# Ekeko/X matching algorithm

02/05/'18

Tim Molderez

VRIJE
UNIVERSITEIT
BRUSSEL

# Ekeko/X template groups

```
[public void acceptVisitor(ComponentVisitor v) {
    ...
}]@[(equals ?method)]

[....acceptVisitor(...)]@[(invokes ?method)]
```

```
public class PrototypicalLeaf extends Component {
    public void acceptVisitor(ComponentVisitor v) {
        System.out.println("Prototypical.");
        v.visitPrototypicalLeaf(this);
    }
}
```

```
...
cs.acceptVisitor(vstor);
...
```

# Ekeko/X template groups

Wildcard

```
[public void acceptVisitor(ComponentVisitor v) {
    ...
}]@[(equals ?method)]

[....acceptVisitor(...)]@[(invokes ?method)]
```

```
public class PrototypicalLeaf extends Component {
    public void acceptVisitor(ComponentVisitor v) {
        System.out.println("Prototypical.");
        v.visitPrototypicalLeaf(this);
    }
}
```

```
...
cs.acceptVisitor(vstor);
...
```

# Ekeko/X template groups

```
[public void acceptVisitor(ComponentVisitor v) {
    ...
}]@[(equals ?method)]

[....acceptVisitor(...)]@[(invokes ?method)]
```

```
public class PrototypicalLeaf extends Component {
    public void acceptVisitor(ComponentVisitor v) {
        System.out.println("Prototypical.");
        v.visitPrototypicalLeaf(this);
    }
}
```

```
...
cs.acceptVisitor(vstor);
...
```

# Ekeko/X template groups

```
[public void acceptVisitor(ComponentVisitor v) {
    ...
}]@[(equals ?method)]
```

*Directive*

*Equals directive binds subject to logic variable*

```
[...acceptVisitor(...)]@[(invokes ?method)]
```

```
public class PrototypicalLeaf extends Component {
    public void acceptVisitor(ComponentVisitor v) {
        System.out.println("Prototypical.");
        v.visitPrototypicalLeaf(this);
    }
}
```

```
...
cs.acceptVisitor(vstor);
...
```

# Ekeko/X template groups

```
[public void acceptVisitor(ComponentVisitor v) {
    ...
}]@[(equals ?method)]

[....acceptVisitor(...)]@[(invokes ?method)]
```

```
public class PrototypicalLeaf extends Component {
    public void acceptVisitor(ComponentVisitor v) {
        System.out.println("Prototypical.");
        v.visitPrototypicalLeaf(this);
    }
}
```

```
...
cs.acceptVisitor(vstor);
...
```

# Ekeko/X template groups

```
[public void acceptVisitor(ComponentVisitor v) {
    ...
}]@[(equals ?method)]

[....acceptVisitor(...)]@[(invokes ?method)]
```

*Only matches if subject is a call to ?method*

```
public class PrototypicalLeaf extends Component {
    public void acceptVisitor(ComponentVisitor v) {
        System.out.println("Prototypical.");
        v.visitPrototypicalLeaf(this);
    }
}
```

```
...
cs.acceptVisitor(vstor);
...
```

# Match|set directive

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;

=>

private @EntityProperty(value=Expression.class) EntityIdentifier<Expression> array;
```

# Match|set directive

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;

=>

private @EntityProperty(value=Expression.class) EntityIdentifier<Expression> array;
```

```
[@EntityProperty(value=?annoType.class) private]@[match|set] [EntityIdentifier]
  @[(equals ?fieldType)] ...;

=>

[EntityIdentifier<?annoType>]@[(replace ?fieldType)]
```

# Match|set directive

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;

=>

private @EntityProperty(value=Expression.class) EntityIdentifier<Expression> array;
```

*Can appear in any order; can have more elements*

```
[@EntityProperty(value=?annoType.class) private]@[match|set] [EntityIdentifier]
  @[(equals ?fieldType)] ...;

=>

[EntityIdentifier<?annoType>]@[(replace ?fieldType)]
```

# Child* directive

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;

=>

private @EntityProperty(value=Expression.class) EntityIdentifier<Expression> array;
```

# Child* directive

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;

=>

private @EntityProperty(value=Expression.class) EntityIdentifier<Expression> array;
```

```
[@...(value=?annoType.class) private]@[match|set] [EntityIdentifier]
  @[child* (equals ?fieldType)] ?field;


=>

[EntityIdentifier<?annoType>]@[(replace ?fieldType)]
```

# Child* directive

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;

=>

private @EntityProperty(value=Expression.class) EntityIdentifier<Expression> array;
```

*Matches if subject is descendant of parent node*

```
[@...(value=?annoType.class) private]@[match|set] [EntityIdentifier]
  @[child* (equals ?fieldType)] ?field;


=>

[EntityIdentifier<?annoType>]@[(replace ?fieldType)]
```

# Child* directive

```
private @EntityProperty(value=Expression.class) List<EntityIdentifier> array;

=>

private @EntityProperty(value=Expression.class) List<EntityIdentifier<Expression>> array;
```

*Matches if subject is descendant of parent node*

```
[@...(value=?annoType.class) private]@[match|set] [EntityIdentifier]
  @[child* (equals ?fieldType)] ?field;


=>

[EntityIdentifier<?annoType>]@[(replace ?fieldType)]
```

# Child* directive

```
private @EntityProperty(value=Expression.class) List<EntityIdentifier> array;

=>

private @EntityProperty(value=Expression.class) List<EntityIdentifier<Expression>> array;
```

*Matches if subject is descendant of parent node*

```
[@...(value=?annoType.class) private]@[match|set] [EntityIdentifier]
  @[child* (equals ?fieldType)] ?field;


=>

[EntityIdentifier<?annoType>]@[(replace ?fieldType)]
```

