Program Comparison

Overview

- Program Representation
- Clone Detection
- AST Differencing

Program Representation

- String
- Token sequence
- Abstract Syntax Tree
- Control Flow Graph
- Program Dependence Graph
- Points-to Graph
- Call Graph

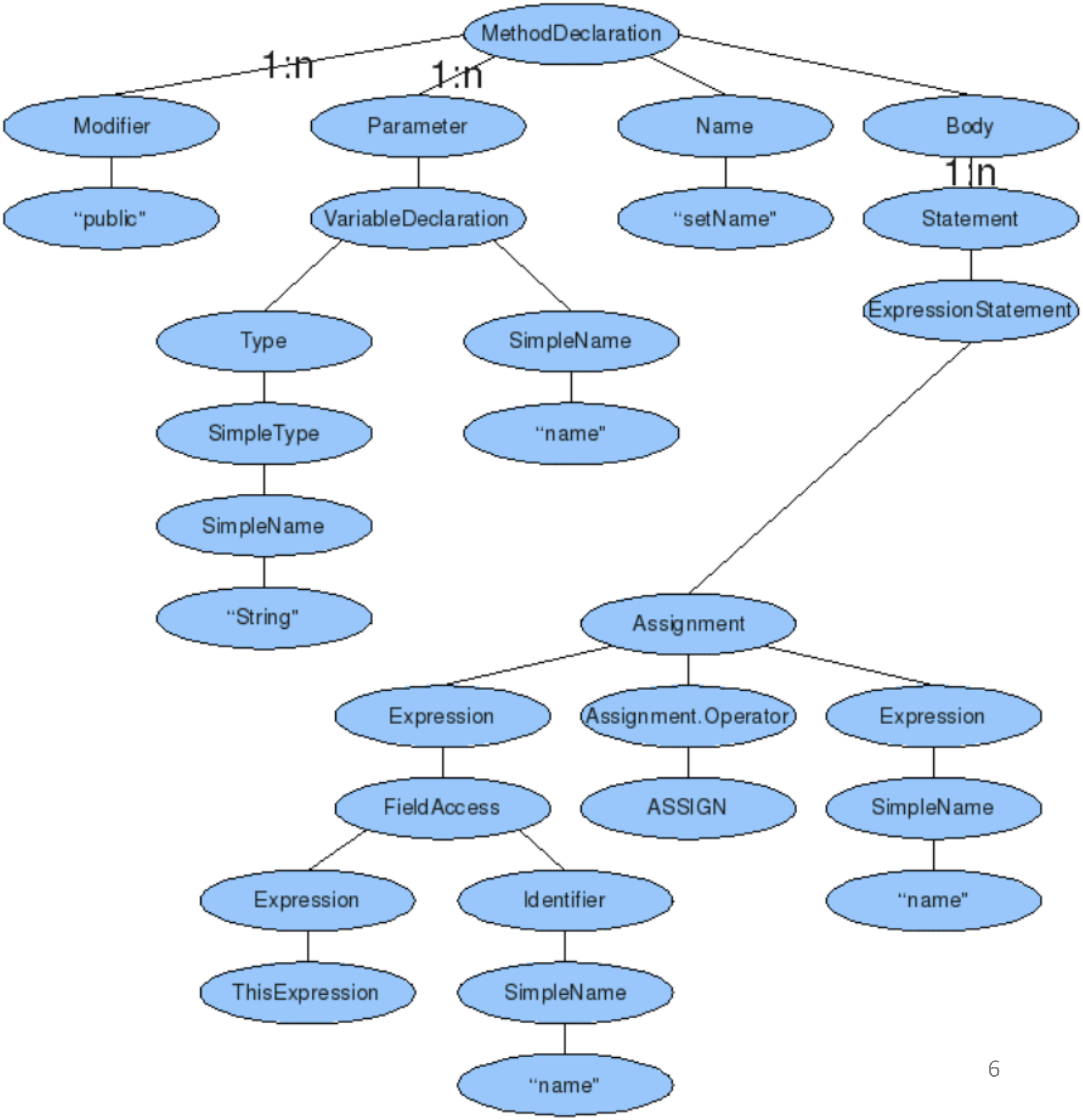
Abstract Syntax Tree (AST)

- Created by the compiler at the end of semantic analysis phase
- A tree representation for the abstract syntactic structure of source code
  - Node: construct, such as statement and loop
  - Edge: containment relationship
- Different compilers can define different AST representations

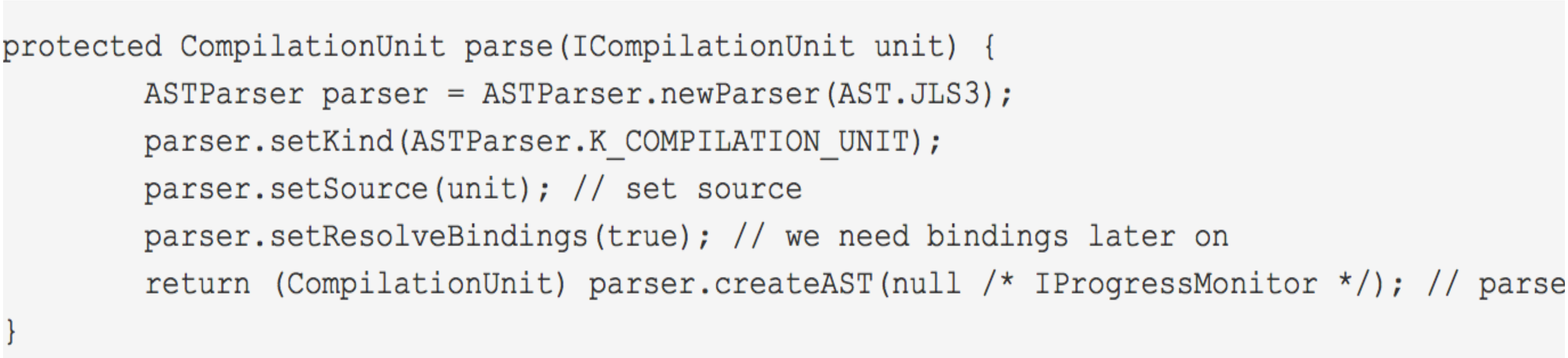
Eclipse JDT

- The Eclipse Java Development Tools project (JDT) provides
  - tools to develop Java application
  - APIs to access, create, and manipulate Java projects' source code
- It provides access to Java source code via two ways: Java Model and Abstract Syntax Tree

Eclipse AST[3]



How do we generate Eclipse AST from source code?



```
protected CompilationUnit parse(ICompilationUnit unit) {
        ASTParser parser = ASTParser.newParser(AST.JLS3);
        parser.setKind(ASTParser.K_COMPILATION_UNIT);
        parser.setSource(unit); // set source
        parser.setResolveBindings(true); // we need bindings later on
        return (CompilationUnit) parser.createAST(null /* IProgressMonitor */); // parse
}
```

How do we use Eclipse AST?

- Use ASTVisitor to parse any source code information from the AST
- Conduct program analysis based on the AST information
- Manipulate AST to insert/delete code

Control Flow Graph (CFG)

- *A* representation, using graph notation, of all paths that might be traversed through a program during its execution

Formal Definition[5]

- CFG = <V, E, Entry, Exit>, where
  - V = vertices or nodes, representing an instruction or basic block (a group of instructions)
  - E = edges, potential flow of control, $E \subseteq V \times V$
  - $Entry \in V$, unique program entry $(\forall v \in V)[Entry \xrightarrow{*} v]$
  - $Exit \in V$, unique program exit $(\forall v \in V)[v \xrightarrow{*} Exit]$

Basic Block

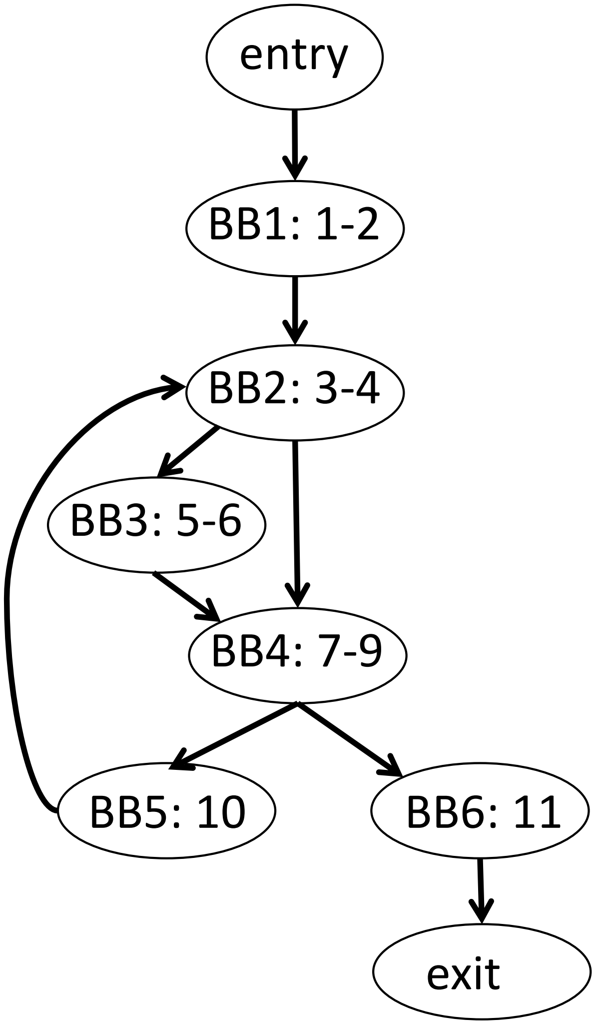
- A maximal sequence of consecutive instructions such that inside the basic block, an execution can only proceed from one instruction to the next
- Single entry, single exit

CFG Example

```
1          A = 4
2          t1 = A * B

3  L1:     t2 = t1/C
4           if t2 < W goto  L2

5          M = t1 * k
6          t3 = M + I

7  L2:          H = I
8           M = t3 – H
9           if t3 >= 0 goto L3

10          goto L1

11 L3:          halt
```

- What are the basic blocks?
- What are the edges between them?
- What is the CFG?

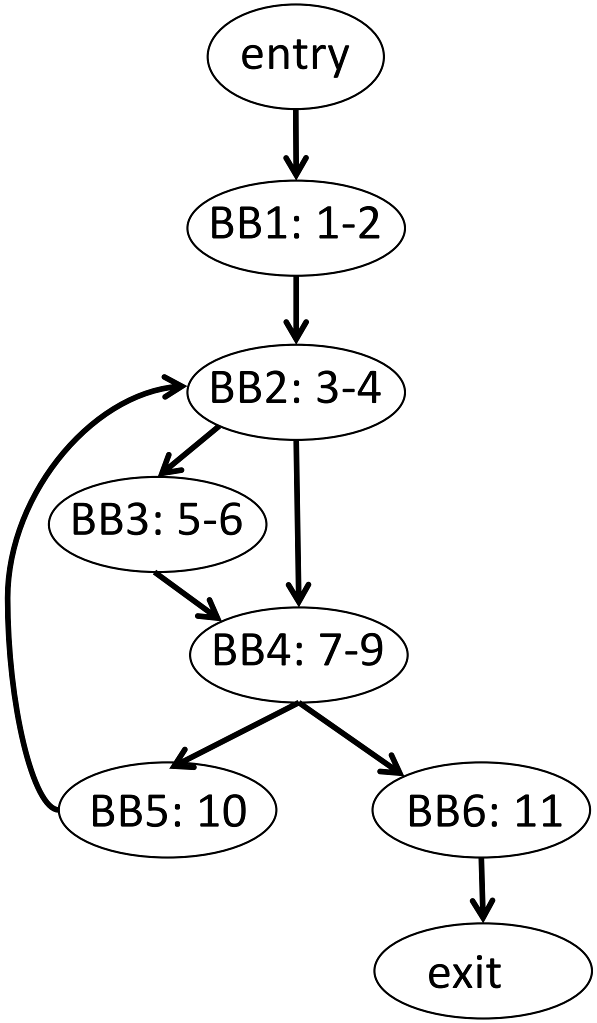
CFG Example



Why is CFG important?

