Machine Programming

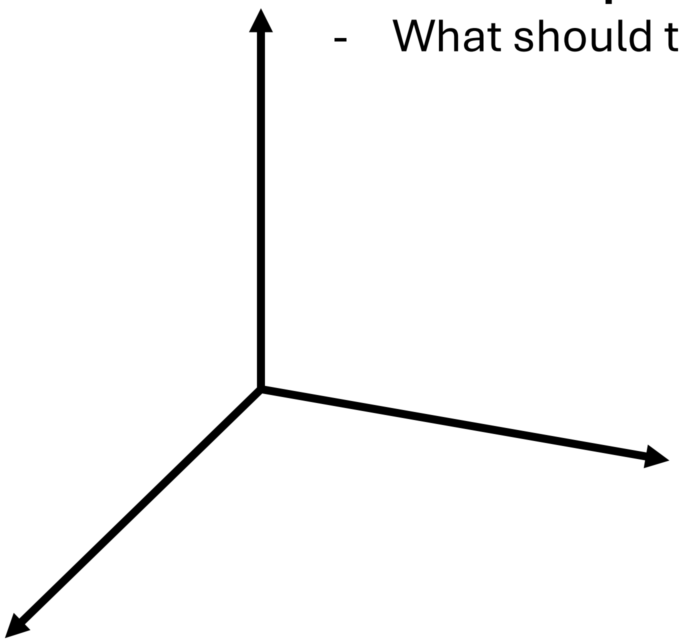
Lecture 5 – Language Modeling for Synthesis

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

Structural Specification

- What is the space of the programs?

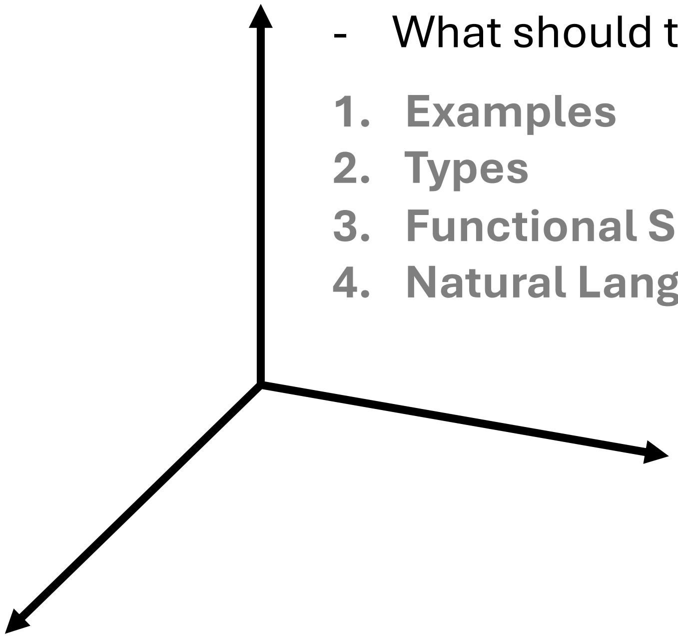
Context-Free / Regular Tree Grammar

Expr e ::= c | e + e | e * e

Today

Behavioral Specification
- What should the program do?

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



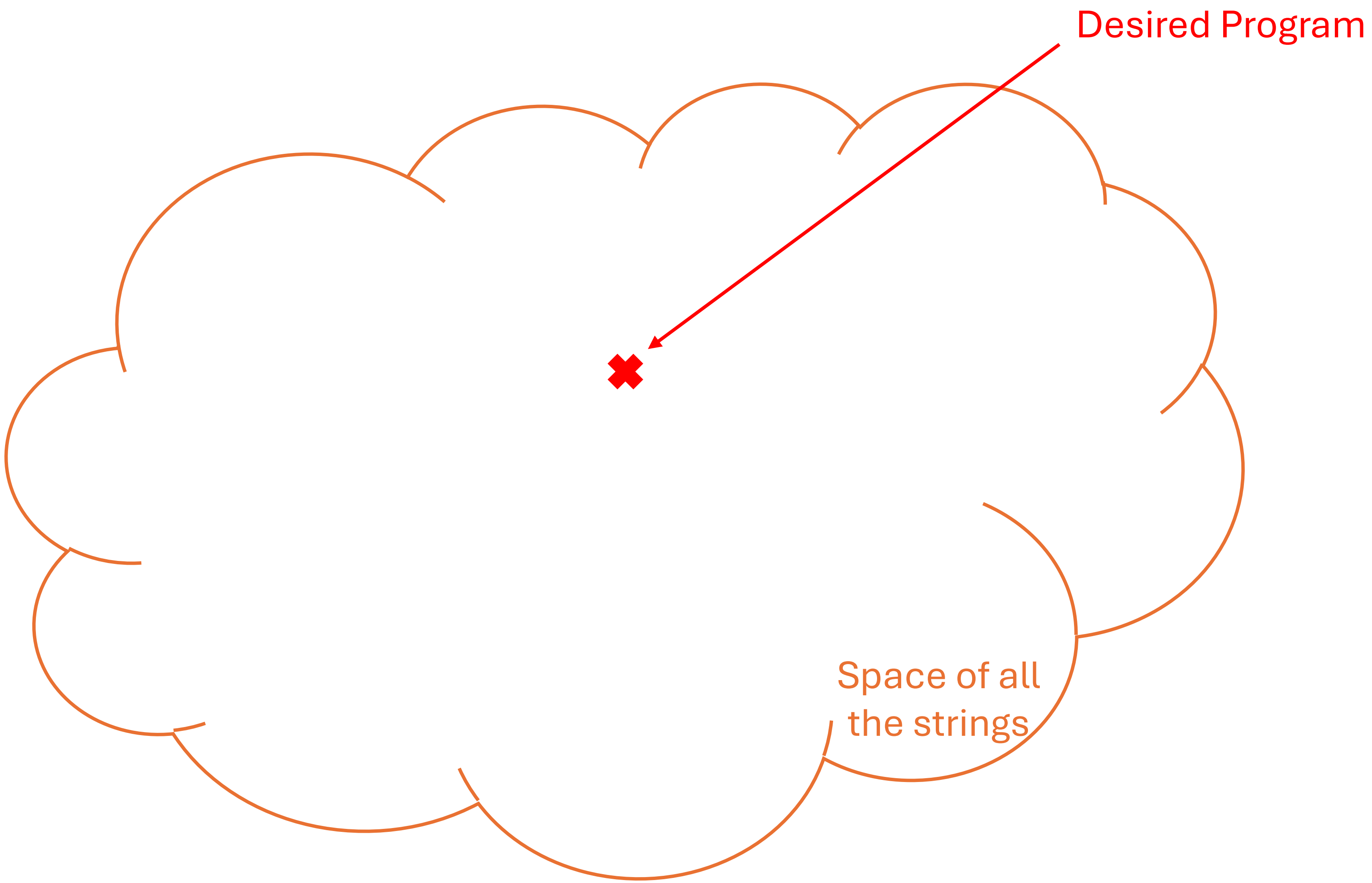
Synthesis Strategy
- How do we find such a program?

Data-Driven Approaches
- Modeling programs as token sequences

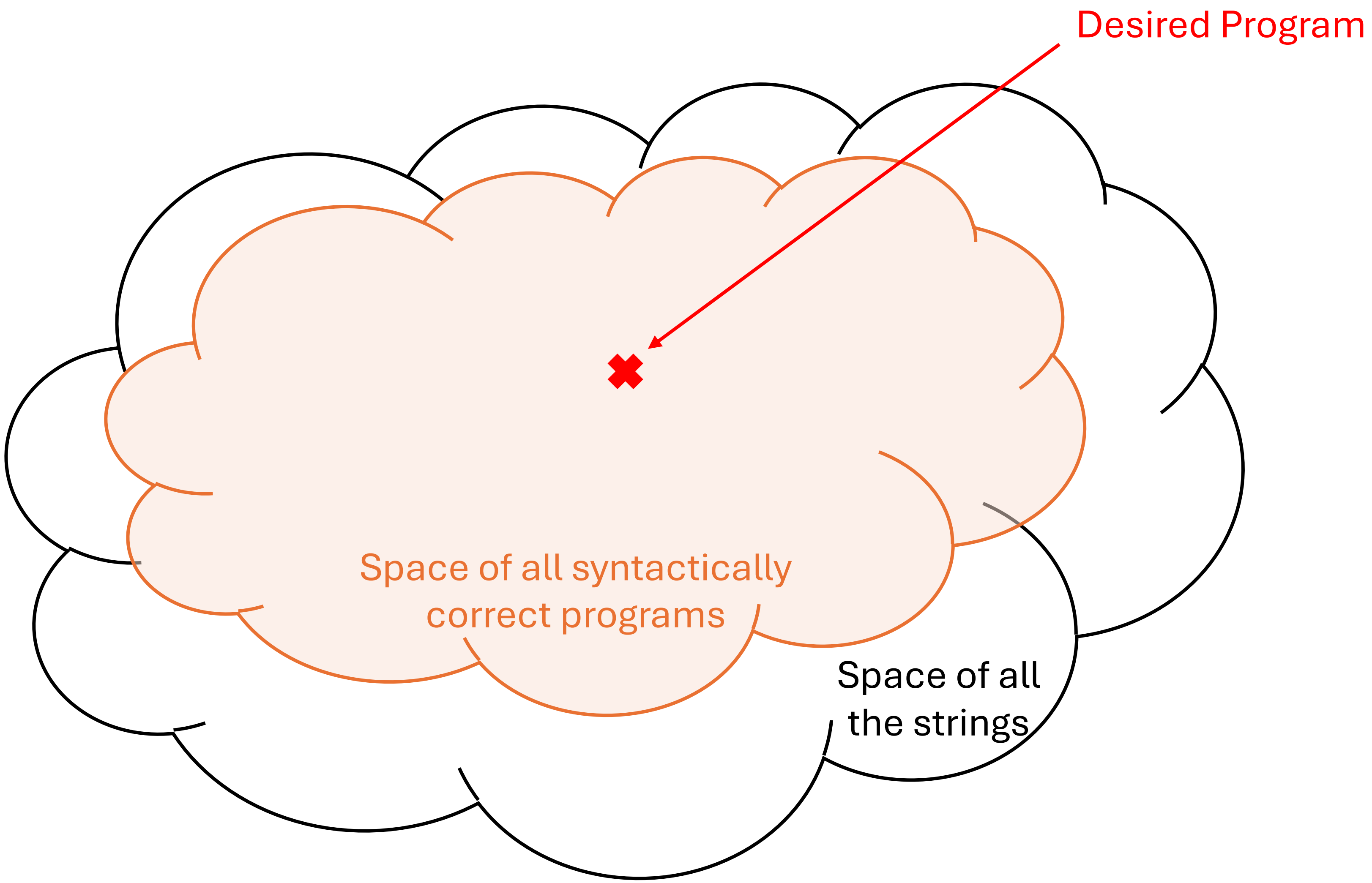
Structural Specification
- What is the space of the programs?

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

High Level Picture

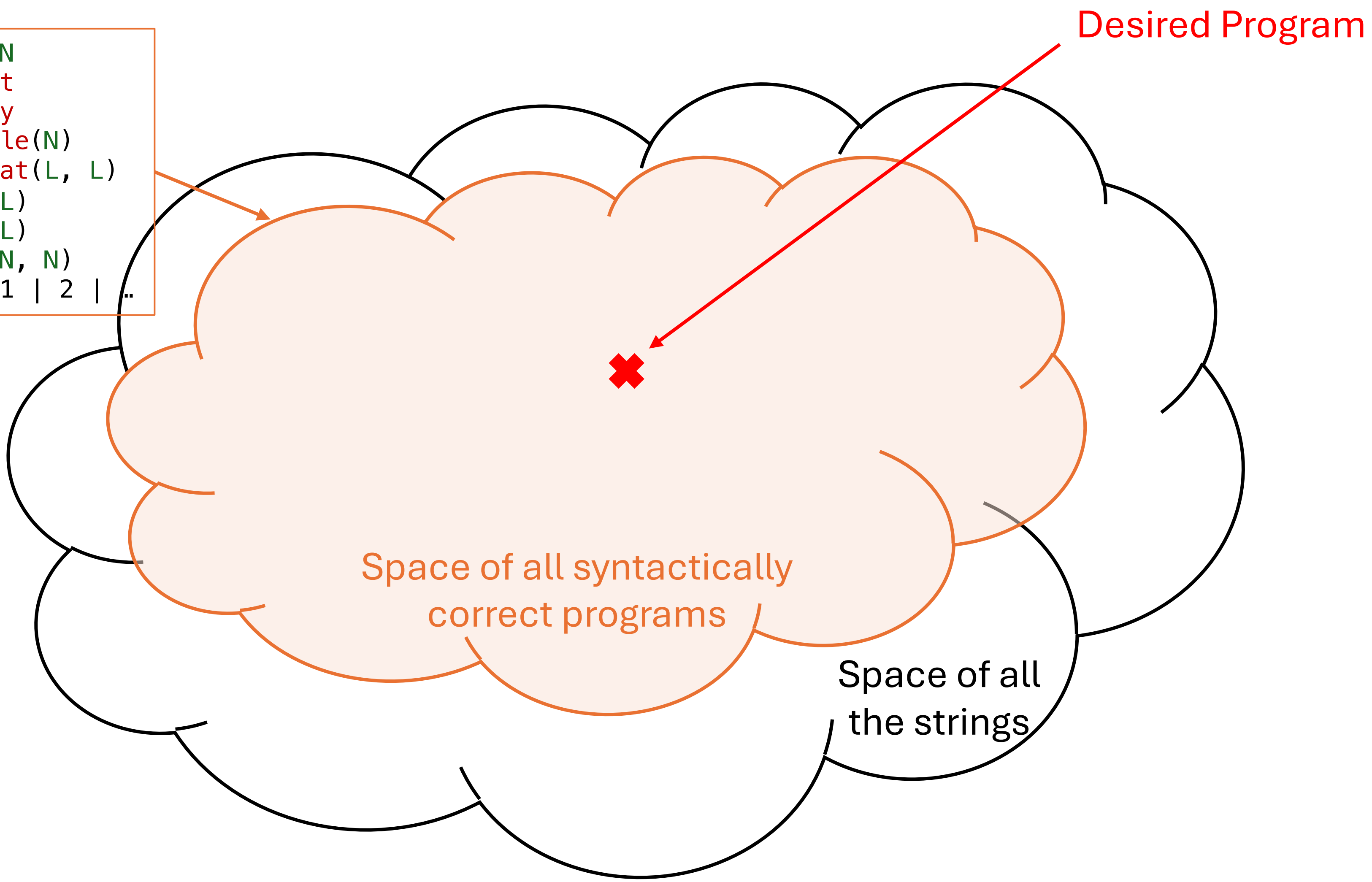


High Level Picture



High Level Picture

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

Small Domain Specific Language (DSL)

```
stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list, expr? returns,
                   string? type_comment, type_param* type_params)
      | AsyncFunctionDef(identifier name, arguments args,
                         stmt* body, expr* decorator_list, expr? retu
                         string? type_comment, type_param* type_param

      | ClassDef(identifier name,
         expr* bases,
         keyword* keywords,
         stmt* body,
         expr* decorator_list,
         type_param* type_params)
      | Return(expr? value)

      | Delete(expr* targets)
      | Assign(expr* targets, expr value, string? type_comment)
      | TypeAlias(expr name, type_param* type_params, expr value)
      | AugAssign(expr target, operator op, expr value)
      -- 'simple' indicates that we annotate simple name without pare
      | AnnAssign(expr target, expr annotation, expr? value, int simp

      -- use 'orelse' because else is a keyword in target languages
      | For(expr target, expr iter, stmt* body, stmt* orelse, string?
      | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, st
      | While(expr test, stmt* body, stmt* orelse)
      | If(expr test, stmt* body, stmt* orelse)
      | With(withitem* items, stmt* body, string? type_comment)
      | AsyncWith(withitem* items, stmt* body, string? type_comment)

      | Match(expr subject, match_case* cases)

      | Raise(expr? exc, expr? cause)
      | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt*
      | TryStar(stmt* body, excepthandler* handlers, stmt* orelse, st
      | Assert(expr test, expr? msg)

      | Import(alias* names)
      | ImportFrom(identifier? module, alias* names, int? level)

      | Global(identifier* names)
      | Nonlocal(identifier* names)
      | Expr(expr value)
      | Pass | Break | Continue

      -- col_offset is the byte offset in the utf8 string the parser
      attributes (int lineno, int col_offset, int? end_lineno, int? e

      -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
     | NamedExpr(expr target, expr value)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generators)
     | DictComp(expr key, expr value, comprehension* generators)
     | GeneratorExp(expr elt, comprehension* generators)
     -- the grammar constrains where yield expressions can occur
     | Await(expr value)
     | Yield(expr? value)
     | YieldFrom(expr value)
     -- need sequences for compare to distinguish between
     -- x < 4 < 3 and (x < 4) < 3
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Call(expr func, expr* args, keyword* keywords)
     | FormattedValue(expr value, int conversion, expr? format_spec)
     | JoinedStr(expr* values)
     | Constant(constant value, string? kind)

     -- the following expression can appear in assignment context
     | Attribute(expr value, identifier attr, expr_context ctx)
     | Subscript(expr value, expr slice, expr_context ctx)
     | Starred(expr value, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
     | Tuple(expr* elts, expr_context ctx)

     -- can appear only in Subscript
     | Slice(expr? lower, expr? upper, expr? step)

      -- col_offset is the byte offset in the utf8 string the parser
      attributes (int lineno, int col_offset, int? end_lineno, int? e

expr_context = Load | Store | Del

boolop = And | Or

operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
             | RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs, int is_async)
```

Formal Python 3 Abstract Syntax

https://docs.python.org/3/library/ast.html

```
     -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
     | NamedExpr(expr target, expr value)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
```

```
      -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
     | NamedExpr(expr target, expr value)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
```

```
lambda a, b:  a < b
```

```
Expr(
  value=Lambda(
    args=arguments(
      args=[
        arg(arg='a’),
        arg(arg='b')]),
    body=Compare(
      left=Name(id='a’),
      ops=[Lt()],
      comparators=[
        Name(id='b’)
      ])
  ))
```

```
| UnaryOp(unaryop op, expr operand)
| Lambda(arguments args, expr body)
| IfExp(expr test, expr body, expr orelse)
| Dict(expr* keys, expr* values)
| Set(expr* elts)
| ListComp(expr elt, comprehension* generators)
| SetComp(expr elt, comprehension* generators)
| DictComp(expr key, expr value, comprehension* ge
| GeneratorExp(expr elt, comprehension* generators
-- the grammar constrains where yield expressions
```

```
res = [i for i in arr if i > 0]
```

```
Assign(
  targets=[Name(id='res')],
  value=ListComp(
    elt=Name(id='i'),
    generators=[
      comprehension(
        target=Name(id='i'),
        iter=Name(id='arr'),
        ifs=[
          Compare(
            left=Name(id='i'),
            ops=[Gt()],
            comparators=[
              Constant(value=0)
            ])
        ])
    ]))
```

Practice

```
-- use 'orelse' because else is a keyword in target lar
  | For(expr target, expr iter, stmt* body, stmt* orelse,
  | AsyncFor(expr target, expr iter, stmt* body, stmt* or
  | While(expr test, stmt* body, stmt* orelse)
  | If(expr test, stmt* body, stmt* orelse)
  | With(withitem* items, stmt* body, string? type_commer
  | AsyncWith(withitem* items, stmt* body, string? type_c

  | Match(expr subject, match_case* cases)
```

```
  | Delete(expr* targets)
  | Assign(expr* targets, expr value, string? type_commer
  | TypeAlias(expr name, type_param* type_params, expr va
  | AugAssign(expr target, operator op, expr value)
```

```
while i < 10: i += 1
```

???