# Match|regexp directive

```
public void ...(...) {[
  [...]@[multiplicity-*]
  if(...) { ... }
  ?stmt
  [...]@[multiplicity-*]
]@[match|regexp]}
```

# Match|regexp directive

```
public void ...(...) {[
  [...]@[multiplicity-*]
  if(...) { ... }
  ?stmt
  [...]@[multiplicity-*]
]@[match|regexp]}
```

*Interpret list subject as a regex*

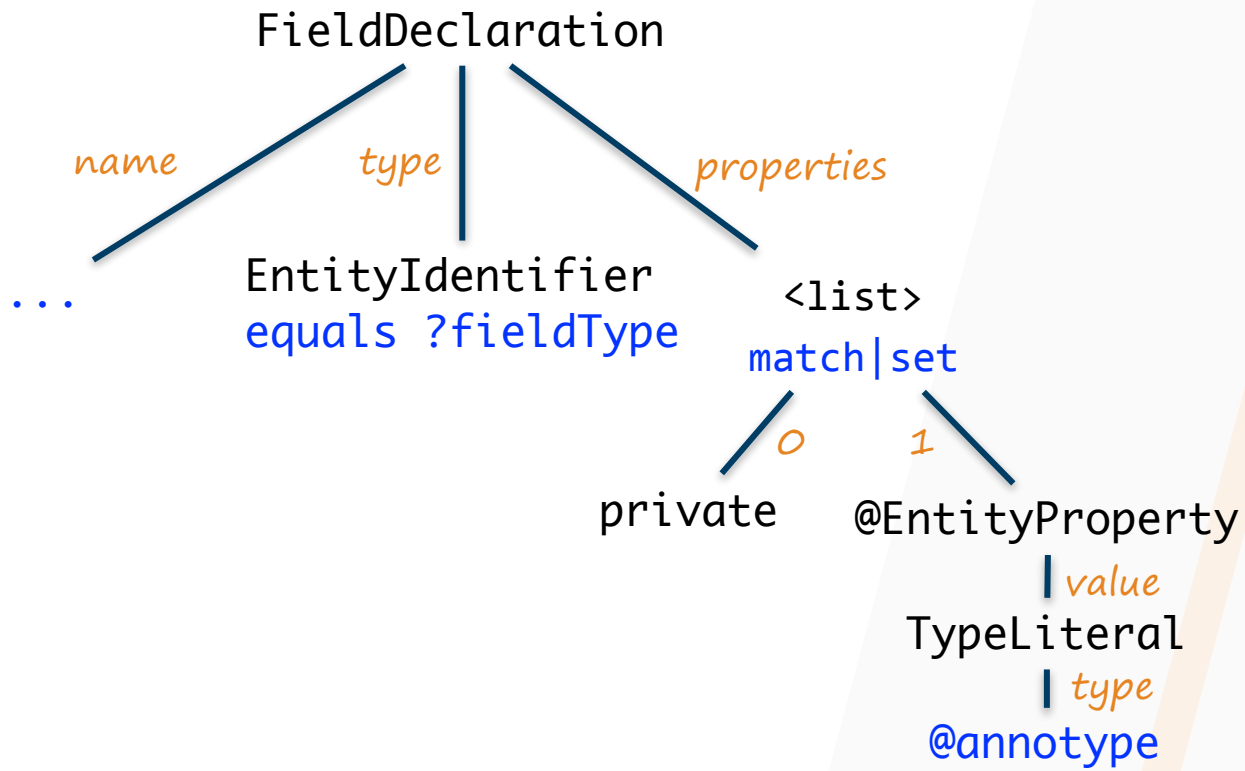
*Find the ?stmt right after an if statement*

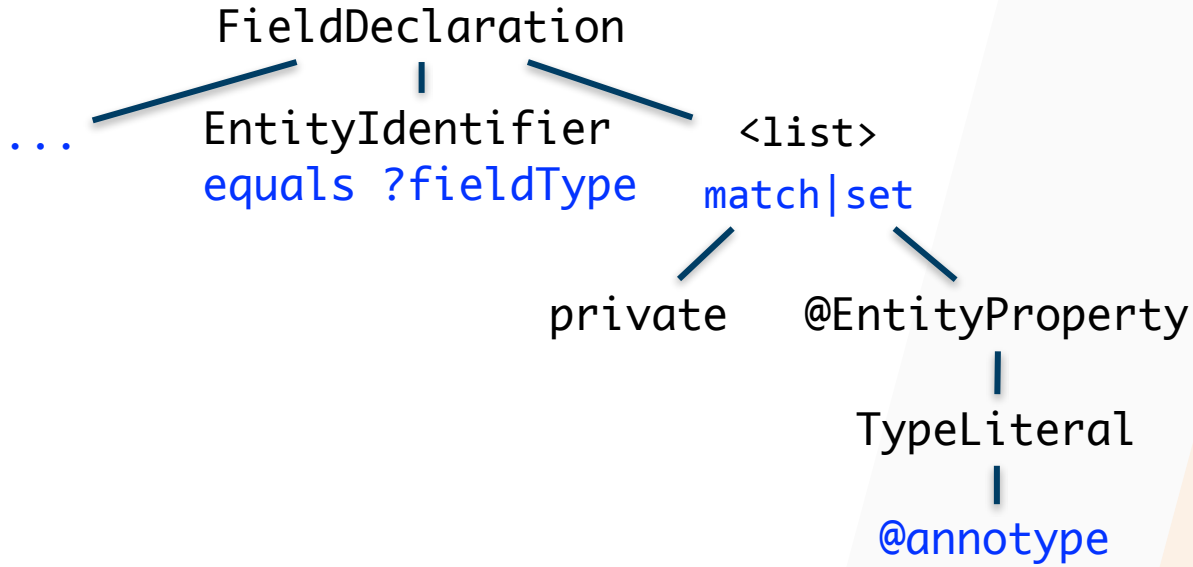
```
?modList class ?className {
[  [@...(value=?annoType.class) private]@[match|set] [EntityIdentifier]@[child* (equals ?fieldType)] ?
field;

  public [EntityIdentifier]@[child* (equals ?returnType)] ?getterName() {
    return [?returned]@[(refers-to ?field)];
  }

  public void ?setterName(  [EntityIdentifier]@[child* (equals ?paramType)] ?param) {
    [?assignee]@[(refers-to ?field)]=[?assigned]@[(refers-to ?param)];
  }
]@[match|set]
}

=>

[EntityIdentifier<?annoType>]@[(replace ?fieldType)]

[EntityIdentifier<?annoType>]@[(replace ?paramType)]

[EntityIdentifier<?annoType>]@[(replace ?returnType)]
```

