On the Architecture of a (Verifying) Compiler

David J. Pearce

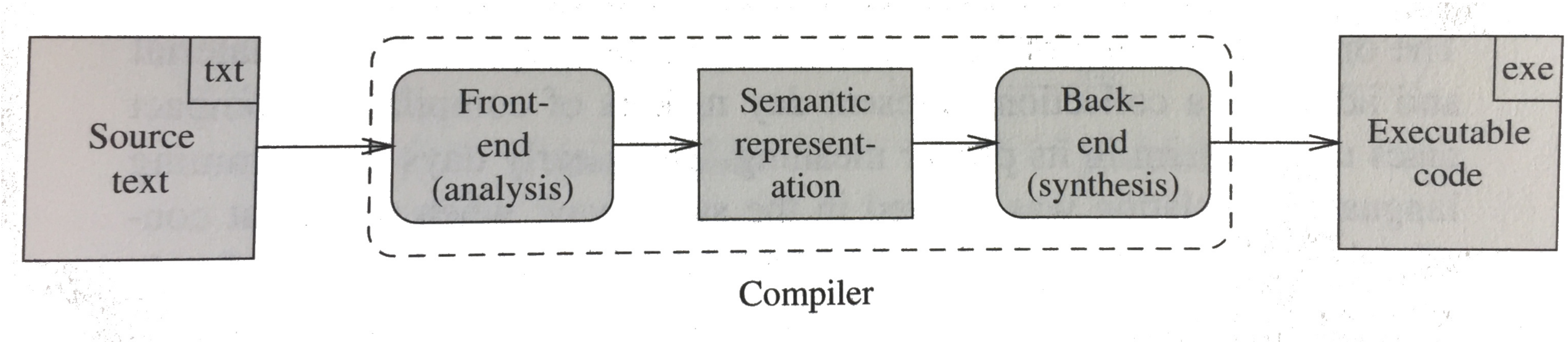
School of Engineering and Computer Science
Victoria University of Wellington

@WhileyDave
http://whiley.org http://github.com/Whiley

What is a Compiler?

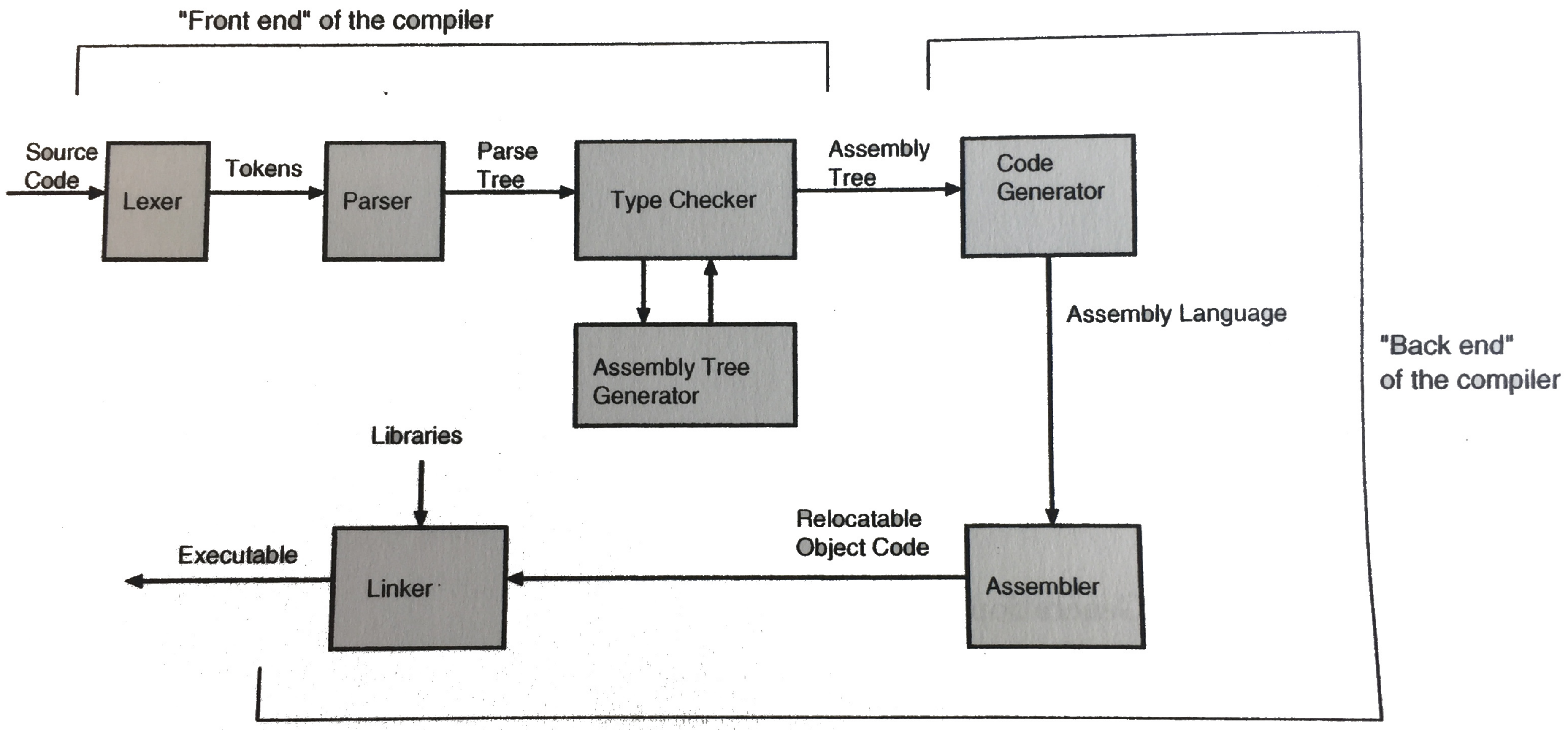
"A **compiler** *is likely to perform many or all of the following operations:* **preprocessing***,* **lexical analysis***,* **parsing***,* **semantic analysis** *(syntax-directed translation),* **conversion of input programs to an intermediate representation***,* **code optimization** *and* **code generation***"* *–Wikipedia*

