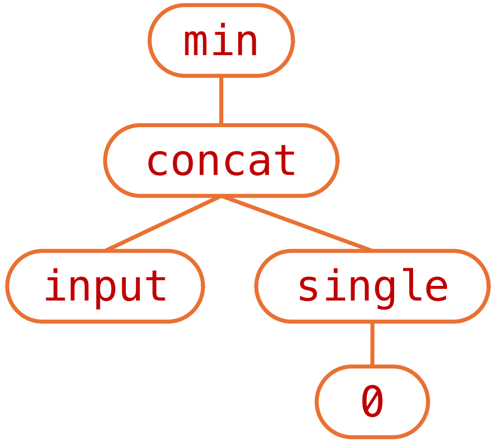
Token Sequence

`[‘min’, ‘(’, ‘input’, ‘++’, ‘[’, ‘0’, ‘]’]`

Concrete Syntax Tree



Abstract Syntax Tree



Concrete Grammar (Context Free Grammar)

```
(Program) P <- L | N
   (List) L <- ‘input’
             | ‘[‘ ‘]’
             | ‘[‘ N ‘]’
             | L ‘++’ L
 (Number) N <- ‘len’ ‘(‘ L ‘)’
             | ‘min’ ‘(‘ L ‘)’
             | N ‘+’ N
             | ‘0’ | ‘1’ | ‘2’ | …
```

Abstract Grammar (Regular Tree Grammar)

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

From Program to Abstract Syntax Tree

Concrete Program

`min(input ++ [0])`

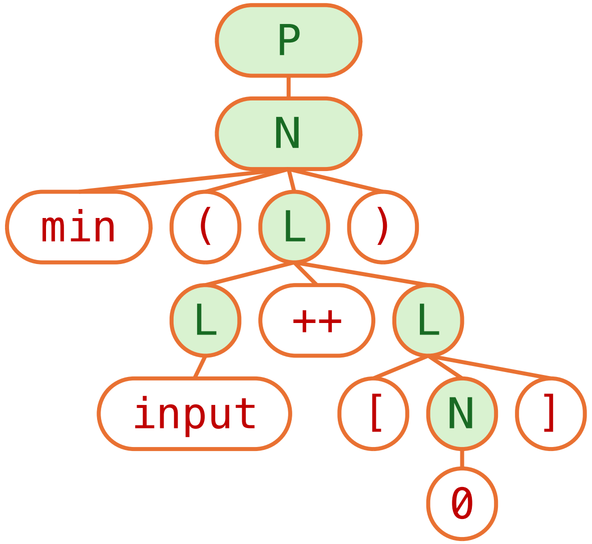
Lexer

Token Sequence

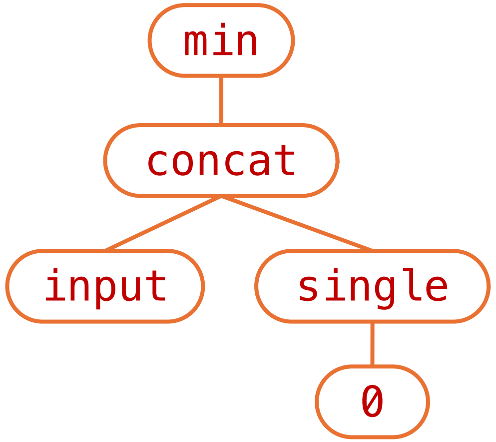
`[‘min’, ‘(’, ‘input’, ‘++’, ‘[’, ‘0’, ‘]’]`

Parser

Concrete Syntax Tree



Abstract Syntax Tree



Concrete Grammar (Context Free Grammar)

```
(Program) P <- L | N
   (List) L <- ‘input’
             | ‘[‘ ‘]’
             | ‘[‘ N ‘]’
             | L ‘++’ L
 (Number) N <- ‘len’ ‘(‘ L ‘)’
             | ‘min’ ‘(‘ L ‘)’
             | N ‘+’ N
             | ‘0’ | ‘1’ | ‘2’ | …
```

Abstract Grammar (Regular Tree Grammar)

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
              | 0 | 1 | 2 | …
```

Lexer and Parser

Concrete Program

`min(input ++ [0])`

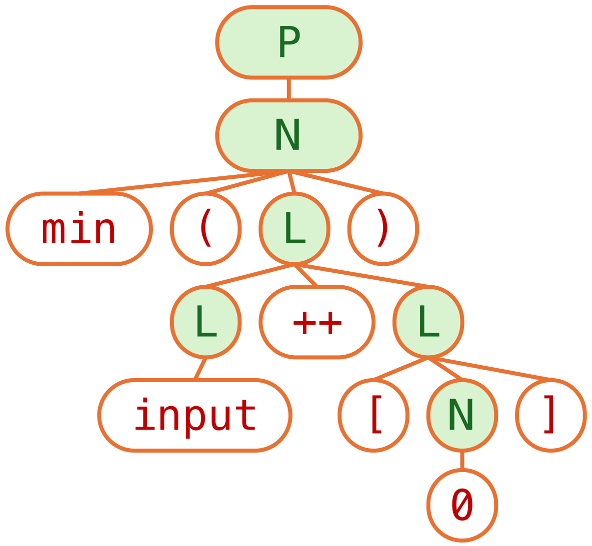
Lexer

Token Sequence

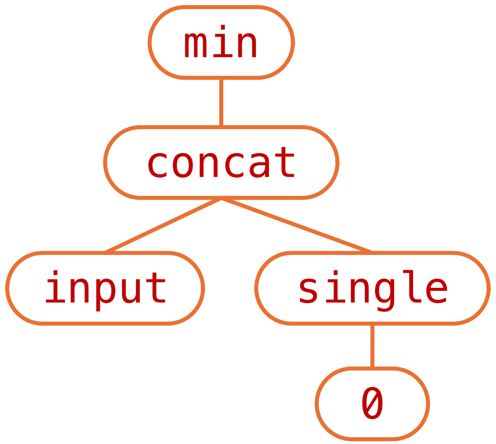
`['min', '(', 'input', '++', '[', '0', ']']`

Parser

Concrete Syntax Tree



Abstract Syntax Tree



I have a concrete grammar in CFG

(Program) P <- L | N
(List) L <- 'input'
| '[' ']'
| '[' N ']'
| L '++' L
(Number) N <- 'len' '(' L ')'
| 'min' '(' L ')'
| N '+' N
| '0' | '1' | '2' | ...

Corresponding to it there is an abstract grammar

(Program) P ::= L | N
(List) L ::= input
| empty
| single(N)
| concat(L, L)
(Number) N ::= len(L)
| min(L)
| add(N, N)
| 0 | 1 | 2 | ...

Can you help me write a treesitter parser in javascript?

Lexer and Parser

Concrete Program

`min(input ++ [0])`

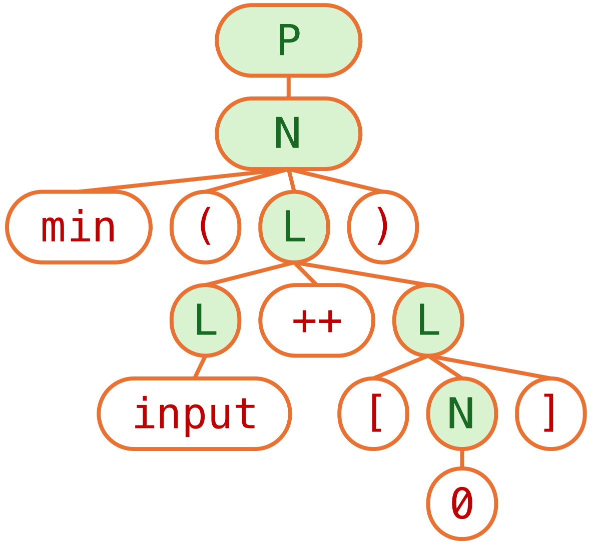
Lexer

Token Sequence

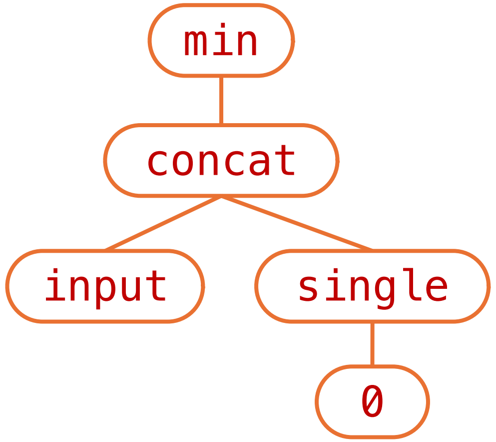
`[‘min’, ‘(’, ‘input’, ‘++’, ‘[’, ‘0’, ‘]’]`

Parser

Concrete Syntax Tree



Abstract Syntax Tree




```
I have a concrete grammar in CFG

 (Program) P <- L | N
   (List) L <- ‘input’
          | ‘[‘ ‘]’
          | ‘[‘ N ‘]’
          | L ‘++’ L
  (Number) N <- ‘len’ ‘(‘ L ‘)’
          | ‘min’ ‘(‘ L ‘)’
          | N ‘+’ N
          | ‘0’ | ‘1’ | ‘2’ | ...

Corresponding to it there is an abstract grammar

(Program) P ::= L | N
   (List) L ::= input
          | empty
          | single(N)
          | concat(L, L)
  (Number) N ::= len(L)
          | min(L)
          | add(N, N)
          | 0 | 1 | 2 | ...

Can you help me write a treesitter parser in javascript?
```

```javascript
module.exports = grammar({
  name: 'listnum',
  rules: {
    program: $ => choice($.list, $.number),

    list: $ => choice($.input, $.empty_list, $.single, $.concat),
    input: _ => 'input',
    empty_list: _ => seq('[', ']'),
    single: $ => seq('[', $.number, ']'),
    concat: $ => prec.left(1, seq($.list, '++', $.list)),

    number: $ => choice($.len, $.min, $.add, $.int_lit),

    len: $ => seq('len', '(', $.list, ')'),
    min: $ => seq('min', '(', $.list, ')'),
    add: $ => prec.left(2, seq($.number, '+', $.number)),
    int_lit: $ => alias($.int, 'int'),
    int: _ => /[0-9]+/,
  }
});
```

Lexer and Parser

Concrete Program

`min(input ++ [0])`

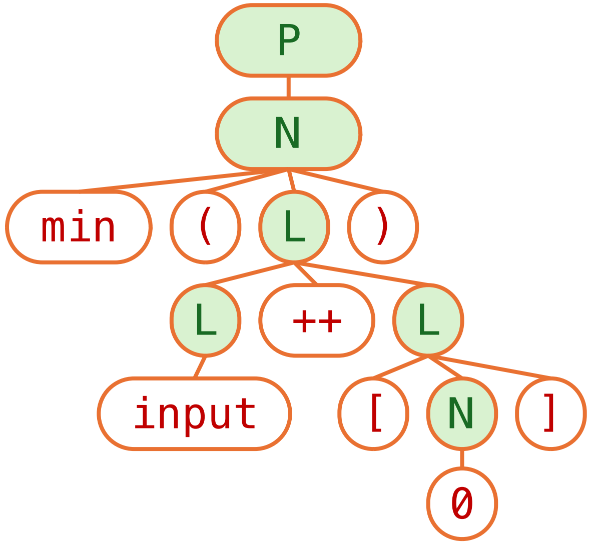
Lexer

Token Sequence

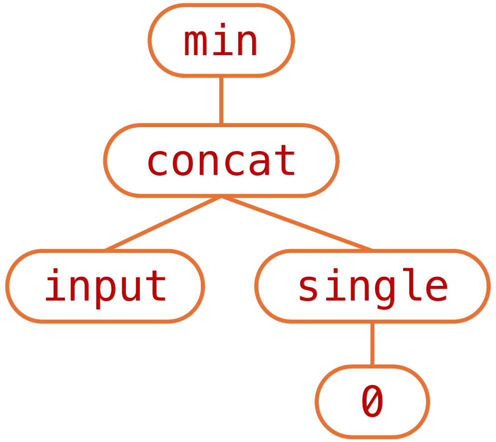
`[‘min’, ‘(’, ‘input’, ‘++’, ‘[’, ‘0’, ‘]’]`

Parser

Concrete Syntax Tree



Abstract Syntax Tree



I have a concrete grammar in CFG

```
(Program) P <- L | N
   (List) L <- ‘input’
             | ‘[‘ ‘]’
             | ‘[‘ N ‘]’
             | L ‘++’ L
 (Number) N <- ‘len’ ‘(‘ L ‘)’
             | ‘min’ ‘(‘ L ‘)’
             | N ‘+’ N
             | ‘0’ | ‘1’ | ‘2’ | ...
```

Corresponding to it there is an abstract grammar

```
(Program) P ::= L | N
   (List) L ::= input
              | empty
              | single(N)
              | concat(L, L)
 (Number) N ::= len(L)
              | min(L)
              | add(N, N)
```

```
module.exports = grammar({
  name: 'listnum',
  rules: {
    program: $ => choice($.list, $.number)

    list: $ => choice($.input, $.empty_lis
    input: _ => 'input',
    empty_list: _ => seq('[', ']'),
    single: $ => seq('[', $.number, ']'),
    concat: $ => prec.left(1, seq($.list,

    number: $ => choice($.len, $.min, $.ad
```

```
> echo "min(input ++ [0])" | npx tree-sitter parse

(program [0, 0] - [1, 0]
 (number [0, 0] - [0, 17]
  (min [0, 0] - [0, 17]
   (list [0, 4] - [0, 16]
    (concat [0, 4] - [0, 16]
     (list [0, 4] - [0, 9]
      (input [0, 4] - [0, 9]))
     (list [0, 13] - [0, 16]
      (single [0, 13] - [0, 16]
       (number [0, 14] - [0, 15]
        (int_lit [0, 14] - [0, 15])))))))))
```

Lexer and Parser in the Wild

JSON (`json.l`)

```
%{
#include "y.tab.h"
%}


%%
"true"      return TRUE;
"false"     return FALSE;
"null"      return NULLVAL;
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
\"([^"]*)\" { yylval = strdup(yytext); return STRING; }
[ \t\n]+    ; /* skip */
"{"         return '{';
"}"         return '}';
"["         return '[';
"]"         return ']';
":"         return ':';
","         return ',';
%%
```

JSON (`json.y`)

```
%token TRUE FALSE NULLVAL NUMBER STRING

%%
value: STRING
     | NUMBER
     | object
     | array
     | TRUE
     | FALSE
     | NULLVAL
     ;

object: '{' '}'
      | '{' members '}'
      ;

members: pair
       | members ',' pair
       ;

pair: STRING ':' value ;

array: '[' ']'
     | '[' elements ']'
     ;

elements: value
        | elements ',' value
        ;
```

Python (`python.lalrpop`)

main · Parser / parser / src / python.lalrpop

Code | Blame | 1796 lines (1644 loc) · 60.4 KB