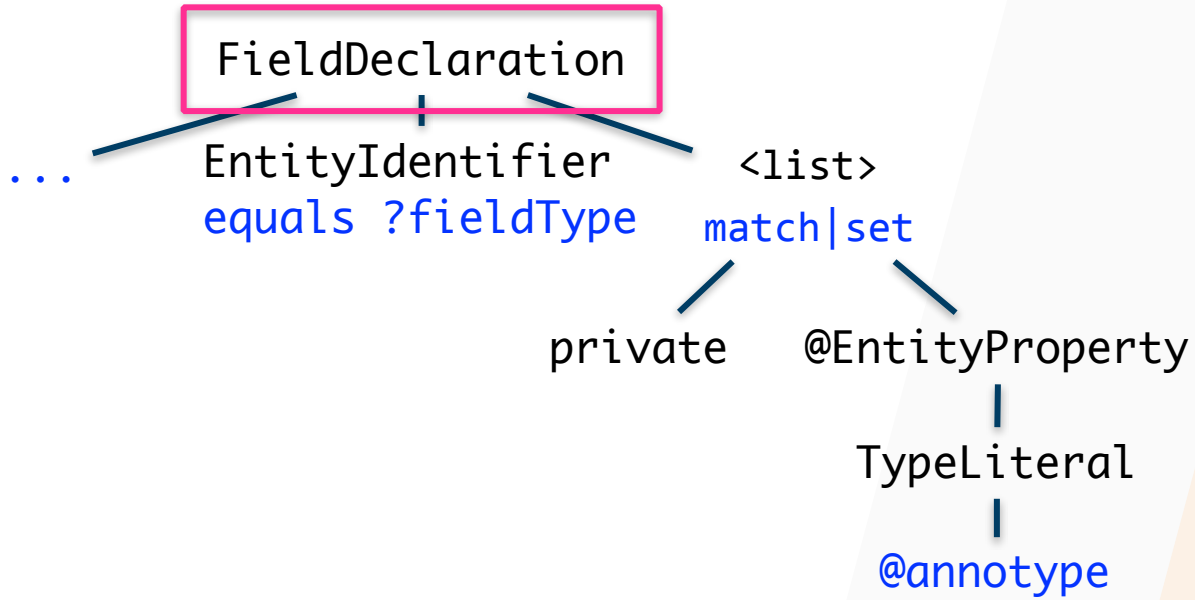
| Directive signature | Subject | Description |
|---|---|---|
| `child,child+,child*` | Any | Relates the subject node to its parent template node $x$. The matching node of $x$ is the parent (`child`) / indirect ancestor (`child+`) / ancestor (`child*`) of the subject's matching node. |
| `(equals ?var)` | Any | The subject now unifies with the given metavariable. |
| `match` | Any | Checks that the subject node type and its properties correspond to those of the matching node. |
| `match\|set` | List | The list elements of the subject must also appear (in any order) in the matching node's list elements. |
| `(type ?type), (type\|sname <str>), (type\|qname <str>)` | Type, variable declaration/reference or expression | The matching node should resolve to or declare the given type. (specified as either a metavariable, its simple name or its qualified name) |
| `(subtype+/* ?type), (subtype\|sname+/* <str>), (subtype\|qname+/* <str>),` | Type, variable declaration/reference or expression | The matching node should resolve to or declare a (reflexive) transitive subtype of the given type. |
| `(refers-to ?var)` | Identifier in method body | Matching node lexically refers to a local variable, parameter or field denoted by the argument. |
| `(referred-by ?expr)` | Field/var. decl. or formal method parameter | Matching node declares a local variable, parameter or field lexically referred to by `?expr`. |
| `(invokes ?method), (invokes\|qname <string>)` | Method call | Matching node is an invocation that calls the given method considering the receiver's static type. |
| `(invoked-by ?call)` | Method declaration | Inverse of the above: matching node is a method declaration that was invoked by `?inv`. |
| `(constructed-by ?ctor)` | Constructor | Matching node is a constructor that was invoked by `?ctor` instantiation. |
| `(constructs ?ctor)` | Instantiation expression | Matching node is an instantiation that invokes the constructor `?ctor`. |
| `(overrides ?methdecl)` | Method declaration | Matching node is a method declaration that overrides the `?methdecl` declaration. |
| `protect` | Any | Prevents operators from removing or abstracting away this node. |

