EECS 598-008 & EECS 498-008: Intelligent Programming Systems

Lecture 3

Announcements

- A1 out, due **midnight Tuesday September 14**
  - Implement top-down search algorithm. More interesting. Start early!
- Grade will be displayed in **percentage** on Canvas
- Submit you assignment as a **single** .zip file (instead of .java and .pdf separately)
- **Remote & live** discussion section **3-4pm Friday September 10**
  - Discussion zoom password same as lecture zoom password
  - Primarily walk through top-down search algorithm lecture, A1, also briefly Z3
  - See schedule for details
- Piazza for questions
  - GSI and instructor will monitor for questions

Agenda

- **Inductive program synthesis, in particular, Programming-by-Example**
- Domain-Specific Languages (DSLs)
- Abstract Syntax Trees (ASTs)
- Overview of search techniques

Inductive Program Synthesis

"Specification"

Inductive

"Program"

Inductive Program Synthesis

- Specification is "inductive" (topic today)
  - Inductive: incomplete, under-specified
  - E.g., test cases, input-output examples, under-constrained logical formulas, etc.
- Counterpart: complete specifications (will talk about this in a few lectures)

Inductive Program Synthesis

- Specification is “inductive” (topic today)
  - Inductive: incomplete, under-specified
  - E.g., test cases, input-output examples, under-constrained logical formulas, etc.
- Counterpart: complete specifications (will talk about this in a few lectures)
- Why inductive specification? Simple!
  - A broader class of users can provide
  - One of the simplest interfaces for program synthesis

Inductive Program Synthesis

- E.g., 1 —> 2
  - Say, the CFG is $e ::= x \mid e + e \mid e \times e \mid 1 \mid 2 \mid 3 \mid 4$
  - What are some programs in this CFG that satisfy the spec?

Inductive Program Synthesis

- E.g., 1 —> 2
  - Say, the CFG is $e ::= x \mid e + e \mid e \times e \mid 1 \mid 2 \mid 3 \mid 4$
  - What are some programs in this CFG that satisfy the spec?
  - How about this CFG $e ::= x \mid e + e \mid e \times e \mid e - e \mid 1 \mid 2 \mid 3 \mid 4$

Inductive Program Synthesis

- Does an inductive program synthesizer always give back the “correct” program?

Inductive Program Synthesis

- Does an inductive program synthesizer always give back the “correct” program?
  - Yes, it always gives back a program that satisfies the spec
  - No, it may not give back a program that actually meets the user’s intent

Inductive Program Synthesis

- Does an inductive program synthesizer always give back the “correct” program?
  - Yes, it always gives back a program that satisfies the spec
  - No, it may not give back a program that actually meets the user’s intent
- Known as the “overfitting” or “generalization” problem
  - Inductive spec is a partial representation of the user’s intent
  - We’ll talk about this later in the course

Problems in Inductive Program Synthesis

- **Generalization**: Is the program you found the one that you're **actually** looking for? (Constrained search space, ranking, etc.)

Problems in Inductive Program Synthesis

- **Generalization**: Is the program you found the one that you're **actually** looking for? (Constrained search space, ranking, etc.)

- **Search**: How to find **a program** that satisfies the spec? (Top-down search, bottom-up search, etc.)

Problems in Inductive Program Synthesis

- **Generalization**: Is the program you found the one that you're **actually** looking for? (Constrained search space, ranking, etc.)
- **Search**: How to find **a program** that satisfies the spec? (Top-down search, bottom-up search, etc.)
- **Efficiency**: How to **efficiently** find a program that satisfies the spec? (Pruning, prioritization, etc.)

Problems in Inductive Program Synthesis

- **Generalization**: Is the program you found the one that you're **actually** looking for? (Constrained search space, ranking, etc.)
- **Search**: How to find **a program** that satisfies the spec? (Top-down search, bottom-up search, etc.)
- **Efficiency**: How to **efficiently** find a program that satisfies the spec? (Pruning, prioritization, etc.)
- **Search space**: How to **define the space** of programs in the first place? (Domain-specific languages, CFGs, functional languages, etc.)

Problems in Inductive Program Synthesis

- **Generalization**: Is the program you found the one that you're **actually** looking for? (Constrained search space, ranking, etc.)
- **Search**: How to find **a program** that satisfies the spec? (Top-down search, bottom-up search, etc.)
- **Efficiency**: How to **efficiently** find a program that satisfies the spec? (Pruning, prioritization, etc.)
- **Search space**: How to **define the space** of programs in the first place? (Domain-specific languages, CFGs, functional languages, etc.)
- **Specs**: How to support **different** inductive specs? (Examples, types, demonstrations, etc.)

Problems in Inductive Program Synthesis

- **Generalization**: Is the program you found the one that you're **actually** looking for? (Constrained search space, ranking, etc.)
- **Search**: How to find **a program** that satisfies the spec? (Top-down search, bottom-up search, etc.)
- **Efficiency**: How to **efficiently** find a program that satisfies the spec? (Pruning, prioritization, etc.)
- **Search space**: How to **define the space** of programs in the first place? (Domain-specific languages, CFGs, functional languages, etc.)
- **Specs**: How to support **different** inductive specs? (Examples, types, demonstrations, etc.)
- **Etc. such as noise, multi-modality, interaction, …**

Programming-by-Example (PBE)

- Most lectures in this course focus on PBE
- A particular form of inductive synthesis where specs are examples

Programming-by-Example (PBE)

- Most lectures in this course focus on PBE
- A particular form of inductive synthesis where specs are examples
- Still work with syntax-guided synthesis (SYGUS) paradigm
  - Spec: examples (which can be represented as logical formulas)
  - Search space: CFGs, in particular, functional domain-specific languages (DSLs)
  - Search: we will talk about different search techniques

Agenda

- Inductive program synthesis, in particular, Programming-by-Example
- **Domain-Specific Languages (DSLs)**
- Abstract Syntax Trees (ASTs)
- Overview of search techniques

Domain-Specific Languages (DSLs)

- DSLs are PLs, but more specialized and less universal

Domain-Specific Languages (DSLs)

- DSLs are PLs, but more specialized and less universal
- What’s a program?
- What’s a programming language?

