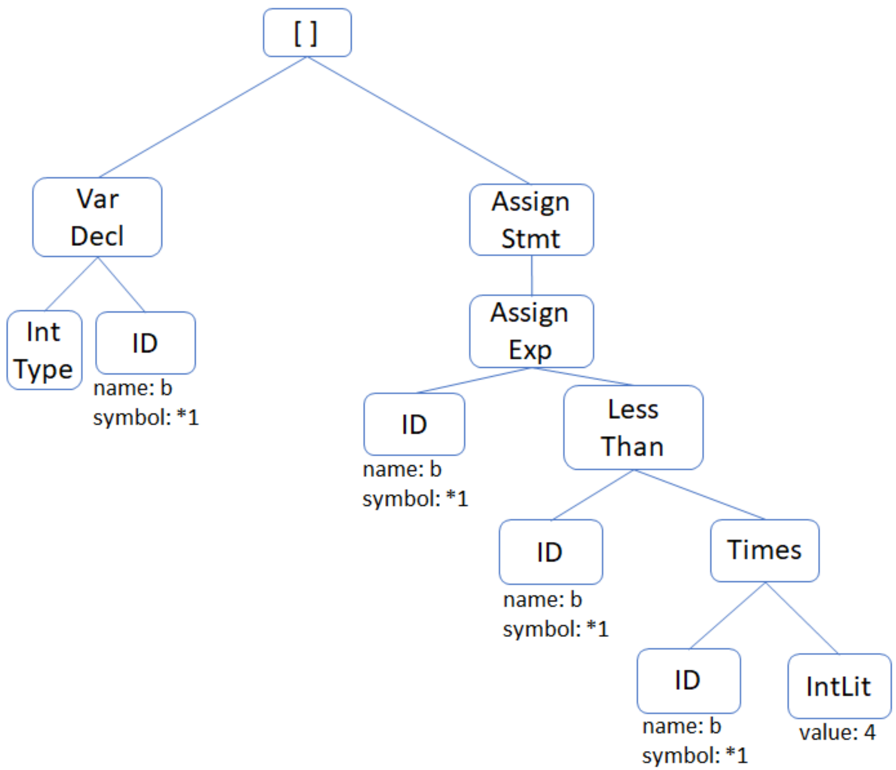
Check-In

Review – Type Checking

Assume a program snippet has generated the following AST. Annotate each node with the type it corresponds to (or error if it is an error type). If a type analysis would issue a report, indicate that as well.



University of Kansas | Drew Davidson

EECS 665 COMPILER CONSTRUCTION

Error Reporting

Last Time

Lecture Review – Type Analysis

Types

- What they are
- Why we have them

Type Rules

- Examples

Connecting operations to their types

- Enrich our static analysis pass

You Should Know

- The meaning of different aspects of type systems
- The simple AST-based type analysis
- How to propagate type errors