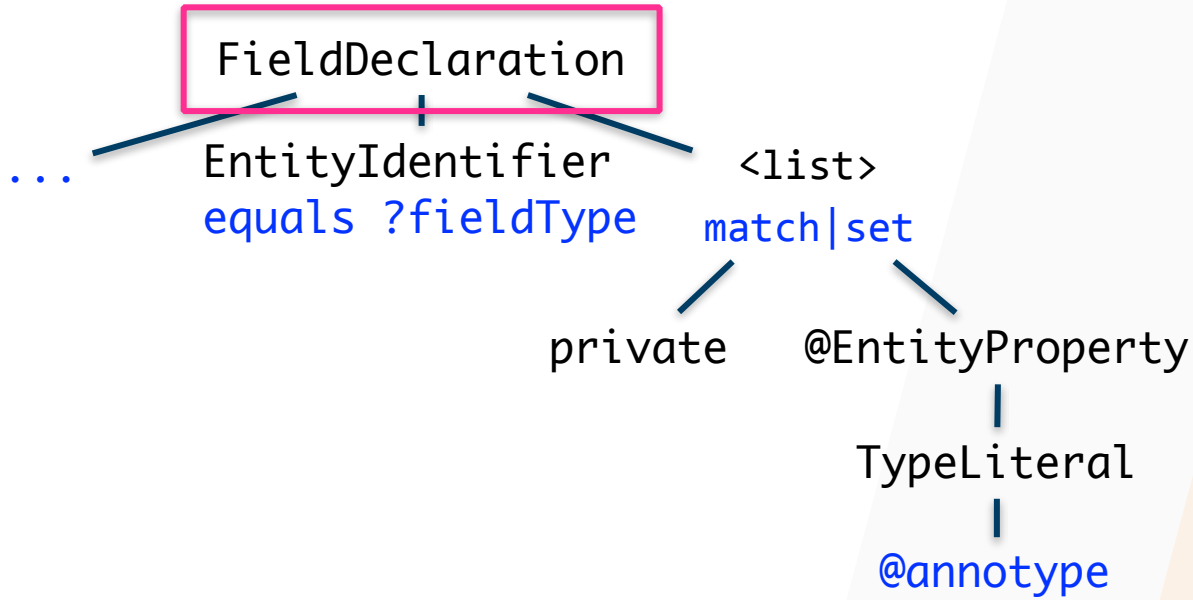
# Template tree representation

```
[@EntityProperty(value=?annoType.class) private]@[match|set] [EntityIdentifier]
  @[(equals ?fieldType)] ...;
```



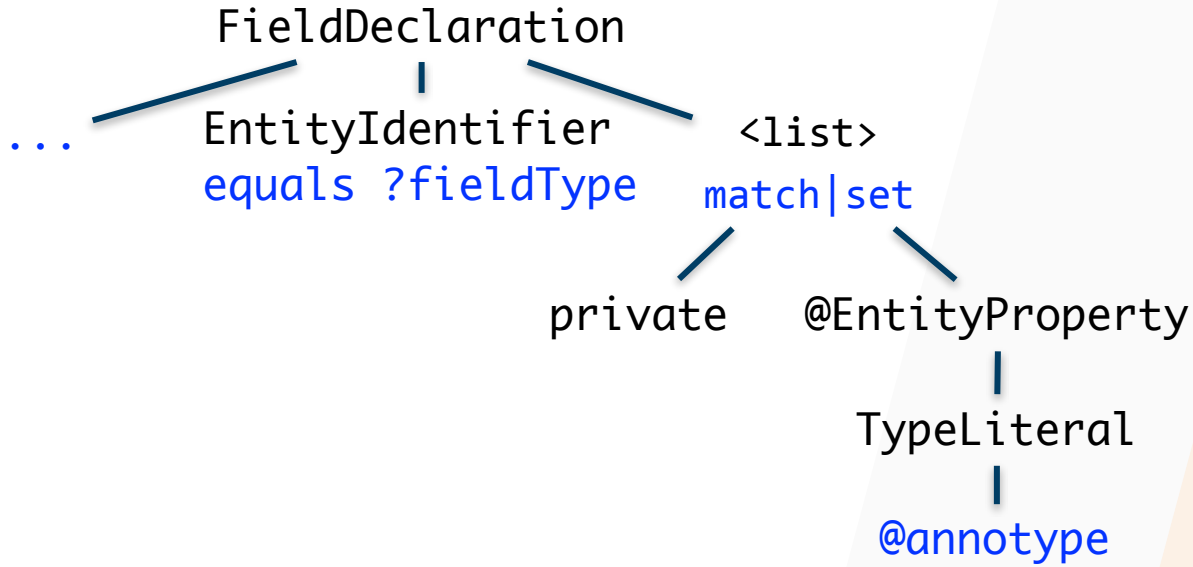

FieldDeclaration
...
EntityIdentifier
equals ?fieldType
<list>
match|set
private
@EntityProperty
TypeLiteral
@annotype


FieldDeclaration
...
EntityIdentifier
equals ?fieldType
<list>
match|set
private
@EntityProperty
TypeLiteral
@annotype



