Lesson 21
PROGRAM ANALYSIS

August 3, 2023

Last time

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.

Part of what makes software engineering a lucrative profession is that there is not, and will never be, a shortage for software engineers.

Regardless of if a company is a recruiting company, a think tank, a massage parlor, or a pet food retailer, everyone needs software developers. Every business needs a website, every business deals with data, and every business needs a way to keep up with every other business, which is doing exactly the same.

Unfortunately, not all of them are educated at Carnegie Mellon, and have taken 15-150, so not all of them are very well-informed on the importance of writing safe code.

One theme that has cropped up throughout this course 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:

✗

```
fun fact 0 = 1
  | fact n = n * fac (n - 1)
```

✓

```
fun fact 0 = 1
  | fact n = n * fact (n - 1)
```

Or maybe you declared a variable, and then forgot to use it:

✗

```
fun treefoldl f acc Empty = acc
  | treefoldl f acc (Node (L, x, R)) =
      let
        val left_folded = treefoldl f acc L
      in
        treefoldl f (f (x, acc)) R
      end
```

✓

```
fun treefoldl f acc Empty = acc
  | treefoldl f acc (Node (L, x, R)) =
      let
        val left_folded = treefoldl f acc L
      in
        treefoldl f (f (x, left_folded)) R
      end
```

Or maybe you just have a simple type error:

✗

```
fun foldr f acc [] = acc
  | foldr f acc (x::xs) =
      f (x, foldr acc xs)
```

✓

```
fun foldr f acc [] = acc
  | foldr f acc (x::xs) =
      f (x, foldr f acc xs)
```

These kinds of simple mistakes crop up all the time!

Thankfully, we are working in a language which is disciplined enough to warn you about most of these things, albeit not all.[2]

What about all the software being produced elsewhere, though? In some languages, such as Python, **none of these errors** are able to be caught, until they happen at runtime!

We say that making the programmer aware of these errors at **compile time**, before the program runs, is a **static** warning, versus a **dynamic** warning, which would only occur once the program runs into it while executing.

[2] Though it could. SML/NJ in particular just doesn't.

I love telling this story whenever anyone asks about why it is important to catch errors statically.

Imagine that you are a machine learning engineer.[3]

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.

[3] The horror.

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. We need to build tools, compilers, and programming languages that can make sure that these errors do not make it to real applications, because the cost of doing so is too high.

Remember Tony Hoare's billion-dollar mistake. 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.

How can we fix these kinds of mistakes? 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]

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

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**.

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 crashes, then dynamic program analysis will do the same.

More concerning is that dynamic program analysis will generally have computational cost equal to that of the program being analyzed. This means that a program which takes a very long time to run will take a very long time to test. We don't always have time for that.

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

Because as we saw in the last lecture, programs are trees, this means that static program analysis really just takes the form of recursive functions on trees.

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)

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 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.*

This is a straightforward corollary of the Halting Problem, which essentially just states that it is impossible to write a program to tell if a program loops forever or not.[5] The reason why this is impossible come out of asking a simple question: what should the following function return?

```
fun not_halts () =
  if halts(not_halts) then loop () else ()
```

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 possible question that program analysis could answer, meaning that all of them are inherently impossible.

Well, that sucks.

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

So then, with this knowledge, is this lecture over?

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

Recall our idea of **strengthening the implementation** or **weakening the specification**, for solving some problem, meaning that we can either put some elbow grease in and make our program more powerful, or lower our expectations.

Well, this 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*.

Compilers and program analysis are dual in a certain sort of sense, in that a compiler is itself a kind of program analysis. It follows much the same process, by analyzing the source text and producing answers, albeit while also producing an executable file.

The main thing to realize is that **a compiler can never be wrong**. As stated in the previous lecture, the day that our compilers are untrustworthy is the day that programming becomes impossible.

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.

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.

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

```
fun halts (program : string) : bool option =
  case program of
    "val x = 1" => SOME true
  | "val x = 2" => SOME true
  | "val x = 3" => SOME true
  | "fun loop x = loop x val _ = loop ()" => SOME false
  (* add more cases here! *)
  | _ => NONE
```

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.

This might seem demoralizing, but this is somehow actually a motivational statement that I will always have a job.