ChatGPT 5

Share

write me a python program that prints the ast of the program itself, while computing the depth and the size (number of AST nodes within the program)

Thought for 9s

python

Copy code

```python
#!/usr/bin/env python3
"""
Self-analysing script:
- Parses its own source into a Python AST
- Prints the AST (pretty-dumped)
- Computes and prints:
    * size  = total number of AST nodes (including the root `Module`)
    * depth = maximum path length from the root to any leaf (in nodes)
"""
import ast
import inspect
import sys

def ast_size(node: ast.AST) -> int:
    """Total count of nodes in the AST (includes the root)."""
    return sum(1 for _ in ast.walk(node))

def ast_depth(node: ast.AST) -> int:
    """Maximum depth of the AST (in nodes). Leaf has depth 1."""
    children = list(ast.iter_child_nodes(node))
    if not children:
        return 1
    return 1 + max(ast_depth(ch) for ch in children)

def main() -> None:
    # Get the current module's source (more robust than reading __file__)
    src = inspect.getsource(sys.modules[__name__])
```

Ask anything

Python: 43 lines

```python
import ast
import inspect
import sys

def ast_size(node: ast.AST) -> int:
    """Total count of nodes in the AST (includes the root)."""
    return sum(1 for _ in ast.walk(node))

def ast_depth(node: ast.AST) -> int:
    """Maximum depth of the AST (in nodes). Leaf has depth 1."""
    children = list(ast.iter_child_nodes(node))
    if not children:
        return 1
    return 1 + max(ast_depth(ch) for ch in children)

def main() -> None:
    # Get the current module's source (more robust than reading __file__)
    src = inspect.getsource(sys.modules[__name__])
    tree = ast.parse(src)

    # Pretty-print the AST
    print("=== AST (ast.dump with indentation) ===")
    print(ast.dump(tree, indent=2))

    # Compute metrics
    size = ast_size(tree)
    depth = ast_depth(tree)

    print("\n=== Metrics ===")
    print(f"Size (number of AST nodes): {size}")
    print(f"Depth (max root-to-leaf length): {depth}")

if __name__ == "__main__":
    main()
```

➔

AST: 181 Nodes, Depth 10

```
Module(
  body=[
    Expr(
      value=Constant(value='\nSelf-analysing script:\n- Parses its own source into a Python AST\n- Prints the AST (pretty-dumped)\n- Computes and prints:\n    * size  = total number of AST nodes (including the root `Module`)\n    * depth = maximum path length from the root to any leaf (in nodes)\n')),
    Import(
      names=[
        alias(name='ast')]),
    Import(
      names=[
        alias(name='inspect')]),
    Import(
      names=[
        alias(name='sys')]),
    FunctionDef(
      name='ast_size',
      args=arguments(
        args=[
          arg(
            arg='node',
            annotation=Attribute(
              value=Name(id='ast', ctx=Load()),
              attr='AST',
              ctx=Load()))]),
      body=[
        Expr(
          value=Constant(value='Total count of nodes in the AST (includes the root).')),
        Return(
          value=Call(
            func=Name(id='sum', ctx=Load()),
            args=[
              GeneratorExp(
                elt=Constant(value=1),
                generators=[
                  comprehension(
                    target=Name(id='_', ctx=Store()),
                    iter=Call(
                      func=Attribute(
                        value=Name(id='ast', ctx=Load()),
                        attr='walk',
                        ctx=Load()),
                      args=[
                        Name(id='node', ctx=Load())]),
                    is_async=0)])]))],
      returns=Name(id='int', ctx=Load())),
    FunctionDef(
      name='ast_depth',
      args=arguments(
        args=[
          arg(
            arg='node',
            annotation=Attribute(
              value=Name(id='ast', ctx=Load()),
              attr='AST',
              ctx=Load()))]),
      body=[
        Expr(
          value=Constant(value='Maximum depth of the AST (in nodes). Leaf has depth 1.')),
        Assign(
          targets=[
            Name(id='children', ctx=Store())],
          value=Call(
            func=Name(id='list', ctx=Load()),
            args=[
              Call(
                func=Attribute(
                  value=Name(id='ast', ctx=Load()),
                  attr='iter_child_nodes',
                  ctx=Load()),
                args=[
                  Name(id='node', ctx=Load())])])),
        If(
          test=UnaryOp(
            op=Not(),
            operand=Name(id='children', ctx=Load())),
          body=[
            Return(
              value=Constant(value=1))]),
        Return(
          value=BinOp(
            left=Constant(value=1),
            op=Add(),
            right=Call(
              func=Name(id='max', ctx=Load()),
              args=[
                GeneratorExp(
                  elt=Call(
                    func=Name(id='ast_depth', ctx=Load()),
                    args=[
                      Name(id='ch', ctx=Load())]),
                  generators=[
                    comprehension(
                      target=Name(id='ch', ctx=Store()),
                      iter=Name(id='children', ctx=Load()),
                      is_async=0)])])))],
      returns=Name(id='int', ctx=Load())),
    FunctionDef(
      name='main',
      args=arguments(),
      body=[
        Assign(
          targets=[
            Name(id='src', ctx=Store())],
          value=Call(
            func=Attribute(
              value=Name(id='inspect', ctx=Load()),
              attr='getsource',
              ctx=Load()),
            args=[
              Subscript(
                value=Attribute(
                  value=Name(id='sys', ctx=Load()),
                  attr='modules',
                  ctx=Load()),
                slice=Name(id='__name__', ctx=Load()),
                ctx=Load())])),
        Assign(
          targets=[
            Name(id='tree', ctx=Store())],
          value=Call(
            func=Attribute(
              value=Name(id='ast', ctx=Load()),
              attr='parse',
              ctx=Load()),
            args=[
              Name(id='src', ctx=Load())])),
        Expr(
          value=Call(
            func=Name(id='print', ctx=Load()),
            args=[
              Constant(value='=== AST (ast.dump with indentation) ===')])),
        Expr(
          value=Call(
            func=Name(id='print', ctx=Load()),
            args=[
              Call(
                func=Attribute(
                  value=Name(id='ast', ctx=Load()),
                  attr='dump',
                  ctx=Load()),
                args=[
                  Name(id='tree', ctx=Load())],
                keywords=[
                  keyword(
                    arg='indent',
                    value=Constant(value=2))])])),
        Assign(
          targets=[
            Name(id='size', ctx=Store())],
          value=Call(
            func=Name(id='ast_size', ctx=Load()),
            args=[
              Name(id='tree', ctx=Load())])),
        Assign(
          targets=[
            Name(id='depth', ctx=Store())],
          value=Call(
            func=Name(id='ast_depth', ctx=Load()),
            args=[
              Name(id='tree', ctx=Load())])),
        Expr(
          value=Call(
            func=Name(id='print', ctx=Load()),
            args=[
              Constant(value='\n=== Metrics ===')])),
        Expr(
          value=Call(
            func=Name(id='print', ctx=Load()),
            args=[
              JoinedStr(
                values=[
                  Constant(value='Size (number of AST nodes): '),
                  FormattedValue(
                    value=Name(id='size', ctx=Load()),
                    conversion=-1)])])),
        Expr(
          value=Call(
            func=Name(id='print', ctx=Load()),
            args=[
              JoinedStr(
                values=[
                  Constant(value='Depth (max root-to-leaf length): '),
                  FormattedValue(
                    value=Name(id='depth', ctx=Load()),
                    conversion=-1)])]))],
      returns=Constant(value=None)),
    If(
      test=Compare(
        left=Name(id='__name__', ctx=Load()),
        ops=[
          Eq()],
        comparators=[
          Constant(value='__main__')]),
      body=[
        Expr(
          value=Call(
            func=Name(id='main', ctx=Load())))])])
```

Python: 43 lines

```python
import ast
import inspect
import sys

def ast_size(node: ast.AST) -> int:
    """Total count of nodes in the AST (includes the root)."""
    return sum(1 for _ in ast.walk(node))

def ast_depth(node: ast.AST) -> int:
    """Maximum depth of the AST (in nodes). Leaf has depth 1."""
    children = list(ast.iter_child_nodes(node))
    if not children:
        return 1
    return 1 + max(ast_depth(ch) for ch in children)

def main() -> None:
    # Get the current module's source (more robust than reading __file__)
    src = inspect.getsource(sys.modules[__name__])
    tree = ast.parse(src)

    # Pretty-print the AST
    print("=== AST (ast.dump with indentation) ===")
    print(ast.dump(tree, indent=2))

    # Compute metrics
    size = ast_size(tree)
    depth = ast_depth(tree)

    print("\n=== Metrics ===")
    print(f"Size (number of AST nodes): {size}")
    print(f"Depth (max root-to-leaf length): {depth}")

if __name__ == "__main__":
    main()
```

➔

```
          alias(name='sys')]),
    FunctionDef(
      name='ast_size',
      args=arguments(
        args=[
          arg(
            arg='node',
            annotation=Attribute(
              value=Name(id='ast', ctx=Load()),
              attr='AST',
              ctx=Load()))]),
      body=[
        Expr(
          value=Constant(value='Total count of nodes
in the AST (includes the root).')),
        Return(
          value=Call(
            func=Name(id='sum', ctx=Load()),
            args=[
              GeneratorExp(
                elt=Constant(value=1),
                generators=[
                  comprehension(
                    target=Name(id='_', ctx=Store()),
                    iter=Call(
                      func=Attribute(
                        value=Name(id='ast',
ctx=Load()),
                        attr='walk',
                        ctx=Load()),
                      args=[
                        Name(id='node', ctx=Load())]),
                    is_async=0)])]))],
      returns=Name(id='int', ctx=Load())),
    FunctionDef(
      name='ast_depth',
      args=arguments(
        args=[
          arg(
            arg='node',
            annotation=Attribute(
              value=Name(id='ast', ctx=Load()),
              attr='AST',
              ctx=Load()))]),
      body=[
        Expr(
```

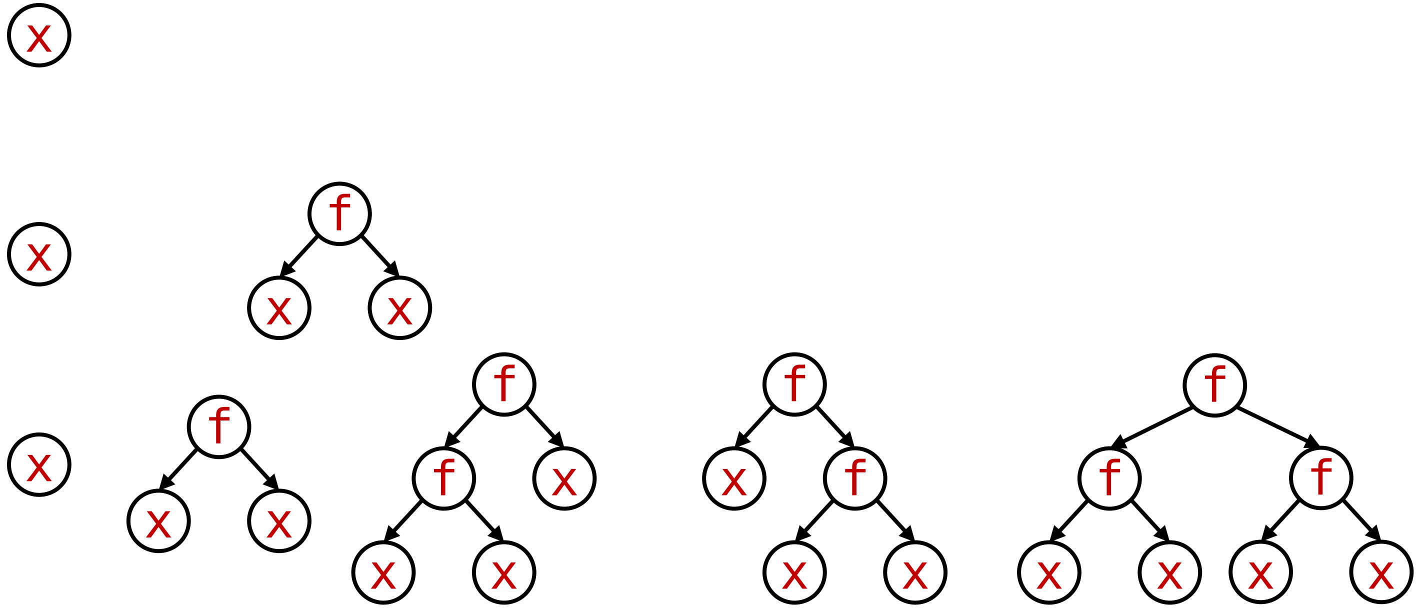
Syntax: How big is the space of programs?

```
E ::= x | f(E, E)
```

Depth <= 0

Depth <= 1

Depth <= 2



`Size(0) = 1`

`Size(1) = 2`

`Size(2) = 5`

`Size(depth) = 1 + Size(depth – 1)²`

Syntax: How big is the space of programs?

```
E ::= x | f(E, E)
```

$$\text{Size}(\text{depth}) = 1 + \text{Size}(\text{depth} - 1)^2$$

```
size(1) = 1
size(2) = 2
size(3) = 5
size(4) = 26
size(5) = 677
size(6) = 458330
size(7) = 210066388901
size(8) = 44127887745906175987802
size(9) = 1947270476915296449559703445493848930452791205
size(10) = 3791862310265926082868235028027893277370233152247388584761734150717768254410341175325352026
```

Syntax: How big is the space of programs?

$$E ::= \underbrace{x \mid f1(E,\ E) \mid \dots \mid fn(\overbrace{E1,\ E2,\ \dots,\ Em}^{m \text{ arguments}})}_{n \text{ production rules}}$$

$$\mathrm{Size}(\mathrm{depth}) = ???$$

$$E ::= x \mid f(E,\ E)$$

$$\mathrm{Size}(\mathrm{depth}) = 1 + \mathrm{Size}(\mathrm{depth} - 1)^2$$

Syntax: How big is the space of programs?

$$E ::= \underbrace{x \mid f1(E, E) \mid \ldots \mid fn(\overbrace{E1, E2, \ldots, Em}^{m \text{ arguments}})}_{n \text{ production rules}}$$

$$\mathrm{Size}(0) = 1$$

$$\mathrm{Size}(depth + 1) = 1 + (n - 1)\ \mathrm{Size}(depth)^{m}$$

$$E ::= x \mid f(E, E)$$

$$\mathrm{Size}(depth) = 1 + \mathrm{Size}(depth - 1)^{2}$$

Syntax: How big is the space of programs?

$$\texttt{E ::= } \underbrace{\texttt{x | f1(E, E) | ... | fn}\overbrace{\texttt{(E1, E2, …, Em)}}^{\text{m arguments}}}_{\text{n production rules}}$$

```
        Size(0) = 1
Size(depth + 1) = 1 + (n − 1) Size(depth)^m
```

$$\mathrm{Size}(d) = \boldsymbol{O}(n^{m^d})$$

```
E ::= x | f(E, E)
```

```
Size(depth) = 1 + Size(depth − 1)^2
```

How big is the space of Python programs?

```
stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list, expr? returns,
                   string? type_comment, type_param* type_params)
      | AsyncFunctionDef(identifier name, arguments args,
                         stmt* body, expr* decorator_list, expr? retu
                         string? type_comment, type_param* type_param

      | ClassDef(identifier name,
         expr* bases,
         keyword* keywords,
         stmt* body,
         expr* decorator_list,
         type_param* type_params)
      | Return(expr? value)

      | Delete(expr* targets)
      | Assign(expr* targets, expr value, string? type_comment)
      | TypeAlias(expr name, type_param* type_params, expr value)
      | AugAssign(expr target, operator op, expr value)
      -- 'simple' indicates that we annotate simple name without pare
      | AnnAssign(expr target, expr annotation, expr? value, int simp

      -- use 'orelse' because else is a keyword in target languages
      | For(expr target, expr iter, stmt* body, stmt* orelse, string?
      | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, st
      | While(expr test, stmt* body, stmt* orelse)
      | If(expr test, stmt* body, stmt* orelse)
      | With(withitem* items, stmt* body, string? type_comment)
      | AsyncWith(withitem* items, stmt* body, string? type_comment)

      | Match(expr subject, match_case* cases)

      | Raise(expr? exc, expr? cause)
      | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt*
      | TryStar(stmt* body, excepthandler* handlers, stmt* orelse, st
      | Assert(expr test, expr? msg)

      | Import(alias* names)
      | ImportFrom(identifier? module, alias* names, int? level)

      | Global(identifier* names)
      | Nonlocal(identifier* names)
      | Expr(expr value)
      | Pass | Break | Continue

      -- col_offset is the byte offset in the utf8 string the parser
      attributes (int lineno, int col_offset, int? end_lineno, int? e

      -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
     | NamedExpr(expr target, expr value)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generators)
     | DictComp(expr key, expr value, comprehension* generators)
     | GeneratorExp(expr elt, comprehension* generators)
     -- the grammar constrains where yield expressions can occur
     | Await(expr value)
     | Yield(expr? value)
     | YieldFrom(expr value)
     -- need sequences for compare to distinguish between
     -- x < 4 < 3 and (x < 4) < 3
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Call(expr func, expr* args, keyword* keywords)
     | FormattedValue(expr value, int conversion, expr? format_spec)
     | JoinedStr(expr* values)
     | Constant(constant value, string? kind)

     -- the following expression can appear in assignment context
     | Attribute(expr value, identifier attr, expr_context ctx)
     | Subscript(expr value, expr slice, expr_context ctx)
     | Starred(expr value, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
     | Tuple(expr* elts, expr_context ctx)

     -- can appear only in Subscript
     | Slice(expr? lower, expr? upper, expr? step)

      -- col_offset is the byte offset in the utf8 string the parser
      attributes (int lineno, int col_offset, int? end_lineno, int? e

expr_context = Load | Store | Del

boolop = And | Or

operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
              | RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs, int is_async)
```

$$\mathrm{Size}(0) = 1$$

$$\mathrm{Size}(\mathrm{depth} + 1) = 1 + (n - 1)\ \mathrm{Size}(\mathrm{depth})^m$$

$$\mathrm{Size}(d) = \boldsymbol{O}(n^{m^d})$$

Python 3:
- Production rules: ~110 ($n \approx 110$)
- Maximum Arity: ~7 ($m \approx 7$)

Note:
- Not counting that there are an enormous number of integers and string literals

General Purpose Programming Languages

```
stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list, expr? returns,
                   string? type_comment, type_param* type_params)
      | AsyncFunctionDef(identifier name, arguments args,
                         stmt* body, expr* decorator_list, expr? retu
                         string? type_comment, type_param* type_param

      | ClassDef(identifier name,
         expr* bases,
         keyword* keywords,
         stmt* body,
         expr* decorator_list,
         type_param* type_params)
      | Return(expr? value)

      | Delete(expr* targets)
      | Assign(expr* targets, expr value, string? type_comment)
      | TypeAlias(expr name, type_param* type_params, expr value)
      | AugAssign(expr target, operator op, expr value)
      -- 'simple' indicates that we annotate simple name without pare
      | AnnAssign(expr target, expr annotation, expr? value, int simp

      -- use 'orelse' because else is a keyword in target languages
      | For(expr target, expr iter, stmt* body, stmt* orelse, string?
      | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, st
      | While(expr test, stmt* body, stmt* orelse)
      | If(expr test, stmt* body, stmt* orelse)
      | With(withitem* items, stmt* body, string? type_comment)
      | AsyncWith(withitem* items, stmt* body, string? type_comment)

      | Match(expr subject, match_case* cases)

      | Raise(expr? exc, expr? cause)
      | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt*
      | TryStar(stmt* body, excepthandler* handlers, stmt* orelse, st
      | Assert(expr test, expr? msg)

      | Import(alias* names)
      | ImportFrom(identifier? module, alias* names, int? level)

      | Global(identifier* names)
      | Nonlocal(identifier* names)
      | Expr(expr value)
      | Pass | Break | Continue

      -- col_offset is the byte offset in the utf8 string the parser
      attributes (int lineno, int col_offset, int? end_lineno, int? e

      -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
     | NamedExpr(expr target, expr value)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generators)
     | DictComp(expr key, expr value, comprehension* generators)
     | GeneratorExp(expr elt, comprehension* generators)
     -- the grammar constrains where yield expressions can occur
     | Await(expr value)
     | Yield(expr? value)
     | YieldFrom(expr value)
     -- need sequences for compare to distinguish between
     -- x < 4 < 3 and (x < 4) < 3
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Call(expr func, expr* args, keyword* keywords)
     | FormattedValue(expr value, int conversion, expr? format_spec)
     | JoinedStr(expr* values)
     | Constant(constant value, string? kind)

     -- the following expression can appear in assignment context
     | Attribute(expr value, identifier attr, expr_context ctx)
     | Subscript(expr value, expr slice, expr_context ctx)
     | Starred(expr value, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
     | Tuple(expr* elts, expr_context ctx)

     -- can appear only in Subscript
     | Slice(expr? lower, expr? upper, expr? step)

      -- col_offset is the byte offset in the utf8 string the parser
      attributes (int lineno, int col_offset, int? end_lineno, int? e

expr_context = Load | Store | Del

boolop = And | Or

operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
             | RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs, int is_async)
```

Python (link)