Domain-Specific Languages (DSLs)

- DSLs are PLs, but more specialized and less universal
- What's a program?
  - A description of how to perform a computation
- What's a programming language?

Domain-Specific Languages (DSLs)

- DSLs are PLs, but more specialized and less universal
- What's a program?
  - A description of how to perform a computation
- What's a programming language?
  - A description of many computations by compositing individual syntactic elements each with well-defined meaning.
  - Syntax: How to write a program in a PL?
  - Semantics: What does this program mean?

Domain-Specific Languages (DSLs)

- Examples of "universal" PLs: Python, Java, C/C++, …
  - Describe many computations: add numbers, sort lists, transform trees, …

Domain-Specific Languages (DSLs)

- Examples of “universal” PLs: Python, Java, C/C++, …
  - Describe many computations: add numbers, sort lists, transform trees, …
- DSLs: PLs specialized to specific tasks and not universal
  - Describe different computations by composing individual syntactic elements each with well defined meaning
  - E.g., SQL
  - E.g., arithmetic expressions $e ::= x \mid e + e \mid e \times e \mid 1 \mid 2 \mid 3 \mid 4$
  - Or could be defined by you

Domain-Specific Languages (DSLs)

- To define a DSL
  - Syntax: CFG (operators and compositions)
  - Semantics: What does every operator and composition mean?
    - Often it's enough to use examples to define semantics
    - Or translating into a general-purpose PL
    - But to fully specify semantics, need formal semantics (not this course)
    - This course: use informal semantics

Functional Programming Languages

- Programming paradigms: functional (e.g., Haskell), imperative (e.g., C), …

Functional Programming Languages

- Programming paradigms: functional (e.g., Haskell), imperative (e.g., C), …
- We will mainly use functional PLs for synthesis, because:
  - No side effects. Computation in functional PLs is by evaluating **pure** functions, without side effects or mutations. This greatly simplifies synthesis.

```
List reverseList(List input) {
  List output = new ArrayList();
  for (int i = 0; I < input.size(); i ++) {
    output.add(input.get(input.size() - 1 - i);
  }
  return output;
}
```

```
reverse l = case l of
              [] -> []
              head : rest -> (reverse rest) ++ [head]
```

Functional Programming Languages

- Programming paradigms: functional (e.g., Haskell), imperative (e.g., C), …
- We will mainly use functional PLs for synthesis, because:
  - No side effects. Computation in functional PLs is by evaluating **pure** functions, without side effects or mutations. This greatly simplifies synthesis.
  - Concise language.
  - Expressiveness.

```
List reverseList(List input) {
  List output = new ArrayList();
  for (int i = 0; I < input.size(); i ++) {
    output.add(input.get(input.size() - 1 - i);
  }
  return output;
}
```

```
reverse l = case l of
              [] -> []
              head : rest -> (reverse rest) ++ [head]
```

Example DSL

- Consider the following DSL
  - Syntax in CFG $df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$

    $$k ::= 1 \mid 2 \mid 3 \mid 4$$

    $$s ::= tmp1 \mid tmp2 \mid tmp3$$

  - What are some programs in this DSL?

Example DSL

- Consider the following DSL
  - Syntax in CFG $df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$

    $$k ::= 1 \mid 2 \mid 3 \mid 4$$

    $$s ::= tmp1 \mid tmp2 \mid tmp3$$

  - Semantics
    - What does every DSL construct/operator mean? What does it evaluate to?
    - E.g., *gather* function: http://statseducation.com/Introduction-to-R/modules/tidy%20data/gather/

Example DSL

- Semantics
  - What does every DSL construct/operator mean? What does it evaluate to?

```
gather(data, key, value, ...)
```

where **This is really the syntax of *gather***

- `data` is the dataframe you are working with.
- `key` is the name of the `key` column to create.
- `value` is the name of the `value` column to create.
- `...` is a way to specify what columns to gather from.

Example DSL

- Semantics
  - What does every DSL construct/operator mean? What does it evaluate to?

```
## # A tibble: 3 × 3
##       country `1999` `2000`
##        <fctr>  <int>  <int>
## 1 Afghanistan    745   2666
## 2      Brazil  37737  80488
## 3       China 212258 213766
```

```
table4 %>%
  gather("year", "cases", 2:3)
```

```
## # A tibble: 6 × 3
##       country  year  cases
##        <fctr> <chr>  <int>
## 1 Afghanistan  1999    745
## 2      Brazil  1999  37737
## 3       China  1999 212258
## 4 Afghanistan  2000   2666
## 5      Brazil  2000  80488
## 6       China  2000 213766
```

This example gives you an idea what *gather* actually does

Example DSL

- Consider the following DSL (which is used in assignments)
  - Syntax in CFG

$$df ::= x \mid gather(df, k, k, s, s) \mid unite(df, k, k, s)$$

$$k ::= 1 \mid 2 \mid 3 \mid 4$$

$$s ::= tmp1 \mid tmp2 \mid tmp3$$

  - Semantics
    - What does every DSL construct/operator mean? What does it evaluate to?
    - E.g., *gather* function: http://statseducation.com/Introduction-to-R/modules/tidy%20data/gather/
    - Note that *gather* is recursive: the first parameter *df* can also be a *gather* function

Agenda

- Inductive program synthesis, in particular, Programming-by-Example
- Domain-Specific Languages (DSLs)
- **Abstract Syntax Trees (ASTs)**
- Overview of search techniques

DSL Programs as Abstract Syntax Trees (ASTs)

- In actual programming, programs are strings/text with indentation, special chars, etc.
- When reasoning about programs:
  - Programs are presented as data structures
  - A common one is Abstract Syntax Trees (ASTs)
- Abstract: ASTs ignore uninteresting details such has spacing, parenthesis, …
- Syntax: No semantic information
- Tree: It’s essentially a tree

