SOEN 6431 SOFTWARE MAINTENANCE AND Program Comprehension

DR. JUERGEN RILLING

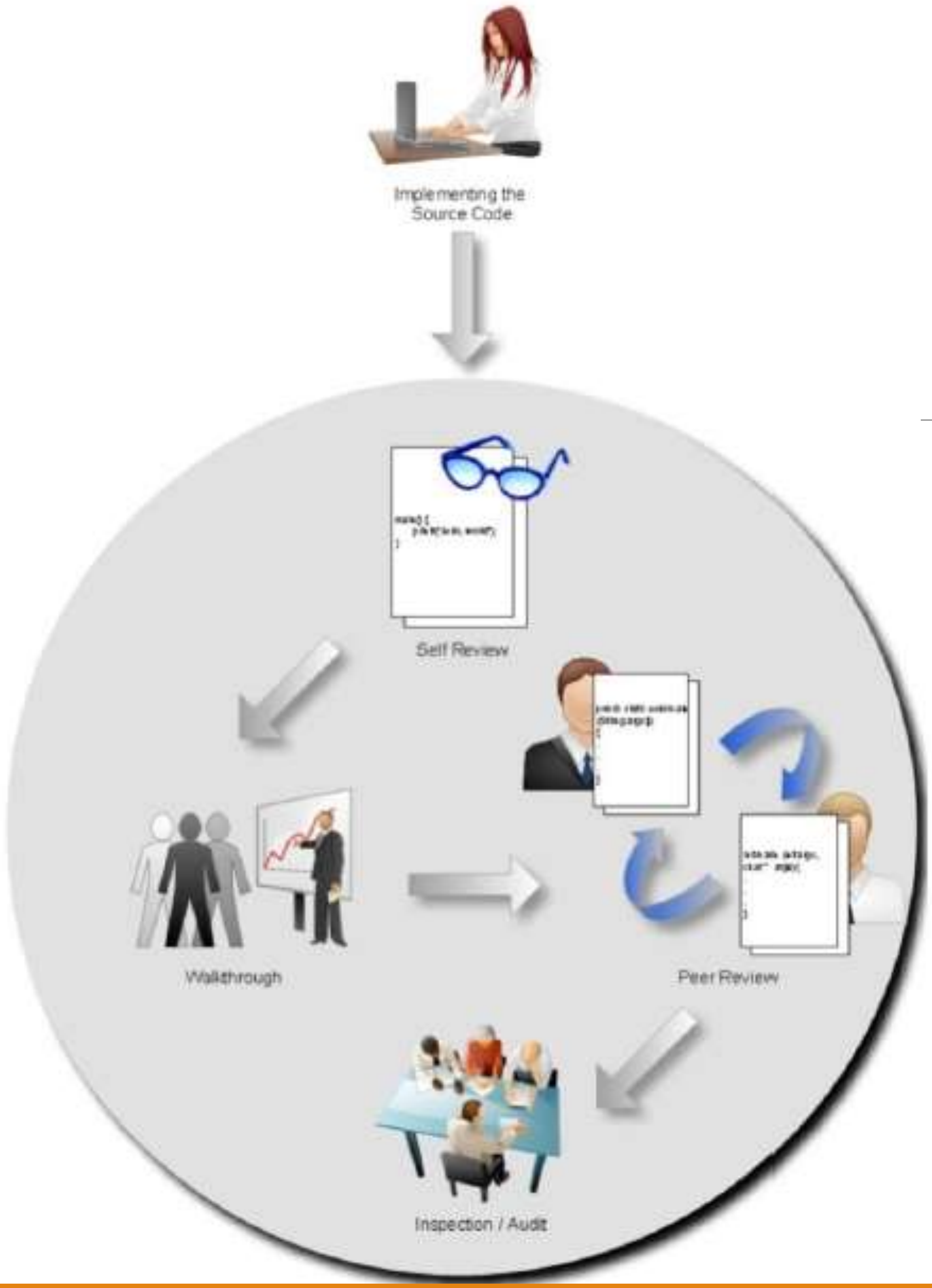
Week 4

Source Code Analysis

Overview

Comprehension - we looked so far at the general idea behind comprehension – including cognitive models/mental models

What we are looking now at are technique(s) that can be used to support program comprehension (code cognition).

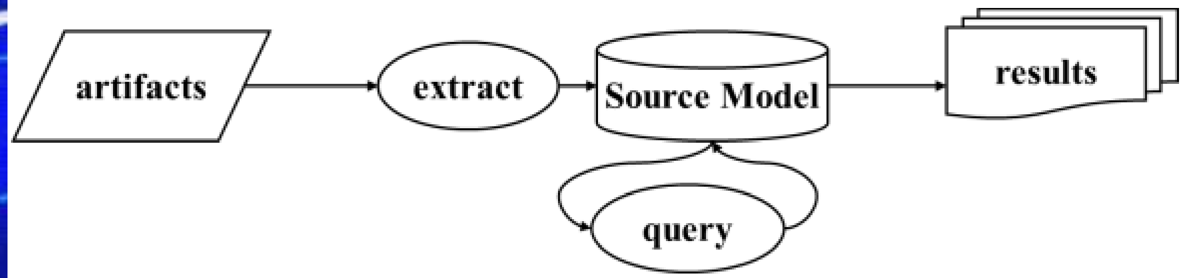


Static Code Analysis

MANUAL ANALYSIS

Automated Code Analysis

- ***Extract source code models from system artefacts***
- ***Query/manipulate to infer new knowledge***
- ***Present different views on results***



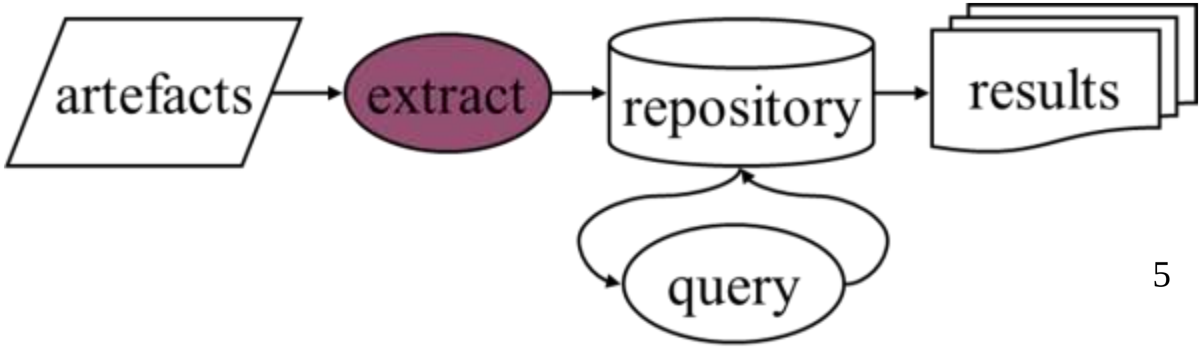
Source Model Extraction

Derive information from system artifacts

- variable usage, call graphs, file dependencies, database access, …

Challenges

- *Accurate* & *complete* results
- *Flexible*: easy to write and adapt
- *Robust*: deal with irregularities in input



Parsing of artifacts

- ***Syntactical analysis***
  - ***generate / hand-code / reuse parser***
- ***Lexical analysis***
  - ***tools like perl, grep, Awk or LSME, MultiLex***
  - ***generally easier to develop***

	accurate	*complete*	*flexible*	*robust*
syntactical	+	+	–	–
lexical	–	–	+	+

Scanning/Lexical analysis

Break program down into its smallest meaningful symbols (tokens, atoms)

Tools for this include lex, flex

Tokens include e.g.:

- "Reserved words": do if float while
- Special characters: ({ , + - = ! /
- Names & numbers: myValue 3.07e02

