OWASP LOS ANGELES

BRANDON WU

PROGRAM ANALYSIS

October 23rd, 2024

1 - The State of Software

On August 20th, 2011, Silicon Valley venture capitalist and and entrepreneur Marc Andreessen[1] published an essay entitled "Software is eating the world".

This essay included a lot of business-oriented reasons for why software was immensely disrupting each individual economic sector, for reasons of ease of use, speed of execution, and reach of influence, among others.

Now, more than a decade after this article, it's an incredibly obvious fact that software already has eaten the world. You cannot get away from it – it is everywhere, and it is everything.

[1] Currently a board director for Meta Platforms.

One theme that crops up in the general practice of software engineering is to try to produce as little code as possible, because any human writing any amount of code has some probability of producing a bug.

The less code we write, the less possibility of writing a bug.

So what can we say about the immense volume of code produced by the tens of millions of software developers around the world?

Answer: It is horribly, immensely buggy, and full of mistakes.

When you write a mistake in your code, what does it often look like?

Maybe you made a typo:

✗

```
def fact(n):
  if n == 0:
    return 1
  return n * fac(n - 1)
```

✓

```
def fact(n):
  if n == 0:
    return 1
  return n * fact(n - 1)
```

Or maybe you just have a simple type error:

✗

```
def initialize(data):
  d = {}
  for s in data:
    d[s.strip] = None
  return d
```

✓

```
def initialize(data):
  d = {}
  for s in data:
    d[s.strip()] = None
  return d
```

For most of these errors, they can seem innocent enough. Most are able to be identified and fixed before code ever sees the light of day.

Not all are so lucky, however. Imagine that you are a machine learning engineer.

You have spent the past six months working on a state of the art large language model, and finally you are ready to put it to the test. You just make a few adjustments (mostly adding comments and clarifying names), before you run the model and then decide to go on a vacation to France for two weeks.

When you return from your vacation, you discover that your model failed with:

```
NameError: name 'modle' is not defined. Did you mean: 'model'?
```

This can actually happen.

What's the point? **Programmers make mistakes**.

There are many programmers in the world. Programs that make these kinds of silly, one-off mistakes happen millions of times in a single day. For every mistake which is caught before reaching a sensitive application, there are millions more waiting to be uncovered. We need to build tools, compilers, and programming languages that can guard against these mistakes because the cost of allowing them is too high.

Tony Hoare once referred to his invention of the null pointer[2] as his *billion-dollar mistake*[3]. We are talking about fighting a war upon which rests not only billions upon billions of dollars across every conceivable industry, but upon which rests the security and continued operation of our society.

[2] A programming idiom which has led to countless errors.

[3] Still, one must imagine the person *capable* of making a billion-dollar mistake fortunate.

Well, he tried his best.

How can we prevent these bugs? How do we make sure that, for the prodigious, gargantuan, and overflowing deluge of software that is pumped out every day, it is as safe and as correct as possible? The alternative is a reality that is horrifying to contemplate. [4]

We need a solution which scales with the amount of software in the world. We have to somehow tackle the problem of software correctness, at scale.

Software is eating the world.

It's time to bite back.

[4] I highly recommend the book *This Is How They Tell Me the World Ends: The Cyberweapons Arms Race* by Nicole Perlroth.

2 - Program Analysis

My name is Brandon Wu, and I am a program analysis engineer working at Semgrep.

I have been working at Semgrep for two-and-a-half years now, and I was educated in computer science at Carnegie Mellon University, where I previously lectured on the subject of functional programming.

Semgrep is a software security startup and application security platform that helps developers find and fix security vulnerabilities in their code through **secure guardrails**, at minimal friction to their workflow.

Mission To profoundly improve software security.

@onefiftyman

LinkedIn

As security professionals, I know that I don't have to stress the importance of securing software to any of you.

What I hope that you gain out of this talk is an appreciation of these tools[5] that we *use* to secure software.

There are many vendors out there, and many solutions, and it is quite easy to consider these tools as black boxes, or as magic. I hope that demystifying these tools does a few things for you:

1. shows you what to realistically expect that a SAST tool is capable of
2. gives you the ability to discern when a SAST tool is performing well or not
3. gives you the ability to make discerning choices when choosing a SAST tool

[5] He says, being someone who develops one such tool

Make it **cheap** to make it **expensive** to exploit software.

today, slightly differently:

Make it **easy** to understand why doing program analysis is **hard**.

Def **Program analysis** is the art of discovering undesirable behaviors in programs, usually by automated, programmatic means.

These undesirable behaviors may include correctness, performance, security, and legibility. Ultimately, it encompasses any property which is worth testing, of a program.

In essence, program analysis entails **writing programs to analyze programs**.

```
def analyze(program):
  if programIsBad(program):
    dont()
```

Figure 1: An extremely simplified view of any program analysis tool.

Program analysis generally comes in one of two flavors:

Def **Dynamic program analysis** has to do with figuring out program behavior by observing its behavior at run-time.

This can include things like **fuzzing**, which involves running the program on a wide range of random inputs, **profiling**, which involves measuring the run-time of a program on some inputs, and even the simple act of writing tests.

Dynamic program analysis is useful, and goes straight to the source in terms of the program's actual behavior, but it is limited in some other ways. Notably, if the target program loops forever or takes a really long time, then dynamic program analysis will do the same.

Another is that this doesn't necessarily stop the code from being written or committed in the first place. We want something even farther "left", if possible.

Def **Static program analysis** concerns ascertaining properties of programs **without ever running the program**.

" Costs to fix in development are 10 times lower than in testing, and 100 times lower than in production. "

This will be our focus for today. Specific applications of this analysis include:

- **static application security testing** (or SAST), which is the process of applying static program analysis to code for security purposes
- syntax highlighting, which looks at a (possibly incomplete) program and tries to color it, as its being written
- autoformatting, which looks at a program and tries to make it adhere to a certain stylistic convention
- type-checking, which looks at a program and ascertains what type its constituent parts have (if any)

A computer program can be compared to a loaded pistol.

To know whether a pistol is defective, we could just shoot it, and see whether or not it blows up. This is analogous to detecting an error at **runtime**, when actually running a given program.

Or, we could be a little more thorough about it, and inspect the pistol for damage, or if someone put .50 AE into a 9mm pistol[6].