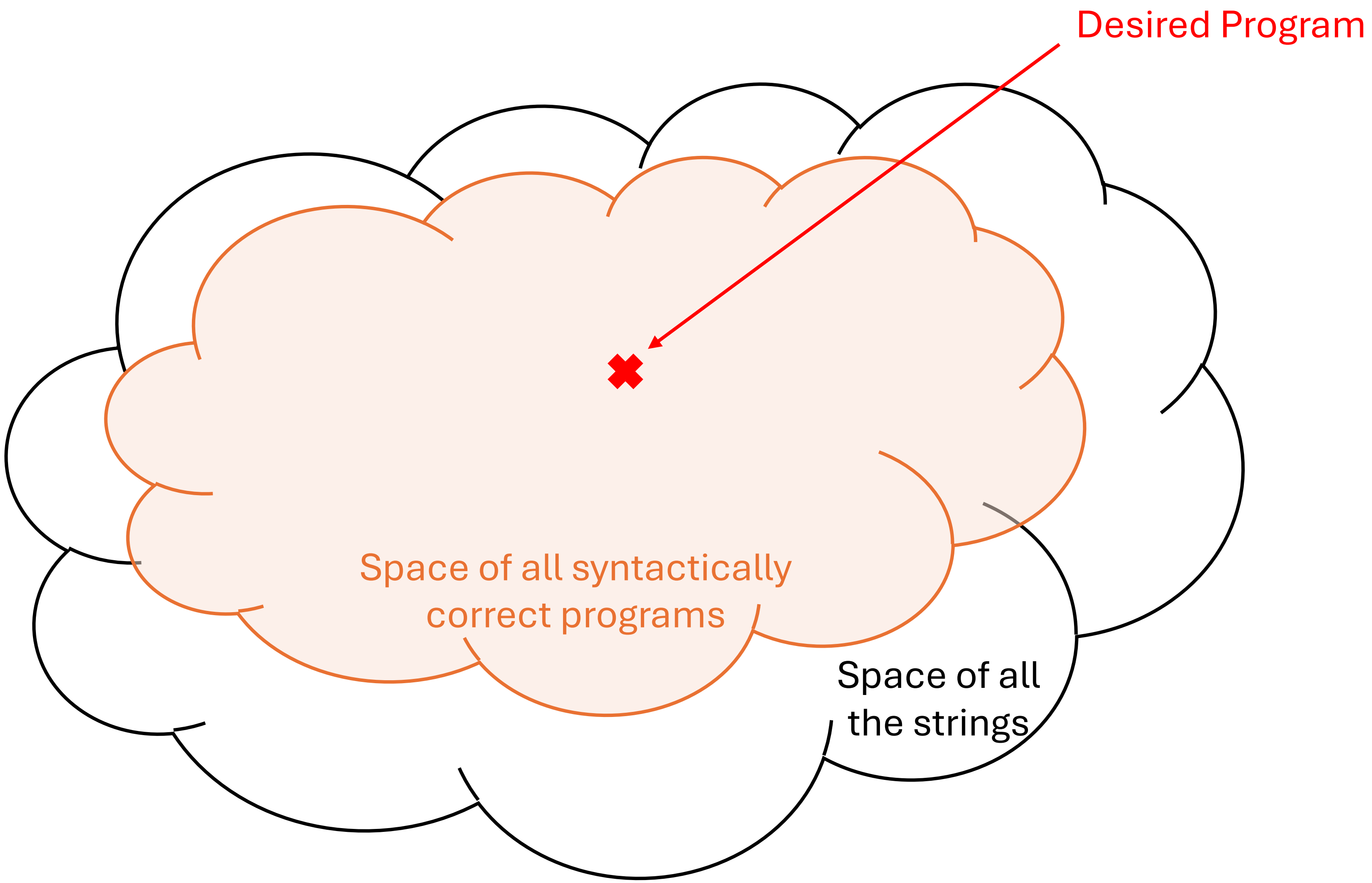
```
primary_expression : IDENTIFIER | constant | string | '(' expression ')' | generic_selection
constant : I_CONSTANT | F_CONSTANT | ENUMERATION_CONSTANT
enumeration_constant : IDENTIFIER
string : STRING_LITERAL | FUNC_NAME
generic_selection : GENERIC '(' assignment_expression ',' generic_assoc_list ')'
generic_assoc_list : generic_association | generic_assoc_list ',' generic_association
generic_association : type_name ':' assignment_expression | DEFAULT ':' assignment_expression
postfix_expression : primary_expression | postfix_expression '[' expression ']' | postfix_expression '(' ')' | postfix_expression '(' argument_expression_list
')' | postfix_expression '.' IDENTIFIER | postfix_expression PTR_OP IDENTIFIER | postfix_expression INC_OP | postfix_expression DEC_OP | '(' type_name ')' '{'
initializer_list '}' | '(' type_name ')' '{' initializer_list ',' '}'
argument_expression_list : assignment_expression | argument_expression_list ',' assignment_expression
unary_expression : postfix_expression | INC_OP unary_expression | DEC_OP unary_expression | unary_operator cast_expression | SIZEOF unary_expression | SIZEOF '('
type_name ')' | ALIGNOF '(' type_name ')'
unary_operator : '&' | '*' | '+' | '-' | '~' | '!'
cast_expression : unary_expression | '(' type_name ')' cast_expression
multiplicative_expression : cast_expression | multiplicative_expression '*' cast_expression | multiplicative_expression '/' cast_expression |
multiplicative_expression '%' cast_expression
additive_expression : multiplicative_expression | additive_expression '+' multiplicative_expression | additive_expression '-' multiplicative_expression
shift_expression : additive_expression | shift_expression LEFT_OP additive_expression | shift_expression RIGHT_OP additive_expression
relational_expression : shift_expression | relational_expression '<' shift_expression | relational_expression '>' shift_expression | relational_expression LE_OP
shift_expression | relational_expression GE_OP shift_expression
equality_expression : relational_expression | equality_expression EQ_OP relational_expression | equality_expression NE_OP relational_expression
and_expression : equality_expression | and_expression '&' equality_expression
exclusive_or_expression : and_expression | exclusive_or_expression '^' and_expression
inclusive_or_expression : exclusive_or_expression | inclusive_or_expression '|' exclusive_or_expression
logical_and_expression : inclusive_or_expression | logical_and_expression AND_OP inclusive_or_expression
logical_or_expression : logical_and_expression | logical_or_expression OR_OP logical_and_expression
conditional_expression : logical_or_expression | logical_or_expression '?' expression ':' conditional_expression
assignment_expression : conditional_expression | unary_expression assignment_operator assignment_expression
assignment_operator : '=' | MUL_ASSIGN | DIV_ASSIGN | MOD_ASSIGN | ADD_ASSIGN | SUB_ASSIGN | LEFT_ASSIGN | RIGHT_ASSIGN | AND_ASSIGN | XOR_ASSIGN | OR_ASSIGN
expression : assignment_expression | expression ',' assignment_expression
constant_expression : conditional_expression                    /* with constraints */
declaration : declaration_specifiers ';' | declaration_specifiers init_declarator_list ';' | static_assert_declaration
declaration_specifiers : storage_class_specifier declaration_specifiers | storage_class_specifier | type_specifier declaration_specifiers | type_specifier |
type_qualifier declaration_specifiers | type_qualifier | function_specifier declaration_specifiers | function_specifier | alignment_specifier
declaration_specifiers | alignment_specifier
init_declarator_list : init_declarator | init_declarator_list ',' init_declarator
init_declarator : declarator '=' initializer | declarator
storage_class_specifier : TYPEDEF        /* identifiers must be flagged as TYPEDEF_NAME */ | EXTERN | STATIC | THREAD_LOCAL | AUTO | REGISTER
type_specifier : VOID | CHAR | SHORT | INT | LONG | FLOAT | DOUBLE | SIGNED | UNSIGNED | BOOL | COMPLEX | IMAGINARY
                                         /* non-mandated extension */ | atomic_type_specifier | struct_or_union_specifier | enum_specifier | TYPEDEF_NAME
                                                                          /* after it has been defined as such */
struct_or_union_specifier : struct_or_union '{' struct_declaration_list '}' | struct_or_union IDENTIFIER '{' struct_declaration_list '}' | struct_or_union
IDENTIFIER
struct_or_union : STRUCT | UNION
struct_declaration_list : struct_declaration | struct_declaration_list struct_declaration
struct_declaration : specifier_qualifier_list ';'                       /* for anonymous struct/union */ | specifier_qualifier_list
struct_declarator_list ';' | static_assert_declaration
specifier_qualifier_list : type_specifier specifier_qualifier_list | type_specifier | type_qualifier specifier_qualifier_list | type_qualifier
struct_declarator_list : struct_declarator | struct_declarator_list ',' struct_declarator
struct_declarator : ':' constant_expression | declarator ':' constant_expression | declarator
enum_specifier : ENUM '{' enumerator_list '}' | ENUM '{' enumerator_list ',' '}' | ENUM IDENTIFIER '{' enumerator_list '}' | ENUM IDENTIFIER '{' enumerator_list
',' '}' | ENUM IDENTIFIER
enumerator_list : enumerator | enumerator_list ',' enumerator
enumerator                               /* identifiers must be flagged as ENUMERATION_CONSTANT */ : enumeration_constant '=' constant_expression |
enumeration_constant
atomic_type_specifier : ATOMIC '(' type_name ')'
type_qualifier : CONST | RESTRICT | VOLATILE | ATOMIC
function_specifier : INLINE | NORETURN
alignment_specifier : ALIGNAS '(' type_name ')' | ALIGNAS '(' constant_expression ')'
declarator : pointer direct_declarator | direct_declarator
direct_declarator : IDENTIFIER | '(' declarator ')' | direct_declarator '[' ']' | direct_declarator '[' '*' ']' | direct_declarator '[' STATIC
type_qualifier_list assignment_expression ']' | direct_declarator '[' STATIC assignment_expression ']' | direct_declarator '[' type_qualifier_list '*' ']' |
direct_declarator '[' type_qualifier_list STATIC assignment_expression ']' | direct_declarator '[' type_qualifier_list assignment_expression ']' |
direct_declarator '[' type_qualifier_list ']' | direct_declarator '[' assignment_expression ']' | direct_declarator '(' parameter_type_list ')' |
direct_declarator '(' ')' | direct_declarator '(' identifier_list ')'
pointer : '*' type_qualifier_list pointer | '*' type_qualifier_list | '*' pointer | '*'
type_qualifier_list : type_qualifier | type_qualifier_list type_qualifier
parameter_type_list : parameter_list ',' ELLIPSIS | parameter_list
parameter_list : parameter_declaration | parameter_list ',' parameter_declaration
parameter_declaration : declaration_specifiers declarator | declaration_specifiers abstract_declarator | declaration_specifiers
identifier_list : IDENTIFIER | identifier_list ',' IDENTIFIER
type_name : specifier_qualifier_list abstract_declarator | specifier_qualifier_list
abstract_declarator : pointer direct_abstract_declarator | pointer | direct_abstract_declarator
direct_abstract_declarator : '(' abstract_declarator ')' | '[' ']' | '[' '*' ']' | '[' STATIC type_qualifier_list assignment_expression ']' | '[' STATIC
assignment_expression ']' | '[' type_qualifier_list STATIC assignment_expression ']' | '[' type_qualifier_list assignment_expression ']' | '['
type_qualifier_list ']' | '[' assignment_expression ']' | direct_abstract_declarator '[' ']' | direct_abstract_declarator '[' '*' ']' |
direct_abstract_declarator '[' STATIC type_qualifier_list assignment_expression ']' | direct_abstract_declarator '[' STATIC assignment_expression ']' |
direct_abstract_declarator '[' type_qualifier_list assignment_expression ']' | direct_abstract_declarator '[' type_qualifier_list STATIC assignment_expression
']' | direct_abstract_declarator '[' type_qualifier_list ']' | direct_abstract_declarator '[' assignment_expression ']' | '(' ')' | '(' parameter_type_list ')' |
direct_abstract_declarator '(' ')' | direct_abstract_declarator '(' parameter_type_list ')'
initializer : '{' initializer_list '}' | '{' initializer_list ',' '}' | assignment_expression
initializer_list : designation initializer | initializer | initializer_list ',' designation initializer | initializer_list ',' initializer
designation : designator_list '='
designator_list : designator | designator_list designator
designator : '[' constant_expression ']' | '.' IDENTIFIER
static_assert_declaration : STATIC_ASSERT '(' constant_expression ',' STRING_LITERAL ')' ';'
statement : labeled_statement | compound_statement | expression_statement | selection_statement | iteration_statement | jump_statement
labeled_statement : IDENTIFIER ':' statement | CASE constant_expression ':' statement | DEFAULT ':' statement
compound_statement : '{' '}' | '{'  block_item_list '}'
block_item_list : block_item | block_item_list block_item
block_item : declaration | statement
expression_statement : ';' | expression ';'
selection_statement : IF '(' expression ')' statement ELSE statement | IF '(' expression ')' statement | SWITCH '(' expression ')' statement
iteration_statement : WHILE '(' expression ')' statement | DO statement WHILE '(' expression ')' ';' | FOR '(' expression_statement expression_statement ')'
statement | FOR '(' expression_statement expression_statement expression ')' statement | FOR '(' declaration expression_statement ')' statement | FOR '('
declaration expression_statement expression ')' statement
jump_statement : GOTO IDENTIFIER ';' | CONTINUE ';' | BREAK ';' | RETURN ';' | RETURN expression ';'
translation_unit : external_declaration | translation_unit external_declaration
external_declaration : function_definition | declaration
function_definition : declaration_specifiers declarator declaration_list compound_statement | declaration_specifiers declarator compound_statement
declaration_list : declaration | declaration_list declaration
```

C (link)

```
grammar Clojure;
file_ : form* EOF ;
form : literal | list_ | vector | map_ | reader_macro ;
forms : form* ;
list_ : '(' forms ')' ;
vector : '[' forms ']' ;
map_ : '{' (form form)* '}' ;
set_ : '#{' forms '}' ;
reader_macro : lambda_ | meta_data | regex | var_quote | host_expr | set_
    | tag | discard | dispatch | deref | quote
    | backtick | unquote | unquote_splicing | gensym ;
quote : '\'' form ;
backtick : '`' form ;
unquote : '~' form ;
unquote_splicing : '~@' form ;
tag : '^' form form ;
deref : '@' form ;
gensym : SYMBOL '#' ;
lambda_ : '#(' form* ')' ;
meta_data : '#^' (map_ form | form) ;
var_quote : '#\'' symbol ;
host_expr : '#+' form form ;
discard : '#_' form ;
dispatch : '#' symbol form ;
regex : '#' string_ ;
literal : string_ | number | character | nil_
    | BOOLEAN | keyword | symbol | param_name ;
string_ : STRING ;
hex_ : HEX ;
bin_ : BIN ;
bign : BIGN ;
number : FLOAT
    | hex_ | bin_ | bign | LONG ;
character : named_char | u_hex_quad | any_char ;
named_char : CHAR_NAMED ;
any_char : CHAR_ANY ;
u_hex_quad : CHAR_U ;
nil_ : NIL ;
keyword : macro_keyword | simple_keyword ;
simple_keyword : ':' symbol ;
macro_keyword : ':' ':' symbol ;
symbol : ns_symbol | simple_sym ;
simple_sym : SYMBOL ;
ns_symbol : NS_SYMBOL ;
param_name : PARAM_NAME ;
```

Clojure (link)

High Level Picture



```
      -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
     | NamedExpr(expr target, expr value)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
```

```
lambda a, b:  a < b
```

```
Expr(
  value=Lambda(
    args=arguments(
      args=[
        arg(arg='a'),
        arg(arg='b')]),
    body=Compare(
      left=Name(id='a'),
      ops=[Lt()],
      comparators=[
        Name(id='b')
      ])
  ))
```

Modeling Language

```
lambda a, b:  a < b
```

Abstract Syntax Tree

```
Expr(
  value=Lambda(
    args=arguments(
      args=[
        arg(arg='a’),
        arg(arg='b')]),
    body=Compare(
      left=Name(id='a’),
      ops=[Lt()],
      comparators=[
        Name(id='b’)
      ])
  ))
```

Modeling Language

```
lambda a, b:  a < b
```

Abstract Syntax Tree

```
Expr(
  value=Lambda(
    args=arguments(
      args=[
        arg(arg='a’),
        arg(arg='b')]),
    body=Compare(
      left=Name(id='a’),
      ops=[Lt()],
      comparators=[
        Name(id='b’)
      ])
  ))
```

Tokens (bert-base-uncased)

```
Tokens: [
  '[CLS]’,
  'lambda’,
  'a’,
  ',’,
  'b’,
  ':’,
  'a’,
  '<‘,
  'b’,
  '[SEP]’
]
```

Modeling Language

```
lambda a, b:  a < b
```

Abstract Syntax Tree

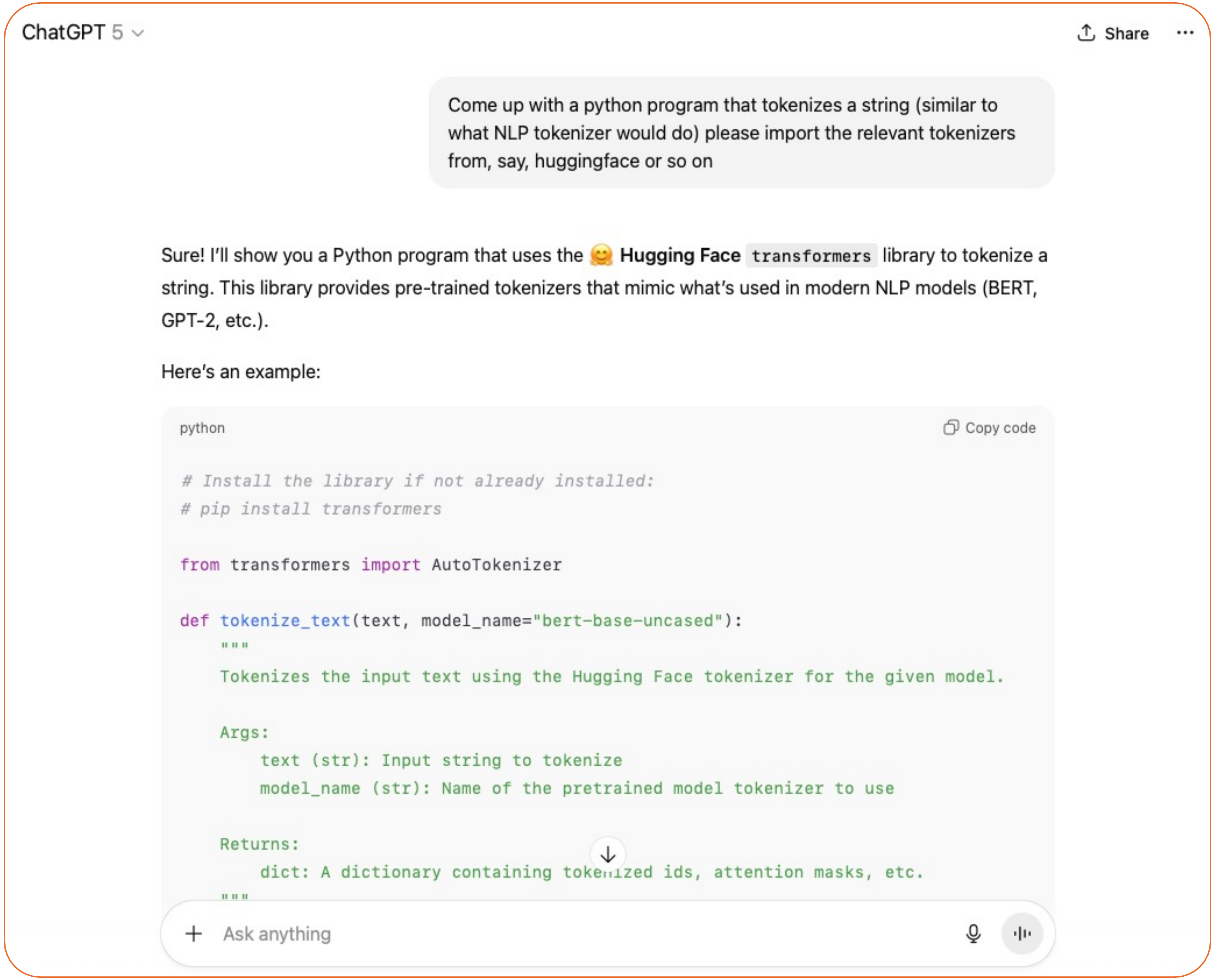
```
Expr(
  value=Lambda(
    args=arguments(
      args=[
        arg(arg='a'),
        arg(arg='b')]),
    body=Compare(
      left=Name(id='a'),
      ops=[Lt()],
      comparators=[
        Name(id='b')
      ])
  ))
```

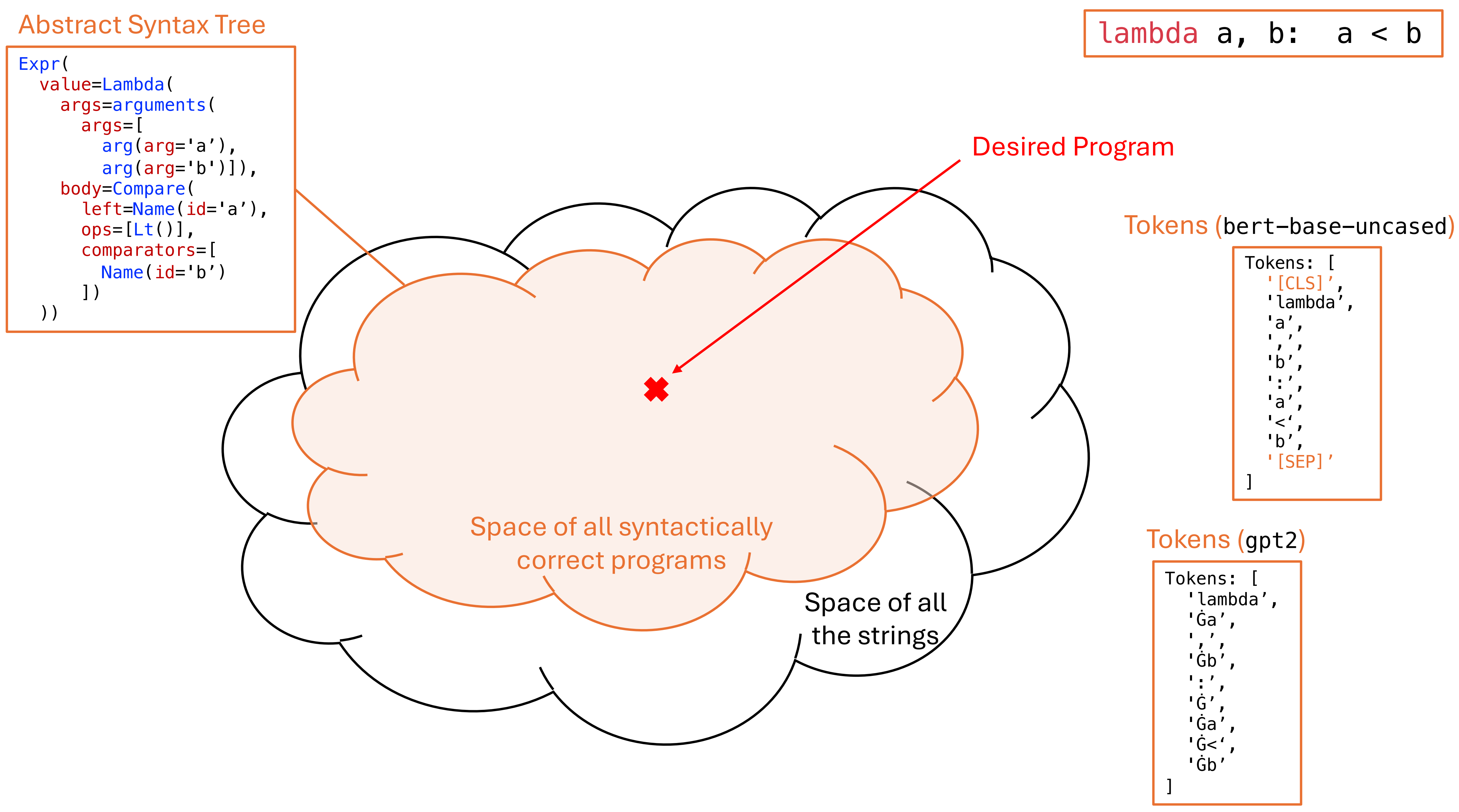
Tokens (bert-base-uncased)

