软件分析与验证前沿

苏亭

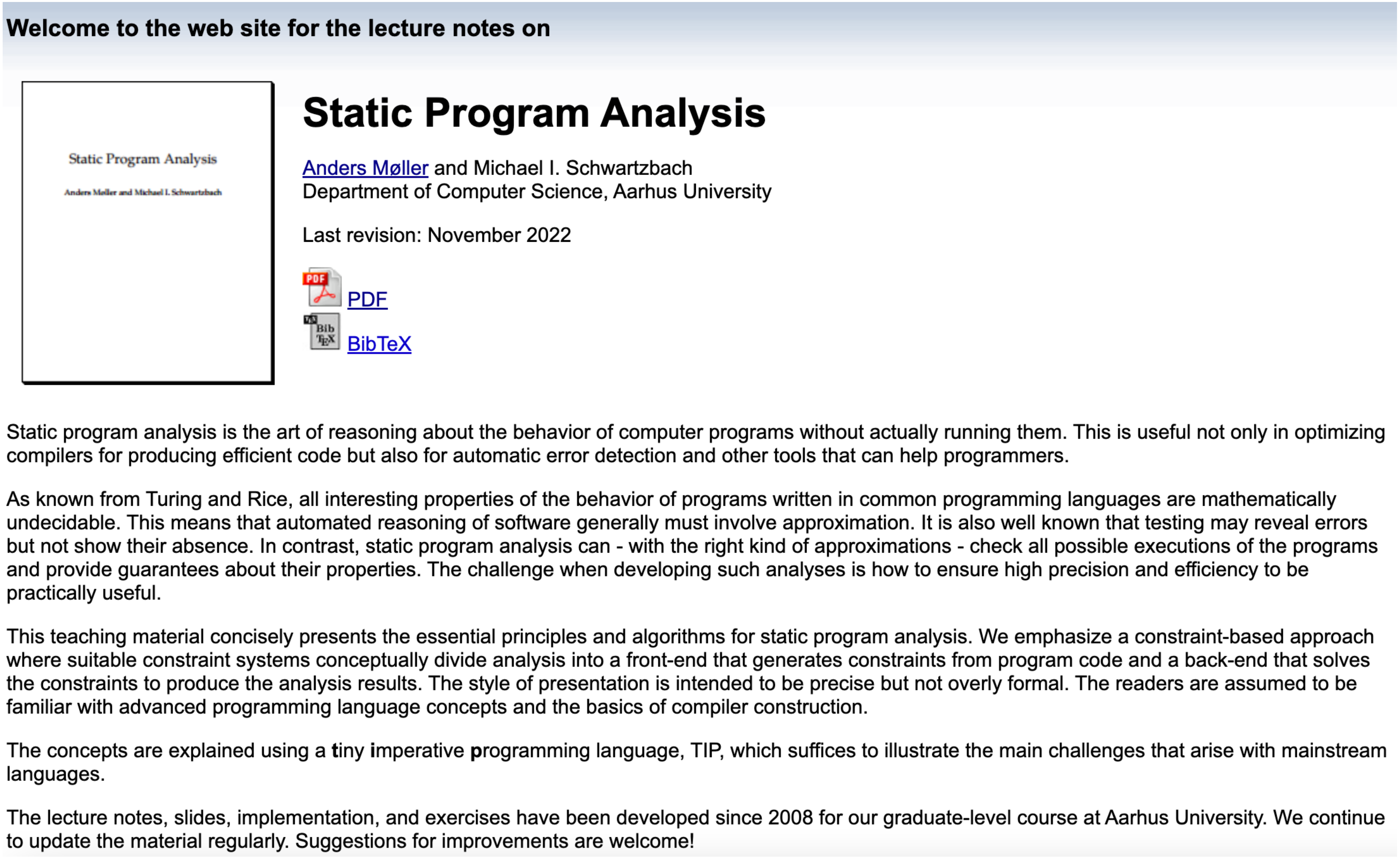
软件科学与技术系

Big Picture

- Introduction to static analysis
- Many ways of formulating and implementing analyses
- One popular way of formulating a static analysis: Data flow analysis

Big Picture

- Introduction to static analysis
- Many ways of formulating and implementing analyses
- One popular way of formulating a static analysis: Data flow analysis



Welcome to the web site for the lecture notes on

Static Program Analysis

Anders Møller and Michael I. Schwartzbach

Static Program Analysis

Anders Møller and Michael I. Schwartzbach
Department of Computer Science, Aarhus University

Last revision: November 2022

PDF

BibTeX

Static program analysis is the art of reasoning about the behavior of computer programs without actually running them. This is useful not only in optimizing compilers for producing efficient code but also for automatic error detection and other tools that can help programmers.

As known from Turing and Rice, all interesting properties of the behavior of programs written in common programming languages are mathematically undecidable. This means that automated reasoning of software generally must involve approximation. It is also well known that testing may reveal errors but not show their absence. In contrast, static program analysis can - with the right kind of approximations - check all possible executions of the programs and provide guarantees about their properties. The challenge when developing such analyses is how to ensure high precision and efficiency to be practically useful.

This teaching material concisely presents the essential principles and algorithms for static program analysis. We emphasize a constraint-based approach where suitable constraint systems conceptually divide analysis into a front-end that generates constraints from program code and a back-end that solves the constraints to produce the analysis results. The style of presentation is intended to be precise but not overly formal. The readers are assumed to be familiar with advanced programming language concepts and the basics of compiler construction.

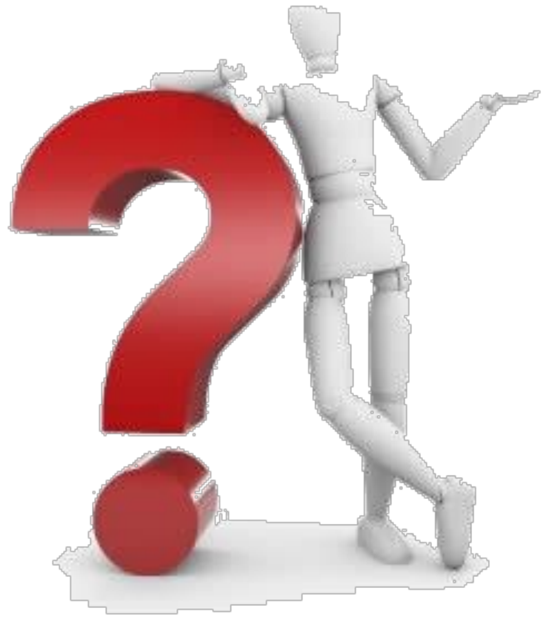
The concepts are explained using a **t**iny **i**mperative **p**rogramming language, TIP, which suffices to illustrate the main challenges that arise with mainstream languages.

The lecture notes, slides, implementation, and exercises have been developed since 2008 for our graduate-level course at Aarhus University. We continue to update the material regularly. Suggestions for improvements are welcome!