```
                ast::Stmt::Assign(
                    ast::StmtAssign { targets, value, type_comment: None, range: (location..end_location).into() }
                )
            }
        },
        <location:@L> <target:TestOrStarExprList> <op:AugAssign> <rhs:TestListOrYieldExpr> <end_location:@R> => {
            ast::Stmt::AugAssign(
                ast::StmtAugAssign {
                    target: Box::new(set_context(target, ast::ExprContext::Store)),
                    op,
                    value: Box::new(rhs),
                    range: (location..end_location).into()
                },
            )
        },
        <location:@L> <target:Test<"all">> ":" <annotation:Test<"all">> <rhs:AssignSuffix?> <end_location:@R> => {
            let simple = target.is_name_expr();
            ast::Stmt::AnnAssign(
                ast::StmtAnnAssign {
                    target: Box::new(set_context(target, ast::ExprContext::Store)),
                    annotation: Box::new(annotation),
                    value: rhs.map(Box::new),
                    simple,
                    range: (location..end_location).into()
                },
            )
        },
    };
```

High Level Picture

- We studied foundations of programming languages and synthesis
  - Syntax (regular tree grammar), Semantics (denotational, operational)
  - Enumeration (top-down, bottom-up) that searches within grammar
- We studied basic usage of language models
  - Purely neural: next token prediction → sequential decoding
  - Prompting and iterative refinement: elicit better programs from LLM
- Question:
  - Can we inject more **structure** during **neural** generation process

Possible Next Token by Parser

Partial Program

`min(input ++ [0])`

Token Sequence

`['min', '(', 'input', '++', '[', '0', ']']`

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```

Possible Next Token by Parser

Partial Program

`min(input ++ [0])`

Token Sequence

`['min', '(', 'input', '++', '[', '0', ']']`

Step 1

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```

Possible Next Token by Parser

Partial Program

`min(input ++ [0])`

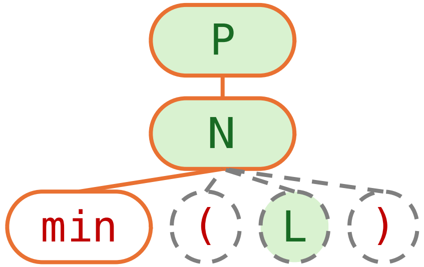
Token Sequence

`['min', '(', 'input', '++', '[', '0', ']']`

Step 1: Goal is P

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```

`'min'` only appears in the rule `N <- 'min' '(' L ')'`



Possible Next Token by Parser

Partial Program

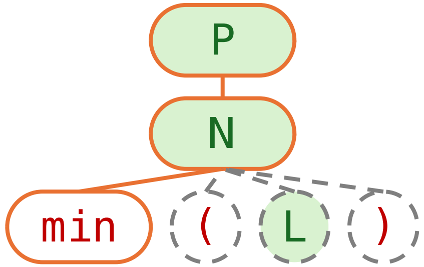
`min(input ++ [0])`

Token Sequence

`['min', '(', 'input', '++', '[', '0', ']']`

Step 1: Goal is P

'min' only appears in the rule `N <- 'min' '(' L ')'`



The next token could only be '('

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```

Possible Next Token by Parser

Partial Program

`min(input ++ [0])`

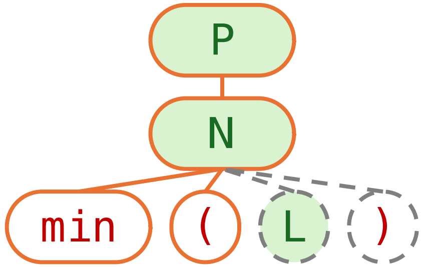
Token Sequence

`[‘min’, ‘(’, ‘input’, ‘++’, ‘[’, ‘0’, ‘]’]`

Step 2: Goal is P

```
(Program) P <- L | N
   (List) L <- ‘input’
             | ‘[‘ ‘]’
             | ‘[‘ N ‘]’
             | L ‘++’ L
 (Number) N <- ‘len’ ‘(‘ L ‘)’
             | ‘min’ ‘(‘ L ‘)’
             | N ‘+’ N
             | ‘0’ | ‘1’ | ‘2’ | …
```

`‘min’ ‘(‘` only appears in the rule `N <- ‘min’ ‘(’ L ‘)’`



Possible Next Token by Parser

Partial Program

`min(input ++ [0])`

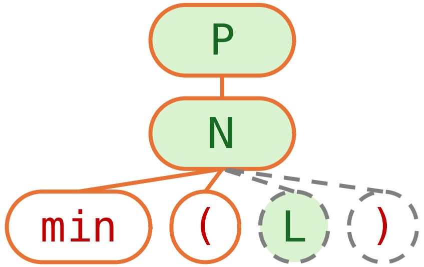
Token Sequence

`['min', '(', 'input', '++', '[', '0', ']']`

Step 2: Goal is P

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```

`'min' '('` only appears in the rule `N <- 'min' '(' L ')'`



The next token could only be the first token expanded from L: `'input', '['`

Possible Next Token by Parser

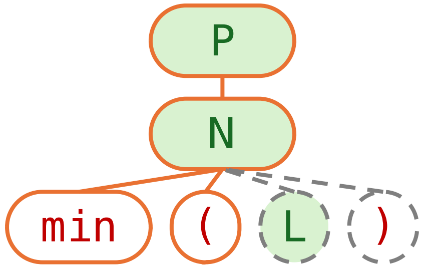
Partial Program

`min(input ++ [0])`

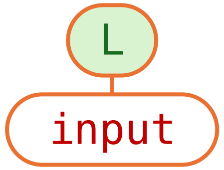
Token Sequence

`['min', '(', 'input', '++', '[', '0', ']']`

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```



Step 3: Goal is L



We can expand L into `input`, matching the third token

Possible Next Token by Parser

Partial Program

`min(input ++ [0])`

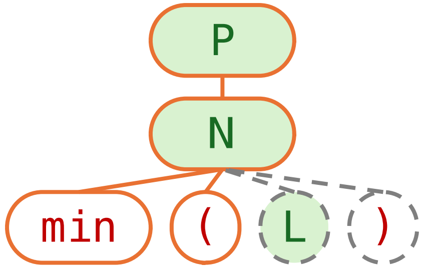
Token Sequence

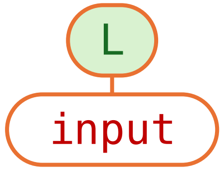
`['min', '(', 'input', '++', '[', '0', ']']`

↑

Step 3: Goal is L

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```





We can expand L into `input`, matching the third token

➔ Finishing L, next token is ') '

Possible Next Token by Parser

Partial Program

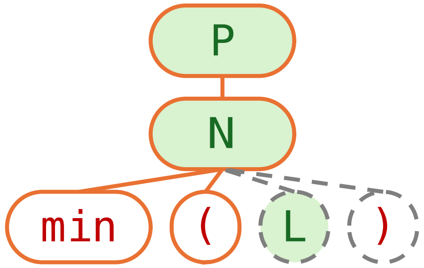
`min(input ++ [0])`

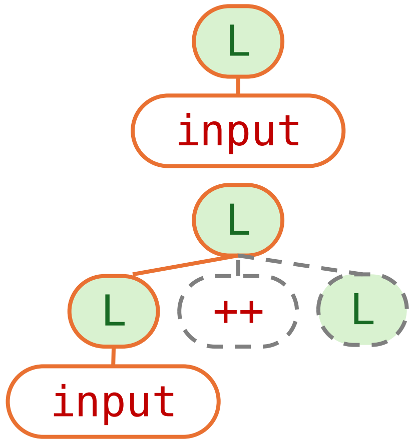
Token Sequence

`['min', '(', 'input', '++', '[', '0', ']']`

Step 3: Goal is L

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```





We can expand L into `input`, matching the third token

➔ Finishing L, next token is ') '

We can expand L into L + L, where the first L can be expanded into `input`, matching the third token

Possible Next Token by Parser

Partial Program

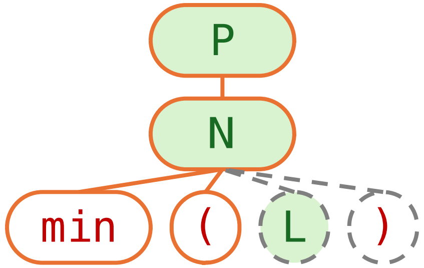
`min(input ++ [0])`

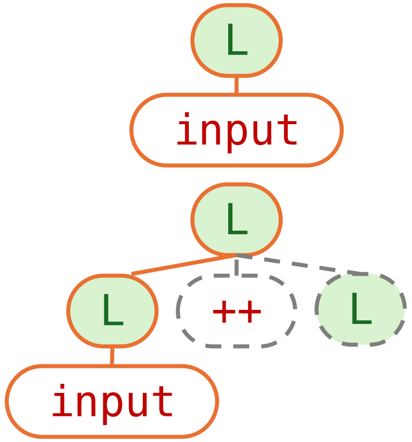
Token Sequence

`['min', '(', 'input', '++', '[', '0', ']']`

Step 3: Goal is L

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```





We can expand L into `input`, matching the third token

➔ Finishing L, next token is ')'

We can expand L into L + L, where the first L can be expanded into `input`, matching the third token

➔ Finishing the left L, next token is '++'

Possible Next Token by Parser

Partial Program

```
min(input ++ [0])
```

Token Sequence

```
['min', '(', 'input', '++', '[', '0', ']']
```

```
{'('}
{'input', '['}
{')', '++'}
```

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```

Alternatives of Next Token Filtering

- Problem:
  - Given a prefix, find the set of tokens that could be the next
- Alternative problem:
  - Given a prefix and a predicted next token, check if the next token is a plausible completion
    - Prefix viability problem / Prefix membership problem
    - Given a grammar G and a prefix w, does there exist x s.t. $wx \in L(G)$

Algorithms for parsing

- Basic parsing
  - Earley parsing: maintains a set of items that implicitly answer the “prefix membership” question at each position
  - CKY (Cocke–Younger–Kasami) parsing: bottom-up finding substrings that can be turned into abstract syntax trees
  - Automaton: convert the grammar into an Automaton, keep an active state, reject the string if the next token leads to an error state

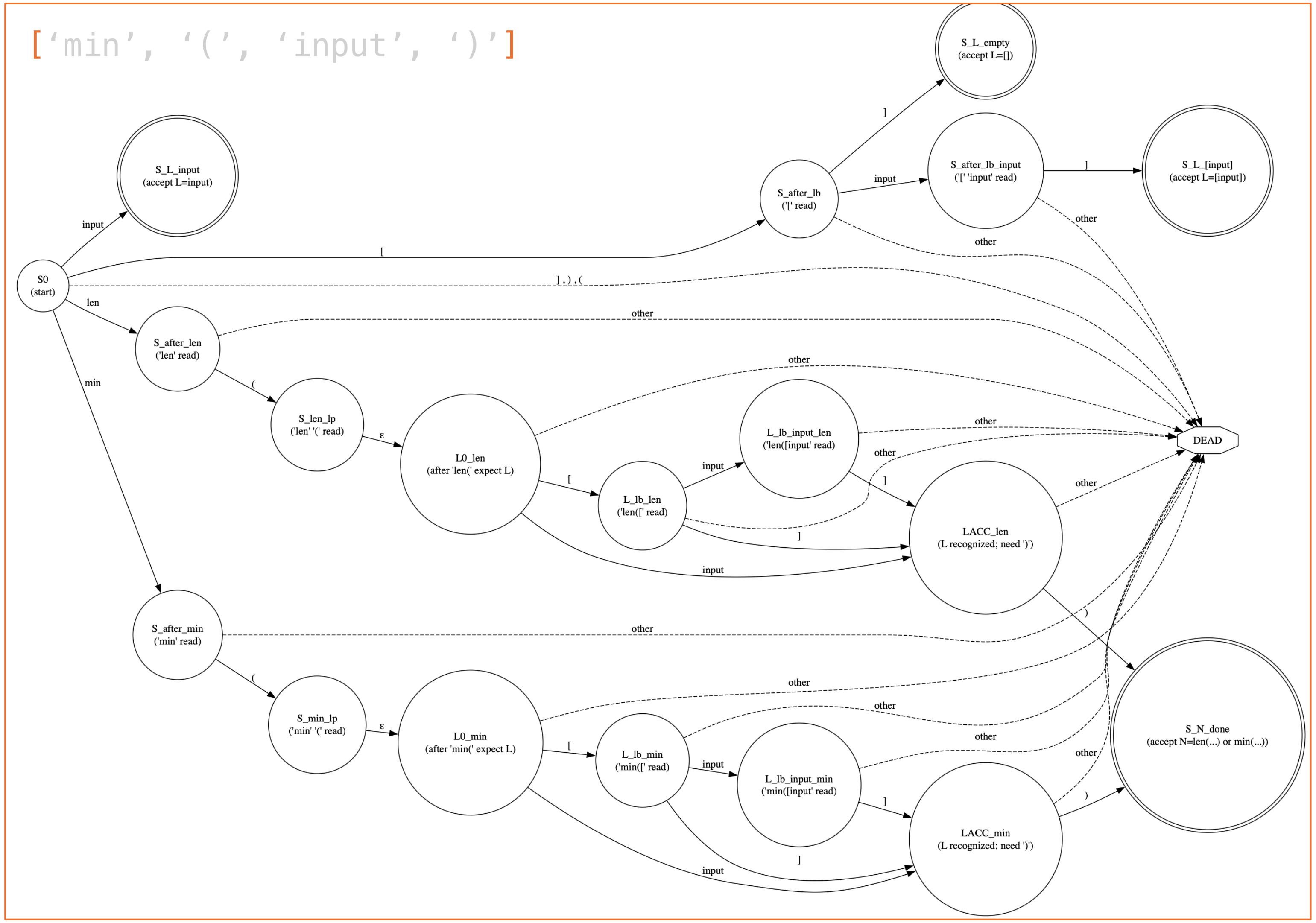
Constrained Decoding with Automaton

```
state = start_of_automaton(grammar)
current_sequence = input_tokens
output_sequence = []
for step in 1 .. max_length:
  logits = predict_next_token(current_sequence) / temperature
  for (token_id, logit) in enumerate(logits):
    if not allowed_by(state, token_id):
      logits[token_id] = -inf
  probs = softmax(logits)
  next_token = nucleus_sample_from(probs, P)
  current_sequence += [next_token]
  output_sequence += [next_token]
  if next_token == <EOS>: break
  state = advance(state, next_token)
return output_sequence
```

Constrained Decoding with Automaton

```
 (Program) P <- L | N
    (List) L <- 'input'
              | '[' ']'
              | '[' N ']'
  (Number) N <- 'len' '(' L ')'
              | 'min' '(' L ')'
```

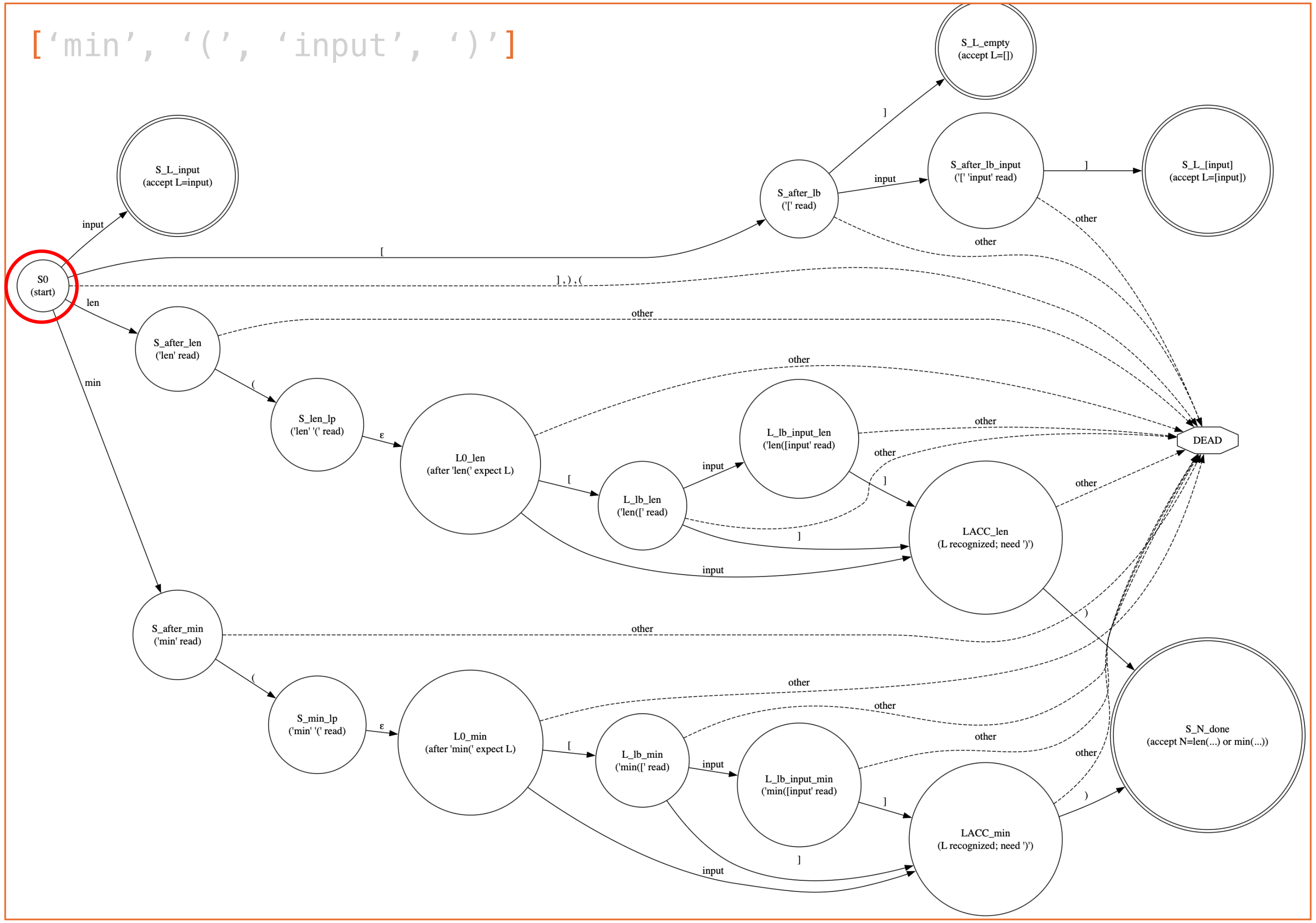
```
state = start_of_automaton(grammar)
current_sequence = input_tokens
output_sequence = []
for step in 1 .. max_length:
  logits = predict_next_token(current_sequence)…
  for (token_id, logit) in enumerate(logits):
    if not allowed_by(state, token_id):
      logits[token_id] = -inf
  probs = softmax(logits)
  next_token = nucleus_sample_from(probs, P)
  current_sequence += [next_token]
  output_sequence += [next_token]
  if next_token == <EOS>: break
  state = advance(state, next_token)
return output_sequence
```



Constrained Decoding with Automaton

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
```