- A lot of program analysis and abstract representations are built based on it
- In testing scenario, CFG is leveraged to design test cases in order to have enough path/statement coverage

CFG Used for Selective Testing



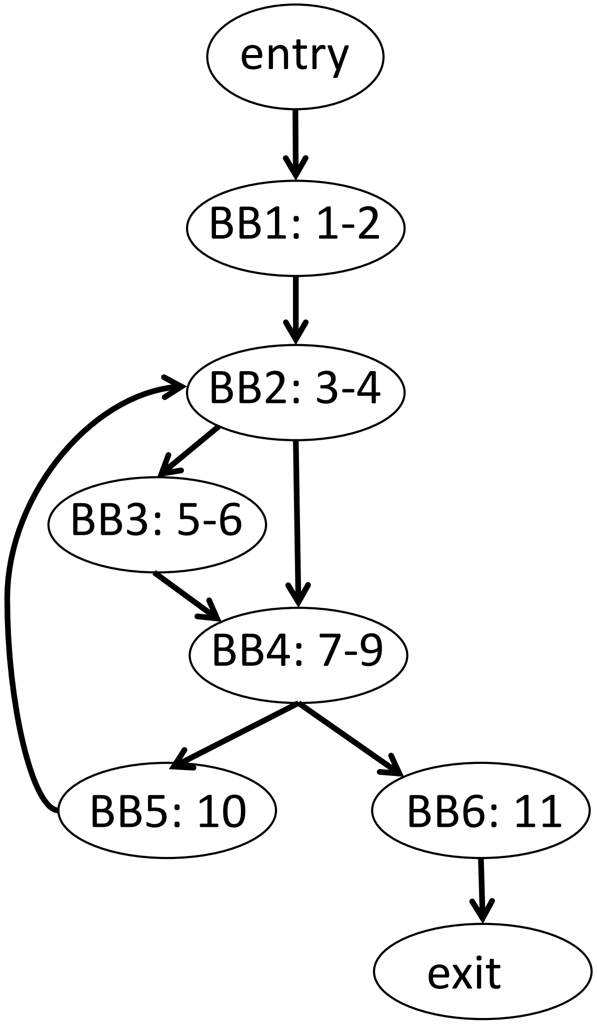
- Basic Path Testing
  - Cyclomatic complexity V(G)
    - number of simple decisions + 1
    - number of enclosed areas + 1
  - What are the paths to test?

Program Dependence Graph (PDG)

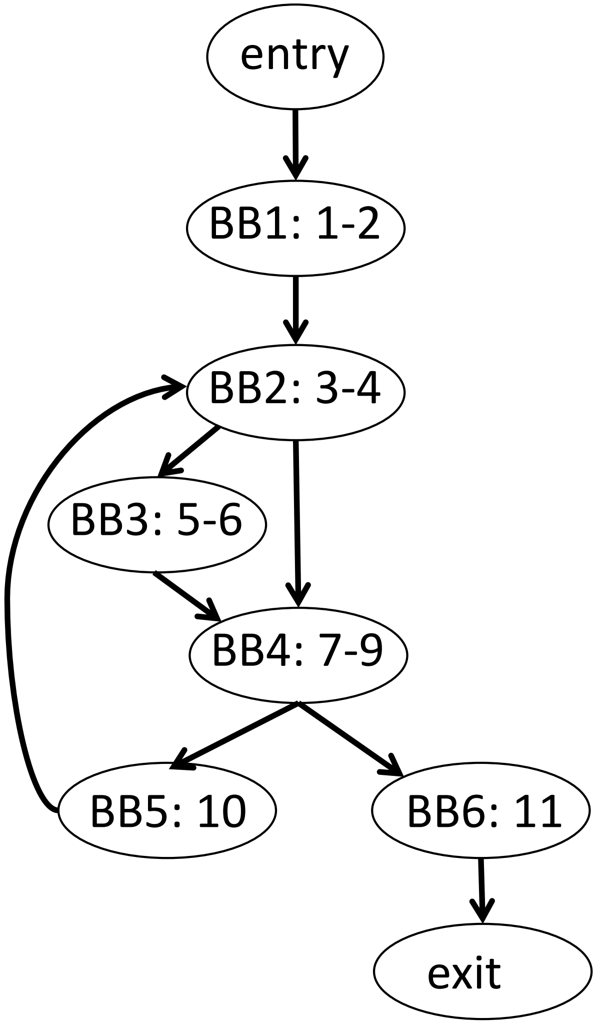
- A directed graph representing dependencies among code
  - Control dependence
    - A **control depends on** B if B's execution decides whether or not A is executed
  - Data dependence
    - A **data depends on** B if A uses variable defined in B

Control Dependence Example

- BB3 control depends on BB2 because whether or not BB3 is executed depends on the branch taken at BB2
  - Every block control depends on entry block



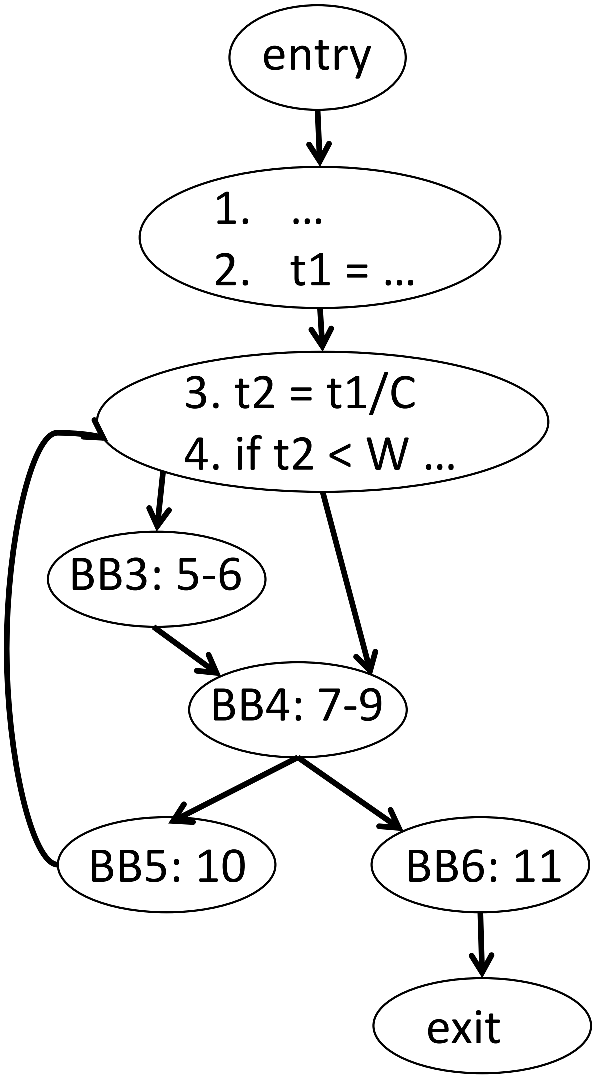
- In most cases, statements control depend on their AST container constructs, such as loop, switch, if. Can you think about cases violating this observation?



Data Dependence Example

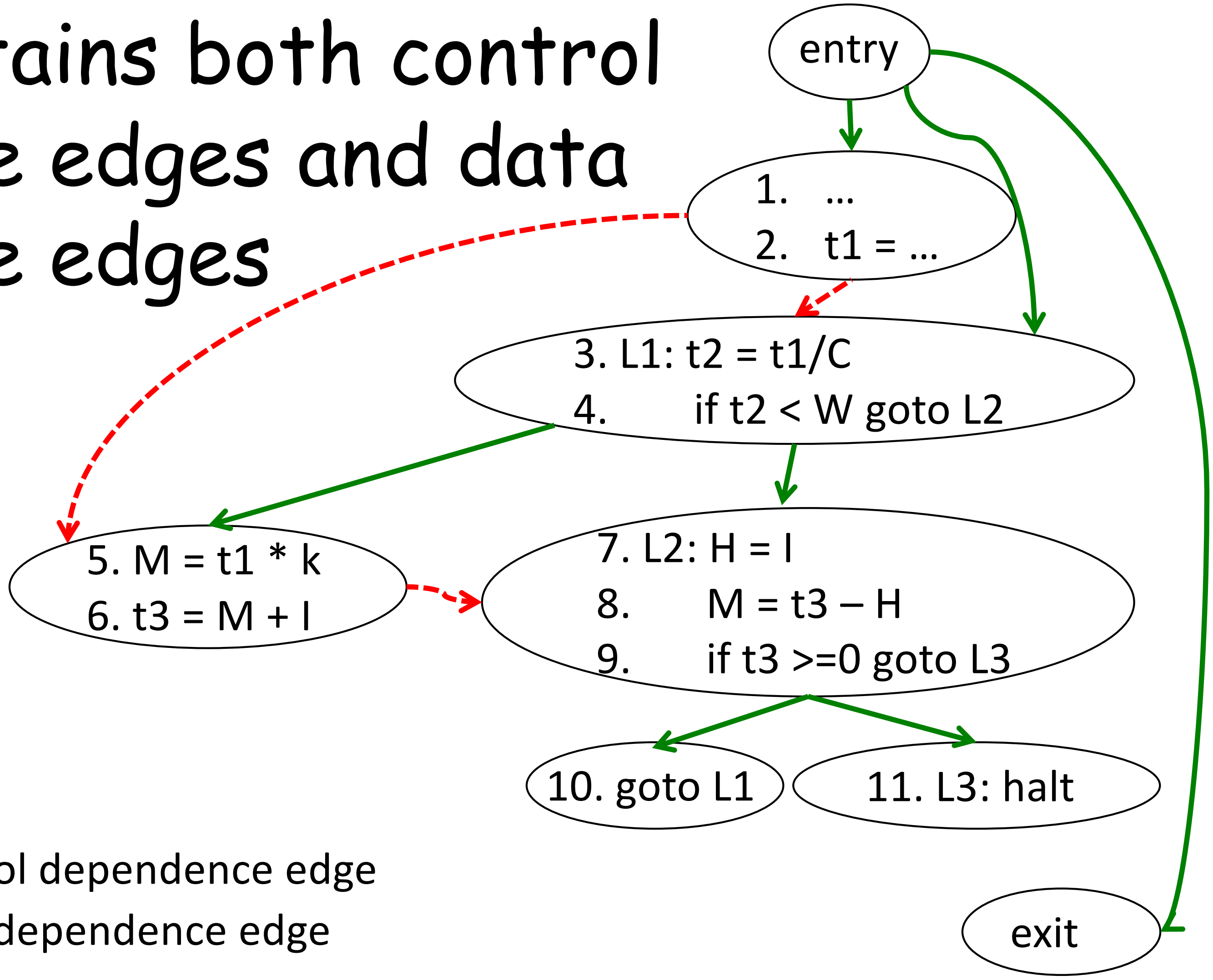
- BB2 data depends on BB1 because BB2 uses the variable t1, whose value is defined by instruction(s) in BB1
  - Which statement does "sum = sum + i" data depend on?

```
sum = 0;
i = 1;
while (i < N) {
    i = i + 1;
    sum = sum + i;
}
```



PDG

- A PDG contains both control dependence edges and data dependence edges



entry

1. …
2. t1 = …

3. L1: t2 = t1/C
4. if t2 < W goto L2

5. M = t1 * k
6. t3 = M + I

7. L2: H = I
8. M = t3 – H
9. if t3 >=0 goto L3

10. goto L1

11. L3: halt

exit

🟩 Direct control dependence edge

🟥 Direct data dependence edge

Why is PDG important?