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 - Doing the Impossible

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

Now, type-checking always terminates, and it always returns a correct answer. If the type-checker says an expression has some type, then we can trust that answer, and if it says our program is ill-typed, then indeed we messed up somewhere.

However, consider the problem of typing the following function:

```
val div : int * int -> int
```

For the `div` function, we assign it a type such that it can take in any two integers. This seems reasonable, with the caveat that if we pass it `0` in the second parameter, it will crash on us!

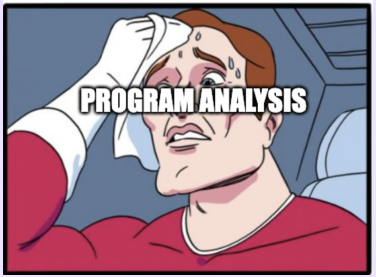
Well, this is a dynamic error. We want to catch such things before running the program. Can we assign `div` a type, such that it will prevent us from ever running code which passes `0` to `div`?

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 `div` 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 a more sophisticated type system than Standard ML, called **dependently typed** languages, where you can actually express this.

The issue is that type-checking in such languages can loop forever, meaning we would get the second property, but not the first. We will call this **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 parable 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 `"div"`? Well, if we don't mind being wrong sometimes...

```
fun containsUnsafeDiv (prog : string) =
  stringContains (prog, "div")
```

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 a `div`.

Now, with more sophisticated techniques, we can do a little bit better than rejecting every program containing a `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. The reason why this is OK is that "non-integer" is an approximative query – "zero" is specific.

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.[6] So for our purposes, we are generally interested in Type B and Type C analysis.

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

But, we still have significant problems left to answer.

Type-checking is only in the Type C category because of decades of work by type theorists and language designers, to figure out what buckets of questions can be answered tractably by machines, and to what extent.[7]

For many other problems, such as code reachability, vulnerability to outside attackers, and unsafe behavior, we have no such guarantees. We fall squarely into Type B, Type A is not an option.

So we need to accept being wrong sometimes. Let's see how.

[7] Though somehow, many modern languages fall decades behind still in that respect

4 - Dataflow Analysis

A classic technique used in program analysis to obtain *approximative* [8] answers in a finite amount of time is called **dataflow analysis**.

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

[8] 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.

$$ans_1, ans_2 ... ans_n$$

Program analysis is hard because information can change a lot, infinitely much in fact, over the course of a program's run-time. 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.

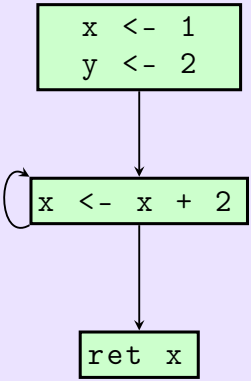
For dataflow analysis, we will make use of this kind of analysis to iterate over our control-flow graph, constantly updating our answer, but only in a way that "increases", and eventually caps out. If we can do that, then we will guarantee that we will terminate.

This also usually makes our answers sometimes wrong, though.

For instance, consider the following control-flow graph:

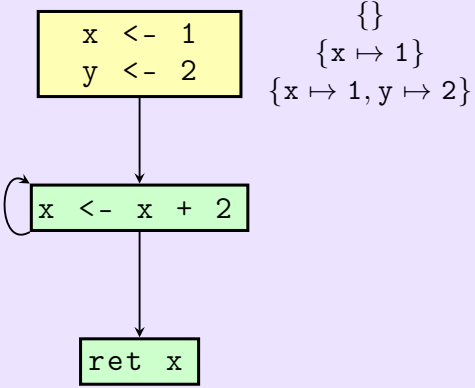
We would like to perform an analysis known as **constant propagation** on it, by noting which variables are set to be constant.

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.



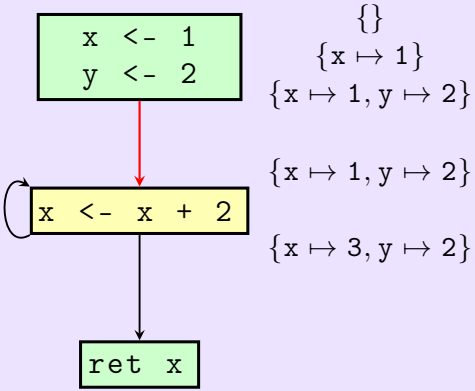
So for instance, first we traverse the entering block, by simply penciling in `x` and `y` as we see them get assigned to constants.

