

# 1 Wildcard Pattern

**Definition 1.1.** A wildcard pattern  $pt$  is a tree structure for matching ASTs. Suppose the user-provided example code snippet is defined on a set of vertices  $V$ , a wildcard pattern  $pt$  is defined on  $V \cup \{*\}$ , where  $*$  is a symbol for wildcard matching. A wildcard pattern  $pt$  is defined recursively:

- $pt$  can be a terminal  $v \in V$  and matches AST terminals with the name  $v.name$ .
- $pt$  can be a wildcard symbol  $*$  and matches any AST.
- $pt$  can be a tree structure  $v(pt_1, \dots, pt_k)$ , i.e., the root is a nonterminal  $v \in V$ , and the children are wildcard patterns  $pt_1, \dots, pt_k$ . An AST  $v'$  is matched if  $v.name = v'.name$  and  $v'$  has  $k$  ordered children that can be matched by  $pt_1, \dots, pt_k$ , respectively.

**Example 1.1.** An example wildcard pattern is `expression(*, '+', *)`, where `expression` is a nonterminal and `'+'` is a terminal. This wildcard pattern matches any ASTs that are addition expressions.

Enumerating all possible wildcard patterns present in the user-given example code is difficult due to the excessive number of patterns. Consider that every subtree of an AST is a choice of being  $*$ , and these choices have diverse combinations.

Therefore, we propose to group wildcard patterns by their effects on the example code, i.e., the ASTs matched by patterns. For each group, we choose the most “explicit” pattern as the group’s representative because an explicit pattern can be turned less explicit by replacing some subtrees with  $*$ . We refer to such representative patterns as *wildcard patterns modulo example*. In the following, we formulate explicitness and the most explicit pattern in each group.

**Definition 1.2.** We define a partial order  $\preceq$  among wildcard patterns to compare their explicitness in matching ASTs. A pattern  $pt^a$  is more explicit than or equal to a pattern  $pt^b$  (denoted as  $pt^a \preceq pt^b$ ) if all the ASTs that can be matched by  $pt^a$  can also be matched by  $pt^b$ .  $pt^a \preceq pt^b$  holds if and only if any of the following holds:

- $pt^b = *$ .
- $pt^a$  and  $pt^b$  are terminals with the same name.

- $pt^a = v^a(pt_1^a, \dots, pt_k^a)$  and  $pt^b = v^b(pt_1^b, \dots, pt_k^b)$  have the same root nonterminals (i.e.,  $n^a.name = n^b.name$ ) and  $pt_i^a \preceq pt_i^b$  for  $i = 1, \dots, k$ .

**Theorem 1.1.** *The  $\preceq$  relation on wildcard patterns is a partial order.*

*Proof.* First,  $\preceq$  is reflexive:

- If  $pt = *$ , then  $pt \preceq pt$  by definition.
- If  $pt$  is a terminal, then  $pt \preceq pt$  by definition.
- Otherwise,  $pt = v(pt_1, \dots, pt_k)$ , then  $pt \preceq pt$  holds if and only if  $pt_i \preceq pt_i$  for  $i = 1, \dots, k$ , which is proved recursively.

Also,  $\preceq$  is antisymmetric. Supposed  $pt^a \preceq pt^b$  and  $pt^b \preceq pt^a$ .

- If  $pt^a = *$  and  $pt^a \preceq pt^b$ , by definition,  $pt^b$  can only be  $*$ , so  $pt^a = pt^b$ .
- If  $pt^a$  and  $pt^b$  are terminals with the same name,  $pt^a = pt^b$  obviously.
- Otherwise,  $pt^a = v^a(pt_1^a, \dots, pt_k^a)$ ,  $pt^b = v^b(pt_1^b, \dots, pt_k^b)$ , and  $v^a.name = v^b.name$ . Also,  $pt_i^a \preceq pt_i^b \wedge pt_i^b \preceq pt_i^a$  for  $i = 1, \dots, k$ . As proved recursively,  $pt_i^a = pt_i^b$  for  $i = 1, \dots, k$ . Hence,  $pt^a = pt^b$ .

Finally,  $\preceq$  is transitive, i.e.,  $pt^a \preceq pt^b \wedge pt^b \preceq pt^c \Rightarrow pt^a \preceq pt^c$ .

- If  $pt^c = *$ , then  $pt^a \preceq pt^c$  by definition.
- If  $pt^c$  is a terminal, then we must have  $pt^c = pt^b = pt^a$ .
- Otherwise,  $pt^c = v^c(pt_1^c, \dots, pt_k^c)$ ,  $pt^b = v^b(pt_1^b, \dots, pt_k^b)$ , and  $pt^a = v^a(pt_1^a, \dots, pt_k^a)$ . We must have  $v^c.name = v^b.name = v^a.name$  and  $pt_i^a \preceq pt_i^b \wedge pt_i^b \preceq pt_i^c$  for  $i = 1, \dots, k$ . Then  $pt_i^a \preceq pt_i^c$  for  $i = 1, \dots, k$ , which is proved recursively.