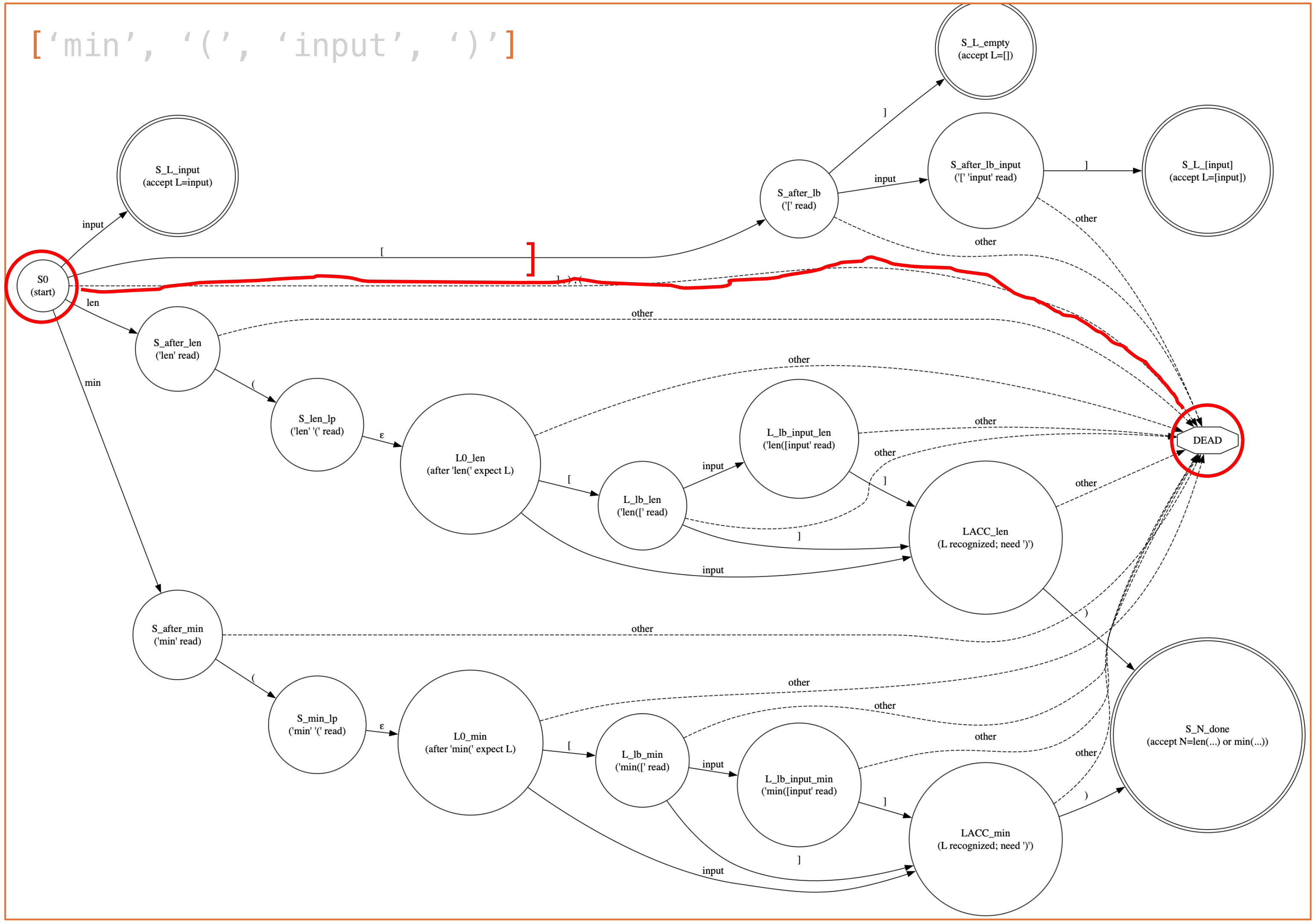
```
state = start_of_automaton(grammar)
current_sequence = input_tokens
output_sequence = []
for step in 1 .. max_length:
  logits = predict_next_token(current_sequence)…
  for (token_id, logit) in enumerate(logits):
    if not allowed_by(state, token_id):
      logits[token_id] = -inf
  probs = softmax(logits)
  next_token = nucleus_sample_from(probs, P)
  current_sequence += [next_token]
  output_sequence += [next_token]
  if next_token == <EOS>: break
  state = advance(state, next_token)
return output_sequence
```



Constrained Decoding with Automaton

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
```

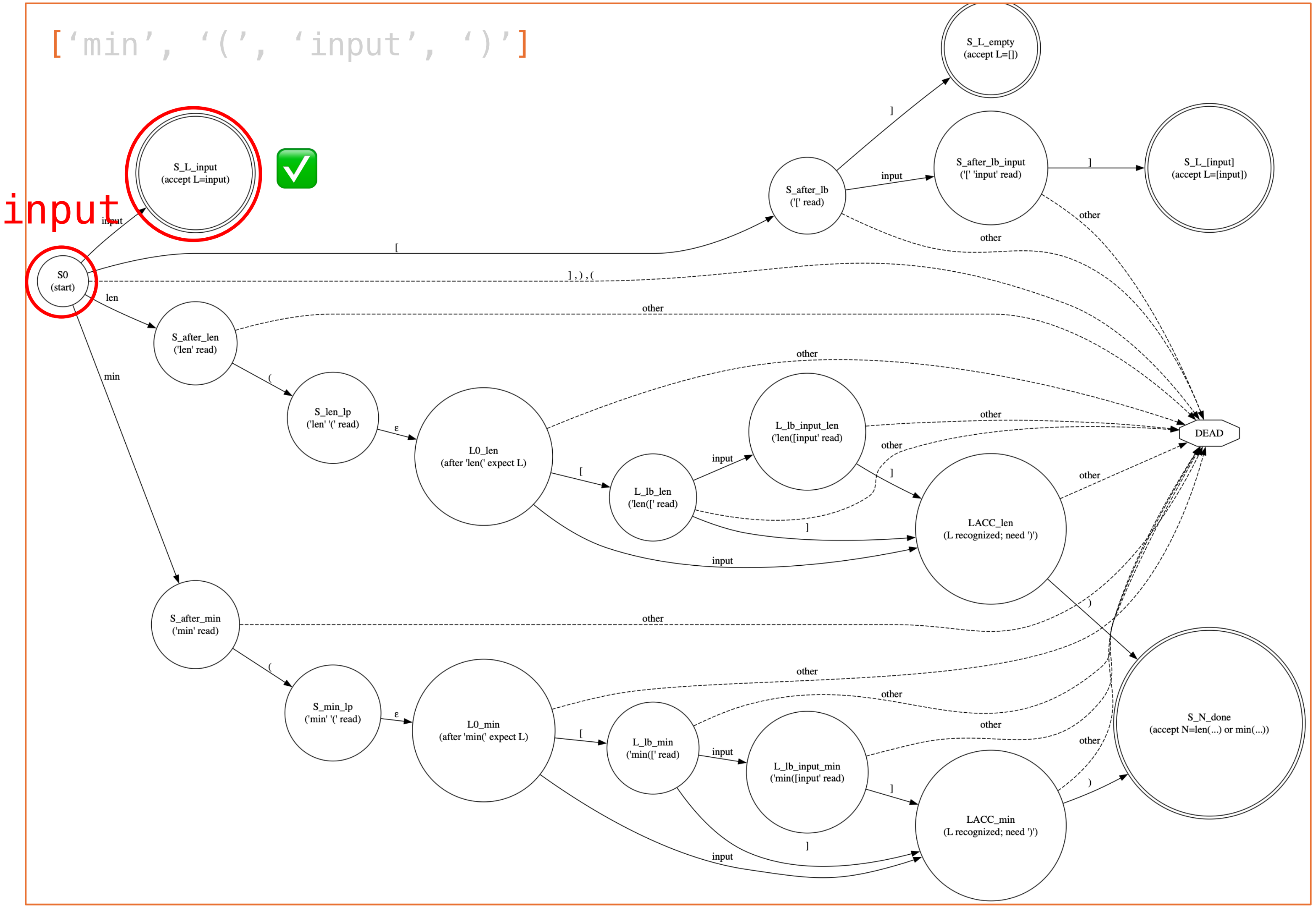
```
state = start_of_automaton(grammar)
current_sequence = input_tokens
output_sequence = []
for step in 1 .. max_length:
  logits = predict_next_token(current_sequence)…
  for (token_id, logit) in enumerate(logits):
    if not allowed_by(state, token_id):
      logits[token_id] = -inf
  probs = softmax(logits)
  next_token = nucleus_sample_from(probs, P)
  current_sequence += [next_token]
  output_sequence += [next_token]
  if next_token == <EOS>: break
  state = advance(state, next_token)
return output_sequence
```



Constrained Decoding with Automaton

```
 (Program) P <- L | N
    (List) L <- 'input'
              | '[' ']'
              | '[' N ']'
  (Number) N <- 'len' '(' L ')'
              | 'min' '(' L ')'
```

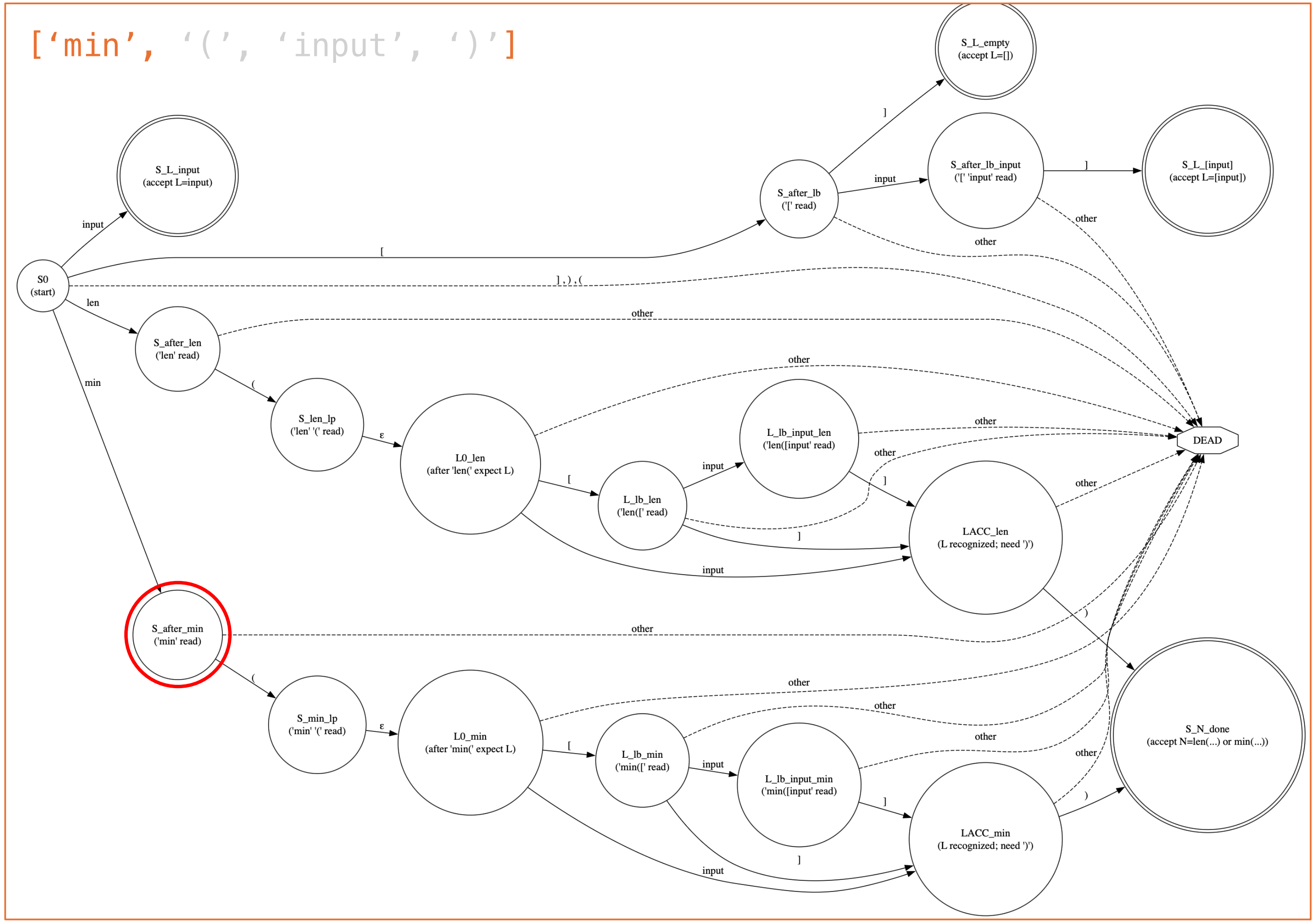
```
state = start_of_automaton(grammar)
current_sequence = input_tokens
output_sequence = []
for step in 1 .. max_length:
  logits = predict_next_token(current_sequence)…
  for (token_id, logit) in enumerate(logits):
    if not allowed_by(state, token_id):
      logits[token_id] = -inf
  probs = softmax(logits)
  next_token = nucleus_sample_from(probs, P)
  current_sequence += [next_token]
  output_sequence += [next_token]
  if next_token == <EOS>: break
  state = advance(state, next_token)
return output_sequence
```



Constrained Decoding with Automaton