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.

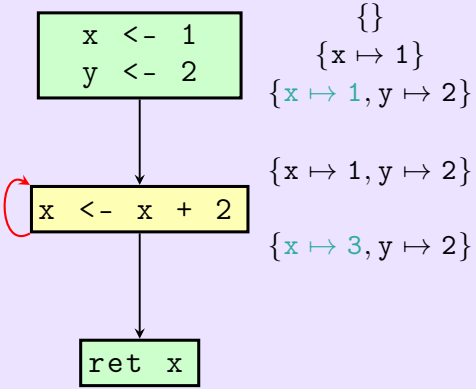
Then, we see that `x` is incremented by two, and thus must be constant at `3` at the conclusion of the block.



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

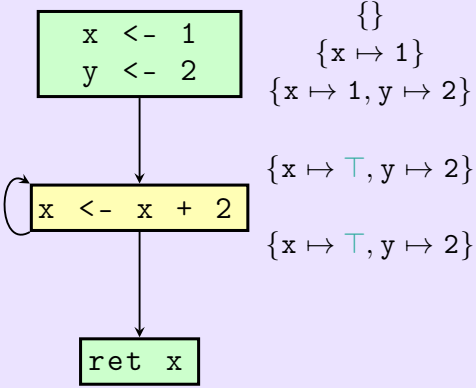
There are two conflicting out-sets that are going in to the second block. One is the one we just computed, $\{\texttt{x} \mapsto 3, \texttt{y} \mapsto 1\}$, from the output of the second block itself. The other is $\{\texttt{x} \mapsto 1, \texttt{y} \mapsto 1\}$, from the original out-set from the first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

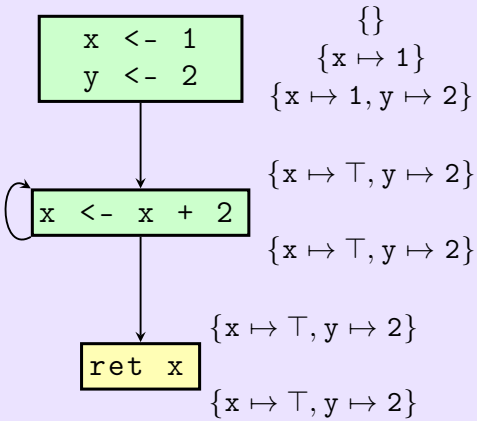
So we set the value of `x` to instead be $\top$, which means "not constant". Note that this is different than `x` not having a value, which denotes not knowing if it is constant or not.



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.

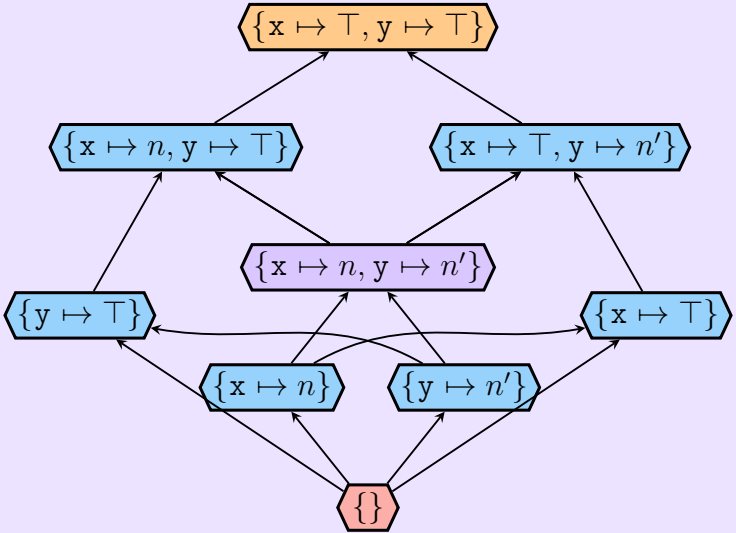
Different story if we had returned `y`!



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**.


