

## Code duplication

Alexander Serebrenik



**TU** / **e**

Technische Universiteit  
**Eindhoven**  
University of Technology

Where innovation starts

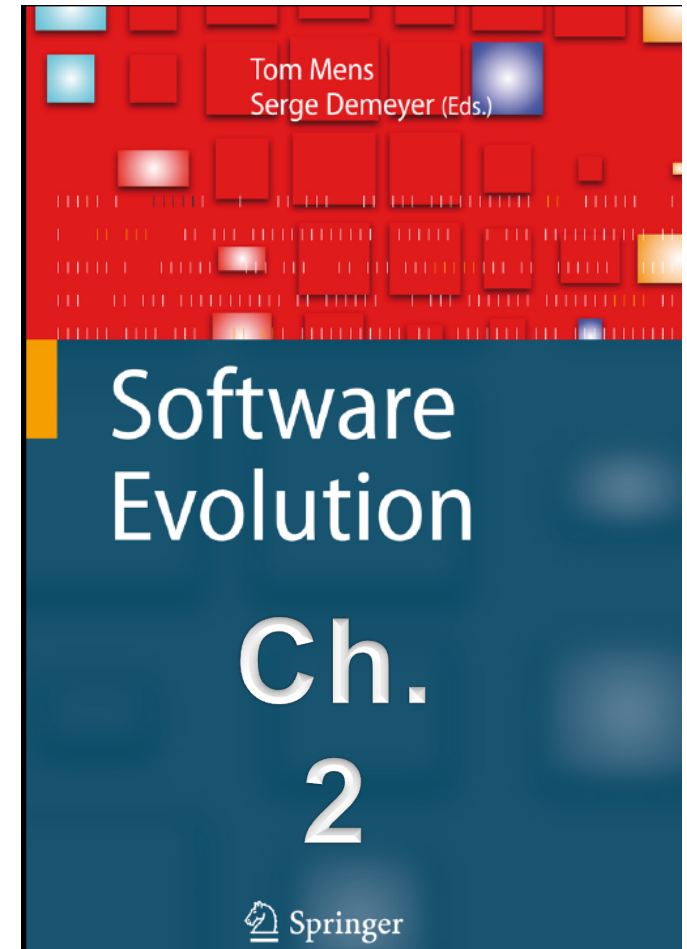
# Assignments

- **Assignment 2: March 2, 2015, 23:59.**
- **Assignment 3 already open.**
  - **Code duplication**
  - **Individual**
  - **Deadline: March 17, 2015, 23:59.**

# Sources

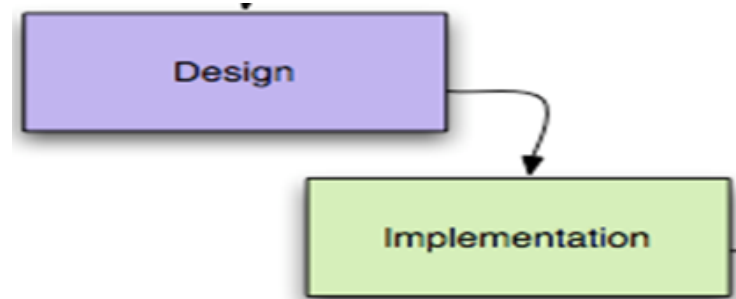


**“Clone detection” slides**  
**Rainer Koschke (in German)**  
<http://www.informatik.uni-bremen.de/st/lehre/re09/softwareklone.pdf>



# Where are we now?

- Last time: architecture
  - Behaviour
    - static/dynamic,
    - sequence diagrams/state machines,
    - focusing/visualization



- This week: **code duplication**
  - Occurs in the code
  - Can reflect suboptimal architecture

# Duplication?

- Beck and Fowler, “Stink Parade of Bad Smells”: 1
- Common?

Author	System	Min. length (lines)	%
Baker (1995)	X Windows	30	19
Baker <i>et alii</i> (1998)	Process control	?	29
Ducasse <i>et alii</i> (1999)	Payroll	10	59

- Frequent and problematic!

# A rose by any other name

- **Popular terms**
  - **Software redundancy**
    - Not every type of redundancy is harmful
  - **Code cloning = Code duplication**
    - Clone is identical to the original form

# A rose by any other name

- Popular terms
  - Software redundancy
    - Not every type of redundancy is harmful
  - Code cloning = Code duplication
    - Clone is identical to the original form
- Questions
  1. When are two fragments to be considered as clones?
  2. When is cloning harmful/useful?
  3. How do the clones evolve?
  4. What can one do about clones: ignore, prevent, eliminate?
  5. How to detect and present the clones?

# Clones?

```
1586 {
1587     if( GlobalConfig.DEBUG_LEVEL & DEBUG_WARNINGS ) {
1588         printf( __STR_WARNING__MEM_ALLOC_FAILED,
1589             acModuleName, pMsg->ServerName );
1590     }
1591     if( rcv_id != 0 ) {
1592         pMsg->type = TYPE_MSGUNKNOWN;
1593         MsgReply ( rcv_id, 0, pMsg, MSG_LENGTH_ACK );
1594     }
1595     return( MIRPA_ERROR_MEM_ALLOC_FAILED );
1596 }

1173 {
1174     if( GlobalConfig.DEBUG_LEVEL & DEBUG_WARNINGS ) {
1175         printf( __STR_WARNING__MEM_ALLOC_FAILED,
1176             acModuleName, pMsg->ServerName );
1177     }
1178     if( rcv_id != 0 ) {
1179         pMsg->type = TYPE_MSGUNKNOWN;
1180         MsgReply ( rcv_id, 0, pMsg, MSG_LENGTH_ACK );
1181     }
1182     return( MIRPA_ERROR_MEM_ALLOC_FAILED );
1183 }
```



# Clones?

```
1586 {
1587     if( GlobalConfig.DEBUG_LEVEL & DEBUG_WARNINGS ) {
1588         printf( __STR_WARNING__MEM_ALLOC_FAILED,
1589             acModuleName, pMsg->ServerName );
1590     }
1591     if( rcv_id != 0 ) {
1592         pMsg->type = TYPE_MSGUNKNOWN;
1593         MsgReply ( rcv_id, 0, pMsg, MSG_LENGTH_ACK );
1594     }
1595     return( MIRPA_ERROR_MEM_ALLOC_FAILED );
1596 }

1173 {
1174     if( GlobalConfig.DEBUG_LEVEL & DEBUG_WARNINGS ) {
1175         printf( __STR_WARNING__MEM_ALLOC_FAILED,
1176             acModuleName, pMsg->ServerName );
1177     }
1178     if( rcv_id != 0 ) {
1179         pMsg->type = TYPE_MSGUNKNOWN;
1180         MsgReply ( rcv_id, 0, pMsg, MSG_LENGTH_ACK );
1181     }
1182     return( MIRPA_ERROR_MEM_ALLOC_FAILED );
1183 }
```

**Type 1**

# Clones?

```
4278 case TYPE_SHMEM:
4279     if( GlobalConfig.DEBUG_LEVEL & DEBUG_WARNINGS ) {
4280         printf( "%s: WARNING : SHMEM msg received after
4281             sending ANSWER \"%s\\n\",
4282         acModuleName,
4283         sMsgList.asTxMsg[ uiMsgHandle ].name );
4284     }
4285 return( MIRPA_ERROR_RX_UNEXPECTED_TYPE );
```

```
4270 case TYPE_MSGOK:
4271     if( GlobalConfig.DEBUG_LEVEL & DEBUG_INFO ) {
4272         printf( "%s: INFO : MSG_OK received after
4273             sending ANSWER \"%s\\n\",
4279         acModuleName,
4280         sMsgList.asTxMsg[ uiMsgHandle ].name );
4281     }
4282 return( MIRPA_OK );
```

# Clones?

```
4278 case TYPE_SHMEM:
4279     if( GlobalConfig.DEBUG_LEVEL & DEBUG_WARNINGS ) {
4280         printf( "%s: WARNING : SHMEM msg received after
4281             sending ANSWER \"%s\\n\",
4282         acModuleName,
4283         sMsgList.asTxMsg[ uiMsgHandle ].name );
4284     }
4285 return( MIRPA_ERROR_RX_UNEXPECTED_TYPE );
```

## Type 2

```
4270 case TYPE_MSGO:
4271     if( GlobalConfig.DEBUG_LEVEL & DEBUG_INFO ) {
4272         printf( "%s: INFO : MSG_OK received after
4273             sending ANSWER \"%s\\n\",
4279         acModuleName,
4280         sMsgList.asTxMsg[ uiMsgHandle ].name );
4281     }
4282 return( MIRPA_OK );
```

# Clones?

```
if ( ! parse( ) ) {  
    print_error(stdout , 0) ;  
    return FALSE ;  
}  
  
fclose( fp ) ;  
  
if ( debug_flag ) {  
    printf(" result of parser " ) ;  
    if ( ! print_tree( FALSE ) ) {  
        print_error(stdout , 0) ;  
        return FALSE ;  
    }  
}
```

```
if ( ! type_check( ) ) {  
    print_error(stdout , 0) ;  
    return FALSE ;  
}  
  
if ( debug_flag ) {  
    printf(" result of type check" ) ;  
    if ( ! print_tree( TRUE ) ) {  
        print_error(stdout , 0) ;  
        return FALSE ;  
    }  
}
```

# Clones?

```
if ( ! parse( ) ) {  
    print_error(stdout , 0) ;  
    return FALSE ;  
}  
  
fclose( fp ) ;  
  
if ( debug_flag ) {  
    printf(" result of parser ") ;  
    if ( ! print_tree( FALSE ) )  
        print_error(stdout , 0) ;  
    return FALSE ;  
}  
}
```

```
if ( ! type_check( ) ) {  
    print_error(stdout , 0) ;  
    return FALSE ;  
}  
  
if ( debug_flag ) {  
    printf(" result of type check") ;  
    if ( ! print_tree( TRUE ) ) {  
        print_error(stdout , 0) ;  
        return FALSE ;  
    }  
}
```

# Clones?

```
/*  
By Bob Jenkins, 1996. hashtab.h Public Domain  
...  
htab *hcreate(/*_ word logsize _*/);  
  
void hdestroy(/*_ htab *t _*/); */  
...
```

## Type 4

```
/* Copyright (C) 2002 Christopher Clark  
<firstname.lastname@cl.cam.ac.uk> */  
...  
struct hashtable  
*create_hashtable(unsigned int minsize,  
                  unsigned int (*hashfunction) (void*),  
                  int (*key_eq_fn) (void*,void*));  
...  
void  
hashtable_destroy(struct hashtable *h, int free_values);
```

# Types are too rough!

If we want to eliminate the duplicates we need to understand the differences between them!