Semantics

Handling Errors

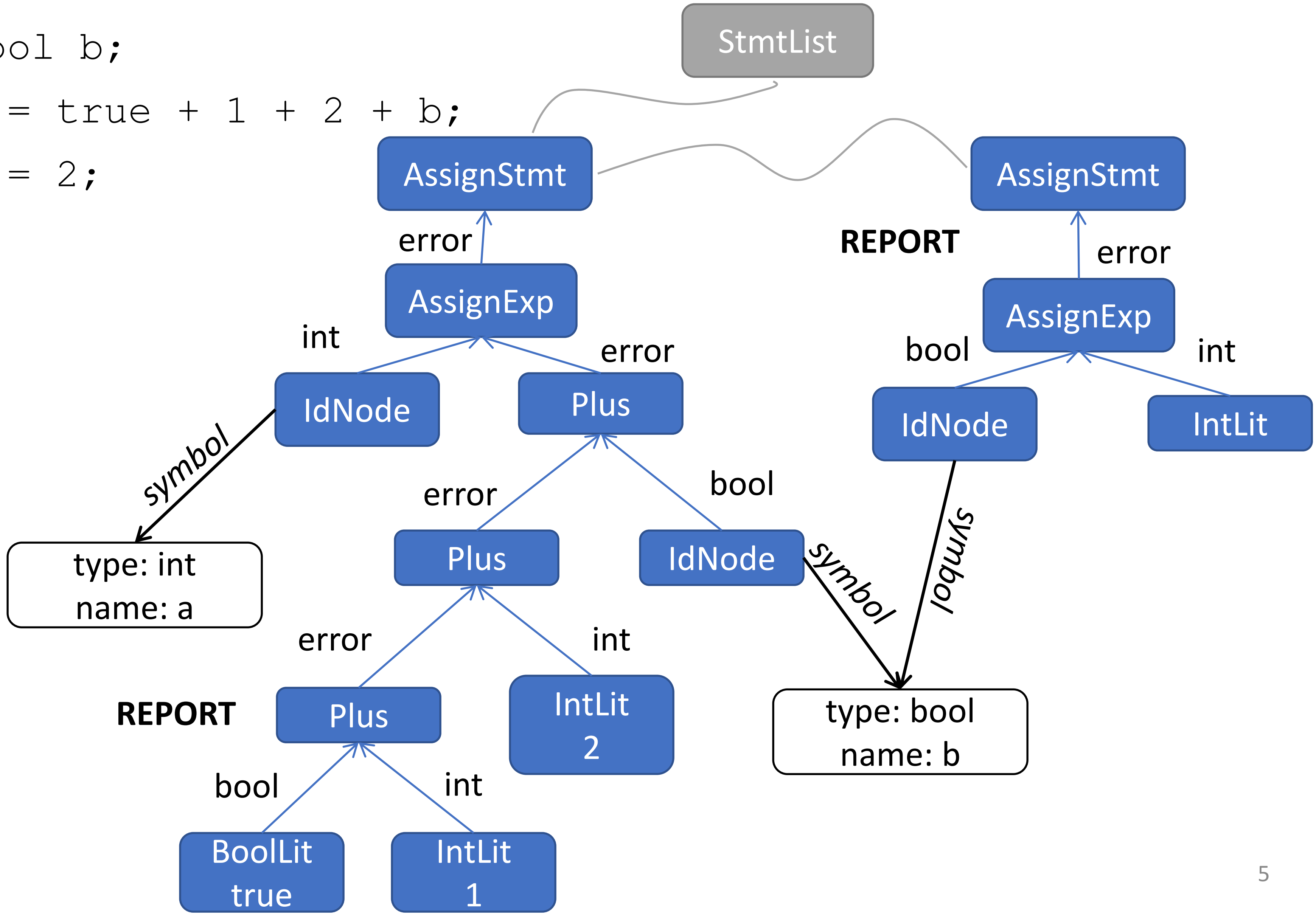
Type Analysis – Implementing Type Checking

- We'd like all *distinct* errors at the same time
  - Don't give up at the first error
  - Don't report the same error multiple times
- When you get error as an operand
  - Don't (re)report an error
  - Again, pass **error** up the tree

Type Error Example

Type Analysis – Implementing Type Checking

```
int a;
bool b;
a = true + 1 + 2 + b;
b = 2;
```



Today's Outline

Lecture Overview – Error Reporting

Error Checking

- What counts as a bad program?
- How do we detect bad programs?

Limits of Analysis

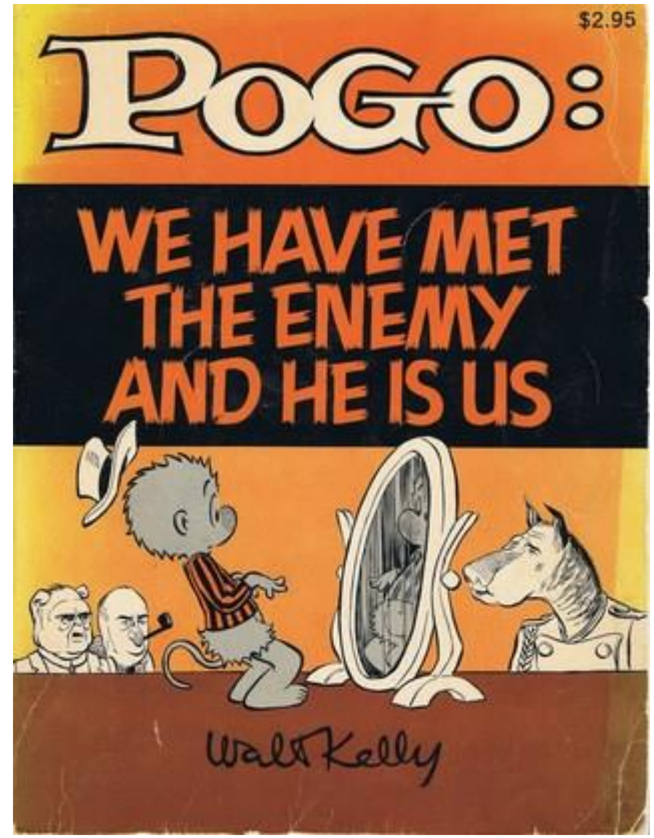
- The halting problem

Semantics

Error Checking
Semantic Analysis

Goal: save programmers from themselves

- It's not enough to compile the programmer's code
- Need to figure out what programmer *meant to code*