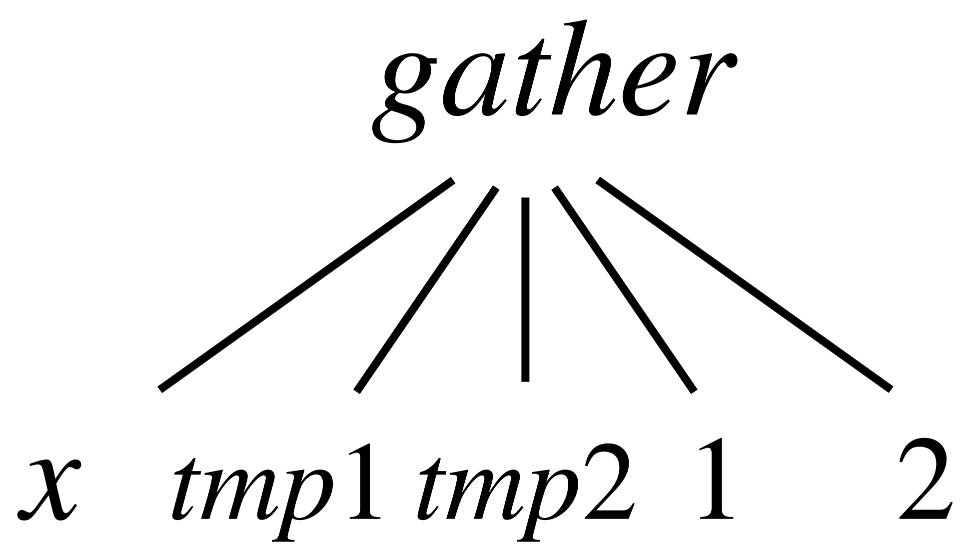
DSL Programs as Abstract Syntax Trees (ASTs)

- Consider the following CFG $df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$

$$k ::= 1 \mid 2 \mid 3 \mid 4$$

$$s ::= tmp1 \mid tmp2 \mid tmp3$$

- Consider program $gather(x, tmp1, tmp2, 1, 2)$



DSL Programs as Abstract Syntax Trees (ASTs)

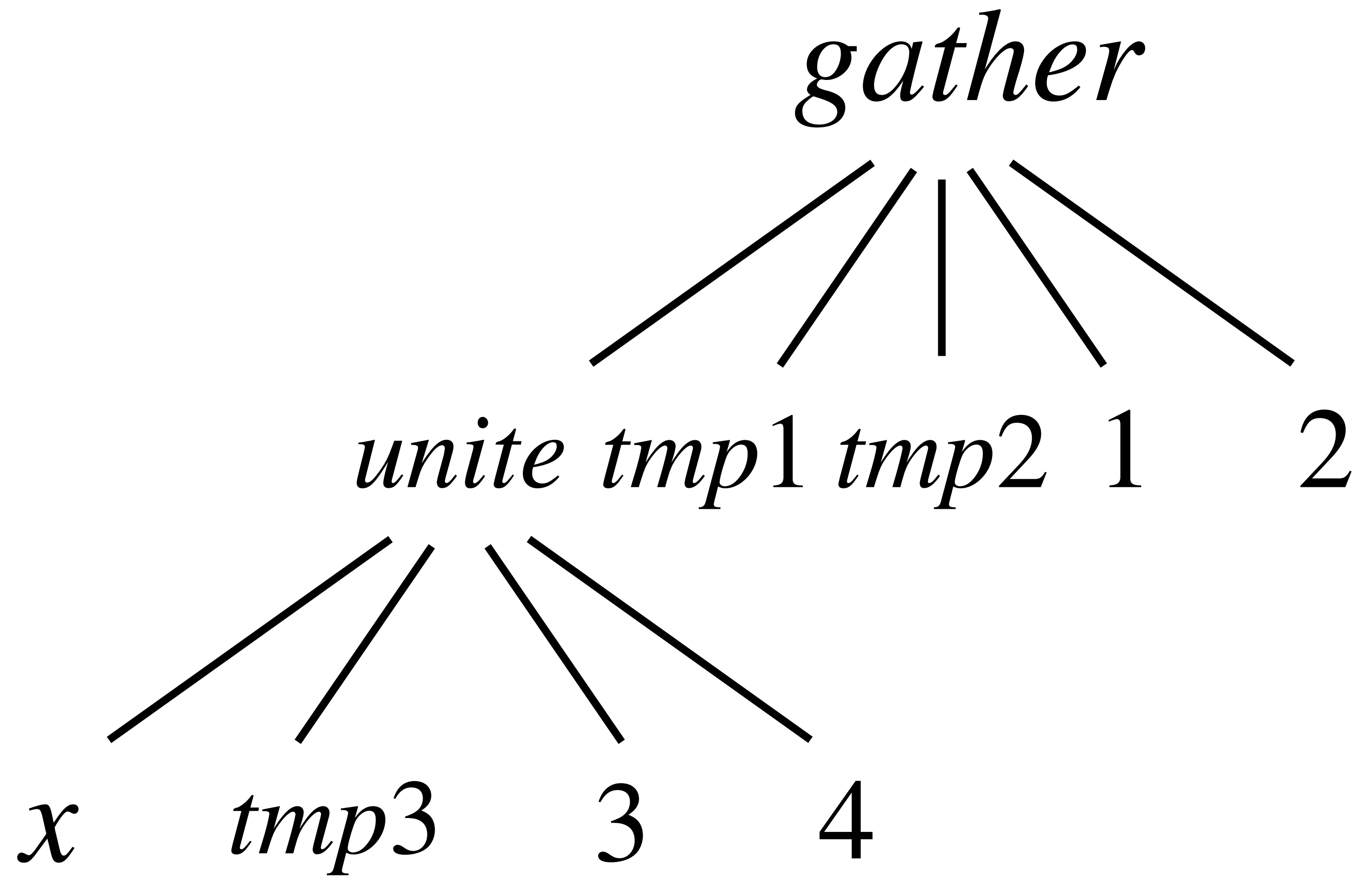
- Consider the following CFG

$$df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$$

$$k ::= 1 \mid 2 \mid 3 \mid 4$$

$$s ::= tmp1 \mid tmp2 \mid tmp3$$

- Consider program $gather(unite(x, tmp3,3,4), tmp1,tmp2,1,2)$



Agenda

- Inductive program synthesis, in particular, Programming-by-Example
- Domain-Specific Languages (DSLs)
- Abstract Syntax Trees (ASTs)
- **Overview of search techniques**

