Static Program Analysis

Part 1 – the TIP language

http://cs.au.dk/~amoeller/spa/

Anders Møller & Michael I. Schwartzbach

Computer Science, Aarhus University

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?
- XSS?
- …

Program points

```
foo(p,x) {
  var f,q;
  if (*p==0) { f=1; }
  else {
    q = malloc;
    *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

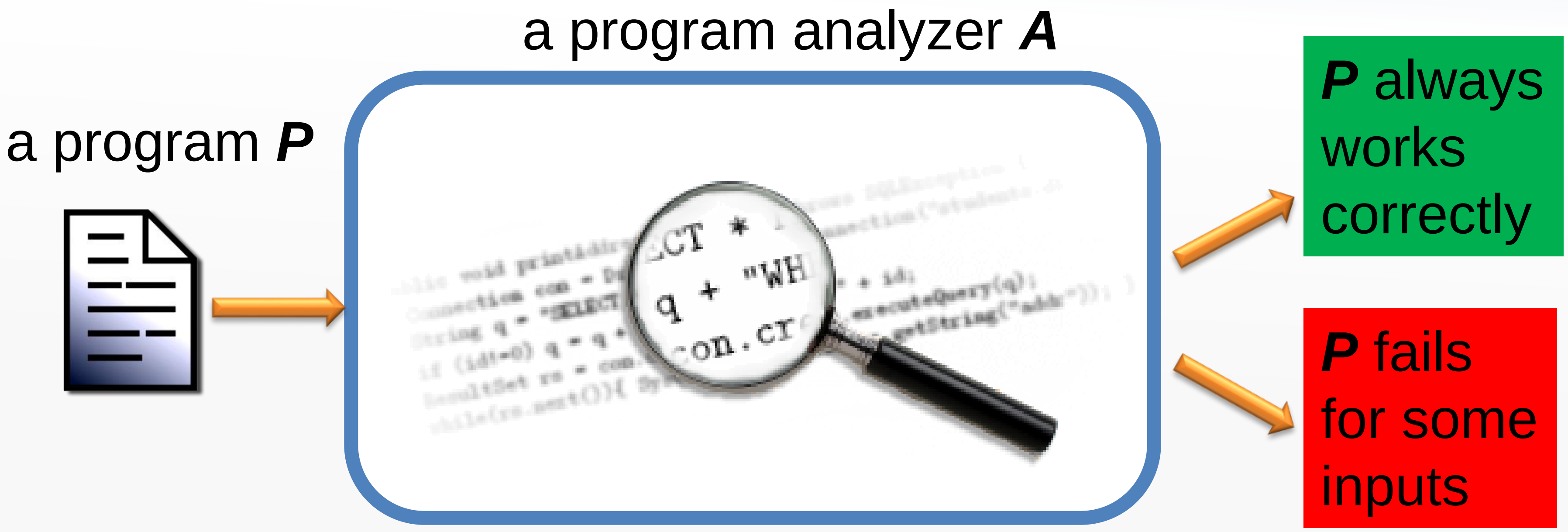
Testing?

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

[Dijkstra, 1972]

- Testing often takes 50% of development cost
- Concurrency errors are hard to (re)produce with testing (“Heisenbugs”)

Programs that reason about programs


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