```
 (Program) P <- L | N
    (List) L <- 'input'
              | '[' ']'
              | '[' N ']'
  (Number) N <- 'len' '(' L ')'
              | 'min' '(' L ')'
```

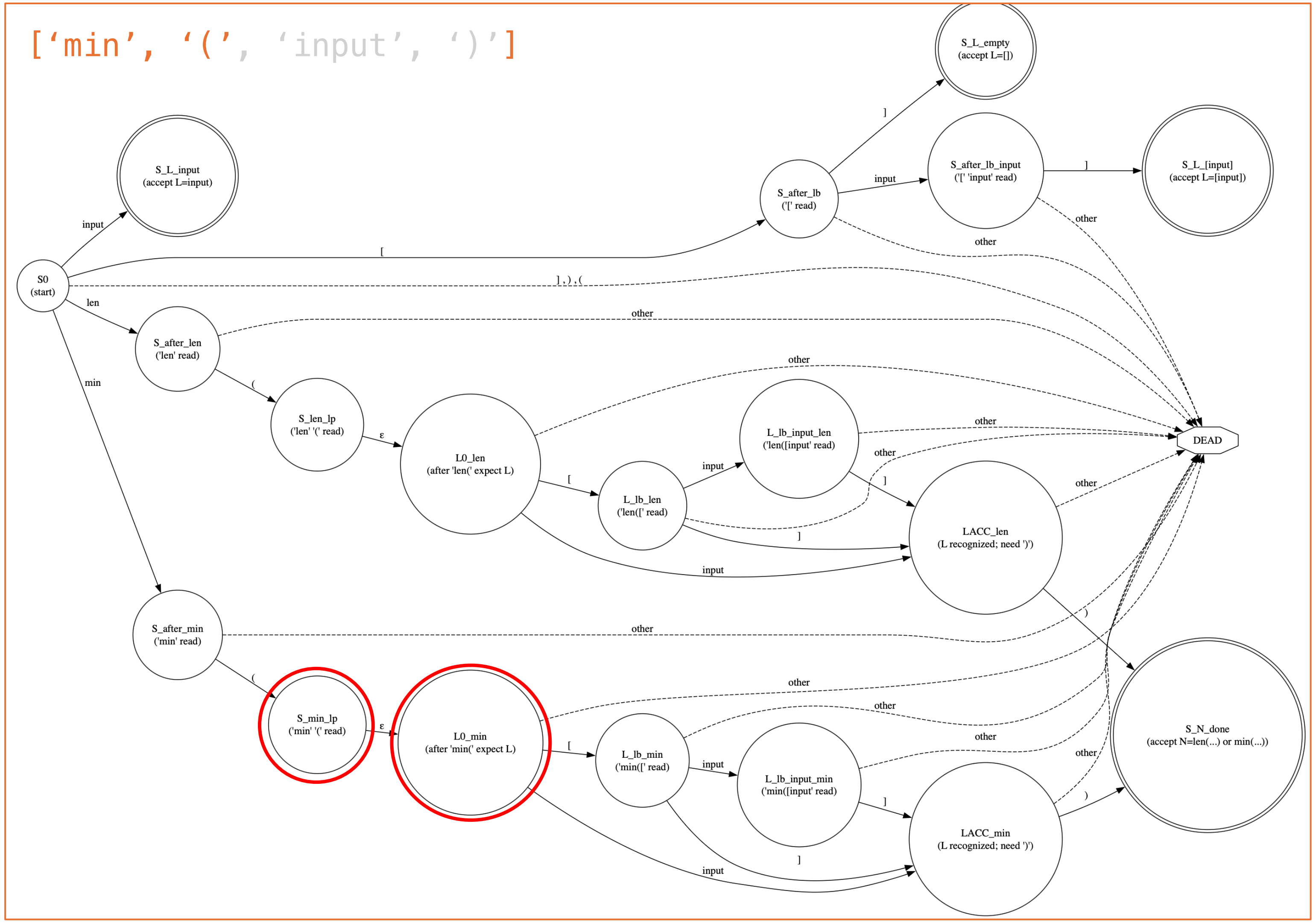
```
state = start_of_automaton(grammar)
current_sequence = input_tokens
output_sequence = []
for step in 1 .. max_length:
  logits = predict_next_token(current_sequence)…
  for (token_id, logit) in enumerate(logits):
    if not allowed_by(state, token_id):
      logits[token_id] = -inf
  probs = softmax(logits)
  next_token = nucleus_sample_from(probs, P)
  current_sequence += [next_token]
  output_sequence += [next_token]
  if next_token == <EOS>: break
  state = advance(state, next_token)
return output_sequence
```



Constrained Decoding with Automaton

```
(Program) P <- L | N
   (List) L <- ‘input’
             | ‘[‘ ‘]’
             | ‘[‘ N ‘]’
 (Number) N <- ‘len’ ‘(‘ L ‘)’
             | ‘min’ ‘(‘ L ‘)’
```

```
state = start_of_automaton(grammar)
current_sequence = input_tokens
output_sequence = []
for step in 1 .. max_length:
  logits = predict_next_token(current_sequence)…
  for (token_id, logit) in enumerate(logits):
    if not allowed_by(state, token_id):
      logits[token_id] = -inf
  probs = softmax(logits)
  next_token = nucleus_sample_from(probs, P)
  current_sequence += [next_token]
  output_sequence += [next_token]
  if next_token == <EOS>: break
  state = advance(state, next_token)
return output_sequence
```

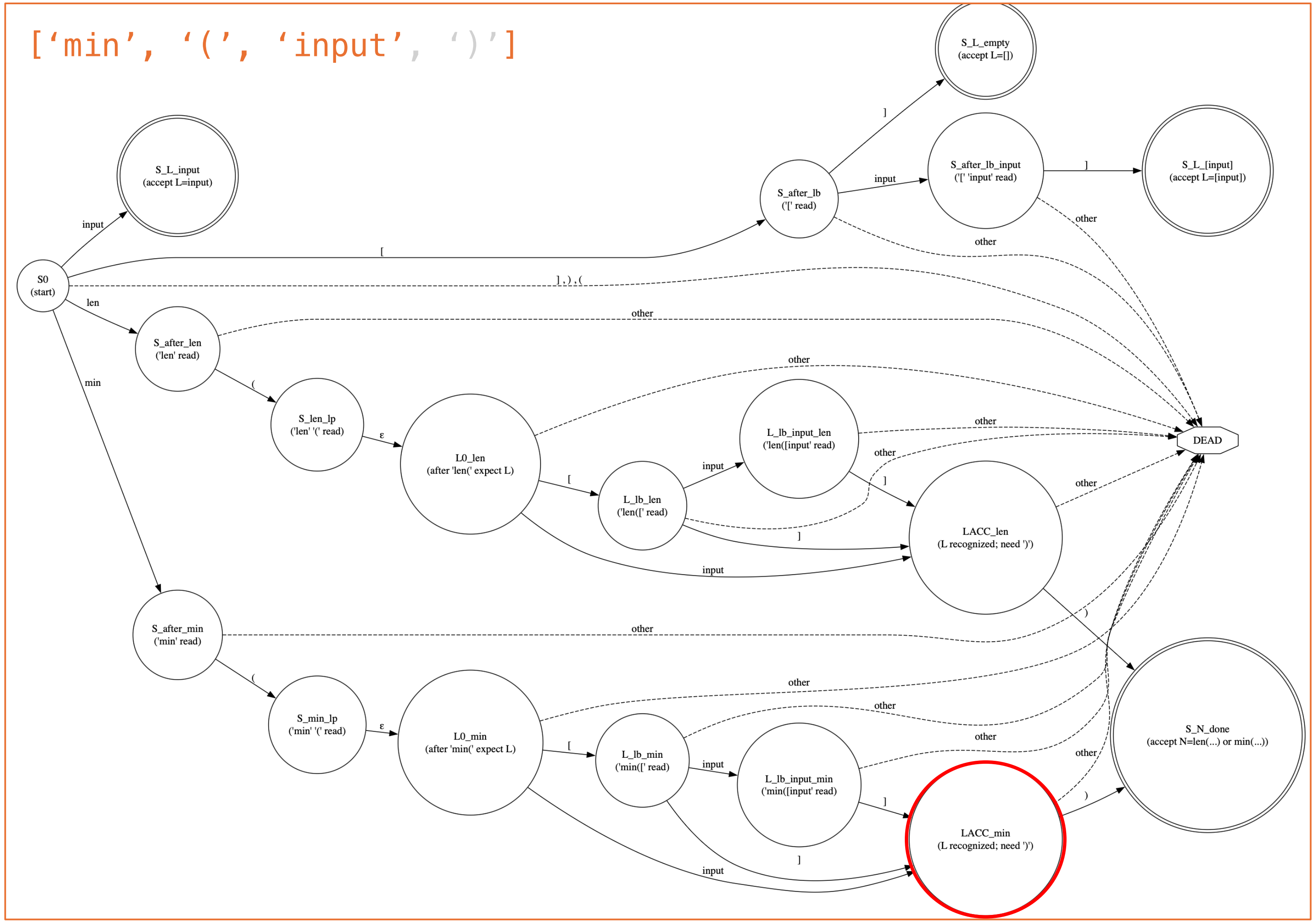


Constrained Decoding with Automaton

```
 (Program) P <- L | N
    (List) L <- 'input'
              | '[' ']'
              | '[' N ']'
  (Number) N <- 'len' '(' L ')'
              | 'min' '(' L ')'
```

```
state = start_of_automaton(grammar)
current_sequence = input_tokens
output_sequence = []
for step in 1 .. max_length:
  logits = predict_next_token(current_sequence)…
  for (token_id, logit) in enumerate(logits):
    if not allowed_by(state, token_id):
      logits[token_id] = -inf
  probs = softmax(logits)
  next_token = nucleus_sample_from(probs, P)
  current_sequence += [next_token]
  output_sequence += [next_token]
  if next_token == <EOS>: break
  state = advance(state, next_token)
return output_sequence
```

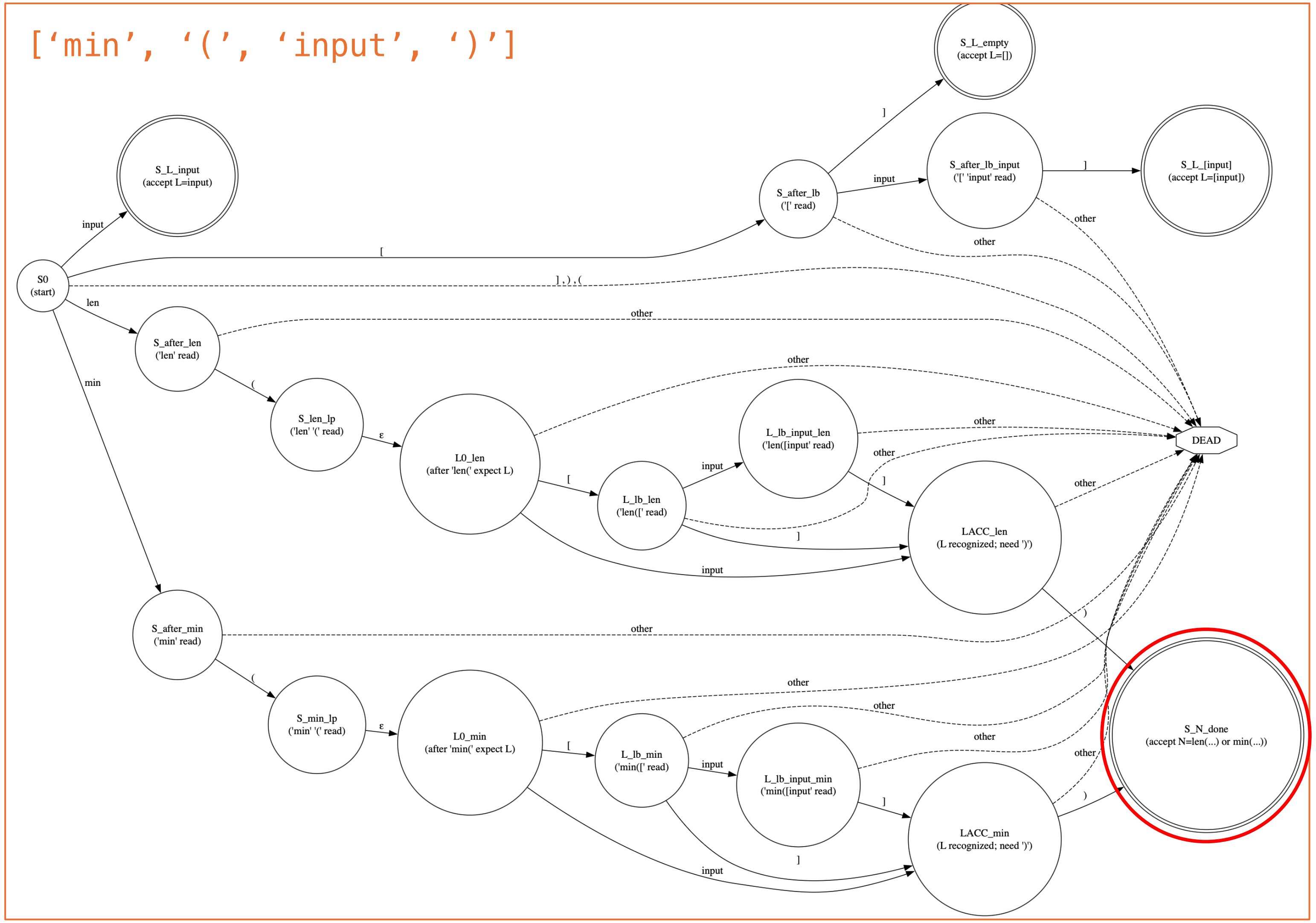
['min', '(', 'input', ')']



Constrained Decoding with Automaton

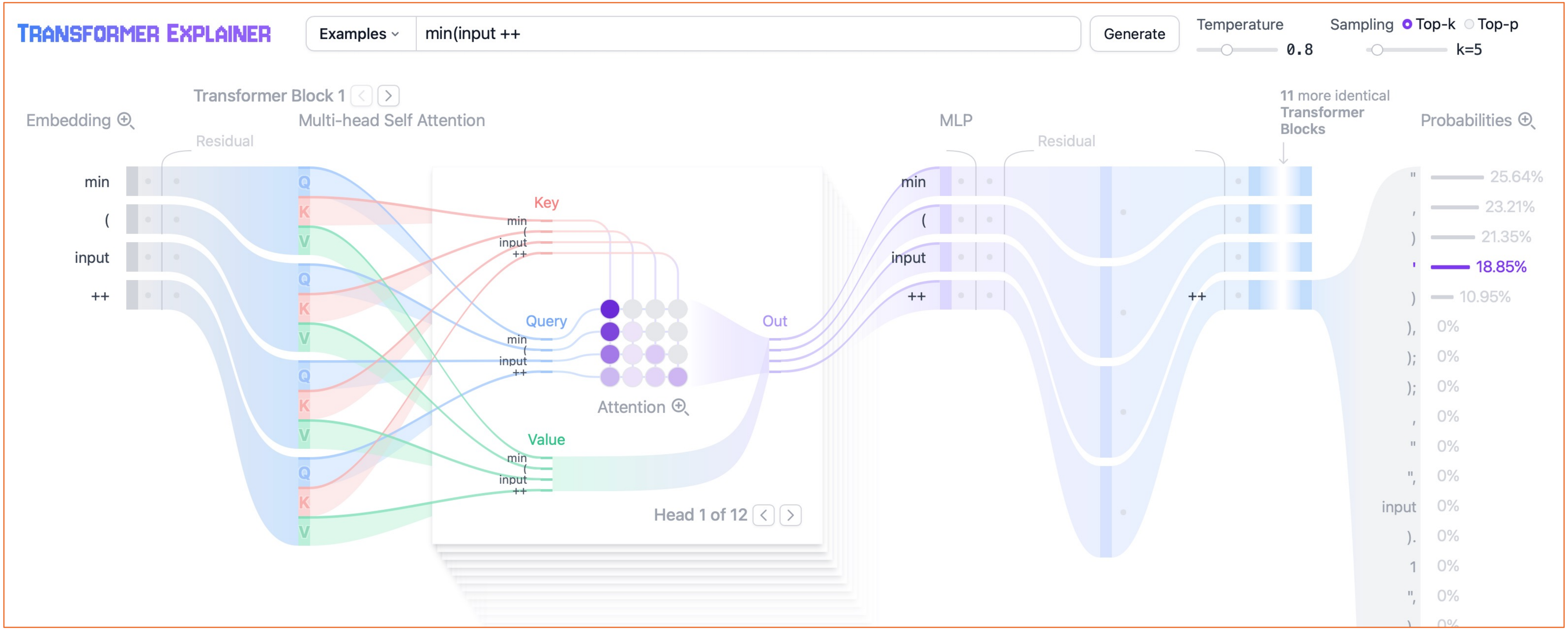
```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
```