Start symbol table with new symbols found

Parsing

Construct a parse tree from symbols

A pattern-matching problem

If no pattern matches, it's a syntax error

yacc, bison are tools for this (generate c code that parses specified language)

- Language **grammar** defined by set of rules that identify legal (meaningful) combinations of symbols
- Each application of a rule results in a node in the parse tree
- Parser applies these rules repeatedly to the program until leaves of parse tree are "atoms"

Parse tree

Output of parsing

Top-down description of program syntax

- Root node is entire program

Constructed by repeated application of rules in Context Free Grammar (CFG)

Leaves are tokens that were identified during lexical analysis

Example: Parsing rules for Pascal

These are like the following:

program PROGRAM *identifier* (*identifier,more_identifiers*) ;

block

block *variables* BEGIN *statement* *more_statements* END

statement *do_statement* | *if_statement* | *assignment* | …

if_statement IF *logical_expression* THEN *statement* ELSE …

Pascal code example

```
program gcd (input, output)
var i, j : integer
begin
  read (i , j)
  while i <> j do
            if i>j then i := i – j;
            else j := j – i ;
  writeln (i);
end .
```

Semantic analysis

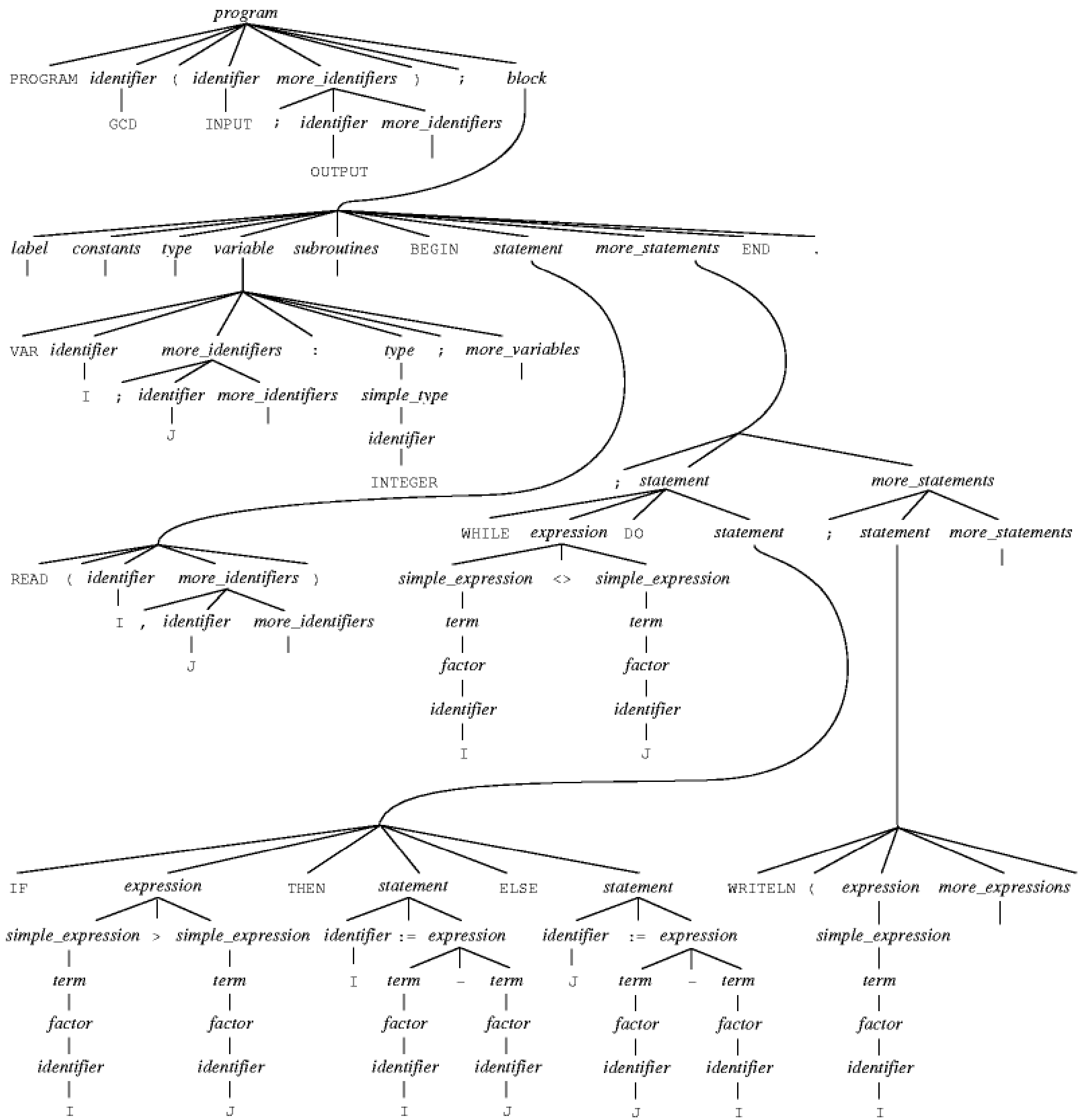
Discovery of meaning in a program using the symbol table

- Do static semantics check
- Simplify the structure of the parse tree (from parse tree to abstract syntax tree (AST))