课程资料：http://cs.au.dk/~amoeller/spa/

Questions about programs

- Does the program terminate on all inputs?
- How large can the heap become during execution?
- Can sensitive information leak to non-trusted users?
- Can non-trusted users affect sensitive information?
- Are buffer-overruns possible?
- Data races?
- SQL injections?
- Cross-Site Scripting (XSS)?
- …



Program points

```
foo(p,x) {
    var f,q;
    if (*p==0) { f=1; }
    else {
        q = alloc 10;
        *q = (*p)-1;
        f=(*p)*(x(q,x));
    }
    return f;
}
```

any point in the program
= any value of the PC

Invariants:

A property holds at a program point if it holds in any such state for any execution with any input

Questions about program points

- Will the value of x be read in the future?
- Can the pointer p be null?
- Which variables can p point to?
- Is the variable x initialized before it is read?
- What is a lower and upper bound on the value of the integer variable x?
- At which program points could x be assigned its current value?
- Do p and q point to disjoint structures in the heap?
- Can this assert statement fail?

Why are the answers interesting?

- Increase efficiency
  - resource usage
  - compiler optimizations
- Ensure correctness
  - verify behavior
  - catch bugs early
- Support program understanding
- Enable refactorings

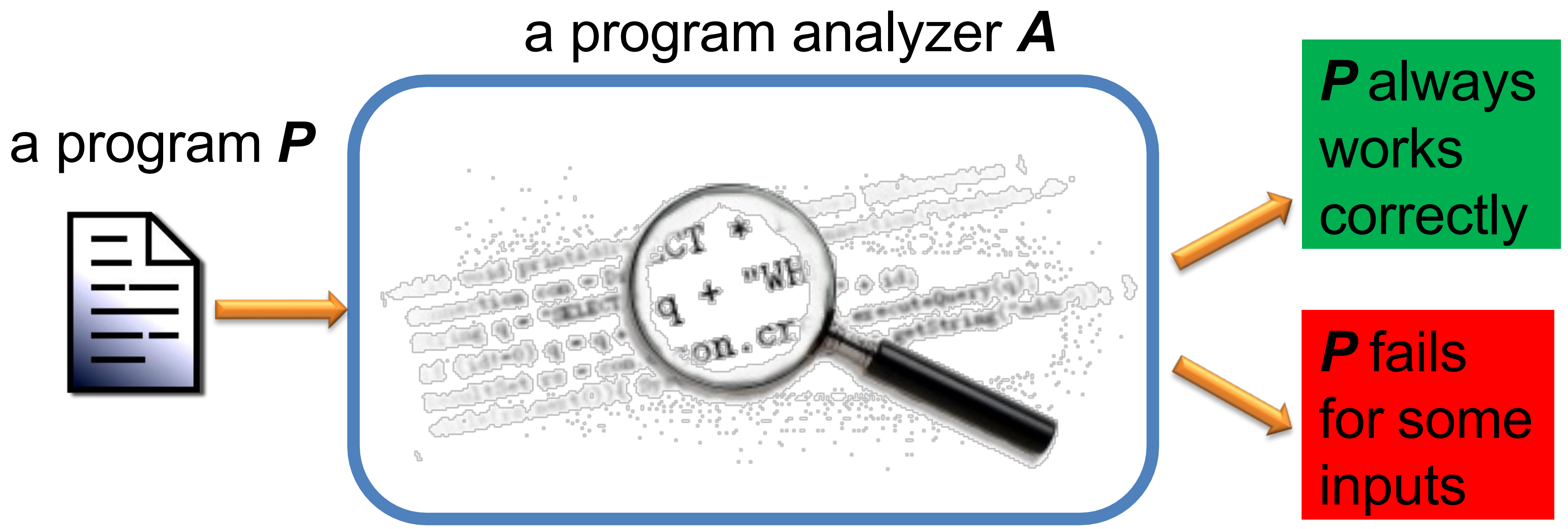
Testing?

“*Program testing can be used to show the presence of bugs, but never to show their absence.*”

[Dijkstra, 1972]