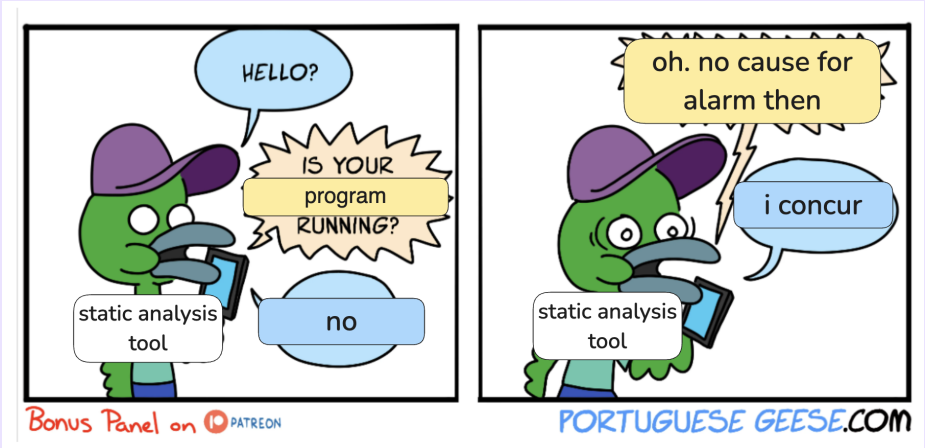
Obviously, the solution which ends up with the gun not exploding in our hands is the preferable one.

[6] I had to look this up. Basically, big ammo goes in small gun.

This is just another way of saying that we need a better method than just running into errors blindly, when running code in real environments. Discovering a bug in the Curiosity rover before it takes off is a miracle. Discovering a bug in the Curiosity rover *after* it takes off is a failure.

This is the basic principle behind **shift left**, which means evaluating our programs by some criteria as far *before* runtime as possible, or farther left, chronologically.

This will motivate today's discussion of static analysis specifically, as a better way of ensuring that we find mistakes as *far left as possible*.


HELLO?
IS YOUR program RUNNING?
static analysis tool
no
Bonus Panel on PATREON
oh. no cause for alarm then
i concur
static analysis tool
PORTUGUESE GEESE.COM

This is the mission we have ahead of us. Before we can dive into more technical details, however, we have one small issue before us:

Program analysis is inherently impossible.

Rice's Theorem is a mathematical theorem in computability theory[7] which states:

> *All non-trivial semantic properties of programs are undecidable.*

In English:

> *It is impossible to definitively answer yes or no for any property of a program's behavior, in a finite amount of time.*

[7] The field of computability theory is kind of similar to forum posts speculating on what would happen in a fight between Batman and Goku. It's just nerds getting together and thinking about increasingly more complicated levels of impossibility.

This is a corollary of the Halting Problem, which states that it is impossible to write a program to tell if a program loops forever or not.[8] The reason why this is impossible come out of asking a simple question: what should the following function return?

```
def foo():
  if halts(foo):
    loop()
  else:
    return
```

This ends up producing a paradox, in much the same way as the liar's paradox.

Because any program can loop forever, this ends up tainting every other question that program analysis could answer, meaning that all of them are inherently impossible.

[8] Note that these claims of impossibility are *in general*. For instance, I can look at the program `while True: pass` with my eyes and tell you that it loops forever, but we cannot write a program which does that for *every* program.

Well, that sucks. So then, with this knowledge, is this talk over?

No, because **it only sucks if you're a quitter.**

Generally, when solving a hard problem, it's usually either a sign that we are not working hard enough, or our expectations are too high.

Well, in this case, this problem's impossibility is a mathematical truth, so it's not a skill issue in terms of our ability to implement. That's not the problem here.

So let's lower our expectations.

> *It is impossible to definitively answer yes or no for any property of a program's behavior, in a finite amount of time.*

The main innovation out of program analysis is – *it's only impossible if you insist on being right all the time*.

Generally, we want our tools to be trustworthy. The day that a pacemaker or ladder stops working is the day that someone gets hurt. Nobody wants to use something which only works *some of the time*.

So, for most products, they must work *all the time*. To do otherwise would be sure way to head straight for bankruptcy in the free marketplace.

Program analysis suffers from no such thing. Our goal will be to implement analyses which always complete within a finite amount of time, albeit with the caveat that sometimes they might be inaccurate or incomplete, or throw their hands up and say "I don't know".

A funny corollary of this sentiment is something called the full-employment theorem, which essentially states that there will always be jobs in program analysis and compiler-writing, i.e. employment is always ensured[9].

This is because, due to the fact that the task is inherently impossible, it's always possible to write a better analysis that covers more cases, or a compiler which can produce better binaries. You just need more casework.

[9] Knock on wood.

For instance, no one is stopping you from doing this:

```
def halts(program: str) -> Optional[bool]:
  if program == "x = 1":
    return true
  elif program == "x = 2":
    return true
  elif program == "while True:\n  pass":
    return false
```

Far from the cutting edge, but it works. A team of monkeys at typewriters could eventually churn out a more effective `halts` function than exists anywhere else.

You might find this demoralizing. I find it motivating, somehow.

So, this is the scope of the task in front of us.

We have finite resources and finite time to solve a problem which is impossible to solve.

And still, software is eating the world. The cost of failing is too high.

Time to put in some elbow grease and get to work.

3 - Implementing the Impossible

Consider a more specific example of program analysis, namely that of type-checking a program.

Consider a function `f` which adds two ints together. We might say that it has the **type** of `(int, int) -> int` – that is, it takes in two integers and produces an integer. Automatically determining this is the problem of type-checking.

Def A **type** is a thing that something might be.[10]

Let's consider the problem of giving a type to `//`, which takes two integers and computes their integer division.

[10] This is only slightly a joke, I saw this on Twitter once.

We might say that it also has type `(int, int) -> int`, because it seems to take two `int`s and return an integer.

There is one caveat, however. If we pass in `0` as the second parameter, it will crash on us!

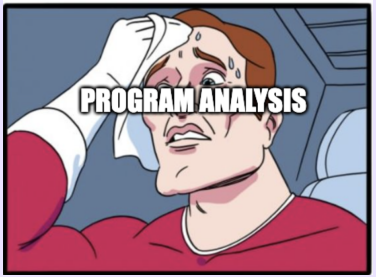
Well, this is a dynamic error. We want to catch such things before running the program. Can we assign `//` a type, such that it will prevent us from ever running code which passes `0` to `//`?

There are two properties that we would desire of such a type:

- we can ascertain it in a finite amount of time
- it says that a usage of `//` is ill-typed if and only if we pass `0`, or a non-`int`, to it

It is impossible to have both of these things at the same time.


looping forever
being wrong


PROGRAM ANALYSIS

There is a class of languages which have very sophisticated type systems, called **dependently typed** languages, where you can actually express this.

The issue is that type-checking in such languages can take forever, meaning we would get the second property, but not the first. We will call this a **Type A program analysis**.