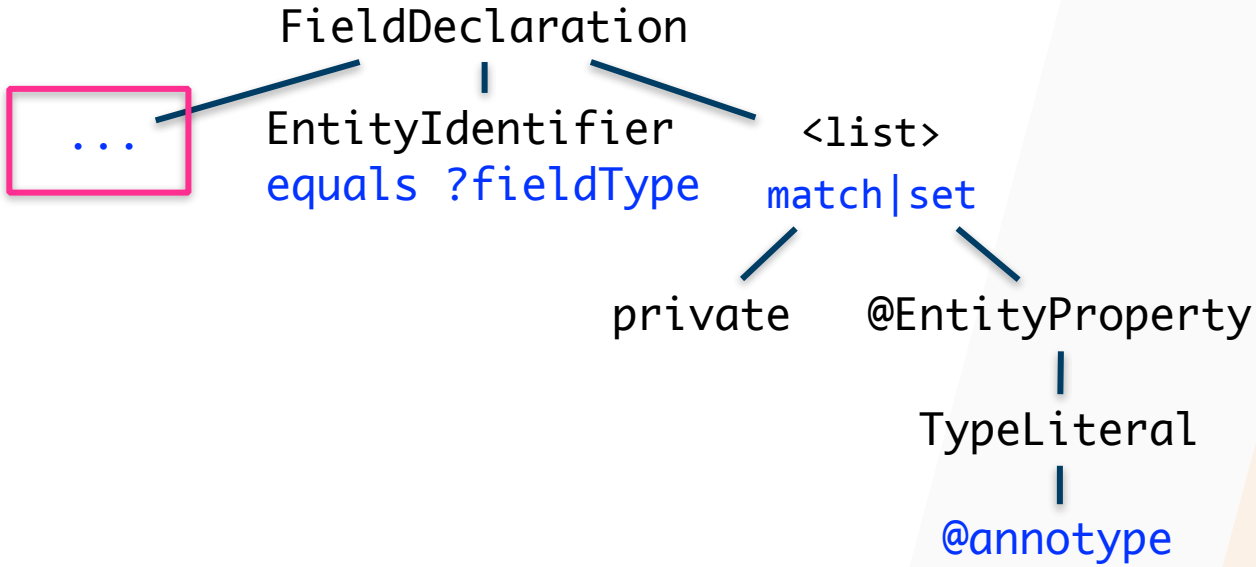

*Potential matches; all field declarations:*

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;

public Account acc;

private static EntityIdentifier ei;

...
```



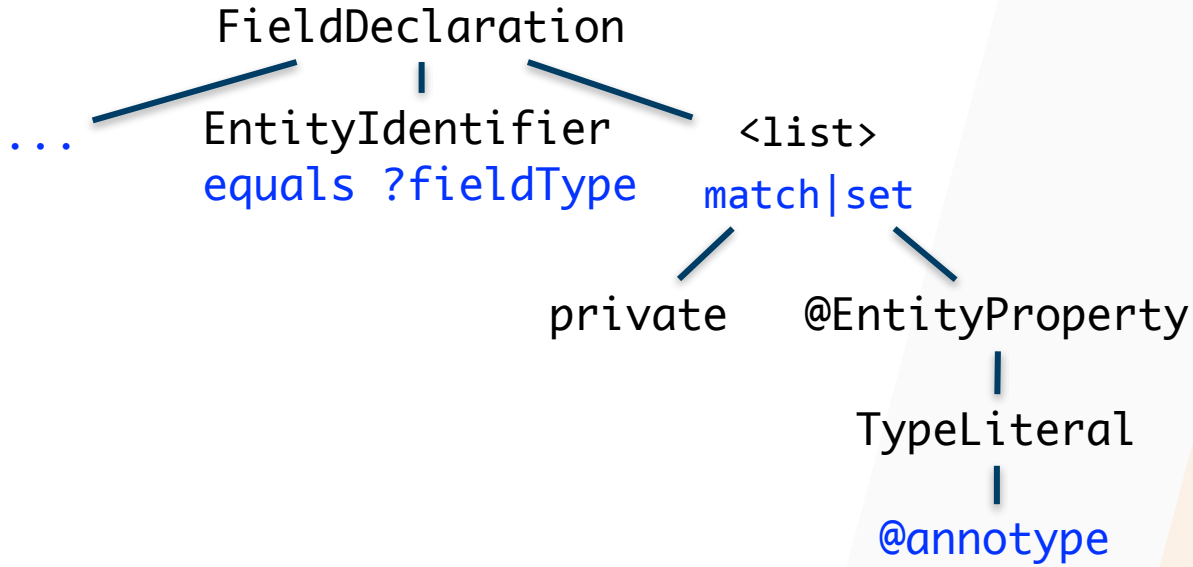

*Potential matches; all field declarations:*

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;

public Account acc;

private static EntityIdentifier ei;

...
```