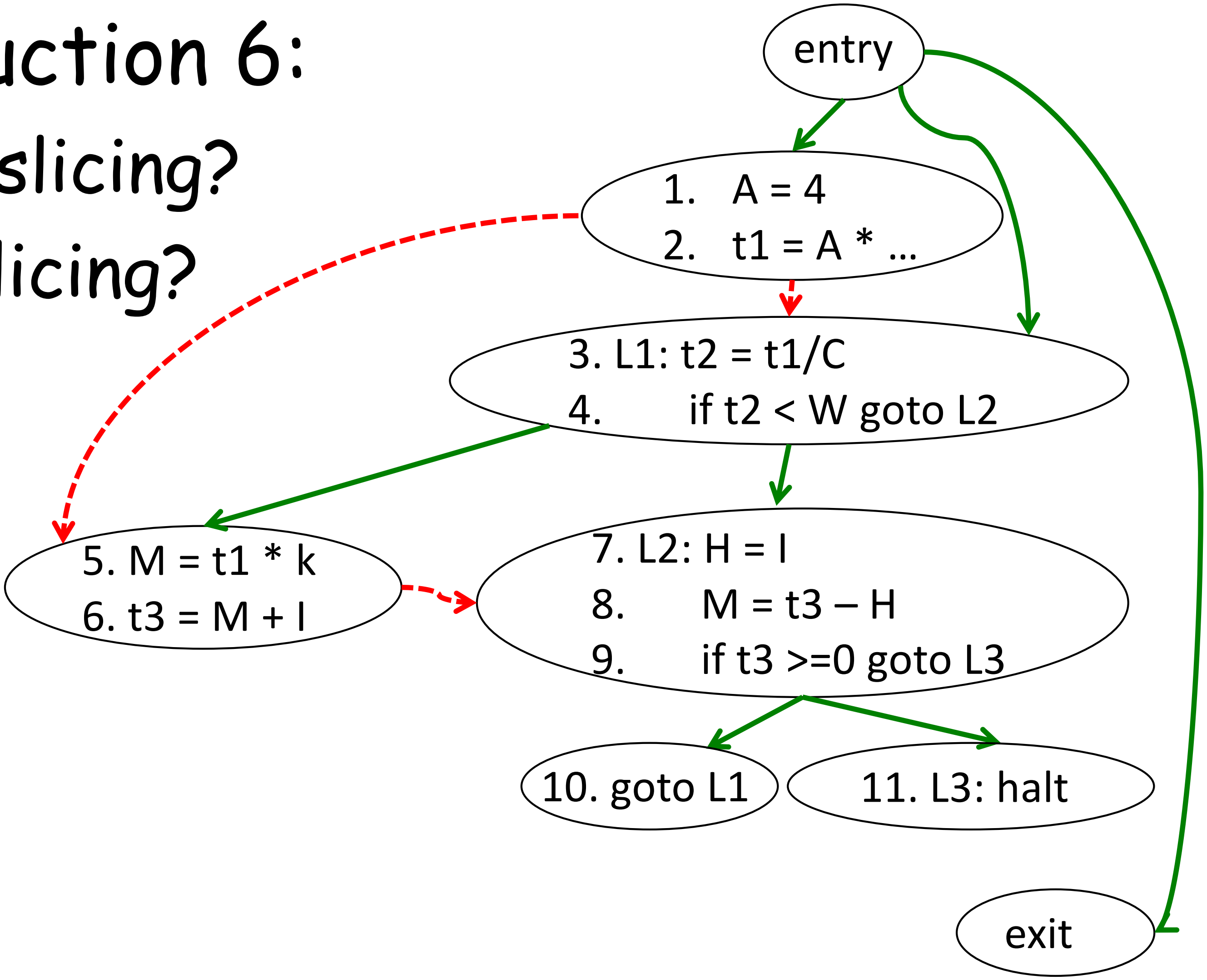
- It demonstrates some program semantics and facilitates program comprehension
  - find bugs, program slicing
- Guide safe program transformations/optimizations which modify code without compromising dependency relations
  - Automatic parallelism, common sub-expression elimination, code motion

Program Slicing

- Set of statements that may affect the values at some point of interest
  - data/control dependence relationship
- Backward slicing
  - The statements the current value is dependent on
- Forward slicing
  - The statements which depend on the current value

Example

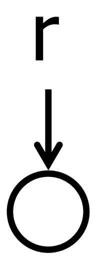
- t3 at instruction 6:
  - Backward slicing?
  - Forward slicing?



Points-to Graph

- For a program location, for any object reference/pointer, calculate all the possible objects/variables it may/must refer/point to

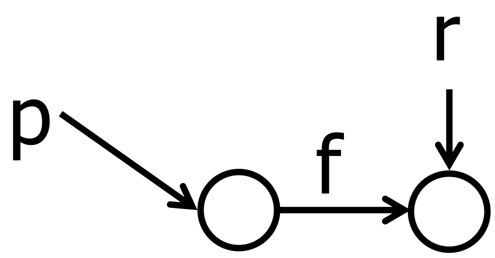
```
r = new C();
p.f = r;
t = new C();
if (…)
    q=p;
r->f = t;
```



Points-to Graph

- For a program location, for any object reference/pointer, calculate all the possible objects/variables it may/must refer/point to

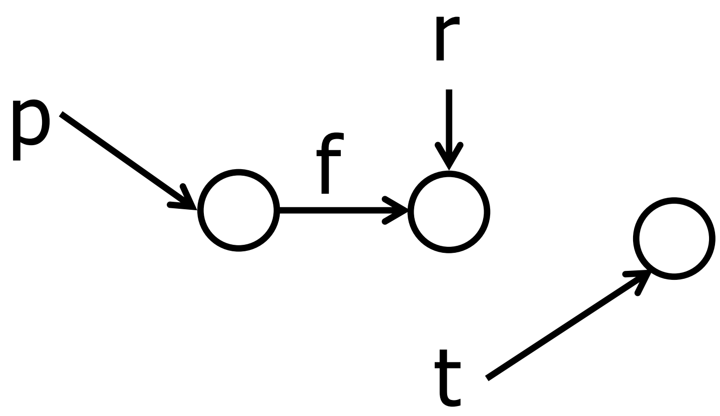
```
  r = new C();
→ p.f = r;
  t = new C();
  if (…)
      q=p;
  r->f = t;
```



Points-to Graph[4]

- For a program location, for any object reference/pointer, calculate all the possible objects/variables it may/must refer/point to

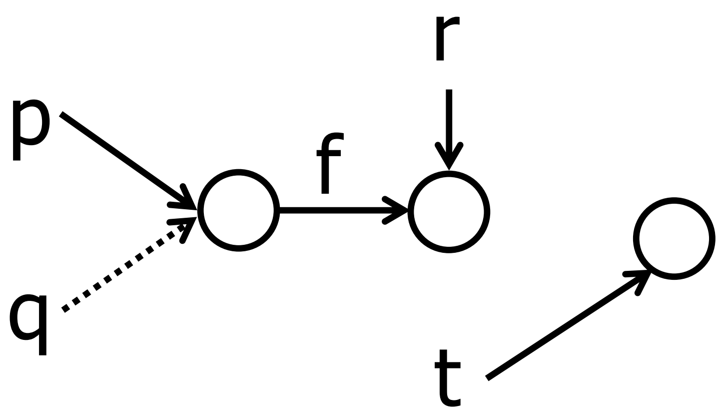
```
  r = new C();
  p.f = r;
→ t = new C();
  if (…)
      q=p;
  r->f = t;
```



Points-to Graph

- For a program location, for any object reference/pointer, calculate all the possible objects/variables it may/must refer/point to

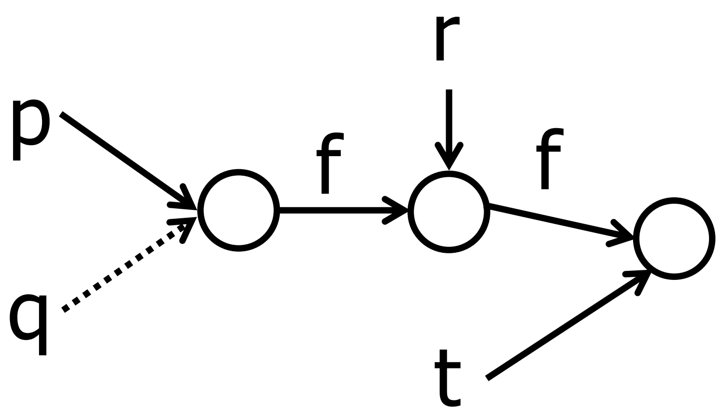
```
r = new C();
p.f = r;
t = new C();
if (…)
    q=p;
r.f = t;
```



Points-to Graph

- For a program location, for any object reference/pointer, calculate all the possible objects/variables it may/must refer/point to

```
r = new C();
p.f = r;
t = new C();
if (…)
    q=p;
r->f = t;
```



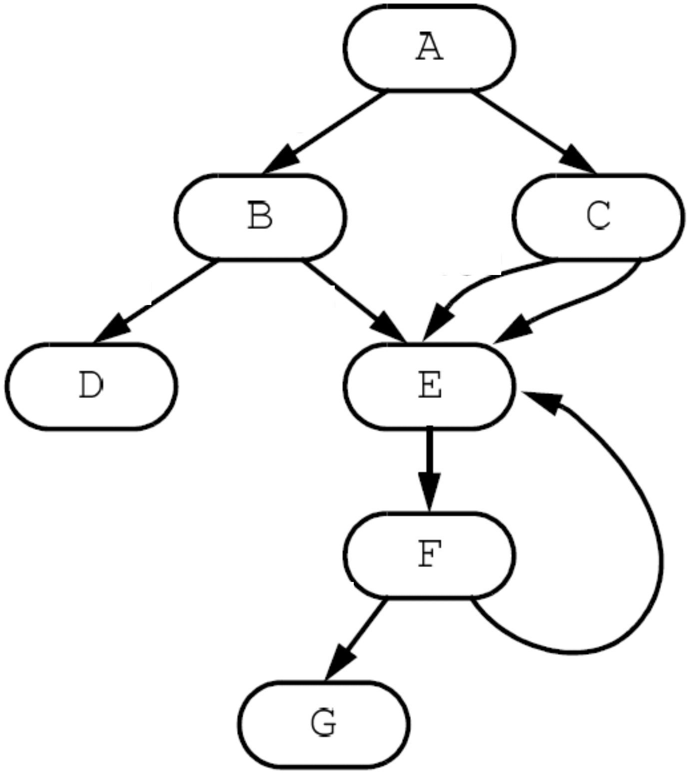
p.f.f and t are aliases

Why is Points-to Graph important?