```
state = start_of_automaton(grammar)
current_sequence = input_tokens
output_sequence = []
for step in 1 .. max_length:
  logits = predict_next_token(current_sequence)…
  for (token_id, logit) in enumerate(logits):
    if not allowed_by(state, token_id):
      logits[token_id] = -inf
  probs = softmax(logits)
  next_token = nucleus_sample_from(probs, P)
  current_sequence += [next_token]
  output_sequence += [next_token]
  if next_token == <EOS>: break
  state = advance(state, next_token)
return output_sequence
```

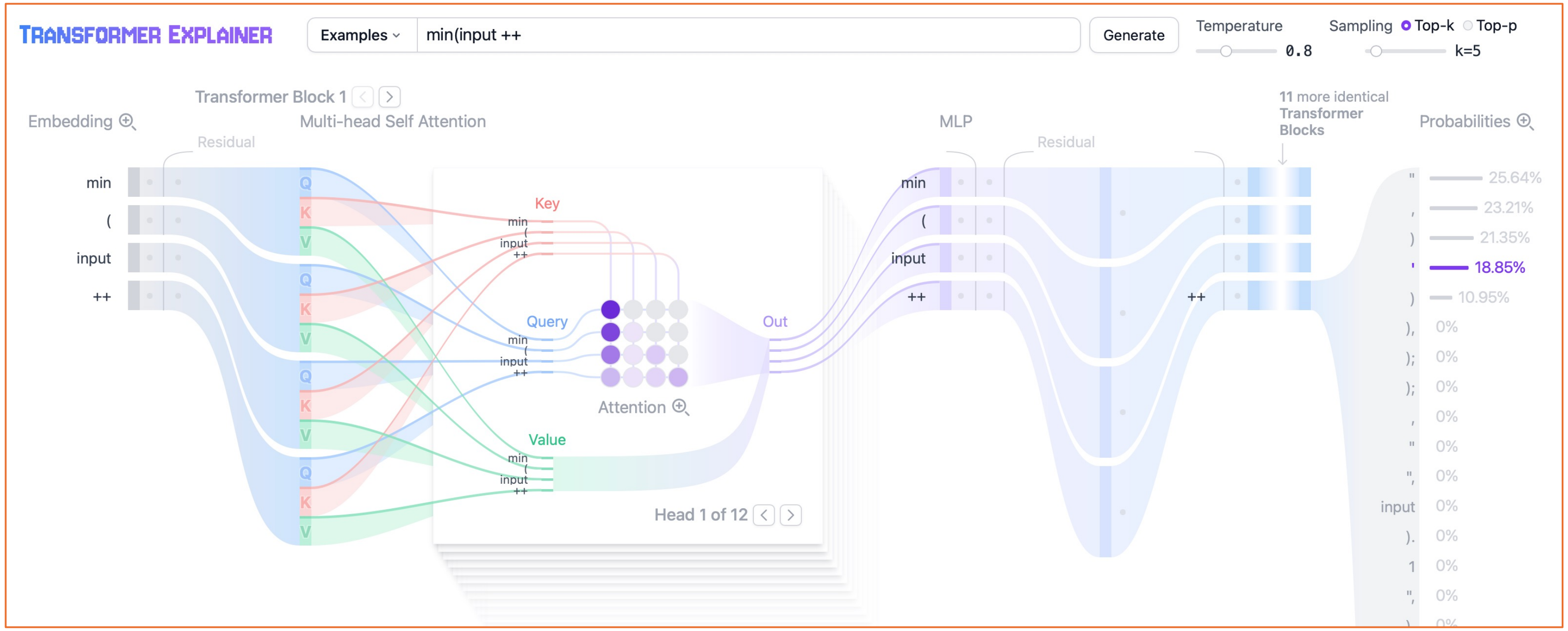


Algorithms for parsing

- Basic parsing
  - Earley parsing: maintains a set of items that implicitly answer the “prefix membership” question at each position
  - CKY (Cocke–Younger–Kasami) parsing: bottom-up finding substrings that can be turned into abstract syntax trees
  - Automaton: convert the grammar into an Automaton, keep an active state, reject the string if the next token leads to an error state
- Compiler Parsing
  - Batched parsing: takes the entire program and parses
- Interactive / Incremental parsing
  - Error-tolerant / resilient parsers; skip until synchronizing symbols (like ;,})
  - Keeps tree of interactions; maintain the global AST structure and only create bad subtrees when there is error


TRANSFORMER EXPLAINER
Examples
min(input ++
Generate
Temperature
0.8
Sampling
Top-k
Top-p
k=5
Transformer Block 1
Embedding
Multi-head Self Attention
Residual
MLP
Residual
11 more identical Transformer Blocks
Probabilities
min
(
input
++
Key
Query
Value
Attention
Out
Head 1 of 12
25.64%
23.21%
21.35%
18.85%
10.95%
0%

Prefix: `min(input ++`



LLM Predicted Next Token		Prefix Viability Mask
input	0.01	1
)	0.21	0
‘	0.19	0
“	0.26	0
;	0.01	0
[	0.02	1

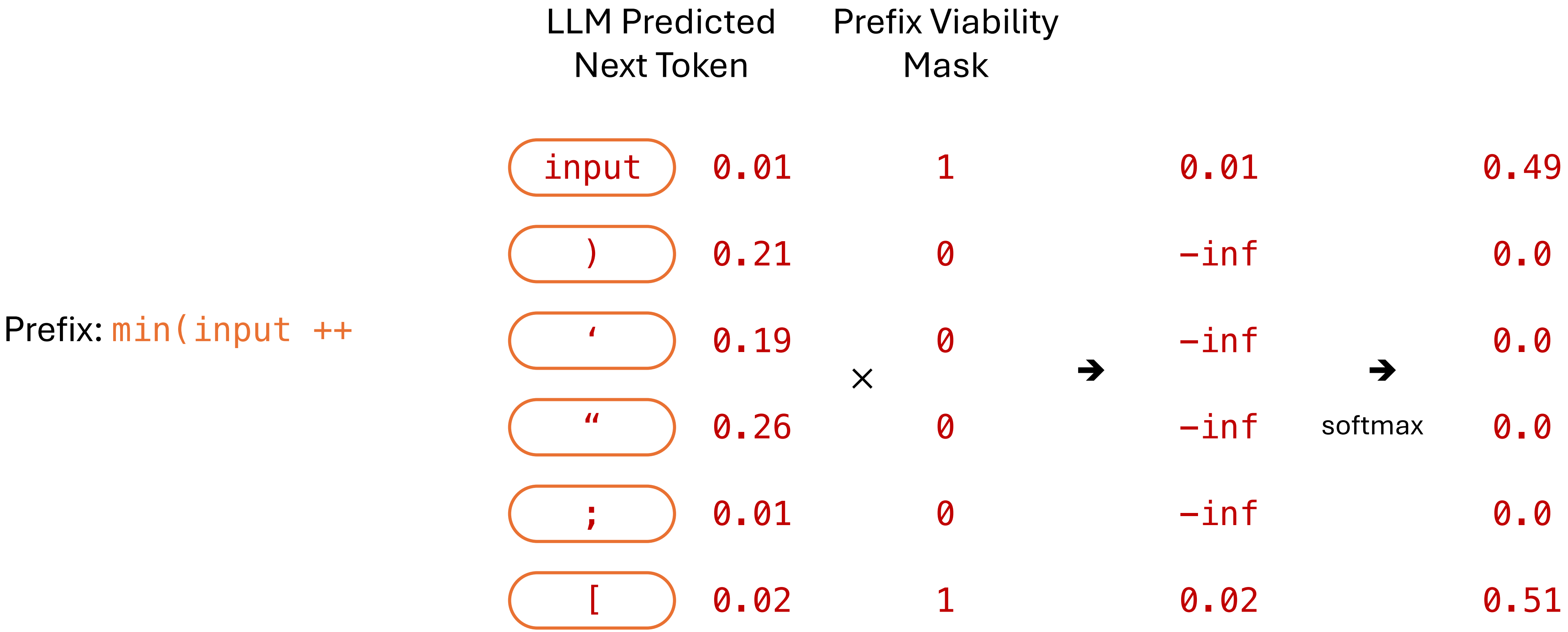
Prefix: `min(input ++`

LLM Predicted Next Token			Prefix Viability Mask
`input`	0.01		1
`)`	0.21		0
`‘`	0.19	$\times$	0
`“`	0.26		0
`;`	0.01		0
`[`	0.02		1

$=$

Prefix: `min(input ++`

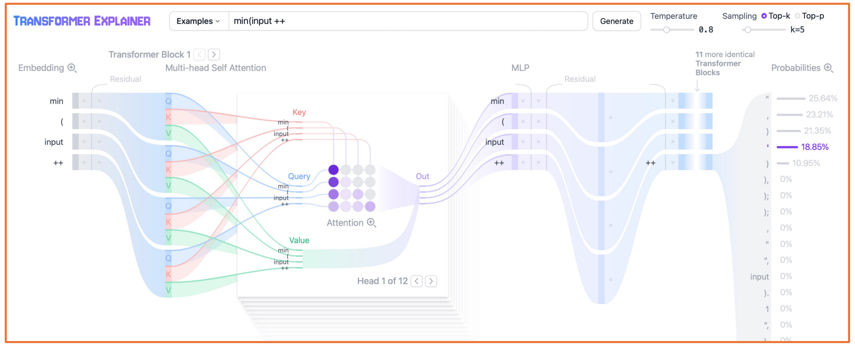
LLM Predicted Next Token			Prefix Viability Mask		
input	0.01		1		0.01
)	0.21		0		0
‘	0.19		0		0
“	0.26	×	0	=	0
;	0.01		0		0
[	0.02		1		0.02



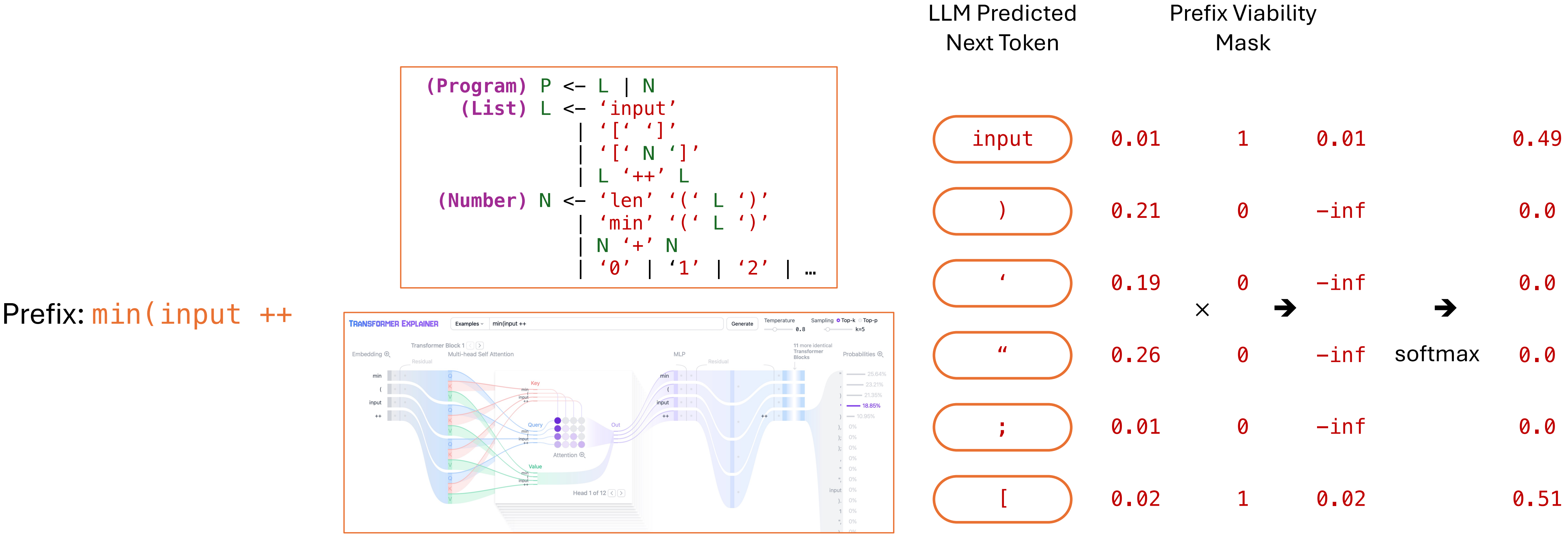
LLM Predicted Next Token
Prefix Viability Mask
Prefix: min(input ++
input
0.01
1
0.01
0.49
)
0.21
0
-inf
0.0
‘
0.19
0
-inf
0.0
“
0.26
0
-inf
0.0
;
0.01
0
-inf
0.0
[
0.02
1
0.02
0.51
×
softmax

Prefix: min(input ++