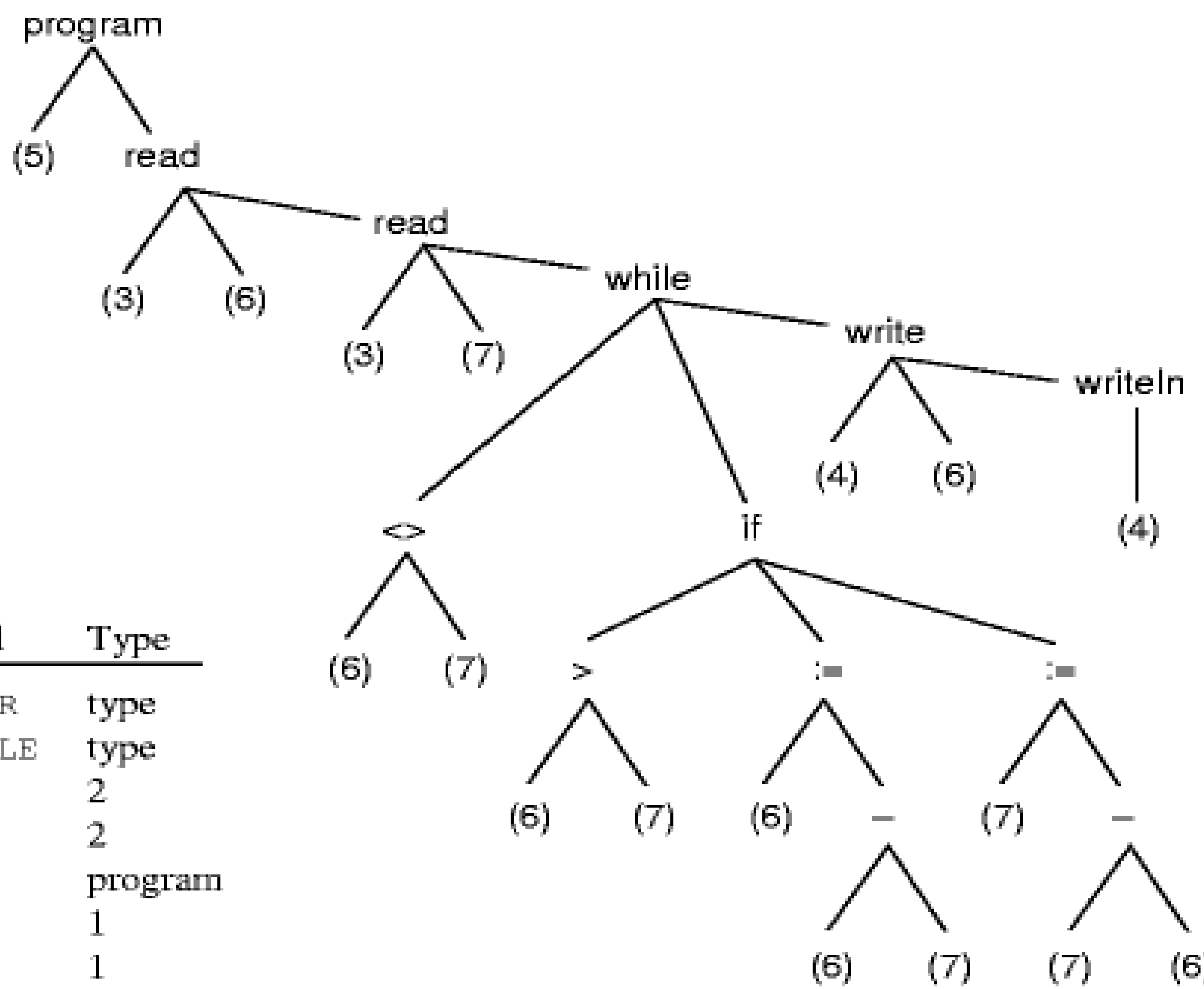
Static semantics check

- Making sure identifiers are declared before use
- Type checking for assignments and operators
- Checking types and number of parameters to subroutines
- Making sure functions contain return statements
- Making sure there are no repeats among switch statement labels

Example: parse tree



Example: AST

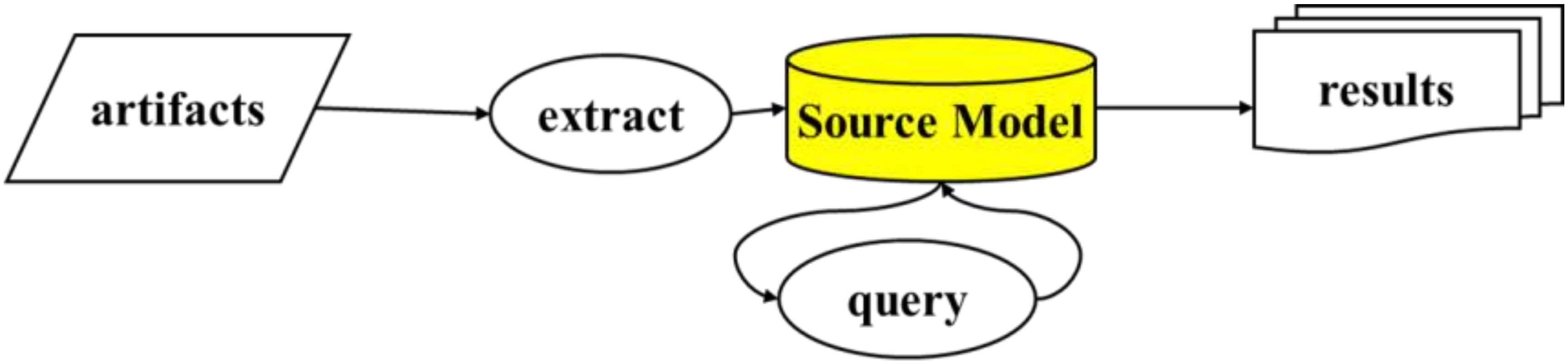


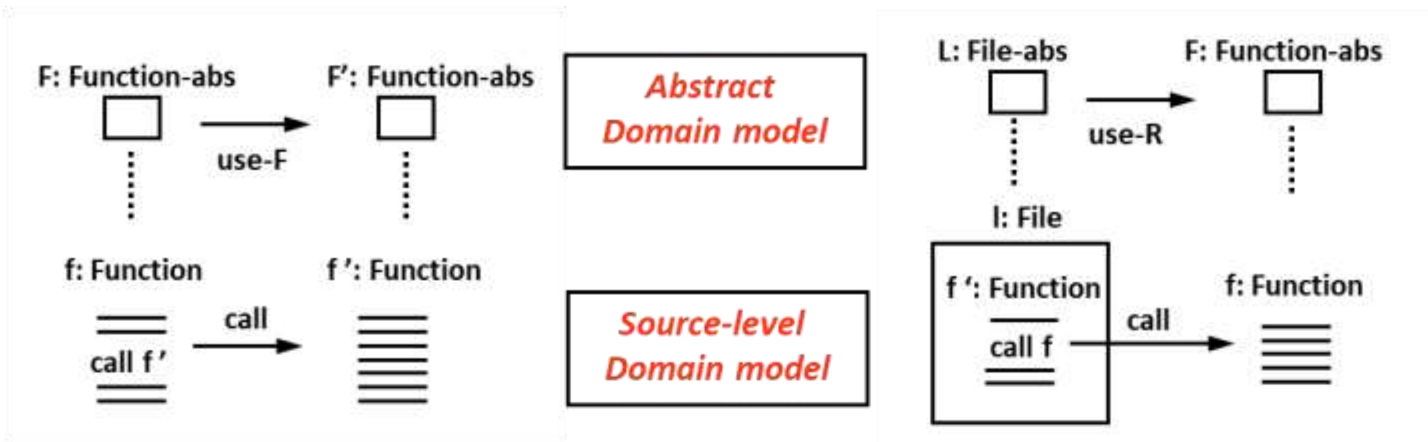
Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

i+++++i;

Parsing challenges

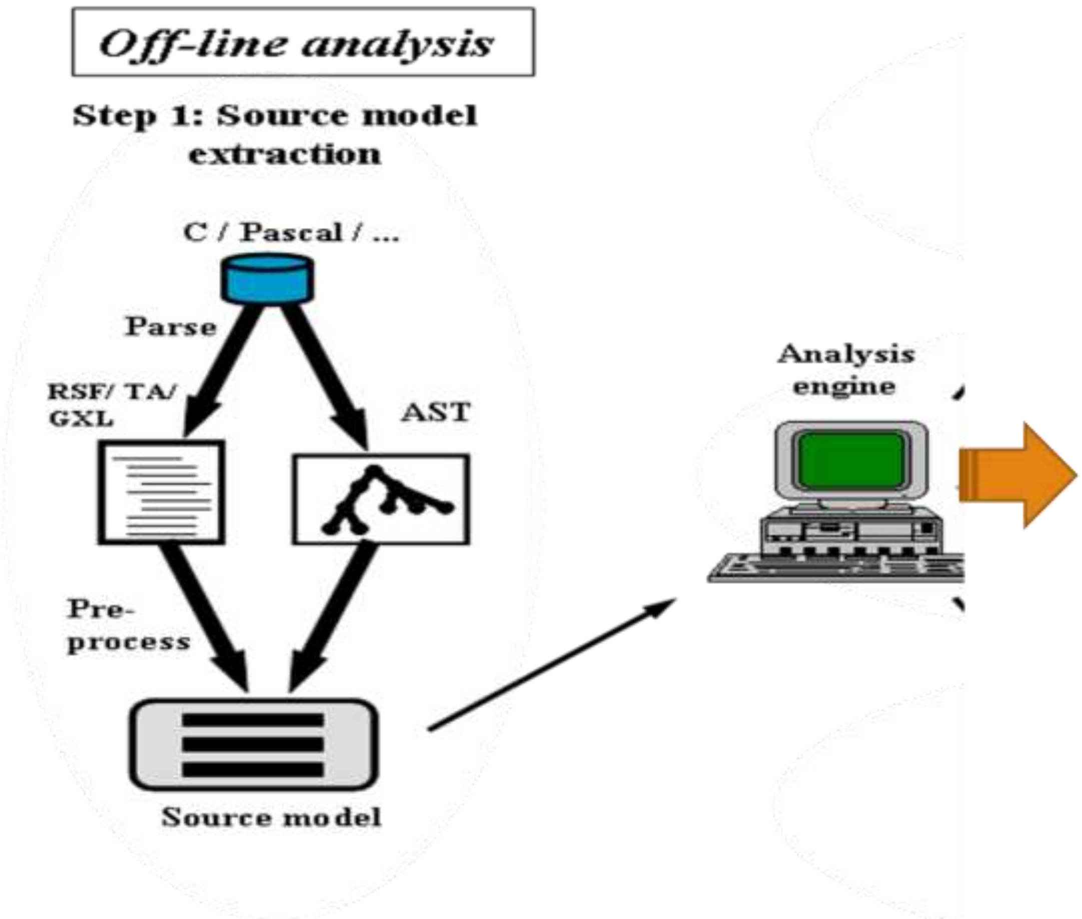
- *Syntax Errors*
- *Language Dialects*
- *Local Idioms*
- *Missing Parts*
- *Embedded Languages*
- *Preprocessing*
- *Additional problem:* grammar availability
  - *process languages without grammar (e.g., undisclosed proprietary languages)*
  - *development of full grammar is expensive*