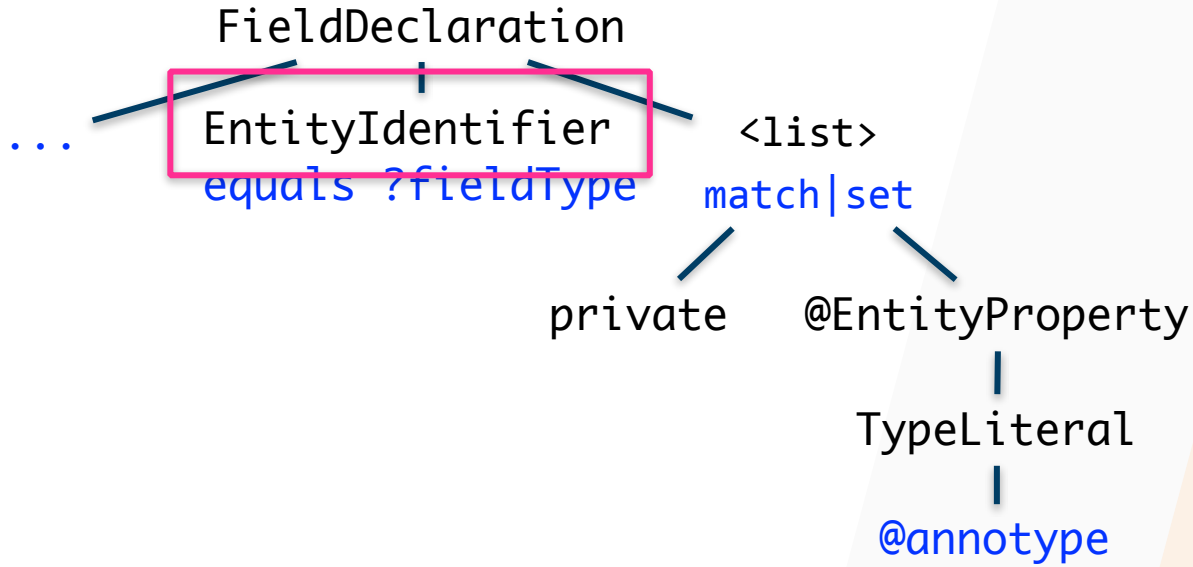

*Potential matches; all field declarations:*

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;
public Account acc;
private static EntityIdentifier ei;
...
```



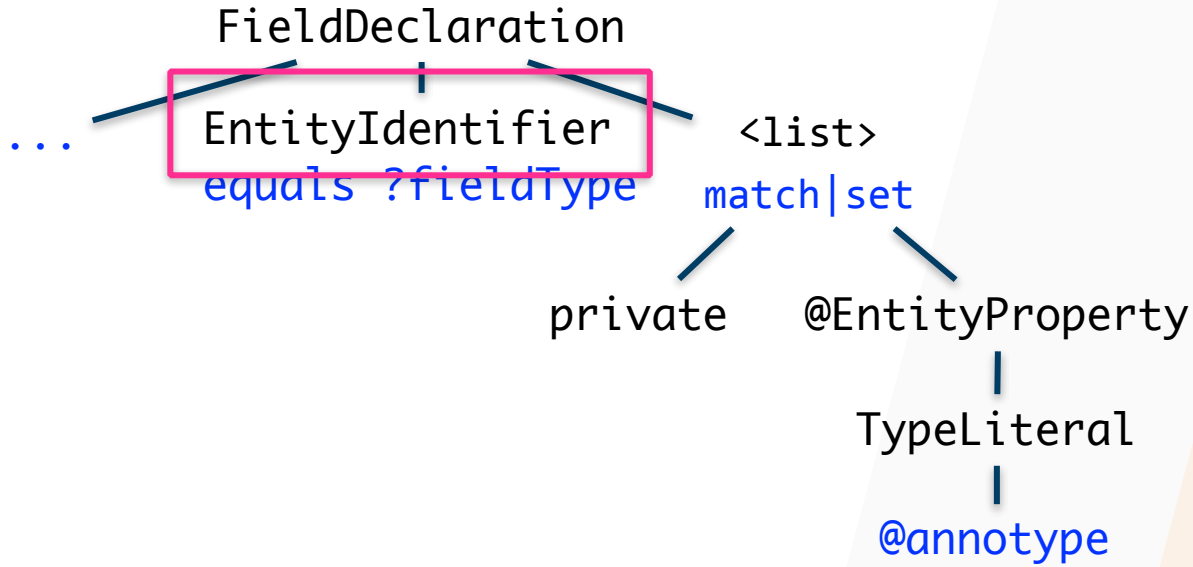

*Potential matches; all field declarations:*

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;

public Account acc;

private static EntityIdentifier ei;

...
```