So the other track will be to get the first property, but not the second. We will have to accept being wrong sometimes, and either rejecting programs which do not divide by zero, or accepting programs which do. We will call this **Type B program analysis**.

Consider the following analogy.

You are in charge of security at an airport.

You are acutely aware of the fact that in the early 2000s, a man tried to set off plastic explosives concealed in his shoes, during a transatlantic flight from France to Florida.

The issue is that you don't have a good way of telling whether an arbitrary individual might have explosives in their shoes, or not.

So, how do you minimize the chances?

This is the story of why everyone needs to take off their shoes in the airport.

The moral of this story is that, while it may be difficult or impossible to gather an *exact* answer (who has explosives in their shoes), it's easy to obtain an *approximative* answer: assume that *everyone* has explosives in their shoes.

So just scan everybody's shoes. Problem solved.

So how can we tell which programs contain an unsafe call to `//`? Well, if we don't mind being wrong sometimes...

```
def containsUnsafeDivision(program: str) -> bool:
  return "//" in program
```

Let's see how it behaves:

- `containsUnsafeDivision("x = 2")` returns `False`. ✓
- `containsUnsafeDivision("1 // 0")` returns `True`. ✓
- `containsUnsafeDivision("1 // 1")` returns `True`. ✗
- `containsUnsafeDivision("# https://google.com")` returns `True`. ✗

We see that it works... sometimes.

But this is precisely what a Type B program analysis purports to do. We accept that we might be wrong, sometimes, in exchange for the fact that this test always completes in short order.

Which outcome is more preferred? Well, it turns out the answer is "neither of them".

We are basically saying that, to be able to reject all programs which might divide by zero, we can either accept infinitely looping compile times, or we can reject every program which contains the characters `div` in its source text.

Now, with more sophisticated techniques, we can do a little bit better than rejecting every program containing `div`. But not by much.

It turns out, in practice, the right solution will be to simply not care so much about the division by zero case. It's not worth the trade-offs.

We said it was impossible to have the virtues of Type A and Type B analysis at the same time. That's true, if we fix the problem statement. We might say that **Type C program analysis** is to both terminate and be correct, but at the cost of simplifying the problem we are trying to solve.

Type-checking is usually an example of a Type C analysis. So, thus we end up with not being able to statically catch division by zero errors. For the question of "does this program divide by zero?", we decided the answer is "we don't care".

What about the question of "does this program divide by a non-integer"?

It turns out, this is perfectly solvable in a terminating manner.[11] The reason why this is OK is that "non-integer" is an approximative query – "zero" is specific.

[11] For the purposes of this presentation, it's not actually important this problem is solvable. Just know there are some program analysis problems which can be solved without looping or approximating (but few).

So let's recap for a second:

- we would like to answer specific questions about programs, which are impossible to do in general.
- We need to **give up one** of guaranteed termination, perfect accuracy, or solving that exact problem. Types A, B, and C analysis correspond to giving up each of these things, respectively.

Specific examples include:

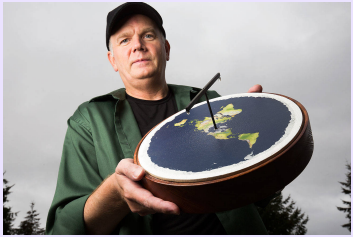
- **dependent typechecking** is a Type A analysis (can loop forever)
- **rejecting all programs with** `div` is a Type B analysis (rejects valid programs)
- **regular typechecking** is a Type C analysis (give up catching divide by zero)

For most practical program analysis tools, looping forever is not an option.[12] So for our purposes, we are generally interested in Type B and Type C analysis.

[12] We might call this "doubly impossible".

An easy way to remember program analysis is to envision it as three different kinds of people.

Type A analysis is personified by an old person (they might take forever to answer)



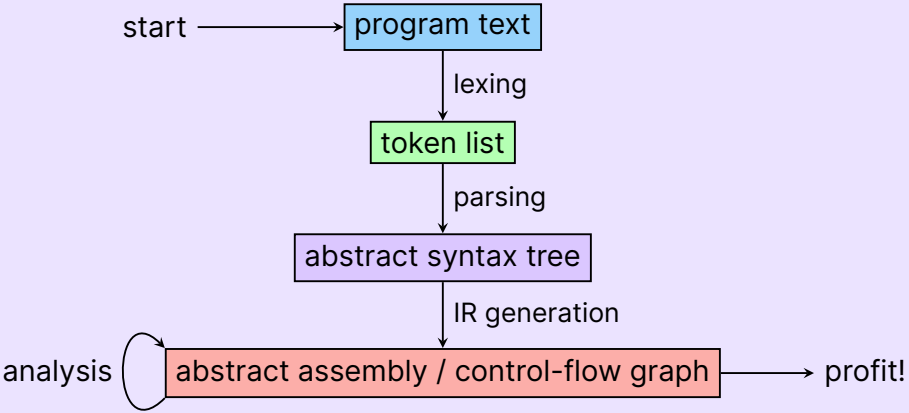
Type B analysis is personified by a flat earther (they sometimes spout nonsense)

Type C analysis is personified by a literal child (they can only answer simple questions)

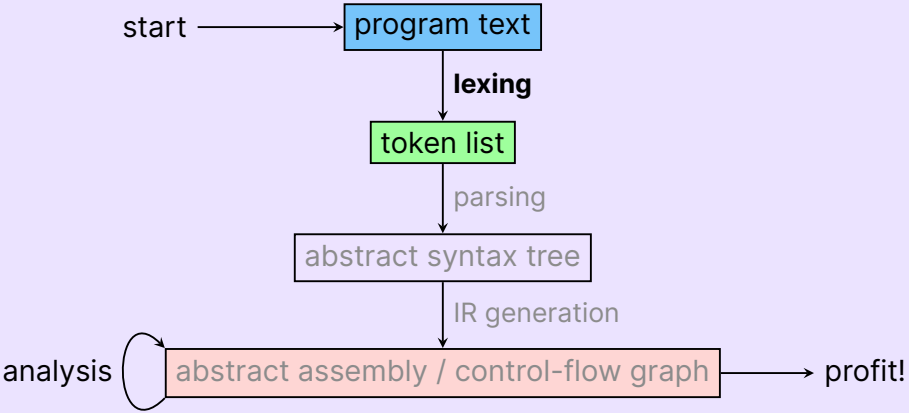
The lifecycle of any program analysis tool begins as many things do–not with a bang, but with a text file.