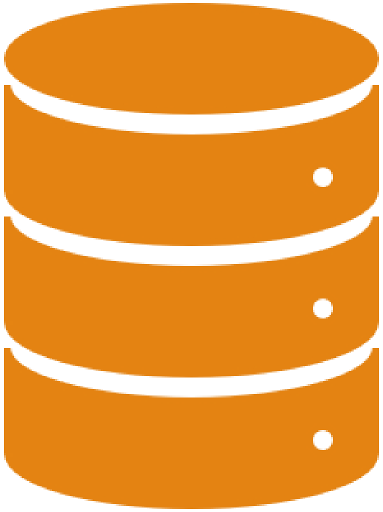


Graph formalism is widely used. Example of graph formalism:

- ❑ Abstraction of the source-level domain model
- ❑ Entity-types: a subset of entity-types in source-code
- ❑ Relation-type: an aggregation of one or more relation-types in source-code



Program Comprehension



Automated parsing and creating source models are cool, but…

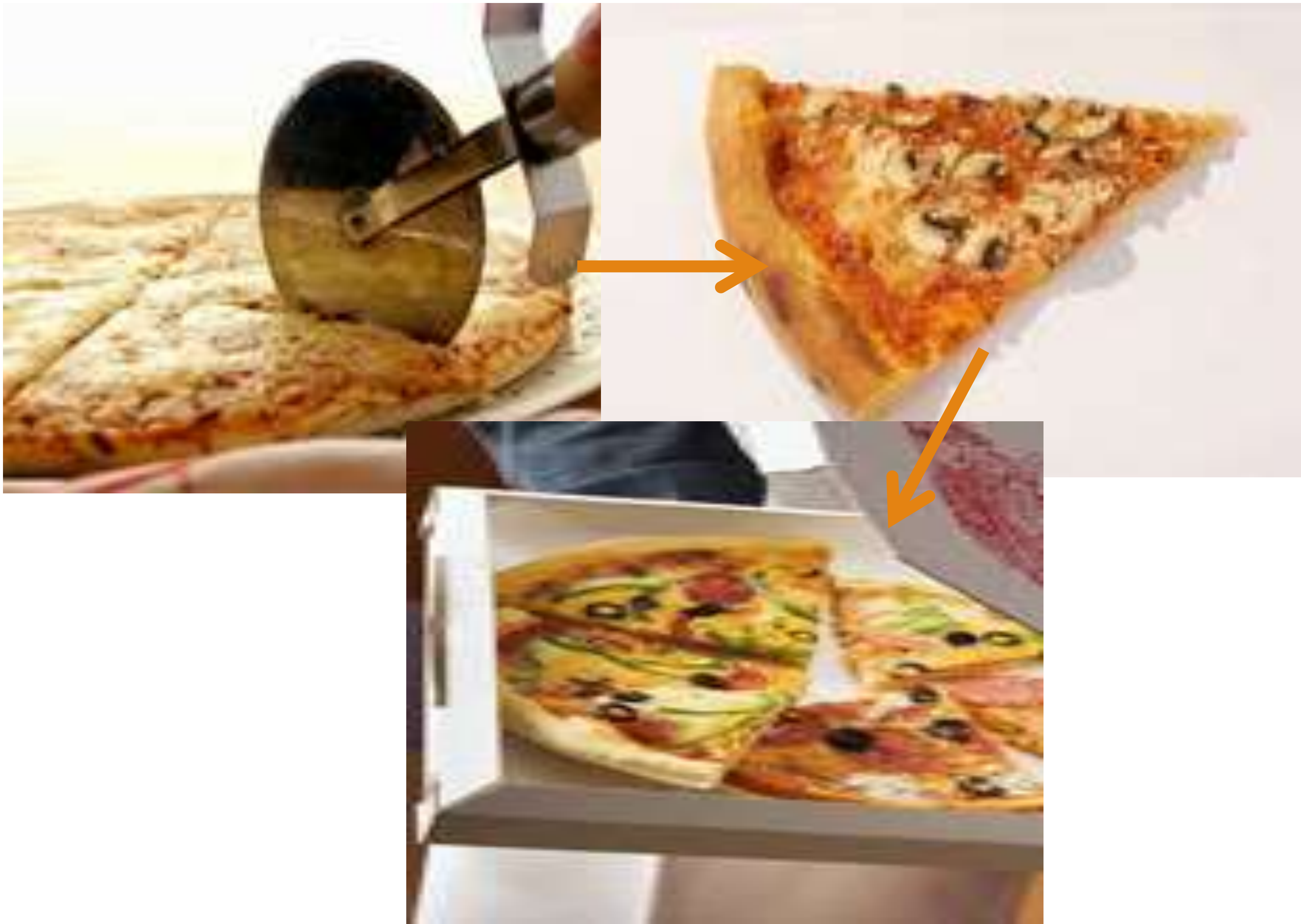
what and **how** do we analyze now extracted source models

Is there maybe something we can learn from other domains?

How can we address these problems?

Solution:

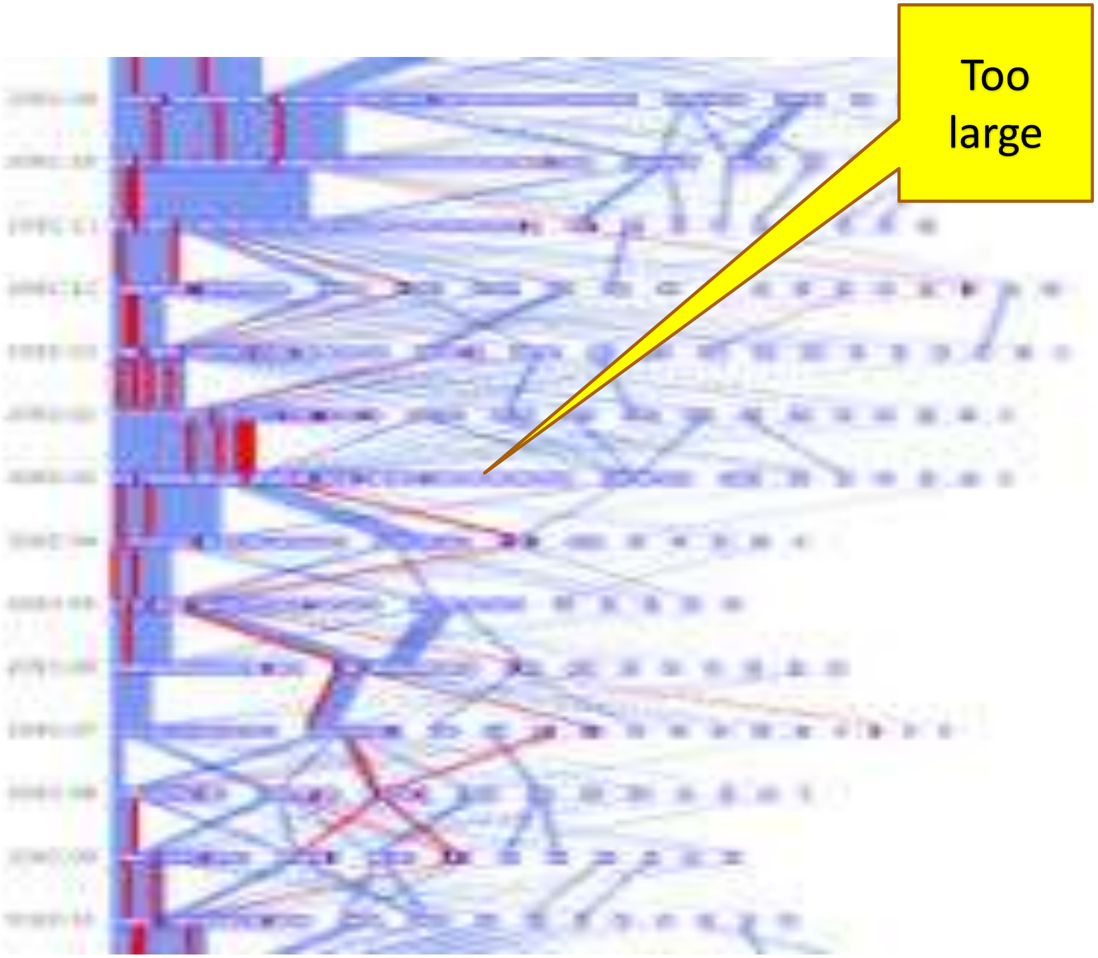
Let’s slice
(a pizza) !!