- Connect together analyzed program semantics for individual methods
  - Essential to expand from intra-procedural analysis to inter-procedural analysis
- Detect consistent usage of resources
  - File open/close, lock/unlock, malloc/free
- Garbage collection

Call Graph

- A directed graph representing caller-callee relationship between methods/functions
  - Node: methods/functions
  - Edges: calls



Why is Call Graph important?

- Facilitate program comprehension and optimization
  - When a program crashes, what is the possible calling context?
  - Detect anomalies of program execution

Code Clones

Spiros Mancoridis[1]
Modified by Na Meng

Code Clones

- Code clone is a code fragment in source files that is identical or similar to another
- Code clones are either within a program or across different programs
- Clone pair: two clones
- Clone class: a set of fragments which are clones to each other

Code Clone Categorization

- Type-1 clones
  - Identical code fragments but may have some variations in whitespace, layout, and comments
- Type-2 clones
  - Syntactically equivalent fragments with some variations in identifiers, literals, types, whitespace, layout and comments

Code Clone Categorization

- Type-3 clones
  - Syntactically similar code with inserted, deleted, or updated statements
- Type-4 clones
  - Semantically equivalent, but syntactically different code

Key Points of Code Clones

- Pros
  - Increase performance
    - Code inlining vs. function call
  - Increase program readability
- Cons
  - Increase maintenance cost
    - If one code fragment contains a bug and gets fixed, all its clone peers should be always fixed in similar ways.
  - Increase code size

Clone Detection Strategies

- Text matching
- Token sequence matching
- Graph matching

Text Matching

- Older, studied extensively
- Less complex, and most widely used
- No program structure is taken into consideration
- Type-1 clones & some Type-2 clones
- Two types of text matching
  - Exact string match
    - Diff (cvs, svn, git) is based on exact text matching
  - Ambiguous match

Ambiguous Match

- Longest Common Subsequence match
- N-grams match

Token Sequence Matching

- A little more complex, less widely used
- No program structure is taken into account, either
- Type-1 and Type-2 clones
- CCFinder[2]
- CP-Miner[3]

CCFinder

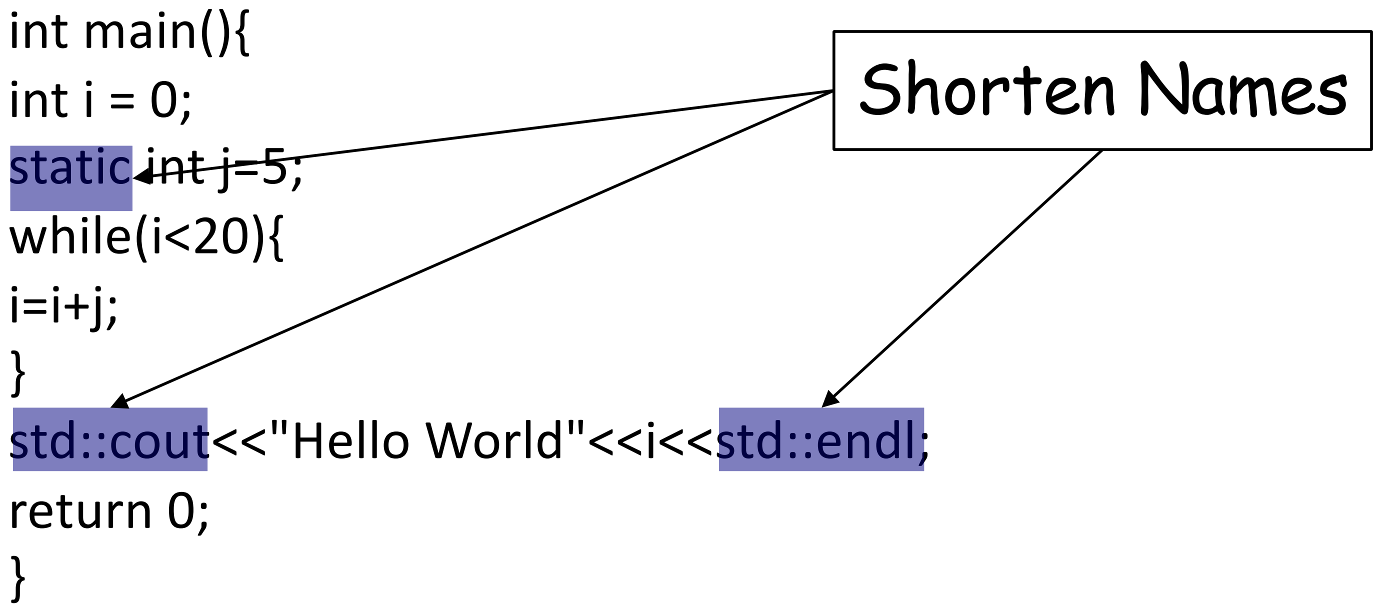
- Step 1: Convert a program with multiple files to a single long token sequence
- Step 2: Find longest common subsequence of tokens

Step 1: Tokenization

```
int main(){
    int i = 0;
    static int j=5;
    while(i<20){
        i=i+j;
    }
    std::cout<<"Hello World"<<i<<std::endl;
    return 0;
}
```

Remove white spaces

Step 1: Tokenization



```
int main(){
int i = 0;
static int j=5;
while(i<20){
i=i+j;
}
std::cout<<"Hello World"<<i<<std::endl;
return 0;
}
```

Shorten Names

Step 1: Tokenization

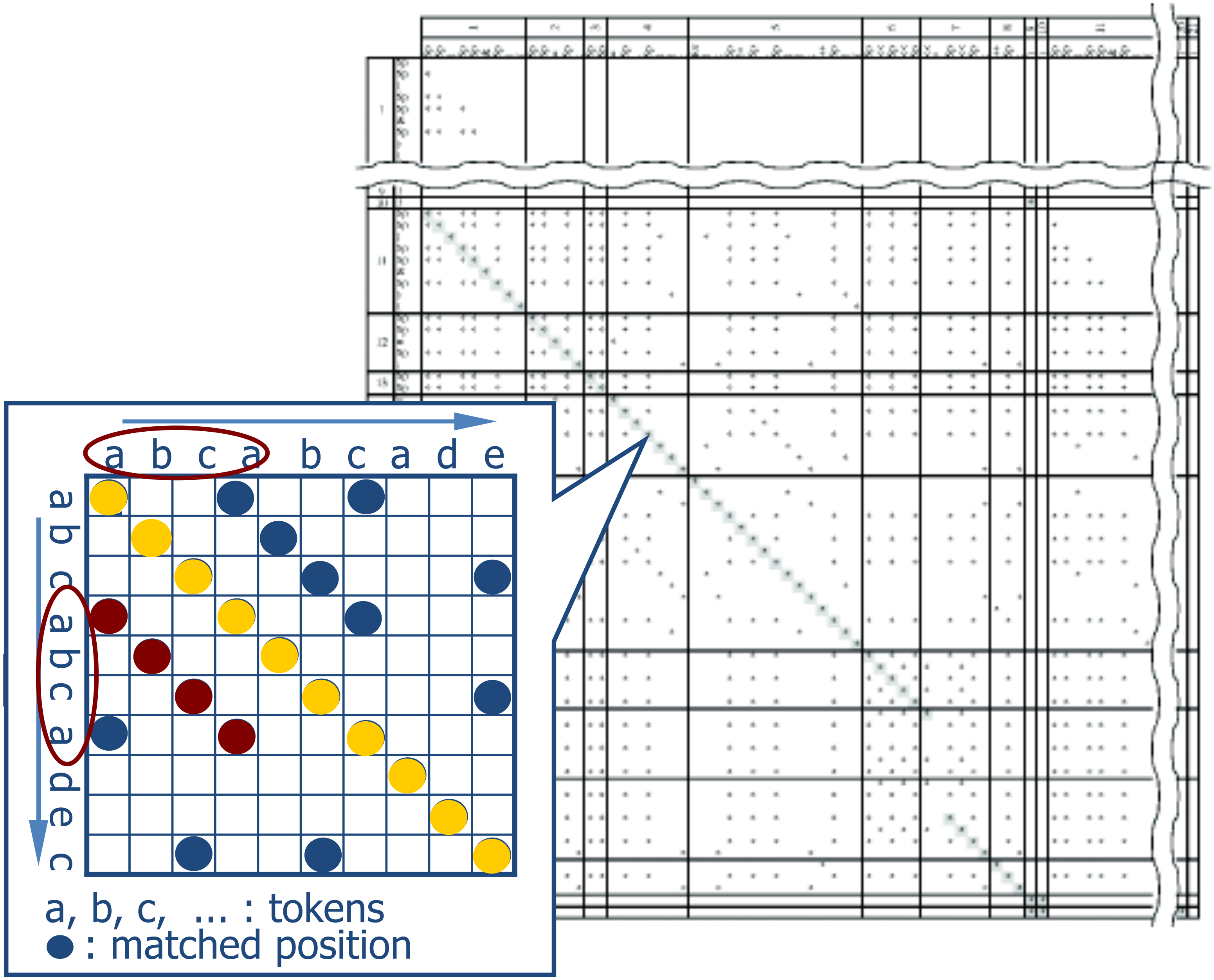
```
int main (){
int i = 0;
int j = 5;
while (i < 20){
i  =  i + j;
}
cout << "Hello World” << i << endl;
return 0;
}
```

Tokenize literals, and identifiers of types, methods, and variables.