**Method clones**

**[Balazinska et al. 1999]**

**3-9 – one token only**

**10-12 – aggregated changes**

- **Interface: 3-6**
- **Implementation: 7-9**
- **Interface and implem.: mix**

Category number	Type of clones
1	Identical
2	Superficial changes
3	Called methods
4	Global variables
5	Return type
6	Parameters types
7	Local variables
8	Constants
9	Type usage
10	Interface changes
11	Implementation changes
12	Interface and implementation changes
13	One long difference
14	Two long differences
15	Several long differences
16	One long difference, interface and implementation
17	Two long differences, interface and implementation
18	Several long differences, interface and implementation

# Structural classification [Kapser et alii 2003]

- **Alternative based on the locations of the clones.**
- **Intra-file or inter-file cloning**
- **Type of location:**
  - **function, declaration, macro, hybrid, other (typedef)**
- **Type of the code sequence**
  - **initialization, finalization, loop, switch**



# Q1: Two fragments are clones if...

- **Type 1:** They are identical up to whitespace/comments
- **Type 2:** They are structurally identical (rename variables, types or method calls)
- **Type 3:** They are similar but statements/expressions could have been added, removed or modified
- **Type 4:** They implement the same concepts
- **Alternative classifications** have been proposed:
  - [Balazinska et al. 1999] based on the differences
  - [Kapser et al. 2003] based on the location

## Q2: Is cloning bad? Good reasons for cloning

- **Improves reliability**
  - *n*-version programming, IEC 61508
- **Reduces development time**
  - “Copy and modify” is faster than “generalize”
- **Avoids breaking the existing code**
  - Re-testing effort might be prohibitive
- **Clarifies structure**
  - E.g., disentangles dependencies (but do not overdo!)
- **By lack of choice**
  - Programming language does not provide appropriate flexibility mechanisms

# However (bad news)...

- **More code**
  - More effort required to comprehend, test and modify
  - Higher resource usage
- **Interrelated code**
  - Bug duplication
  - Incomplete or inconsistent updates
- **Indicative of**
  - Poor or decaying architecture
  - Lack of appropriate knowledge sharing between the developers

# Even more: duplication and bugs

- **[Monden et al. 2002]**
  - 2000 modules, 1MLOC Cobol
  - Most errors in modules with  $\geq 200$  LOC cloned
  - Many errors in modules with  $\leq 50$  LOC cloned
  - Least errors in modules with 50-100 LOC clones
  - No explanation of this phenomenon
- **[Chou et al. 2001]**
  - Linux and Open BSD kernels
  - In presence of clones: one error  $\Rightarrow$  many errors

# Empirical evaluation of clones [Godfrey, Kapser, 2011]

- **Systems:** Apache 2.2.4, Gnumeric 1.6.3
- **Clone groups:** 1580 in Apache, 3437 in Gnumeric
- **Randomly select 100 clone groups from each system**
- **Manual review:**
  - *Drop:* not really a clone group
  - *Good:* no better solution is possible
  - *Harmful:* better solution is obvious, cloning due to laziness
  - *Unavoidable:*
    - either too trivial to bother
    - or no alternative is possible (often the case with APIs)

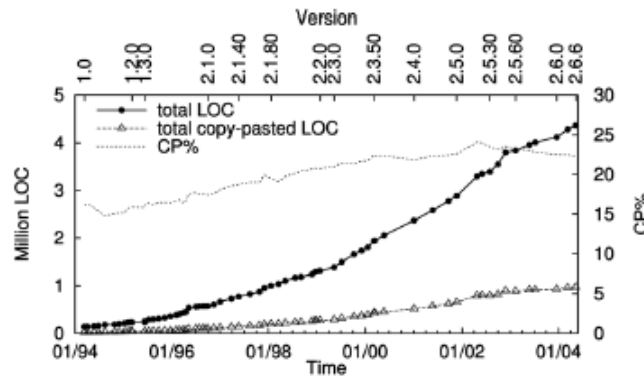
# How harmful is cloning (3)

Category	Pattern	Apache: Good	Apache: Unavoidable	Apache: Harmful	Gnumeric: Good	Gnumeric: Unavoidable	Gnumeric: Harmful
Forking	Hardware variation	0	0	0	0	0	0
Forking	Platform variation	10	0	0	0	0	0
Forking	Experimental variation	4	0	0	0	0	0
Templating	Boilerplating	5	0	0	6	0	1
Templating	API protocols	0	17	0	0	8	1
Templating	Programming idioms	0	0	12	1	0	0
Templating	Parameterized code	5	1	12	10	0	24
Customizing	Bug workarounds	0	0	0	0	0	0
Customizing	Replicate and specialize	12	0	4	15	0	1
Other		3	0	8	1	0	3
Total		39	18	36	33	8	30

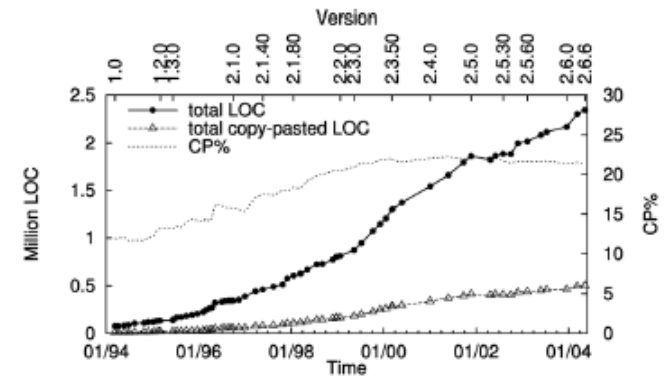
# Q3. How do the clones evolve?

Li et al.  
2006

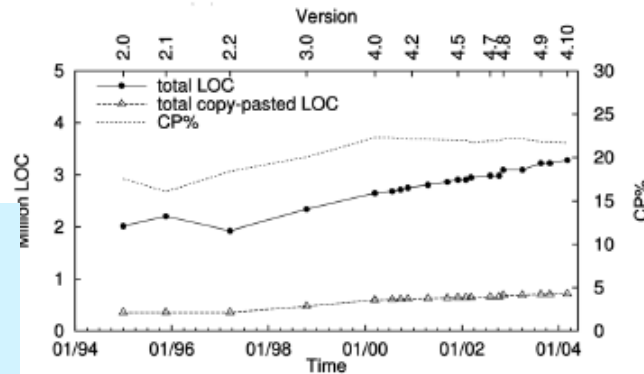
- a) Linux
- b) Linux “drivers”
- c) Free BSD
- d) Free BSD “sys”



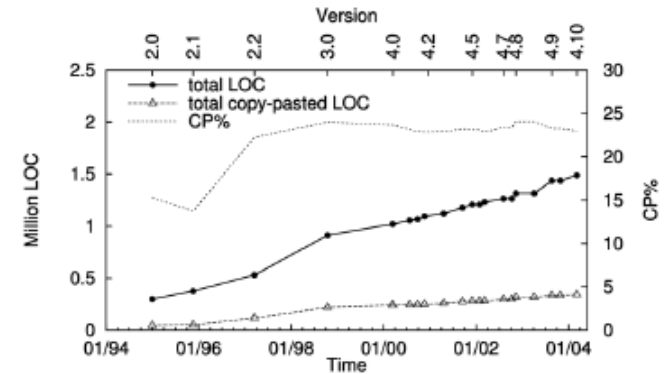
(a)



(b)



(c)



(d)

Increase  
followed by  
stabilization

## Q4. What can we do about clones?

- **Ignore: the simplest way**
- **Correct (eliminate):**
  - **Manual: design patterns**
  - **Automated:**
    - **Type 1 or 2 (variable names): function abstraction**
    - **Type 2 (types) or 3: macros, conditional compilation**
      - **The programming language should support it**
      - **Can make the code more complex**
    - **Develop code generators**
    - **Challenges:**
      - **how to invent meaningful names?**
      - **how to determine the appropriate level of abstraction?**



## Q4. What can we do about clones?

- **Prevent:**
  - Check on-the-fly while the code is being edited
  - Check during the check-in
- **Manage**
  - Link the clones (automatically or manually)
  - Once one of the clones is being modified the user is notified that other clones might require modification as well.