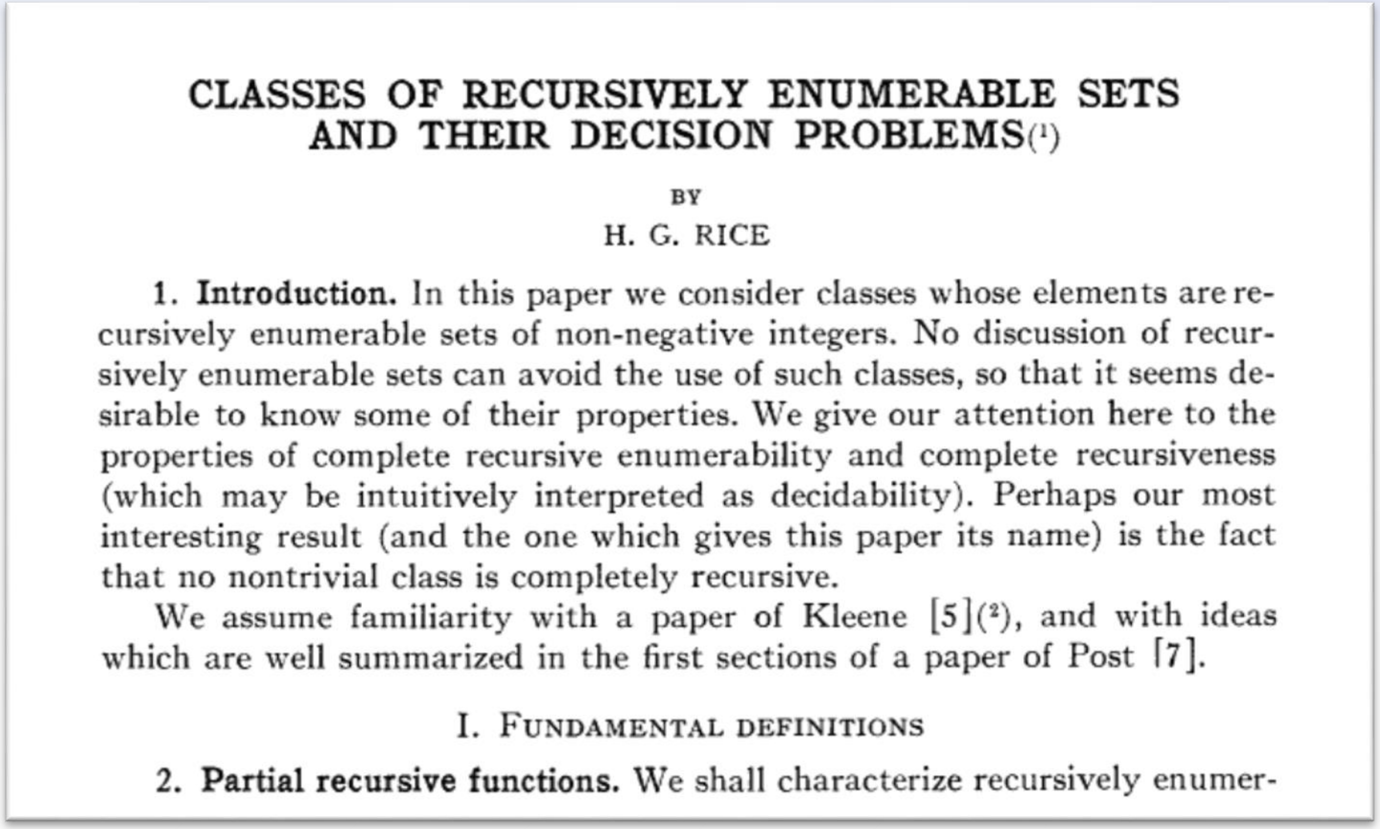
Requirements to the perfect program analyzer

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

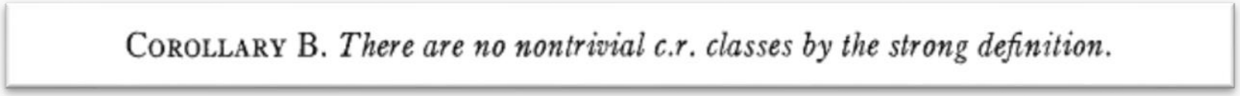
H. G. RICE

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!

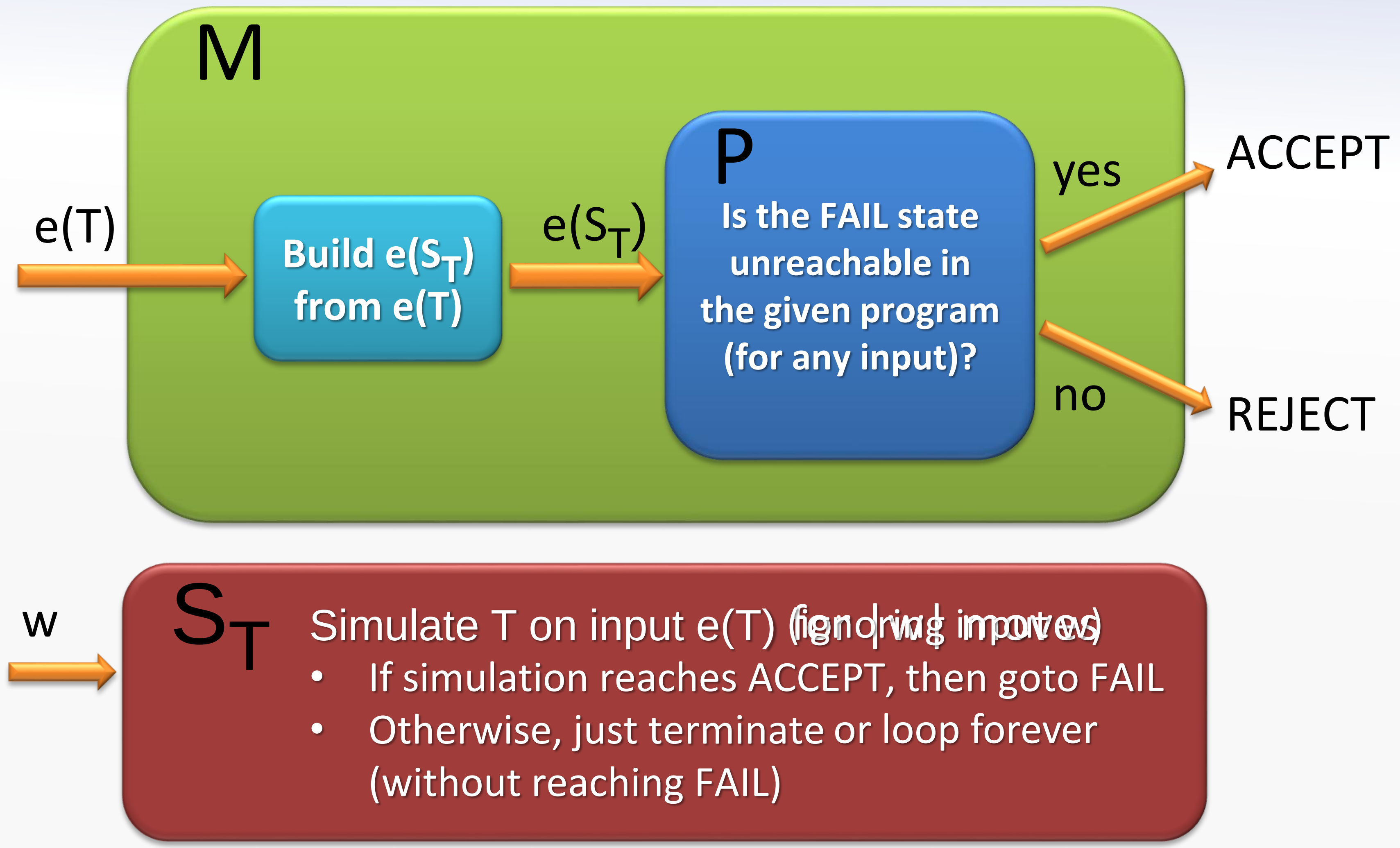
Reduction to the halting problem

- Can we decide if a variable has a constant value?

```
x = 17; if (TM(j)) x = 18;
```

- Here, x is constant if and only if the *j*'th Turing machine does not halt on empty input

Undecidability of program correctness


M
e(T)
Build e(S_T) from e(T)
e(S_T)
P
Is the FAIL state unreachable in the given program (for any input)?
yes
ACCEPT
no
REJECT
w
S_T

Simulate T on input e(T) (ignoring input w) (for |w| moves)

- If simulation reaches ACCEPT, then goto FAIL
- Otherwise, just terminate or loop forever (without reaching FAIL)

Does M accept input e(M)?

(Note: this proof works even if we only consider programs that always terminate!)

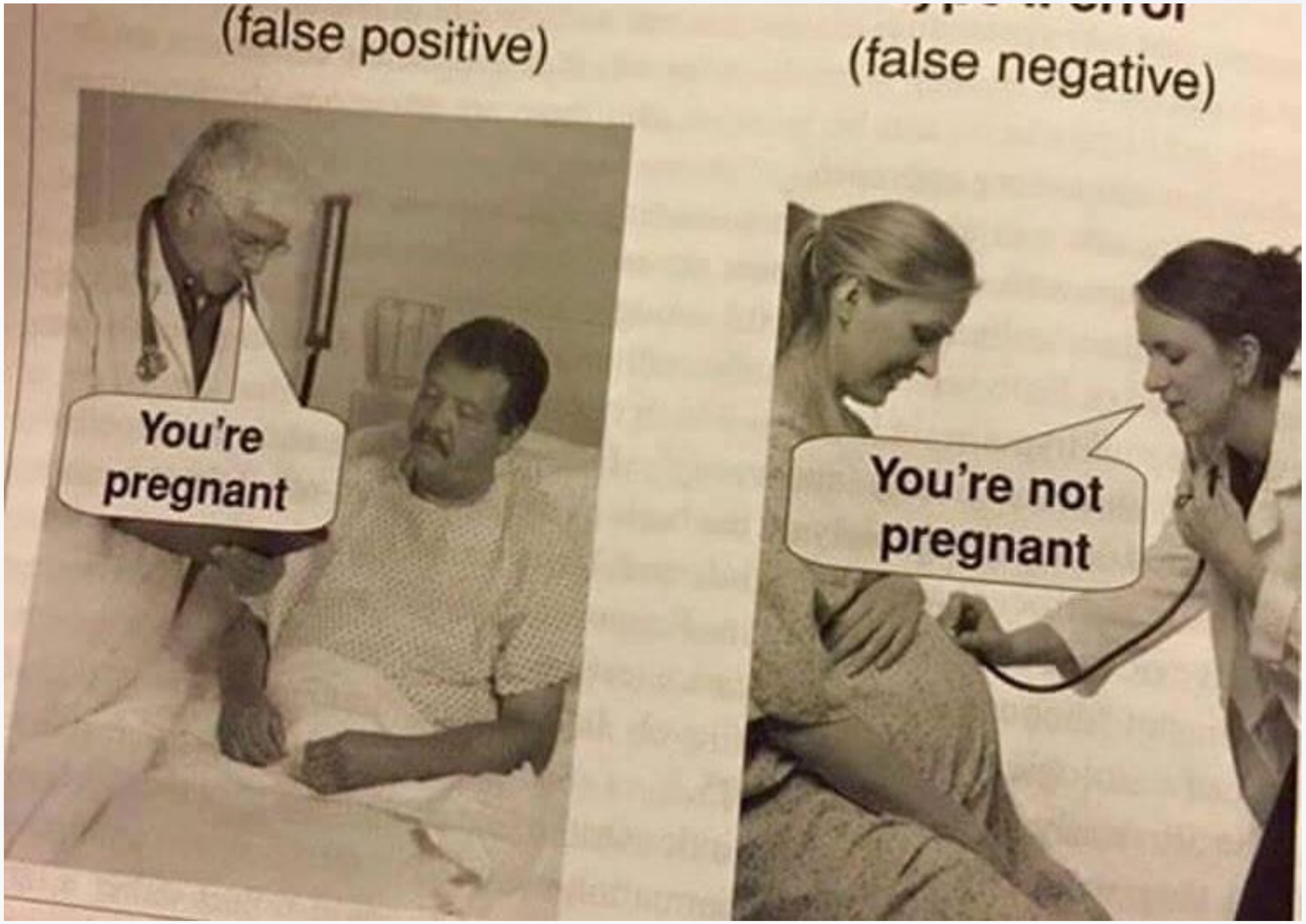
Approximation

- *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

False positives and false negatives



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

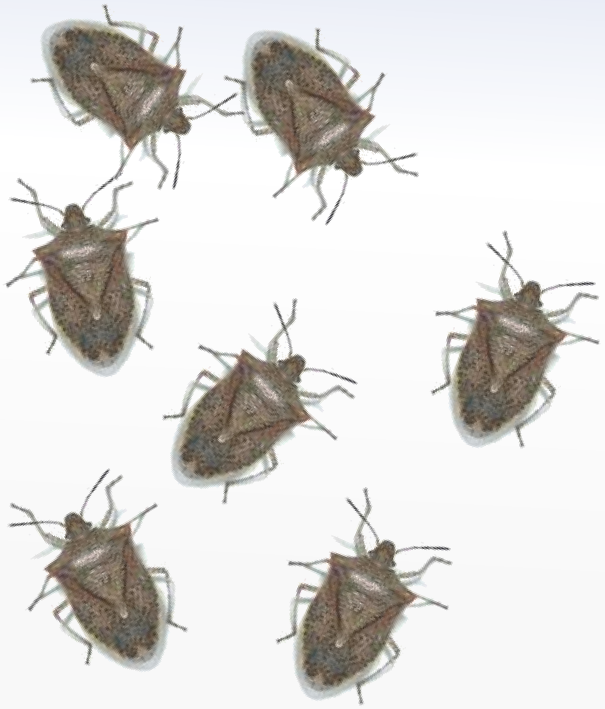
- How much memory / time may be used in any execution?
- Which variables may be the targets of a pointer variable p?

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);
  p = (char *)malloc(100);
  q = p;
  strcat(p,q);
}
```



```
gcc -Wall foo.c
lint foo.c

No errors!
```

Does anyone use static program analysis?

For optimization:

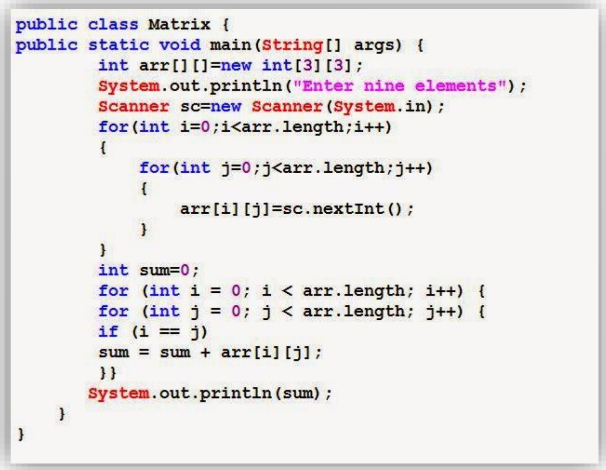
- every optimizing compiler and modern JIT

For verification or error detection:

- Astrée
- Infer
- COVERITY
- PVS-Studio
- FORTIFY
- Semmle™
- klocwork
- GrammaTech CodeSonar
- IBM Security AppScan
- CHECKMARX

A constraint-based approach

Conceptually separates the analysis specification from algorithmic aspects and implementation details



```
public class Matrix {
public static void main(String[] args) {
        int arr[][]=new int[3][3];
        System.out.println("Enter nine elements");
        Scanner sc=new Scanner(System.in);
        for(int i=0;i<arr.length;i++)
        {
            for(int j=0;j<arr.length;j++)
            {
                arr[i][j]=sc.nextInt();
            }
        }
        int sum=0;
        for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j++) {
        if (i == j)
        sum = sum + arr[i][j];
        }}
       System.out.println(sum);
    }
}
```

program to analyze

constraint solver

mathematical constraints

⟦p⟧ = &int
⟦q⟧ = &int
⟦alloc⟧ = &int
⟦x⟧ = ϕ
⟦foo⟧ = ϕ
⟦&n⟧ = &int
⟦main⟧ = ()->int

solution

Challenging features in modern programming language

- Higher-order functions
- Mutable records or objects, arrays
- Integer or floating-point computations
- Dynamic dispatching
- Inheritance
- Exceptions
- Reflection
- …

The TIP language

- ***T***iny ***I***mperative ***P***rogramming language
- Example language used in this course:
  - minimal C-style syntax
  - cut down as much as possible
  - enough features to make static analysis challenging and fun
- Scala implementation available

Expressions

$$
\begin{aligned}
E \rightarrow\ & I \\
\mid\ & X \\
\mid\ & E{+}E \mid E{-}E \mid E{*}E \mid E/E \mid E{>}E \mid E{==}E \\
\mid\ & (\,E\,) \\
\mid\ & \texttt{input}
\end{aligned}
$$

- *I* represents an integer constant
- *X* represents an identifier (x, y, z,…)
- `input` expression reads an integer from the input stream
- comparison operators yield 0 (false) or 1 (true)

Statements

$$\begin{array}{lll}
S & \rightarrow & X = E; \\
 & | & \texttt{output}\ E; \\
 & | & S\ S \\
 & | & \\
 & | & \texttt{if}\ (E)\ \{\ S\ \}\ [\texttt{else}\ \{\ S\ \}]^{?} \\
 & | & \texttt{while}\ (E)\ \{\ S\ \}
\end{array}$$

- In conditions, 0 is false, all other values are true
- The `output` statement writes an integer value to the output stream

Functions

- Functions take any number of arguments and return a single value:

```
F → X ( X, …, X ) {
        [var X, …, X;]?
        S
        return E;
    }
```

- The optional `var` block declares a collection of uninitialized variables
- Function calls are an extra kind of expressions:

```
E → X ( E, …, E )
```

Records

$$
\begin{aligned}
E \rightarrow\ & \{\ X : E\ ,\ \ldots\ ,\ X : E\ \} \\
\mid\ & E\,.X
\end{aligned}
$$

Heap pointers

$E \rightarrow$ `alloc` E
  | $\&X$
  | $*E$
  | `null`

$S \rightarrow *X = E$;

(No pointer arithmetic)

Function pointers

- Function names denote function pointers

- Generalized function calls:

$$\boxed{E \rightarrow E(\, E \,,\, \ldots \,,\, E \,)}$$

- Function pointers suffice to illustrate the main challenges with methods and higher-order functions

Programs

- A program is a collection of functions
- The function named `main` initiates execution
  - its arguments are taken from the input stream
  - its result is placed on the output stream
- We assume that all declared identifiers are unique

$$\boxed{P \rightarrow F \ldots F}$$

An iterative factorial function

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

A recursive factorial function

```
rec(n) {
  var f;
  if (n==0) {
    f=1;
  } else {
    f=n*rec(n-1);
  }
  return f;
}
```

An unnecessarily complicated function

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

```
main() {
  var n;
  n = input;
  return foo(&n,foo);
}
```

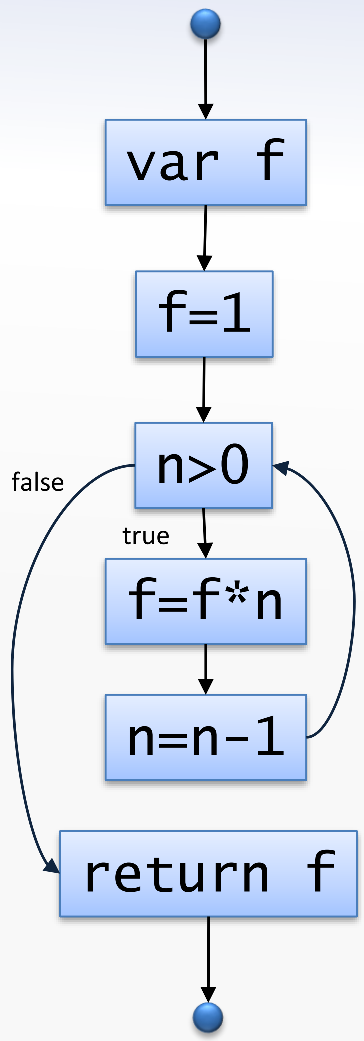
Beyond TIP

Other common language features
in mainstream languages:

- global variables
- objects
- nested functions
- …

Control flow graphs

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



Control flow graphs

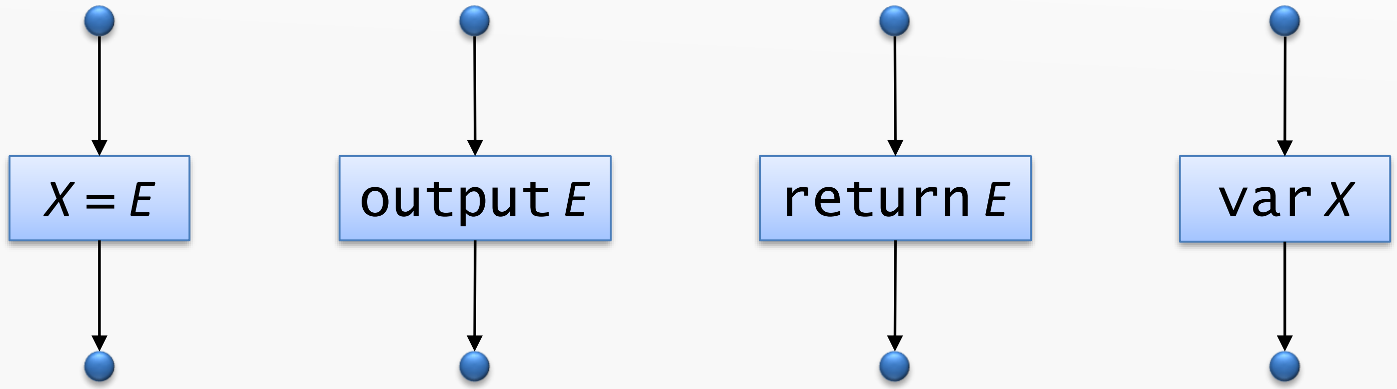
- A *control flow graph* (CFG) is a directed graph:
  - *nodes* correspond to program points
    (either immediately before or after statements)
  - *edges* represent possible flow of control
- A CFG always has
  - a single point of *entry*
  - a single point of *exit*

  (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)

- For the simple `while` fragment of TIP, CFGs are constructed inductively

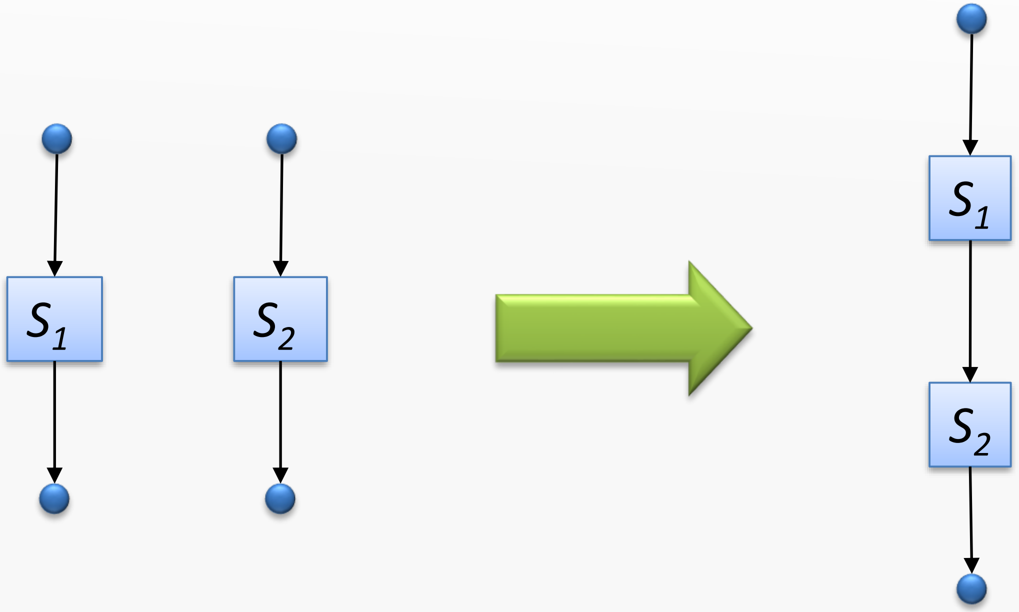
- CFGs for simple statements etc.:



CFG construction (2/3)

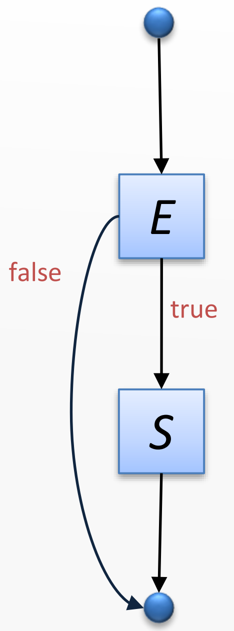
For a statement sequence S_1 S_2:

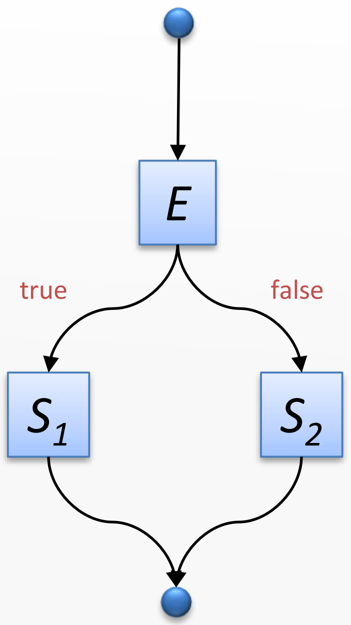
- eliminate the exit node of S_1 and the entry node of S_2
- glue the statements together

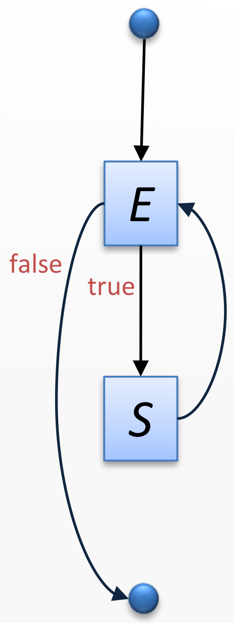


CFG construction (3/3)

Similarly for the other control structures:







Normalization

- Sometimes convenient to assume that there are no nested expressions

- *Normalization*: flatten nested expressions, using fresh variables

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

$\Longrightarrow$

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