Nevertheless, testing often takes 50% of the development cost

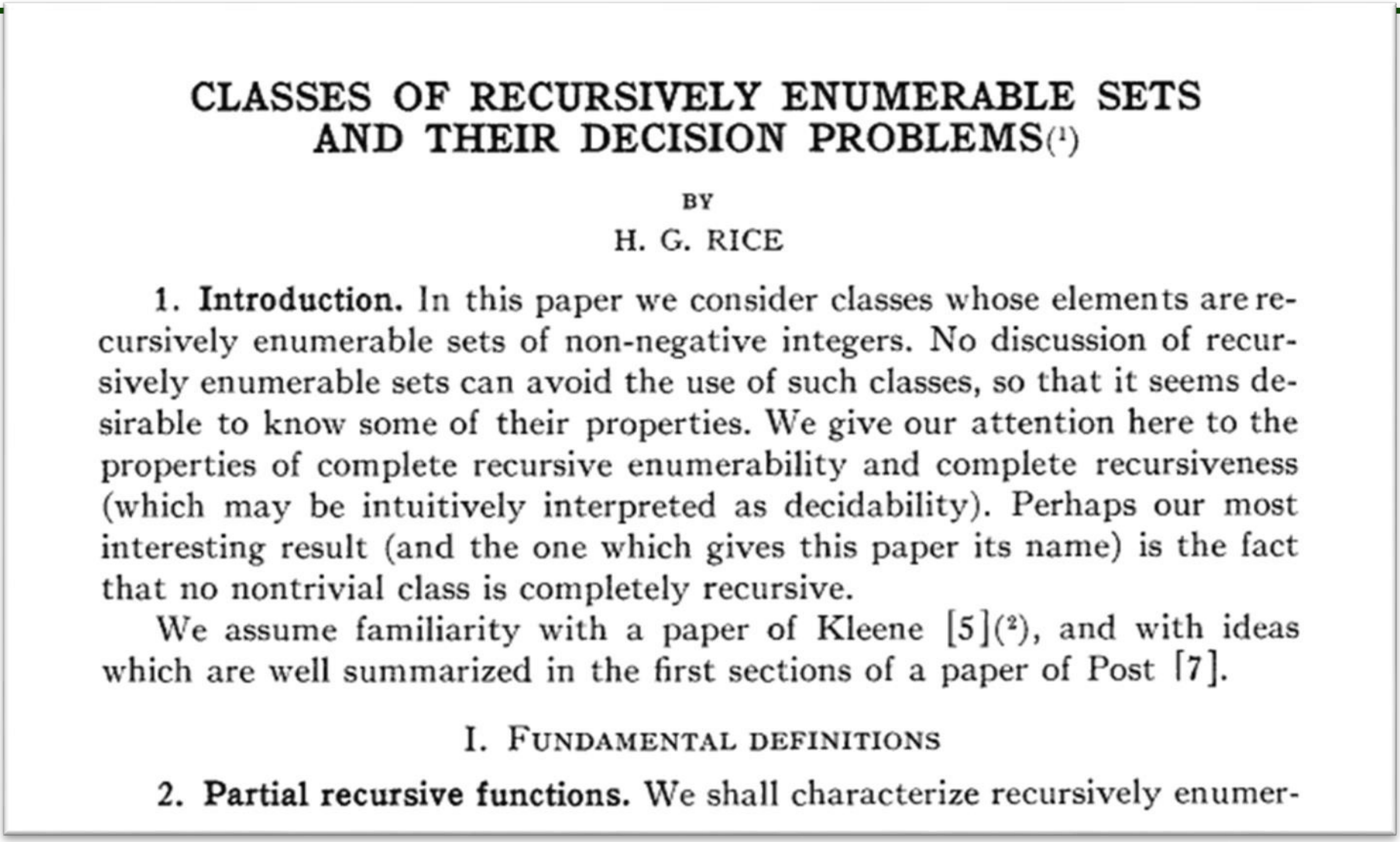
Programs that reason about programs


a program analyzer ***A***
a program ***P***
P always works correctly
P fails for some inputs

Requirements to the perfect program analyzer

- SOUNDNESS (don’t miss any errors)
- COMPLETENESS (don’t raise false alarms)
- TERMINATION (always give an answer)

Rice's theorem, 1953



CLASSES OF RECURSIVELY ENUMERABLE SETS AND THEIR DECISION PROBLEMS[1]

BY

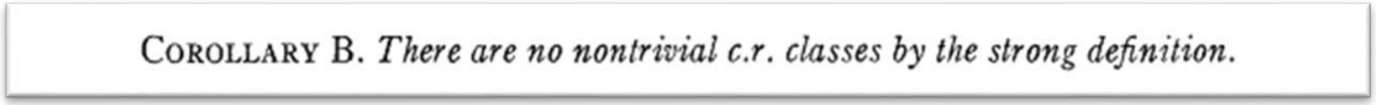
H. G. RICE

1. **Introduction.** In this paper we consider classes whose elements are recursively enumerable sets of non-negative integers. No discussion of recursively enumerable sets can avoid the use of such classes, so that it seems desirable to know some of their properties. We give our attention here to the properties of complete recursive enumerability and complete recursiveness (which may be intuitively interpreted as decidability). Perhaps our most interesting result (and the one which gives this paper its name) is the fact that no nontrivial class is completely recursive.

We assume familiarity with a paper of Kleene [5][2], and with ideas which are well summarized in the first sections of a paper of Post [7].

I. Fundamental definitions

2. **Partial recursive functions.** We shall characterize recursively enumer-



Corollary B. *There are no nontrivial c.r. classes by the strong definition.*

Rice’s theorem

Any non-trivial property of the behavior of programs in a Turing-complete language is undecidable!

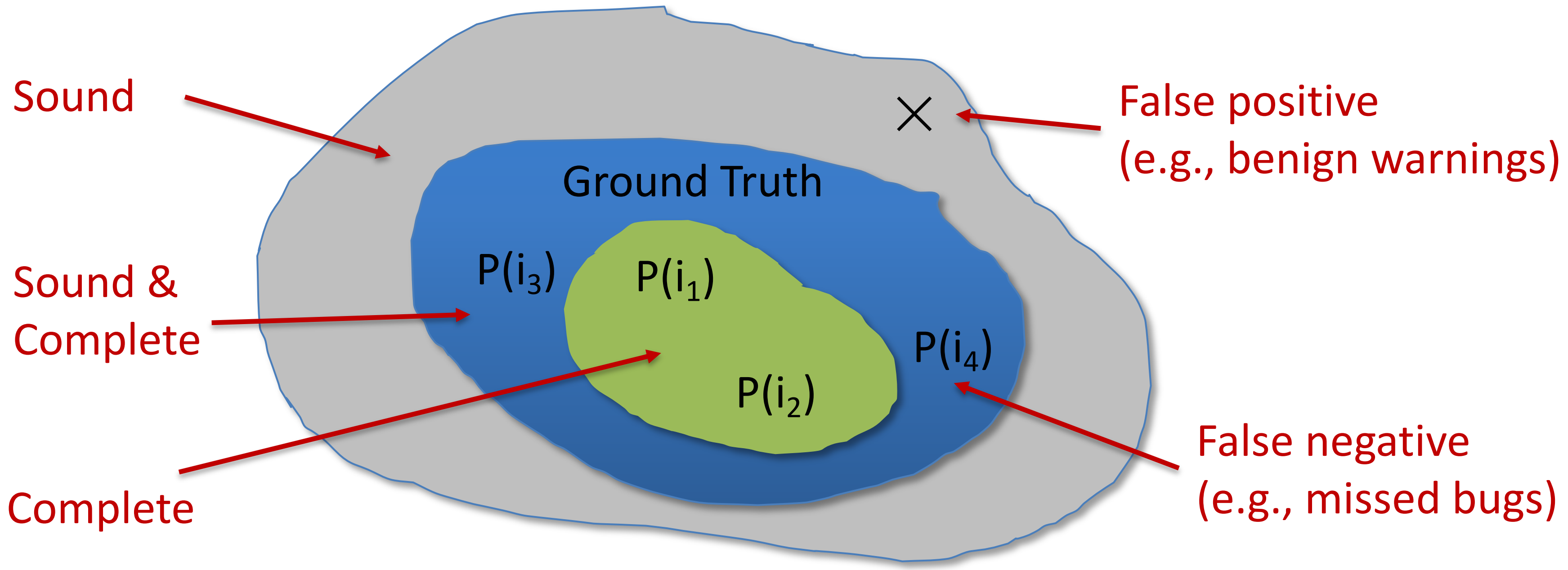
图灵完备语言（Turing-Complete Language）: https://en.wikipedia.org/wiki/Turing_completeness

Approximation

- *Approximate* answers may be decidable!

Soundness & Completeness

Program P, Input i, Behavior P(i)



🟦	All possible behaviors (what we want, ideally)
🟩	Under-approximation (e.g., testing, dynamic analysis)
⬜	Over-approximation (most static analysis)

Approximation

- *Approximate* answers may be decidable!