# Questions and answers so far...

## 1. When are two fragments to be considered as clones?

- Type 1, 2, 3, 4
- More refined classification possible

## 2. When is cloning harmful/useful?

- reliability, reduced time, structure?, code preservation
- more interrelated code, more bugs

## 3. How do the clones evolve?

- Increase followed by stabilization

## 4. What can one do about clones?

- ignore, eliminate, prevent (check on the fly), manage (link and notify the user upon change)

## Q5. How to detect clones?

- **Granularity**
  - **Classes, functions, statements**
- **Objects of comparison**
  - **Text, identifiers, tokens, AST, control and data dependencies**
- **Related techniques**
  - **textual diff, dot plot, data mining, suffix tree, tree and graph matching, latent semantic indexing, metric vector comparison, hashing**

# Basic challenges in clone detection

- **Pairwise comparison of classes, functions, lines**

- Naïve way:  $O(n^2)$

- Might

- **Type 2: Clones with different types, ...?**

- Renaming

- We still have to deal with clones that are not identical

**We are going to see how these challenges are addressed by different clone detection approaches.**

- **Type 3: Clones can be combined into larger clones**

- Clones can have “gaps”

- Identity vs. Similarity – similarity measures?

# Clone detection techniques

- **Text-based**
  - [Ducasse et al. 1999, Marcus and Maletic 2001]
- **Metrics-based**
  - [Mayrand et al. 1996]
- **Token-based**
  - [Baker 1995, Kamiya et al. 2002]
- **AST-based**
  - [Baxter 1996]
  - AST+Tokens combined [Koschke et al. 2006]
- **Program Dependence Graph**
  - [Krinke 2001]

# Textual comparison

- **Programs are just text!**
- **“Programming language independent”**
- **[Ducasse et al, 1999]**
  - **Remove whitespaces and comments**

This is the house that Jack built.

This is the rat  
That ate the malt  
That lay in the house that Jack  
built.

This is the cat,  
That killed the rat,  
That ate the malt  
That lay in the house that Jack  
built.

# Textual comparison

- **Programs are just text!**
- **“Programming language independent”**
- **[Ducasse et al, 1999]**
  - **Remove whitespaces and comments**
  - **Calculate hashes for code lines**

ThisisthehousethatJackbuilt.

Thisistherat  
Thatatethemalt  
ThatlayinthehousethatJackbuilt.

Thisisthecat,  
Thatkilledtherat,  
Thatatethemalt  
ThatlayinthehousethatJackbuilt.

# Textual comparison

- Programs are just text!
- “Programming language independent”
- [Ducasse et al, 1999]
  - Remove whitespaces and comments
  - Calculate hashes for code lines
  - Partition lines into classes based on hashes

ThisisthehousethatJackbuilt. **1**

Thisistherat **1**

Thatatethemalt **f**

ThatlayinthehousethatJackbuilt. **b**

Thisisthecat, **b**

Thatkilledtherat, **6**

Thatatethemalt **f**

ThatlayinthehousethatJackbuilt. **b**



# Textual comparison

- Programs are just text!
- “Programming language independent”
- [Ducasse et al, 1999]
  - Remove whitespaces and comments
  - Calculate hashes for code lines
  - Partition lines into classes based on hashes
  - Compare lines in the same partition

ThisisthehousethatJackbuilt. **1**  
Thisistherat

Thatkilledtherat, **6**

Thisisthecat, **b**  
ThatlayinthehousethatJackbuilt.  
ThatlayinthehousethatJackbuilt.

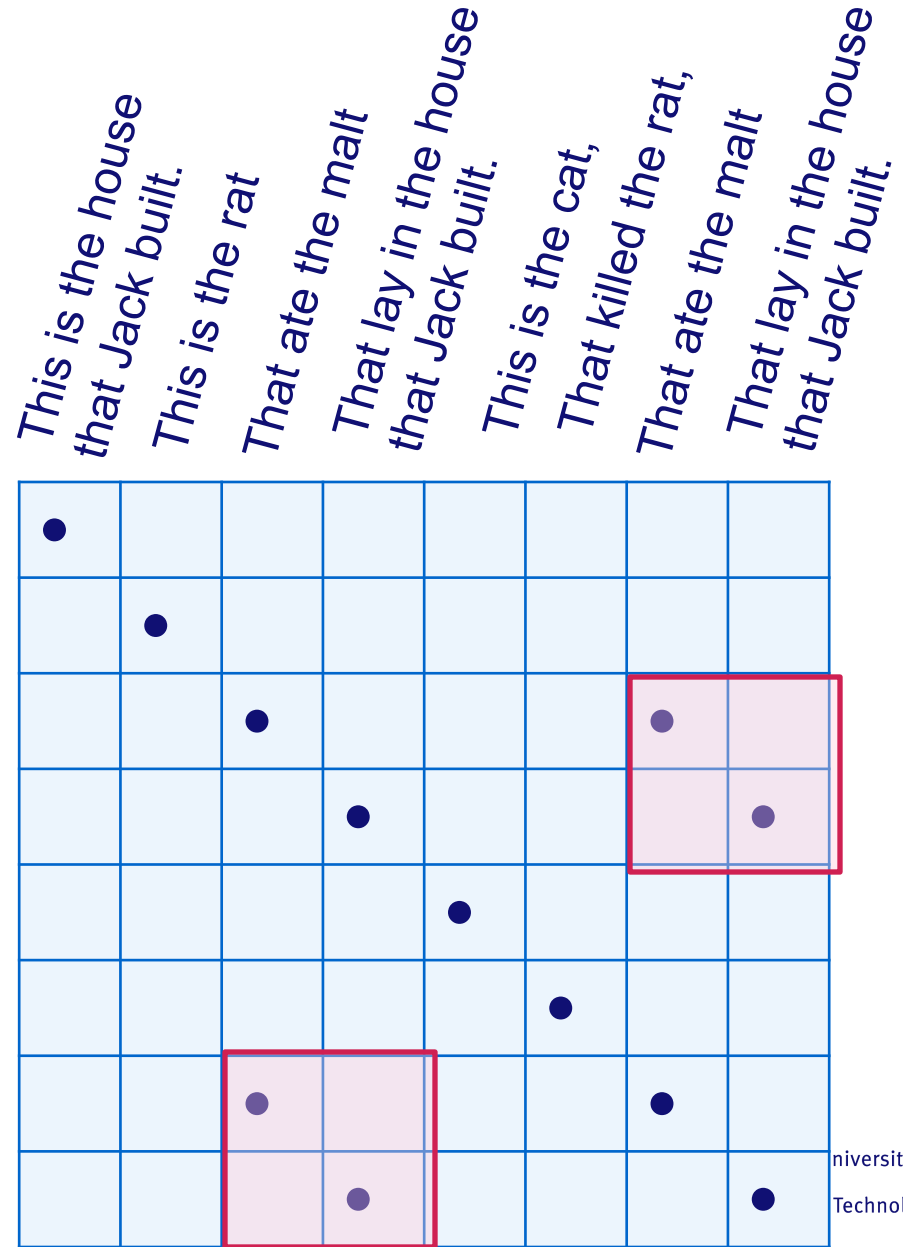
Thatatethemalt **f**  
Thatatethemalt

# Textual comparison

- **Programs are just text!**  
ThisisthehousethatJackbuilt.
- **“Programming language independent”**  
Thisistherat
- **[Ducasse et al, 1999]**
  - **Remove whitespaces and comments**  
Thatkilledtherat,
  - **Calculate hashes for code lines**  
Thisisthecat,
  - **Partition lines into classes based on hashes**  
ThatlayinthehousethatJackbuilt.  
ThatlayinthehousethatJackbuilt.
  - **Compare lines in the same partition**  
Thatatethemalt  
Thatatethemalt
  - **Visualize using dot plot**

# Textual comparison

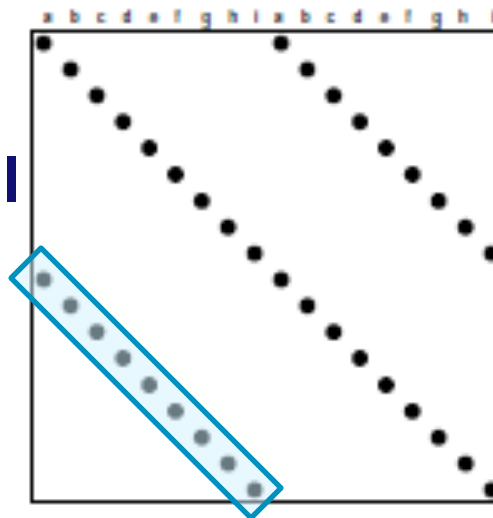
- Programs are just text!
- “Programming language independent”
- [Ducasse et al, 1999]
  - Remove whitespaces and comments
  - Calculate hashes for code lines
  - Partition lines into classes based on hashes
  - Compare lines in the same partition
  - Visualize using **dot plot**
  - Recognize larger clones by **dot plot patterns**



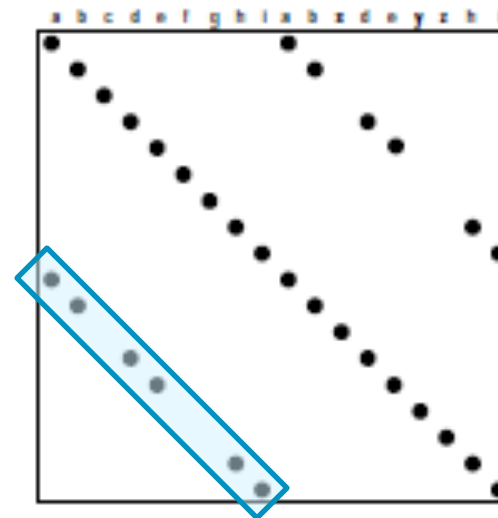
# Dot plot patterns

[Ducasse et al., 1999]