{x ↦ ⊤, y ↦ ⊤}
{x ↦ n, y ↦ ⊤}
{x ↦ ⊤, y ↦ n′}
{x ↦ n, y ↦ n′}
{y ↦ ⊤}
{x ↦ ⊤}
{x ↦ n}
{y ↦ 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 $\{\texttt{x} \mapsto 1\}$ when we read `x <- 1`.
- Or, we went from $\{\texttt{x} \mapsto 1, \texttt{y} \mapsto 2\}$ to $\{\texttt{x} \mapsto \top, \texttt{y} \mapsto 2\}$ upon seeing that `x` was set as two different constants, along two different paths to the block.

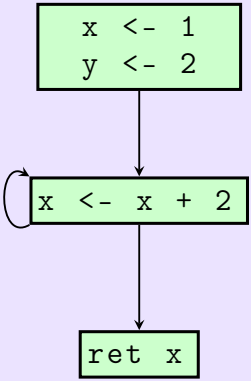
The main thing to take away here is that *every arrow goes up*. We talked earlier about putting answers on a number line, which is represented in terms of the lattice's height, here. No matter what edge you pick, you go up some amount, which means we must terminate.

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)
- **liveness analysis** - what variables might actually be needed in the future? useful in eliminating dead code.
- **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, unfortunately, has almost nothing to do with functional programming.

I've been avoiding showing you code for this part because it's both pretty complex, and there aren't extreme benefits to doing it in SML.

Dataflow analysis can be considered a more "imperative" approach to program analysis, because it occurs at the lower level of the control-flow graph and the abstract assembly instructions. What if we want to stay at the level of the AST, where all the nice semantics and program structure live?

It turns out, this will be a very effective approach to program analysis.

This is the story of Semgrep.

5 - Semgrep

Before we delve deeper, it's worth developing: why should we stay at the AST level, as opposed to abstract assembly? Isn't that the opposite of what a compiler does? Shouldn't closer to the source be better?

Before I can answer that question, it's worth noting some things about security and software engineering.

In the absence of external pressure, these things do not generally go together.

Software engineers are paid to write code, are judged on their code, and spend large amounts of time thinking about their code. What they are not necessarily, however, is educated in the ways of safety. So this tends to take a back seat.

For a compiler, being wrong is a hellish scenario that is too horrifying to even contemplate.

For a program analysis tool, being wrong is Tuesday.

Still, to the developers who are receiving security notifications on their pull requests, as well as the security engineers that are constantly on the lookout for big vulnerabilities, it matters **how wrong** the tool usually is.

The easier way to make change is not to preach at someone the right way to do things, but to make them want to do things the right way. A SAST tool needs to be fruitful, but frictionless.

Why am I mentioning this?

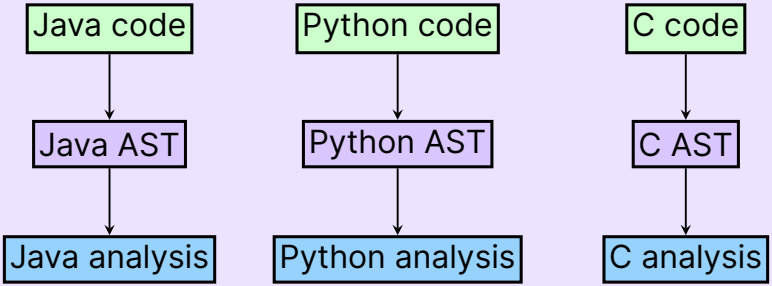
We want to be able to provide as little friction as possible, while producing the best possible results. In the case of Semgrep, **staying primarily at the language level is an enormous labor saving device**.

We talked earlier in the compilers lecture about how a compiler will have a type `token list` and a type `ast`, which it converts from and to during parsing.

For a program analysis tool for Python, they may need to do this parsing process from a token list to a Python AST. For a program analysis tool for C, they may need to do the same for a C AST, which is a different type.

This can lead to massive duplication of efforts.

So the picture looks like this:



This introduces a massive amount of boilerplate.

For as much as people complain about programming languages[9], after some time with them you begin to realize that most of them look quite similar.

For instance, many languages have for loops, and many languages have exception raising and handling, and almost every language has functions, ways to define variables, and ways to call functions.

Now, every language has its own idiosyncrasies which set them apart in miniscule ways, which makes writing something like a universal compiler a pipe dream. But what if we're working in an application where being wrong in small ways doesn't really matter?