□

**Definition 1.3.** We define a join operation  $\vee$  such that  $pt^a \vee pt^b$  is a wildcard pattern that covers the ASTs matched by the inputted wildcard patterns  $pt^a$  and  $pt^b$ . Specifically,  $pt^a \vee pt^b$  is defined as follows:

- $pt^a \vee pt^b = *$ , if either  $pt^a$  or  $pt^b$  is  $*$ , or the root nodes in  $pt^a$  and  $pt^b$  have different names.
- If  $pt^a$  and  $pt^b$  are terminals with the same name,  $pt^a \vee pt^b = pt^a = pt^b$ .

- In the case  $pt^a = v^a(pt_1^a, \dots, pt_k^a)$ ,  $pt^b = v^b(pt_1^b, \dots, pt_k^b)$ , and  $v^a = v^b = v$ ,  $pt^a \vee pt^b = v(pt_1^a \vee pt_1^b, \dots, pt_k^a \vee pt_k^b)$ .

**Theorem 1.2.** *Wildcard patterns partially ordered by  $\preceq$  is a join-semilattice under  $\vee$ .*

*Proof.* We can show that  $pt^a \preceq pt^a \vee pt^b$  with the previous two conditions (the pattern is  $*$  or a terminal) as the base conditions. For the condition where  $pt^a = v^a(pt_1^a, \dots, pt_k^a)$ ,  $pt^b = v^b(pt_1^b, \dots, pt_k^b)$ , and  $v^a = v^b$ , we can show  $pt_i^a \preceq pt_i^a \vee pt_i^b$  for  $i = 1, \dots, k$  recursively.

Similarly, we have  $pt^b \preceq pt^a \vee pt^b$ .

Furthermore, we can show that if the least upper bound of  $pt^a$  and  $pt^b$  is  $l$ , then  $l = pt^a \vee pt^b$ .

For the two base conditions, the result is obvious.

For the condition where  $pt^a = v^a(pt_1^a, \dots, pt_k^a)$ ,  $pt^b = v^b(pt_1^b, \dots, pt_k^b)$ , and  $v^a = v^b = v$ ,  $l$  must have the form  $v(l_1, \dots, l_k)$ , and  $l = pt^a \vee pt^b$  if and only if  $l_i = pt_i^a \vee pt_i^b$  for  $i = 1, \dots, k$ , which can be proved recursively.  $\square$

We have defined the concept of explicitness ( $\preceq$ ) and the most explicit pattern covering a group of patterns ( $\vee$ ). We give an iterative algorithm (Algorithm 1) that mines the wildcard patterns modulo example. We show the properties of Algorithm 1 in the remainder of this section.

---

**Algorithm 1** Mining wildcard patterns modulo example

---

```

1: function MINELUBS( $I$ )
2:    $\Sigma \leftarrow \{\text{PATTERN}(v) \mid v \in I.V\}$        $\triangleright \Sigma$  is the accumulated wildcard
      patterns.
3:    $L \leftarrow \Sigma$                                  $\triangleright L$  is the newly mined wildcard patterns.
4:   while  $L \neq \emptyset$  do
5:      $L' \leftarrow \{u \vee v \mid u \in L, v \in L \cup \Sigma\}$ 
6:      $L \leftarrow L' - \Sigma$ 
7:      $\Sigma \leftarrow \Sigma \cup L$ 
8:   return  $\Sigma$ 

```

---

**Theorem 1.3.** *Algorithm 1 is bound to terminate.*

*Proof.* As shown in Theorem 1.2, the  $u \vee v$  returns the least upper bound of  $u$  and  $v$ , and  $\Sigma$  is initialized by the explicit AST subtrees in  $I$ , which are also least upper bounds of themselves. Hence,  $\Sigma$  stores at most all the least upper bounds of the AST subsets in  $I$ , which is finite. After each

iteration,  $L$  is either updated with unseen wildcard patterns, or becomes empty. If  $L$  becomes empty, the algorithm terminates. If  $L$  is updated with unseen wildcard patterns, the algorithm will terminate after a finite number of iterations.  $\square$

**Theorem 1.4.** *Algorithm 1 is sound (i.e., all the discovered patterns are LUBs of some AST subtrees in  $I$ ) and complete (i.e., no LUB or equivalent class is missed).*

*Proof.* (Soundness)  $\Sigma$  is initialized with the LUBs for each AST subtree in  $I$  (LUB for a subtree is the subtree itself), and  $u \vee v$  returns the LUB of  $u$  and  $v$ , which is the LUB of the subsets of ASTs represented by  $u$  and  $v$ . Therefore, all the wildcard patterns in  $\Sigma$  are LUBs of some AST subtrees in  $I$ .