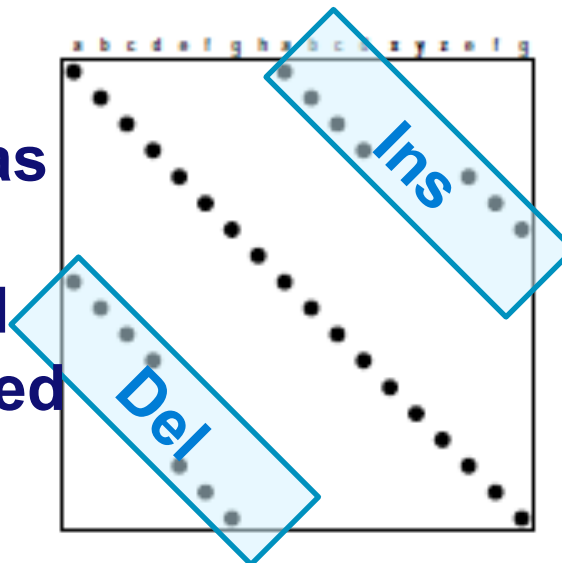
**Identical  
code  
clones,  
Type 1**



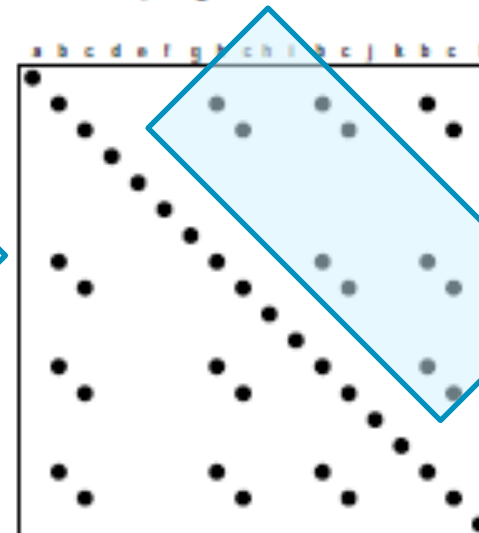
**Modified  
clones  
Type 2-3**



**Code has  
been  
inserted  
or deleted  
Type 3**



**Recurrent  
code (break;  
preprocess)**



# Advantages and disadvantages

- **Good news**
  - Language independent
  - Can detect Type 1,2,3 clones
- **Bad news**
  - **Granularity:** line of code, cannot detect duplication between parts of lines
  - **Almost no distinction between “important” and “not important” code parts**
    - Variable names
    - Syntactic sugar: `if (a==0) {b}`

# Alternative textual comparison approach

- [Marcus and Maletic 2001]: Clones discuss the same concepts
  - **Higher-level clones: Type 4!**
  - Identifier names should be the same!
    - If/while/... can be neglected
  - Latent semantic analysis (Information retrieval)
  - Mosaic 2.7, C, 269 files

**Linked lists:**  
**list.c, list.h,**  
**listP.h**

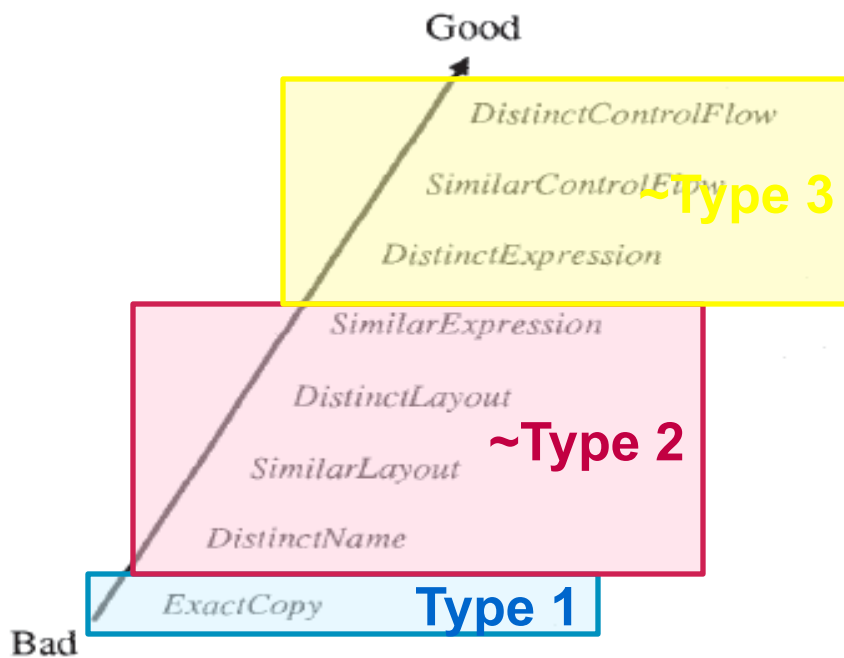
**Two additional**  
**implementations:**  
**hotlist and HTList**

**Two more!**

# Extending the text-based approach

- Program structure instead of text
- **Metrics** instead of hash-functions [Mayrand et al. 1996]
  - Name: identical or not
  - Layout (5 metrics):
    - avg variable name length, num of blank lines...
  - Expression (5 metrics):
    - num of calls, num of executable statements, ...
  - Control flow (11 metrics):
    - num of loops, num of decisions, ...
- Many metrics  $\Rightarrow$  lower chance of occasional collisions

# Metrics-based clone detection



Scale	Nam	Lay	Exp	Con
1-ExactCopy	=	=	=	=
2-DistinctName	!=	=	=	=
3-SimilarLayout	X	~	=	=
4-DistinctLayout	X	!=	=	=
5-SimilarExpression	X	X	~	=
6-DistinctExpression	X	X	!=	=
7-SimilarControlFlow	X	X	X	~
8-DistinctControlFlow	X	X	X	!=

- = all metrics are equal
- ~ some metrics not equal but all differences are within the allowed range (per metrics)
- != outside the range
- X not considered



# Metrics-based approaches: Discussion

- **Problems:**
  - Metrics are not independent (num uni calls  $\leq$  num calls)
  - “Allowed range” is arbitrarily chosen
  - Precision?
    - $\text{Code}_1 = \text{Code}_2 \Rightarrow \text{Metrics}(\text{Code}_1) = \text{Metrics}(\text{Code}_2)$
    - $\text{Code}_1 \sim \text{Code}_2 \Rightarrow \text{Metrics}(\text{Code}_1) \sim \text{Metrics}(\text{Code}_2)$
    - $\text{Metrics}(\text{Code}_1) = \text{Metrics}(\text{Code}_2) \Rightarrow \text{Code}_1 = \text{Code}_2$  ?
    - $\text{Metrics}(\text{Code}_1) \sim \text{Metrics}(\text{Code}_2) \Rightarrow \text{Code}_1 \sim \text{Code}_2$  ???
  - Precision can be improved if metrics are combined with textual comparison
    - Still  $O(n^2)$
    - But  $n$  is small for the “good choice” of metrics

# More fine-grained approaches: Tokens!

- [Baker 1995]
- We want to recognize  $x=x+y$  and  $u=u+v$  as clones
- Identify tokens in the code
  - Ignore the keywords.
- Split structure and parameters

```
j = length(list);
```

```
if (j < 3) { x = x + y; }
```

# More fine-grained approaches: Tokens!

- [Baker 1995]
- We want to recognize  $x=x+y$  and  $u=u+v$  as clones
- Identify tokens in the code
  - Ignore the keywords.
- Split structure and parameters
- For every structure invent an identifier

$\alpha$  `[ ] = [ ] ( [ ] )`

$\beta$  `if ( [ ] < [ ] ) { [ ] = [ ] + [ ] }`

j length list

j 3 x x y

# More fine-grained approaches: Tokens!

- [Baker 1995]
- We want to recognize  $x=x+y$  and  $u=u+v$  as clones
- Identify tokens in the code
  - Ignore the keywords.
- Split structure and parameters
- For every structure invent an identifier
- Drop the structures and merge the identifiers with the parameters: **P-string**
- Concatenate the P-strings

$\alpha$   `= ( )`

$\beta$   `if ( < ) { = + }`

j length list

j 3 x x y

# More fine-grained approaches: Tokens!

- [Baker 1995]
- We want to recognize  $x=x+y$  and  $u=u+v$  as clones

- Representation of the program so far:

- Encode the parameters:
  - First time encountered: 0
  - Next time: distance from the previous occurrence (structure identifiers included)

$\alpha$  j length list  $\beta$  j 3 x x y

$\alpha$  0 0 0  $\beta$  4 0 0 1 0

# More fine-grained approaches: Tokens!

- [Baker 1995]

- Clones – repeated fragments
- Construct a suffix tree
  - Represents all suffixes
  - Can be done in  $O(n)$
  - ~ Every branch represents a clone