The typical stages of program analysis are as follows:

- **lexing**, which takes in a program as a string, and outputs a `token list`, which simply groups together fundamental units of the program
- **parsing**, which takes in a `token list`, and outputs an **abstract syntax tree** of type `ast`, which is a tree representing the program's structure
- **intermediate representation**, which turns the abstract syntax tree into **abstract assembly**, that breaks apart the high-level constructs into assembly-like primitives
- **analysis**, which may involve a variety of techniques, such as dataflow analysis, pointer analysis, and symbolic execution


start
program text
lexing
token list
parsing
abstract syntax tree
IR generation
analysis
abstract assembly / control-flow graph
profit!

4 - Stages of Program Analysis: Lexing


start
program text
lexing
token list
parsing
abstract syntax tree
IR generation
analysis
abstract assembly / control-flow graph
profit!

Let's visually see how we can think the process of lexing. We'll start with an example Python program, represented as a string.

```
x = 2 - 1

def foo(y):
  return 5 + x
```

We then **tokenize** the input program, so that instead of thinking of it as a list of characters, we group together all characters that are part of the same semantic unit.

Linguistically, tokenization is similar to reading sentences as words, instead of as a list of letters. Let's highlight all the "words" of this program.

```
x = 2 - 1

def foo (y):
  return 5 + x
```

During the process of lexing, we would have some way of representing this data as a list of tokens. We would need to know that they are distinct, and be able to associate data to each token when needed (for instance, an integer token would need to contain the information of the integer it represents).

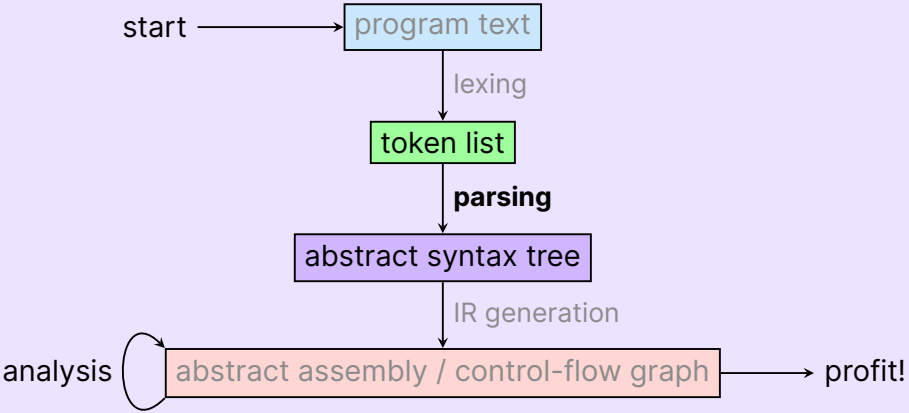
This is typically achieved via a **tagged union**, effectively an **enum** with data.

So after tokenization we might get something like this:

```
x     =     2     -     1
ID    EQ    NUM   MINUS NUM

def   foo   (     y     )     :
DEF   ID    LPAREN ID   RPAREN COLON

return 5    +     x
RETURN NUM  PLUS  ID
```

Caveat Python is a whitespace-sensitive language, so technically this is a slightly simplified account of the information you would need to produce to parse correctly. We have elided the details here for simplicity, but the idea is the same.

5 - Stages of Program Analysis: Parsing


start
program text
lexing
token list
parsing
abstract syntax tree
IR generation
analysis
abstract assembly / control-flow graph
profit!

One of the most fundamental scientific discoveries of the 19th and 20th centuries was concerning the nature of light.

Through a series of experiments, the wave-particle duality of light was discovered, which demonstrated that light displayed properties of both waves and particles, depending on the context. This meant that light could be thought of as *both* a particle and a wave.

Programs display a similar behavior, in that they experience a similar dual existence. **Programs can be thought of as both text and as trees.**

Before we can discuss the tree nature of programs, we should talk a little bit about the pros and cons of thinking of programs as text.

Pros:

1. easy to edit in an application-independent way
2. easy to search and refactor at scale
3. simple data format

Cons:

1. difficult to analyze the semantics of
2. must be validated for syntactic correctness before use

For instance, suppose that we're interested in figuring out whether a program ever prints the value `0`. This might seem easy:

```
def hasZeroPrint(program: str) -> bool:
  return "print(0)" in program
```

This is a simple first cut. What's wrong?

The problem is that this is only a very rough solution. It wouldn't cover, for instance, the case where we assign a variable to `0` before printing it.

```
is_zero = 0
print(is_zero)
```

Nor would it deal with the case where we format our code slightly differently:

```
# now there's a newline before the 0
print(
  0
)
```

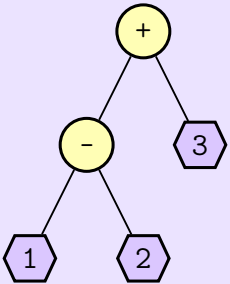
Text is ultimately a very poor representation of the structure of a program. It's usually nice for development purposes[13], but it ultimately doesn't do a great job of telling us what a program *means*.

To solve this problem, we're going to need to go hug some trees.

[13] And even this is under dispute.

To help understand this idea by analogy, consider the following example.

If you are familiar with the idea of **op trees**[14], recall that we can have a tree corresponding to some arithmetic expression:



This tree happens to denote the expression `(1 - 2) + 3`.

[14] You might be familiar with this as a classic homework assignment from undergrad.

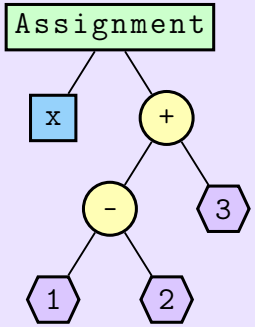
This is what we call **abstract syntax**, since it elides some of the specific syntactic details, like the fact that there is a left and right paren around the subtraction.

In the end, this doesn't matter, because the tree structure serves as a proxy for what the parentheses were trying to tell us. We thus can get away from the precise coding details, while preserving the meaning, by using an **abstract syntax tree**, or AST for short.

We can do something very similar to op trees with programs. We will instead have an **abstract syntax tree** which denotes the structure of the program.