(Completeness) For any non-empty subset  $\{v_1, \dots, v_k\} \subseteq I.V$ , the corresponding least upper bound must be in  $\Sigma$ . The iteration at line 5 of Algorithm 1 ensures that  $v_1 \vee v_2, v_1 \vee v_2 \vee v_3, \dots, v_1 \vee \dots \vee v_k$  are all computed. Theorem 1.2 shows that  $a \vee b$  is the least upper bound of  $a$  and  $b$ . Hence, the least upper bound of  $\{v_1, \dots, v_k\}$  must be in  $\Sigma$ .  $\square$

## 2 Prompt

This section listed two prompts and their few-shot examples used in the experiment. The experiments in the evaluation adopted the Recall-First prompt. The case study in the discussion compared the Precision-First prompt with the Recall-First prompt.

### 2.1 Recall First

You are a Java expert and user assistant. The user would describe what kind of codes they want to extract. Your task is to analyze the user task and classify some code snippets according to their relations with the user task. Below are some examples.

### User Task

“arithmetic expressions that do not involve integers”

### Example Code Snippet

...

```
public class Foo {
    public void test() {
```

```

        int a = 12 * 3;
        double b = 2.3 - 8.6;
        double c = a / b;
        double d = b + b;
    }
}
...

```

The matched codes are ` 2.3 - 8.6 `, ` b + b `.

### ### Assistant's Task

The user-desired codes would be extracted by an XPath-like program, which navigates through AXE nodes to reach the desired nodes. An AXE node usually matches some part of the user description but NEEDS NOT match ALL. It can sometimes be the opposite. Your task is to first identify the AXE constructs from the user description and list them one by one. Then, you need to read each given code, identify its syntactic type (e.g., expression, statement), and compare the code with the AXEs you identified above. Finally, classify the nodes as one of the following:

- `Axe (x)` denotes the code is exactly the (x) Axe construct you identify.
- `Outer` denotes the code is an outer scope of target codes or axes.
- `Inner` denotes the code is an inner part of target codes or axes.
- `Irrelevant` denotes the code has no relevance to the target codes or axes.

Nodes to classify:

- (1) ` 12 `
- (2) ` 12 \* 3 `
- (3) ` c `
- (4) ` b + b `
- (5) ` 8.6 `

### ### Answer

According to the user task “arithmetic expressions that do not involve integers”, an axe construct can be any one of:

- (a) arithmetic expressions
- (b) integer expressions
- (c) arithmetic operators

The classifications:

- (1) ` 12 ` is an expression. It is integer-type, so it matches the axe => Axe (b).
- (2) ` 12 \* 3 ` is an arithmetic expression. Albeit it is on integers, it still matches the axe listed above => Axe (a) (b).
- (3) ` c ` is a declared id. It is with double type. It does not match any axe definition => Irrelevant.
- (4) ` + ` is an arithmetic operator. It matches the axe => Axe (c).

(5) ` 8.6 ` is an expression. It is a double expression and nested in target codes, but it does not match the axe listed above => Inner.

---

### User Task

<user-given description>

### Example Code Snippet

``` <example input> ```

The matched codes are <example outputs>.

### Assistant's Task

The user-desired codes would be extracted by an XPath-like program, which navigates through AXE nodes to reach the desired nodes. An AXE node usually matches some part of the user description but NEEDS NOT match ALL. It can sometimes be the opposite. Your task is to first identify the AXE constructs from the user description and list them one by one. Then, you need to read each given code, identify its syntactic type (e.g., expression, statement), and compare the code with the AXEs you identified above. Finally, classify the nodes as one of the following:

    `Axe (x)` denotes the code is exactly the (x) Axe construct you identify.

    `Outer` denotes the code is an outer scope of target codes or axes.

    `Inner` denotes the code is an inner part of target codes or axes.

    `Irrelevant` denotes the code has no relevance to the target codes or axes.

Nodes to classify:

    (1) <code fragment 1>

    (2) <code fragment 2>

    ...

### Answer

## 2.2 Precision First

You are a Java expert and user assistant. The user would describe what kind of codes they want to extract. Your task is to analyze the user task and classify some code snippets according to their relations with the user task. Below are some examples.

### User Task

“toString calls and hashCode calls on Array objects”

### Example Code Snippet

```

import java.util.Arrays;

```

public class ArrayHashCodeAndToStringProcessor {
    public void test() {
        int[] args = {1,2,3};
        String argStr1 = args.toString();
        int argHash1 = args.hashCode();
        String argStr2 = Arrays.toString(args);
        int argHash2 = Arrays.hashCode(args);
    }
}
...