$\alpha y \beta y \alpha x \alpha x$   
 $\alpha 0 \beta 2 \alpha 0 \alpha 2 \$$

$y \beta y \alpha x \alpha x$   
 $0 \beta 2 \alpha 0 \alpha 2 \$$

$\beta y \alpha x \alpha x$   
 $\beta 0 \alpha 0 \alpha 2 \$$

$0 \alpha 0 \alpha 2 \$$

$\alpha 0 \alpha 2 \$$

$0 \alpha 2 \$$

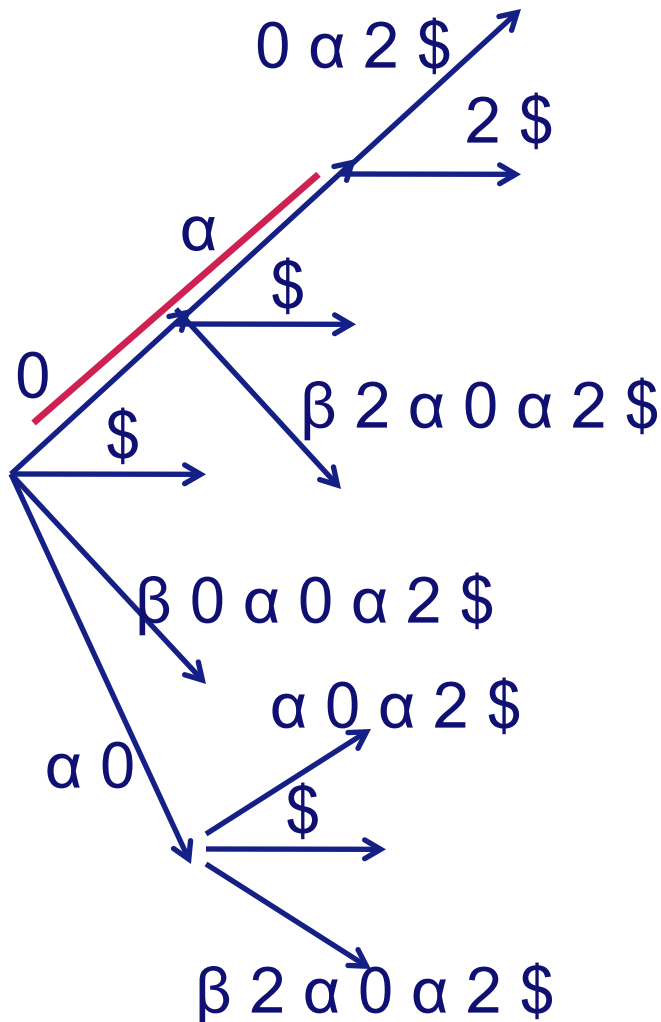
$\alpha 0 \$$

$0 \$$

$\$$

# More fine-grained approaches: Tokens!

- [Baker 1995]



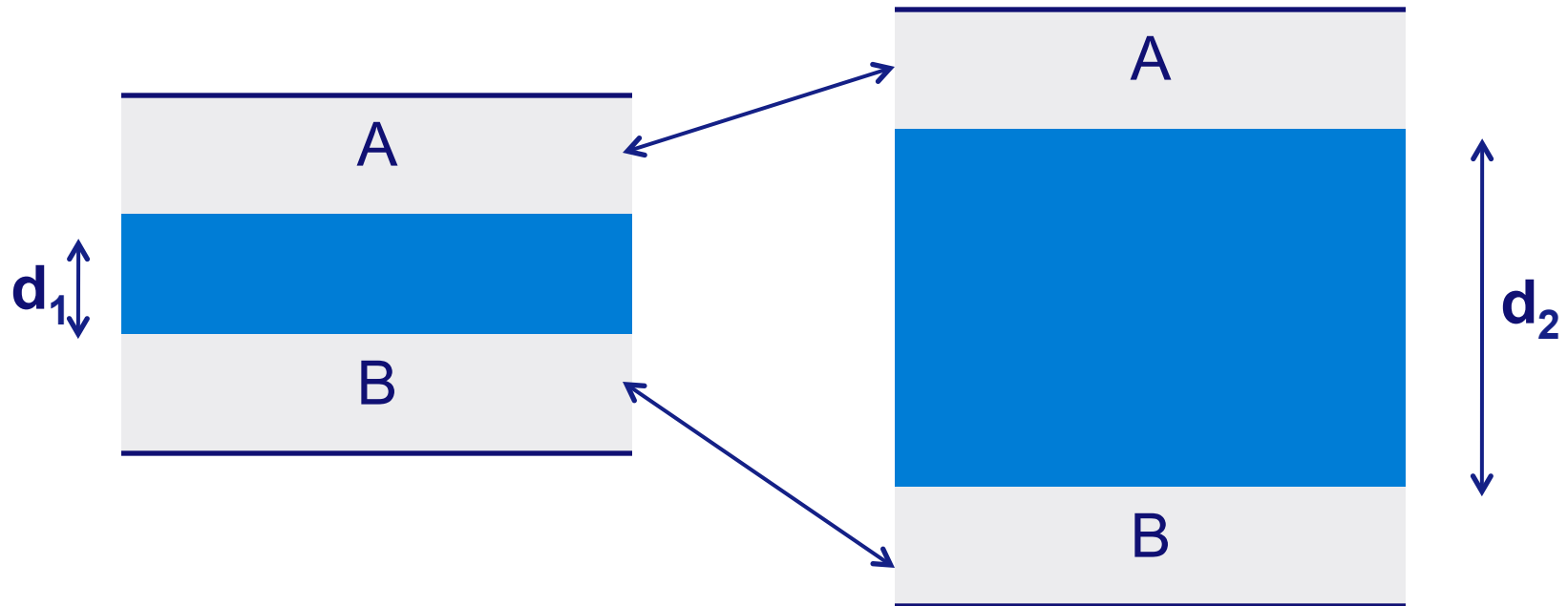
Every branch  
up to a leaf  
represents a  
clone

Size: count  
the symbols  
on the  
branches

α 0 β 2 α 0 α 2 \$  
0 β 2 α 0 α 2 \$  
β 0 α 0 α 2 \$  
0 α 0 α 2 \$  
α 0 α 2 \$  
0 α 2 \$  
α 0 \$  
0 \$  
\$

# So far only Type 1 and Type 2 clones

- Type 3 clones – combination of Type 1/2 clones



- Type 3 clones can be recognized if
  - $d_1 = d_2$
  - $\max(d_1, d_2) \leq \text{threshold}$



# Baker's approach

- **Very fast:**
  - 1.1 MLOC
  - minimal clone size: 30 LOC
  - 7 minutes on SGI IRIX 4.1, 40MHz, 256 MB
- **Close to language independence**
  - Depends solely on the **tokenizer**
- **Can be improved by code normalization**
  - See next slide
- **Can identify duplication across function borders**
  - Might require pre/post-processing

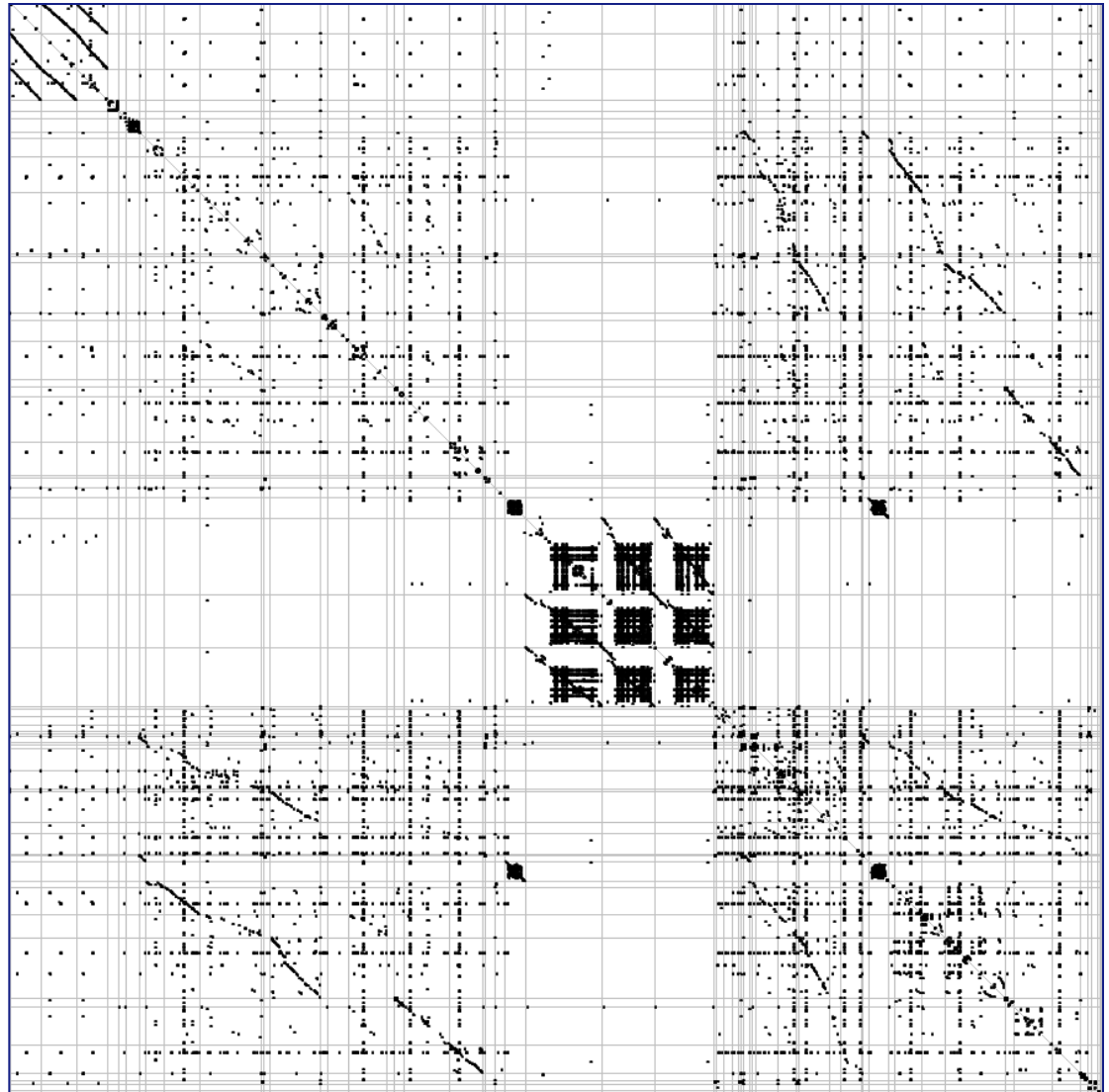
# Code normalization (Kamiya et al. 2002)