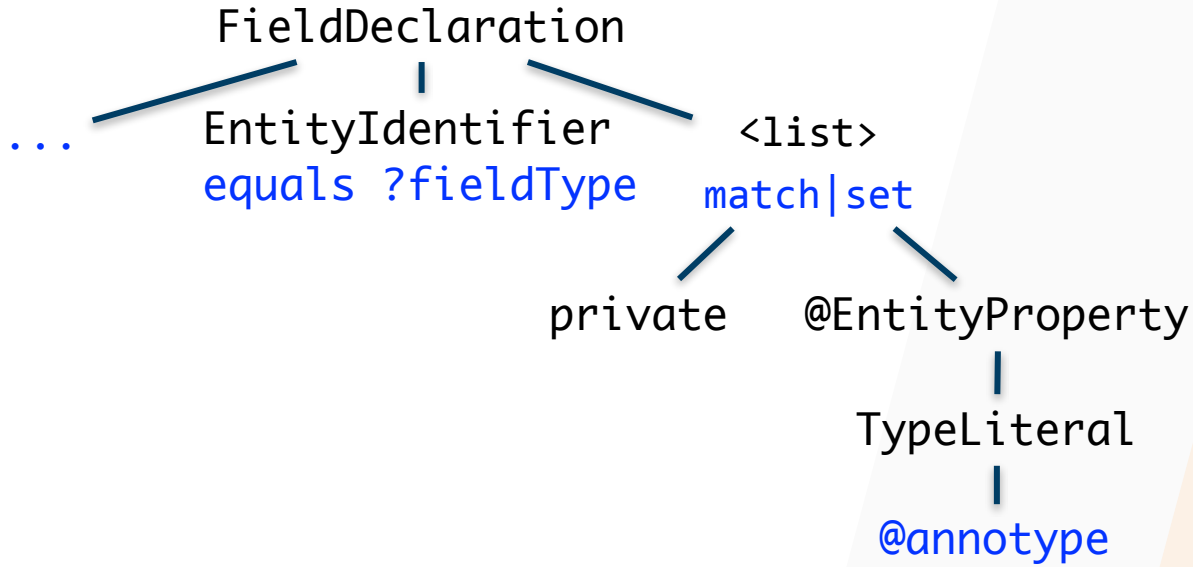

*Potential matches; all field declarations:*

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;
public Account acc;
private static EntityIdentifier ei;
...
```



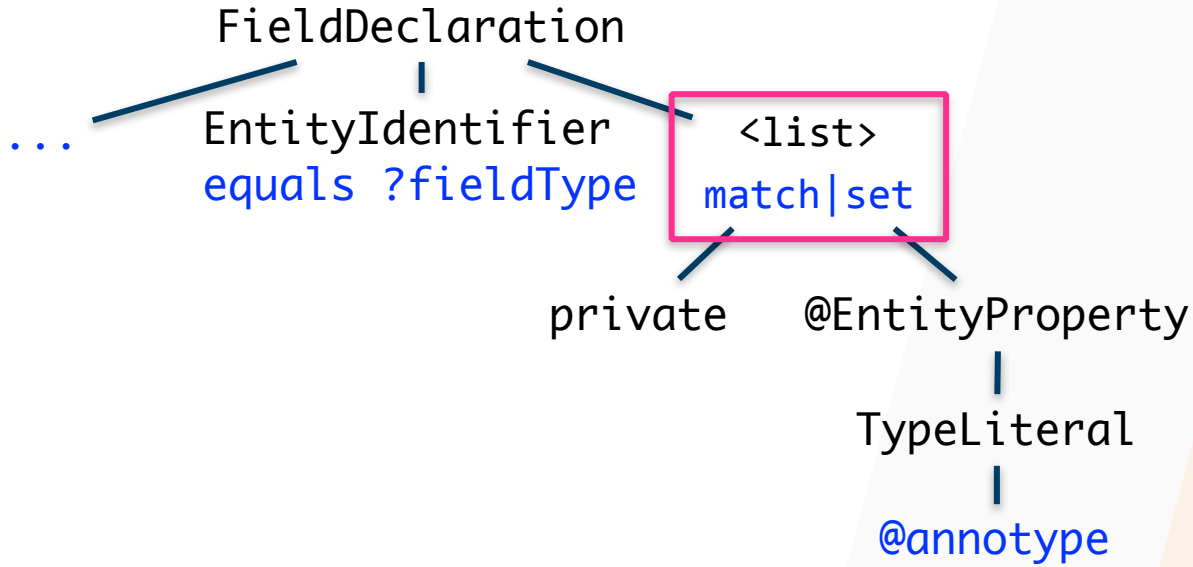

*Potential matches; all field declarations:*

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;
public Account acc;
private static EntityIdentifier ei;
...
```




*Potential matches; all field declarations:*

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;
public Account acc;
private static EntityIdentifier ei;
...
```




*Potential matches; all field declarations:*

```
private @EntityProperty(value=Expression.class) EntityIdentifier array;
public Account acc;
private static EntityIdentifier ei;
...
```

# Ekeko/X matching algorithm

```
potentialMatches = all AST nodes with same type as template's root node
matchMap = new MatchMap(potentialMatches)
m = processNode(template, matchMap)
return m.getMatches