This tree denotes the program

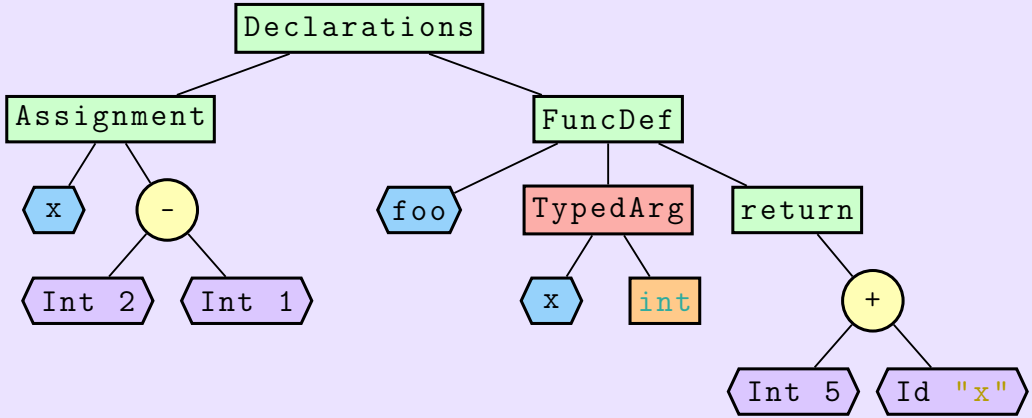
```
x = (1 - 2) + 3
```



Note how it has no mention of parens or the = sign, because they don't actually matter in terms of what the program *means*!

Generally, we can get rid of things like colons, equals signs, keywords, and parentheses in abstract syntax. These syntactic details only existed to let us know what the actual underlying tree looked like.

So, for our running example program, we could obtain the following abstract syntax tree:



What's the point?

When we parse programs into abstract syntax trees, the process of conducting analysis on the program becomes much easier. We can now ignore syntactic noise (how the program **looks**) and focus on the semantics (what the program **means**).

As a child, you were told not to judge a book by its cover. Abstract syntax trees are a technique that allows us to do just that.

Seen in such a way, our query for finding returns of `0` can be a lot simpler:

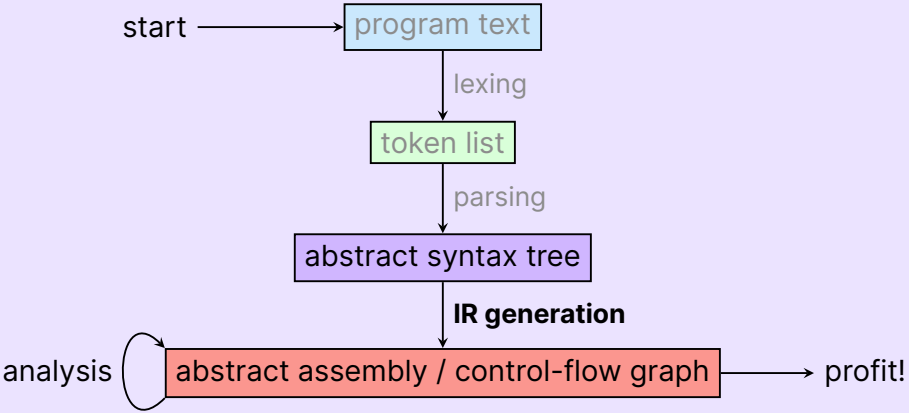
```
def hasZeroReturn(node: ast) -> bool:
  if node.kind == "Return" and len(node.children) == 1:
    child = node.children[0]
    return child.kind == "Int" and child.value == 0
  return any(hasZeroReturn(child) for child in node.children)
```

The eagle-eyed of you might have noticed an issue, however.

Caveat Our new `hasZeroReturn` function still does not deal with cases where we assign a variable to `0`.

While abstract syntax trees allow us to ignore some of the syntactic burdens of the text, they will not tell us the *entire* story of the program. We will still need to descend further, into another format, before we can answer more sophisticated questions that have to do with control-flow.

5 - Stages of Program Analysis: IR Generation


start
program text
lexing
token list
parsing
abstract syntax tree
IR generation
analysis
abstract assembly / control-flow graph
profit!

As stated before, we have some trouble with finding returns of `0` that make use of a previously-declared variable:

```
is_zero = 0
print(is_zero)
```

One might think that we could just iterate over statements in the program, take note of any variables which are assigned to `0`, and then proceed as usual.
This would work, but we might face an issue when we see statements which are not simple variable declarations. For instance:

```
if condition:
    is_zero = 0
else:
    is_zero = 1 # haha tricked you
print(is_zero)
```

The problem is that, statically, `condition` could be a condition which is fully dynamic in nature. We might not be able to determine which branch of the `if` statement is taken, or how many times it runs.

This runs us straight into the Halting Problem, as before. What can we do?

Recall our previous discussion on Type C analyses. In the same spirit, we can downscope and give up on the idea of perfectly knowing which condition is taken.

Moral For most software, being wrong is a hellish scenario that is too horrifying to even contemplate. For program analysis tools, being wrong is Tuesday.

We instead distill our query into two possible questions:

Question Is it **possible** for a given return to return `0`?

Question Is it **always** the case that a given return will return `0`?

Before we can describe the technique that we will use to solve these specific questions, let's look at a few other problems we are interested in answering with program analysis, which will turn out to be related.

There are many, but a few that stand out are as follows:

- is it possible for data from some source to reach some function?
- is an error state reachable?

The problem of **taint tracking** entails whether it is possible for data from some source to reach a specified function.

```
def foo():
  x = input()
  y = x.strip()
  eval(y)
```

In this code example, we see that data from the call to `input()` goes through a `strip()`, which removes whitespace, and then reaches the call to the sensitive `eval()` function.

In this case, we find that even though `input()` does not go *directly* to `eval()`, it's still a security risk!

The problem of **code reachability** has to do with determining whether or not there is a given series of inputs that can cause some code path to execute.

```
def foo(person):
  if isChappellRoanFan(person):
    f()
  elif isLame(person):
    g()
  else:
    throw new Error("bad")
```

In this case, we are interested in whether the `Error` can be reached. It's not clear, but a sophisticated program analysis tool could know that not being a Chappell Roan fan implies being a lame person, so the last condition is not actually reachable.

The commonality in all of these problems is that they are all dependent upon the **control flow** of a program.

Def The **control flow** of a program is the particular order in which it executes its instructions.

Because of the Halting problem, control flow is very hard to know[15]! In a similar vein to how we decided to solve the control-flow question of zero returns, we will restrict ourselves to two questions:

- is it **possible** for a program to exhibit a certain control-flow behavior
- is it **always** the case that a program exhibits a certain control-flow behavior

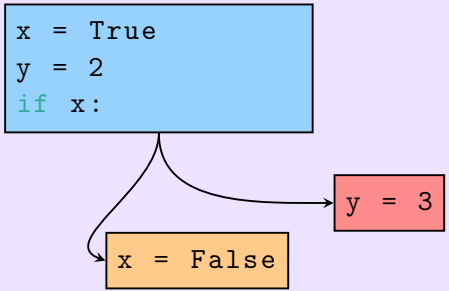
We call these the distinction between these two questions a **may** versus **must** analysis.

[15] Impossible, actually.