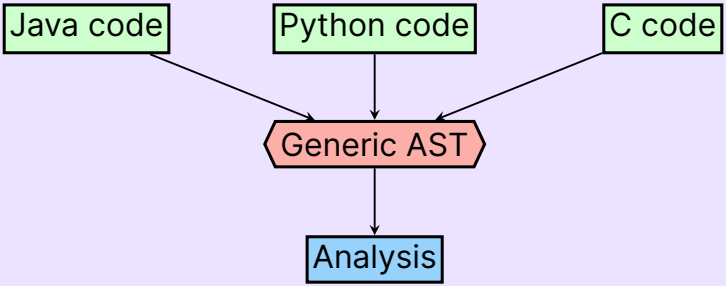
For program analysis tools, being wrong is Tuesday.

[9] As have I.

Semgrep is a code-scanning tool for over 24 languages.

The way that it achieves this is that it parses every single language to the same type, called the Generic AST, which is a smörgåsbord union of all the language features in pretty much every language.

This means that its pipeline looks remarkably simpler:



We see that staying at the AST level saves us work, which can be wrong at times due to imperfect translation, but ultimately doesn't matter for a program analysis tool. More importantly, it allows for cleaner code to be written, which translates to a better tool.

When solving an impossible problem, you have to be pragmatic. Picking battles that you can win is essential.

Semgrep takes a more "syntactic" approach to program analysis that turns out to work very well in practice. Now, let's talk about what it does.

Many program errors aren't purely due to what happens at the level of assembly. Many of them are apparent from the source code, because they were written by a developer who was writing source code.

For instance, consider the following Python code, with a function that has a default argument `to` set to `[]`:

```
def append_to(element, to=[]):
  to.append(element)
  return to
```

What do you think this does?

It turns out that the first time you call `append_to(2)`, you will get `[2]`. That's cool.

The second time you call `append_to(2)`, you will get `[2, 2]`.

Uh oh.

It turns out this is due to the fact that Python default arguments are instantiated at function definition time, and so if it is mutated, that mutation persists through each call.

This is an error that would be very easy to catch in code review. Unfortunately, there are tens of millions of developers in the world, and even a 99% review success rate has bad implications.

So at the end of the day, we might not *need* to descend further down than the AST. Let's see how we can fix such bugs while staying above ground.

It turns out that program analysis can take many forms. We might want to use it for security purposes, we might want to use it for style checking, and we might want to use it to automate code review for things like the error we just saw.

A common theme in program analysis is simply finding certain things in your source code. This might be something like an unused variable, a type error, or even a function call which might lead to a security vulnerability. We call such regions of interest a **finding**.

Semgrep is a program analysis tool[10] that is mainly aimed being customizable to all of these use cases. This means that while that while some tools find security vulnerabilities, and some tools find style errors, the answer to what Semgrep finds is *whatever you want*.

[10] And company.

Semgrep stands for *semantic grep*.

If you've used the tool `grep` before, you know that it is really just a no-nonsense precursor to what is now Ctrl-F in most browsers.[11]

But, it's limited in some ways. Suppose we wanted to find all instances of using the function `print` to debug, before we put the code up for review. What do you think is going to happen when we search for every instance of the literal substring "print"?

The answer: **We are going to be absolutely inundated with results.** We'll find the string `print` if it occurs in comments, if it occurs in literal strings, and even if it's part of a larger method name. That doesn't make it what we were looking for, though!

In other words, `grep` doesn't understand the *meaning* of what we're looking for, it's just looking for sequences of characters. Semgrep does.

[11] Such as Arc, the browser that I am currently using.

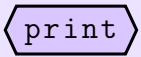
How does Semgrep understand the semantics of the program? Recall that the literal text in a program just serve as a proxy that indicates the underlying tree of the program, which contains its real meaning.

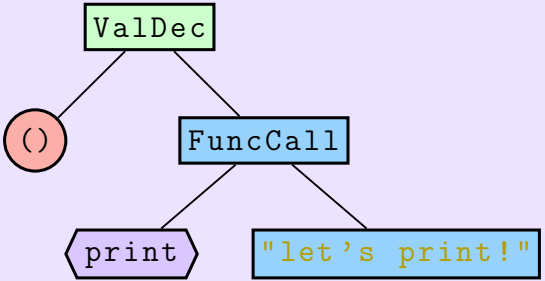
Semgrep doesn't do text search, it does *tree search*. This means that instead of searching for a sequence of characters within a given program, it searches for a *matching subtree* within a program's AST.