Step 1: Tokenization

```
$p $p(){
$p $p = $p;
$p $p = $p;
while($p < $p ){
$p = $p + $p;
}
$p << $p << $p << $p;
return $p;
}
```

Step 2: Find Clones



a, b, c, ... : tokens

● : matched position

Detected Clone Pair Example[2]

```
static void foo() throws RESyntaxException {
    String a[] = new String [] { "123,400", "abc", "orange 100" };
    org.apache.regexp.RE pat = new org.apache.regexp.RE("[0-9,]+");
    int sum = 0;
    for (int i = 0; i < a.length; ++i)
        if (pat.match(a[i]))
            sum += Sample.parseNumber(pat.getParen(0));
    System.out.println("sum = " + sum);
}
static void goo(String [] a) throws RESyntaxException {
    RE exp = new RE("[0-9,]+");
    int sum = 0;
    for (int i = 0; i < a.length; ++i)
        if (exp.match(a[i]))
            sum += parseNumber(exp.getParen(0));
    System.out.println("sum = " + sum);
}
```

Limitations of CCFinder

- All files are converted into a long token sequence
  - When the program contains millions of lines of code, the tool cannot perform efficiently
- Do not take into account the natural boundary between functions and classes

CP-Miner[3]

- Cut the token sequences by considering basic blocks as cutting units
- Calculate a hashcode for each subsequence
- Compare hashcode sequences instead of the original token sequences

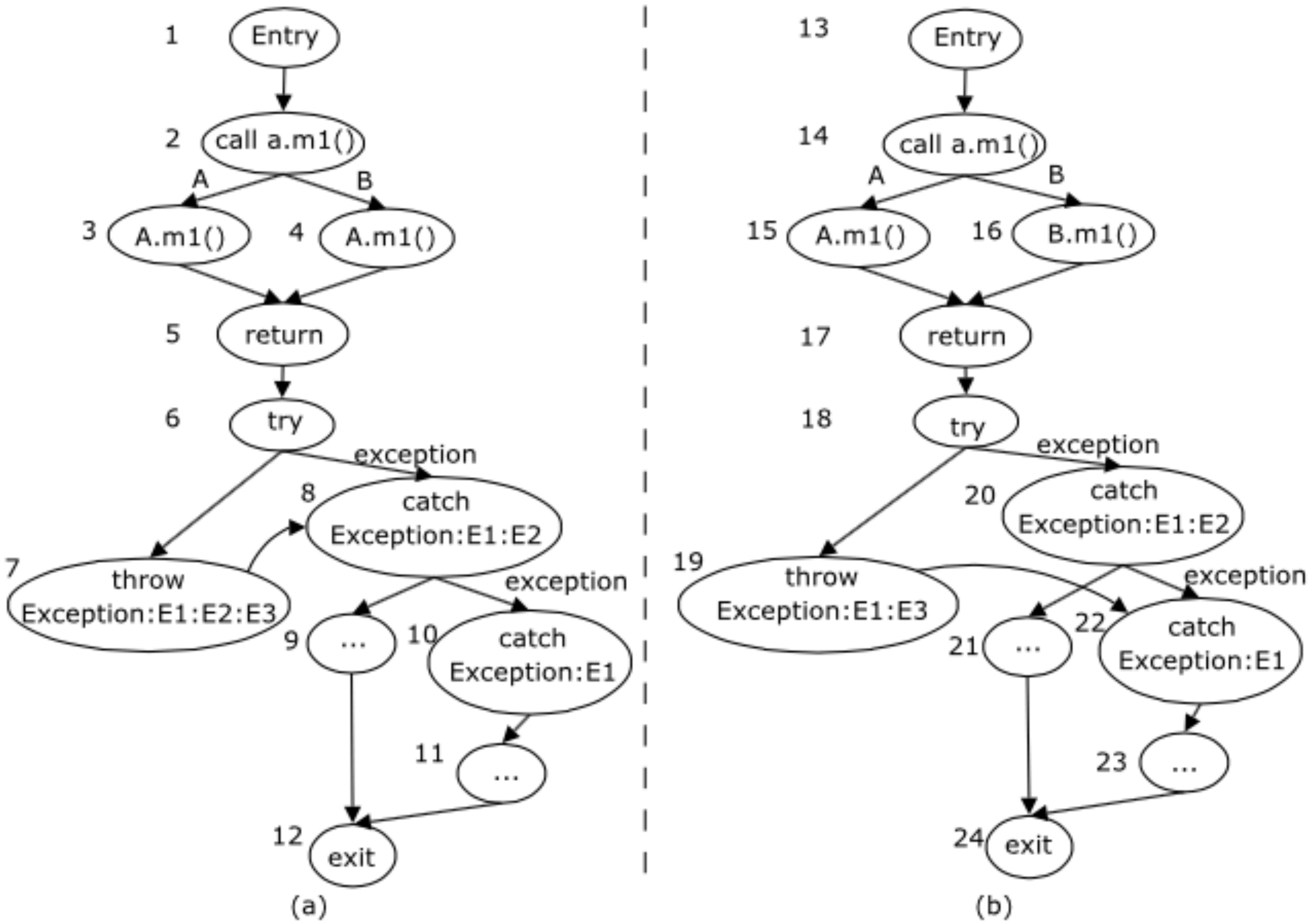
Graph Matching

- Newer, bleeding edge
- More complex
- Type-1, Type-2, and Type-3 clones
- Syntactic and semantic understanding
  - AST matching (ChangeDistiller)
  - CFG matching (Jdiff[4])
  - PDG matching ([5])

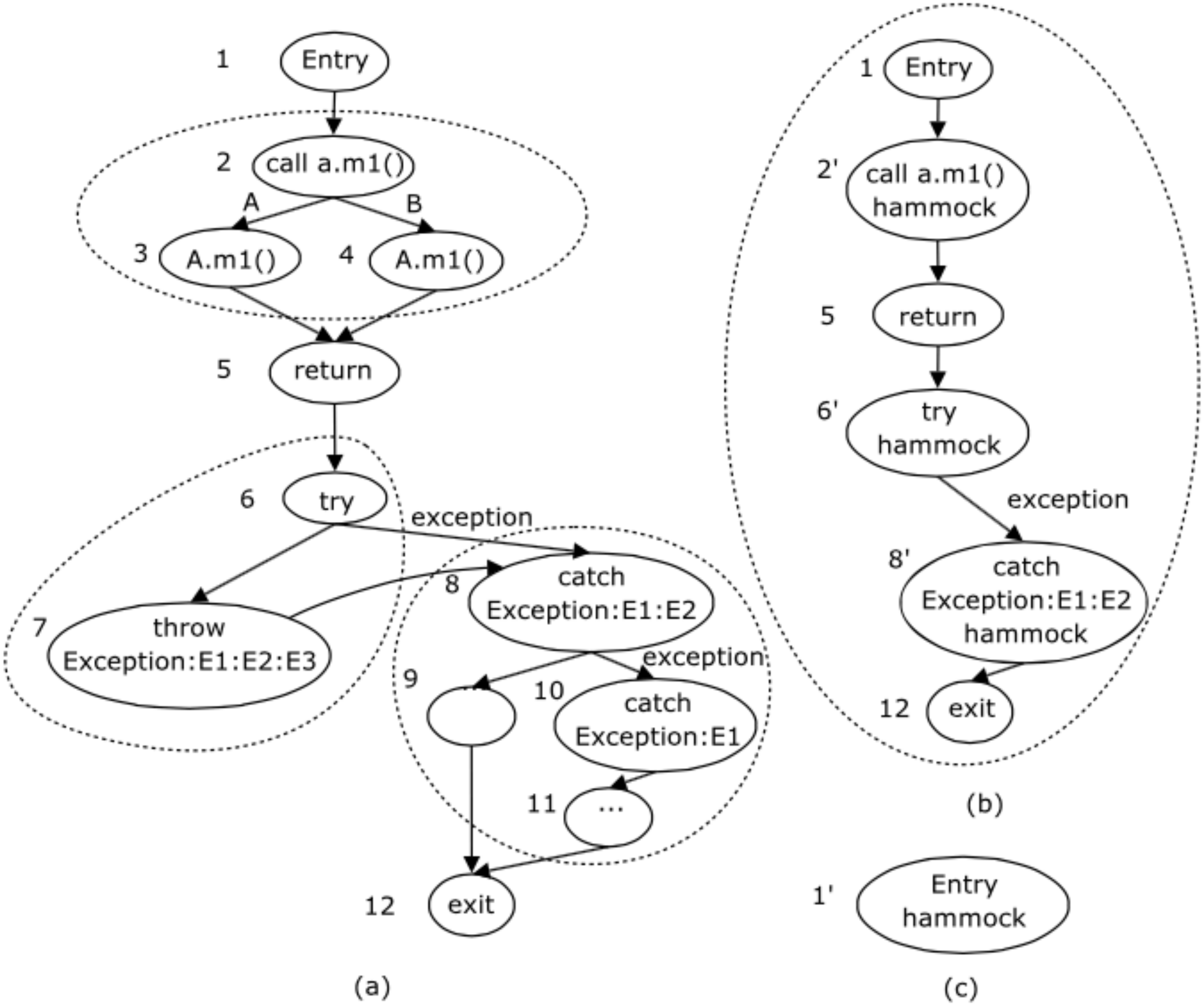
CFG-based Clone Detection[4]

- A Differencing Algorithm for Object-Oriented Programs
  - Match declarations of classes, fields, and methods by name
  - Match content in methods by hammock graphs
    - A hammock is a single entry, single exit subgraph of a CFG

Example: Enhanced CFG comparison for P and P'



Hammock Graph Creation



Algorithm

- Input: hammock node n, n', look-ahead threshold LH
- Output: set of matched pairs N
- Algorithm

```
expand n and n' one level to graph G and G'
Push start node pair <s, s'> to stack ST
while ST is not empty
    pop <c, c'> from ST
    if c or c' is already matched then
        continue;
    if <c, c'> does not match then
        compare c with LH successors of c' or
        compare c' with LH successors of c until finding a match
    if a match is found then
        N = N U {c, c', "unchanged"}
    else
        N = N U {c, c', "modified"}
    push the pair's sink node pair on stack
```

Observations

- The look-ahead process is like bounded LCS algorithm
  - It can tolerate statement insertions at the same level
- The algorithm starts from the outmost Hammock, so it is similar to top-down tree-differencing algorithm
- When statements are inserted at the higher level, the algorithm does not work well
  - <c, c', "modified">

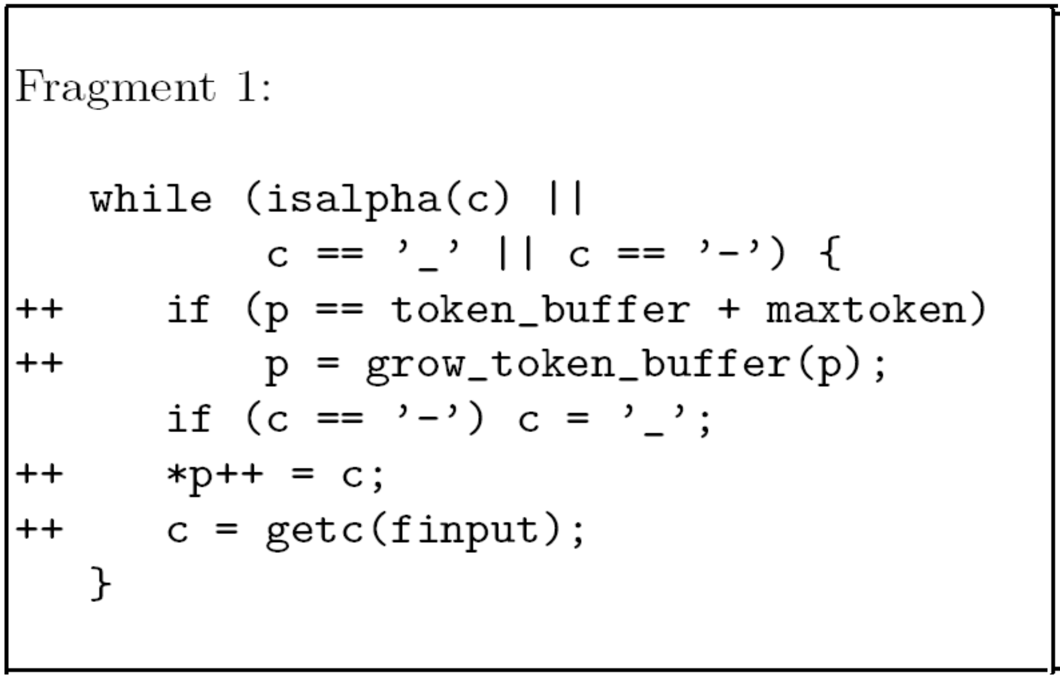
PDG-based Clone Detection [5]