What is a Compiler?



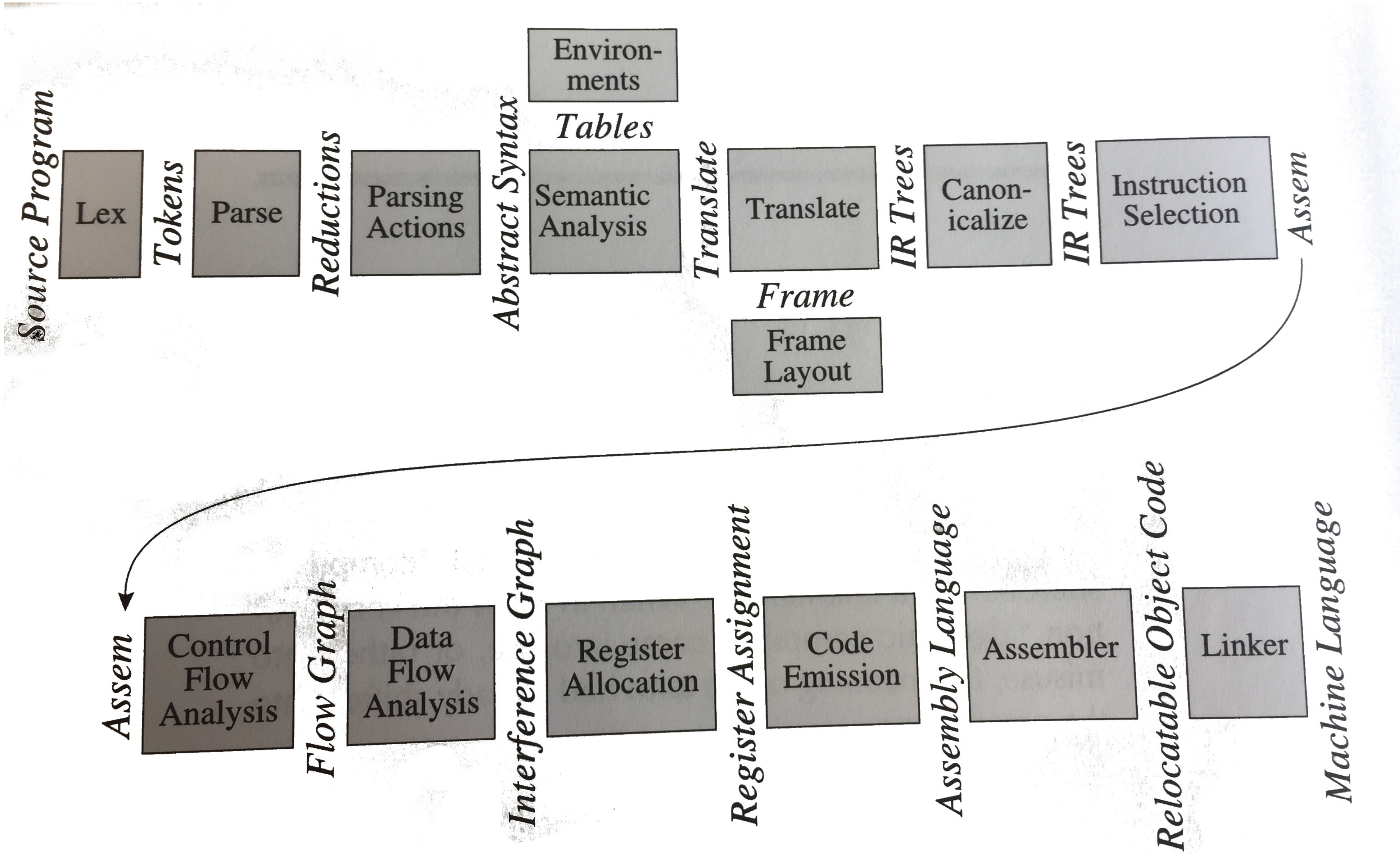
– Grune, Bal, Jacobs, Langendoen

What is a Compiler?