Overview of Search Techniques for PBE

- PBE Problem: Given a DSL with predefined syntax (in a CFG G) and semantics and given a set E of input-output examples, find a program $P \in G$ such that P satisfies E

Overview of Search Techniques for PBE

- PBE Problem: Given a DSL with predefined syntax (in a CFG G) and semantics and given a set E of input-output examples, find a program $P \in G$ such that P satisfies E

- *SYGUS: Given a first-order formula ϕ in a background theory T and a CFG G, the syntax-guided synthesis problem is to find an expression $e \in G$ such that formula $\phi[f/e]$ is valid in theory T.*

Overview of Search Techniques for PBE

- PBE Problem: Given a DSL with predefined syntax (in a CFG G) and semantics and given a set E of input-output examples, find a program $P \in G$ such that P satisfies E
- Different search techniques:
  - **Enumeration-based techniques: Top-down and bottom-up search (Today)**
  - Representation-based techniques: Version Space Algebra, Finite Tree Automata
  - Constraint-based approaches
  - Stochastic search: MCMC
  - Many other techniques such as using genetic programming, NN, RL, …

Enumeration-based Approaches

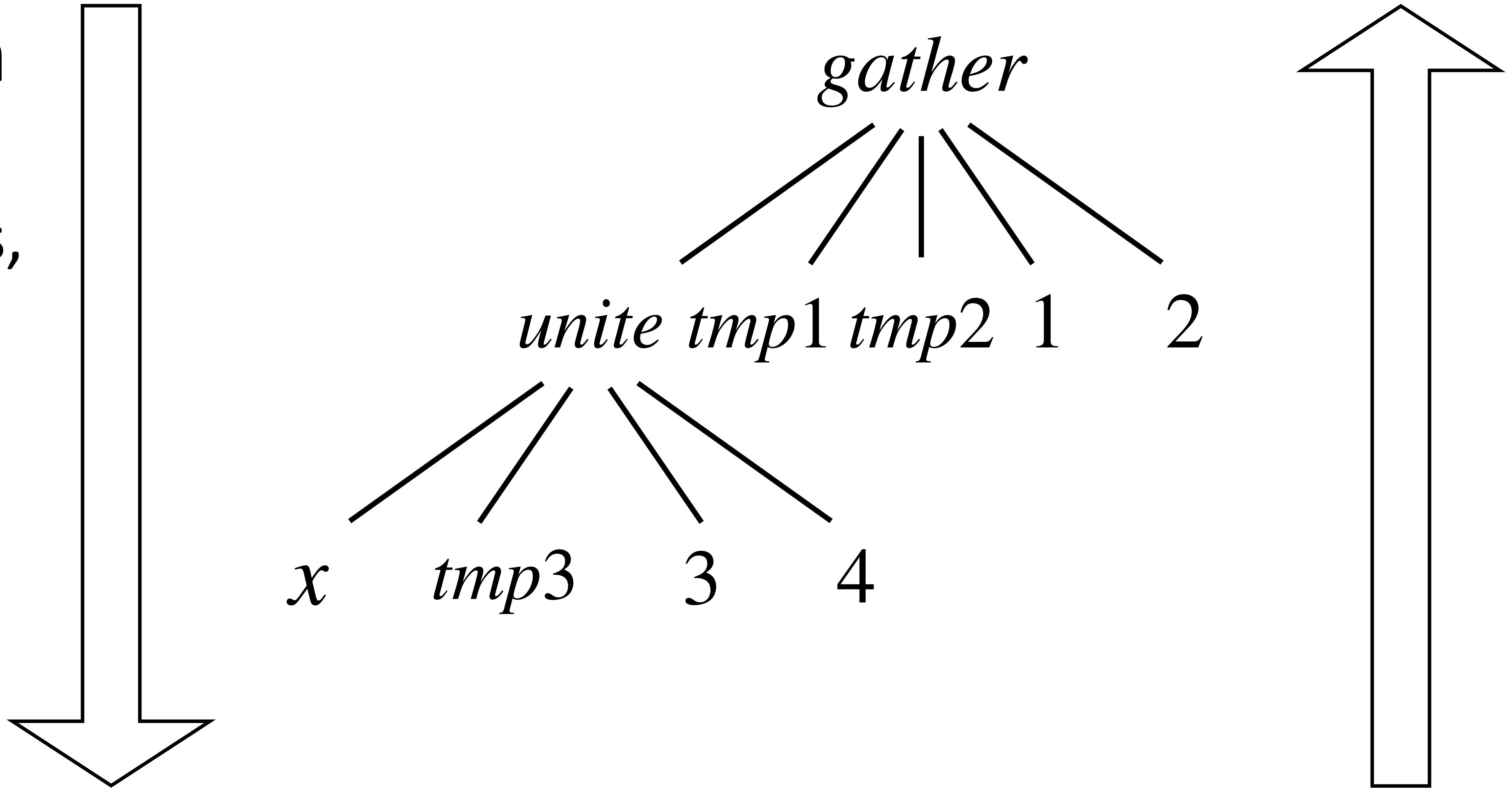
- Key question: How to **systematically enumerate** all programs/ASTs in a given CFG?
  - Another perspective: How to **systematically generate** ASTs?
- Then, we just need to check each AST against examples

Enumeration-based Approaches

- Key question: How to **systematically generate** an AST?
- Two ideas: Top-down and bottom-up