- Using Slicing to Identify Duplication in Source Code
  - Step 1: Partition PDG nodes into equivalence classes based on the syntactic structure, such as while-loops
  - Step 2: For each pair of matching nodes (r1, r2), find two isomorphic subgraphs containing r1 and r2

Algorithm to Find Isomorphic Subgraphs

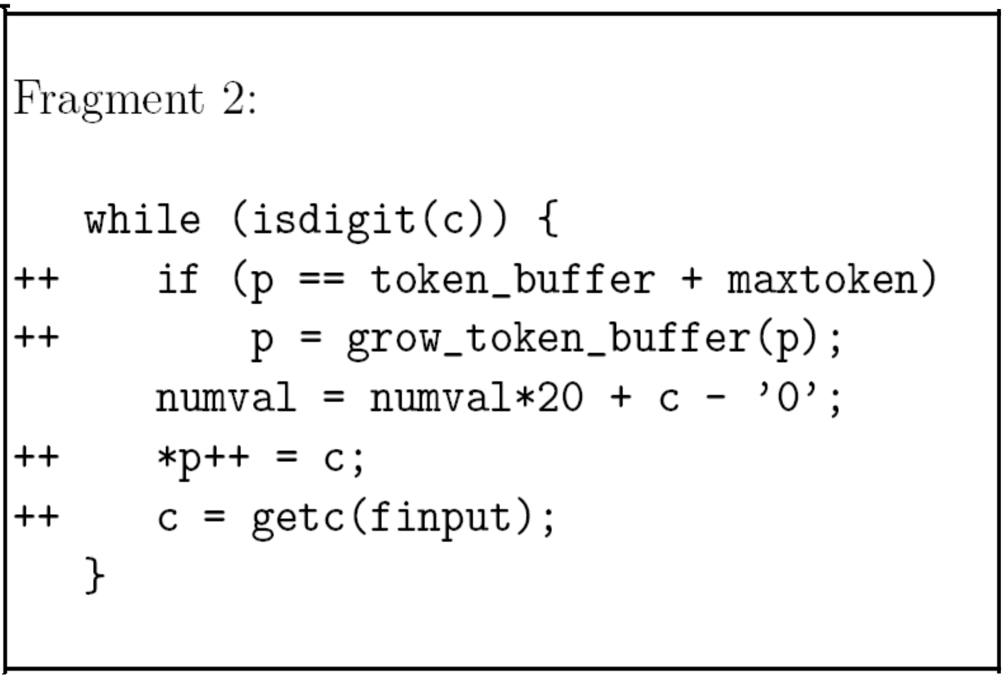
1. Start from r1 and r2, use backward slicing in lock step to add predecessors iff predecessors also match
2. If two matching nodes are loops or if-statements, forward slicing is also used to find control dependence successors (statements contained in the structure)

Example



Fragment 1:

```
   while (isalpha(c) ||
          c == ’_’ || c == ’-’) {
++     if (p == token_buffer + maxtoken)
++         p = grow_token_buffer(p);
       if (c == ’-’) c = ’_’;
++     *p++ = c;
++     c = getc(finput);
   }
```



Fragment 2:

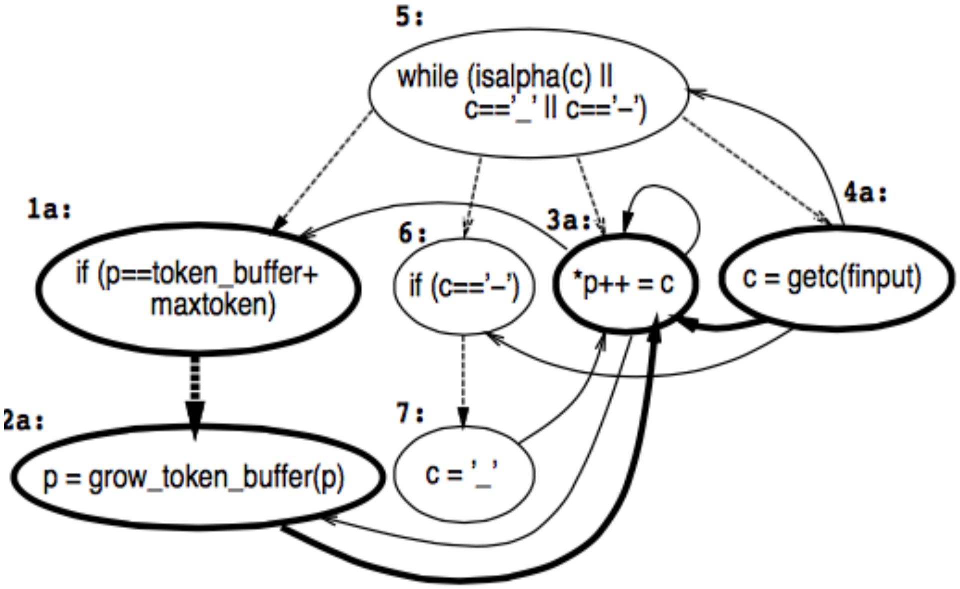
```
   while (isdigit(c)) {
++     if (p == token_buffer + maxtoken)
++         p = grow_token_buffer(p);
       numval = numval*20 + c - ’0’;
++     *p++ = c;
++     c = getc(finput);
   }
```

Fragment 1:

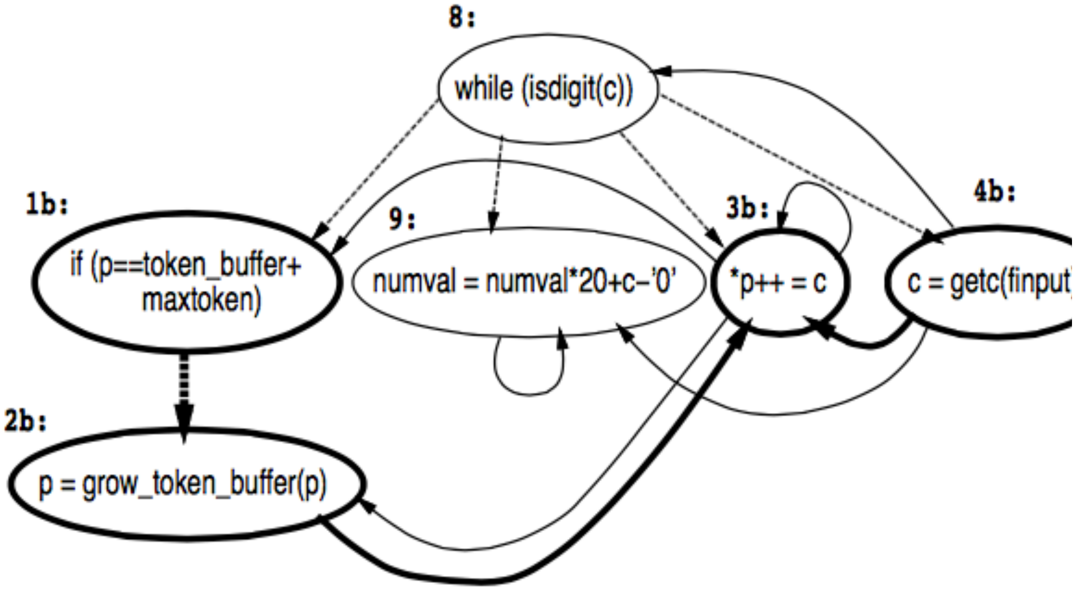
```
   while (isalpha(c) ||
          c == ’_’ || c == ’-’) {
++    if (p == token_buffer + maxtoken)
++        p = grow_token_buffer(p);
      if (c == ’-’) c = ’_’;
++    *p++ = c;
++    c = getc(finput);
   }
```

Fragment 2:

```
   while (isdigit(c)) {
++    if (p == token_buffer + maxtoken)
++        p = grow_token_buffer(p);
      numval = numval*20 + c - ’0’;
++    *p++ = c;
++    c = getc(finput);
   }
```



PDG for Fragment 1



PDG for Fragment 2

---------→ Control dependence

———→ Data dependence

Observations

- Pros
  - Tolerate statement reordering and some program structure changes
- Cons
  - Expensive
    - Points-to analysis
  - Do not allow ambiguous match

Summary

- Clone detection flexibility
  - PDG > CFG|AST > Token > Text
- Cost
  - Text < Token < CFG|AST < PDG

Fine-grained and Accurate Source Code Differencing [6]

Problem Statement

- Existing approaches usually represent code changes or edit operations as line addition or deletion
- Such representations are not precise
  - E.g., code move or update is not properly represented

Contributions

- GumTree—a novel efficient AST differencing algorithm that includes move actions
- An automated evaluation of GumTree
- A manual evaluation to compare GumTree vs. textual diff
- An automated evaluation to compare GumTree vs. ?

The GumTree Algorithm

1. A greedy top-down algorithm to find isomorphic sub-trees of decreasing height. Mappings are established between the nodes of these isomorphic subtrees. They are called anchor mappings.