Quick Audience Poll

Semantics – Error Checking

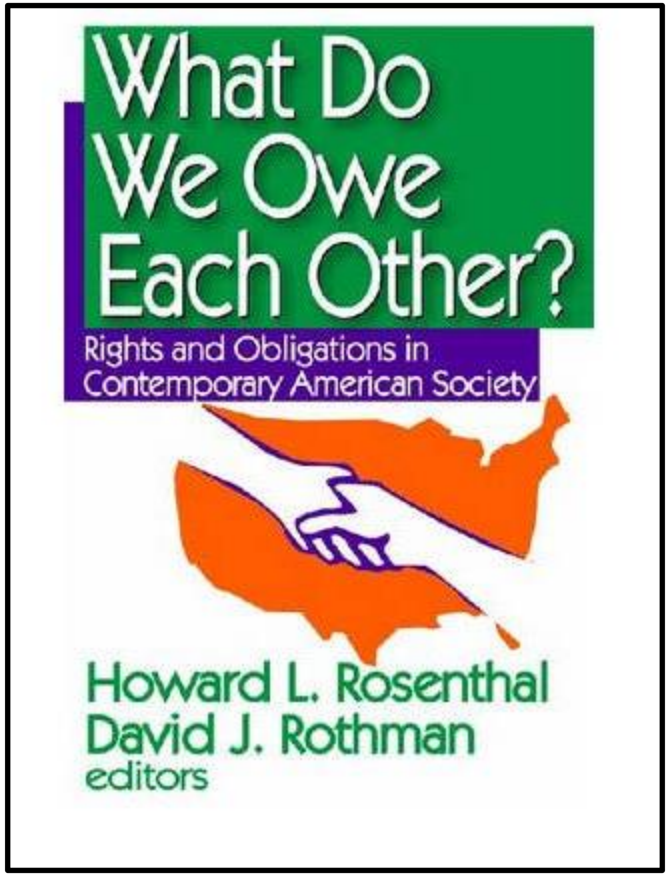
Does this C program compile?

<u>Should</u> this C code compile?

```
int a = 0;
int main(){
   if (false){
      b = 6;
   }
   return a;
}
```