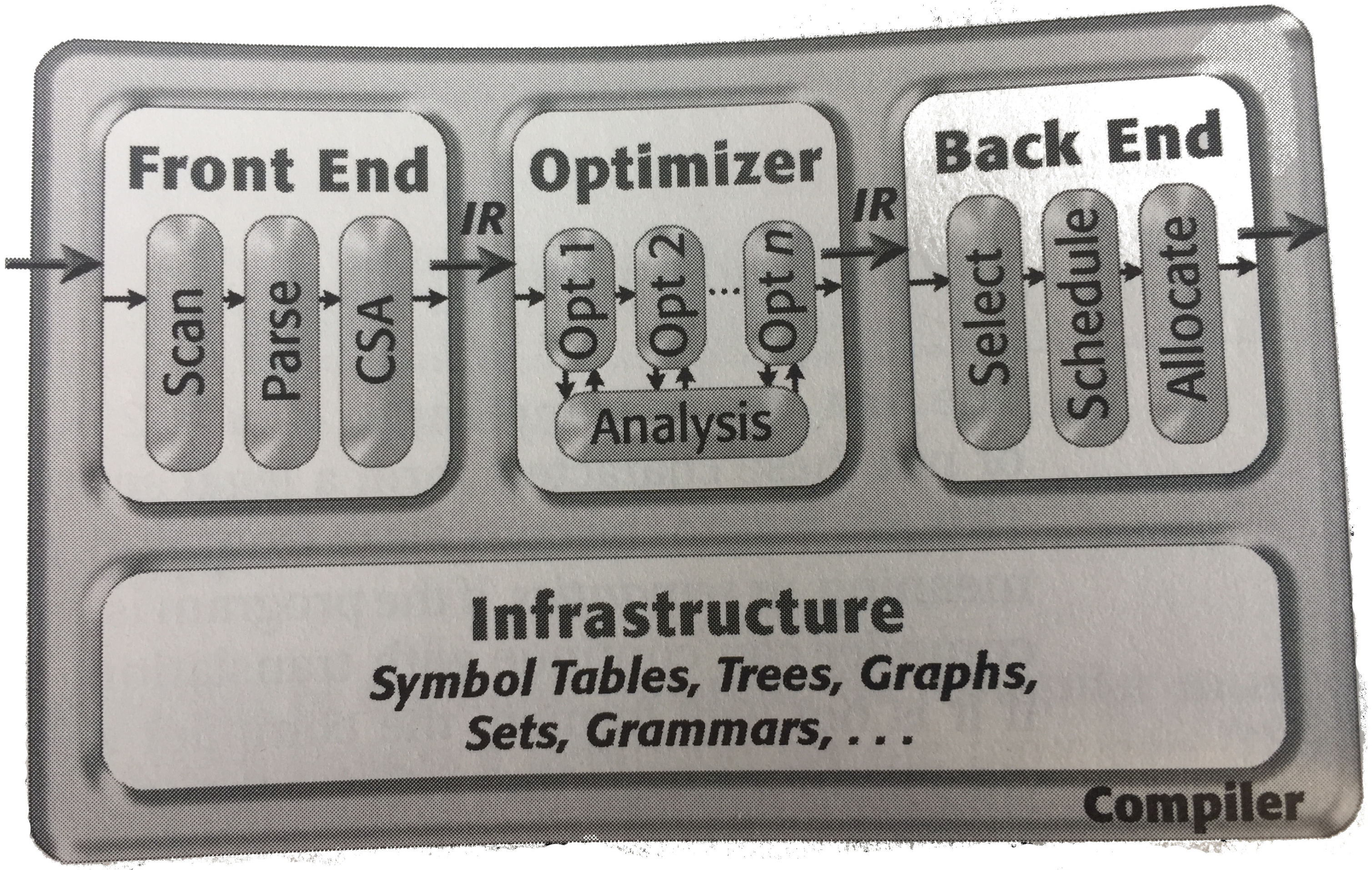
– *Galles*

What is a Compiler?



– Appel

What is a Compiler?



– Cooper & Torczon

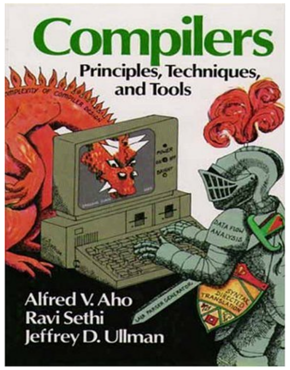
Semantic Analysis

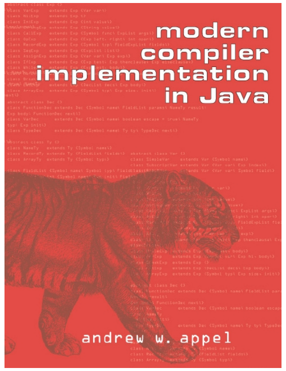
“Semantic analysis or context sensitive analysis is a process in compiler construction, usually after parsing, to gather necessary semantic information from the source code. It usually includes **type checking***, or makes sure a variable is declared before use which is impossible to describe in the extended Backus-Naur form and thus not easily detected during parsing.”* *–Wikipedia*