To facilitate our ability to answer these questions, we turn to a data structure called a **control-flow graph**, or CFG for short.

A **control-flow graph** is a kind of structure which basically describes all the paths that a program might take. When an instruction is *always* followed by another, then they appear in a single "block" together. When the instructions might *branch* (that is, go one place or another), then arrows point towards the blocks that it might lead to.

```
x = True
y = 2

if x:
   x = False
else:
   y = 3
```



Note that control-flow graphs can get pretty massive and complicated.

The important takeaway here is that **control-flow graphs only predict program behavior**, they do not dictate it. Because the arrows just show where a program *might* go, it doesn't magically solve the Halting Problem[16].

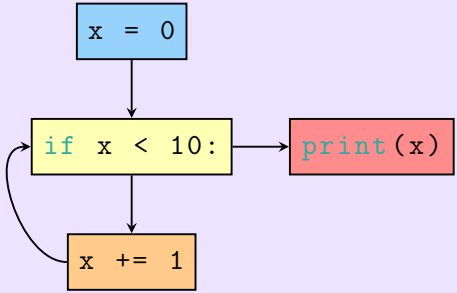
The second important takeaway is that **there is always a way to turn a given program to a control-flow graph**. Let's not worry about how that happens[17].

[16] Another way to think about it is that a control-flow graph describes a state space which a program may traverse over its execution. We cannot say for sure where it will go, but it must exist within the space of the graph.

[17] As they say, this is an exercise left to the reader.

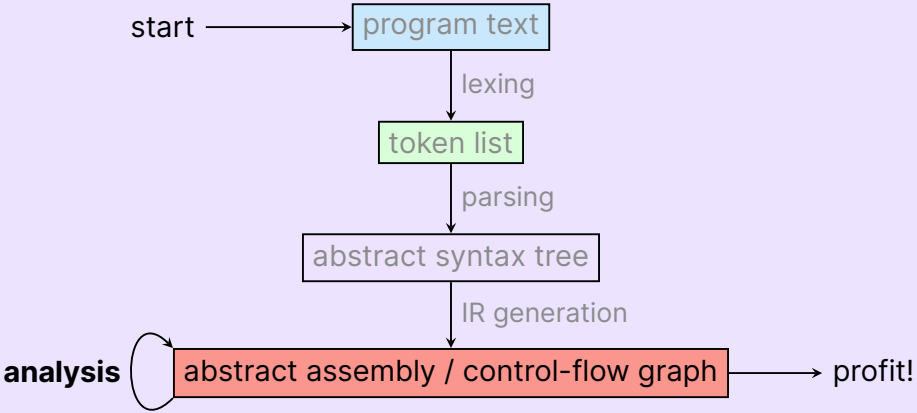
To make sure our foundations are clear, let's look at one last example of a control-flow graph.

```
x = 0

while x < 10:
    x += 1

print(x)
```



Notably, we reduce the complicated `while` construct into a an `if` statement, which leads to a node that points back to the `if` statement. This is a way of encoding the behavior of a `while` loop, which is to constantly re-calculate the condition until it is false.

6 - Stages of Program Analysis: Dataflow Analysis (bonus)


start
program text
lexing
token list
parsing
abstract syntax tree
IR generation
analysis
abstract assembly / control-flow graph
profit!

These problems all come with their own hardships. Due to what was discussed in the last section, they are all impossible to solve completely, but several of them are actually solvable via a Type B (allowed to be wrong) analysis, using the same method.

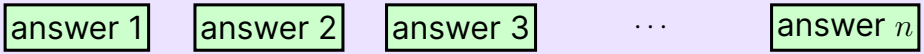
A classic technique used in program analysis to obtain *approximative* [18] answers in a finite amount of time is called **dataflow analysis**.

Before I can define it to you, I must give you an analogy.

[18] Life hack: you can successfully replace "incorrect" with "approximative" in so many different places that it's hilarious.

Suppose you have a query you would like to solve on programs, which has many possible answers. Further suppose that the number of possible answers is finite.

Suppose that you line them all up, one next to the other.



Program analysis is hard because information can change a lot, infinitely much in fact, over the course of a program's run-time. For instance, suppose that the question is "how much memory is being used by the program?". You might pick an answer n, then move to answer $n - 2$, then move to a different answer k altogether. It's possible to jump all around, in the limit of the program's execution.

An observation can be made that, if you can order your answer in a way such that, over the course of your analysis, you only ever change your answer by moving right, you will always eventually terminate.

This is a roundabout way of describing what is known as a **monotonic function**, which is a function which always "increases", according to some proper notion of "increases". In this case, our monotonic function also has an upper limit, i.e. a point beyond which it can no longer grow.

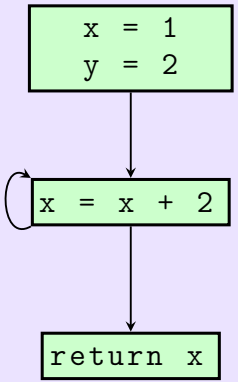
For dataflow analysis, we will make use of this kind of analysis to iterate over our control-flow graph, constantly updating our answer, but only in a way that "increases", and eventually caps out. If we can do that, then we will guarantee that we will terminate.

This also usually makes our answers sometimes wrong, though.

For instance, consider the following control-flow graph[13]:

We would like to perform an analysis known as **constant propagation** on it, by noting which variables are **constant**, i.e. never changed over the duration of the program.

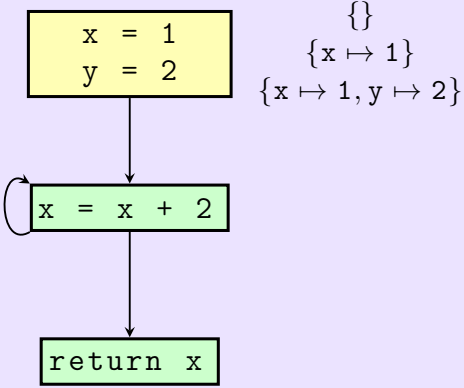
How do we do this? We simply march forward through the CFG, and noting down which variables are constant as we see them, starting with the empty set.



[13] Note that this one has a self-loop. This can happen, for example, from a `while` loop.

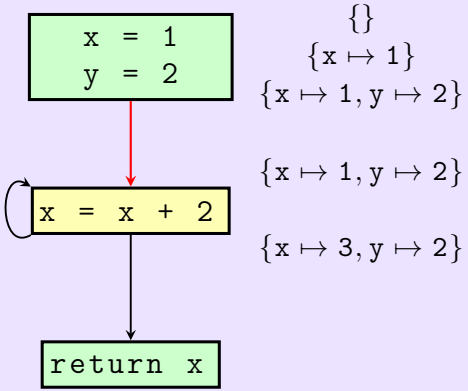
So for instance, first we traverse the entering block, by simply penciling in `x` and `y` as we see them get assigned to constants.