Top-down

First generate parents,
then children,
then grandchildren,
etc…

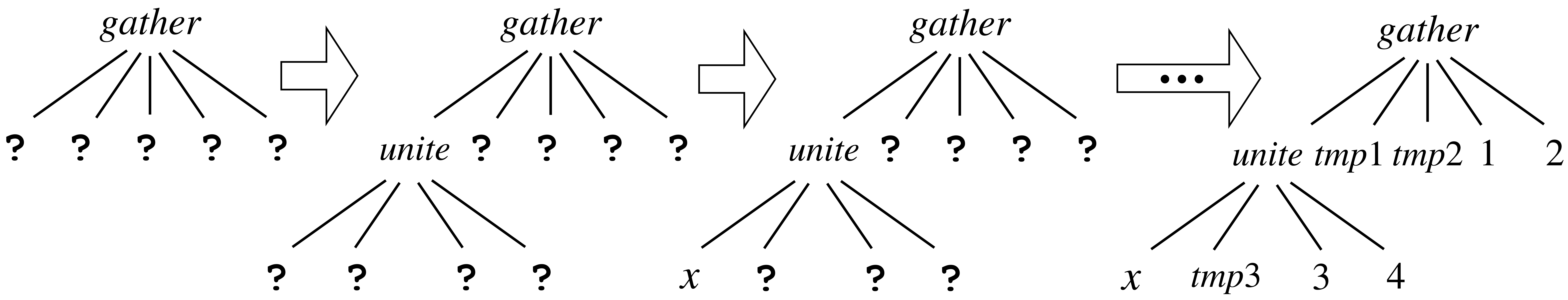


Bottom-up

First leaves,
then parents,
then grandparents,
etc…

Top-Down Search

- Key idea: A parent node was generated before its children are generated
  - Or, generate high(er) level structures first, then fill it with low(er) level fragments



Top-Down Search Algorithm

- Given a CFG $G = (T, N, P, S)$ and a set E of examples:

  <u>Top-Down-Search</u> ((T, N, P, S), E):

    worklist := { S };

    while (*worklist* **is not empty**):

      AST := *worklist*.remove();

      if (AST **is complete &** AST **satisfies** E): **return** AST;

      worklist.addAll(**expand**(AST));

- High-level idea: An iterative algorithm that manipulates ASTs and creates more ASTs

Top-Down Search Algorithm

Top-Down-Search ($(T, N, P, S), E$):

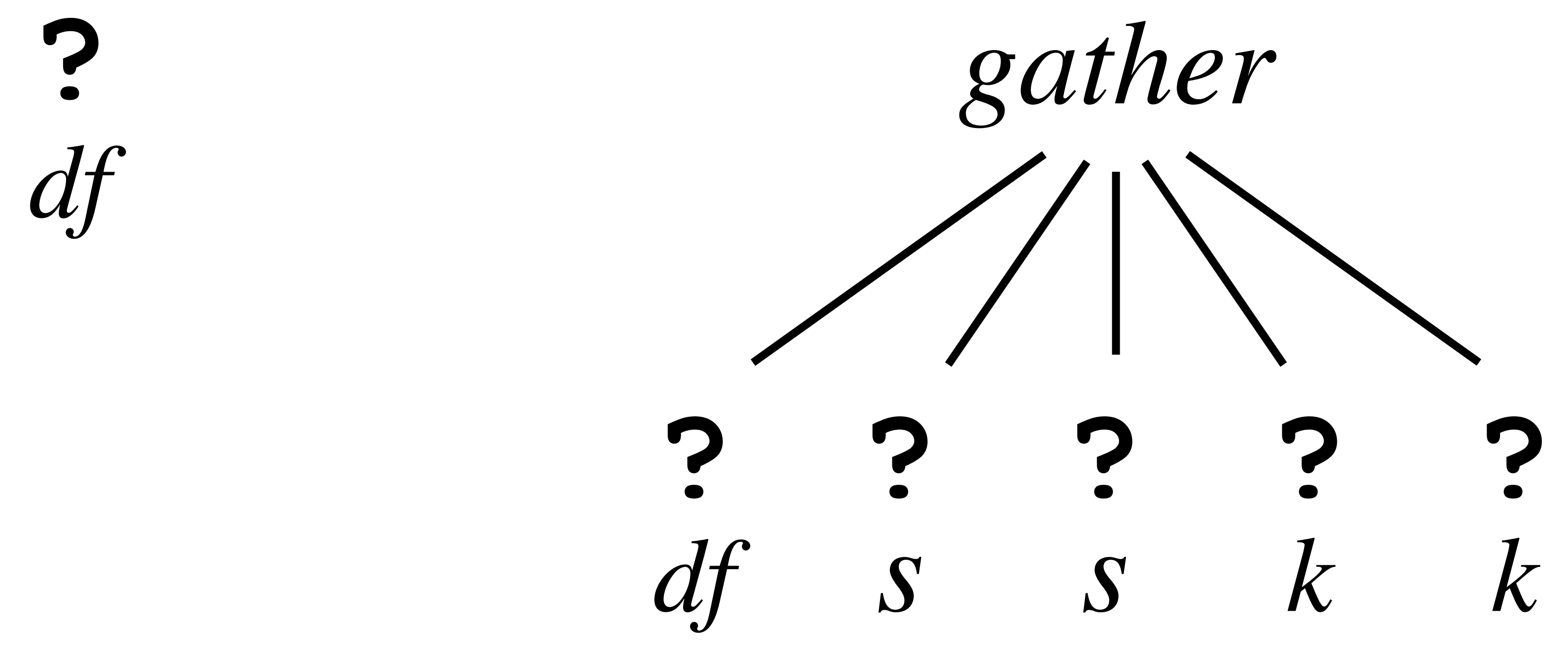
worklist := { S }; → A set of ASTs. We allow AST nodes to be "holes" labeled with the associated grammar symbol.

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));



$df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$

$k ::= 1 \mid 2 \mid 3 \mid 4$

$s ::= tmp1 \mid tmp2 \mid tmp3$

Top-Down Search Algorithm

<u>Top-Down-Search</u> ((T, N, P, S), E):

worklist := { S };

while (*worklist* **is not empty**):

AST := *worklist*.remove(); → Get an AST from the worklist.

