0.1 Renaming Handler

0.1.1 Early Concepts

- **Nodes possibly renamed by left/right without body changes**:
  $R_L = \{b \in B \mid (\neg\exists l \in L)(b.id = l.id) \wedge (\exists l \in L)(b.body - l.body)\}$
  $R_R = \{b \in B \mid (\neg\exists r \in R)(b.id = r.id) \wedge (\exists r \in R)(b.body - r.body)\}$
- **Nodes possibly deleted or renamed by left/right with body changes**:
  $DR_L \leftarrow \{b \in B \mid (\neg\exists l \in L)(b.id = l.id \vee b.body = l.body)\}$
  $DR_R \leftarrow \{b \in B \mid (\neg\exists r \in R)(b.id = r.id \vee b.body = r.body)\}$
- **Nodes IDs similarity**:
  $a.id \approx b.id \leftrightarrow a.id.name = b.id.name \vee a.id.params = b.id.params$

0.1.2 Match Algorithm

Algorithm 1: Match Algorithm

Input: L, B, R, M
Output: Set of quadruples (l, b, r, m) consisting of the base node b and its corresponding left node l, right node r and merge node m

```
1 matches ← ∅;
2 foreach b ∈ R_L ∪ R_R ∪ DR_L ∪ DR_R do
3 |   l ← correspondentNode(b, L);
4 |   r ← correspondentNode(b, R);
5 |   m ← mergeNode(l, r, M);
6 |   matches ← matches ∪ (l, b, r, m);
7 end
8 return matches
```

Algorithm 2: Correspondent Node

Input: b, T
Output: b's correspondent node on tree T

```
1  t ← findFirst(t ∈ T → t.id = b.id);
2  if t = null then
3  |   t ← findFirst(t ∈ T → t.body = b.body);
4  end
5  if t = null then
6  |   t ← findFirst(t ∈ T → t.id ≈ b.id ∧ t.body ≈ b.body);
7  end
8  if t = null then
9  |   t ← findFirst(t ∈ T → t.body = substring(b.body) ∨ b.body = substring(t.body));
10 end
11 return t;
```

Algorithm 3: Merge Node

Input: l, r, M
Output: l and r's merge node on tree M

```
1 if l ≠ null then
2 |   return find(m ∈ M → m.id = l.id);
3 end
4 if r ≠ null then
5 |   return find(m ∈ M → m.id = r.id);
6 end
7 return null;
```

0.1.3 Handler Algorithms

Algorithm 4: Check References and Merge Methods Variant

Input: (l, b, r, m), M

```
1  if l.id = b.id ∨ r.id = b.id then
2  |   m.body ← textualMerge(l, b, r);
3  |   removeUnmatchedNode(l, r, m, M);
4  else if l.id ≠ r.id then
5  |   m.body ← conflict(l.body, b.body, r.body);
6  |   removeUnmatchedNode(l, r, m, M);
7  else if l.body ≠ r.body then
8  |   if newReferenceTo(l) ∨ newReferenceTo(r) then
9  |   |   m.body ← conflict(l.body, b.body, r.body);
10 |   else
11 |   |   m.body ← textualMerge(l, b, r);
12 |   end
13 |   removeUnmatchedNode(l, r, m, M);
14 end
```

Algorithm 5: Merge Methods Variant

Input: (l, b, r, m), M

```
1 m.body ← textualMerge(l, b, r);
2 removeUnmatchedNode(l, r, m, M);
```

Algorithm 6: Check Textual and Keep Both Methods Variant

Input: (l, b, r, m), M

```
1 if l.id = b.id ∨ r.id = b.id then
2 |   if textualMergeHasConflictInvolvingSignature(b) then
3 |   |   m.body ← conflict(l.body, b.body, r.body);
4 |   |   removeUnmatchedNode(l, r, m, M);
5 |   end
6 else if l.id ≠ r.id ∧ l.body = r.body then
7 |   m.body ← conflict(l.body, b.body, r.body);
8 |   removeUnmatchedNode(l, r, m, M);
9 end
```

Algorithm 7: Keep Both Methods Variant

Input: (l, b, r, m), M

```
1 if (l.id = b.id ∨ r.id = b.id) ∧ hasConflict(m) then
2 |   removeConflict(m);
3 end
```

Algorithm 8: Remove Unmatched Node

Input: l, r, m, M

```
1 if l.id = m.id ∧ r.id ≠ m.id then
2 |   removeNode(r, M);
3 end
```