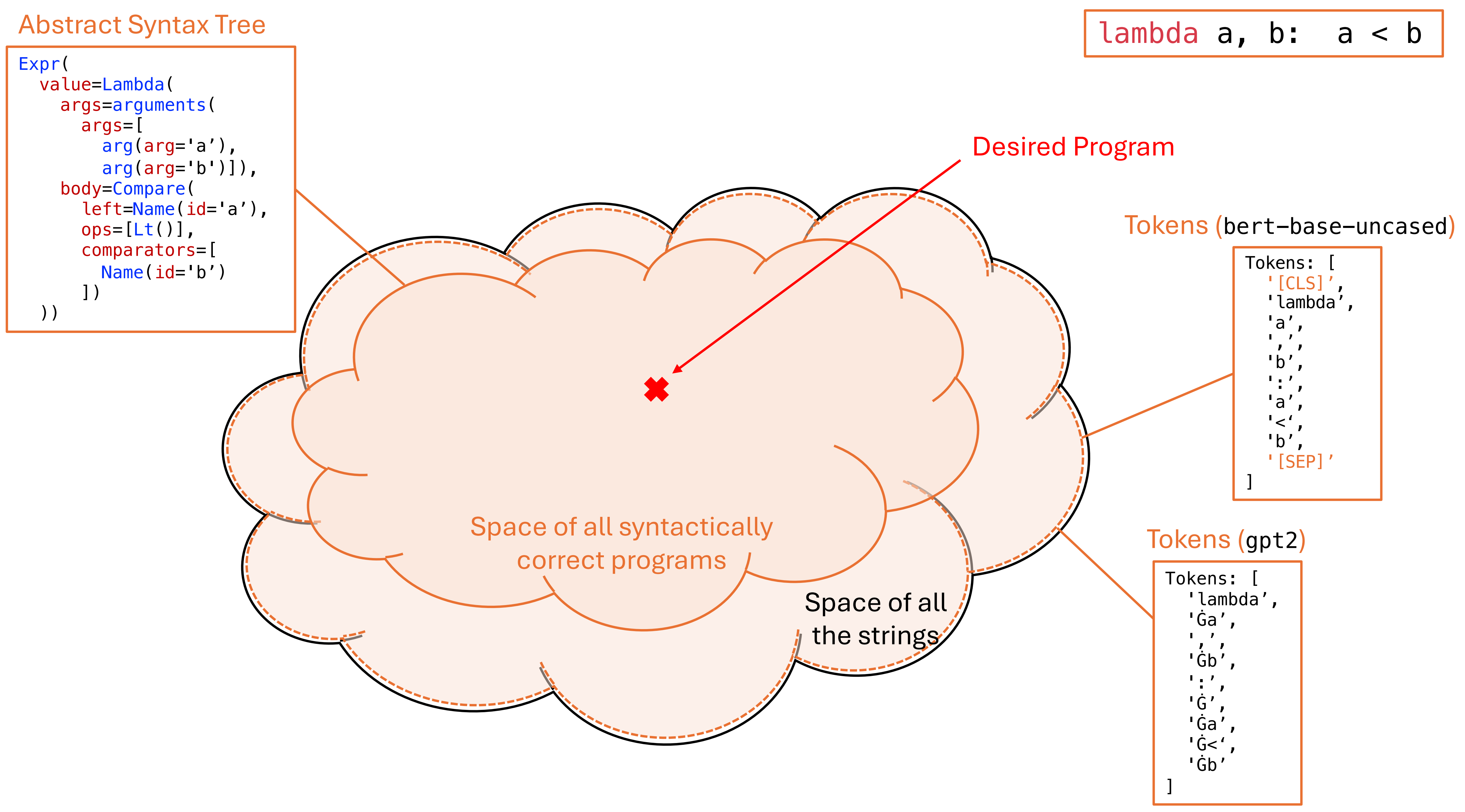
```
Tokens: [
  '[CLS]',
  'lambda',
  'a',
  ',',
  'b',
  ':',
  'a',
  '<',
  'b',
  '[SEP]'
]
```

Tokens (gpt2)

```
Tokens: [
  'lambda',
  'Ġa',
  ',',
  'Ġb',
  ':',
  'Ġ',
  'Ġa',
  'Ġ<',
  'Ġb'
]
```


ChatGPT 5
Share
Come up with a python program that tokenizes a string (similar to what NLP tokenizer would do) please import the relevant tokenizers from, say, huggingface or so on
Sure! I'll show you a Python program that uses the 🤗 Hugging Face transformers library to tokenize a string. This library provides pre-trained tokenizers that mimic what's used in modern NLP models (BERT, GPT-2, etc.).
Here's an example:
python
Copy code
# Install the library if not already installed:
# pip install transformers
from transformers import AutoTokenizer
def tokenize_text(text, model_name="bert-base-uncased"):
"""
Tokenizes the input text using the Hugging Face tokenizer for the given model.
Args:
text (str): Input string to tokenize
model_name (str): Name of the pretrained model tokenizer to use
Returns:
dict: A dictionary containing tokenized ids, attention masks, etc.
"""
Ask anything


Abstract Syntax Tree
Expr(
value=Lambda(
args=arguments(
args=[
arg(arg='a'),
arg(arg='b')]),
body=Compare(
left=Name(id='a'),
ops=[Lt()],
comparators=[
Name(id='b')
])
))
lambda a, b: a < b
Desired Program
Tokens (bert-base-uncased)
Tokens: [
'[CLS]',
'lambda',
'a',
',',
'b',
':',
'a',
'<',
'b',
'[SEP]'
]
Tokens (gpt2)
Tokens: [
'lambda',
'Ġa',
',',
'Ġb',
':',
'Ġ',
'Ġa',
'Ġ<',
'Ġb'
]
Space of all syntactically correct programs
Space of all the strings



Abstract Syntax Tree
Expr(
value=Lambda(
args=arguments(
args=[
arg(arg='a'),
arg(arg='b')]),
body=Compare(
left=Name(id='a'),
ops=[Lt()],
comparators=[
Name(id='b')
])
))
lambda a, b: a < b
Desired Program
Tokens (bert-base-uncased)
Tokens: [
'[CLS]',
'lambda',
'a',
',',
'b',
':',
'a',
'<',
'b',
'[SEP]'
]
Tokens (gpt2)
Tokens: [
'lambda',
'Ġa',
',',
'Ġb',
':',
'Ġ',
'Ġa',
'Ġ<',
'Ġb'
]
Space of all syntactically correct programs
Space of all the strings

Modeling Language

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \ldots, t_{i-1}}$

Output: Next Token $\boxed{t_i}$

Modeling Language

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \dots, t_{i-1}}$

Output: Next Token $\boxed{t_i}$

Problem Example: **Auto-completion**

Input: Prefix `for i in ???`

Output: Next Token `range`

Modeling Language

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \ldots, t_{i-1}}$

Output: Next Token $\boxed{t_i}$

Problem Example: **Auto-completion**

Input: Prefix `for i in ???`

Output: Next Token `range` (t_5)

After tokenizing by GPT-2 tokenizer

`['for', 'Ġi', 'Ġin', 'Ġ']`

t_1 t_2 t_3 t_4

Modeling Language

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \ldots, t_{i-1}}$

Output: Next Token $\boxed{t_i}$

Problem Example: **Auto-completion**

Input: Prefix `['for', 'Ġi', 'Ġin', 'Ġ']`

t_1 t_2 t_3 t_4

Output: Next Token `range`

t_5

Modeling Language

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \ldots, t_{i-1}}$ Output: Next Token $\boxed{t_i}$

Goal: Compute the probability $\Pr(t_i \mid t_1, t_2, \ldots, t_{i-1})$

Problem Example: **Auto-completion**

Input: Prefix `['for', 'Ġi', 'Ġin', 'Ġ']` Output: Next Token `range`

t_1 t_2 t_3 t_4 t_5

Goal: Compute the probability $\Pr(\text{range} \mid \text{for, Ġi, Ġin, Ġ})$

Modeling Language

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \ldots, t_{i-1}}$

Output: Next Token $\boxed{t_i}$

Goal: Compute the probability $\Pr(t_i \mid t_1, t_2, \ldots, t_{i-1})$

N-Gram Models → Recurrent Models → Seq2Seq & Attention → Transformers

Modeling Language

N-Gram Models

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \dots, t_{i-1}}$

Output: Next Token $\boxed{t_i}$

Goal: Compute the probability $\Pr(t_i \mid t_1, t_2, \dots, t_{i-1})$

Solution: **N-Gram**

Idea: Approximating $\Pr(t_i \mid t_1, t_2, \dots, t_{i-1})$ by only considering the $n-1$ tokens preceding it

$$\Pr(t_i \mid t_1, t_2, \dots, t_{i-1}) \approx \Pr(t_i \mid t_{i-n+1}, \dots, t_{i-1})$$

Modeling Language

N-Gram Models

Solution: **N-Gram**

Idea: Approximating $\Pr(t_i \mid t_1, t_2, \ldots, t_{i-1})$ by only considering the $n-1$ tokens preceding it

$$\Pr(t_i \mid t_1, t_2, \ldots, t_{i-1}) \approx \Pr(t_i \mid t_{i-n+1}, \ldots, t_{i-1})$$

Problem Example: **Auto-completion**

Input: Prefix `for i in ???`

Output: Next Token `range`

Trigram Database

```
['print', '('] → 'i'
['for', 'i'] → 'in'
['i', 'in'] → 'range'
['if', '__'] → 'name'
...
```

→

Search

`['i', 'in']`

→

Occurrence Table

```
['i','in','range'] → 30 occurrences
['i','in','list'] → 2 occurrences
['i','in','arr'] → 4 occurrences
```

→

Probability Estimation

$$\Pr(\text{range}|\text{for}, \text{i}, \text{in}, \_) \approx \Pr(\text{range}|\text{i}, \text{in}) \approx \frac{30}{30+2+4} = 83.33\%$$

Modeling Language

N-Gram Models

Trigram Database

```
['print', '('] → 'i'
['for', 'i'] → 'in'
['i', 'in'] → 'range'
['if', '__'] → 'name'
...
```

→

Search

```
['i', 'in']
```

→

Occurrence Table

```
['i','in','range'] → 30 occurrences
['i','in','list'] → 2 occurrences
['i','in','arr'] → 4 occurrences
```

→

Probability Estimation

$$\Pr(\text{range}|\text{for}, \text{i}, \text{in}, \_)$$
$$\approx \Pr(\text{range}|\text{i}, \text{in})$$
$$\approx \frac{30}{30 + 2 + 4}$$
$$= 83.33\%$$

Modeling Language

N-Gram Models

Trigram Database

```
['print', '('] → 'i'
['for', 'i'] → 'in'
['i', 'in'] → 'range'
['if', '__'] → 'name'
...
```

→

Search

```
['i', 'in']
```

→

Occurrence Table

```
['i','in','range'] → 30 occurrences
['i','in','list'] → 2 occurrences
['i','in','arr'] → 4 occurrences
```

→

Probability Estimation

$$\Pr(\text{range} \mid \text{for}, \text{i}, \text{in}, \_) \approx \Pr(\text{range} \mid \text{i}, \text{in}) \approx \frac{30}{30+2+4} = 83.33\%$$

Trigram database = set of 3-tuples of consecutive tokens.
can be built from a dataset of programs

Modeling Language

N-Gram Models

Trigram Database

```
['print', '('] → 'i'
['for', 'i'] → 'in'
['i', 'in'] → 'range'
['if', '__'] → 'name'
...
```

→

Search

```
['i', 'in']
```

→

Occurrence Table

```
['i','in','range'] → 30 occurrences
['i','in','list'] → 2 occurrences
['i','in','arr'] → 4 occurrences
```

→

Probability Estimation

$$\Pr(\text{range}|\text{for}, \text{i}, \text{in}, \_) \approx \Pr(\text{range}|\text{i}, \text{in}) \approx \frac{30}{30+2+4} = 83.33\%$$

Take the preceding $n-1$ tokens from the input
e.g. `[for, i, in]` → `[i, in]`

Modeling Language

N-Gram Models

Trigram Database

```
['print', '('] → 'i'
['for', 'i'] → 'in'
['i', 'in'] → 'range'
['if', '__'] → 'name'
...
```

→

Search

```
['i', 'in']
```

→

Occurrence Table

```
['i','in','range'] → 30 occurrences
['i','in','list'] → 2 occurrences
['i','in','arr'] → 4 occurrences
```

Find the occurrences of 3-tuples given the first 2 tokens

→

Probability Estimation

$$\Pr(\text{range}|\text{for}, \text{i}, \text{in}, \_) \approx \Pr(\text{range}|\text{i}, \text{in}) \approx \frac{30}{30+2+4} = 83.33\%$$

Modeling Language

N-Gram Models

Trigram Database

```
['print', '('] → 'i'
['for', 'i'] → 'in'
['i', 'in'] → 'range'
['if', '__'] → 'name'
...
```

→

Search

```
['i', 'in']
```

→

Occurrence Table

```
['i','in','range'] → 30 occurrences
['i','in','list'] → 2 occurrences
['i','in','arr'] → 4 occurrences
```

→

Probability Estimation

$$\Pr(\text{range}|\text{for}, \text{i}, \text{in}, \_) \approx \Pr(\text{range}|\text{i}, \text{in}) \approx \frac{30}{30+2+4} = 83.33\%$$

Apply Maximum Likelihood Estimation (MLE)

$$\Pr(\text{range}|\text{i}, \text{in}) = \frac{\text{Count}(\text{i}, \text{in}, \text{range})}{\sum_w \text{Count}(\text{i}, \text{in}, w)}$$

On the Naturalness of Software

Abram Hindle, Earl Barr, Mark Gabel, Zhendong Su, Prem Devanbu
devanbu@cs.ucdavis.edu

of NLP is this: *natural language may be complex and admit a great wealth of expression, but what people write and say is largely regular and predictable.*

Our central hypothesis is that the same argument applies to software:

> *Programming languages, in theory, are complex, flexible and powerful, but the programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks.*

These models are estimated from a corpus using simple maximum-likelihood based frequency-counting of token sequences. Thus, if "$*$" is a wildcard, we ask, how relatively often are the tokens a_1, a_2, a_3 followed by a_4:

$$p(a_4|a_1a_2a_3) = \frac{count(a_1a_2a_3a_4)}{count(a_1a_2a_3*)}$$

In practice, estimation of n-gram models is quite a bit more complicated. The main difficulties arise from data sparsity, *i.e.*, the richness of the model in comparison to the available data. For example, with 10^4 token vocabulary, a trigram model must estimate 10^{12} coefficients. Some trigrams may never occur in one corpus, but may in fact occur elsewhere. This will

On the Naturalness of Software

Abram Hindle, Earl Barr, Mark Gabel, Zhendong Su, Prem Devanbu
devanbu@cs.ucdavis.edu

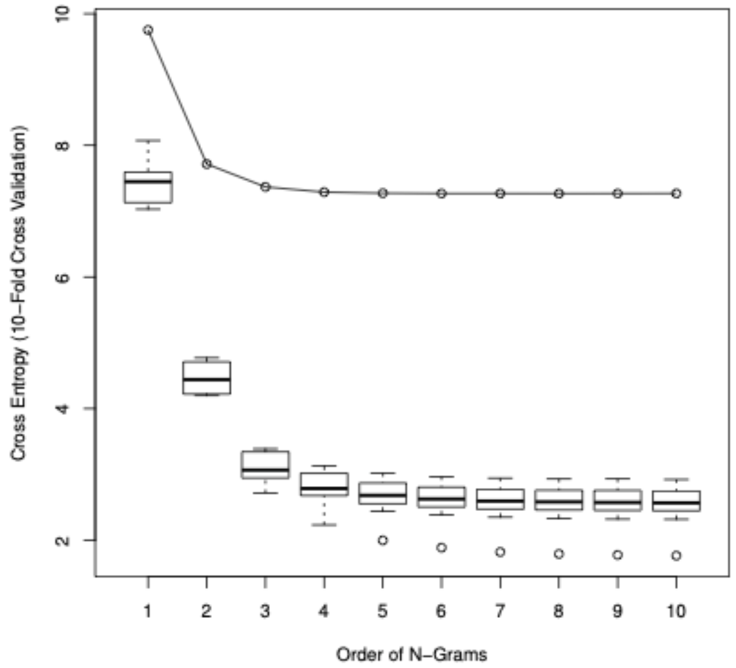


Figure 1: Comparison of English Cross-Entropy versus the Code Cross Entropy of 10 projects.

results are in Figure 1. The single line above is the average over the 10 folds for the English corpus, beginning at about 10 bits for unigram models, and trailing down to under 8 bits for *10*-gram models. The average self cross-entropy for the 10 projects are shown below in boxplots, one for each order from unigram models to 10-gram models. Several observations can be made. First, software unigram entropy is much lower than might be expected from a uniform distribution over unique tokens, because token frequencies are obviously very skewed.

Second, *cross-entropy declines rapidly with n-gram order*, saturating around tri- or 4-grams. The similarity in the decline in English and the Java projects is striking. This decline suggests that there is as much of *repetitive local context that is being captured by the language model* in Java programs, as it is in the English corpora. We take this to be highly encouraging: the ability to model the regularity of the local context in natural languages has proven to be extremely valuable in statistical natural language processing; we hope (and in fact,

On the Naturalness of Software

Abram Hindle, Earl Barr, Mark Gabel, Zhendong Su, Prem Devanbu
devanbu@cs.ucdavis.edu

IV. SUGGESTING THE NEXT TOKEN

The strikingly low entropy (between 3 and 4 bits) produced by the smoothed n-gram model indicates that even at the local token-sequence level, there is a high degree of "naturalness". If we make 8-16 guesses ($2^3 - 2^4$) as to what the next token is, we may very well guess the right one!

Eclipse Suggestion Plug-in We built an Eclipse plug-in to test this idea. Eclipse, like many other IDEs, has a built-in *suggestion* engine that suggests a next token whenever it can. Eclipse (and other IDEs) suggestions are typically based on type information available in context. We conjectured that corpus-based n-gram models suggestion engine (for brevity, $\mathcal{NGSE}$) could enhance eclipse's built-in suggestion engine (for brevity, $\mathcal{ECSE}$) by offering tokens that tend to *naturally* follow from preceding ones in the relevant corpus.

Modeling Language

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \dots, t_{i-1}}$ Output: Next Token $\boxed{t_i}$

Goal: Compute the probability $\Pr(t_i \mid t_1, t_2, \dots, t_{i-1})$

N-Gram Models → Recurrent Models → Seq2Seq & Attention → Transformers

Modeling Language

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \ldots, t_{i-1}}$ Output: Next Token $\boxed{t_i}$

Goal: Compute the probability $\Pr(t_i \mid t_1, t_2, \ldots, t_{i-1})$

N-Gram Models → Recurrent Models → Seq2Seq & Attention → Transformers

Modeling Language

Recurrent Models

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \dots, t_{i-1}}$

Output: Next Token $\boxed{t_i}$

Goal: Compute the probability $\Pr(t_i \mid t_1, t_2, \dots, t_{i-1})$

Modeling Language

Recurrent Models

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \dots, t_{i-1}}$

Output: Next Token $\boxed{t_i}$

Goal: Compute the probability $\Pr(t_i \mid t_1, t_2, \dots, t_{i-1})$

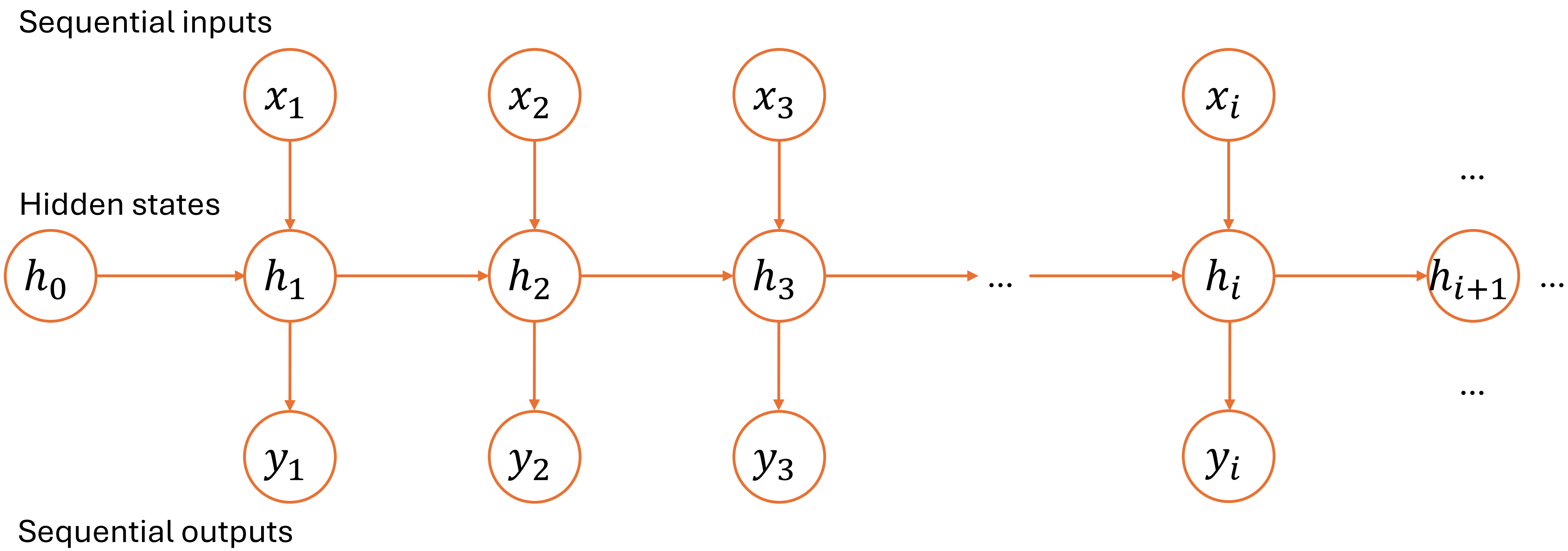
Problems with N-Grams:
- Can't remember beyond a fixed window
- Data sparsity (unseen n-grams)
- Ignores semantic structures

➔ **Recurrent Models**

Motivation:
- A model that can remember context of arbitrary length
- The context can be updated dynamically