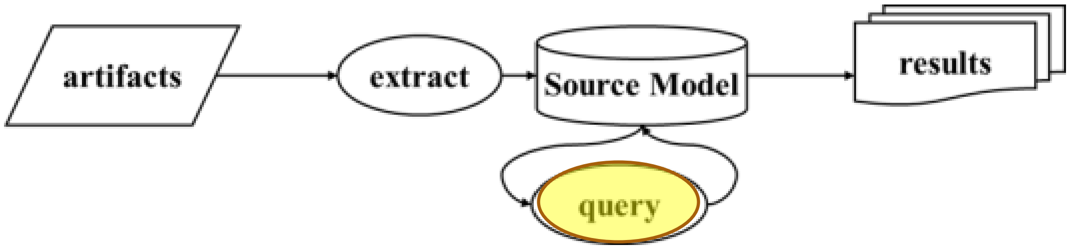
Why we slice a pizza?

....easier to eat

What about a program ?



Solution:



Slicing

Slicing

Why we slice a pizza?

....easier to eat

Why we slice a program?

```
read (n)
i := n
product := 0
while (i ≥ 0) do
  product := product * i
  i := i - 1
write (product)
```

....easier to understand, debug, etc.

Why Program Slicing?

Program Debugging: that's how slicing was discovered!

Testing: reduce cost of regression testing after modifications (only run those tests that needed)

Parallelization: Split program so that it can be executed on several processors, machines, etc.

Integration: merging two programs A and B that both resulted from modifications to BASE

Reverse Engineering: comprehending the design by abstracting out of the source code the design decisions

Software Maintenance: changing source code without unwanted side effects

Software Quality Assurance: validate interactions between safety-critical components

General Idea of Slicing

Given:

(1) A program

(2) A variable *v* at some point *P* in the program

Goal:

Finding the part of the program that is responsible for the computation of variable *v* at point *P.*

A simple example

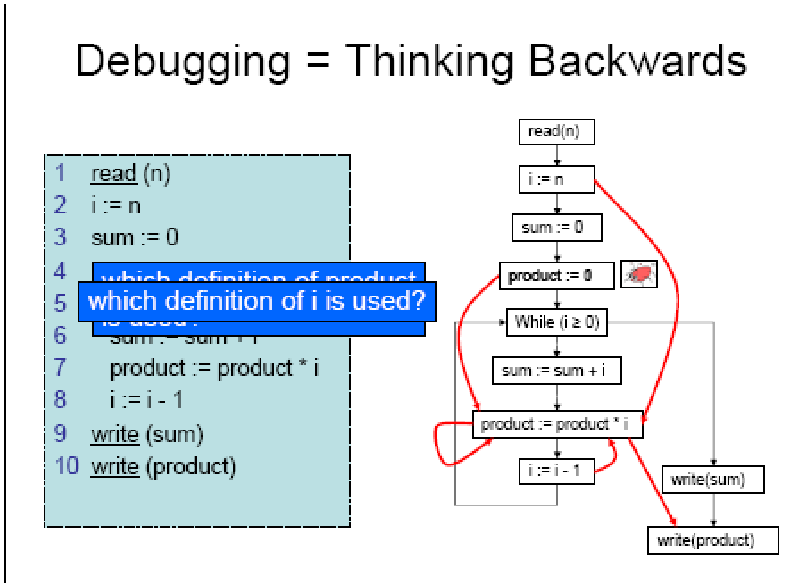
```
1. b = 1;
2. c = 2;
3. d = 3;
4. a = d;
5. d = b + d;
6. b = b + 1;
7. a = b + c
8. print a;
```

- Why is `a` equal to 4 in line 8?
- Which lines do you need to explain the value of `a` in line 8?

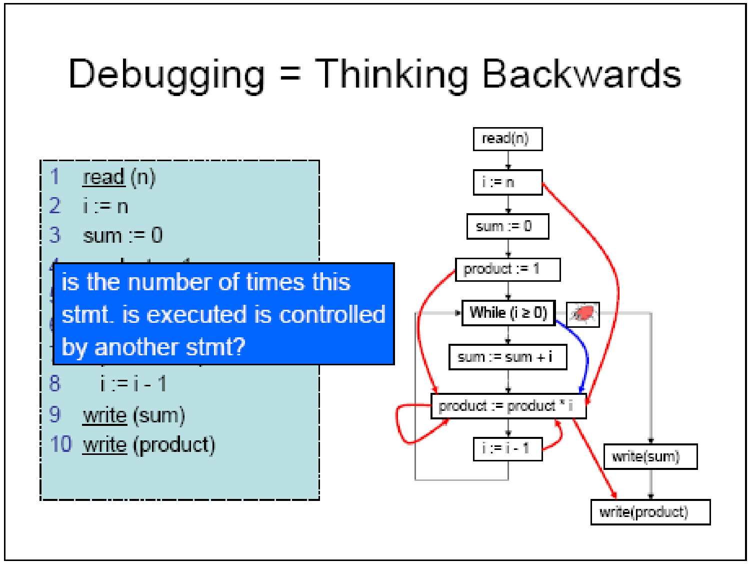
Program Slicing

```
1. b = 1;
2. c = 2;
3. d = 3;
4. a = d;
5. d = b + d;
6. b = b + 1;
7. a = b + c
8. print a;
```

- You only need lines 1, 2, 6, 7 (and 8)



Debugging = Thinking Backwards
1 read (n)
2 i := n
3 sum := 0
4
5 which definition of i is used?
6
7 product := product * i
8 i := i - 1
9 write (sum)
10 write (product)
read(n)
i := n
sum := 0
product := 0
While (i ≥ 0)
sum := sum + i
product := product * i
i := i - 1
write(sum)
write(product)



Debugging = Thinking Backwards
1 read (n)
2 i := n
3 sum := 0
is the number of times this stmt. is executed is controlled by another stmt?
8 i := i - 1
9 write (sum)
10 write (product)
read(n)
i := n
sum := 0
product := 1
While (i ≥ 0)
sum := sum + i
product := product * i
i := i - 1
write(sum)
write(product)

Basic Idea

Types of slices

- Backward static slice
- Executable slice
- Forward static slice
- Dynamic slice
- Execution slice
- Generic algorithm for static slice