Once we finish, we now have the **out-set** for the first block, which we can then use to determine the other blocks.



After following the highlighted edge, we end up at the second block. Since we have some information about what variables are constant, we can carry that information here.

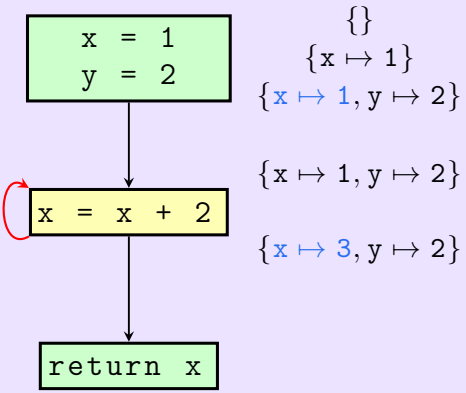
Then, we see that `x` is incremented by two, and thus must be constant at `3` at the conclusion of the block.



But, now we need to follow the self-loop! Something weird happens here.

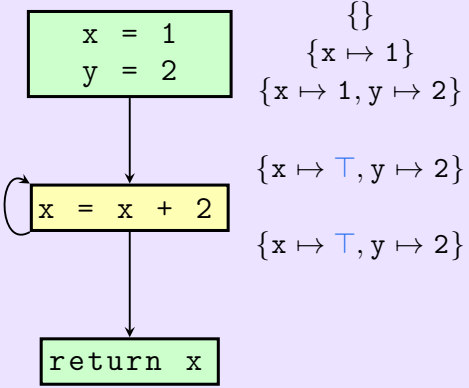
There are two conflicting out-sets that are going in to the second block. One is the one we just computed, $\{\texttt{x} \mapsto 3, \texttt{y} \mapsto 1\}$, from the output of the second block itself. The other is $\{\texttt{x} \mapsto 1, \texttt{y} \mapsto 1\}$, from the original out-set from the first block.

This means we have a conflict. `x` is constant, but at two different values, coming in to the second block.



This must mean that `x` is not constant after all.

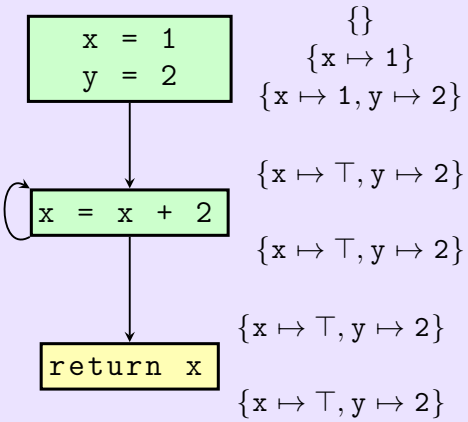
So we set the value of `x` to instead be $\top$, which means "not constant". Note that this is different than `x` not having a value, which denotes not knowing if it is constant or not.



Then, once we are assured that everything looks good, we can proceed to the final block, where we observe that we return `x` at a non-constant value.

This means that we cannot optimize the return value of this function after all.

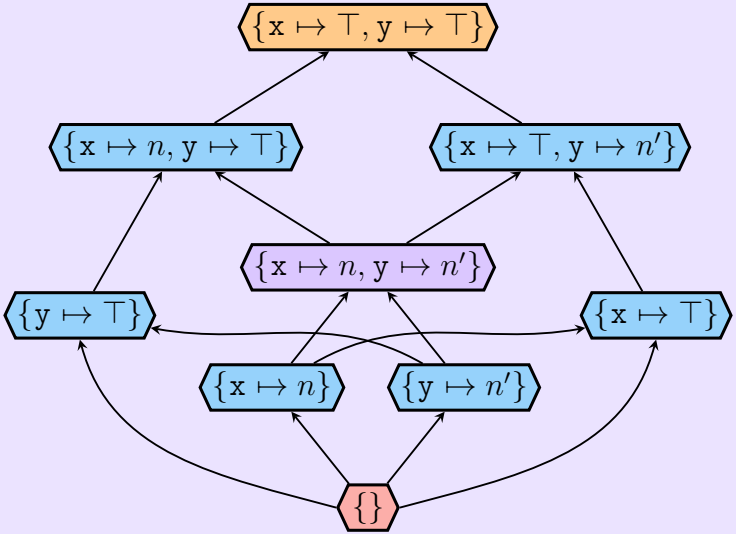
Different story if we had returned `y`!



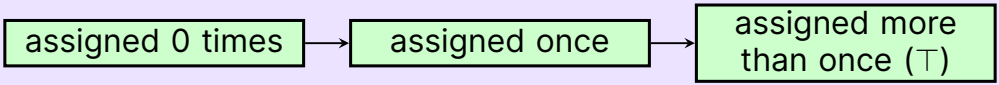
This is a contrived example, but the really interesting thing is that dataflow analysis works for *any* control-flow graph.

The process seemed somewhat silly, as we could determine with our eyes that `x` was non-constant, but for very complicated control-flow graphs this is not an obvious fact at all. Programmatically, we can still run this same analysis, however.

This analysis is also guaranteed to terminate, due to the monotonic reasons we stated earlier. The actual reason for this is that dataflow analysis strictly traverses up a **lattice**.


$\{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top\}$
$\{\mathtt{x} \mapsto n, \mathtt{y} \mapsto \top\}$
$\{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto n'\}$
$\{\mathtt{x} \mapsto n, \mathtt{y} \mapsto n'\}$
$\{\mathtt{y} \mapsto \top\}$
$\{\mathtt{x} \mapsto \top\}$
$\{\mathtt{x} \mapsto n\}$
$\{\mathtt{y} \mapsto n'\}$
$\{\}$

The diagram looks scary, but the key thing is just that it assigns each variable a value of either no value, any constant n or n', or $\top$, which means "not constant".



Edges go from sets to ones which have either added a new variable at a constant, or that have upgraded a variable from a constant to the not-a-constant symbol $\top$. This represents the gaining of information, of either a variable being declared as a constant, or a variable being discovered as not-a-constant.

- For instance, we started at $\{\}$, and then went to $\{\mathtt{x} \mapsto 1\}$ when we read `x = 1`.
- Or, we went from $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}$ to $\{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto 2\}$ upon seeing that `x` was set as two different constants, along two different paths to the block.