Approximation

- *Approximate* answers may be decidable!
- The approximation must be *conservative*:
  - i.e. only err on "the safe side"
  - which direction depends on the *client application*
- We'll focus on decision problems
- More subtle approximations if not only "*yes*"/"*no*"
  - e.g. memory usage, pointer targets

Example approximations

- Decide if a given function is ever called at runtime:
  - if "*no*", remove the function from the code
  - if "*yes*", don't do anything
  - the "*no*" answer *must* always be correct if given

- Decide if a cast (A)x will always succeed:
  - if "*yes*", don't generate a runtime check
  - if "*no*", generate code for the cast
  - the "*yes*" answer *must* always be correct if given

Beyond “yes”/“no” problems

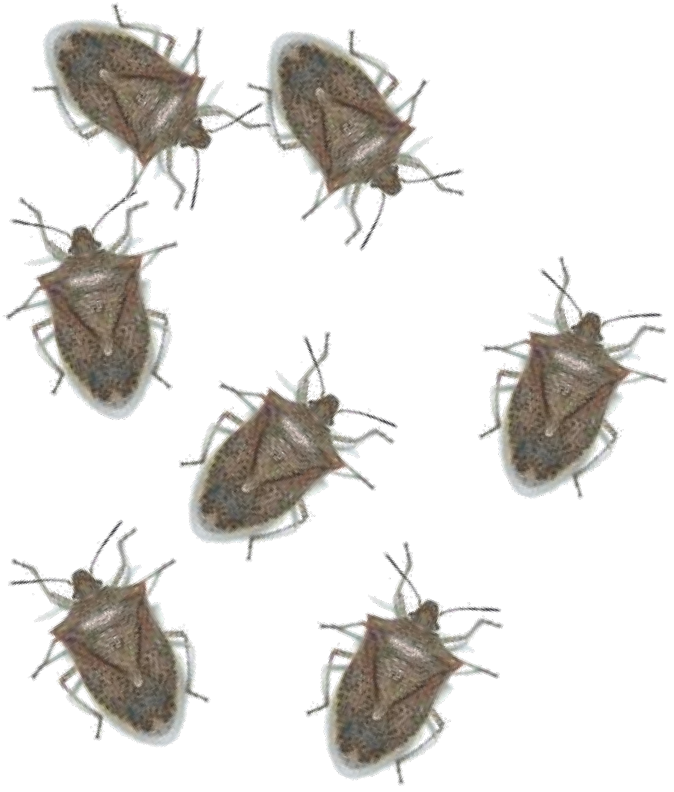
- How much memory / time may be used in any execution?
- Which variables may be the targets of a pointer variable p?

The engineering challenge

- A correct but trivial approximation algorithm may just give the useless answer every time
- The *engineering challenge* is to give the useful answer often enough to fuel the client application
- ... and to do so within reasonable time and space
- This is the hard (and fun) part of static analysis!

Bug finding

```
int main() {
    char *p,*q;
     p = NULL;
     printf("%s",p);
     q = (char *)malloc(100);
     p = q;
     free(q);
     *p = 'x';
     free(p);
     p = (char*)malloc(100);
     q = (char*)malloc(100);
     q = p;
     strcat(p,q);
}
```



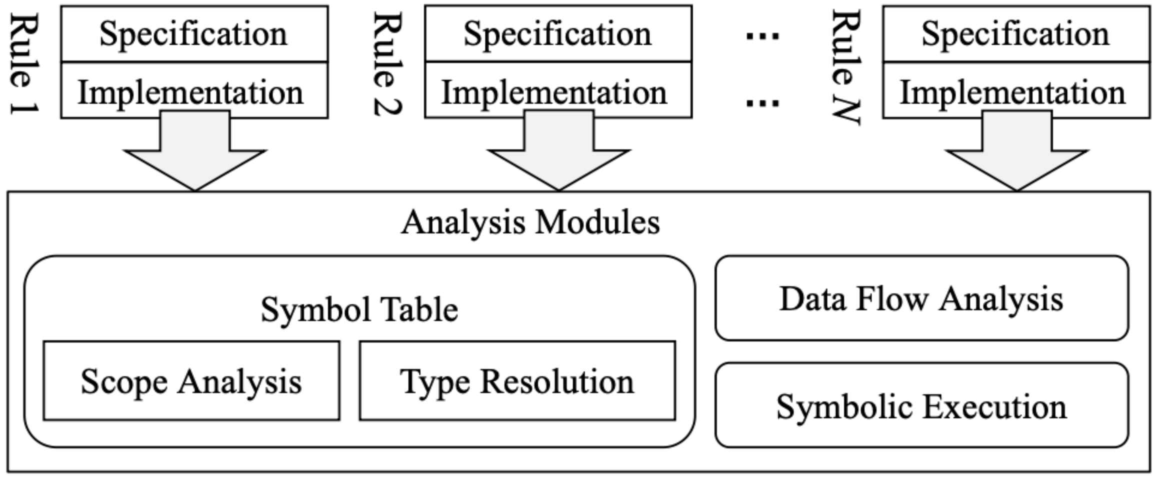
```
gcc –Wall foo.c
lint foo.c
```

No errors!

https://en.wikipedia.org/wiki/Lint_(software)

Static Code Analyzers (Linters)

PMD - source code analyzer



PMD: https://github.com/pmd/pmd
SpotBugs: https://github.com/spotbugs/spotbugs
SonarQube: https://github.com/SonarSource/sonar-java