Levels of slices

- Intraprocedural
- Interprocedural

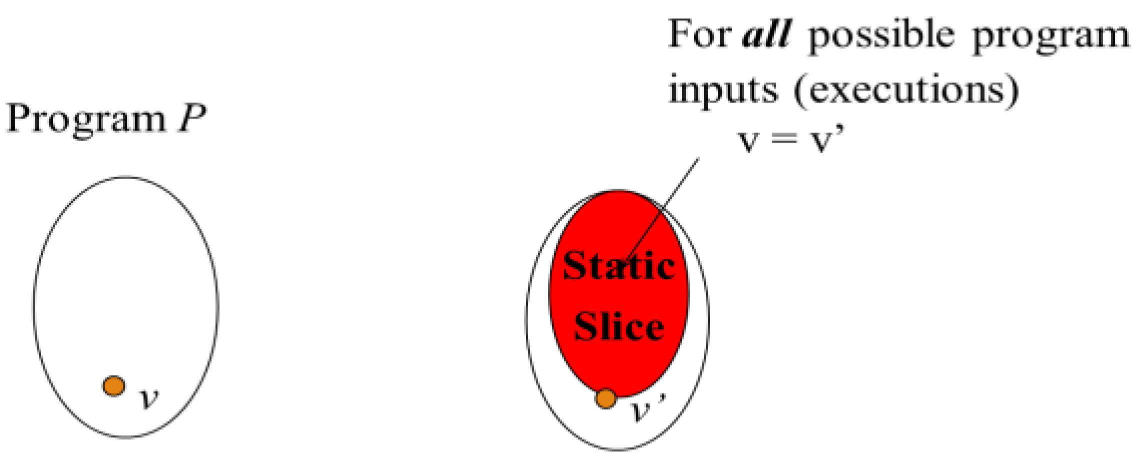
A slice is executable if the statements in the slice form a syntactically correct program that can be executed.

If the slice is computed correctly (safely), the result of running the program that is the executable slice produces the same result for variables in **V** at **p** for all inputs.

Types of Slicing (Executable)

Static Backward Program Slicing

Static Backward Program Slicing was original introduced by Weiser in 1982. A static program slice consists of these parts of a program *P* that potentially could affect the value of a variable *v* at a point of interest.



Slicing Properties:

Static Slicing

- Statically available information only
- **No** assumptions made on input
- Computed slice can never be accurate (minimal slice)
- Problem is **undecidable** – reduction to the halting problem
- Current static methods can only compute approximations
- Result may not be usefull

Data Dependencies

def and use

- An assignment `a = b + c` *defines* `a` and *uses* `b` and `c`
- `b = b + 1` uses and defines b
- `if(a + b < 5)` uses `a`, `b`, does not define anything

- Let's compute the relevant variables in our program

```
main( )
 {
2   sum = 0;
3   i = 1;
4   sum = sum + 1;
5    ++ i;
6  cout<< sum;
7  cout<< i;
}
```

```
main( )
 {
2   sum = 0;
3   i = 1;
4   sum = sum + 1;
5    ++ i;
6  cout<< sum;
7  cout<< i;
}
```

An Example Program & its slice w.r.t. <7, i>

Branching

```
1.  b = 1
2.  c = 2
3.  d = 3
4.  a = d
5.  if(a)then
6.    d = b + d
7.    c = b + d
8.  else
9.    b = b + 1
10.   d = b + 1
11. fi
12. a = b + c
13. print a
```

What part of the program is relevant to the value of a in line 13?

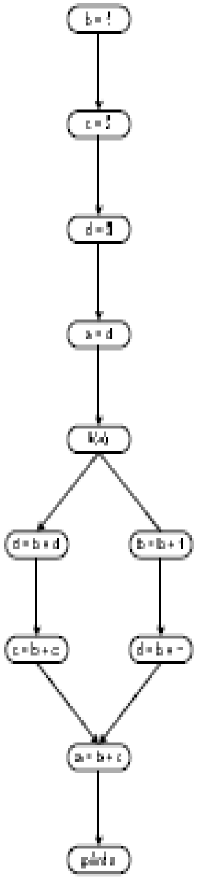
Branching

```
1.  b = 1
2.  c = 2
3.  d = 3
4.  a = d
5.  if(a)then
6.    d = b + d
7.    c = b + d
8.  else
9.    b = b + 1
10.   d = b + 1
11. fi
12. a = b + c
13. print a
```

What part of the program is relevant to the value of a in line 13?

All lines except line 10.

Flow Graph



Creating a PDG

```
1     input (n,a);
2     max := a[1];
3     min  := a[1];
4     i := 2;
5     s:= 0;
6     while i ≤ n do
      begin
7         if max < a[i] then
          begin
8             max := a[i];
9             s := max;
          end;
10        if min > a[i] then
          begin
11            min := a[i];
12            s := min;
          end;
13        output (s);
14        i := i +2;
      end;
15    output (max)   ;
16    output (min);
```

Data Dependence:

Represents a data flow (definition-use chain).

=> Data dependence between 2 and 7 but not between 2 and 8.

Control Dependence:

The execution of a node depends on the outcome of a predicate node.

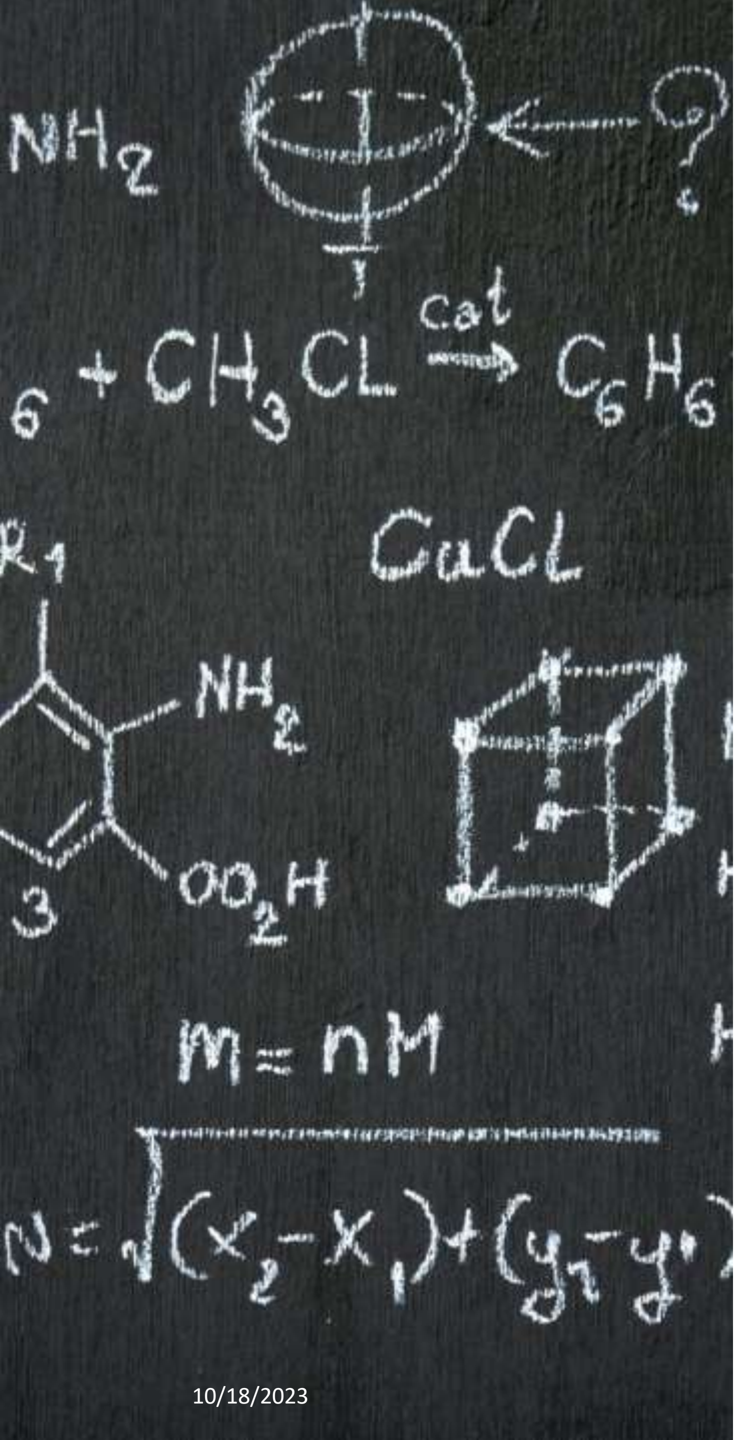
=> Control dependence between node 6 and 8, but not between 6 and 15.