- Many ways to express the same intention

<code>x = y + x</code>	<code>x = x+ y</code>	Sort the operands of commutative operations lexicographically
<code>if (a == 1) x=1;</code>	<code>if (a == 1) {     x=1; }</code>	Add { } and newlines
static global variables in C		Drop “static”

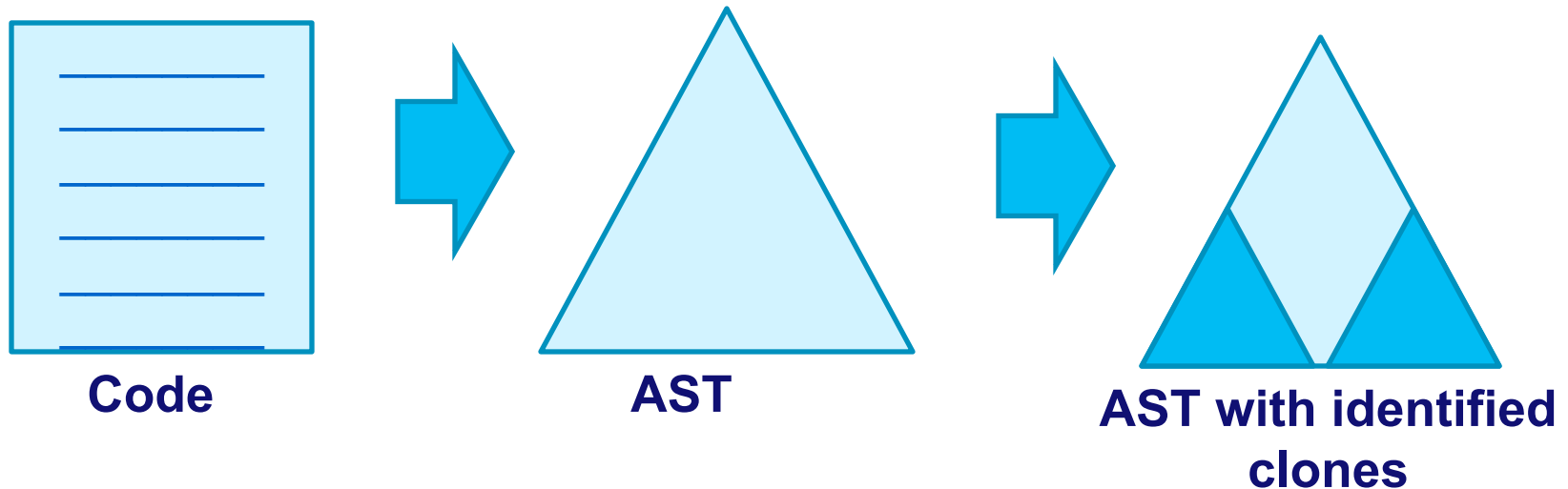
# Case study: Expert system of an insurance company [Kamiya – CCFinder/Gemini]

- **Diacritics elimination**
- **Product line like variants**



# AST-based clone detection [Baxter 1996]

- If we have a tokenizer we might also have a parser!
  - Applicability: the program should be parseable



- Compare every subtree with every other subtree?
  - For an AST of  $n$  nodes:  $O(n^3)$
- Similarly to text: Partitioning with a **hash** function
  - Works for Type 1 clones

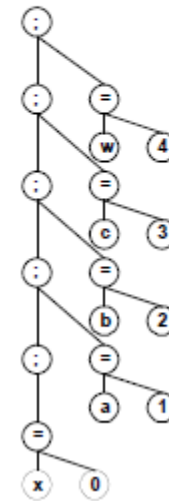
# AST-based detection

- **Type 2**
  - Either take a **bad** hash function ignoring small subtrees, e.g., names
  - Or replace identity by similarity

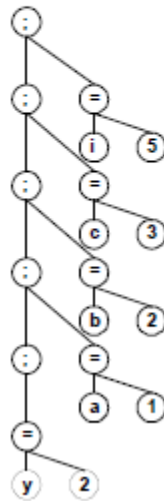
$$\text{Similarity}(T_1, T_2) = \frac{2 * \text{Same}(T_1, T_2)}{2 * \text{Same}(T_1, T_2) + \text{Difference}(T_1, T_2)}$$

- **Type 3**
  - Sequences of subtrees
  - Go from Type 2-cloned subtrees to their parents
- Rather precise but still slow

```
void f ()  
{  
  x=0;  
  a=1;  
  b=2;  
  c=3;  
  w=4;  
}
```



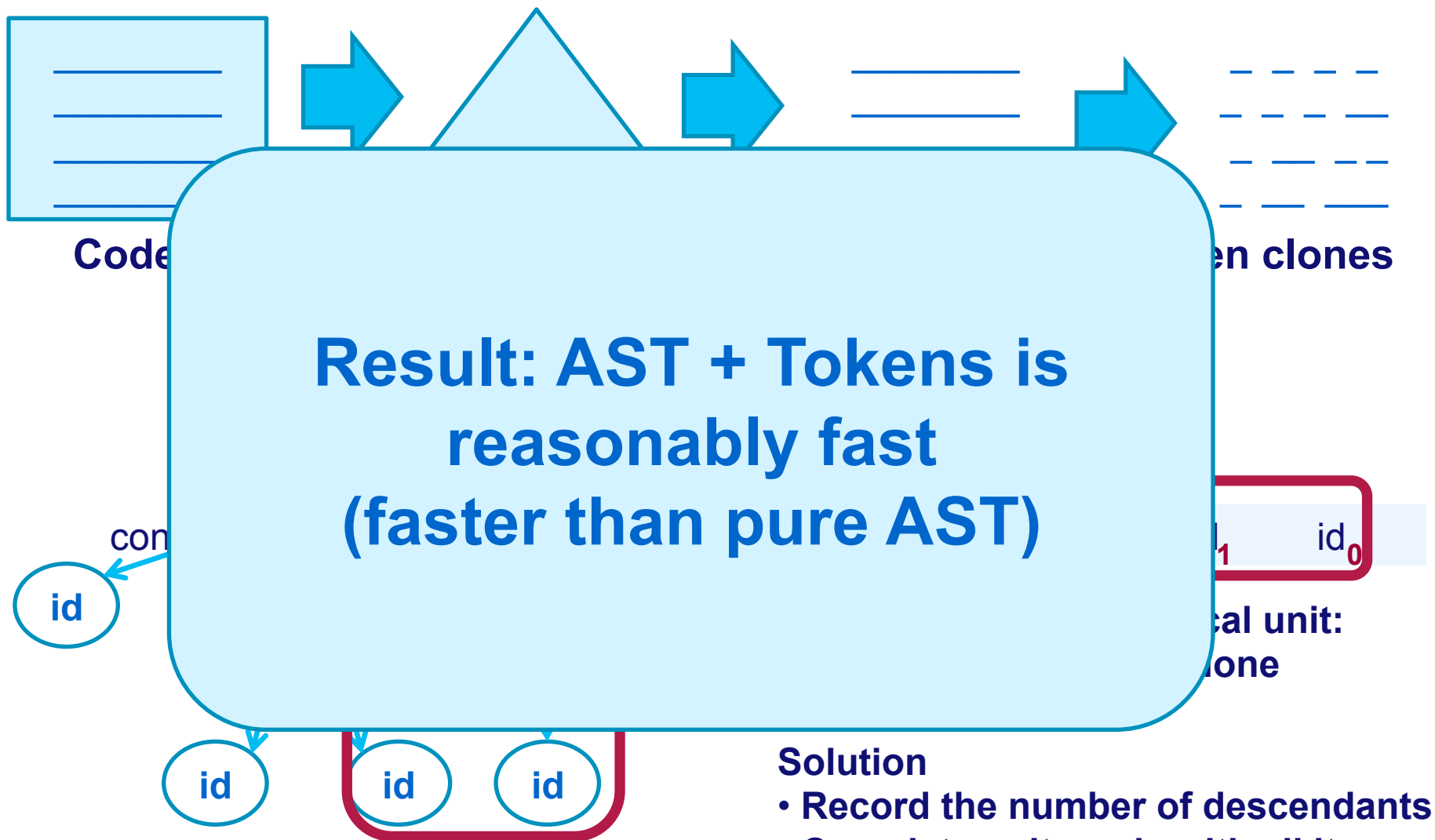
```
void g ()  
{  
  y=2;  
  a=1;  
  b=2;  
  c=3;  
  i=5;  
}
```