A Compiler's Error-Checking Obligation

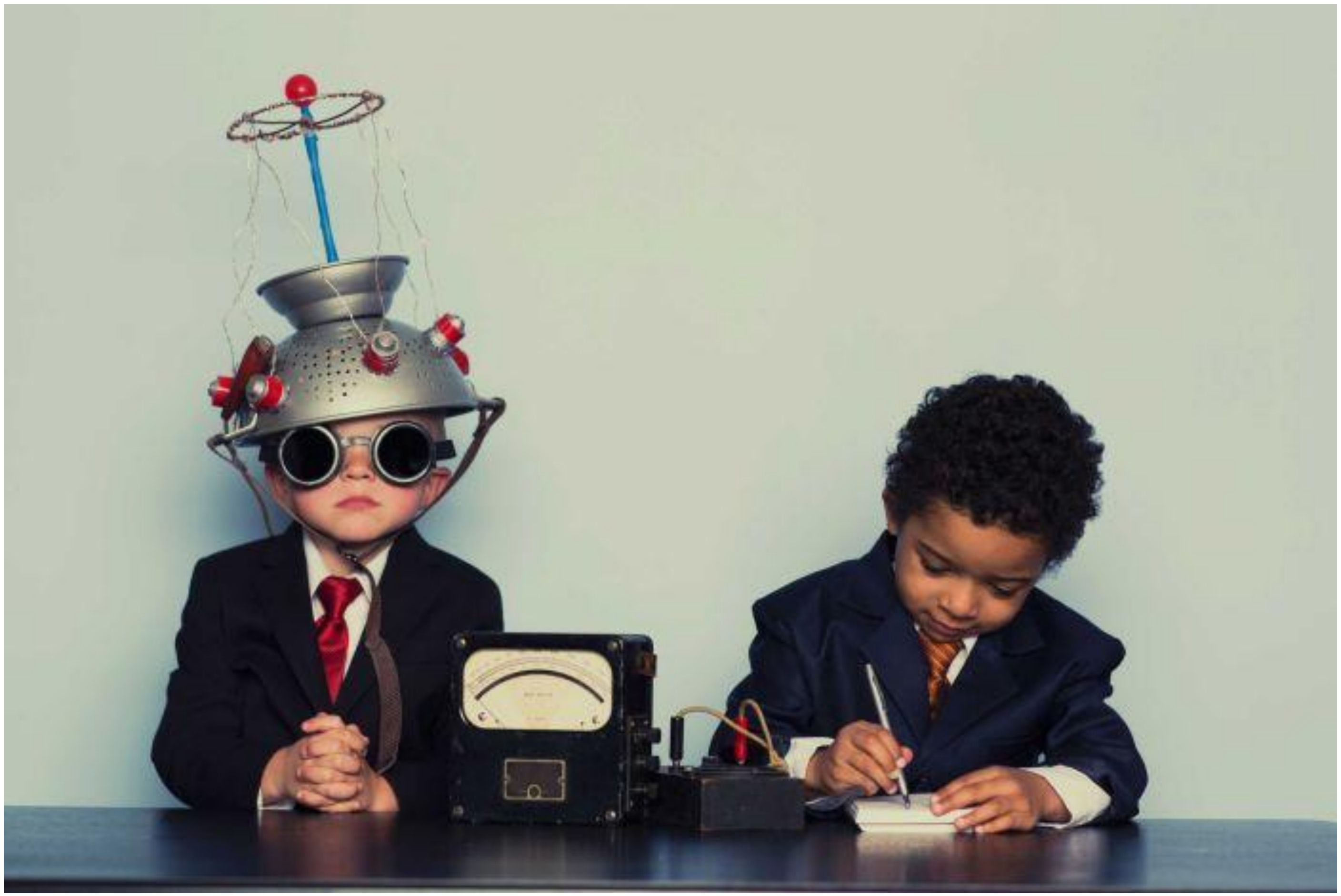
Semantics – Error Checking



Understandability / Consistency

Compiler As Mind Reader

Semantic Analysis – Broad View



A machine that infers your intent

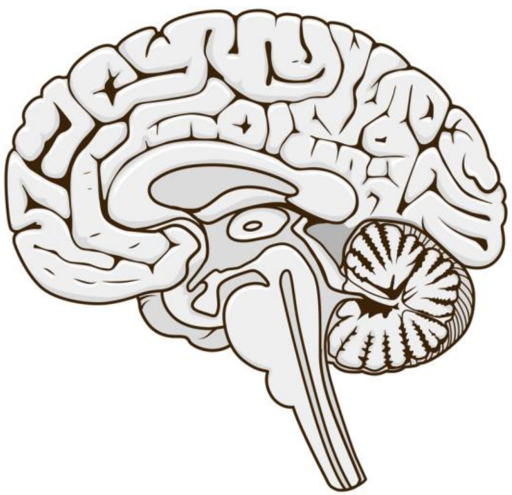
Compiler as Complainer

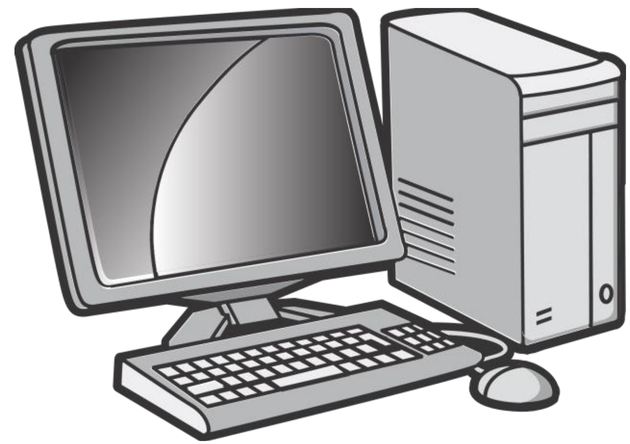
Semantic Analysis – Broad View

A grumpy old man that yells at you for breaking the rules

The Compiler Before the Compiler

Semantic Analysis – Broad View





Bug Hunting

Semantic Analysis – Broad View

How do we prevent nonsense code from executing?