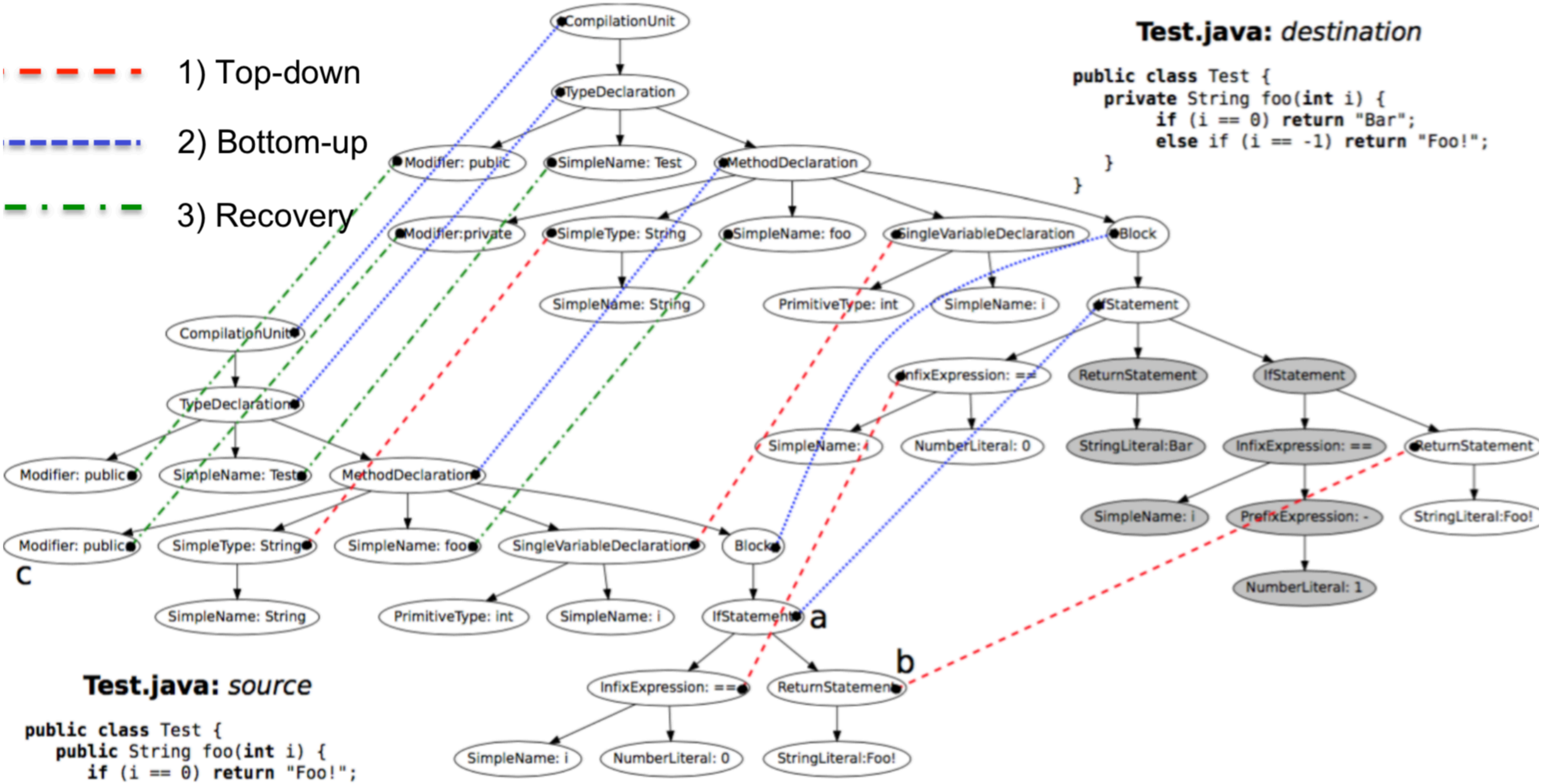
The GumTree Algorithm (cont’d)

2. A bottom-up algorithm where two nodes match (called a container mapping) if their descendants (children of the nodes, and their children, and so on) include a large number of common anchors. When two nodes match, we finally apply an optimal algorithm to search for additional mappings (called recovery mappings) among their descendants.

The GumTree Algorithm (cont'd)

3. Recovery Mappings: to find additional mappings between leaf nodes and similar nodes

4. Generate edit operations for the unmatched nodes:
   - Insert
   - Delete
   - Update
   - Move



1) Top-down

2) Bottom-up

3) Recovery

Test.java: ***destination***

```
public class Test {
   private String foo(int i) {
      if (i == 0) return "Bar";
      else if (i == -1) return "Foo!";
   }
}
```

Test.java: ***source***

```
public class Test {
   public String foo(int i) {
      if (i == 0) return "Foo!";
   }
}
```

Top-Down Phase

- Start with the roots and check if they are isomorphic or identical. If not, the children nodes are then tested
- To identify the unchanged part
- Implementation
  - By hardcoding subtrees, the isomorphism test's complexity is O(1)
  - The worst-case complexity is O(n^2)

Bottom-Up Phase

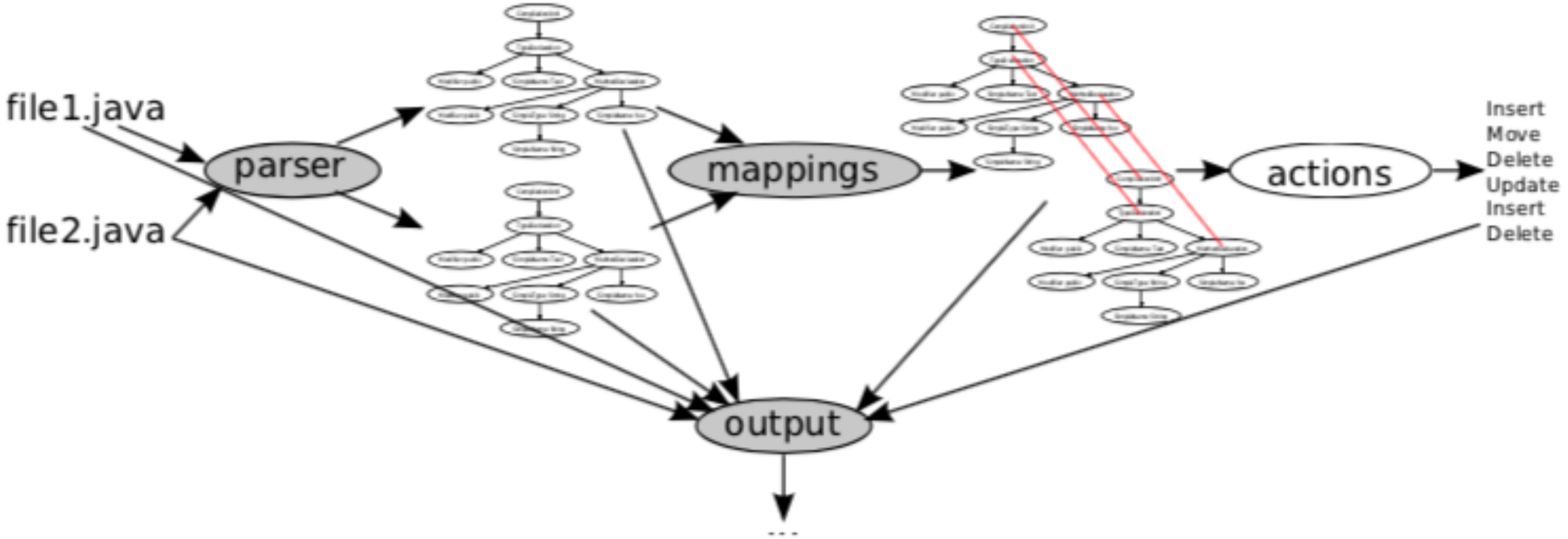
- Search for container mappings, that are established when two nodes have a significant number of matching descendants

$$\mathrm{dice}(t_1, t_2, \mathcal{M}) = \frac{2\times|\{t_1\in s(t_1)|(t_1,t_2)\in\mathcal{M}\}}{|s(t_1)|+|s(t_2)|}$$

Recovery Mappings

- Given two trees, find their additional mappings between the descendants,
  - remove the matched descendants, and
  - apply an optimized algorithm to find a shortest edit script without move actions

Architecture

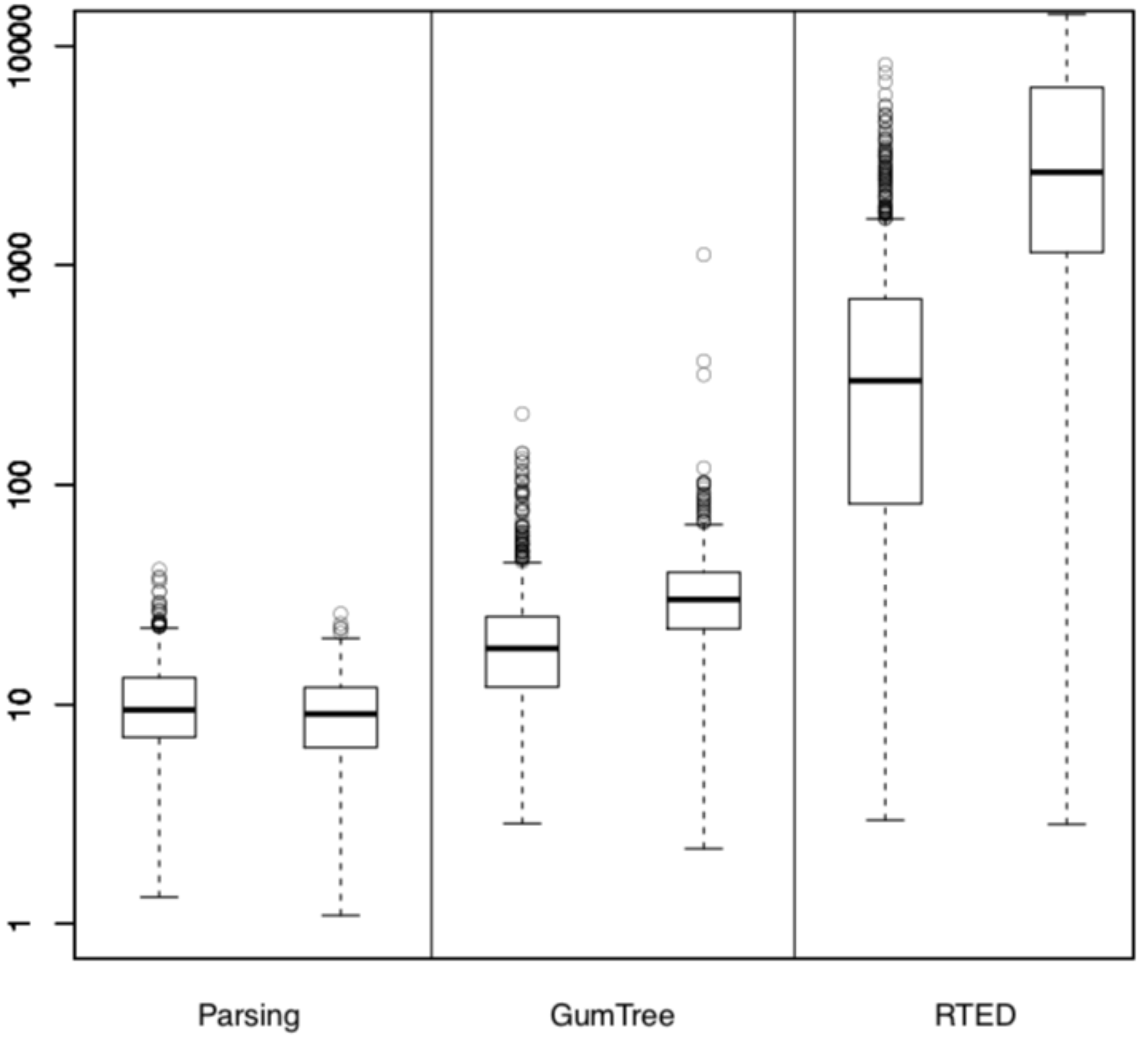


- Parser: Java, JavaScript, R, and C
- Mappings: GumTree, ChangeDistiller, XYDiff, RTED
- Output: XML representation of AST, web-based view of an edit script, XML representation of an edit script