# Recapitulation from the last week

- **[Baker 1995]**
  - Token-based
  - Very fast:
    - 1.1 MLOC, minimal clone size: 30 LOC
    - 7 minutes on SGI IRIX 4.1, 40MHz, 256 MB
- **[Baxter 1996]**
  - AST-based
  - Precise but slow
- **Idea: Combine the two! [Koschke et al. 2006]**
  - In fact they do not use [Baker 1995] but a different token-based approach

# AST + Tokens [Koschke et al. 2006]



## Solution

- Record the number of descendants
- Complete unit: node with all its descendants

# Next step

- **AST is a tree is a graph**
- **There are also other graph representations**
  - **Object Flow Graph (weeks 3 and 4)**
  - **UML class/package/... diagrams**
  - **Program Dependence Graph**
- **These representations do not depend on textual order**
  - **{ x = 5; y = 7; } vs. { y = 7; x = 5; }**



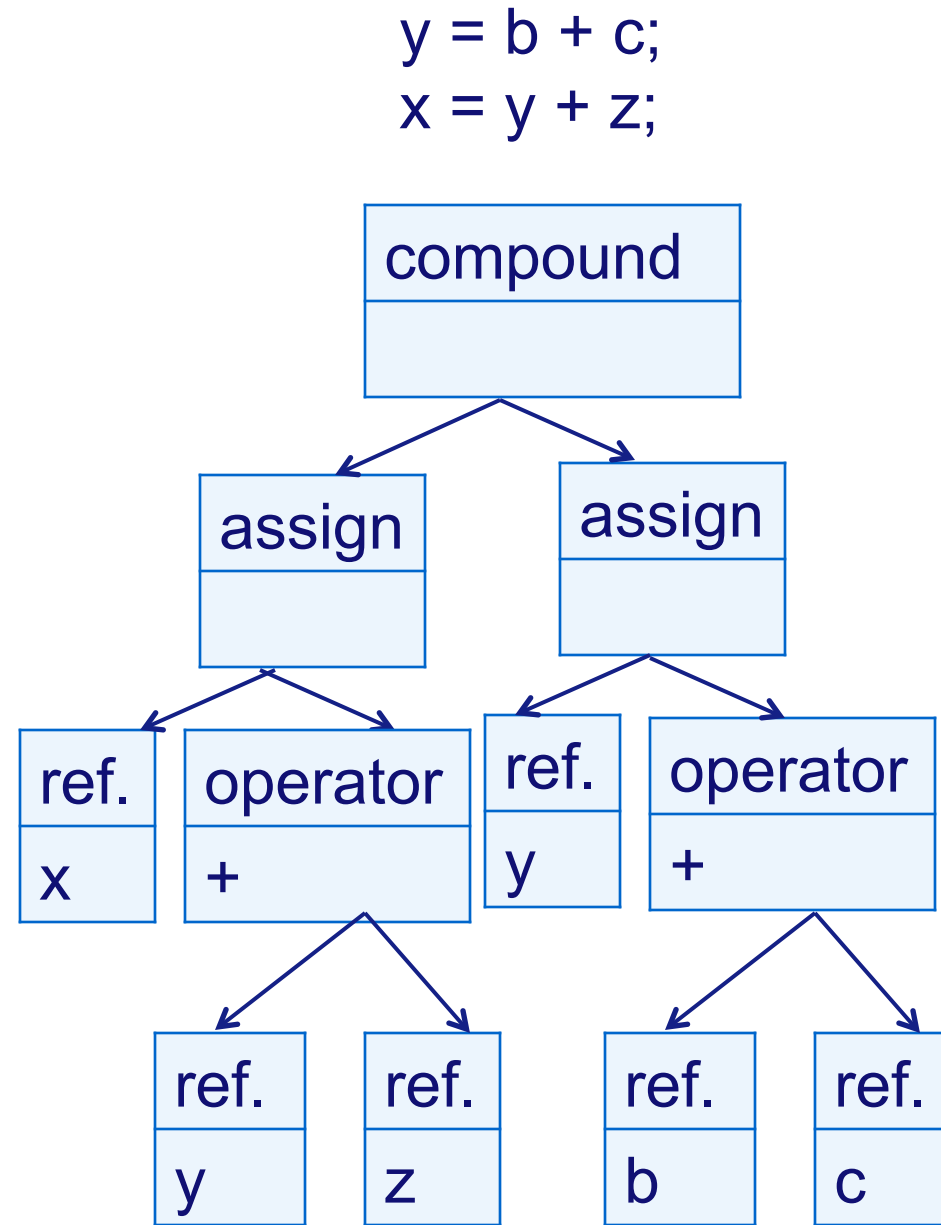
# [Krinke 2001] PDG based

- **Vertices:**

- entry points, in- and output parameters
- assignments, control statements, function calls
- variables, operators

- **Edges:**

- immediate dependencies
  - target has to be evaluated before the source



# [Krinke 2001] PDG based

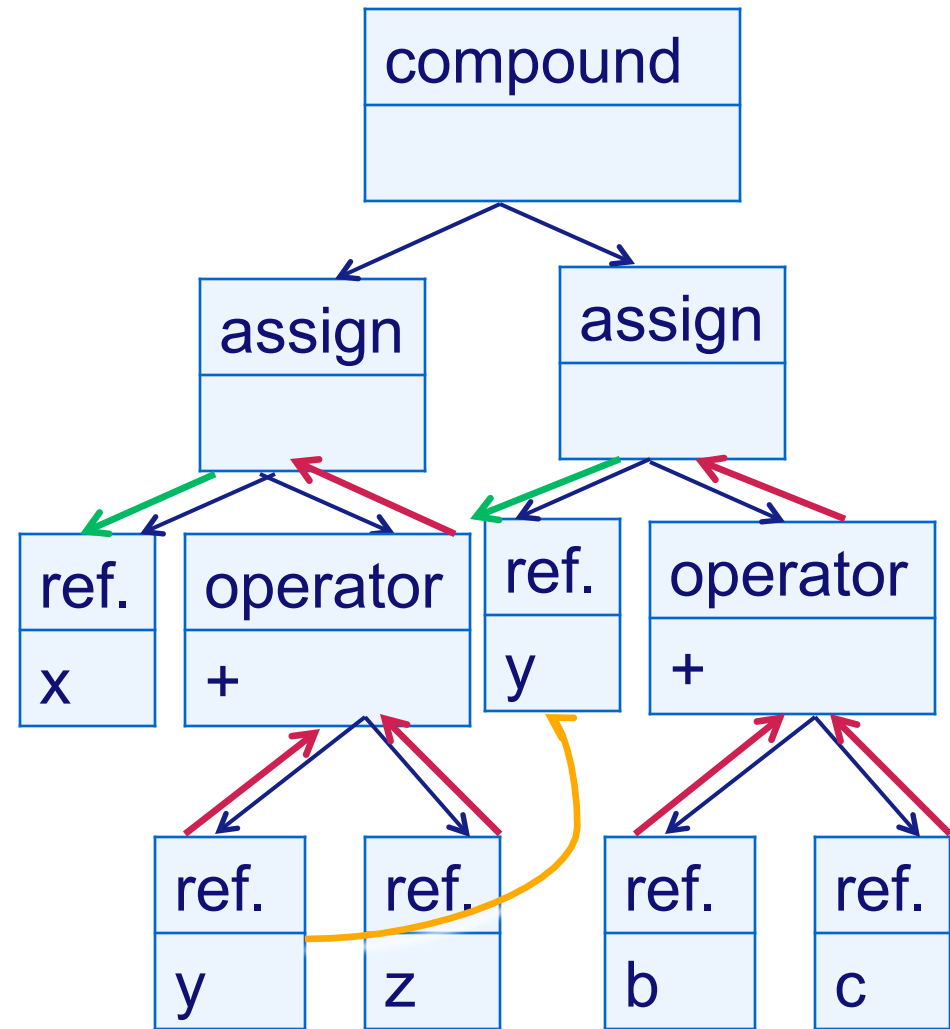
- **Vertices:**

- entry points, in- and output parameters
- assignments, control statements, function calls
- variables, operators