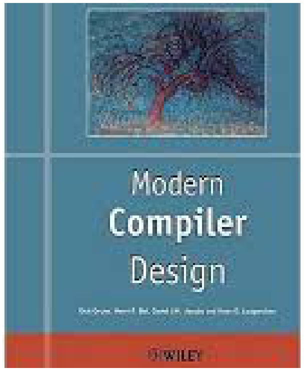
- **Name Resolution.**
- **Type Checking.**
- **Definite Assignment.**
- **Dead Code.**
- **Borrow Checking.**
- **Verification.**

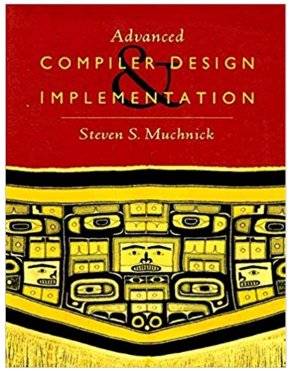
Books on Compilers

Book	Parsing	Semantic Analysis	Code Gen
Aho *et al* ('86)	34.6%	5.7%	33.9%
Appel ('02)	12.1%	7.5%	58.8%
C & T ('04)	17.1%	7.8%	63.7%
Galles ('05)	29.7%	10.5%	17.0%
Grune *et al* ('00)	19.5%	11.0%	43.1%
Scott ('06)	7.6%	11.0%	3.8%
Muchnick ('97)	0.0%	14.7%	47.6%