This means that a user might specify a particular **pattern AST**, which refers to the AST that they would like to search for, and then Semgrep will search the **target AST** to find a subtree which matches it.

The key innovation of Semgrep is that, although it does AST matching, the user doesn't need to be able to construct one, or even know what it is. Pattern ASTs are derived from patterns written *in the source language*, meaning that they look like code.

So, our query might look like:

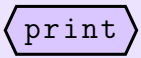


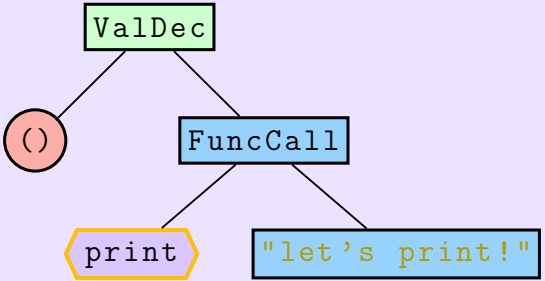



```
print
```

```
val () = print "let's print!"
```

So, our query might look like:






```
print
```

```
val () = print "let's print!"
```

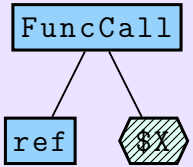
Notably, however, we avoid matching the instance of the literal string of "print" inside of the string that is being passed to its namesake function.

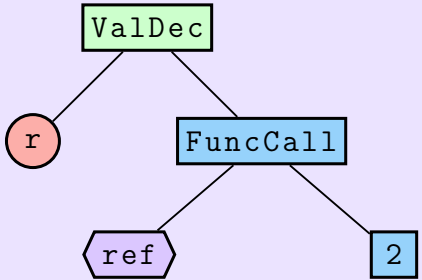
This is because we are only matching at the granularity of nodes of the AST, not of the constituent text. Regular expressions, for instance, would be fooled by instances of "print" within comments or strings.

When it comes to minimizing instances of program analysis tools being flat-out wrong, this is an invaluable help. Most program analysis tools are black-box applications that perform some magic to produce matches, without necessarily being understandable. The algorithm of Semgrep is really quite simple, and still manages to avoid false positives.

We can do a little more advanced of a query, too. Suppose that we were interested in finding all of the examples of a call to the function `ref`.

Well, we might write a pattern like `ref $X`, which uses a *metavariable* to bind to any sub-AST.

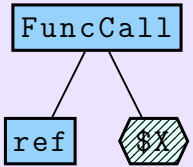


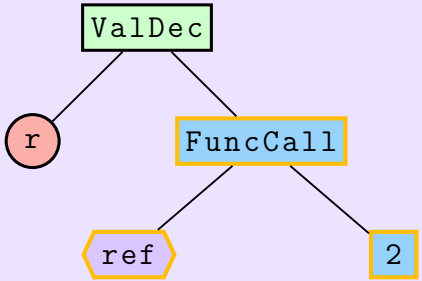



```
ref $X
```

```
val r = ref 2
```

Here, we bind the metavariable `$X` to the node of `2`, and then the entire pattern succeeds at matching the highlighted sub-tree of the target.

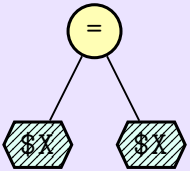


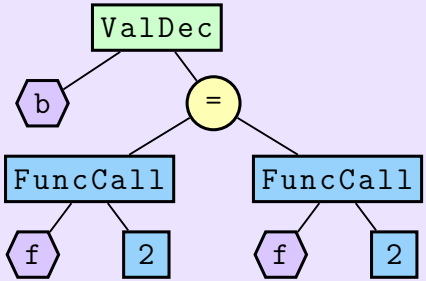



```
ref $X
```

```
val r = ref 2
```

We can also express more complicated queries. What if we want to find instances of a useless comparison, such as testing if an expression is equal to itself?