- We'll consider two ways of analysis:
  - Static
  - Dynamic

Putting guardrails on computation

Compiler Perspective

Semantic Analysis – Broad View

Static

- Code analysis without execution

Dynamic

- Code analysis through execution

Checks done at compile time

Analysis part of the compiler itself

Checks done at run time

Analysis embedded into the program

Compiler Focus: Static Analysis

Semantic Analysis – Broad View

Doesn't slow the program down

- Ok to take longer
- Ok to apply more heavyweight analysis

Has a "holistic" view of the program

- Has access to source code
- Knowledge of non-executed program paths

Limits of Error Checking

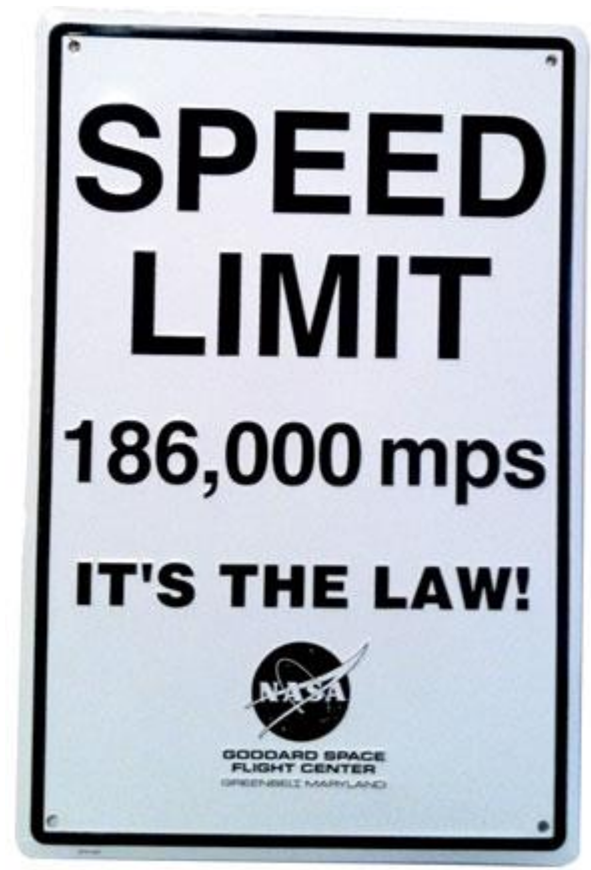
Static Analysis

We'd LOVE to ensure bug-free programs

- Observe and report bugs before they are encountered

Usually we can't do this

- Limits of static analysis



Limits of Static Analysis

Static Analysis

Theoretical argument

Practical argument

The Halting Problem

Static Analysis

Does a computation ever terminate?

Given a description of a Turing machine and its initial input, determine whether the program, when executed on this input, ever halts (completes). The alternative is that it runs forever without halting

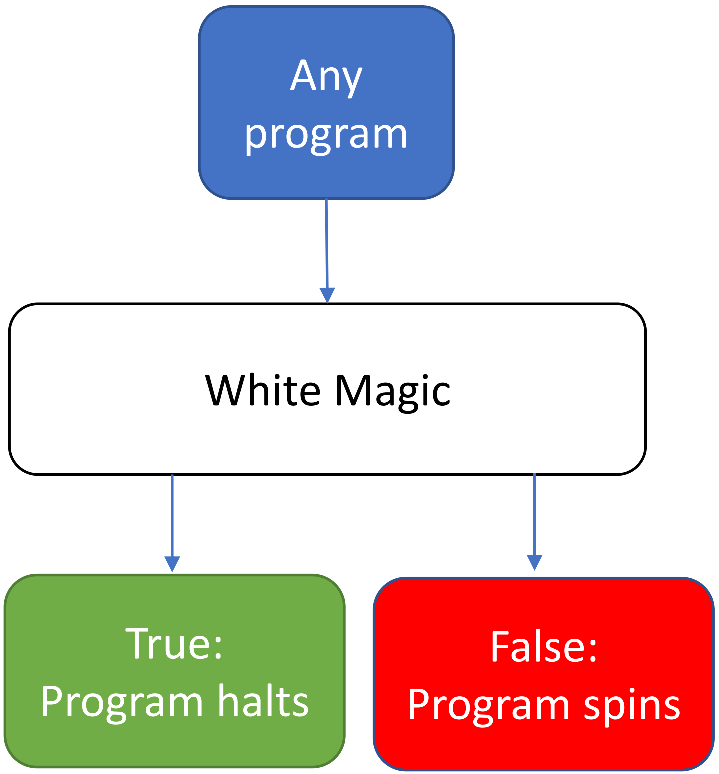
Sketching the Halting Problem

Static Analysis

Effective procedure

- a procedure that is always yields a correct result on any input

Effective method for the halting problem would say:
Return “true” if the program halts on the given input
Return “false” if the program never halts on the given input