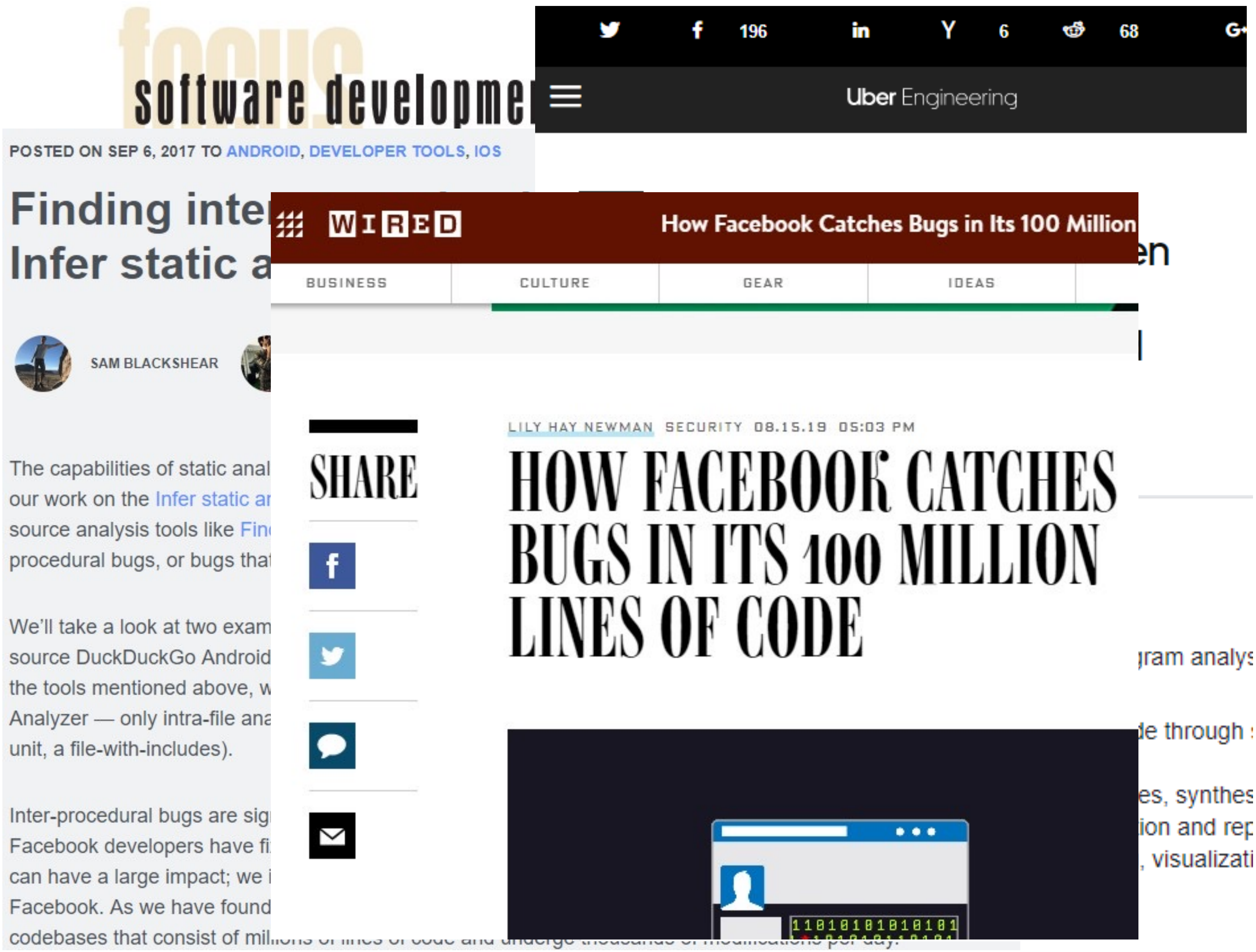
Does anyone use static program analysis?

For optimization:

- every optimizing compiler and modern JIT

For verification or error detection:

- Astrée
- Infer
- COVERITY
- PVS-Studio
- klocwork
- GrammaTech CodeSonar
- IBM Security AppScan



software developme

POSTED ON SEP 6, 2017 TO ANDROID, DEVELOPER TOOLS, IOS

Finding inte

Infer static a

SAM BLACKSHEAR

The capabilities of static anal
our work on the Infer static a
source analysis tools like Fin
procedural bugs, or bugs tha

We'll take a look at two exam
source DuckDuckGo Android
the tools mentioned above, w
Analyzer — only intra-file ana
unit, a file-with-includes).

Inter-procedural bugs are sig
Facebook developers have fi
can have a large impact; we i
Facebook. As we have found
codebases that consist of mil

196 6 68 G

Uber Engineering

en

gram analys

de through

es, synthes

ion and rep

, visualizati

WIRED How Facebook Catches Bugs in Its 100 Million

BUSINESS CULTURE GEAR IDEAS

SHARE

LILY HAY NEWMAN SECURITY 08.15.19 05:03 PM

HOW FACEBOOK CATCHES BUGS IN ITS 100 MILLION LINES OF CODE

Big Picture

- **Introduction to static analysis**
- Many ways of formulating and implementing analyses
- One popular way of formulating a static analysis: Data flow analysis

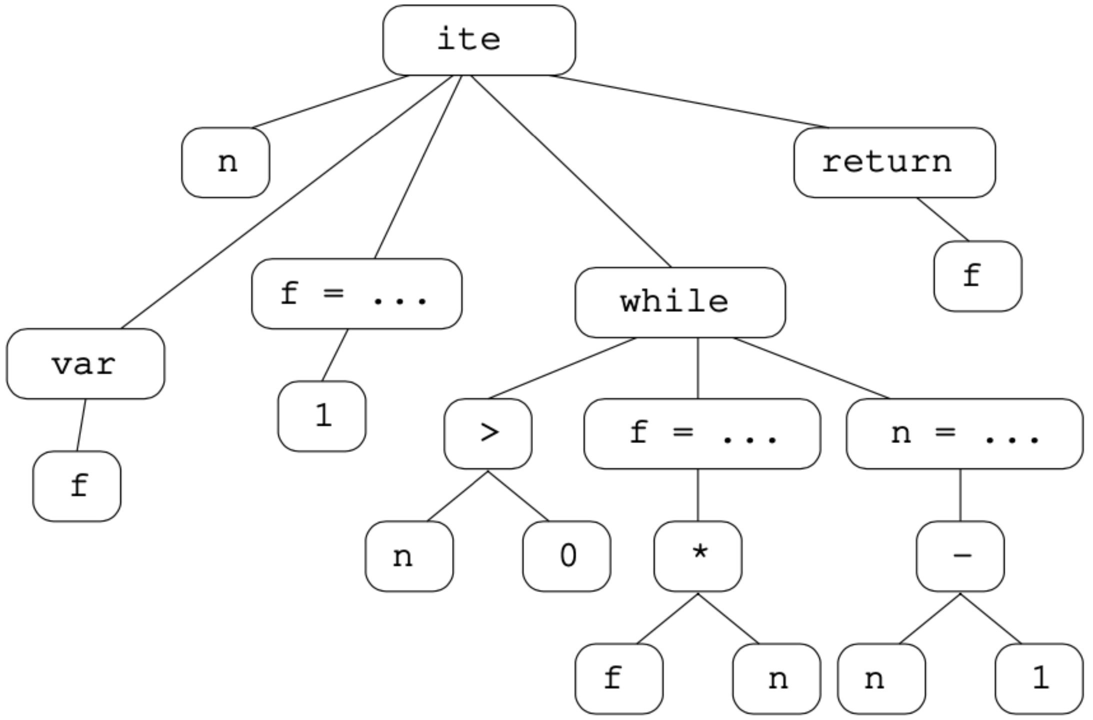
Big Picture

- Introduction to static analysis
- Many ways of formulating and implementing analyses
- One popular way of formulating a static analysis: Data flow analysis

How Programs are Represented?