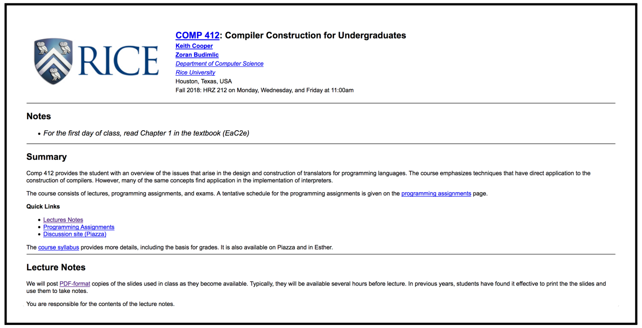


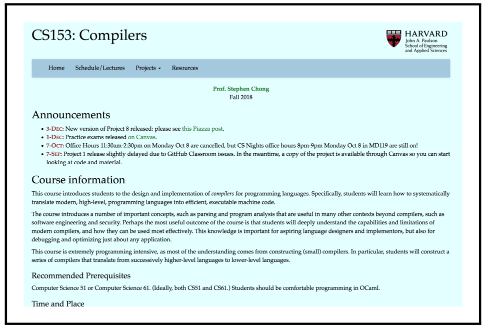




Courses on Compilers

Book	Parsing	Semantic Analysis	Code Gen
CS153	15%	3%	57%
COMP412	42%	5%	26%
CS143	33%	11%	33%
CSE401	27%	13%	48%
IN4303	35%	17%	18%
SWEN430	13%	25%	29%





Inside an Actual Compiler! (Javac, OpenJDK7)

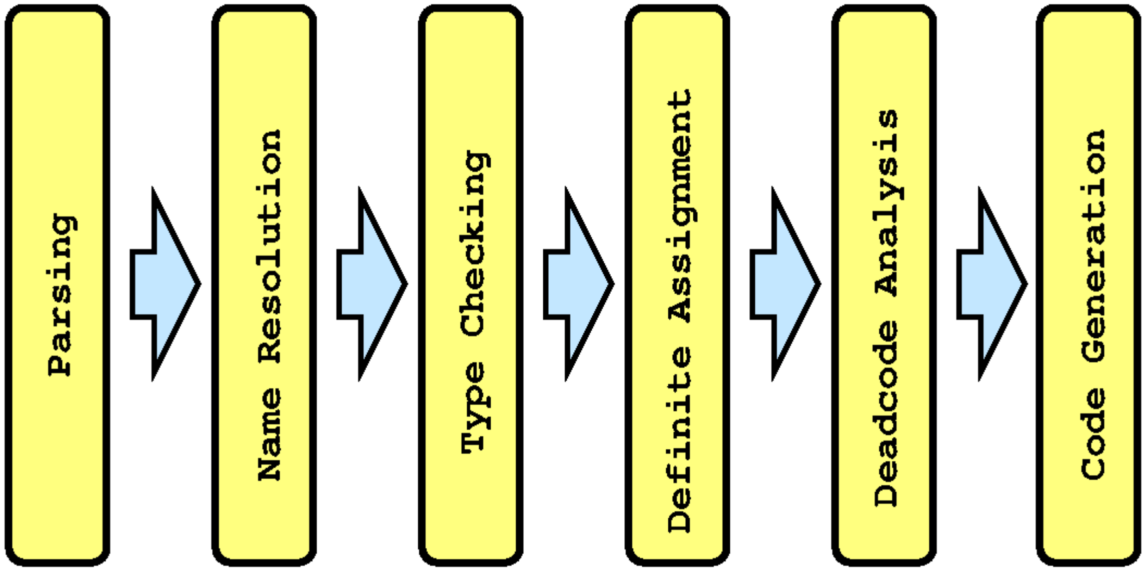
Package	LOC
`com/sun/tools/javac/parser`	5377
`com/sun/tools/javac/comp`	18633
`com/sun/tools/javac/jvm`	11288
`com/sun/tools/javac/tree`	6475

Suite	Parsing	Attribution	Flow	Code Generation
JKit Tests	192ms	821ms	34ms	446ms
JKit Apps	158ms	464ms	31ms	314ms