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

$$df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$$
$$k ::= 1 \mid 2 \mid 3 \mid 4$$
$$s ::= tmp1 \mid tmp2 \mid tmp3$$

Top-Down Search Algorithm

<u>**Top-Down-Search**</u> ((T, N, P, S), E):

worklist := { S };

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST; → We're done if it satisfies the given examples!

worklist.addAll(**expand**(AST));

$$df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$$
$$k ::= 1 \mid 2 \mid 3 \mid 4$$
$$s ::= tmp1 \mid tmp2 \mid tmp3$$

Top-Down Search Algorithm

<u>**Top-Down-Search**</u> ($(T, N, P, S), E$):

worklist := { S };

while (*worklist* **is not empty**):

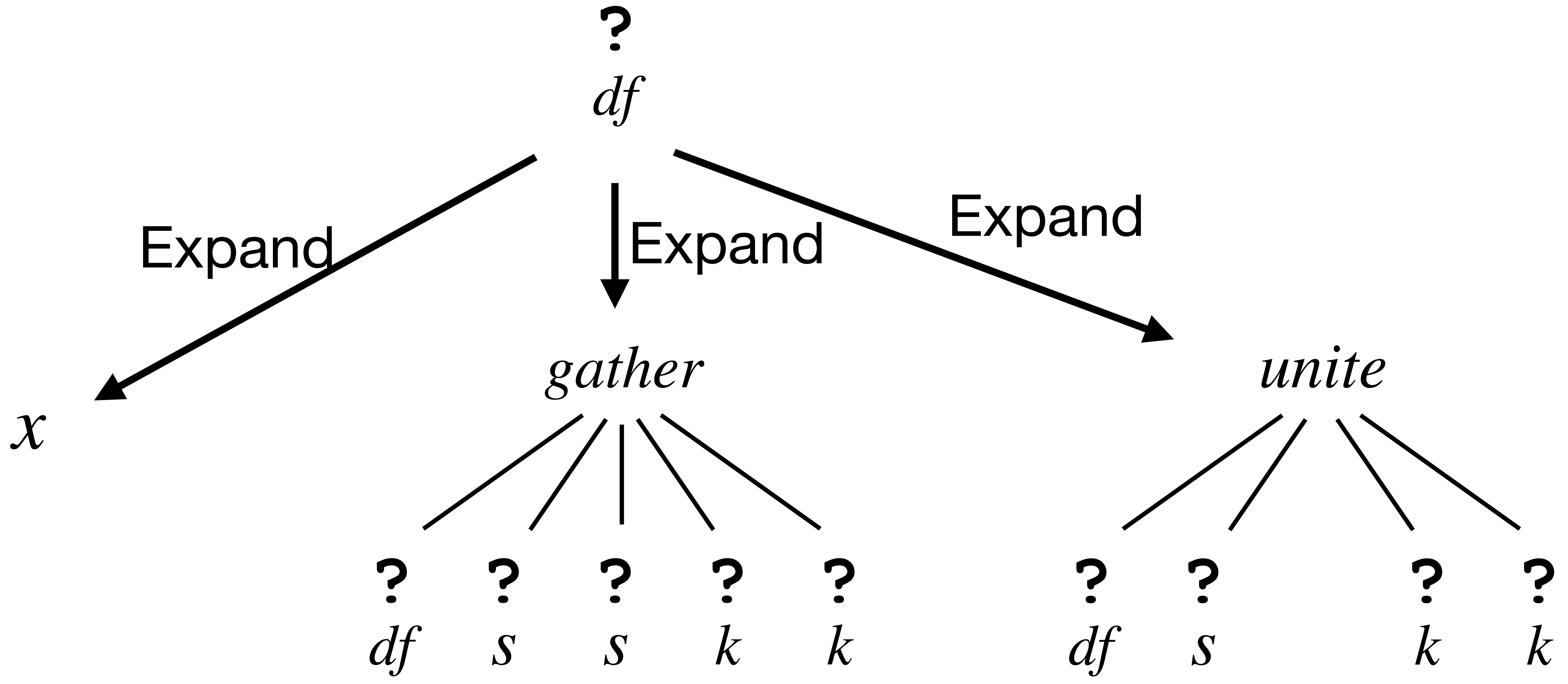
AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

$$df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$$
$$k ::= 1 \mid 2 \mid 3 \mid 4$$
$$s ::= tmp1 \mid tmp2 \mid tmp3$$

Otherwise, we pick a hole in the AST, "expand" the hole, and add new ASTs into the worklist. There may be multiple ways to expand.



Top-Down Search Algorithm

<u>**Top-Down-Search**</u> ($(T, N, P, S), E$):

worklist := { S };

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

- CFG: $e := x \mid 1 \mid e + e$
- Example: $(1{,}2)$
- Worklist (at end of iterations)

iter 0: e

iter 1: x 1 $e + e$

iter 2: 1 $e + e$

iter 3: $e + e$

iter 4: $x + e$ $1 + e$ $e + e + e$
$e + x$ $e + 1$ $e + e + e$

iter 5: $x + x$ $x + 1$ $x + e + e$
$1 + e$ $e + e + e$
$e + x$ $e + 1$ $e + e + e$