```
(Program) P <- L | N
   (List) L <- 'input'
             | '[' ']'
             | '[' N ']'
             | L '++' L
 (Number) N <- 'len' '(' L ')'
             | 'min' '(' L ')'
             | N '+' N
             | '0' | '1' | '2' | …
```



LLM Predicted Next Token			Prefix Viability Mask					softmax
input	0.01		1		0.01			0.49
)	0.21		0		-inf			0.0
‘	0.19	×	0	➔	-inf	➔	softmax	0.0
“	0.26		0		-inf			0.0
;	0.01		0		-inf			0.0
[	0.02		1		0.02			0.51



Prefix: min(input ++
(Program) P <- L | N
(List) L <- 'input'
| '[' ']'
| '[' N ']'
| L '++' L
(Number) N <- 'len' '(' L ')'
| 'min' '(' L ')'
| N '+' N
| '0' | '1' | '2' | …
TRANSFORMER EXPLAINER
Examples
min(input ++
Generate
Temperature
0.8
Sampling
Top-k
Top-p
k=5
Embedding
Transformer Block 1
Multi-head Self Attention
Key
Query
Value
Attention
Out
Head 1 of 12
MLP
Residual
11 more identical Transformer Blocks
Probabilities
LLM Predicted Next Token
Prefix Viability Mask
input
)
'
"
;
[
0.01
0.21
0.19
0.26
0.01
0.02
×
1
0
0
0
0
1
→
0.01
-inf
-inf
-inf
-inf
0.02
→
softmax
0.49
0.0
0.0
0.0
0.0
0.51

Constrained Decoding in the Wild

A Syntactic Neural Model for General-Purpose Code Generation

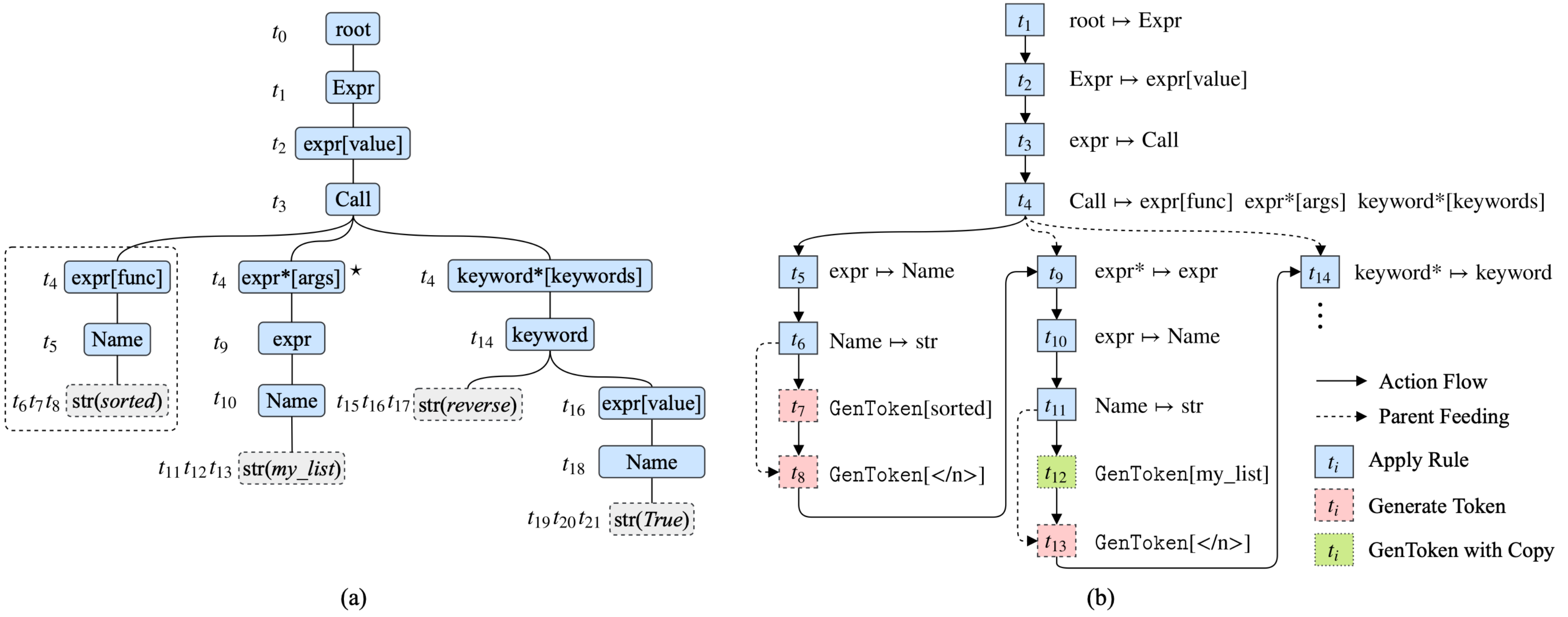
Pengcheng Yin
Language Technologies Institute
Carnegie Mellon University
pcyin@cs.cmu.edu

Graham Neubig
Language Technologies Institute
Carnegie Mellon University
gneubig@cs.cmu.edu

A Syntactic Neural Model for General-Purpose Code Generation

Pengcheng Yin
Language Technologies Institute
Carnegie
pcyi

Graham Neubig
Language Technologies Institute



Input: `sort my_list in descending order` **Code:** `sorted(my_list, reverse=True)`

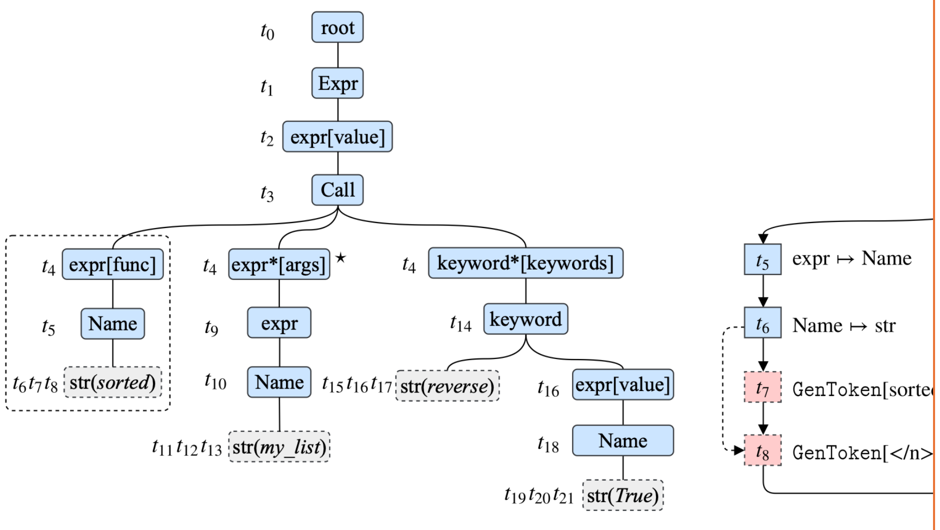
Figure 1: (a) the Abstract Syntax Tree (AST) for the given example code. Dashed nodes denote terminals. Nodes are labeled with time steps during which they are generated. (b) the action sequence (up to t_{14}) used to generate the AST in (a)

A Syntactic Neural Model for General-Purpose Code Generation, Yin et. al., 2017

A Syntactic Neural Model for General-Purpose

Lan

C



(a)

Input: `sort my_list in descending order` **Code:** so

Figure 1: (a) the Abstract Syntax Tree (AST) for the given example code. Das
with time steps during which they are generated. (b) the action sequence (up to

4.2.2 Calculating Action Probabilities

In this section we explain how action probabilities $p(a_t|x, a_{<t})$ are computed based on $\mathbf{s}_t$.

ApplyRule The probability of applying rule r as the current action a_t is given by a softmax[5]:

$$p(a_t = \textsc{ApplyRule}[r]|x, a_{<t}) = \mathrm{softmax}(\mathbf{W}_R \cdot g(\mathbf{s}_t))^\intercal \cdot \mathbf{e}(r) \quad (4)$$

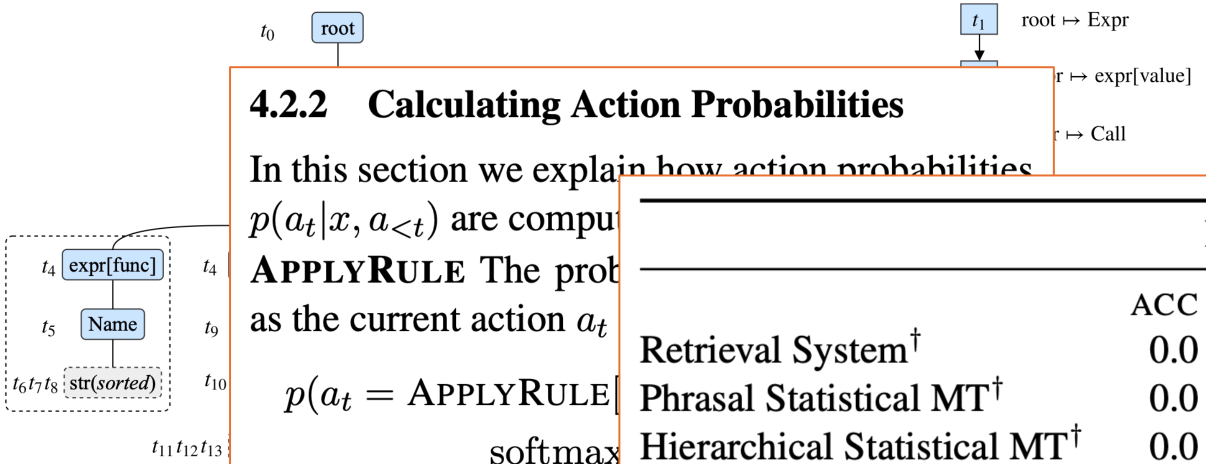
where $g(\cdot)$ is a non-linearity $\tanh(\mathbf{W}\cdot\mathbf{s}_t+\mathbf{b})$, and $\mathbf{e}(r)$ the one-hot vector for rule r.

GenToken As in § 3.2, a token v can be generated from a predefined vocabulary or copied from the input, defined as the marginal probability:

$$p(a_t = \textsc{GenToken}[v]|x, a_{<t}) = p(\mathrm{gen}|x, a_{<t})p(v|\mathrm{gen}, x, a_{<t}) + p(\mathrm{copy}|x, a_{<t})p(v|\mathrm{copy}, x, a_{<t}).$$

A Syntactic Neural Model for General-Purpose Code Generation

Lan
C



Input: `sort`

Figure 1: (a) the A
with time steps dur

4.2.2 Calculating Action Probabilities

In this section we explain how action probabilities
$p(a_t|x, a_{<t})$ are comput
APPLYRULE The prob
as the current action a_t

$$p(a_t = \text{APPLYRULE}[$$
softmax

where $g(\cdot)$ is a non-line
$\mathbf{e}(r)$ the one-hot vector
GENTOKEN As in § 3.
ated from a predefined
the input, defined as the

$$p(a_t = \text{GENTOKEN}[$$
$$p(\text{gen}|x, a_{<t})$$
$$+ p(\text{copy}|x$$

	HS		DJANGO	
	ACC	BLEU	ACC	BLEU
Retrieval System†	0.0	62.5	14.7	18.6
Phrasal Statistical MT†	0.0	34.1	31.5	47.6
Hierarchical Statistical MT†	0.0	43.2	9.5	35.9
NMT	1.5	60.4	45.1	63.4
SEQ2TREE	1.5	53.4	28.9	44.6
SEQ2TREE–UNK	13.6	62.8	39.4	58.2
LPN†	4.5	65.6	62.3	77.6
Our system	16.2	**75.8**	**71.6**	**84.5**
Ablation Study				
– frontier embed.	**16.7**	**75.8**	70.7	83.8
– parent feed.	10.6	75.7	71.5	84.3
– copy terminals	3.0	65.7	32.3	61.7
+ unary closure	–		70.3	83.3
– unary closure	10.1	74.8	–	

A Syntactic Neural Model for General-Purpose Code Generation, Yin et. al., 2017

Constrained Decoding in the Wild

Abstract Syntax Networks for Code Generation and Semantic Parsing

Maxim Rabinovich* **Mitchell Stern*** **Dan Klein**
Computer Science Division
University of California, Berkeley
{rabinovich,mitchell,klein}@cs.berkeley.edu

Abstract Syntax Networks for Code Generation and Semantic Parsing, Rabinovich et. al., 2017

Abstract Syntax Networks for Code Generation and Semantic Pa

Maxim Rabinovich* **Mitchell Stern*** **Dan Klein**
Computer Science Division
University of California, Berkeley
{rabinovich,mitchell,klein}@cs.berkeley.edu


```
name: [
  'D', 'i', 'r', 'e', ' ',
  'W', 'o', 'l', 'f', ' ',
  'A', 'l', 'p', 'h', 'a']
cost: ['2']
type: ['Minion']
rarity: ['Common']
race: ['Beast']
class: ['Neutral']
description: [
  'Adjacent', 'minions', 'have',
  '+', '1', 'Attack', '.']
health: ['2']
attack: ['2']
durability: ['-1']
```

```
class DireWolfAlpha(MinionCard):
  def __init__(self):
    super().__init__(
      "Dire Wolf Alpha", 2, CHARACTER_CLASS.ALL,
      CARD_RARITY.COMMON, minion_type=MINION_TYPE.BEAST)
  def create_minion(self, player):
    return Minion(2, 2, auras=[
      Aura(ChangeAttack(1), MinionSelector(Adjacent()))
    ])
```

Figure 1: Example code for the "Dire Wolf Alpha" Hearthstone card.

```
show me the fare from ci0 to ci1

lambda $0 e
  ( exists $1 ( and ( from $1 ci0 )
                    ( to $1 ci1 )
                    ( = ( fare $1 ) $0 ) ) )
```

Figure 2: Example of a query and its logical form from the ATIS dataset. The `ci0` and `ci1` tokens are entity abstractions introduced in preprocessing (Dong and Lapata, 2016).

Abstract Syntax Networks for Code Generation and Semantic Parsing, Rabinovich et. al., 2017

Abstract Syntax Networks for Code Generation and Semantic Parsin

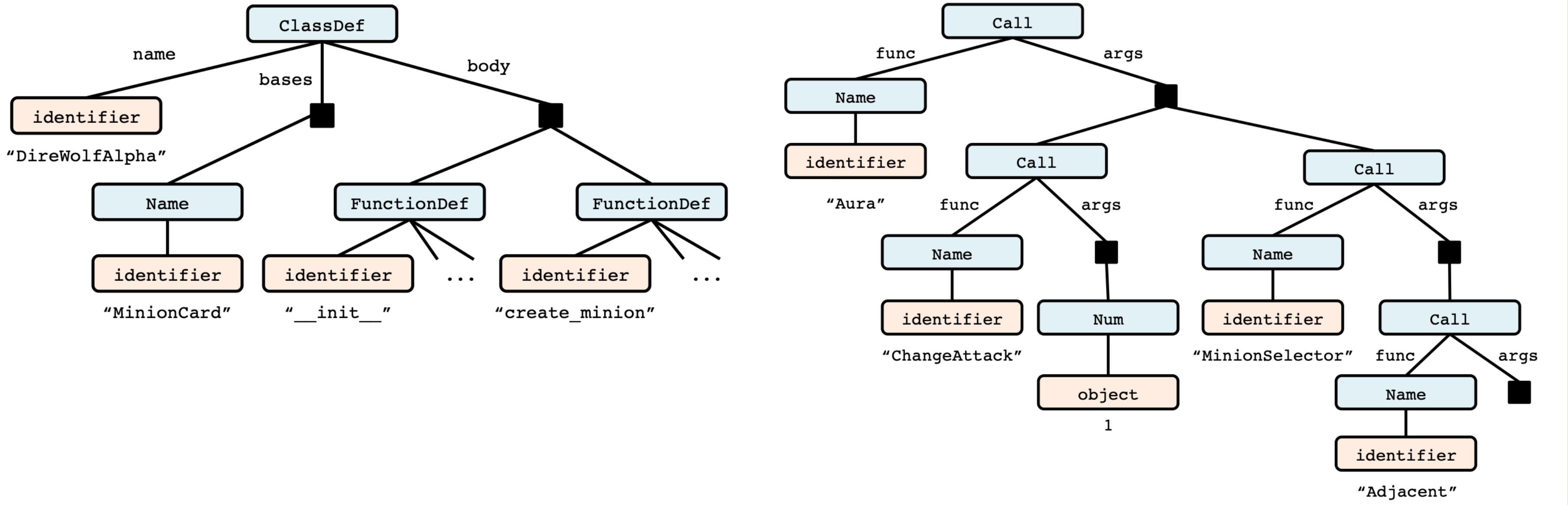
Maxim Rabinovich* **Mitchell Stern*** **Dan Klein**
Computer Science Division
University of California, Berkeley


```
name: [
  'D', 'i', 'r', 'e', ' ',
  'W', 'o', 'l', 'f', ' ',
  'A', 'l', 'p', 'h', 'a']
cost: ['2']
type: ['Minion']
rarity: ['Common']
race: ['Beast']
class: ['Neutral']
description: [
  'Adjacent', 'minions', 'have',
  '+', '1', 'Attack', '.']
health: ['2']
attack: ['2']
durability: ['-1']
```

```
:
ARACTER_CLASS.ALL,
n_type=MINION_TYPE.BEAST)
r):
ionSelector(Adjacent()))
```

e "Dire Wolf Alpha"