- JKit Test Suite — **266** Individual Classes
- JKit Apps Suite — **5** applications comprising **127** Classes

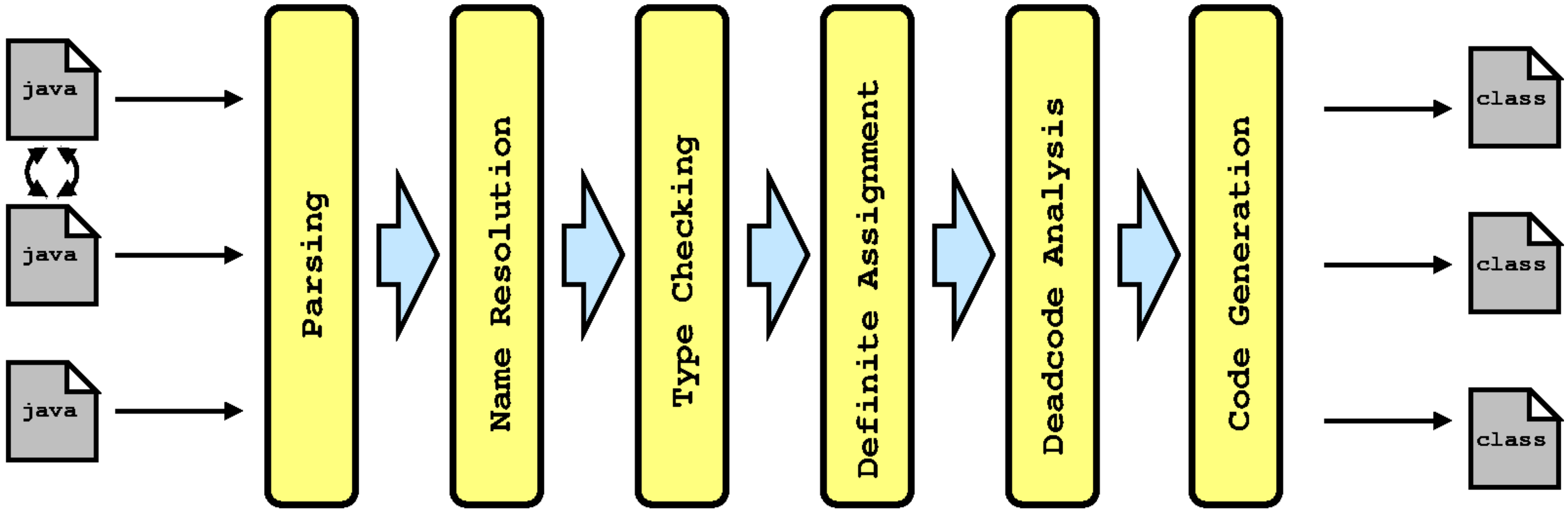
Compilation Pipeline

Overview of Java Compiler



Q) *What goes through our pipeline?*

Overview of Java Compiler



Q) *What goes through our pipeline?*

A) *Compilation "groups"!*

Static Initialisers!

“A properly formed SCJ program should **not** *have cyclic dependencies within class initialization code.” –JSR302*

Parent.java

```
class Parent { static int ZERO = Child.ONE; }
```

Child.java

```
class Child { static int ONE = Parent.ZERO + 1; }
```

Q) *Why is this permitted??*

Name Resolution!

A.java

```
public class A { int field = 0; }
```

B.java

```
public class B {
  protected int field = 123;

  class C extends A { int f() { return field; } }

  public static void main(String[] args) {
     System.out.println(new B().new C().f());
} }
```

Q) *What gets printed?*

Borrow Checking in Rust

```
fn f() -> i32 {
  let mut x = 1;
  let y = &x;
  x = x + 1;
  return x + *y;
}
```

```
struct Point {x: i32, y: i32}

fn f() -> i32 {
  let mut p = Point{x:1,y:2};
  let br = &mut p.y;
  return p.x + *br;
}
```

- Borrow checking in Rust is **flow sensitive**...