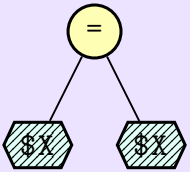


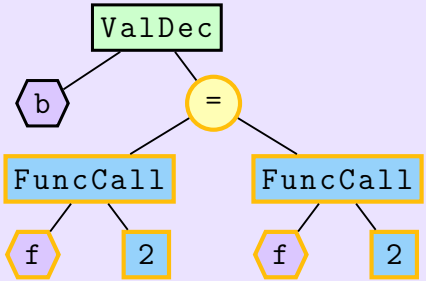



```
$X = $X
```

```
val b = (f 2 = f 2)
```

Here, because we reuse the metavariable `$X` twice, the two instances **unify**, meaning that the ASTs that each binds to must be the same. In this case, that works.

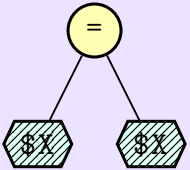


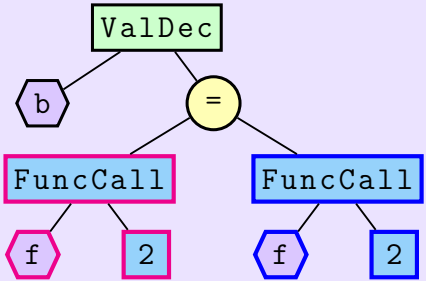



```
$X = $X
```

```
val b = (f 2 = f 2)
```

But in this case, there would be no match, because the blue and magenta subtrees do not match.






```
$X = $X
```

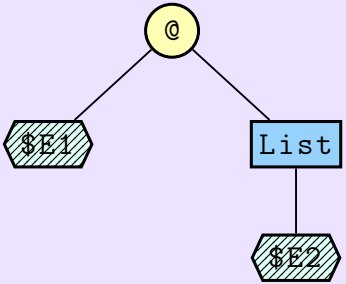
```
val b = (f 2 = g 2)
```

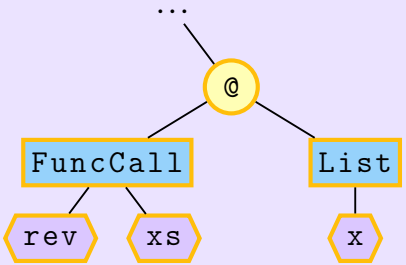
Let's apply this to a very concrete example that we saw earlier this semester. Recall that appending a singleton to the end of the list is an anti-pattern that generally indicates you are doing something inefficient.

We can write a Semgrep query which checks for precisely that!

```
$E1 @ [$E2]
```

Let's see it in action:






```
$E1 @ [$E2]
```

```
fun rev [] = []
  | rev (x::xs) =
      rev xs @ [x]
```

Because all programs parse to the same AST type, we only need a single matching function of type:

```
val match_exprs : expr * expr -> finding list
```

which can do matching for Java, for C, for OCaml, for Python, and many other languages besides.

For the simple example we saw earlier, of matching the Python function with a list as a default argument, we can write a very simple Semgrep rule.

When it comes to program analysis, tree matching isn't necessarily enough. Semgrep's main matching capability is at its core more of a Type C analysis, but sometimes we want to be able to find more semantic bugs!

When it comes to tracking the flow of data, Semgrep uses the theory of dataflow analysis as developed before to implement **taint tracking**, which allows you to specify certain sources of bad data, and sinks where that data should not flow into, during the program's execution.
Here's an example of taint tracking with Semgrep.

It can also perform constant propagation much in the same way that we did earlier: Here's an example.

Semgrep is aware of the semantics of the programming language that it scans, meaning that it is not just a simple matching process. It's a syntactically-based analysis that is supplemented by deeper semantic analysis.

6 - Conclusions

Program analysis is an extremely important field which aims to both enforce that the code we write is safe, as well as supplement the process of writing code by providing programmers with the information they need to execute smoothly. Even in the face of mathematical impossibility, it prevails.

Syntax highlighting, IDE warnings, type-checkers, and autoformatters are all among the niceties that programmers enjoy, due to advances made in analyzing and better making information available about programs.

In a world where there are tens of millions of developers, and billions upon billions of lines of code, it's simply not possible to cognize in our human brains that volume of code. Things will slip through the cracks. Catastrophic things might happen.

But, programs are ultimately just recursive entities, and recursion is just a technique for fitting an unlimited amount of things into your brain.

There is something to be said about making an impact with your work.

One of the things I value a lot is the ability to tackle interesting problems. Another is in being able to use elegant foundations (functional programming), and another is in being able to ultimately make a real impact.

Program analysis is my trinity of all three. There's nothing else in the world like it.

Thank you!