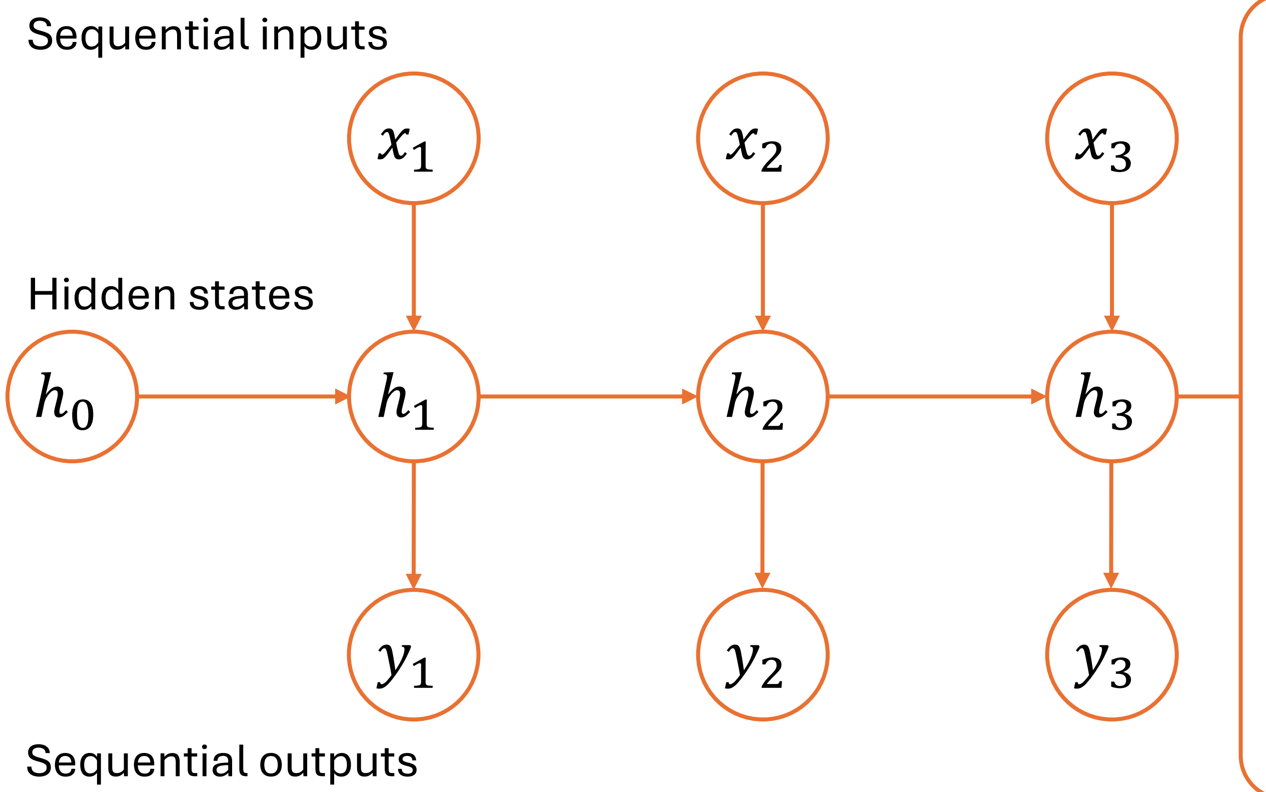
Modeling Language

Recurrent Models



Modeling Language

Recurrent Models

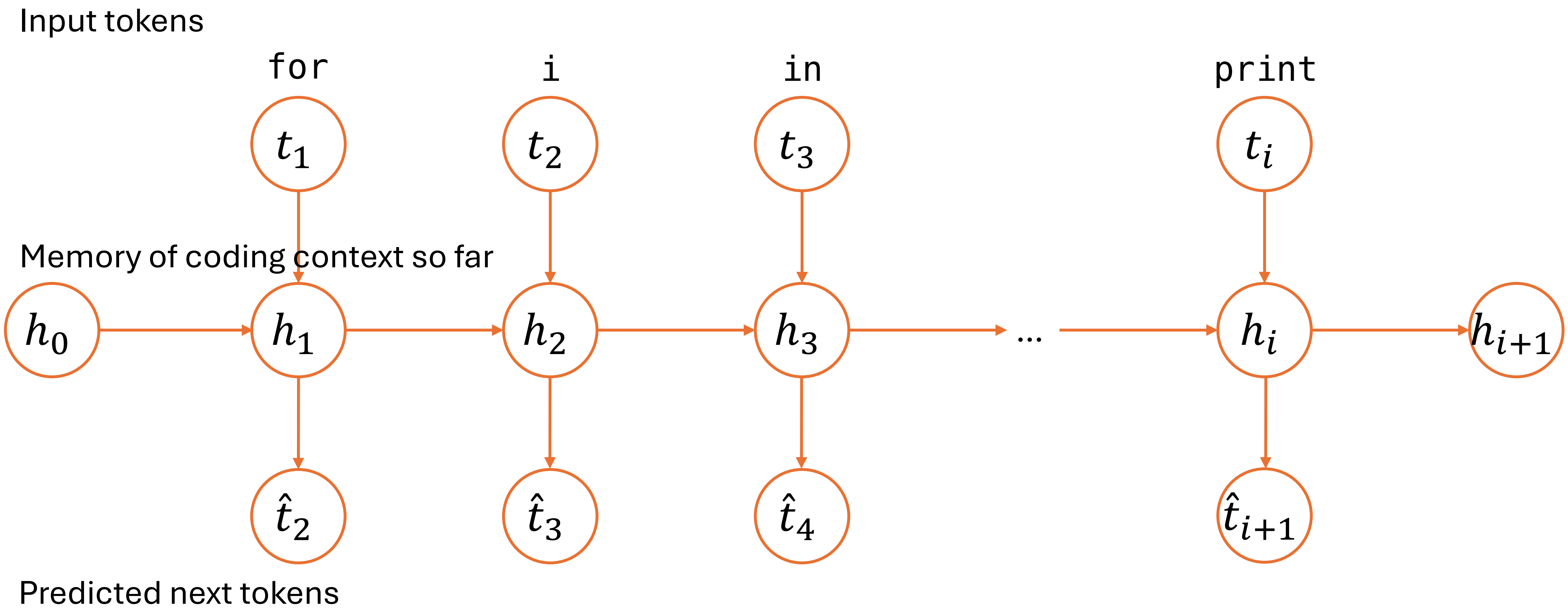


Transition function: $h_i = f_{\theta_1}(x_i, h_{i-1})$

Prediction function: $y_i = g_{\theta_2}(h_i)$

Modeling Language

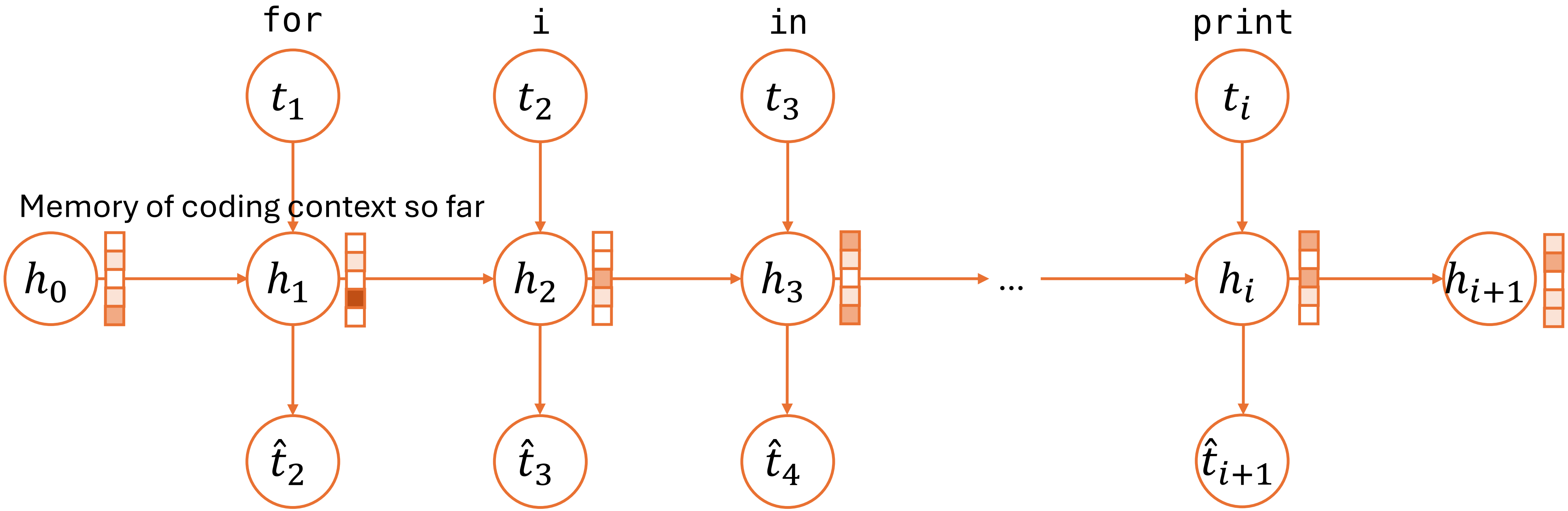
Recurrent Models



Modeling Language

Recurrent Models

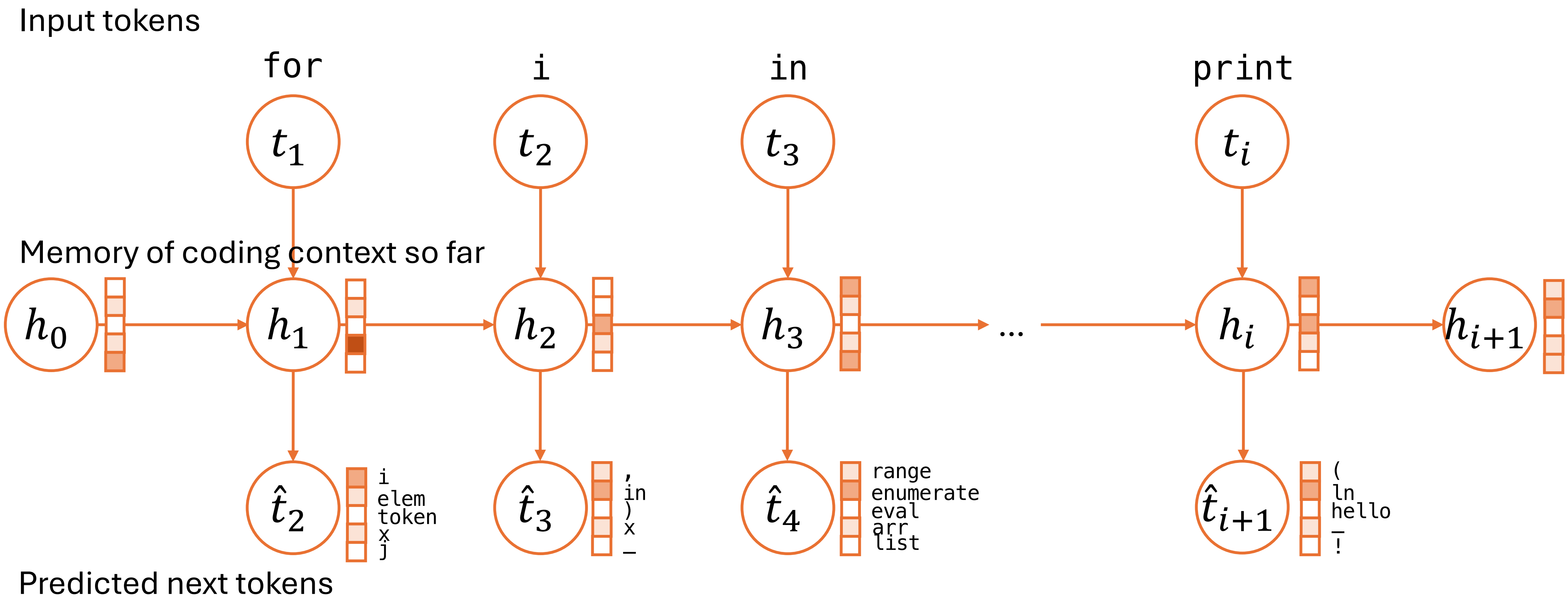
Input tokens



Predicted next tokens

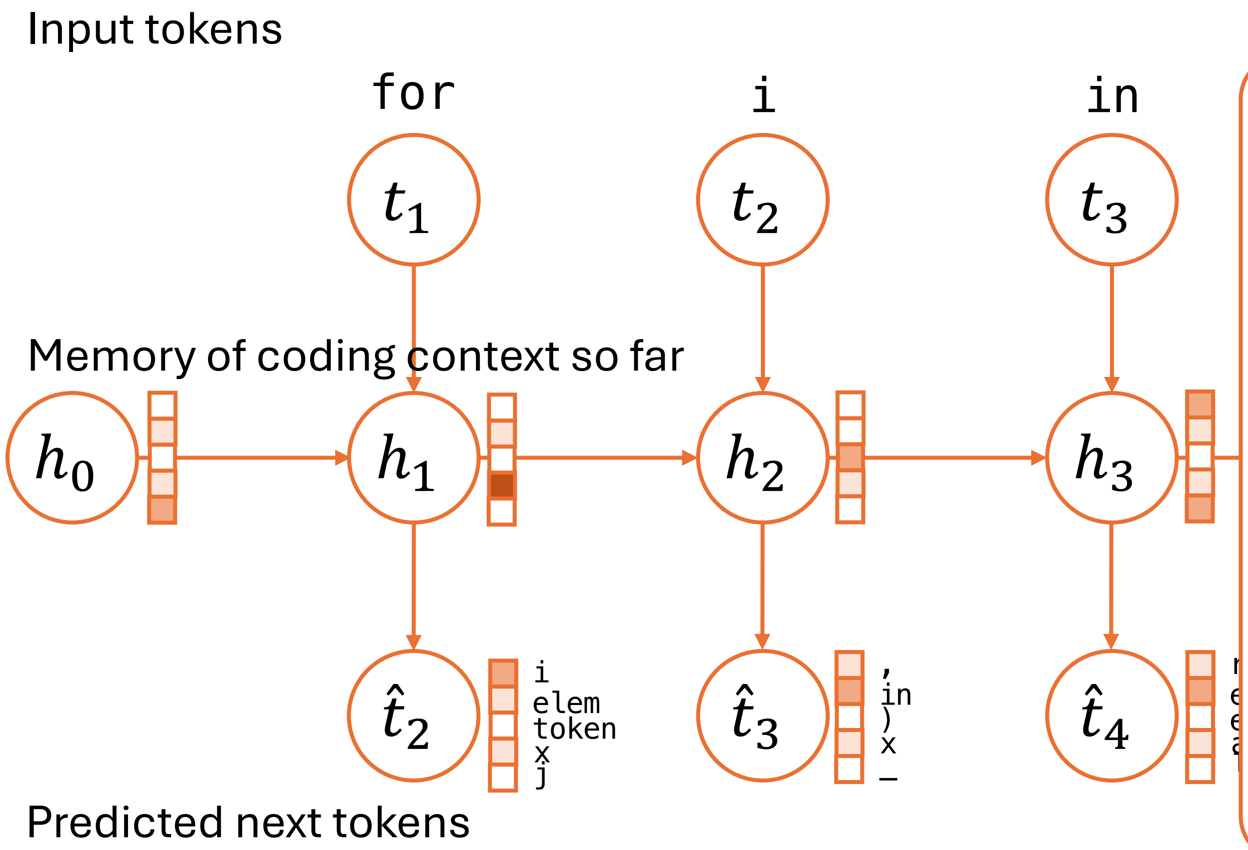
Modeling Language

Recurrent Models



Modeling Language

Recurrent Models



Memory/context encoder:

$$h_i = f_{\theta_1}(t_i, h_{i-1})$$

Next token distribution:

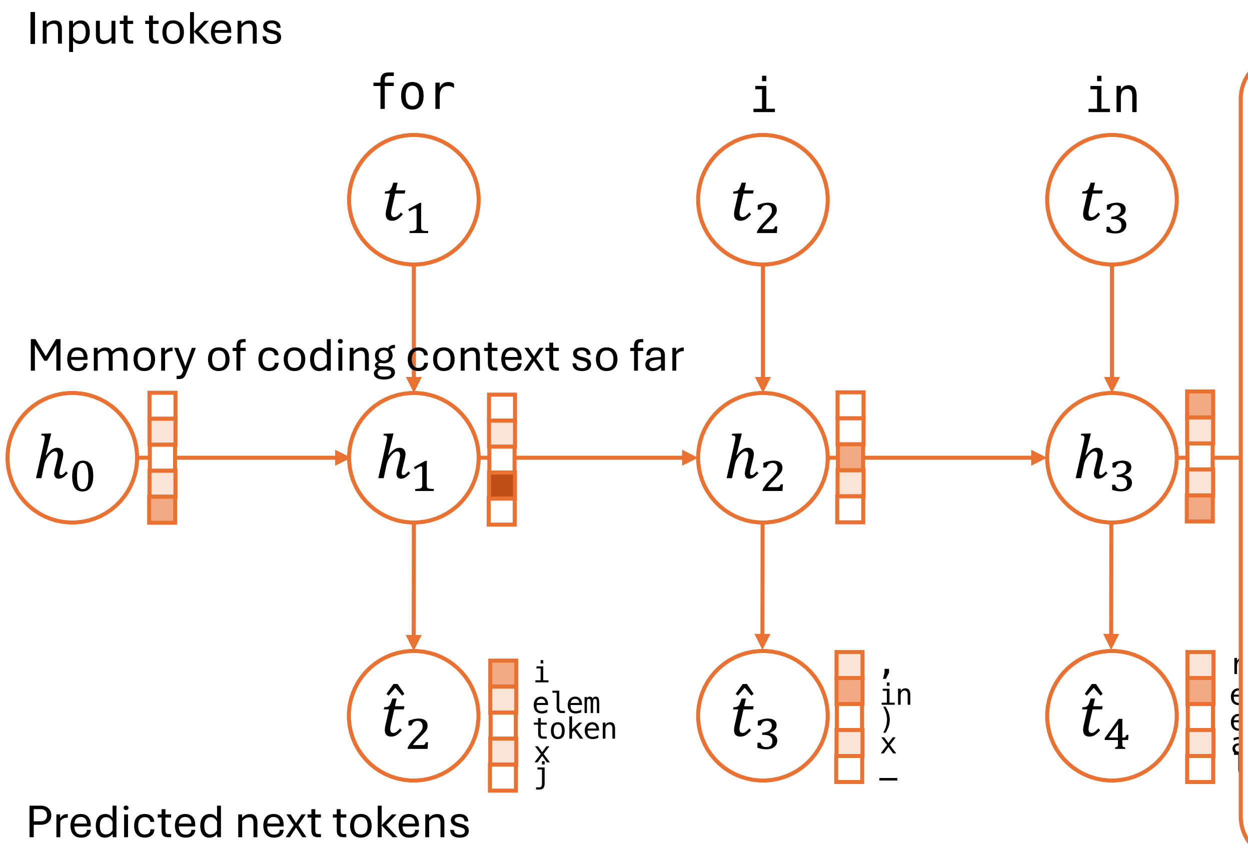
$$\hat{t}_{i+1} = \Pr(t_{i+1} | t_1, \dots, t_i) = g_{\theta_2}(h_i)$$

Draw the predicted next token:

$$t_{i+1} \sim \Pr(\cdot \, | t_1, \dots, t_i)$$

Modeling Language

Recurrent Models

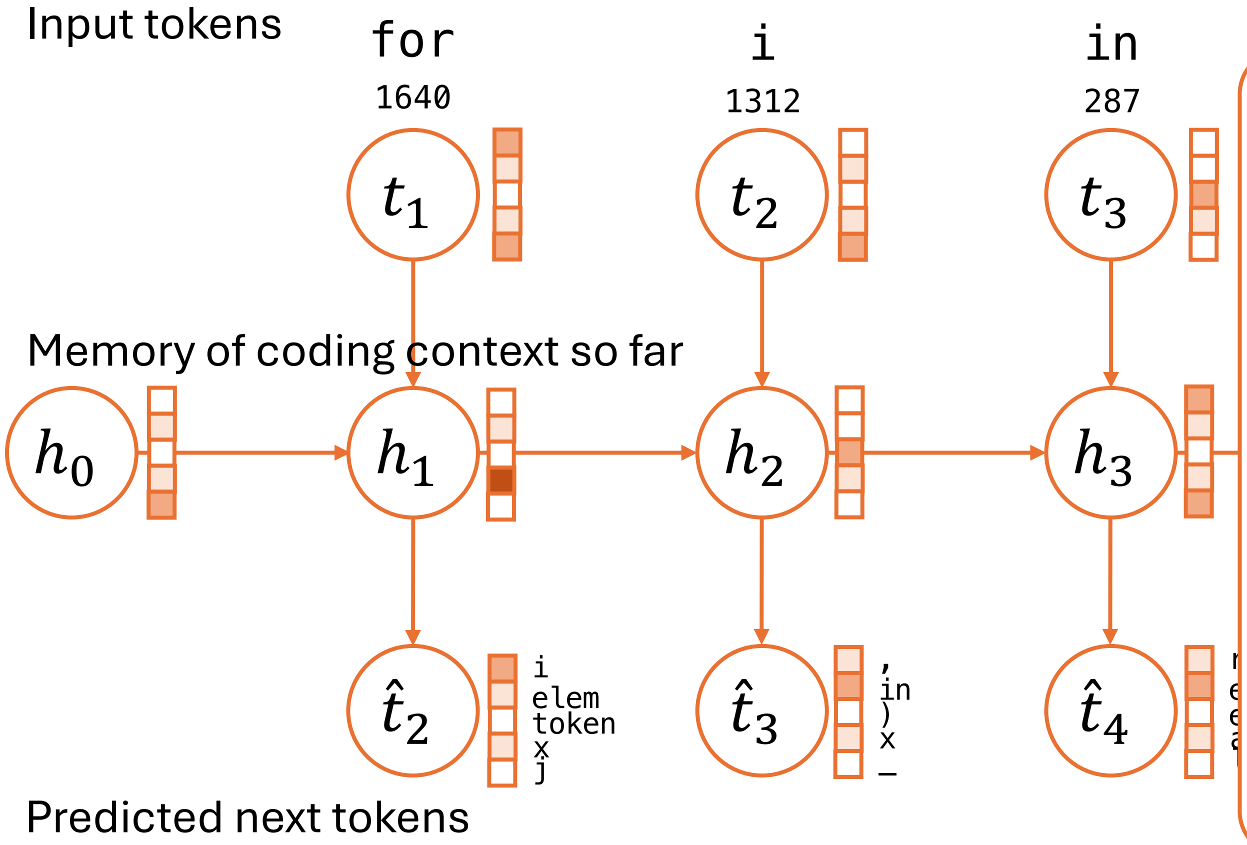



```
text = "for i in range(10):\nprint(i)"
result = tokenize_text(text)
print("Tokens:", result["tokens"])
print("Token IDs:", result["encoded"]["input_ids"])
```

```
Tokens: ['for', 'Ġi', 'Ġin', 'Ġrange',
'(', '10', '):', 'Ċ', 'print', '(',
'i', ')']
Token IDs: tensor([[1640, 1312,  287,
2837,    7,  940, 2599,  198, 4798,
7,   72,    8]])
```