Loops

- Loops may require updating relevant more than once.
- Example:

```
while(a > 0){
  e = d
  d = c
  c = b
}
print e;
```

- Before loop, `a`, `b`, `c`, `d`, and `e` are relevant
  - `a` decides the loop, `e` is relevant if loop not taken, `d` if taken once, `c` if taken twice, `b` if taken trice.

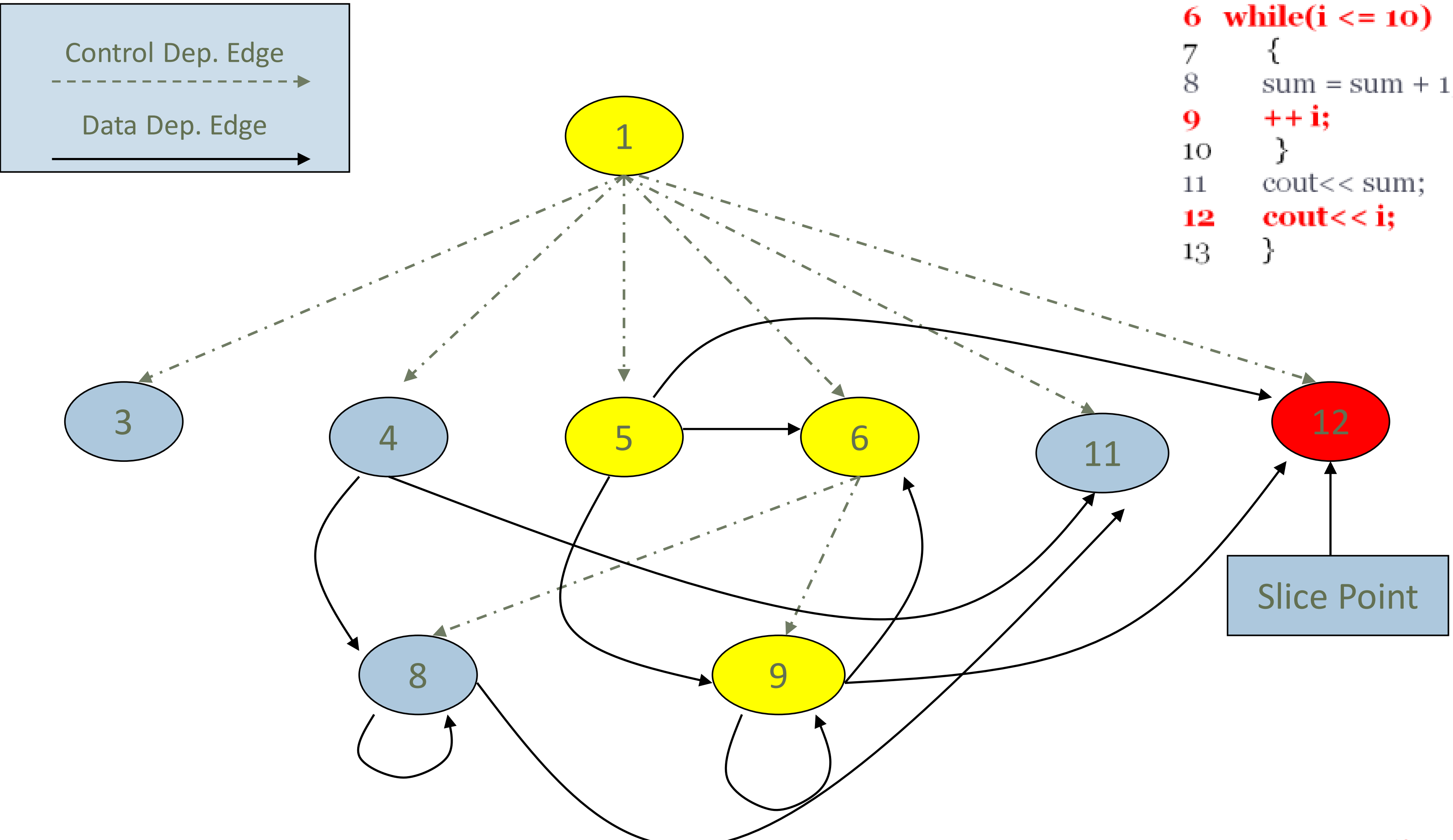


Another example for a loop

```
main( )
{
  int i, sum;
  sum = 0;
  i = 1;
  while(i <= 10)
     {
       sum = sum + 1;
       ++ i;
     }
     cout<< sum;
     cout<< i;
     }
```