- **Edges:**

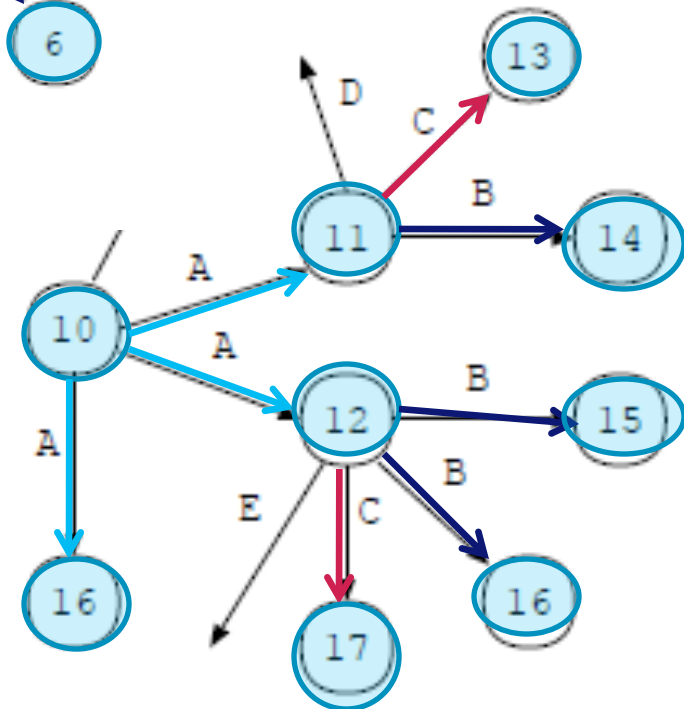
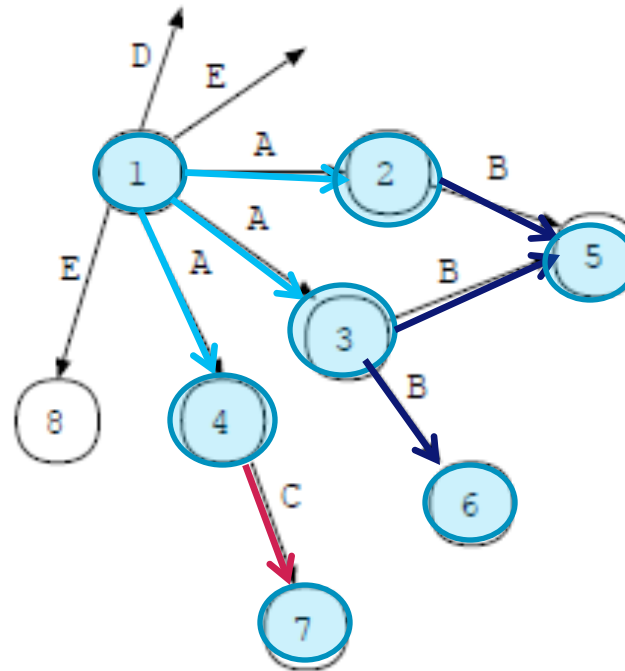
- immediate dependencies
- **value** dependencies
- **reference** dependencies
- **data** dependencies
- control dependencies
  - Not in this example

$y = b + c;$   
 $x = y + z;$



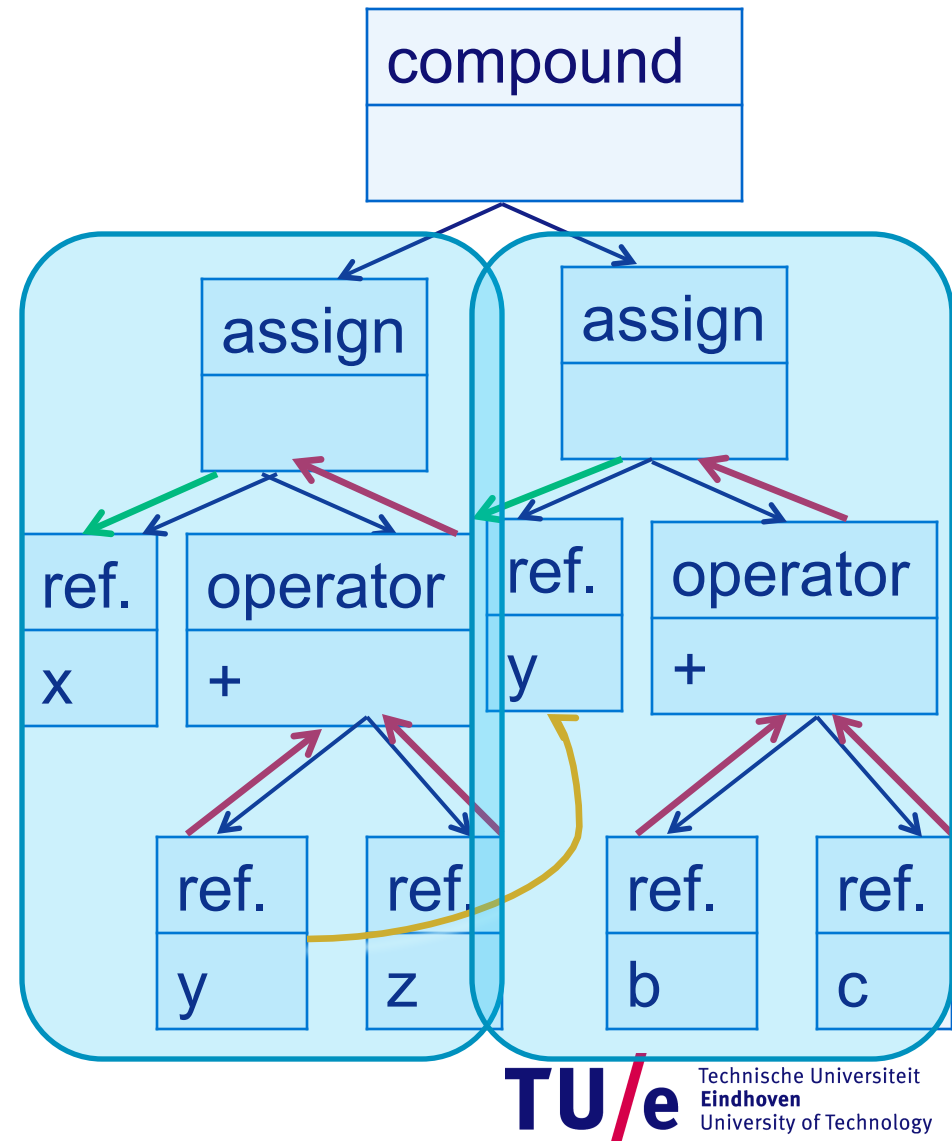
# Identification of similar subgraphs – Theory

- Start with 1 and 10
- Partition the incident edges based on their labels
  - Select classes present in both graphs
- Add the target vertices to the set of reached vertices
- Repeat the process
- “Maximal similar subgraphs”



# Identification of similar subgraphs – Practice

- Sorts of edges are **labels**
- We also need to compare **labels of vertices**
- We should stop after **k iterations**
  - Higher  $k \Rightarrow$  higher recall
  - Higher  $k \Rightarrow$  higher execution time
  - Experiment:  $k = 20$



# Choosing your tools: Precision / Recall

- Quality depends on scenario [Type 1, Type 2, Type 3]
- [Roy *et al.* 2009]: 6 is maximal grade, 0 – minimal

Tool	Technique	Category	S1	S2	S3
Duploc	Ducasse	Text	4	0	2.8
Marcus and Maletic			2.6	1.8	1.6
Dup	Baker	Token	4	2.8	0
CCFinder	Kamiya		5	3.8	0.8
CloneDr	Baxter	AST	6	4.3	3.8
cpdetector	Koschke		6	3.8	0
Mayrand		Metrics	3.3	4.8	3.4
Duplix	Krinke	Graph	5	4.8	4

**More tools: ConQAT, DECKARD, Dude, Simian**

# Which technique/tool is the best one?

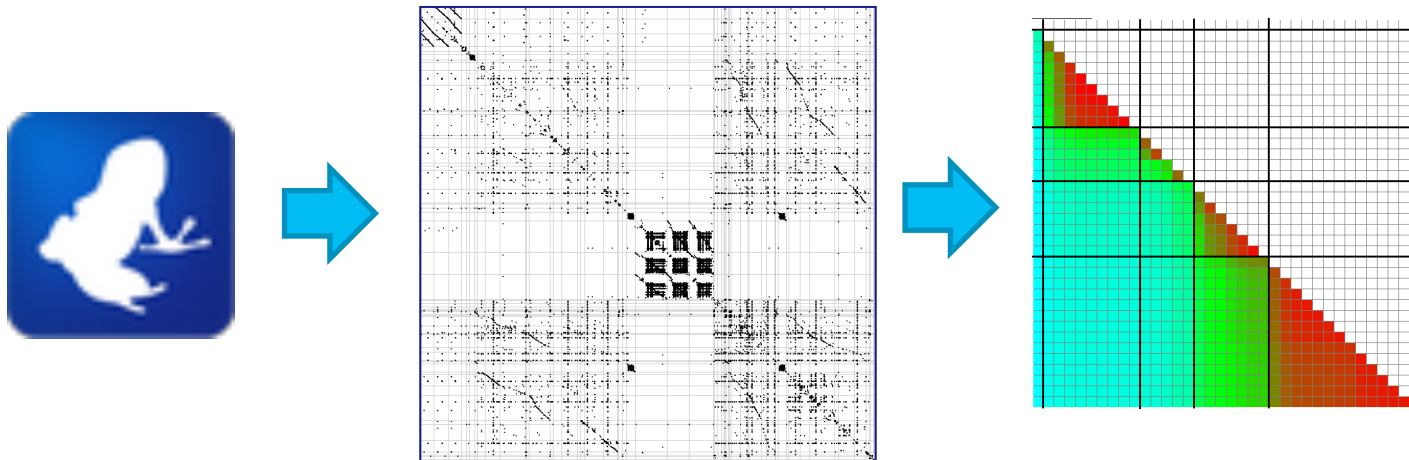
- **Quality**
  - Precision
  - Recall
- **Usage**
  - Availability
  - Dependence on a platform
  - Dependence on an external component (lexer, tokenizer, ...)
  - Input/output format
- **Programming language**
- **Clones**
  - Granularity
  - Types
  - Pairs vs. groups
- **Technique**
  - Normalization
  - Storage
  - Worst-case complexity
  - Pre-/postprocessing
- **Validation**
- *Extra: metrics*

# Clone detection techniques: Summary

- **Many different techniques**
  - Text, metrics, tokens, AST, program dependence graph, combinations
- **Techniques are often supported by tools**
- **Precision depends on what kind of clones we need:**
  - Type 1, Type 2, Type 3, Type 4
- **Extra conditions**
  - Programming language, presence of external tools, platforms, extra's (metrics), normalization, ...

# Assignment 3

- Individual
- Deadline: March 17
- NB: replication study!
  - Be as precise as possible





# Next week

- No classes!



22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering