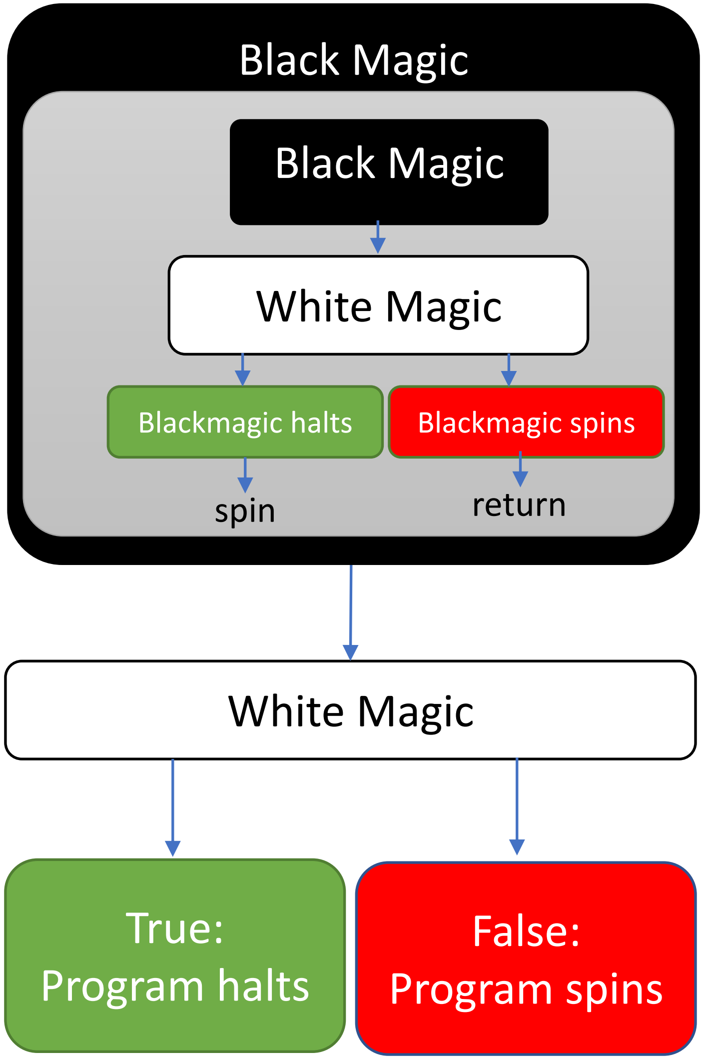


No Effective Method for Halting

Static Analysis

assume `white_magic(Function p)`
returns true if p halts, false if p does not

```
void black_magic(){
    if white_magic(black_magic){
        while true { }
    }
}
```



Implications of the Halting Problem

Static Analysis

What does this have to do with, say, a null pointer analysis?

- No halting solution means no reachability solution

```
int * a = nullptr;
int main(){
   if (a != nullptr){
      *a = 1;
   }
   return a;
}
```

Rice's Theorem
Static Analysis

"All non-trivial semantic properties of programs are undecidable"

Rice's Theorem – Basic Idea

Static Analysis - Limits of Error Checking

What does this have to do with, say, a null pointer analysis?

- No halting means no reachability

```
int main(){
  if (black_magic()){
     int * p = 0;
     *p = 42;
  } else {
     return 0;
  }
}
```

Rice’s Theorem - Implications

Detour – Limits of Error Checking

- We’d like to perfectly capture all bugs
  - We can’t be right all of the time
  - We <u>can</u> choose **HOW** we are wrong