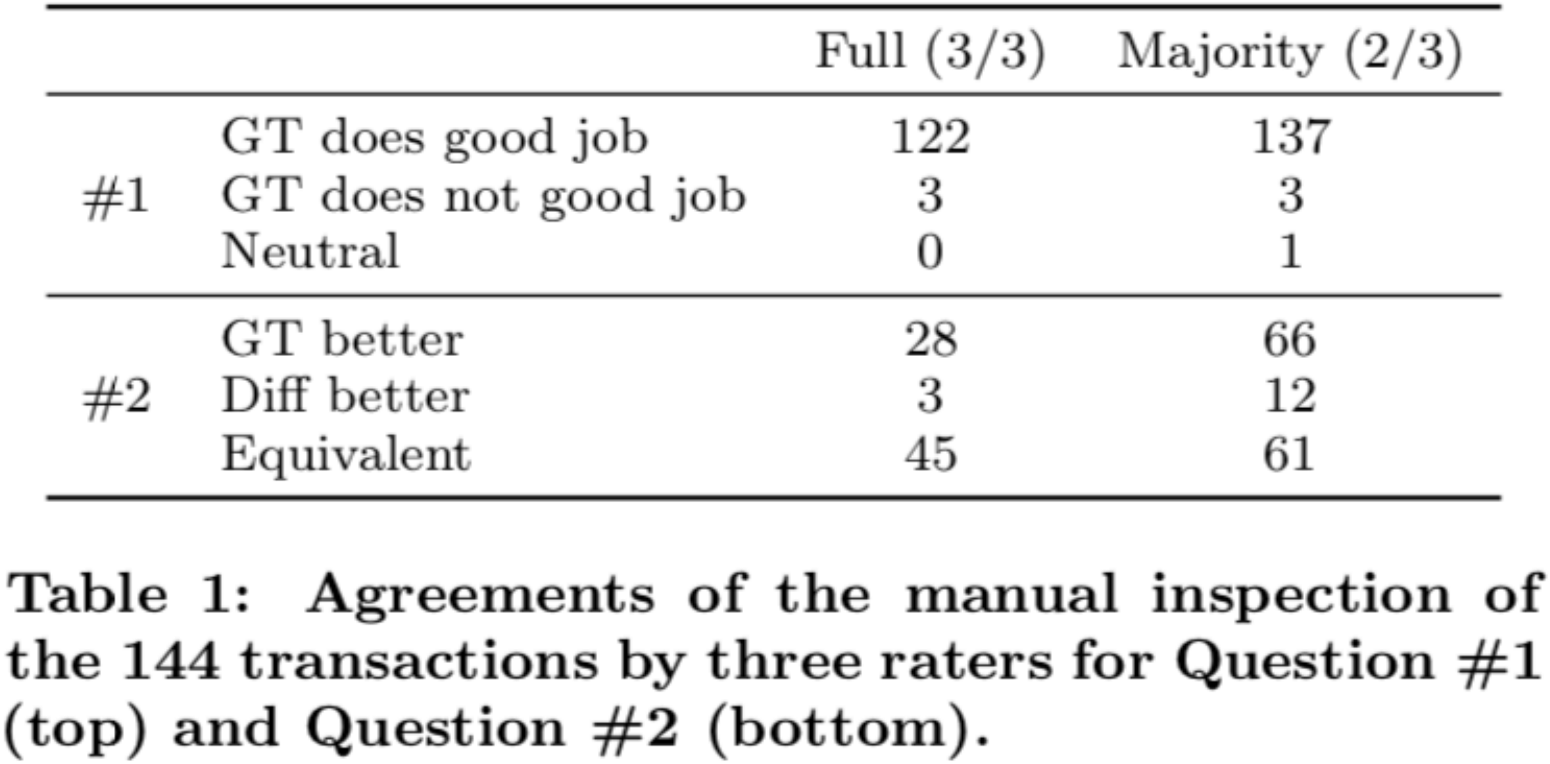
Evaluation

Comparison between GumTree, textual diff, and RTED

- The median of parsing time is 10
- Computing an edit script is only slightly slower than just parsing the files (median at 18 for Jenkins and 30 for JQuery



Manual Evaluation



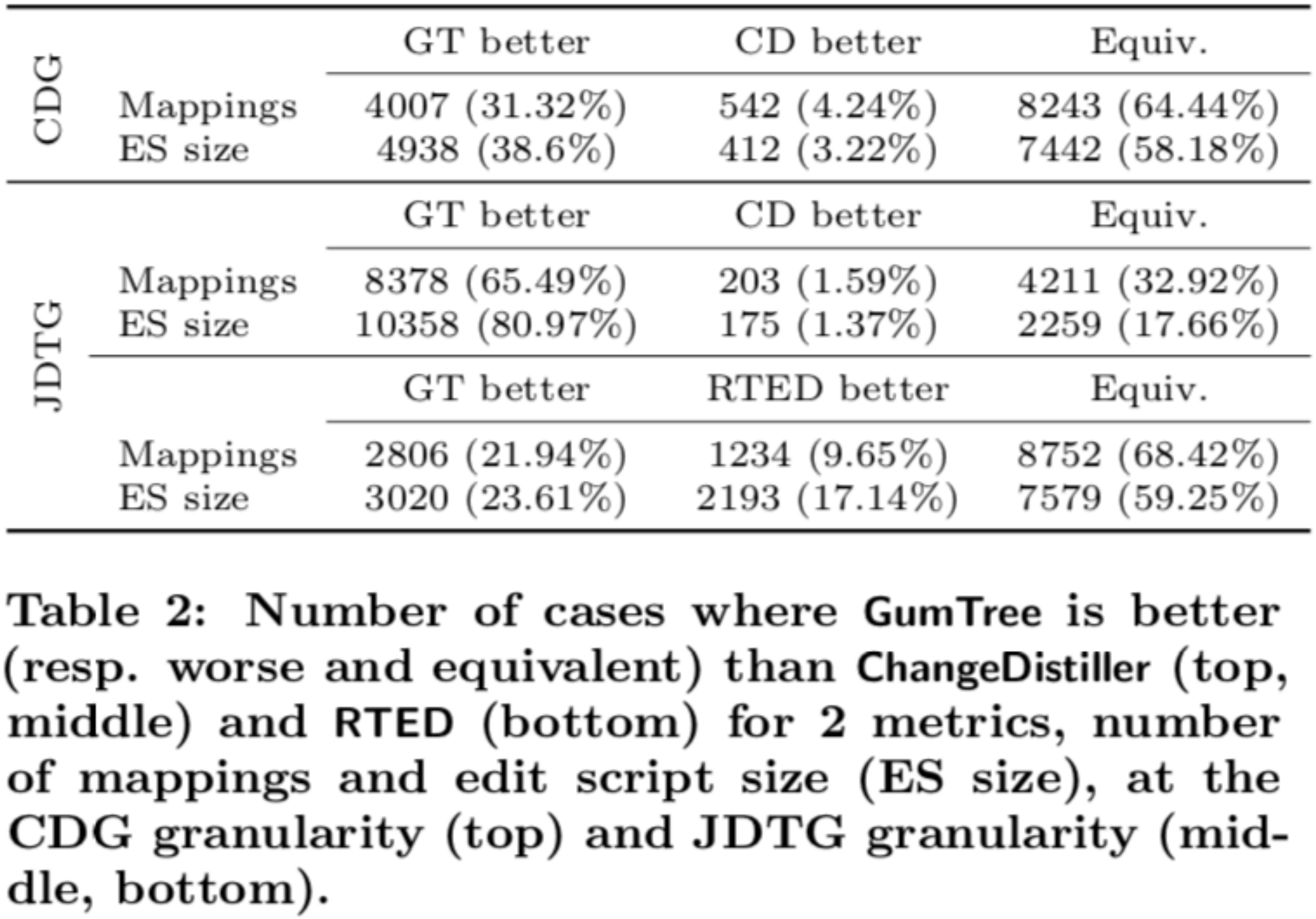
		Full (3/3)	Majority (2/3)
#1	GT does good job	122	137
	GT does not good job	3	3
	Neutral	0	1
#2	GT better	28	66
	Diff better	3	12
	Equivalent	45	61

Table 1: Agreements of the manual inspection of the 144 transactions by three raters for Question #1 (top) and Question #2 (bottom).

- GumTree's output is sometimes better than textual diff

Automatic Evaluation

- More matches = better

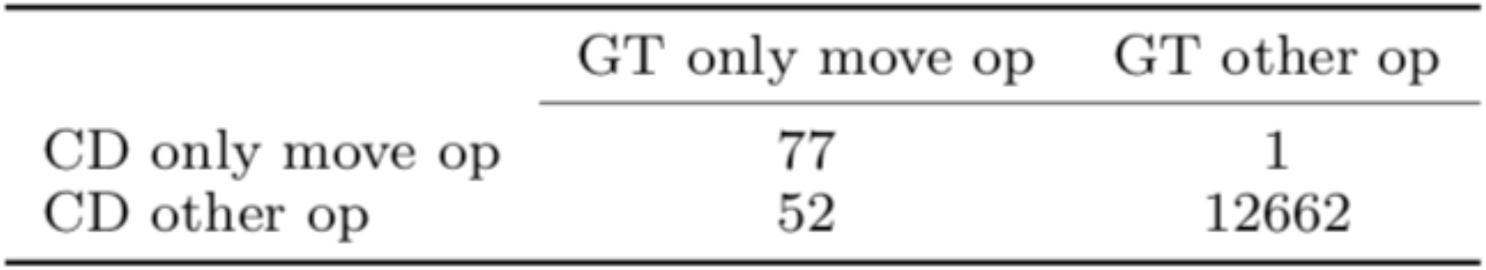


		GT better	CD better	Equiv.
CDG	Mappings	4007 (31.32%)	542 (4.24%)	8243 (64.44%)
	ES size	4938 (38.6%)	412 (3.22%)	7442 (58.18%)
JDTG		GT better	CD better	Equiv.
	Mappings	8378 (65.49%)	203 (1.59%)	4211 (32.92%)
	ES size	10358 (80.97%)	175 (1.37%)	2259 (17.66%)
		GT better	RTED better	Equiv.
	Mappings	2806 (21.94%)	1234 (9.65%)	8752 (68.42%)
	ES size	3020 (23.61%)	2193 (17.14%)	7579 (59.25%)

Table 2: Number of cases where GumTree is better (resp. worse and equivalent) than ChangeDistiller (top, middle) and RTED (bottom) for 2 metrics, number of mappings and edit script size (ES size), at the CDG granularity (top) and JDTG granularity (middle, bottom).

Automatic Evaluation (cont'd)

- GumTree generates smaller edit scripts in most cases than RTED and ChangeDistiller
  - 130 elements include move-only actions



	GT only move op	GT other op
CD only move op	77	1
CD other op	52	12662

Table 3: Comparison of the number of move operations from GumTree and ChangeDistiller for 12 792 file pairs to be compared.

References

[1] Spiros Mancoridis, Code Cloning: Detection, Classification, and Refactoring, https://www.cs.drexel.edu/~spiros/teaching/CS675/slides/code_cloning.ppt .

[2] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, CCFinder, A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, TSE '02

[3] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou, CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code, OSDI '04

References

[4] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold, A Differencing Algorithm for Object-Oriented Programs, ASE '04

[5] Raghavan Komondoor, Susan Horwitz, Using Slicing to Identify Duplication in Source Code, SAS '01

[6] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14).