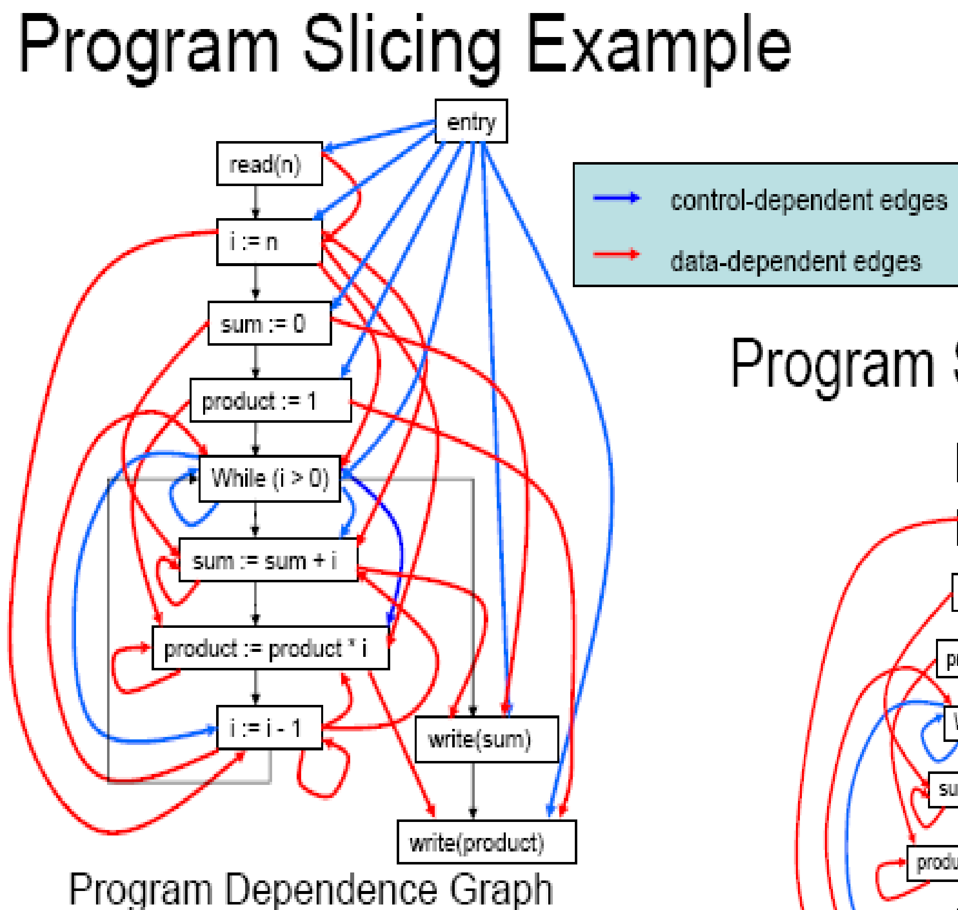
PDG of the Example Program

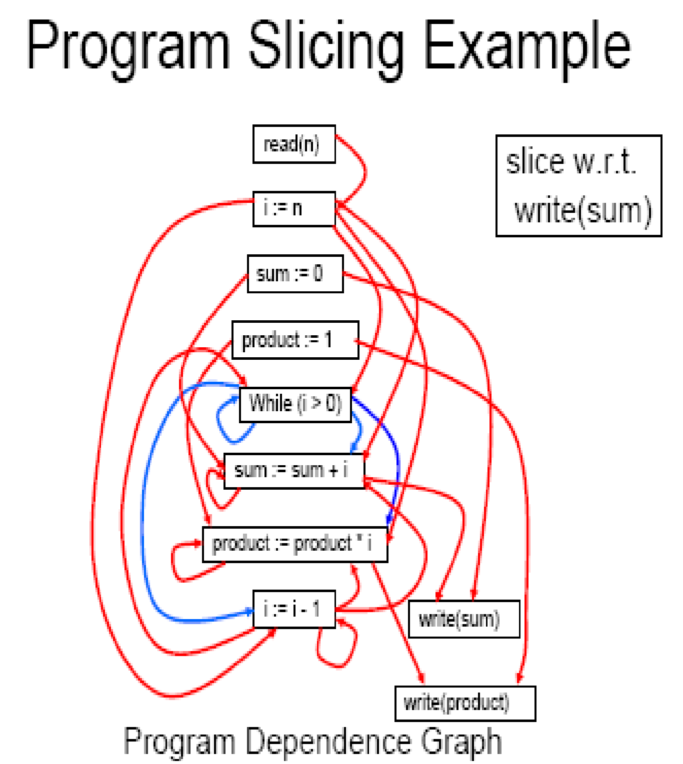
```
1 main( )
2 {
3   int i, sum;
4   sum = 0;
5   i = 1;
6   while(i <= 10)
7       {
8       sum = sum + 1;
9       ++ i;
10      }
11      cout<< sum;
12      cout<< i;
13      }
```



Static Backward slicing example

```
3.      sum :=0;
4.      product:= 1;
3       while (i>0)
   {
4           sum:= sum+i
5           product:= pro
6           i:=i -1;
  }
7       write(sum);
8       write (product);
```

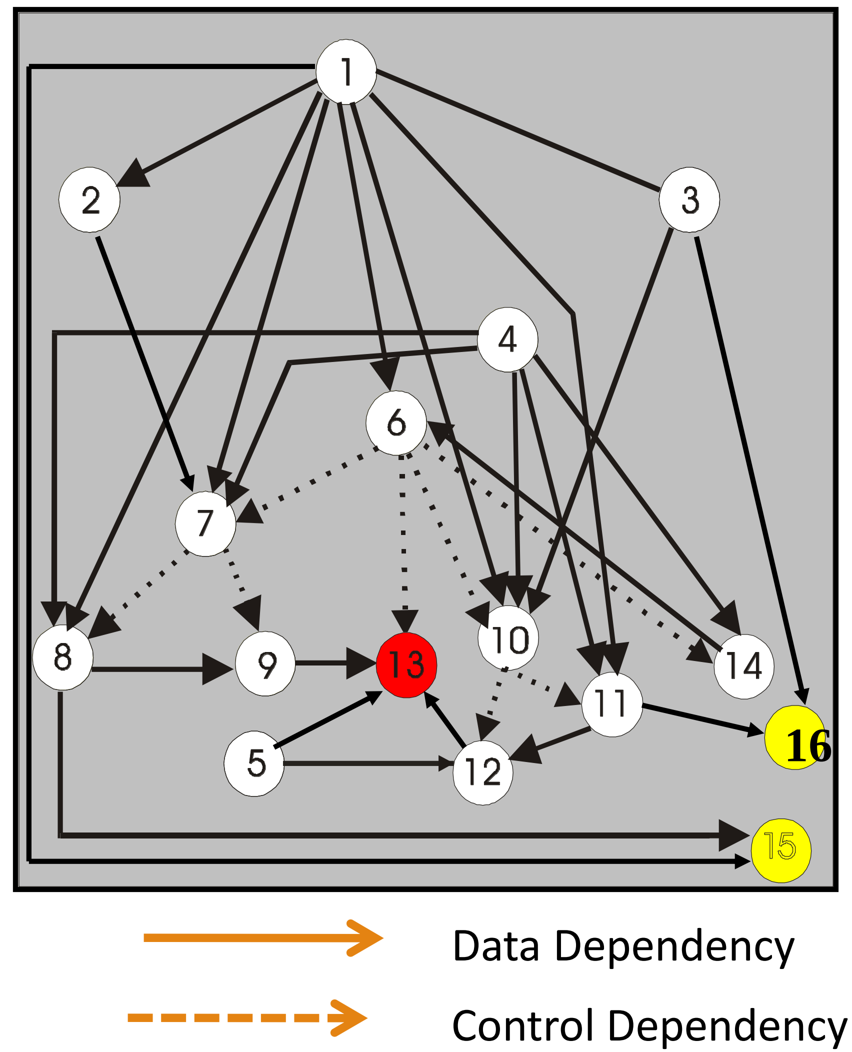




Program Dependence Graph (PDG)

A Program dependence graph is formed by combining *data and control dependencies* between nodes.

```
1     input (n,a);
2     max := a[1];
3     min  := a[1];
4      i := 2;
5     s:= 0;
6     while i ≤ n do
      begin
7           if max < a[i] then
            begin
8                 max := a[i];
9                 s := max;
            end;
10          if min > a[i] then
            begin
11                min := a[i];
12                s := min;
            end;
13          output (s);
14          i := i +2;
      end;
15    output (max);
16    output (min);
```



Any problems within this PDG?

"Controversial" statements:

1. Static forward slicing will always provide a meaningful reduction

2. Can you think about any challenges for static slicing

Forward Slice (static)

Note: It is not necessarily value preserving - meaning the value for the variable in the Slice might not be the same as in the original program.

A forward slice of a program with respect to a program point **p** and set of program variables **V** consists of all statements and predicates in the program that may be affected the value of variables in **V** at **p**

The program point **p** and the variables **V** together form the slicing criterion, usually written <**p, V**>

Slicing – Forward Static

```
1.  read (n)
2.  i := 1
3.  sum := 0
4.  product := 1
5.  while i <= n do
6.      sum := sum + i
7.      product := product * i
8.      i := i + 1
9.  write (sum)
10. write (product)
```

Criterion <3, sum>

Objective: what parts of a program **<u>are affected</u>** by a modification to the the variable specified in the slicing criterion.

Slicing – Forward Static

```
1.  read (n)
2.  i := 1
3.  sum := 0
4.  product := 1
5.  while i <= n do
6.      sum := sum + i
7.      product := product * i
8.      i := i + 1
9.  write (sum)
10. write (product)
```

Criterion <3, sum>

Slicing – Forward Static

```
1.  read (n)
2.  i := 1
3.  sum := 0
4.  product := 1
5.  while i <= n do
6.      sum := sum + i
7.      product := product * i
8.      i := i + 1
9.  write (sum)
10. write (product)
```

Criterion <1, n>

Slicing – Forward Static

```
read (n)
i := 1
sum := 0
product := 1
while i <= n do
    sum := sum + i
    product := product * i
    i := i + 1
write (sum)
write (product)
```

Controversial statement:

Forward slicing provides more meaningful insights compared to backward slicing?

Justify your answer