Modeling Language

Recurrent Models




```python
text = "for i in range(10):\nprint(i)"
result = tokenize_text(text)
print("Tokens:", result["tokens"])
print("Token IDs:", result["encoded"]["input_ids"])
```

```
Tokens: ['for', 'Ġi', 'Ġin', 'Ġrange',
'(', '10', '):', 'Ċ', 'print', '(',
'i', ')']
Token IDs: tensor([[1640, 1312,  287,
2837,    7,  940, 2599,  198, 4798,
7,   72,    8]])
```

Toward Deep Learning Software Repositories

Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk
Department of Computer Science
College of William and Mary
Williamsburg, Virginia 23187–8795
Email: {mgwhite, cvendome, mlinarev, denys}@cs.wm.edu

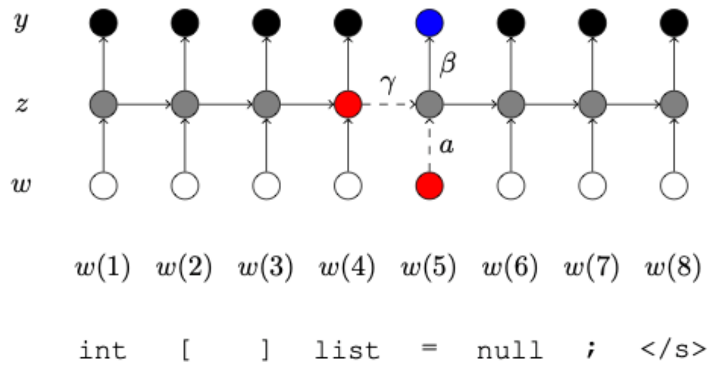


Fig. 1. RNN UNFOLDED IN TIME. The depth of a RNN is evident when the recurrence is unfolded in time. Time steps correspond to software tokens $w(t)$ in a corpus, where $w(0) =$ <s>. Each node in the figure represents a vector of units. White nodes are one-hot token representations; gray nodes are continuous-valued, hidden states; black nodes represent posterior distributions over the vocabulary. Units in the state vectors (gray nodes) compute their activation as a function of the current token and the previous state. Regarding the depth in this notional model, $y(8)$ is a function of $w(8), z(7)$, yet $z(7)$ is a function of $w(7), z(6)$, etc. Hence, predictions are informed by processing data in the past using *multiple levels of nonlinear transformations*.

TABLE IV. TOP-K ACCURACY (%)

Model	Top-1	Top-5	Top-10
Interpolated 8-gram	49.7	71.3	78.1
Interpolated 8-gram 100-cache	4.8	69.5	78.5
RNNs-(400, 5)-1	61.1	78.4	81.4
RNNd-(300, 20)-5	72.2	88.4	92.0

Toward Deep Learning Software Repositories, White et. al., 2015

Modeling Language

Recurrent Models

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \dots, t_{i-1}}$

Output: Next Token $\boxed{t_i}$

Goal: Compute the probability $\Pr(t_i \mid t_1, t_2, \dots, t_{i-1})$

Solution: **Recurrent Neural Network** (RNN, LSTM, BiLSTM)

Idea: Compute $\Pr(t_i \mid t_1, t_2, \dots, t_{i-1})$ by employing a recurrent neural network

Memory/context encoder:

$$h_i = f_{\theta_1}(t_i, h_{i-1})$$

Next token distribution:

$$\hat{t}_{i+1} = \Pr(t_{i+1} | t_1, \dots, t_i) = g_{\theta_2}(h_i)$$

Draw the predicted next token:

$$t_{i+1} \sim \Pr(\cdot \,| t_1, \dots, t_i)$$

Modeling Language

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \ldots, t_{i-1}}$

Output: Next Token $\boxed{t_i}$

Goal: Compute the probability $\Pr(t_i \mid t_1, t_2, \ldots, t_{i-1})$

N-Gram Models → Recurrent Models → Seq2Seq & Attention → Transformers

Modeling Language

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \dots, t_{i-1}}$ Output: Next Token $\boxed{t_i}$

Goal: Compute the probability $\Pr(t_i \mid t_1, t_2, \dots, t_{i-1})$

N-Gram Models → Recurrent Models → Seq2Seq & Attention → Transformers

Modeling Language

Seq2Seq & Attention

Problem Definition: **"Sequence-to-Sequence Translation"**

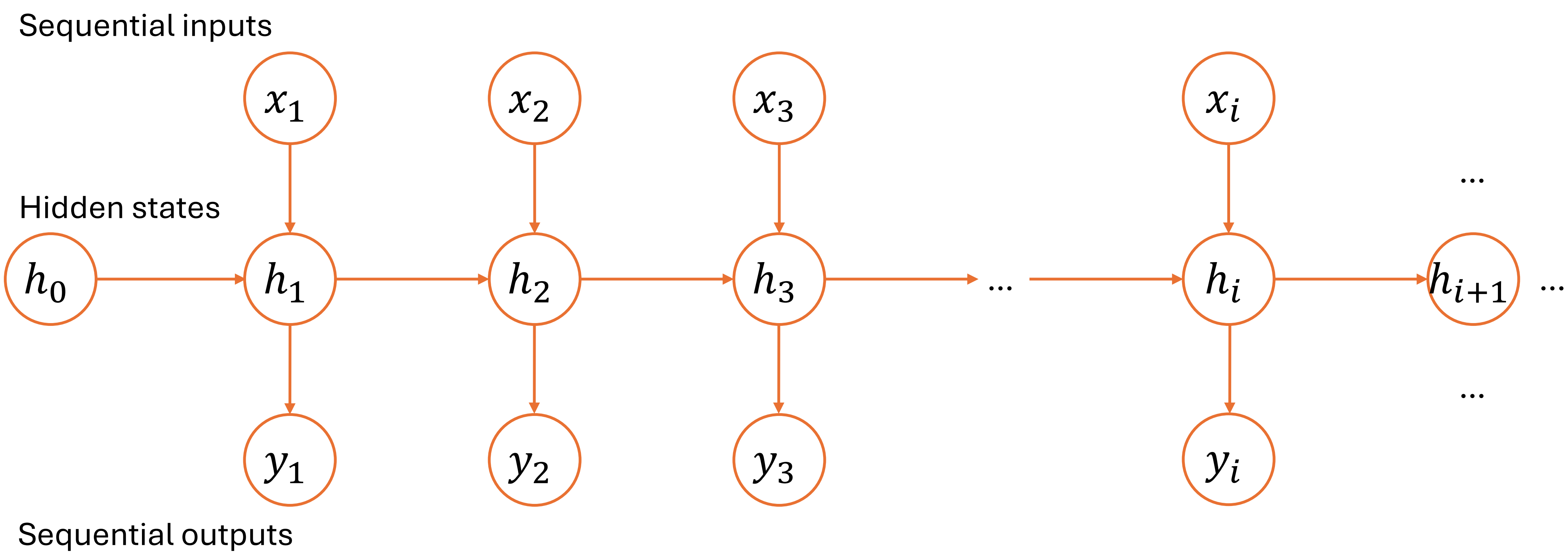
Input: Input Sequence $\boxed{\mathbf{x} = x_1, x_2, \dots, x_n}$ Output: Next Token $\boxed{y_i}$

Input: Generated tokens so far $\boxed{y_1, y_2, \dots, y_{i-1}}$

Goal: Compute the probability $\Pr(y_i \mid y_1, y_2, \dots, y_{i-1}, \mathbf{x})$

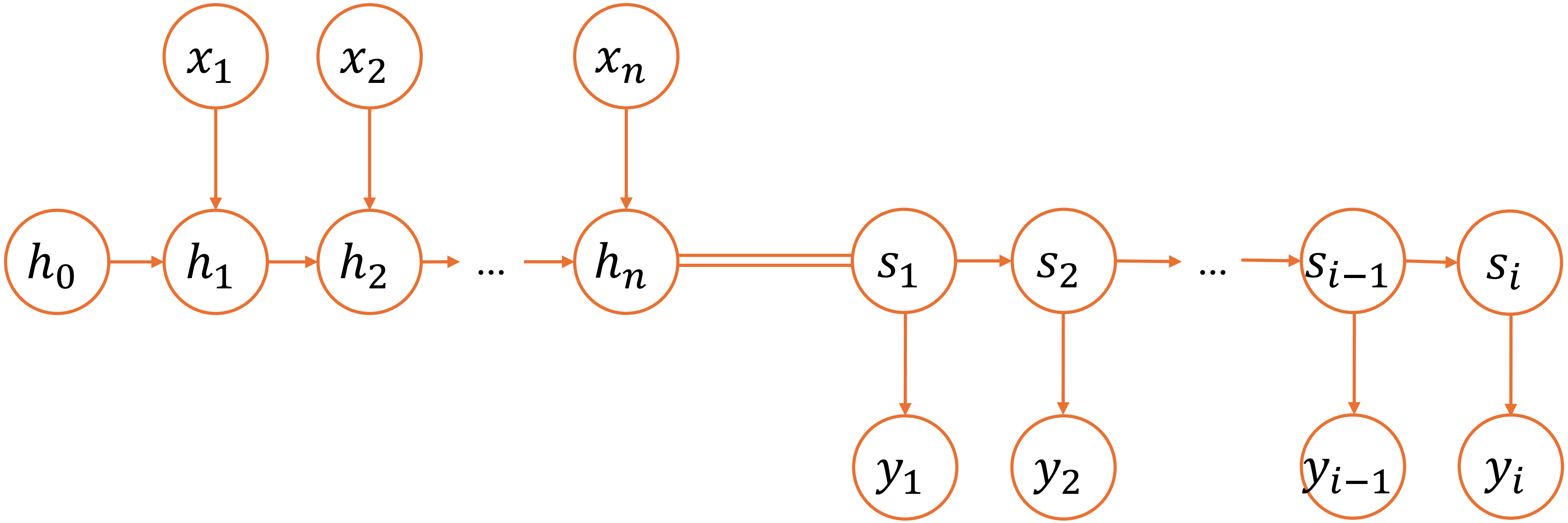
Modeling Language

Recurrent Models



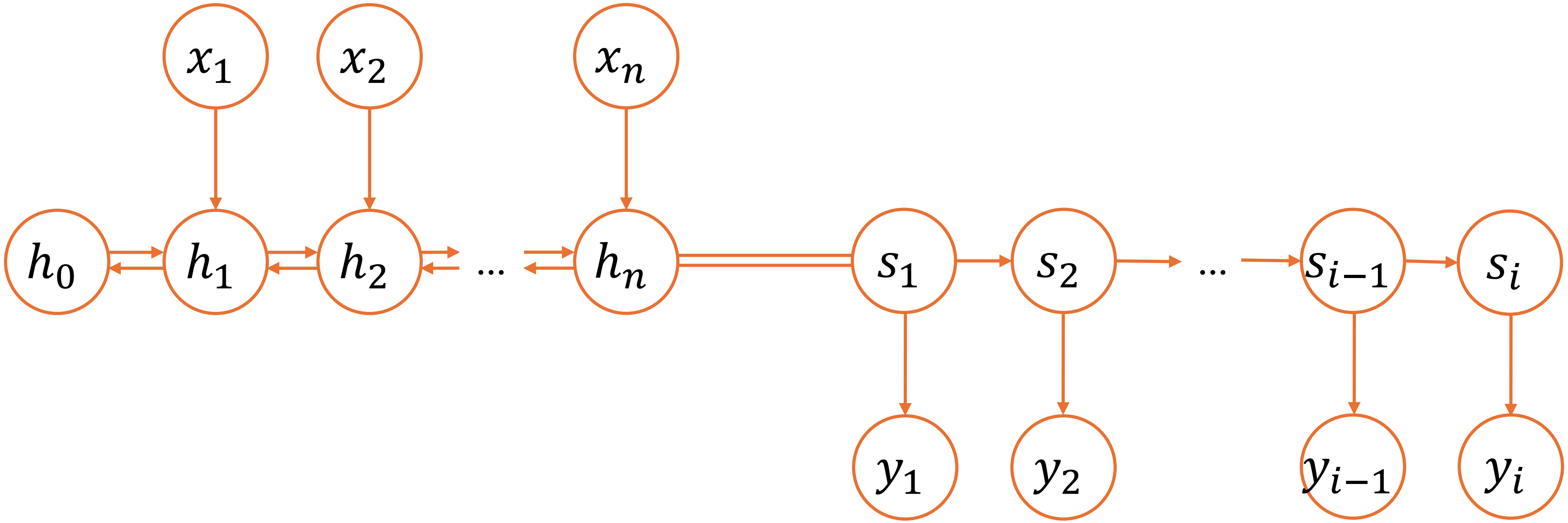
Modeling Language

Seq2Seq Models



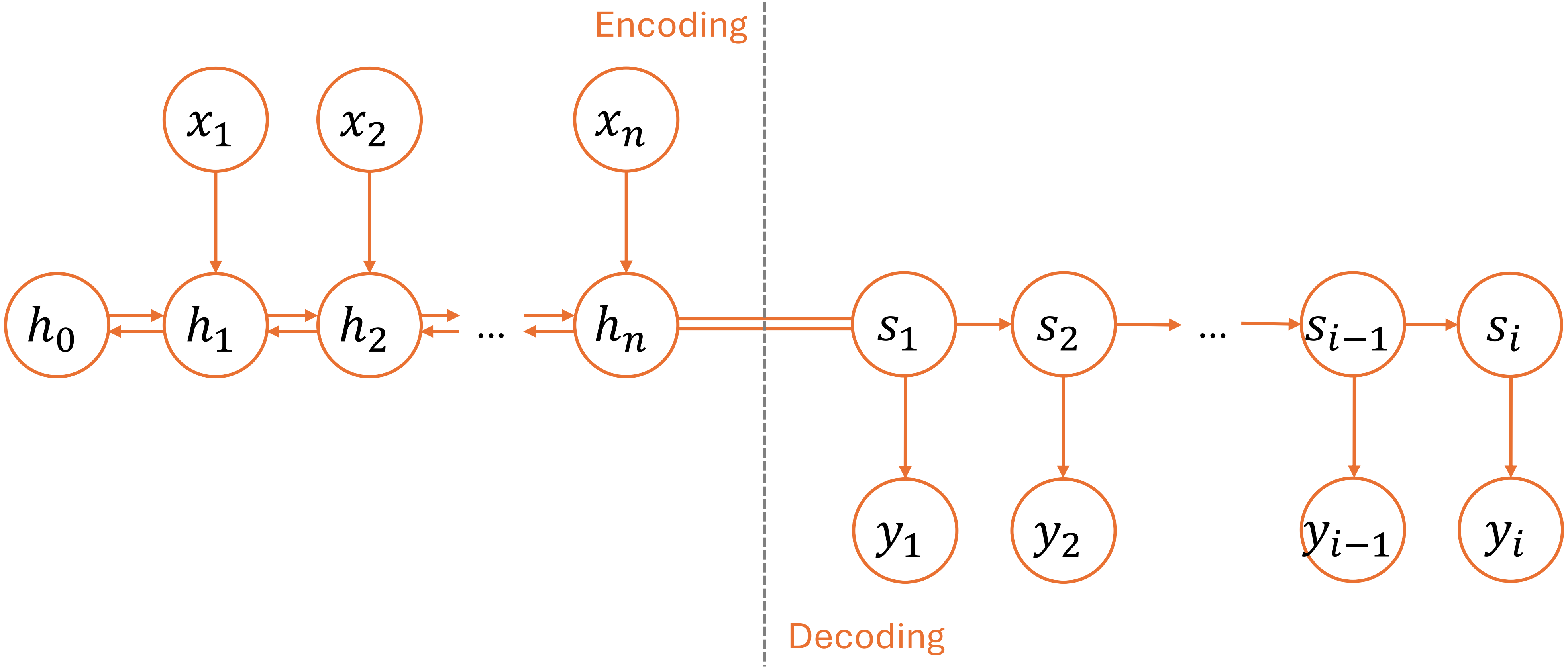
Modeling Language

Seq2Seq Models



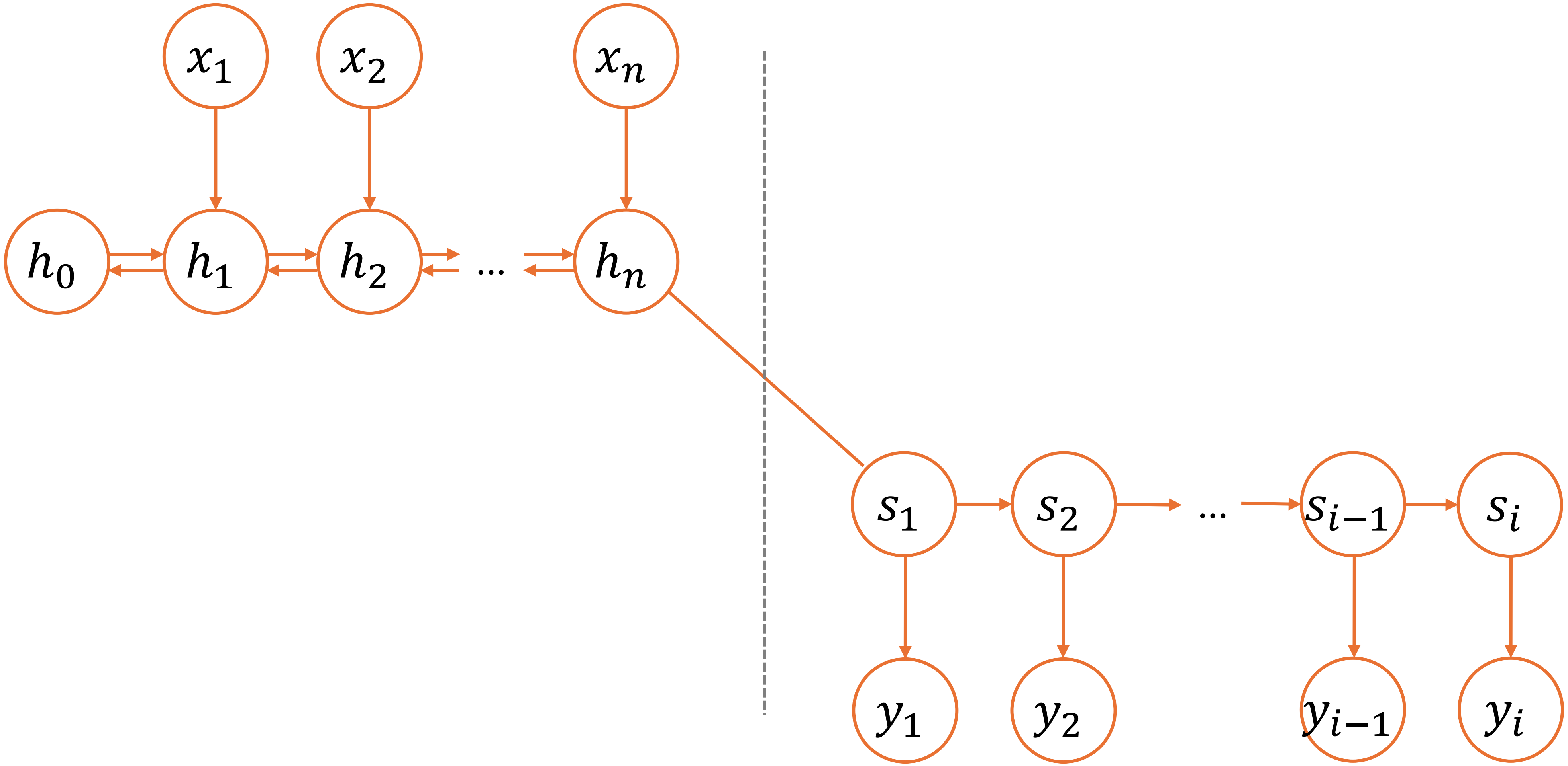
Modeling Language

Seq2Seq Models



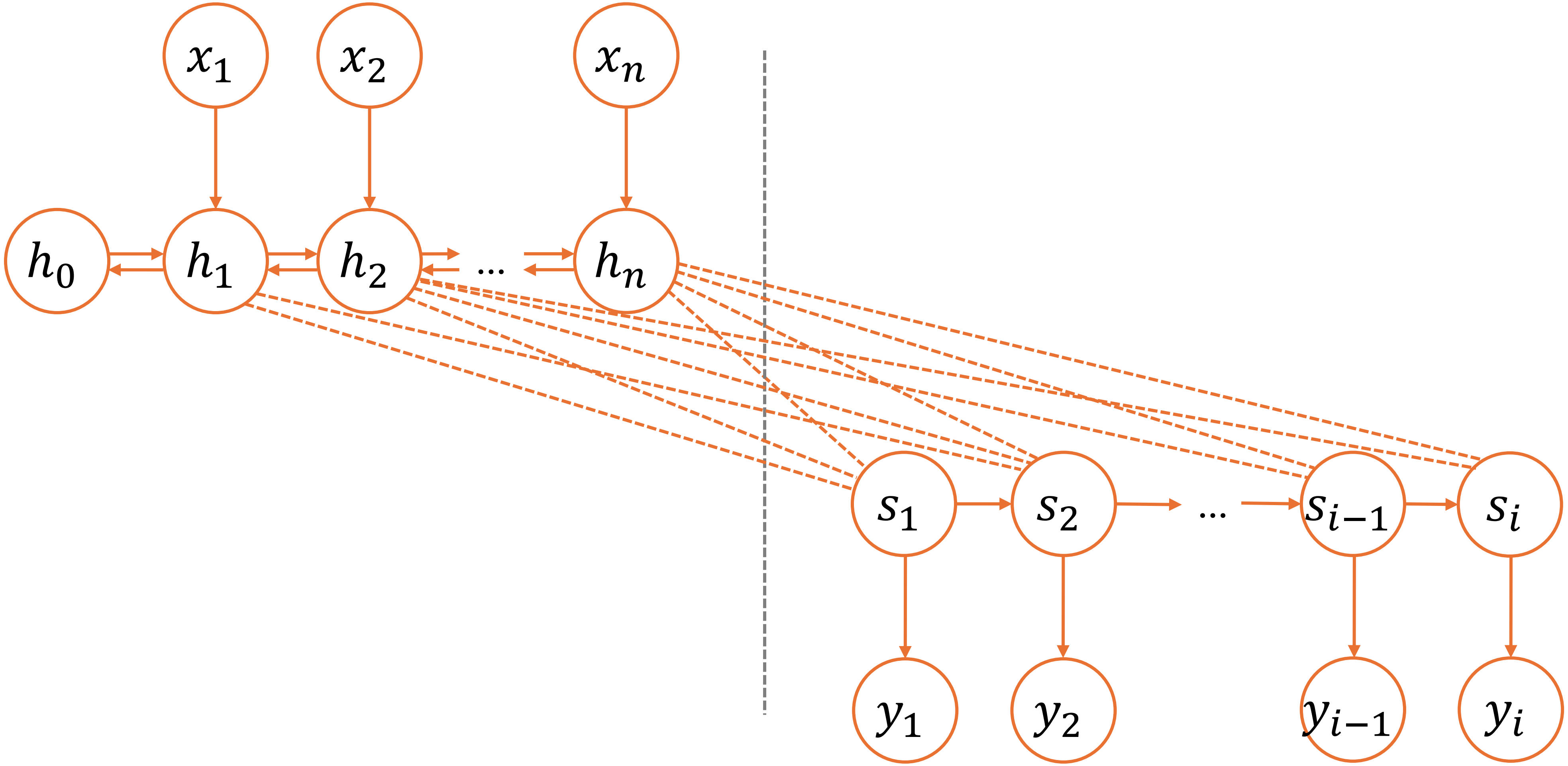
Modeling Language

Seq2Seq Models...