```
0 to ci1
om $1 ci0 )
$1 ci1 )
( fare $1 ) $0 ) ) )
```

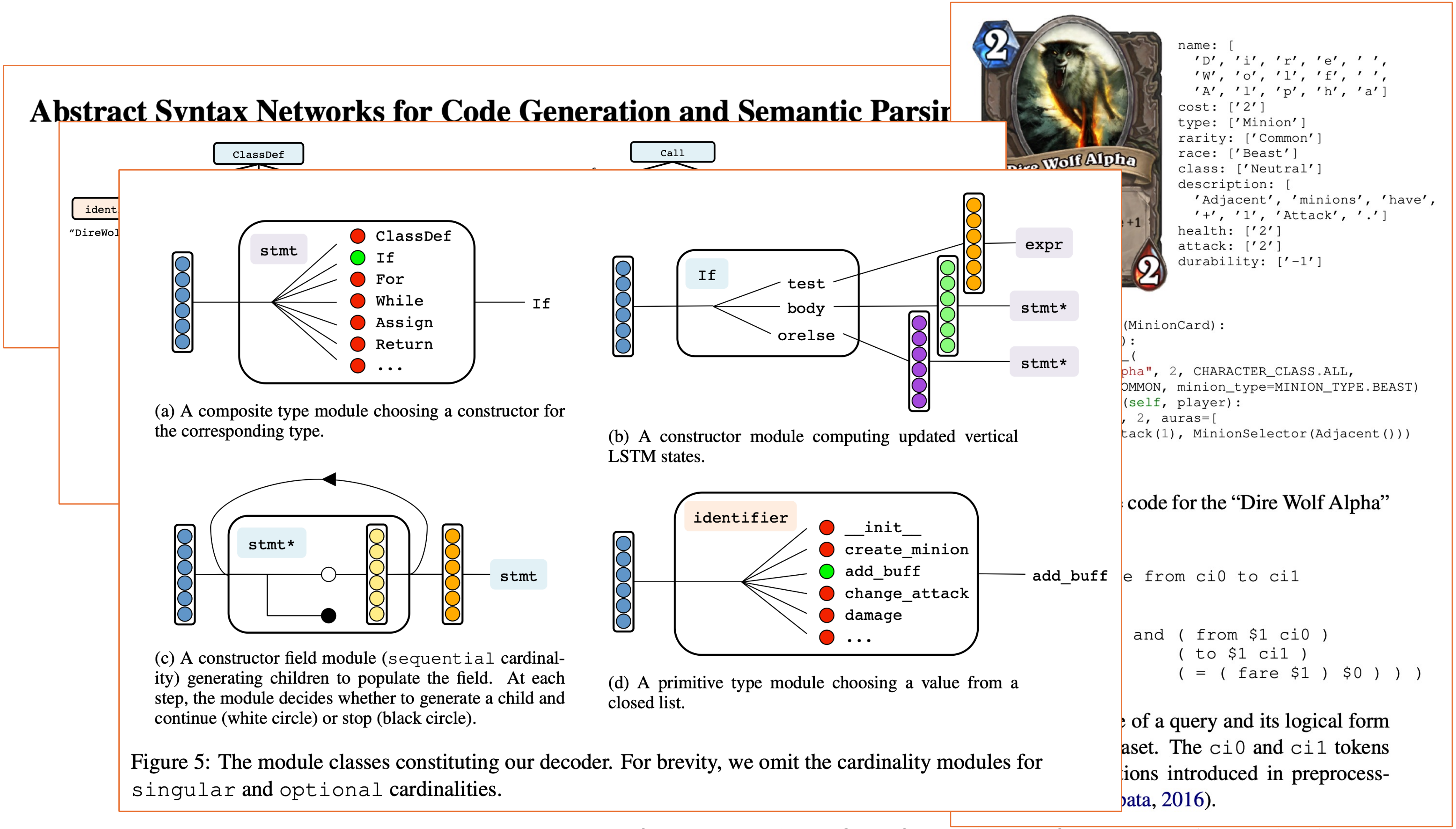


(a) The root portion of the AST.

(b) Excerpt from the same AST, corresponding to the code snippet `Aura(ChangeAttack(1),MinionSelector(Adjacent())).`

Figure 2: Example of a query and its logical form from the ATIS dataset. The ci0 and ci1 tokens are entity abstractions introduced in preprocessing (Dong and Lapata, 2016).

Abstract Syntax Networks for Code Generation and Semantic Parsing, Rabinovich et. al., 2017



Abstract Syntax Networks for Code Generation and Semantic Parsing, Rabinovich et. al., 2017

Constrained Decoding in the Wild

Grammar-Aligned Decoding

Kanghee Park[1*] **Jiayu Wang**[1*] **Taylor Berg-Kirkpatrick**[2]
Nadia Polikarpova[2] **Loris D'Antoni**[1]
[1]University of Wisconsin-Madison [2]University of California San Diego
{kpark247, jwang2782, ldantoni}@wisc.edu, {tberg, npolikarpova}@ucsd.edu

Grammar-Aligned Decoding, Park et. al., 2024

Grammar-Aligned Decoding

Kanghee Park[1*] Jiayu Wang[1*] Taylor Berg-Kirkpatrick[2]

Constrained decoding addresses the inefficiency of rejection sampling by greedily "forcing" the LLM output to satisfy the given constraint. Specifically, when the constraint is given as a grammar, *grammar-constrained decoding* (GCD) [7, 27, 28], can build automata that allow for on-the-fly masking of tokens that will provably lead to outputs outside of the grammar during decoding.

Grammar-Aligned Decoding Given a model distribution P and a CFG $\mathcal{G}$, *grammar-aligned decoding (GAD)* is the task of sampling from the distribution $Q^{P,\mathcal{G}}$ that is *proportional* to P but *restricted* to sentences in $\mathcal{G}$:

$$Q^{P,\mathcal{G}}(w) = \frac{\mathbb{1}[w \in \mathcal{L}(\mathcal{G})] \cdot P(w)}{\sum_{w'} \mathbb{1}[w' \in \mathcal{L}(\mathcal{G})] \cdot P(w')}$$

Grammar-Aligned Decoding, Park et. al., 2024

Grammar-Aligned Decoding

Kanghee Park[1*] **Jiayu Wang**[1*] **Taylor Berg-Kirkpatrick**[2]

Constrained decoding addresses the inefficiency of rejection sampling by greedily “forcing” the LLM output to satisfy the given constraint. Specifically, when the constraint is given as a grammar, *grammar-constrained decoding* (GCD) [7, 27, 28], can build automata that allow for on-the-fly masking of tokens that will provably lead to outputs outside of the grammar during decoding.

Grammar-Aligned Decoding Given a model distribution P and a CFG $\mathcal{G}$, *grammar-aligned decoding (GAD)* is the task of sampling from the distribution $Q^{P,\mathcal{G}}$ that is *proportional* to P but *restri*

```
Start ::=  S
    S ::=  name | " " | "."
        |  str.++ S S | str.at S I
        |  str.replace S S S
        |  str.substr S I I
    I ::=  0 | 1 | 2 | + I I | - I I
        |  str.len S | str.indexof S S I
```

(c) Grammar for `f`

```
Start ::=  BV
   BV ::=  s | t
        |  #x0 | #x7 | #x8
        |  bvneg BV | bvnot BV
        |  bvadd BV BV | bvsub BV BV
        |  bvand BV BV | bvlor BV BV
        |  bvlshl BV BV | bvlshr BV BV
```

(d) Grammar for `inv`

Grammar-Aligned Decoding, Park et. al., 2024

Constrained Decoding in the Wild

Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context

Lakshya A Agrawal
Microsoft Research
Bangalore, India
`t-lakagrawal@microsoft.com`

Aditya Kanade
Microsoft Research
Bangalore, India
`kanadeaditya@microsoft.com`

Navin Goyal
Microsoft Research
Bangalore, India
`navingo@microsoft.com`

Shuvendu K. Lahiri
Microsoft Research
Redmond, United States
`shuvendu.lahiri@microsoft.com`

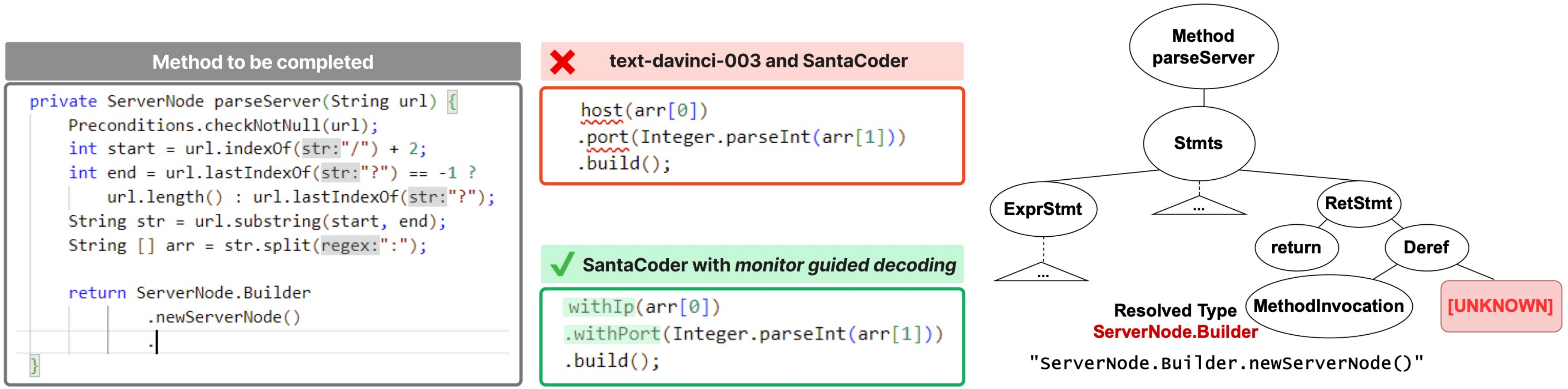
Sriram K. Rajamani
Microsoft Research
Bangalore, India
`sriram@microsoft.com`

Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context, Agrawal et. al., 2023

Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context

Lakshya A Agrawal
Microsoft Research
Bangalore, India
t-lakagrawal@microsoft.com

Aditya Kanade
Microsoft Research
Bangalore, India
kanadeaditya@microsoft.com



(a) Example where text-davinci-003 and SantaCoder generate wrong identifiers, but SantaCoder with MGD generates correct identifiers.

(b) Annotated partial AST for the code to the left.

Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context, Agrawal et. al., 2023

Monitor-Guided Decoding of Code LMs with Static

Integrated development environments (IDEs) have been at the forefront of assisting developers. Our inspiration is the use of static analysis by IDEs to bring the global context at the fingertips of developers. Many analyses are integrated in IDEs (Fuhrer, 2013) to infer and enforce semantic constraints on the code under development, e.g., resolving def-use, symbol references, and type hierarchies. Recently, there has been a rise in the use of Language Server Protocol (LSP) (lsp), which is an open industry standard of communication between IDEs and programming language specific tools like static analyzers and compilers, called Language Servers. There are a large number of Language Servers available, targetting most programming languages (lan, 2023), and providing a variety of syntactic and semantic information. In this work, we focus on the type-directed code completion analysis available through LSP in a language-agnostic manner, to provide guidance to an LM.

Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context, Agrawal et. al., 2023

Monitor-Guided Decoding of Code LMs with Static

Integrated development environments (IDEs) have been at the forefront of assisting developers.

Eq. (1) states that whenever the monitor is in the wait state s_0, we sample x_{n+1} as per the logits ℓ determined by the LM (Eq. (2)). Otherwise, the logits are combined with a mask m using a function $\oplus$ such that if $m[x] = 0$ then $\ell[x]$ is reset to a large negative value $-K$ and is left unchanged otherwise. This mask is computed by the function `maskgen` in Eq. (3) guided by the current state s of the monitor. Eq. (4) defines how the state of the monitor evolves. When the pre-condition $\texttt{pre}(s; x_1, \ldots, x_n)$ evaluates to true, the next state s' of the monitor is determined by the suggestions returned by the static analysis A_φ. Otherwise, it is determined by the `update` function.

$$(L_\theta || M_\varphi)(x_{n+1}|x_1, \ldots, x_n; C, p, s) = \begin{cases} \texttt{softmax}(\ell)[x_{n+1}] & \text{if } s = s_0 \text{ is the wait state} \\ \texttt{softmax}(\ell \oplus m)[x_{n+1}] & \text{otherwise} \end{cases} \tag{1}$$

$$\ell = L_\theta(\,\cdot\,|x_1, \ldots, x_n; p) \tag{2}$$

$$m = \texttt{maskgen}(s, V) \tag{3}$$

$$s' = \begin{cases} A_\varphi(x_1, \ldots, x_n; C) & \text{if } s = s_0 \wedge \texttt{pre}(s; x_1, \ldots, x_n) \\ \texttt{update}(s, x_{n+1}) & \text{otherwise} \end{cases} \tag{4}$$

Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context, Agrawal et. al., 2023

Syntactic and Semantic Control of Large Language Models via Sequential Monte Carlo

João Loula[*1] **Benjamin LeBrun**[*5] **Li Du**[*6] **Ben Lipkin**[1] **Clemente Pasti**[2] **Gabriel Grand**[1]
Tianyu Liu[2] **Yahya Emara**[2] **Marjorie Freedman**[8] **Jason Eisner**[6] **Ryan Cotterell**[2]
Vikash Mansinghka[‡1] **Alexander K. Lew**[‡1,7] **Tim Vieira**[‡2] **Timothy J. O'Donnell**[‡3,4,5]
[1]MIT [2]ETH Zürich [3]McGill [4]Canada CIFAR AI Chair [5]Mila [6]Johns Hopkins [7]Yale [8]ISI
genlm@mit.edu

Syntactic and Semantic Control of Large Language Models via Sequential Monte Carlo

João Loula*[1] Be
Tianyu Liu[2] Yah
Vikash Mansingh
[1]MIT [2]ETH Züri
genlm@mit.edu

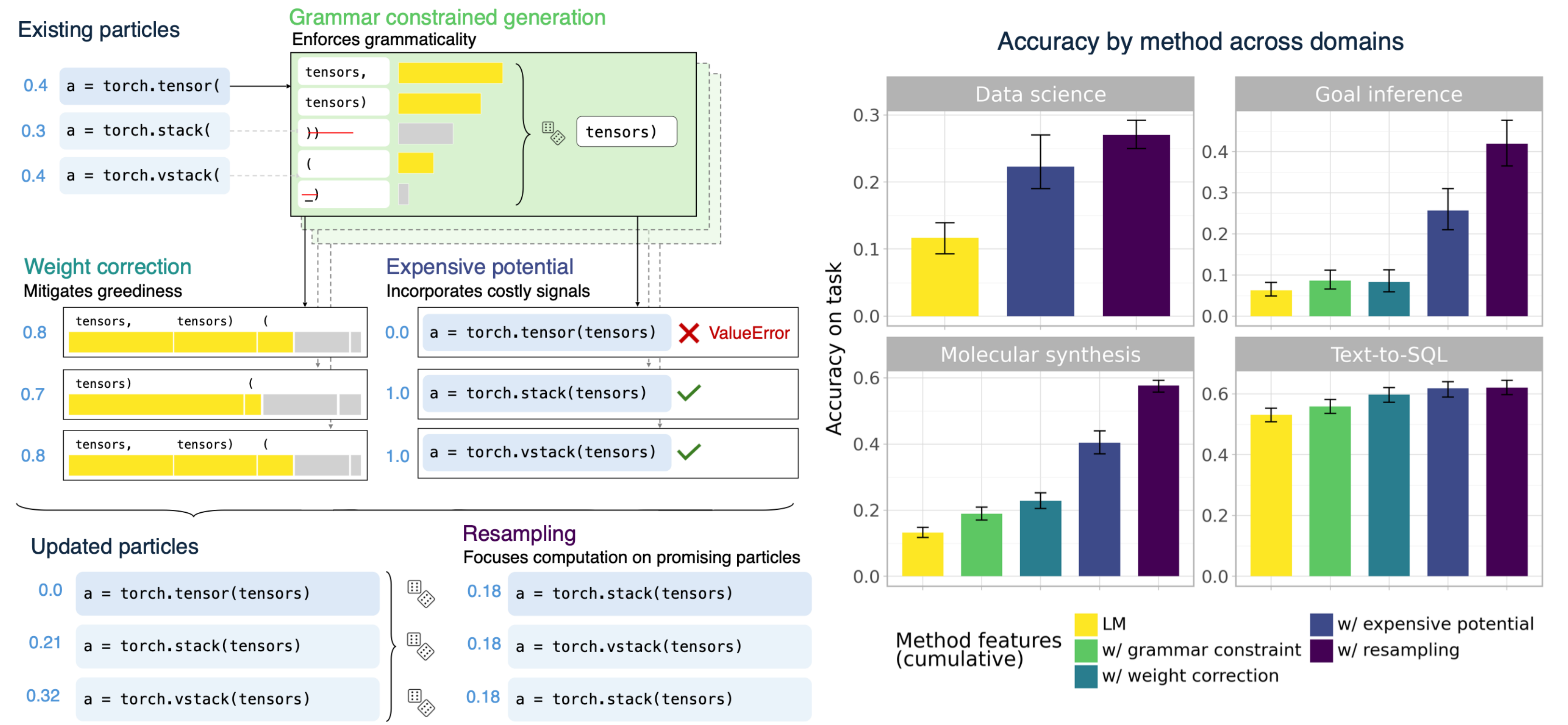


Figure 1: *Controlled generation from LMs via sequential Monte Carlo. Left:* We use sequential Monte Carlo to sample from high-quality approximations to posteriors over LM outputs. Partial sequences are repeatedly extended via grammar-constrained generation. We then apply weight corrections to mitigate the greediness of locally constrained decoding, as well as expensive potentials to encode rich information that cannot be included in logit masks. Finally, resampling focuses computation on promising particles. *Right:* Accuracy gains from these innovations on challenging data science, text-to-SQL, goal inference, and molecule synthesis benchmarks.

SYNTACTIC AND SEMANTIC CONTROL OF LARGE
LANGUAG

João Loula*[1] Be
Tianyu Liu[2] Yal
Vikash Mansing
[1]MIT [2]ETH Zür
genlm@mit.edu

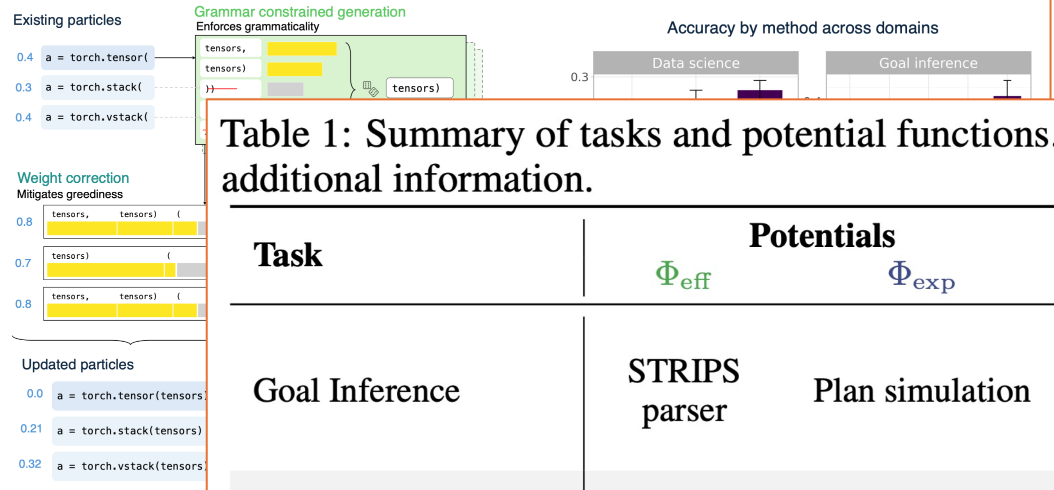
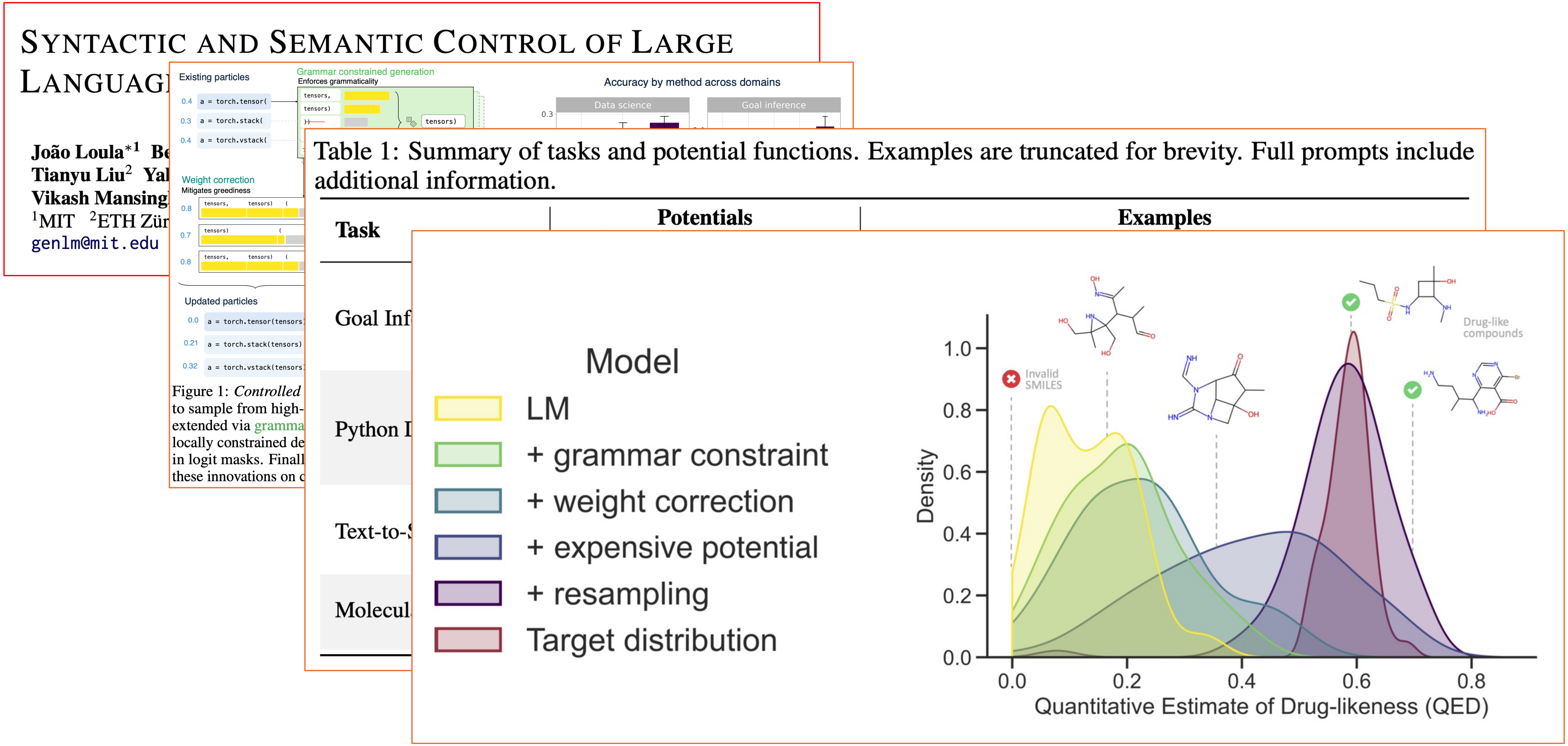


Figure 1: *Controlled*
to sample from high-
extended via gramma
locally constrained de
in logit masks. Finall
these innovations on c

Table 1: Summary of tasks and potential functions. Examples are truncated for brevity. Full prompts include additional information.

Task	Potentials		Examples	
	Φ_{eff}	Φ_{exp}	Prompt	Output
Goal Inference	STRIPS parser	Plan simulation	Write the STRIPS goal condition for the planning problem described below [...]. The STRIPS initial condition is: [...]	`(:goal (and (arm-empty) (on-table b1)` [...]
Python Data Science	-	Error-checking with test cases	Here is a sample dataframe: [...] I'd like to add inverses of each existing column to the dataframe [...]	`result = df.join(df.apply(lambda x: 1/x)` [...]
Text-to-SQL	SQL parser	Alias and table-column checking	Here is a database schema: [...] For each stadium, how many concerts are there?	`SELECT T2.name, COUNT(*) FROM concert AS T1` [...]
Molecular Synthesis	SMILES parser	Incremental molecule validation	Given the following list of molecules in SMILES format, write an additional molecule [...]	`CC1=CC2(OC=N)C(=O)` [...]



Syntactic and Semantic Control of Large Language Models via Sequential Monte Carlo, Loula et. al., ICLR 2025

Beyond Constrained Decoding

Programmatic Structured Steering

LMQL

A programming language for large language models.

Documentation »

Explore Examples · Playground IDE · Report Bug

chat 78 online · pypi package 0.7.3

Watch 21 · Fork 210 · Star 4.1k

Prompting Is Programming: A Query Language for Large Language Models

LUCA BEURER-KELLNER, MARC FISCHER, and MARTIN VECHEV, ETH Zurich, Switzerland