iter 6: **return** $x + x$

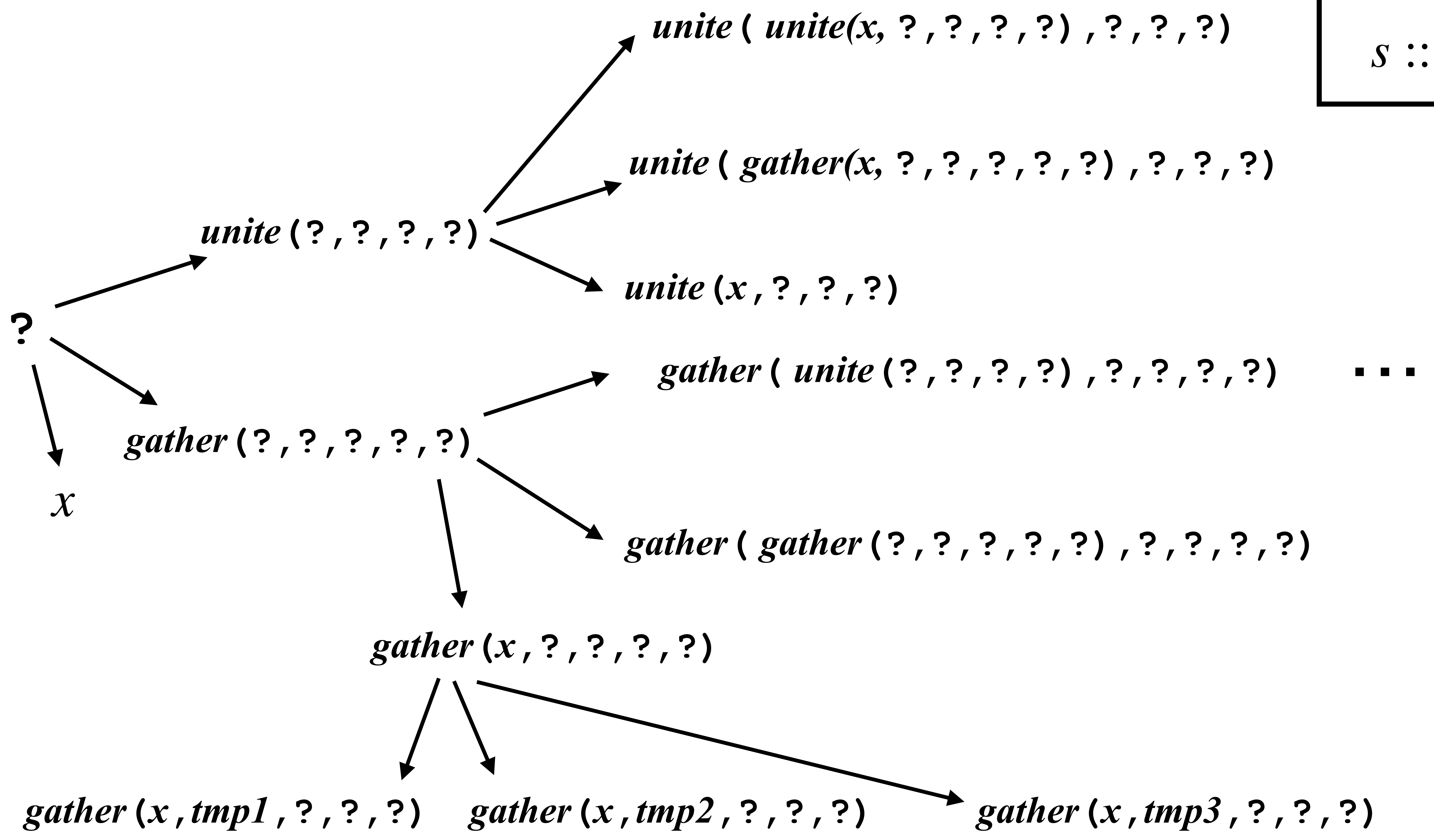
Top-Down Search Algorithm

- One way to "visualize" this algorithm:

$df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$

$k ::= 1 \mid 2 \mid 3 \mid 4$

$s ::= tmp1 \mid tmp2 \mid tmp3$



Bottom-Up Search Algorithm

- Key idea: Generate children first, then generate parents
    - First discover low(er) level components and then discover how to assemble them