```

The matched codes are `args.toString()`, `args.hashCode()`.

### ### Assistant's Task

The user desired codes would be extracted by an XPath-like program, which navigates through AXE nodes to reach the desired nodes. An AXE node usually matches some part of the user description but need not match all. It can sometimes be the opposite. Your task is to first identify the AXE constructs from the user description (list them one by one), and then classify each given code as one of the followings and explain your reason BEFORE your decision.

- `Axe` denotes the code is exactly one of the axe construct you identify.
- `Outer` denotes the code is an outer scope of target codes or axes.
- `Inner` denotes the code is an inner part of target codes or axes.
- `Irrelevant` denotes the code has no relevance to the target codes or axes.

Nodes to classify:

- (1) `int[] args`
- (2) `String argStr1 = args.toString();`
- (3) `toString()`
- (4) `toString`
- (5) `Arrays.hashCode(args)`

### ### Answer

According to the user task "toString calls and hashCode calls on Array objects", the axe constructs should include three parts:

- (a) toString calls.
- (b) hashCode calls.
- (c) Array objects.

The classifications:

- (1) `int[] args` declares an Array object, but this object does not appear in the ast of hashCode/toString calls. => Irrelevant.
- (2) `String argStr1 = args.toString();` is a statement containing the target call, but is not necessary to be visited in the XPath. => Outer.

(3) `` toString() `` matches “toString” call in the task and needs to be visited in the XPath. => Axe.

(4) `` toString `` is an identifier that is part of the target calls, but the XPath needs only visiting `'toString()'` => Inner.

(5) `` Arrays.hashCode(args) `` is called via static class instead of Array object. => Irrelevant.

---

### User Task

“arithmetic expressions that do not involve integers”

### Example Code Snippet

...

```
public class Foo {  
    public void test() {  
        int a = 12 * 3;  
        double b = 2.3 - 8.6;  
        double c = a / b;  
        double d = b + b;  
    }  
}
```

...

The matched codes are `` 2.3 - 8.6 ``, `` b + b ``.

### Assistant's Task

The user desired codes would be extracted by an XPath-like program, which navigates through AXE nodes to reach the desired nodes. An AXE node usually matches some part of the user description but need not matches all. It can sometimes be the opposite. Your task is to first identify the AXE constructs from the user description (list them one by one), and then classify each given code as one of the followings and explain your reason BEFORE your decision.

`` Axe `` denotes the code is exactly one of the axe construct you identify.

`` Outer `` denotes the code is an outer scope of target codes or axes.

`` Inner `` denotes the code is an inner part of target codes or axes.

`` Irrelevant `` denotes the code has no relevance to the target codes or axes.

Nodes to classify:

- (1) `` 12 ``
- (2) `` 12 * 3 ``
- (3) `` c ``
- (4) `` b + b ``
- (5) `` 8.6 ``

### Answer



According to the user task “arithmetic expressions that do not involve integers”, the axe constructs should include two parts:

- (a) arithmetic expressions
- (b) integer expressions

The classifications:

- (1) `` 12 `` is an integer expression and thus the axe => Axe.
- (2) `` 12 * 3 `` is an arithmetic expression albeit on integers. It still matches the axe listed above => Axe.
- (3) `` c `` is a declared id with double type. It does not match any axe definition => Irrelevant.
- (4) `` + `` is an arithmetic operator and what we use to identify arithmetic expressions => Axe.
- (5) `` 8.6 `` is a double expression and nested in target codes, but it does not match the axe listed above => Inner.

---

### User Task

`<user-given description>`

### Example Code Snippet

```` <example input> ````

The matched codes are `<example outputs>`.

### Assistant’s Task

The user desired codes would be extracted by an XPath-like program, which navigates through AXE nodes to reach the desired nodes. An AXE node usually matches some part of the user description but need not matches all. It can sometimes be the opposite. Your task is to first identify the AXE constructs from the user description (list them one by one), and then classify each given code as one of the followings and explain your reason BEFORE your decision.

- ``Axe`` denotes the code is exactly one of the axe construct you identify.
- ``Outer`` denotes the code is an outer scope of target codes or axes.
- ``Inner`` denotes the code is an inner part of target codes or axes.
- ``Irrelevant`` denotes the code has no relevance to the target codes or axes.

Nodes to classify:

- (1) `<code fragment 1>`
- (2) `<code fragment 2>`
- ...

### Answer