Limits of Static Analysis

Static Analysis

Theoretical argument

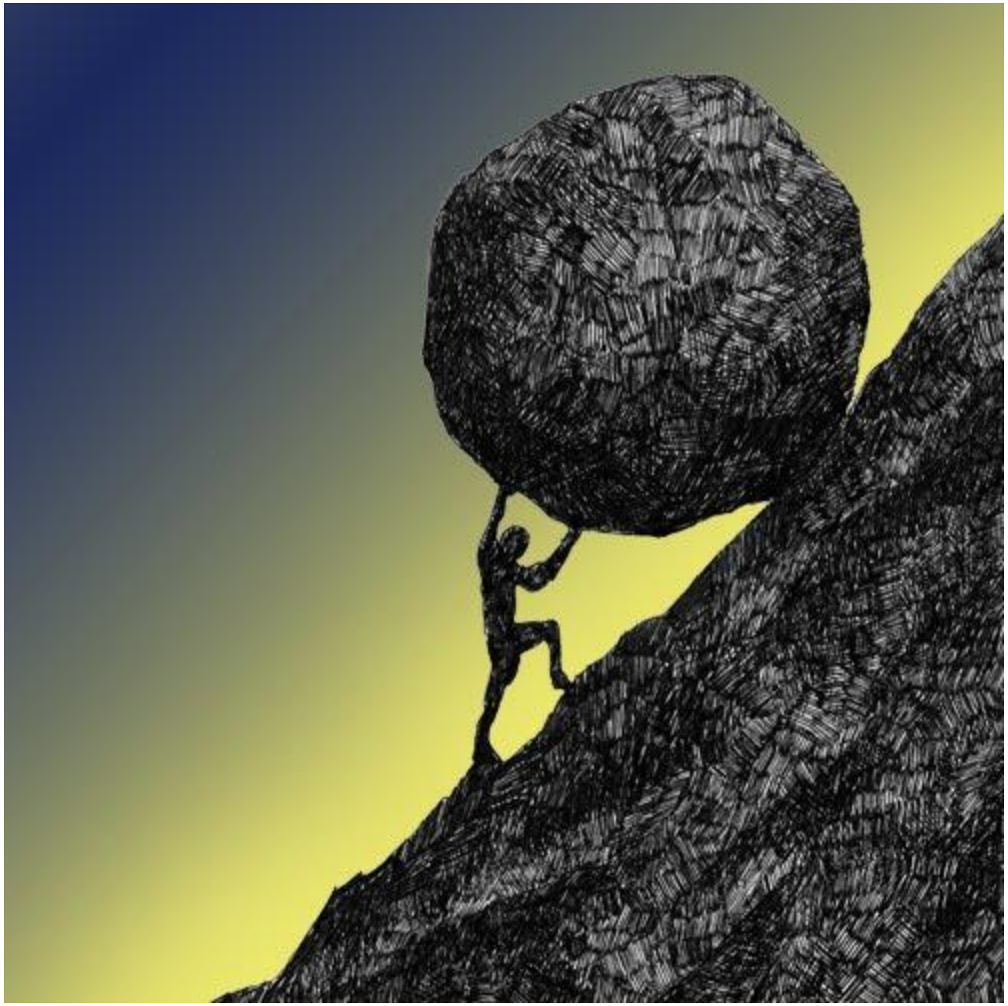
Practical argument

What if we only consider the universe of programs not written by (bleep)heads?

Practical Argument

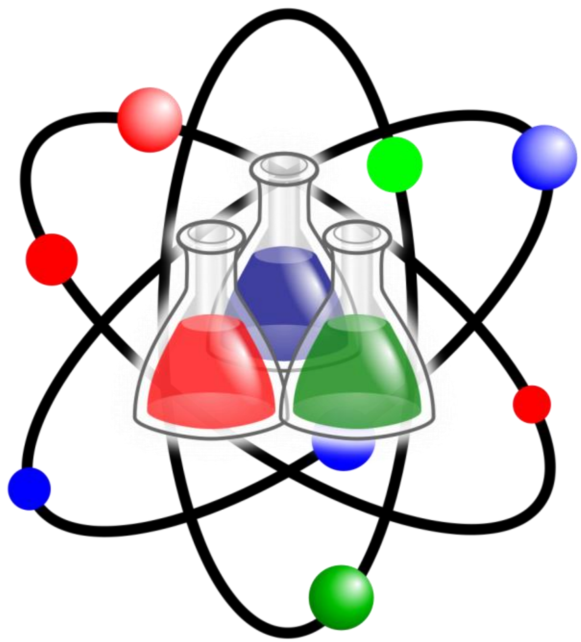
Static Analysis

It’s really hard!



Let’s do some Sciency-Sounding Stuff

Evaluation



Evaluating a Bug Detector

Evaluation

	True	False	
Positive	Has report Has bug Correct	Has report No bug Type I Error	report bug
Negative	No report No bug Correct	No report Has bug Type II Error	No bug report
	Analysis is correct	Analysis is wrong	

Guarantees Under Imperfect Detection

Limits of Error Checking

Consistency / Reliability super important for users

We'd like to limit the <u>kinds</u> of errors we report

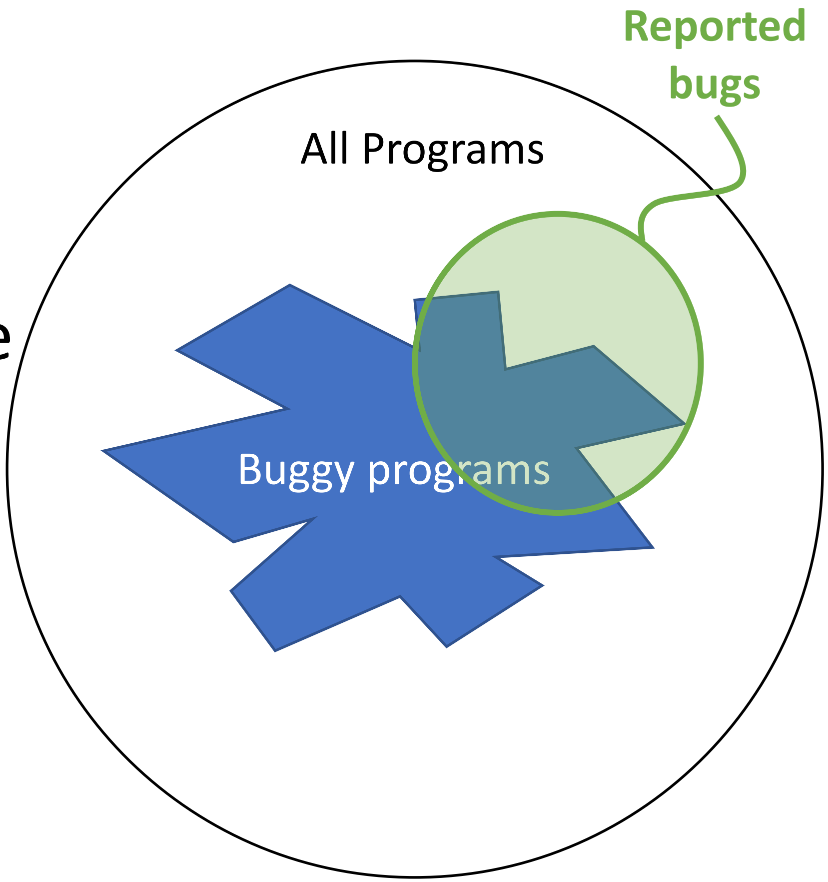
We can choose which type of bug report error to avoid