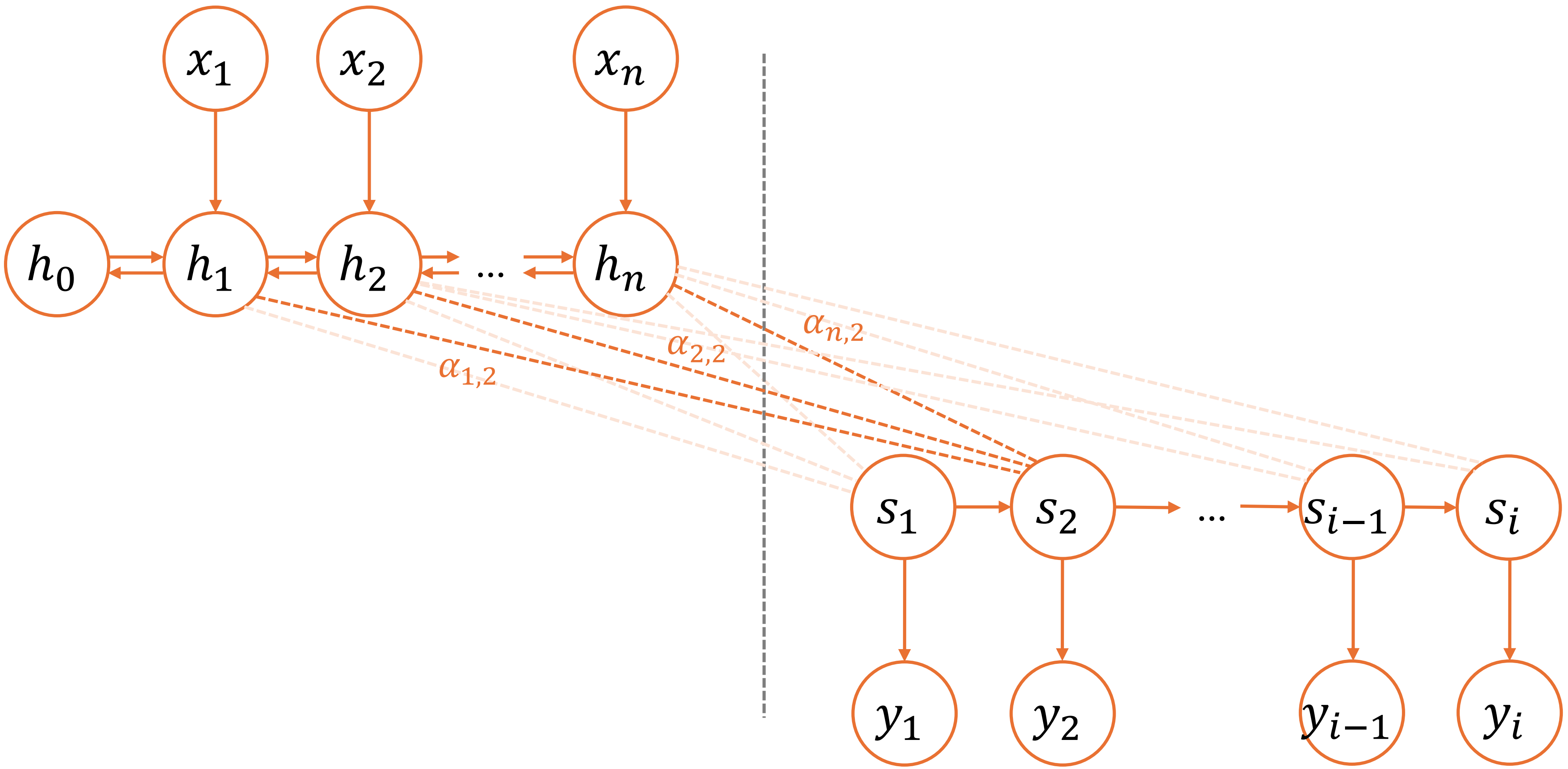
Modeling Language

Seq2Seq & Attention



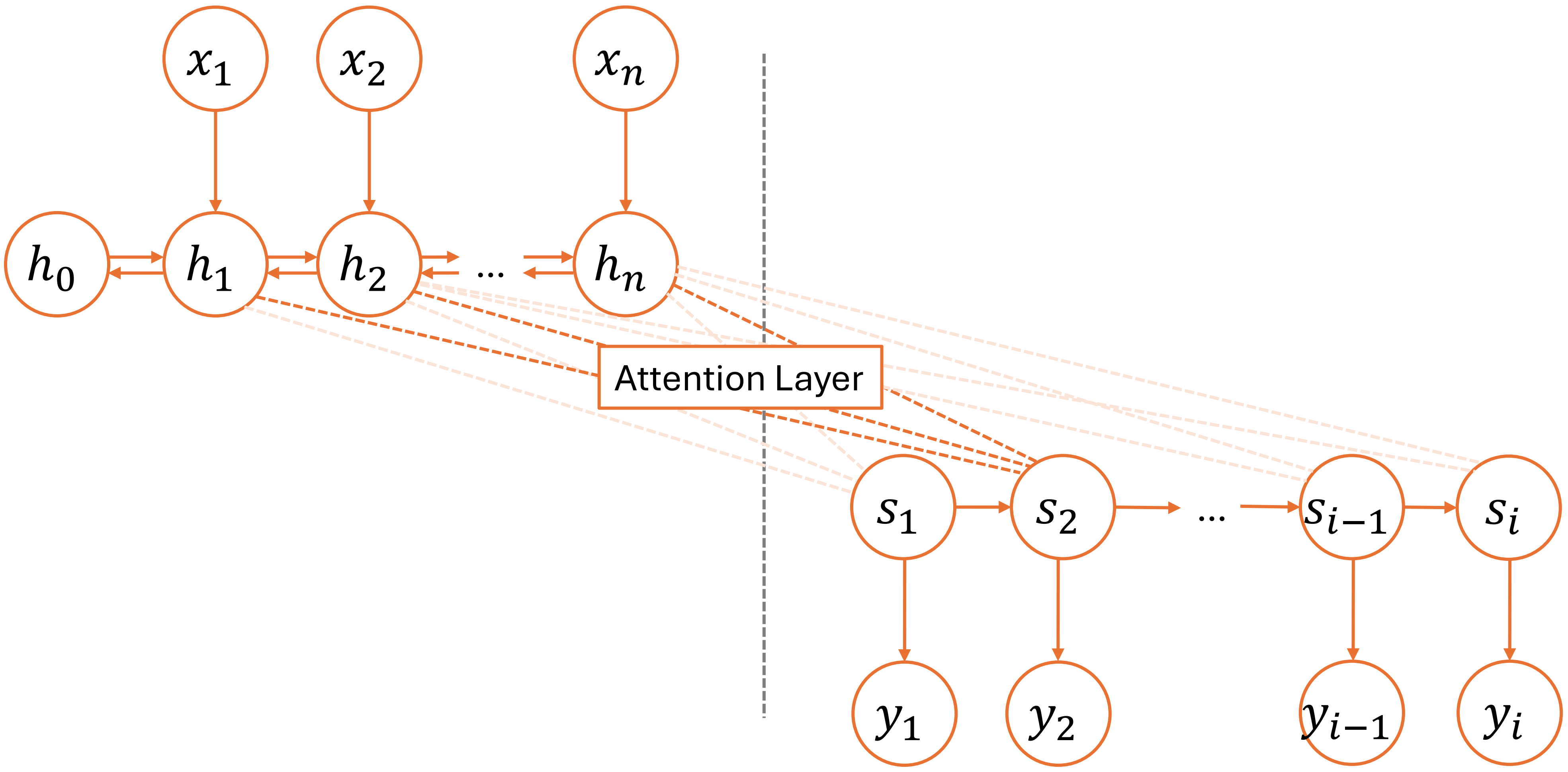
Modeling Language

Seq2Seq & Attention



Modeling Language

Seq2Seq & Attention



Neural Machine Translation by Jointly Learning to Align and Translate

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

NEURAL MACHINE TRANSLATION
BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio**[*]
Université de Montréal

In a new model architecture, we define each conditional probability in Eq. (2) as:

$$p(y_i|y_1, \ldots, y_{i-1}, \mathbf{x}) = g(y_{i-1}, s_i, c_i), \tag{4}$$

where s_i is an RNN hidden state for time i, computed by

$$s_i = f(s_{i-1}, y_{i-1}, c_i).$$

It should be noted that unlike the existing encoder–decoder approach (see Eq. (2)), here the probability is conditioned on a distinct context vector c_i for each target word y_i.

The context vector c_i depends on a sequence of *annotations* $(h_1, \cdots, h_{T_x})$ to which an encoder maps the input sentence. Each annotation h_i contains information about the whole input sequence with a strong focus on the parts surrounding the i-th word of the input sequence. We explain in detail how the annotations are computed in the next section.

The context vector c_i is, then, computed as a weighted sum of these annotations h_i:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j. \tag{5}$$

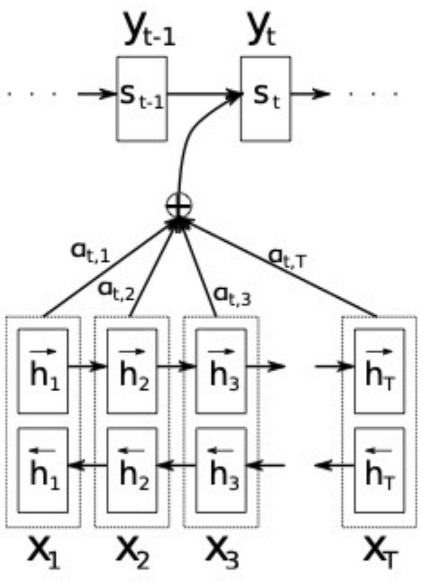


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

The weight α_{ij} of each annotation h_j is computed by

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}, \tag{6}$$

where

$$e_{ij} = a(s_{i-1}, h_j)$$

is an *alignment model* which scores how well the inputs around position j and the output at position i match. The score is based on the RNN hidden state s_{i-1} (just before emitting y_i, Eq. (4)) and the j-th annotation h_j of the input sentence.

Modeling Language

Seq2Seq & Attention

Problem Definition: **"Sequence-to-Sequence Translation"** (Next token prediction)

Input: Input Sequence $\boxed{\mathbf{x} = x_1, x_2, \dots, x_n}$ Output: Next Token $\boxed{y_i}$

Input: Generated tokens so far $\boxed{y_1, y_2, \dots, y_{i-1}}$

Goal: Compute the probability $\Pr(y_i \mid y_1, y_2, \dots, y_{i-1}, \mathbf{x})$

What problem can be phrased as "Translation" in Program Synthesis?

- Natural language to code generation
- Docstring → Implementation
- Code Repair: Buggy → Fixed Code
- Transpilation: Python Code → C Code

Modeling Language

Problem Definition: **"Sequence-to-Sequence Translation"** (Next token prediction)

Input: Input Sequence $\boxed{\mathbf{x} = x_1, x_2, \ldots, x_n}$ Output: Next Token $\boxed{y_i}$

Input: Generated tokens so far $\boxed{y_1, y_2, \ldots, y_{i-1}}$

Goal: Compute the probability $\Pr(y_i \mid y_1, y_2, \ldots, y_{i-1}, \mathbf{x})$

N-Gram Models → Recurrent Models → Seq2Seq & Attention → Transformers

Modeling Language

Problem Definition: **Next token prediction**

Input: Input Sequence $\boxed{\mathbf{x} = x_1, x_2, \ldots, x_n}$ Output: Next Token $\boxed{y_i}$

Input: Generated tokens so far $\boxed{y_1, y_2, \ldots, y_{i-1}}$

Goal: Compute the probability $\Pr(y_i \mid y_1, y_2, \ldots, y_{i-1}, \mathbf{x})$

N-Gram Models → Recurrent Models → Seq2Seq & Attention → Transformers

Modeling Language

Transformers

Problem Definition: **Next token prediction**

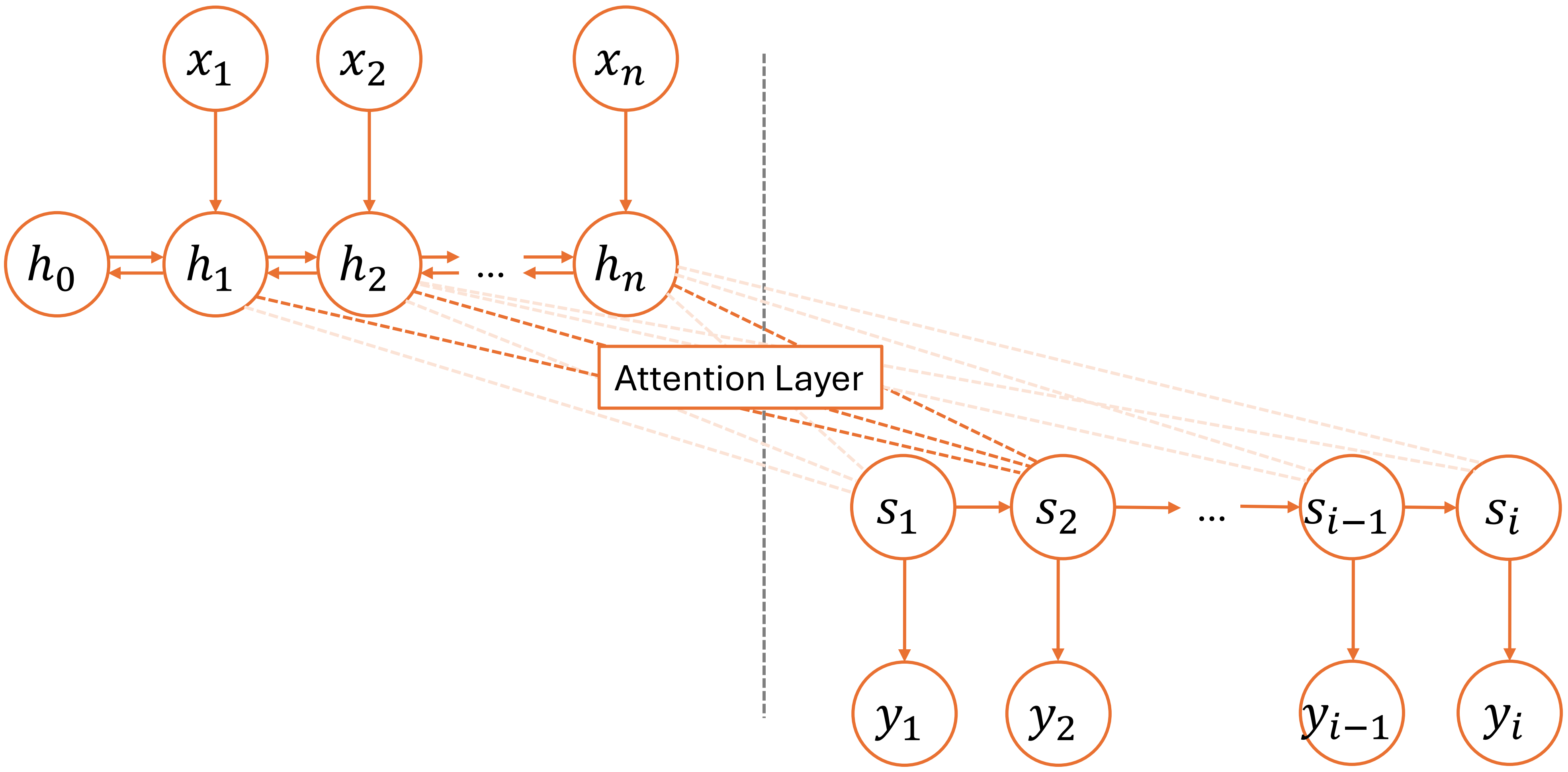
Input: Input Sequence $\boxed{\mathbf{x} = x_1, x_2, \dots, x_n}$ Output: Next Token $\boxed{y_i}$

Input: Generated tokens so far $\boxed{y_1, y_2, \dots, y_{i-1}}$

Goal: Compute the probability $\Pr(y_i \mid y_1, y_2, \dots, y_{i-1}, \mathbf{x})$

Modeling Language

Seq2Seq & Attention



Attention Is All You Need

Ashish Vaswani[*]
Google Brain
avaswani@google.com

Noam Shazeer[*]
Google Brain
noam@google.com

Niki Parmar[*]
Google Research
nikip@google.com

Jakob Uszkoreit[*]
Google Research
usz@google.com

Llion Jones[*]
Google Research
llion@google.com

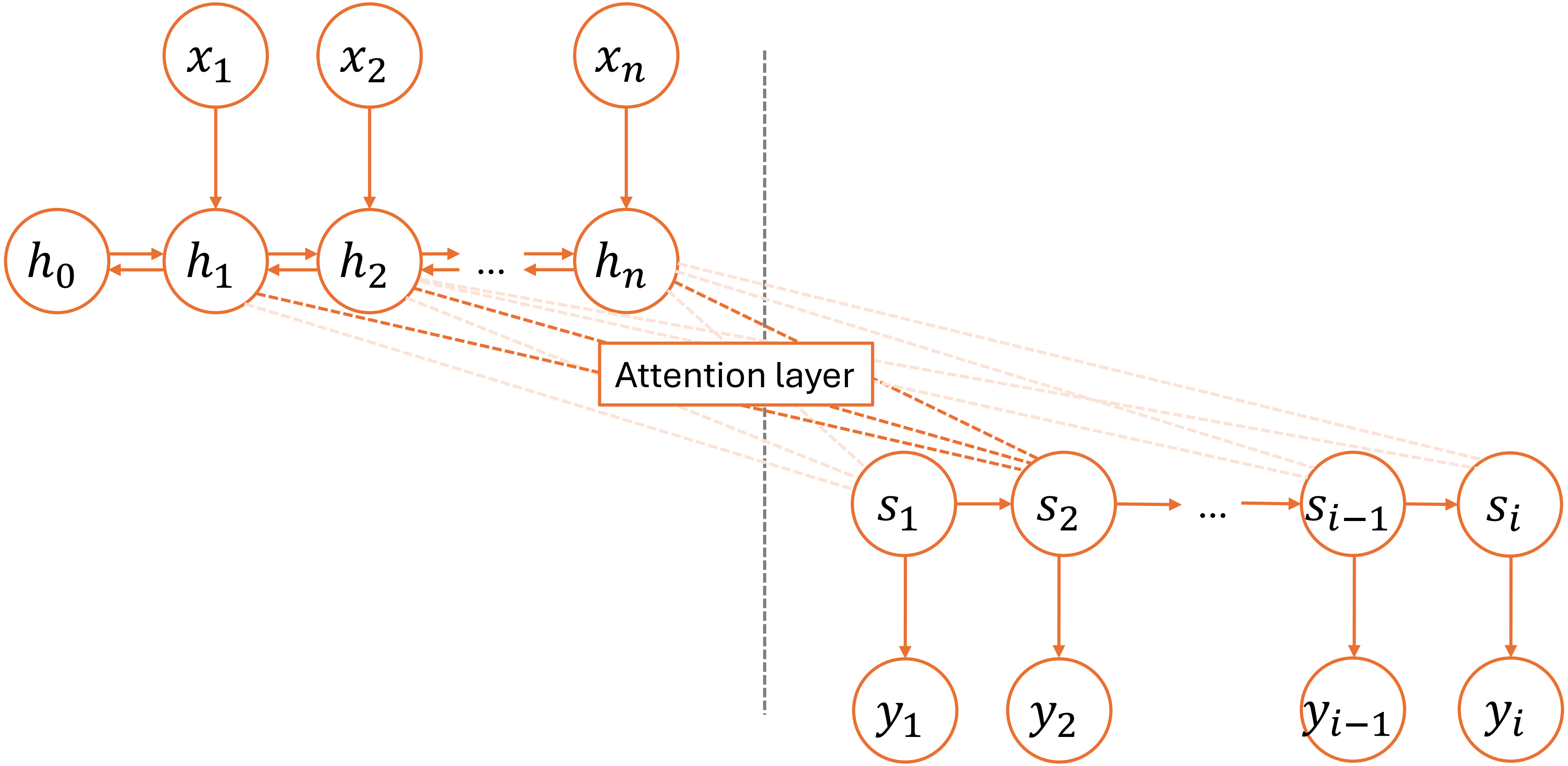
Aidan N. Gomez[*] [†]
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser[*]
Google Brain
lukaszkaiser@google.com