processNode(Node n, MatchMap m) {
  remove all entries in m where n.getClass() does not match
  remove all entries in m where a directive on n does not match
  for each child x in n:
    move the current position(s) of each entry in m, corresponding to x
    m = processNode(x, m)
    move the current position(s) of each entry in m back to n
  return m
}
```

# Moving the current position

- Normal case: Child node
- child* : Next position can be any descendant node
- child+ : Next position can be any descendant, except the direct child
- match|set: Next position can be any list element (except those already checked)

# Matching algorithm data structure

AST A

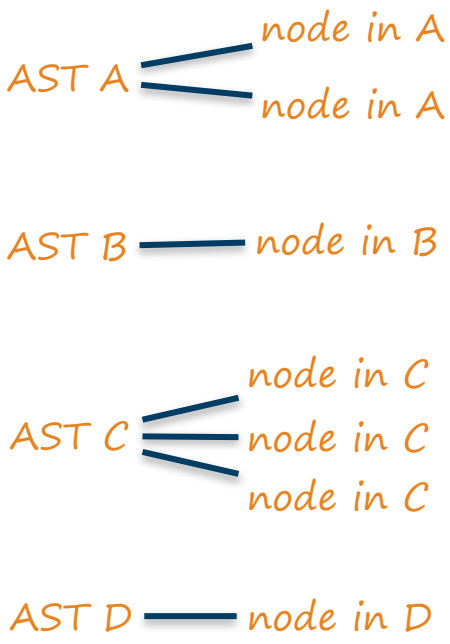
AST B

AST C

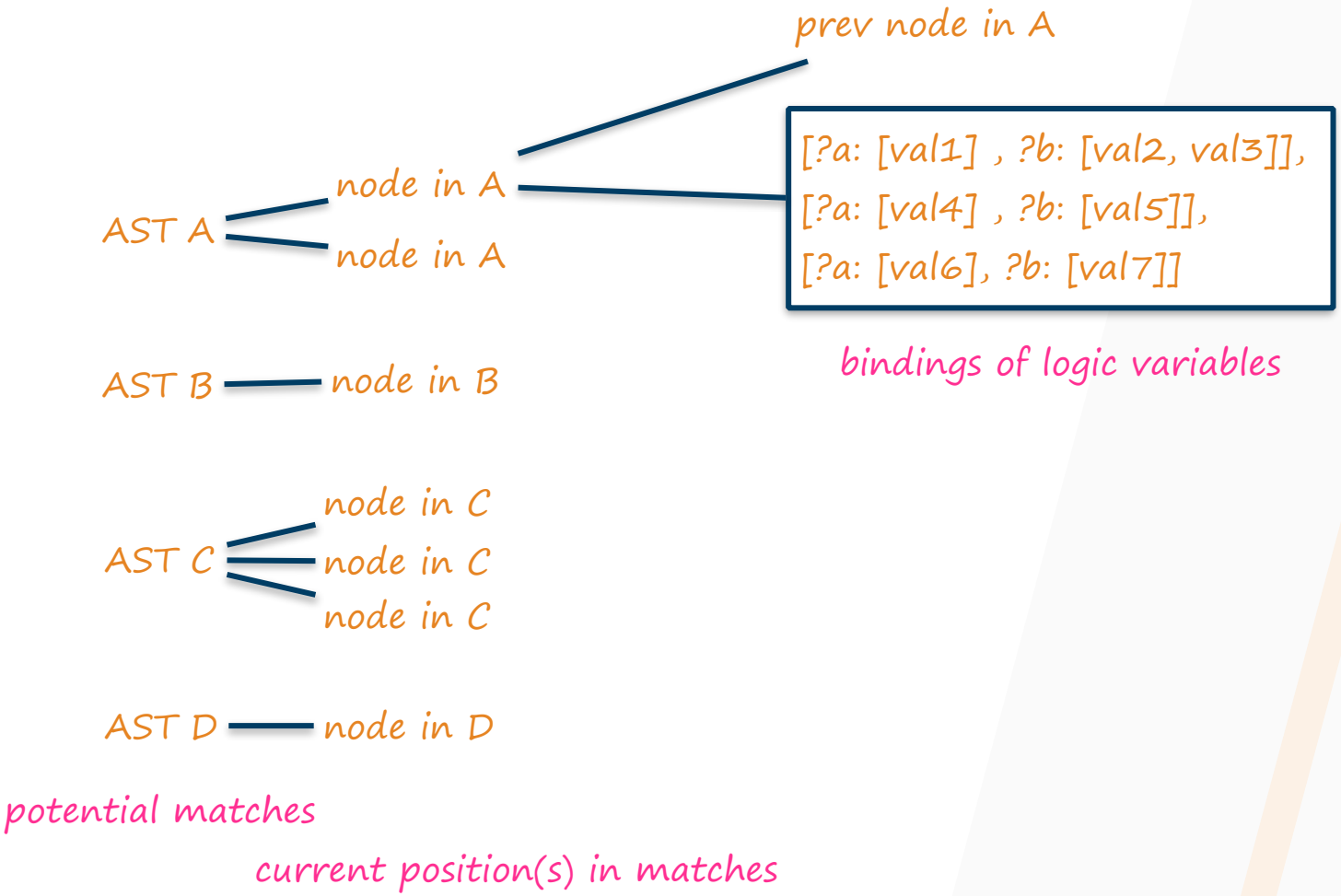
AST D

potential matches

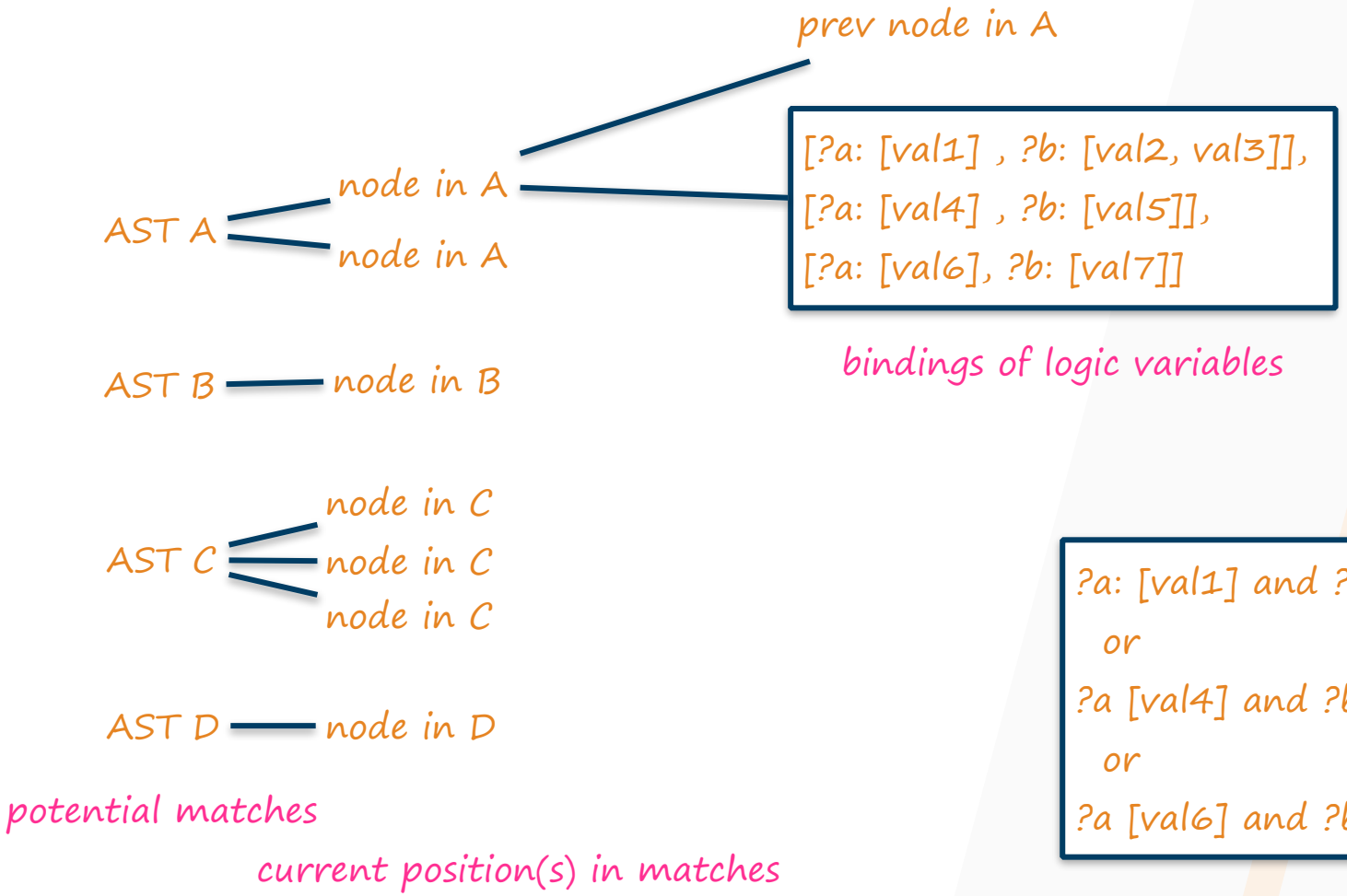
# Matching algorithm data structure



# Matching algorithm data structure



# Matching algorithm data structure



# Partial matches

- Can call `processNode` with only 1 specific potential match of interest
- Simply count how many template nodes succeeded
- Useful to guide genetic search when inferring a template from matches; "how close is this generated template to the desired matches?"