- Soundness: No false positives
- Completeness: No false negatives

Visual Analogy
Limits of Error Checking

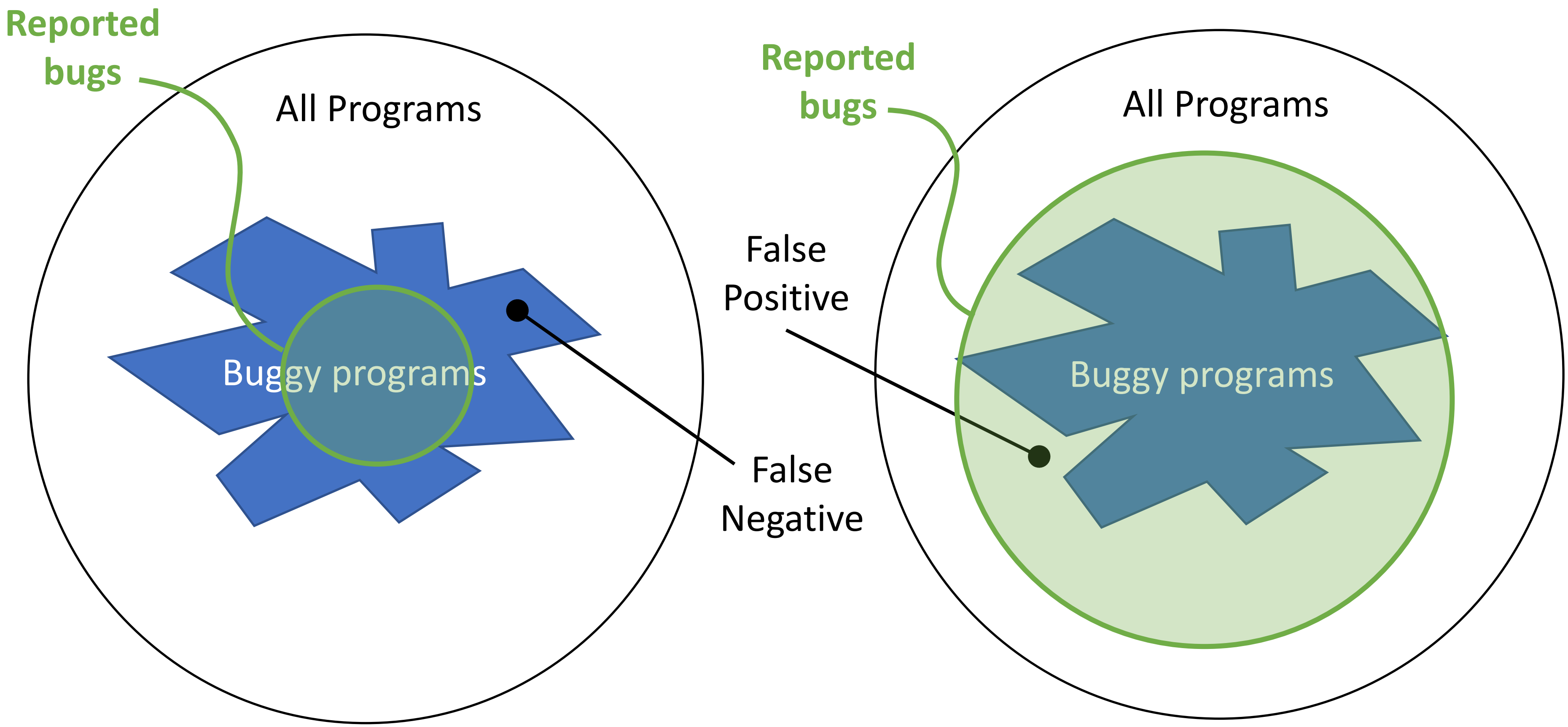
Imagine the universe of all programs is contained in a circle

- You can draw a circle around the programs you report as buggy
- The actual buggy programs occupy a jagged region



Soundness and Completeness

Limits of Error Checking



Sound bug detection

All correct programs pass through
(No false positive problem)

Some buggy programs pass through
(has false negative problem)

Complete bug detection

All buggy programs get flagged
(No false negative problem)

Some correct programs get flagged
(has false positive problem)

Partial Correctness

Limits of Error Checking

- Make best-effort procedures that are neither sound nor complete
- We can analyze the result of a statement under certain assumptions
  - Assume that the statement is executed
  - Assume that the statement actually completes