x

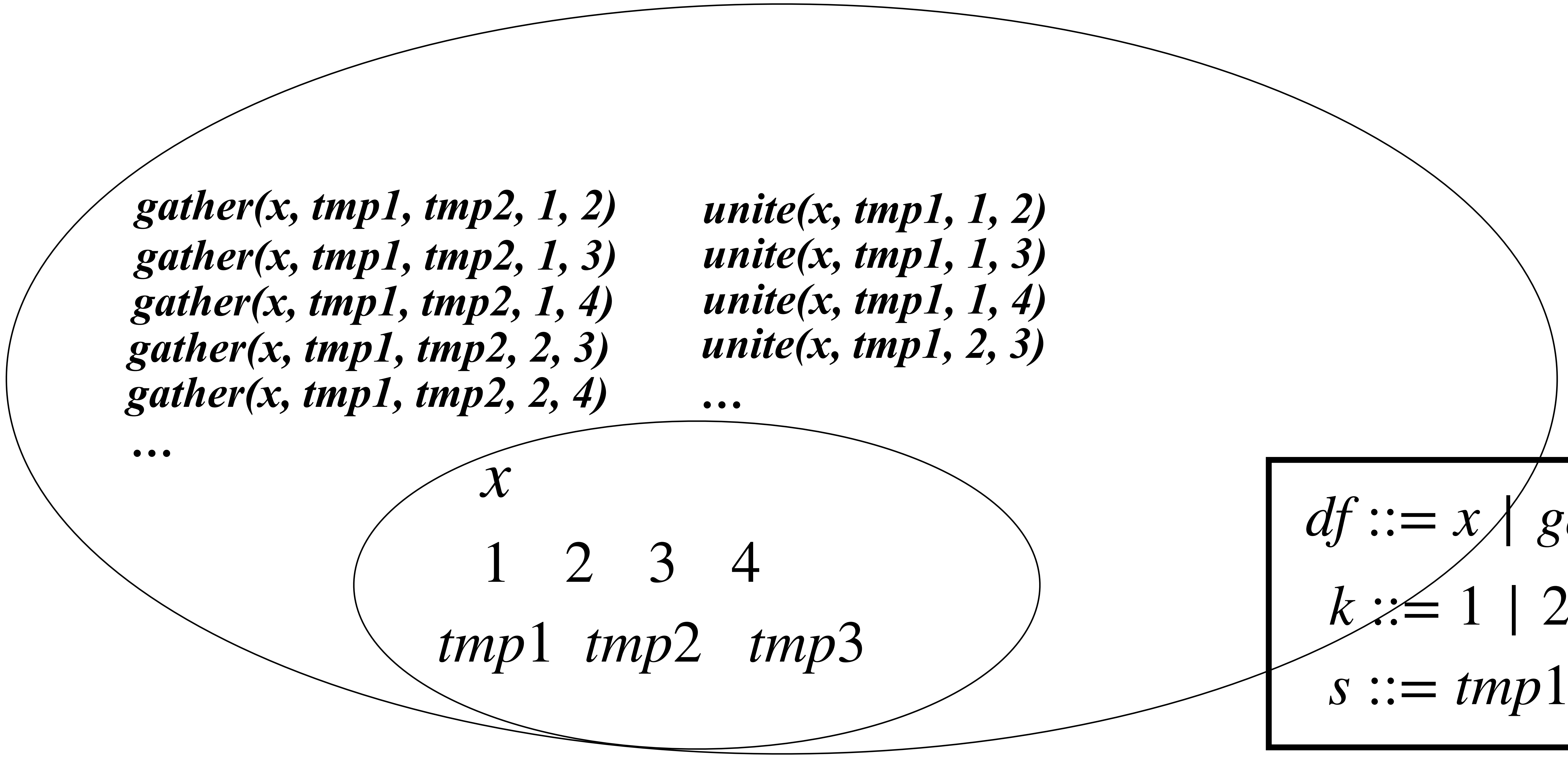
$1 \quad 2 \quad 3 \quad 4$

$tmp1 \quad tmp2 \quad tmp3$

$$df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$$
$$k ::= 1 \mid 2 \mid 3 \mid 4$$
$$s ::= tmp1 \mid tmp2 \mid tmp3$$

Bottom-Up Search Algorithm

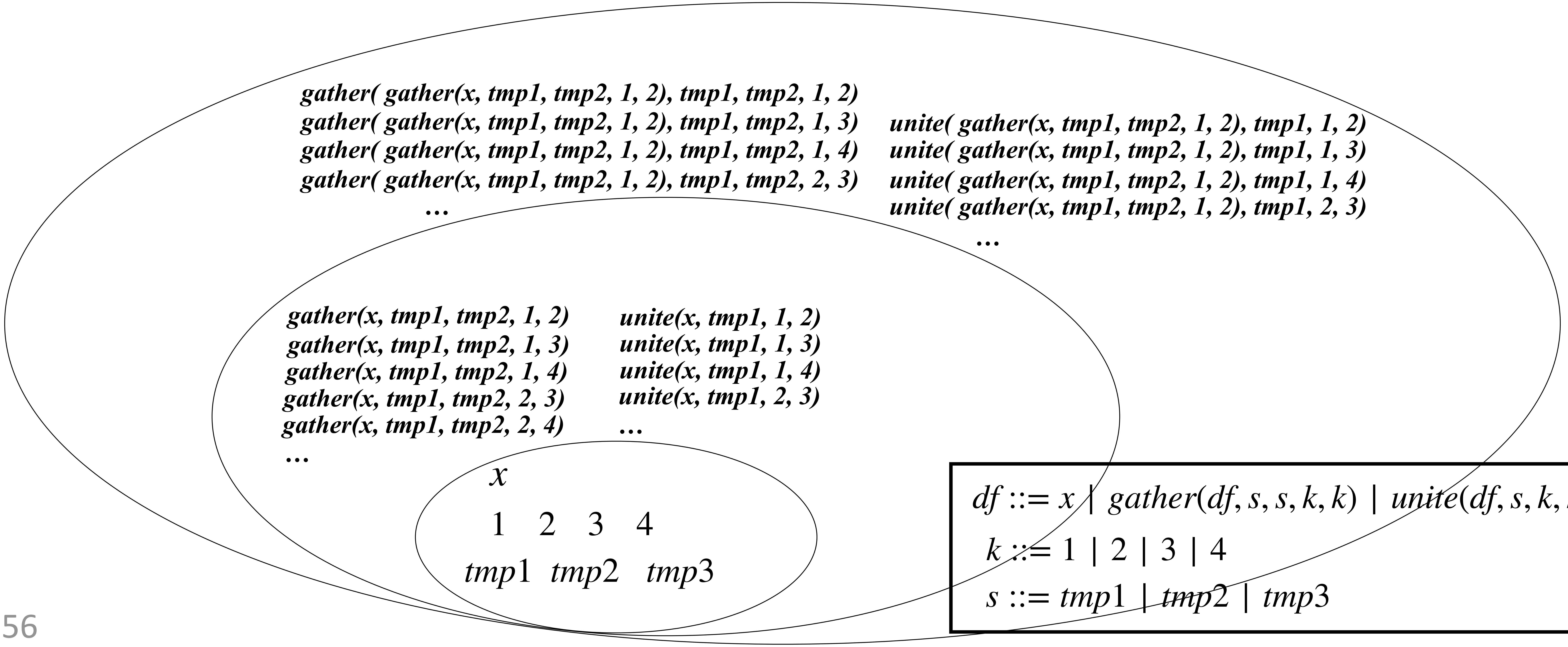
- Key idea: Generate children first, then generate parents
  - First discover low(er) level components and then discover how to assemble them



$$df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$$
$$k ::= 1 \mid 2 \mid 3 \mid 4$$
$$s ::= tmp1 \mid tmp2 \mid tmp3$$

Bottom-Up Search Algorithm

- Key idea: Generate children first, then generate parents
  - First discover low(er) level components and then discover how to assemble them



$$df ::= x \mid gather(df, s, s, k, k) \mid unite(df, s, k, k)$$
$$k ::= 1 \mid 2 \mid 3 \mid 4$$
$$s ::= tmp1 \mid tmp2 \mid tmp3$$

Bottom-Up Search Algorithm

- Given a CFG $G = (T, N, P, S)$ and a set E of examples:

<u>Bottom-Up-Search</u>($(T, N, P, S), E$):

worklist := $\{ \ t \mid t \in T \ \}$;

while (true):

foreach AST **in** *worklist*: **if** (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**grow**(*worklist*));

Summary

- Programming-by-Example
- Search space as DSL (syntax + semantics)
- Programs as ASTs
- Search Techniques: Top-Down and Bottom-Up