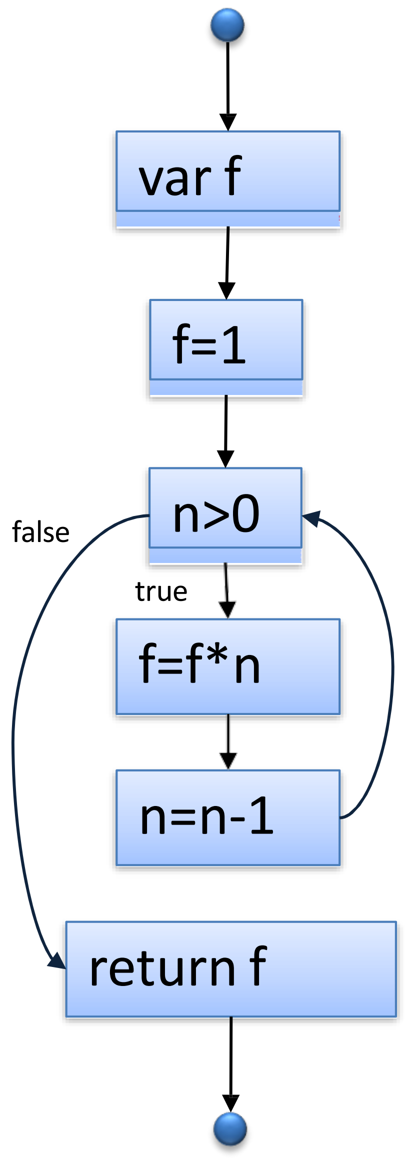
Abstract Syntax Trees

```
ite(n) {
    var f;
    f = 1;
    while (n>0) {
        f = f*n;
        n = n-1;
    }
    return f;
}
```



Control flow graphs

```
ite(n) {
    var f;
    f = 1;
    while (n>0) {
        f = f*n;
        n = n-1;
    }
    return f;
}
```



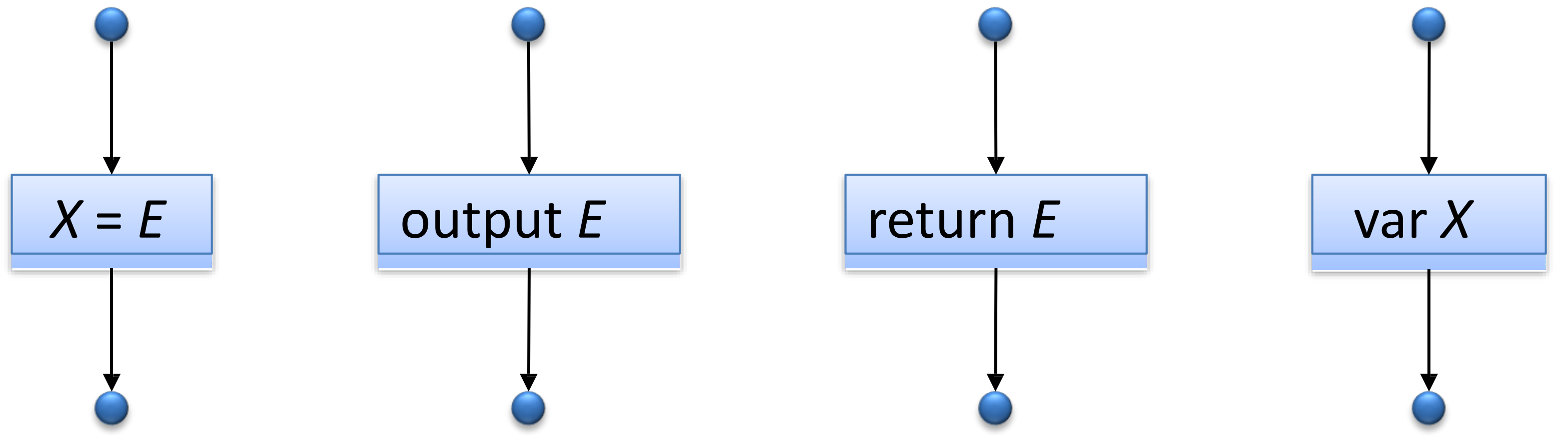
Control flow graphs

- A *control flow graph* (CFG) is a directed graph:
  - *nodes* correspond to program points
    (either immediately before or after statements)
  - *edges* represent possible flow of control
- A CFG always has
  - a single point of *entry*
  - a single point of *exit*

  (think of them as no-op statements)
- Let v be a node in a CFG
  - *pred*(v) is the set of predecessor nodes
  - *succ*(v) is the set of successor nodes

CFG construction (1/3)

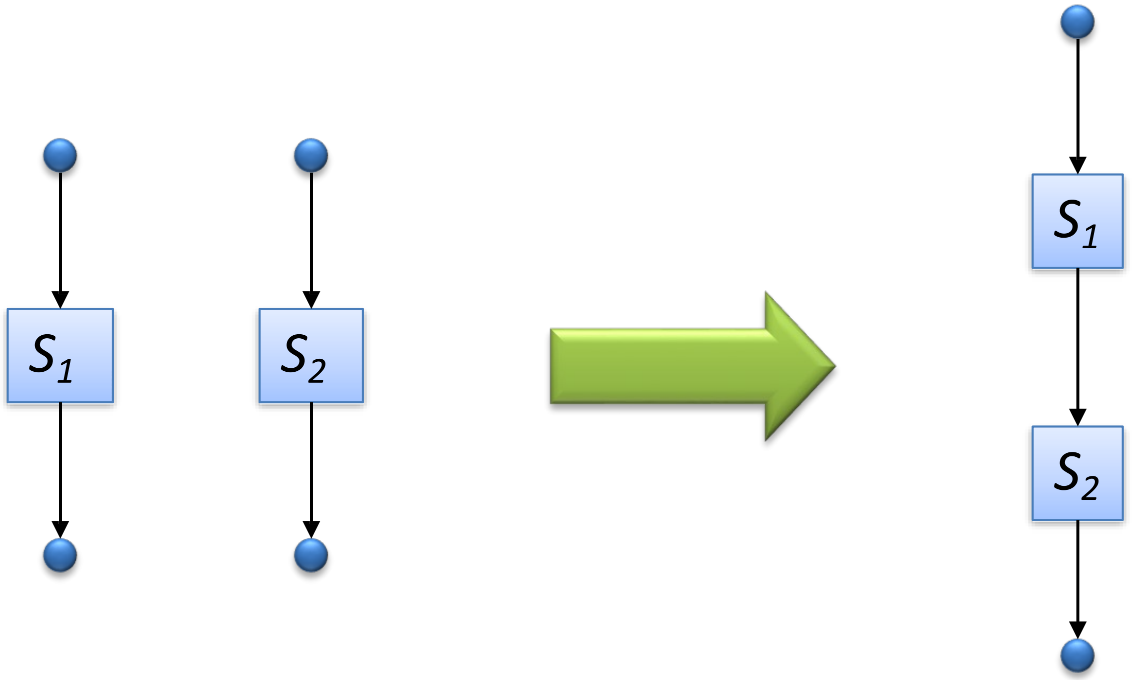
- CFGs are constructed inductively
- CFGs for simple statements etc.:



CFG construction (2/3)