Illia Polosukhin[*] [‡]
illia.polosukhin@gmail.com

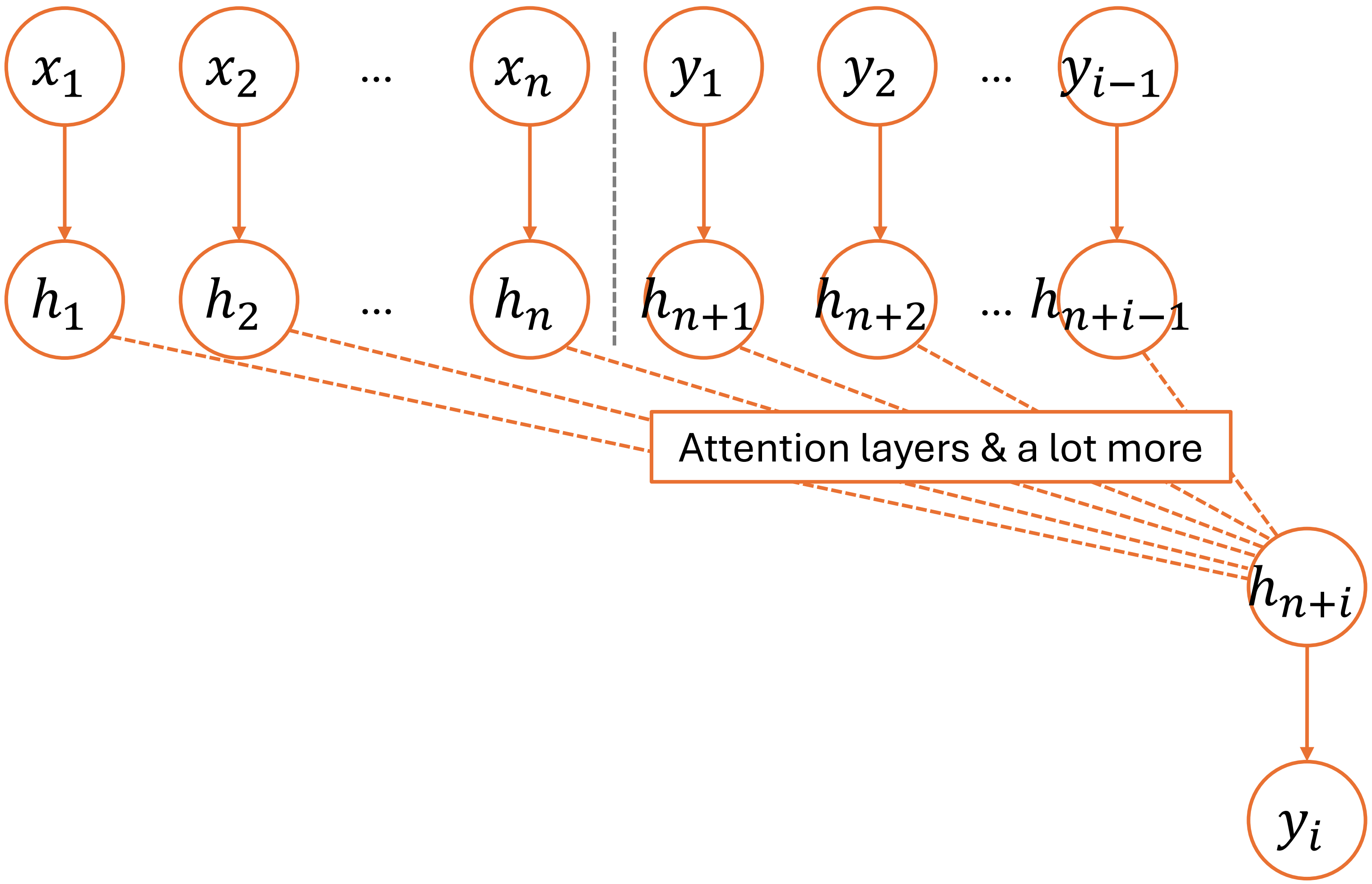
Modeling Language

Seq2Seq & Attention



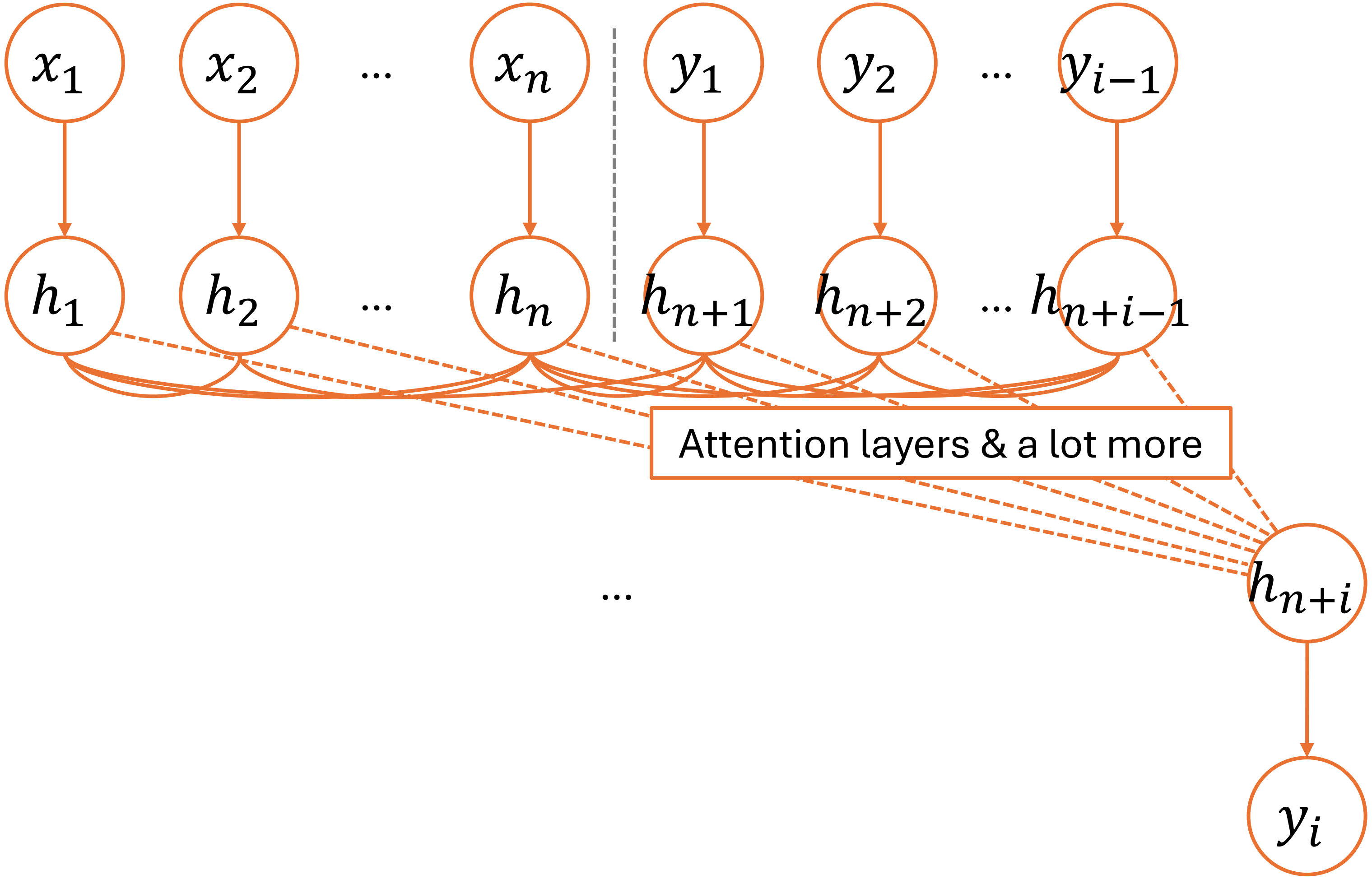
Modeling Language

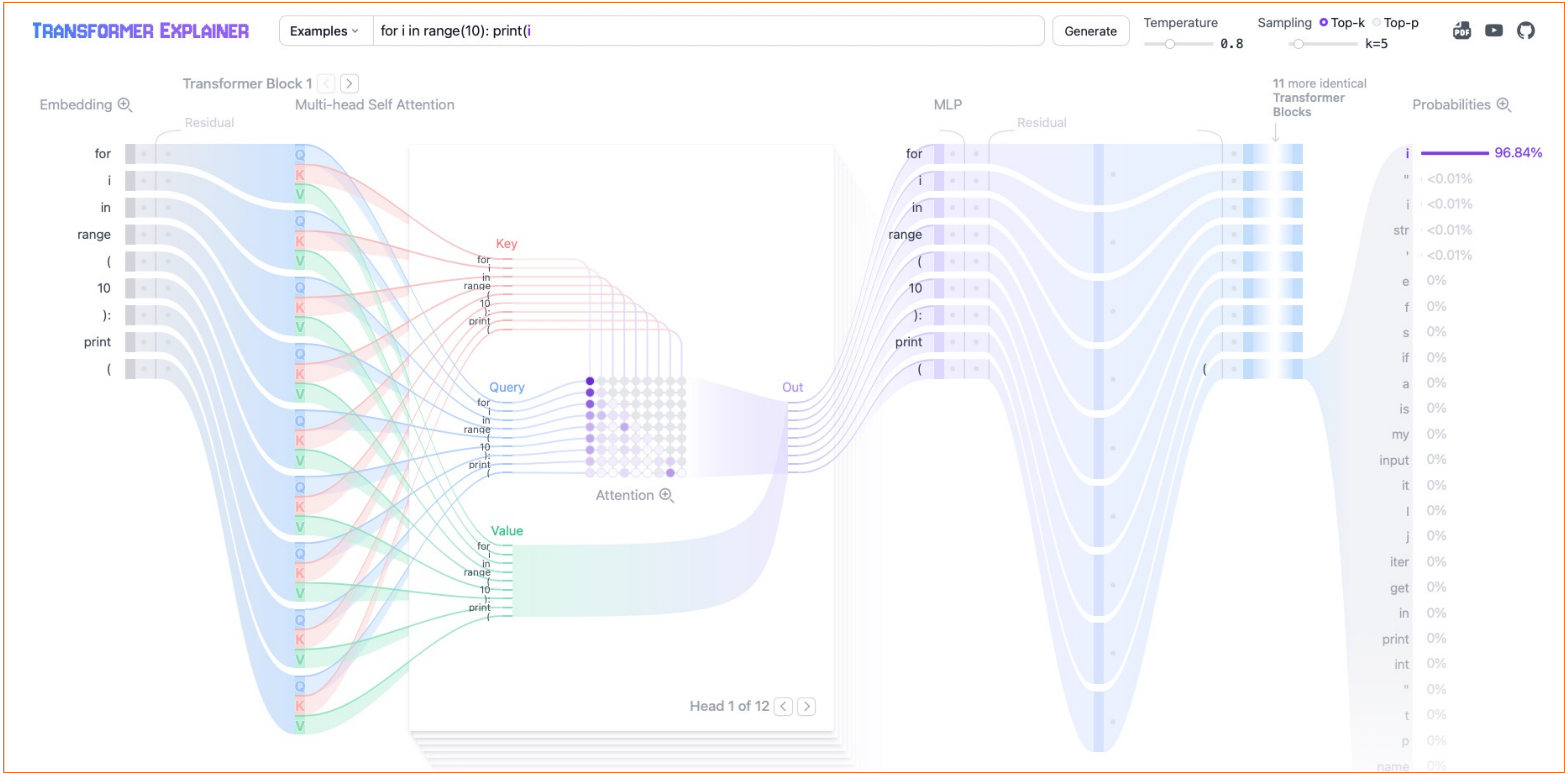
Transformer



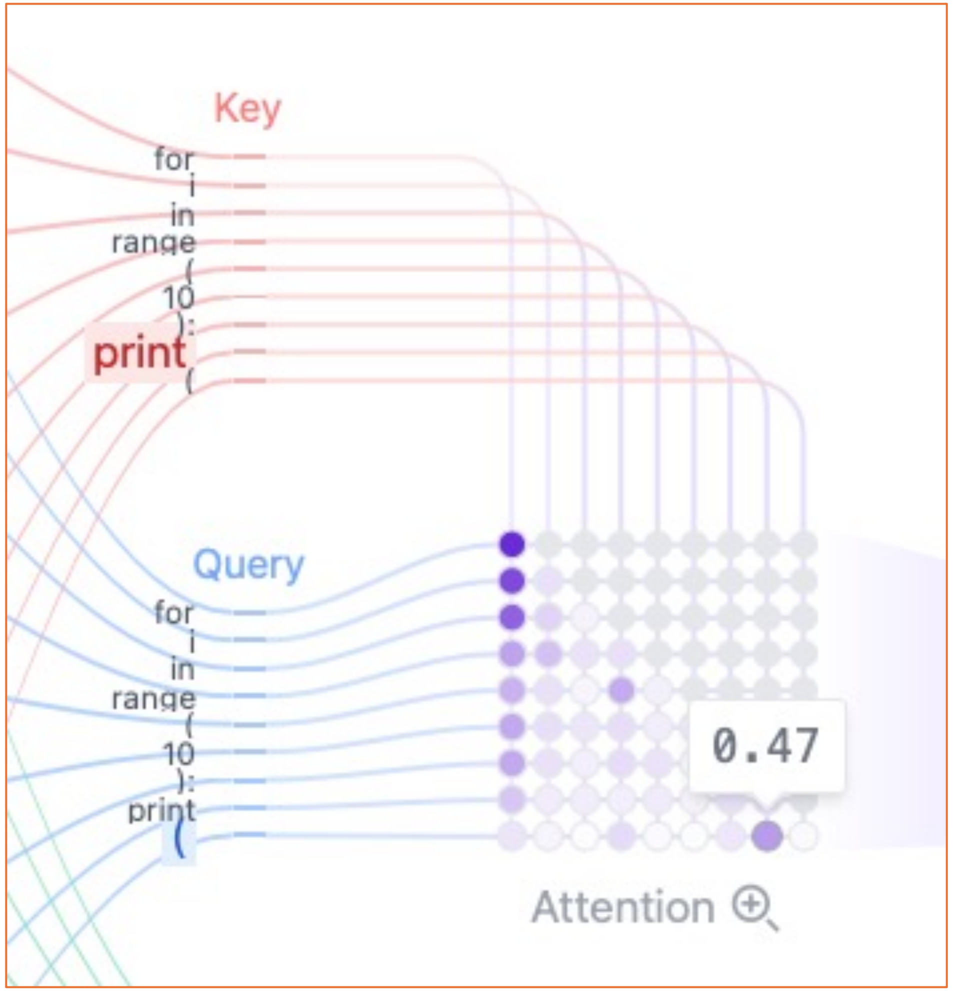
Modeling Language

Transformer





https://poloclub.github.io/transformer-explainer/



Key
for
i
in
range
(
10
):
print
(
Query
for
i
in
range
(
10
):
print
(
0.47
Attention

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

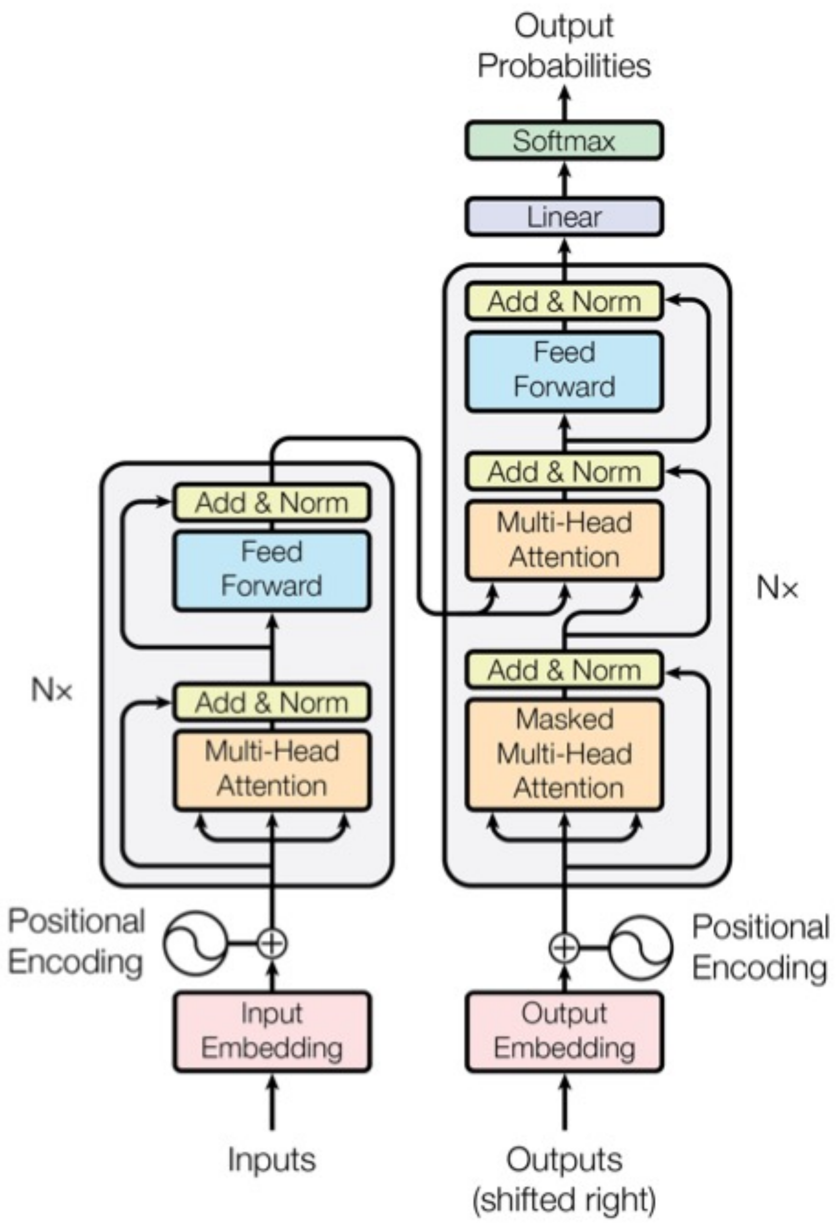


Figure 1: The Transformer - model architecture.

Modeling Language

Transformers

Problem Definition: **Next Token Prediction**

Input: Input Sequence $\boxed{\mathbf{x} = x_1, x_2, \ldots, x_n}$ Output: Next Token $\boxed{y_i}$

Input: Generated tokens so far $\boxed{y_1, y_2, \ldots, y_{i-1}}$

Goal: Compute the probability $\Pr(y_i \mid y_1, y_2, \ldots, y_{i-1}, \mathbf{x})$

Transformers: How they work

- Self-Attention for the input
- Self-Attention for the output
- Cross-Attention for combining inputs and current output together
- Additional machinery: positional encodings, multi-headed attention, etc.
- Causal decoder-only models

Modeling Language

Problem Definition: **Next Token Prediction**

Input: Prefix $\boxed{t_1, t_2, \ldots, t_{i-1}}$ Output: Next Token $\boxed{t_i}$

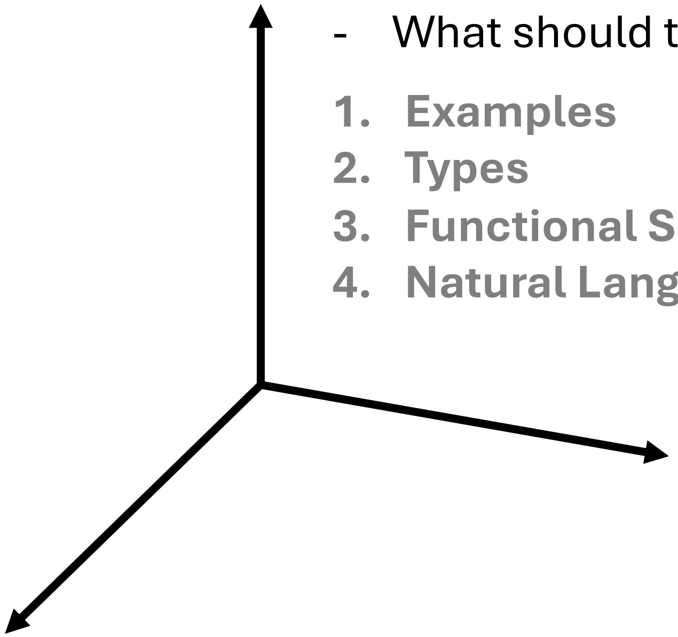
Goal: Compute the probability $\Pr(t_i \mid t_1, t_2, \ldots, t_{i-1})$

N-Gram Models → Recurrent Models → Seq2Seq & Attention → Transformers

Today

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?

Data-Driven Approaches
- Modeling programs as token sequences
- Modeling synthesis as next token prediction

Structural Specification
- What is the space of the programs?

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

Week 3

- Assignment 1
  - https://github.com/machine-programming/assignment-1
  - Start working on the assignment early!
- Assignment 2
  - Will be released later this week!
- Waitlist
  - All waitlisted students should be able to enroll
  - Let me know if you do not have access to courselore and GradeScope
  - API keys should have been sent out