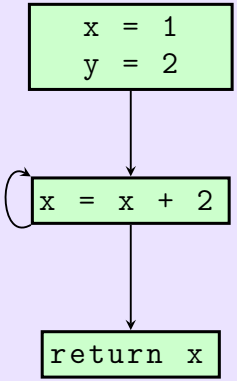
The main thing to take away here is that *every arrow goes up*. We talked earlier about putting answers on a number line, which is represented in terms of the lattice's height, here. No matter what edge you pick, you go up some amount, which means we must terminate.

We can't go down, because it's impossible to update your worldview to either remove a variable from the set, or to set a variable from not-constant to constant. Once you know something about a variable, you can't go back.

I mentioned earlier that this analysis is necessarily wrong, however.

The reason comes out of the fact that the control-flow graph is just how the execution of the program *might* go. In reality, it's quite possible that at runtime, we never enter the self-loop, meaning that `x` really is constant.

But, without running the program, we have no way of knowing, so we assume that `x` *is* updated at some point. This makes our knowledge necessarily possibly wrong, but a good approximation.



This was a really simple example that I hoped you might be able to understand.

Dataflow analysis in general is a very powerful technique, however, and admits many other analyses, many of which are quite useful. These include:

- **available expressions** - is there a definition of this exact expression already at this program point? useful in optimizing away redundant computations.
- **reaching definitions** - what definitions of a variable can reach a given program point? useful in building use-def chains (i.e. "goto definition" in IDEs)
- **taint tracking** - can data from undesirable sources reach some sensitive program point? **very** useful in security applications.

The rabbit hole goes deep. This one is simple, but this is a bread-and-butter technique in program analysis.

Dataflow analysis is a foundational technique that forms the bread and butter of the program analysis toolbox. Despite being a technique which has been used for decades, it is still relevant today. All static analysis tools do some form of dataflow analysis.

In a sense, though, it can be viewed as a "lower level" analysis. We elided a lot of details in this presentation, but a proper dataflow analysis often entails breaking down a program into its most base representation. This tends to lose a lot of the original structure of the code – nobody writes a CFG when programming.

7 - Extra Considerations

By this point, we have seen much on the implementation of SAST tools, in the tradeoffs that have to be made in order to solve an inherently impossible problem.

When it comes to appraising the fit of a SAST tool, there come considerations which are not strictly captured by the program analysis itself.

There are considerations of speed, developer friction, ease of use, and convenience, as well.

Speed is the name of the game when it comes to code scanning, oftentimes.

To truly enable a shift left perspective on security, it is often necessary to meet the developer where they are, such as in their IDE, or in their CI/CD pipeline.

To that end, a SAST tool must be able to scan code quickly, to minimize time spent blocking developers.

Technically, this can be something of a challenge, as analyzing a project can have as bad as quadratic or cubic time complexity in the size of the input program.

A standard way to mitigate this is to do **partial analysis**[19], which entails only scanning relevant files which have been changed, in a given pull request.

Unfortunately, this has potential drawbacks. Consider the following simple program:

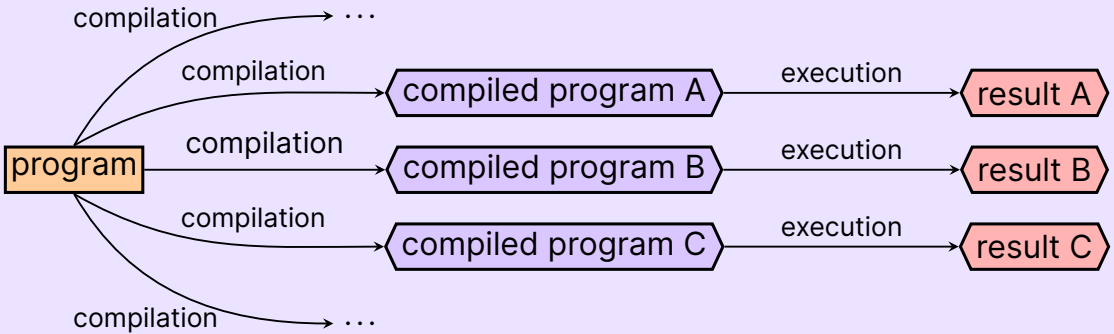
```
def foo(x):
  return sink(x)

def bar(x):
  return safe(x)
```

```
def qux():
-   return bar(input())
+   return foo(input())
```

If these code blocks are in different files, we wouldn't know that the change to `qux` is potentially dangerous unless we also scanned the first file.

[19] I made this precise term up.

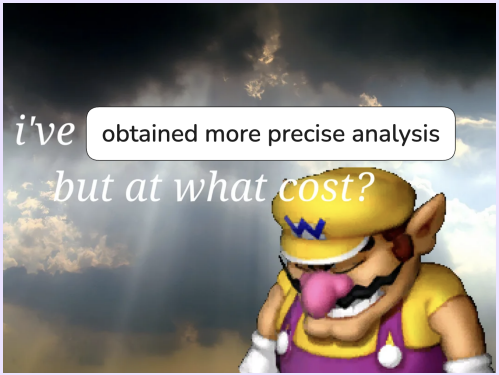
Another important consideration is in ease of rollout.
Program analysis, as an approximative art, is always looking for ways to obtain more correct results.
In terms of code scanning, means that one need not restrict themselves to just source code analysis, but also taking into consideration the precise build context.
This is best surmised by the following diagram:



In terms of correctness, build context is uncontroversially more correct.

The difficulty comes with set-up. Knowing the precise build context requires project-specific tuning, and often requires a more complex infrastructure.

When rolling out a scanner to a large organization, potentially hundreds or thousands of repositories, this can be a significant barrier to entry.



7 - Conclusions

Part of growing up is learning that your guardians are not perfect.

Similarly, part of growing up as a security professional is learning that your tools, too, are not perfect. Throughout this lecture, we have seen that not only are they imperfect, they are fundamentally flawed.

This is not meant to inspire despair, however–quite the opposite. By understanding more the limitations of our tools, we can better understand when they are useful, and when they are not.

Security is a war, and in the words of the famous G.I. Joe, "knowing is half the battle".

I hope that this presentation has instilled a little bit more knowledge in you, and a confidence in your ability to navigate the world of application security, SAST scanning, and static analysis.

The volume of code being written is only increasing, and the need for better guarantees about its security, for scaling up our ability to analyze code, and for enhancing our knowledge of the code that we run and deploy each day is only increasing.

Today's world of application security is not going to be the same as tomorrow's. Tomorrow's advances in generative AI, formal methods, and developer tooling are promising, but hopefully this presentation can help to keep track of how they can apply to today.

Here's to making static analysis **fast**, **fruitful**, and **frictionless**.

Thank you!