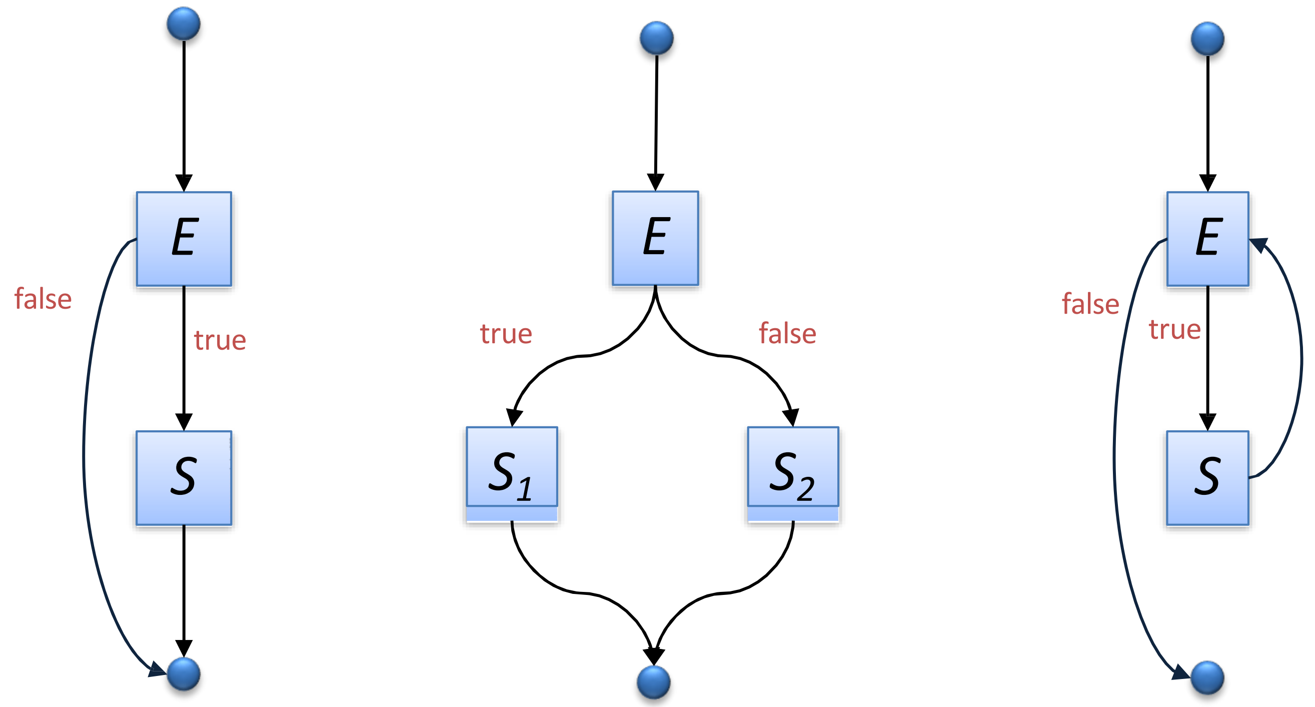
For a statement sequence S_1 S_2:

- eliminate the exit node of S_1 and the entry node of S_2
- glue the statements together



CFG construction (3/3)

Similarly for the other control structures:



Normalization

- Sometimes convenient to ensure that each CFG node performs only one operation

- *Normalization* (*A-normal form*): flatten nested expressions, using fresh variables

```
x = f(y+3)*5;
```

$\Rightarrow$

```
t1 = y+3;
t2 = f(t1);
x = t2*5;
```

Data Flow Analysis

Basic idea

- Propagate analysis information along the edges of a control flow graph
- Goal: Compute analysis state at each program point
- For each statement, define how it affects the analysis state
- For loops: Iterate until fix-point reached

Outline

- First example: Available expressions ←
- Basic principles
- More examples
- Solving data flow problems
- Inter-procedural analysis
- Sensitivities

Available Expression Analysis

Goal: For each program point, compute which expressions must have already been computed, and not later modified

- Useful, e.g., to avoid re-computing an expression
- Used as part of compiler optimizations

To be available on a program point, the ***operands*** of the expression should not be modified on ***any path*** from the occurrence of that expression to the program point.

https://en.wikipedia.org/wiki/Available_expression

Example

```
var x = a + b;
var y = a * b;
while(y > a + b) {
  a = a - 1;
  x = a + b;
}
```

To be available on a program point, the ***operands*** of the expression should not be modified on ***any path*** from the occurrence of that expression to the program point.

https://en.wikipedia.org/wiki/Available_expression

Example

```
var x = a + b;
var y = a * b;
while(y > a + b) {
   a = a - 1;
   x = a + b;
}
```

Available every time execution reaches this point

To be available on a program point, the ***operands*** of the expression should not be modified on ***any path*** from the occurrence of that expression to the program point.

https://en.wikipedia.org/wiki/Available_expression

Transfer Functions

- **Transfer function of a statement:**
  How the statement affects the analysis state
  - Here: Analysis state = available expressions
- **Two functions**
  - gen: Available expressions generated by a statement
  - kill: Available expressions killed by a statement

gen Function

Function $gen : Stmt \rightarrow P(Expr)$

- A statement generates an available expressions e if
  - it evaluates e and
  - it *does not* later write any variable used in e
- Otherwise, function returns empty set

Example:

`var x = a * b;` generates `a * b`

kill Function

Function $kill : Stmt \to P(Expr)$

- A statement kills an available expressions *e* if
  - it modifies any of the variables used in *e*
- Otherwise, function returns empty set

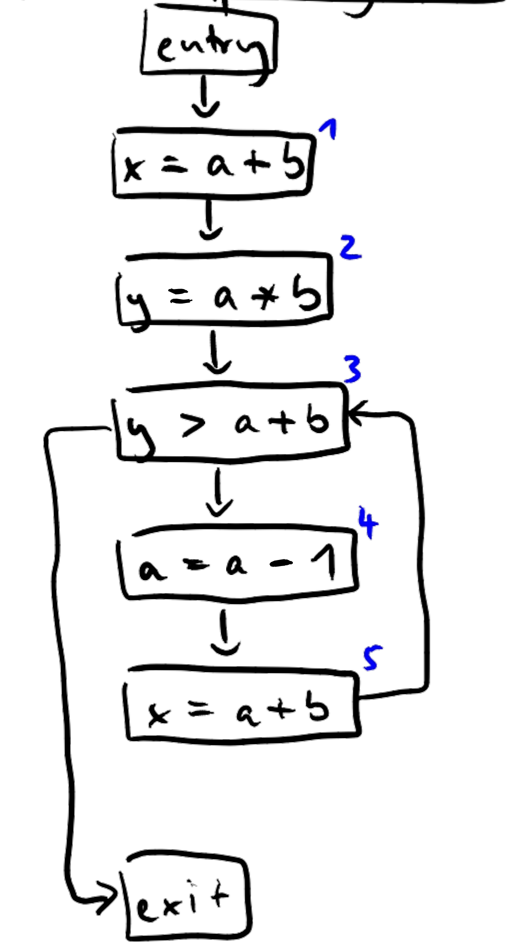
Example:

`a = 23;` kills `a * b`

Example

```
varx = a + b;
vary = a * b;
while(y > a + b) {
  a = a - 1;
  x = a + b;
}
```

Control flow graph



Non-trivial expressions

a+b

a*b

a - 1

Transfer function for each statement:

Statement s	gen(s)	kill(s)
1	{a+b}	∅
2	{a*b}	∅
3	{a+b}	∅
4	∅	{a-1, a+b, a*b}
5	{a+b}	∅

Propagating Available Expressions

- Initially, no available expressions
- Forward analysis: Propagate available expressions in the direction of control flow
- For each statement *s*, outgoing available expressions are:

  incoming avail. exprs. minus *kill*(*s*) plus *gen*(*s*)
- When control flow splits, propagate available expressions both ways
- When control flows merge, intersect the incoming available expressions

Data flow equations

$AE_{entry}(s)$ ... avail. express. at entry of s

$AE_{exit}(s)$ .. avail. express. at exit of s

$AE_{entry}(1) = \emptyset$

$AE_{entry}(2) = AE_{exit}(1)$

$AE_{entry}(3) = AE_{exit}(2) \cap AE_{exit}(5)$

$AE_{entry}(4) = AE_{exit}(3)$

$AE_{entry}(5) = AE_{exit}(4)$

$AE_{exit}(1) = AE_{entry}(1) \cup \{a+b\}$

$AE_{exit}(2) = AE_{entry}(2) \cup \{a*b\}$

$AE_{exit}(3) = AE_{entry}(3) \cup \{a+b\}$

$AE_{exit}(4) = AE_{entry}(4) \setminus \{a+b, a*b, a-1\}$

$AE_{exit}(5) = AE_{entry}(5) \cup \{a+b\}$

Solution of these equations:

s	$AE_{entry}(s)$	$AE_{exit}(s)$
1	$\emptyset$	$\{a+b\}$
2	$\{a+b\}$	$\{a+b, a*b\}$
3	$\{a+b\}$	$\{a+b\}$
4	$\{a+b\}$	$\emptyset$
5	$\emptyset$	$\{a+b\}$

Outline

- First example: Available expressions
- Basic principles ←
- More examples
- Solving data flow problems
- Inter-procedural analysis
- Sensitivities

Quiz

```
var m = x - y;
if(random()) {
  while(m > 0) {
    x = y + 1;
  }
}else{
  n = x - y;
}
z = x - y;
```

Quiz

```
var m = x - y;
if(random()) {
   while(m > 0) {
      x = y + 1;
   }
}else{
   n = x - y;
}
z = x - y;
```

Is `x - y` an available expression when entering this statement?

Quiz

```
var m = x - y;
if(random()) {
  while(m > 0) {
    x = y + 1;
  }
}else{
  n = x - y;
}
z = x - y;
```

No, because modifying `x` kills `x - y`

Is `x - y` an available expression when entering this statement?

Defining a Data Flow Analysis

Any data flow analysis:

Defined by six properties

- Domain (**facts**)
- Direction
- Transfer function
- Meet operator
- Boundary condition
- Initial values

Domain

- **Analysis associates some information with every program point**
  - “Information” means elements of a set
- **Domain of the analysis: All possible elements the set may have**
  - E.g., for available expressions analysis: Domain is set of non-trivial expressions

Direction

- **Analysis propagates information along the control flow graph**
  - Forward analysis: Normal flow of control
  - Backward analysis:Invert all edges
    - Reasons about executions in reverse
- **E.g., available expression analysis: Forward**

Transfer Function

- **Defines how a statement affects the propagated information**
- $DF_{exit}(s)$= **some function of** $DF_{entry}(s)$
- **E.g., for available expression analysis:**
  $AE_{exit}(s) = (AE_{entry}(s) \setminus kill(s)) \cup gen(s)$

Meet Operator

- **What if two statements s_1, s_2 flow to a statement s?**
  - Forward analysis: Execution branches merge
  - Backward analysis: Branching point
- **Meet operator defines how to combine the incoming information**
  - Union: $DF_{entry}(s) = DF_{exit}(s_1) \cup DF_{exit}(s_2)$
  - Intersection: $DF_{entry}(s) = DF_{exit}(s_1) \cap DF_{exit}(s_2)$

Meet Operator

- **What if two statements s_1, s_2 flow to a statement s?**
  - Forward analysis: Execution branches merge
  - Backward analysis: Branching point
- **Meet operator defines how to combine the incoming information**
  - Union: $DF_{entry}(s) = DF_{exit}(s_1) \cup DF_{exit}(s_2)$
  - Intersection: $DF_{entry}(s) = DF_{exit}(s_1) \cap DF_{exit}(s_2)$

E.g., available expressions analysis

Boundary Condition

- **What information to start with at the first CFG node?**
  - Forward analysis: First node is entry node
  - Backward analysis: First node is exit node
- **Common choices**
  - Empty set
  - Entire domain

Boundary Condition

- **What information to start with at the first CFG node?**
  - Forward analysis: First node is entry node
  - Backward analysis: First node is exit node
- **Common choices**
  - Empty set
  - Entire domain

E.g., available expressions analysis

Initial Values

- **What is the information to start with at intermediate nodes?**
- **Common choices**
  - Empty set
  - Entire domain

Initial Values

- **What is the information to start with at intermediate nodes?**
- **Common choices**
  - Empty set
  - Entire domain

E.g., available expressions analysis

Defining a Data Flow Analysis

Any data flow analysis:

Defined by six properties

- Domain
- Direction
- Transfer function
- Meet operator
- Boundary condition
- Initial values

Defining a Data Flow Analysis

Any data flow analysis:

Defined by six properties

Domain	Non-trivial expressions
Direction	Forward
Transfer function	$AE_{exit}(s) = (AE_{entry} \setminus kill(s)) \cup gen(s)$
Meet operator	Intersection ($\cap$)
Boundary condition	$AE_{entry}(entryNode) = \emptyset$
Initial values	$\emptyset$

Example: Available expressions

Classifying Data Flow Analyses

- ***Forward*** or ***backward***, or sometimes both

- Whether facts ***may*** or ***must*** be true
  - ➢ In a *may* analysis, we care about the facts that ***may be true*** at *p*. That is, they are true ***for some path* up to or from *p***, depending on the direction of the analysis.
  - ➢ In a *must* analysis, we care about the facts that ***must be true* at *p***. That is, they are true ***for every path* up to or from *p***.

- Within a procedure (***intra-procedural***) or between procedures (***inter-procedural***)

Outline

- First example: Available expressions
- Basic principles
- More examples ⟵
- Solving data flow problems
- Inter-procedural analysis
- Sensitivities

Data Flow Analyses

- **Seen previously**
  - Available expressions
- **Next**
  - Reaching definitions
  - Very busy expressions
  - Live variables