```
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(THING)) <= 2
```

Prompting Is Programming: A Query Language for Large Language Models, Beurer-Kellner et. al., PLDI 2023

{guidance}

Guidance is an efficient programming paradigm for steering language models. With Guidance, you can control how output is structured and get high-quality output for your use case—*while reducing latency and cost vs. conventional prompting or fine-tuning*. It allows users to constrain generation (e.g. with regex and CFGs) as well as to interleave control (conditionals, loops, tool use) and generation seamlessly.

Watch 123 | Fork 1.1k | Star 20.7k

```python
from guidance.library import one_or_more

@guidance(stateless=True)
def _gen_heading(lm: Model):
    lm += select(
        options=[_gen_text_in_tag("h1"), _gen_text_in_tag("h2"), _gen_text_in_tag("h3")]
    )
    lm += "\n"
    return lm

@guidance(stateless=True)
def _gen_para(lm: Model):
    lm += "<p>"
    lm += one_or_more(
        select(
            options=[
                _gen_text(),
                _gen_text_in_tag("em"),
                _gen_text_in_tag("strong"),
                "<br />",
            ],
        )
    )
    lm += "</p>\n"
    return lm
```

```python
import guidance

from guidance.models import Model

ASCII_OFFSET = ord("a")

@guidance
def zero_shot_multiple_choice(
    language_model: Model,
    question: str,
    choices: list[str],
):
    with user():
        language_model += question + "\n"
        for i, choice in enumerate(choices):
            language_model += f"{chr(i+ASCII_OFFSET)} : {choice}\n"

    with assistant():
        language_model += select(
            [chr(i + ASCII_OFFSET) for i in range(len(choices))], name="string_choice"
        )

    return language_model
```

Sequential Monte Carlo Steering of Large Language Models using Probabilistic Programs

Alexander K. Lew
MIT
alexlew@mit.edu

Tan Zhi-Xuan
MIT
xuan@mit.edu

Gabriel Grand
MIT
grandg@mit.edu

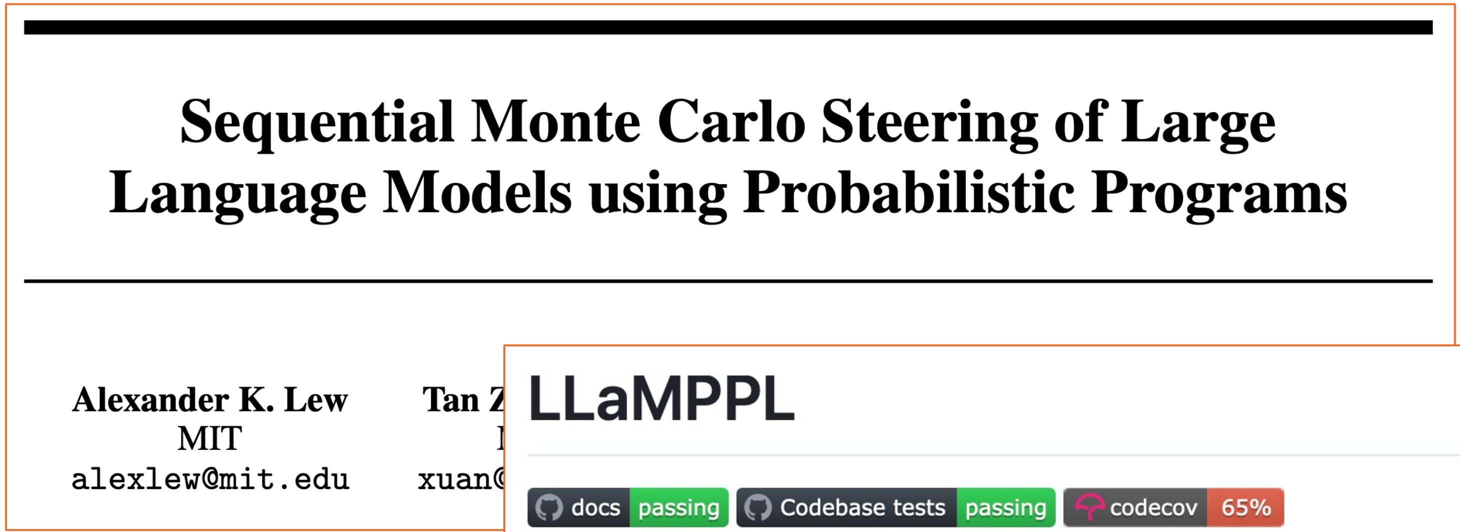
Vikash K. Mansinghka
MIT
vkm@mit.edu

```
# The step method is used to perform a single 'step' of generation.
# This might be a single token, a single phrase, or any other division.
# Here, we generate one token at a time.
async def step(self):
    # Condition on the next token *not* being a forbidden token.
    await self.observe(self.context.mask_dist(self.forbidden_tokens), False)

    # Sample the next token from the LLM -- automatically extends `self.context`.
    token = await self.sample(self.context.next_token())

    # Check for EOS or end of sentence
    if token.token_id == self.eos_token or str(token) in ['.', '!', '?']:
        # Finish generation
        self.finish()
```

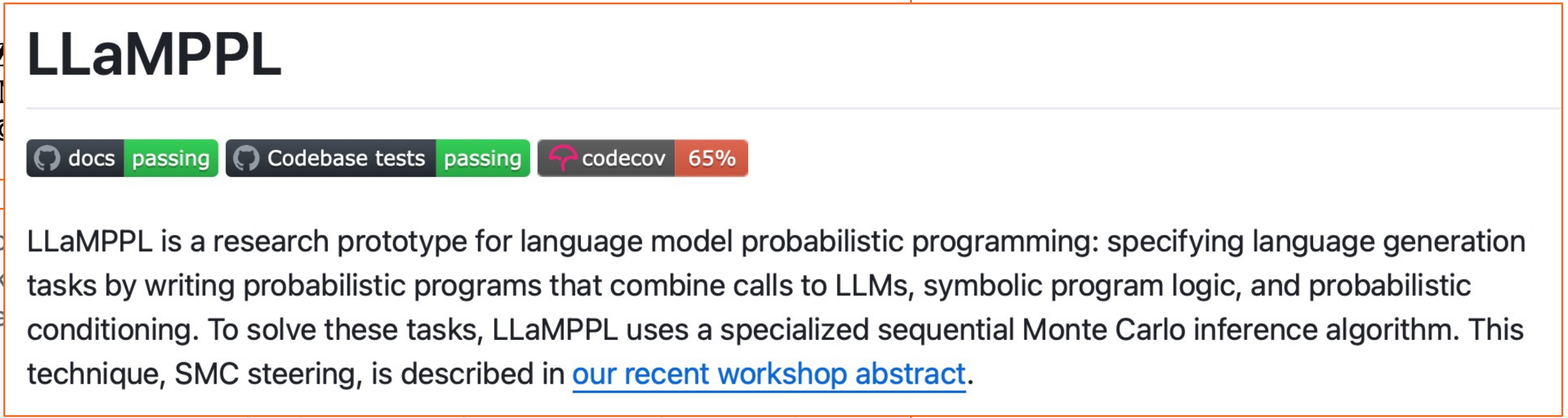
Sequential Monte Carlo Steering of Large Language Models using Probabilistic Programs, Lew et. al., 2025




```
# The step method is used to
# This might be a single tok
# Here, we generate one toke
async def step(self):
    # Condition on the next
    await self.observe(self.context.mask_dist(self.forbidden_tokens), False)

    # Sample the next token from the LLM -- automatically extends `self.context`.
    token = await self.sample(self.context.next_token())

    # Check for EOS or end of sentence
    if token.token_id == self.eos_token or str(token) in ['.', '!', '?']:
        # Finish generation
        self.finish()
```



Sequential Monte Carlo Steering of Large Language Models using Probabilistic Programs, Lew et. al., 2025

Summary

- We are building an arsenal of tools to generate good symbolic code from neural language models
  - Natural (Magical) strategies
    - Prompting
    - Prompt tuning
  - Mechanical strategies
    - Controlled decoding
  - Structural strategies
    - Iterative refinement
    - Programmatic steering
  - Agentic strategies
    - External tool use

We have not discussed…

- Syntactic constraints
  - Context free grammar
  - Regular expressions (Regex)
  - Finite state machine; Automaton; how to compile an automaton from CFG
  - Algorithms for lexing, parsing, compile a parsing, how to write AST
- Steering libraries
  - LMQL
  - Guidance
  - Langchain
  - DSPY

Week 4

- Assignment 1
  - https://github.com/machine-programming/assignment-1
  - Accepting late submissions
- Assignment 2
  - https://github.com/machine-programming/assignment-2
  - Due in two weeks; sending out another set of API keys; autograders
- Oral presentation starting from week 7
  - Sign-up sheet going out this week
- Feedback Questionnaire
  - Sending out this week
  - +0.5% of your overall grade