Incremental Compilation

“An **incremental compiler** *is one that when invoked, takes only the changes of a known set of source files and updates any corresponding output files (in the compiler’s target language, often bytecode) that may already exist from previous compilations.”*

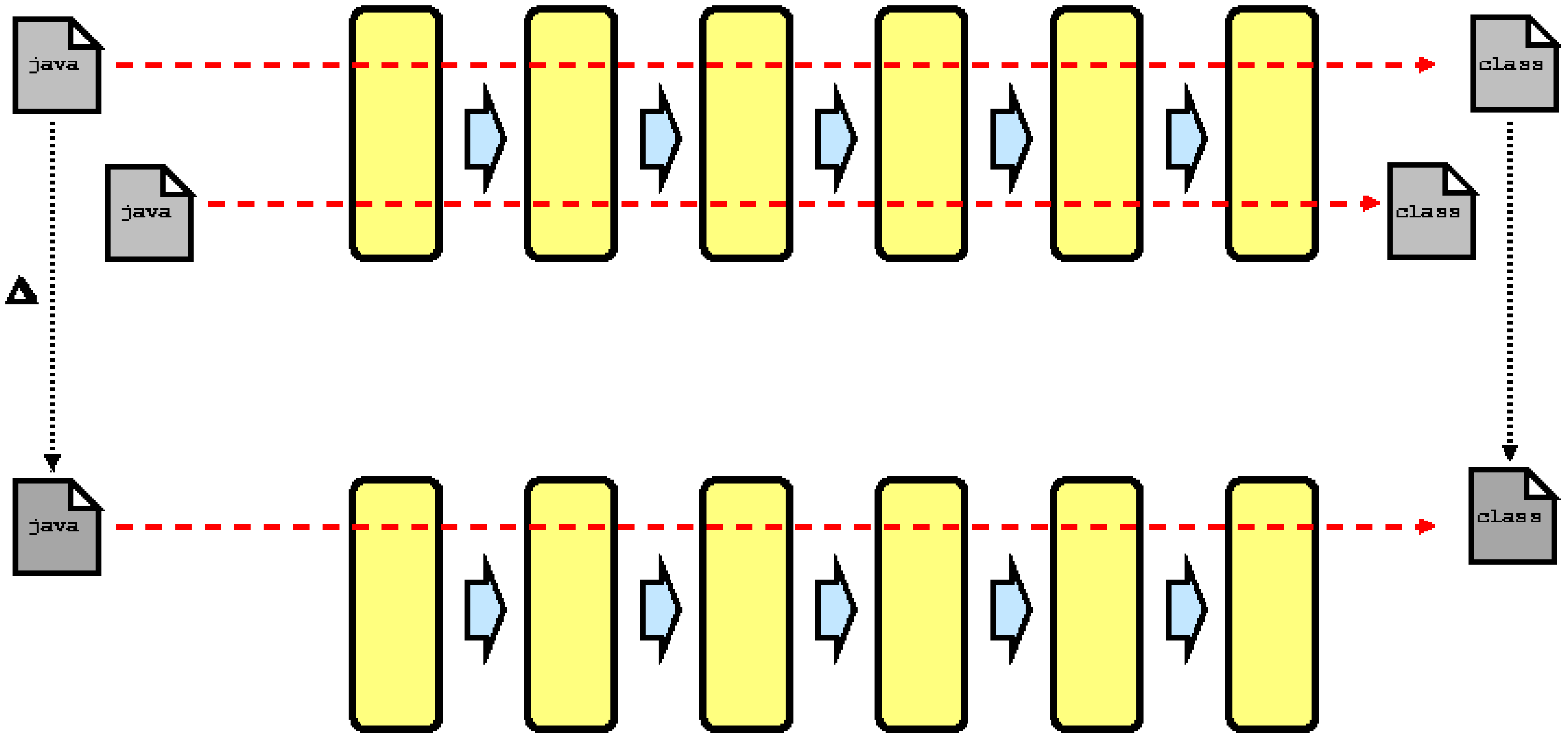
–Wikipedia

Incremental Compilation

Compiler	Incremental	Fine-Grained
GCC/make	Y	N
Javac	Y	N
Eclipse	Y	N
Go	Y	N
Scala	Y	N
Rust	Y	**?**

Q) *Why so few fine-grained incremental compilers?*

Incremental Compilation



Q) *What goes through our pipeline now?*

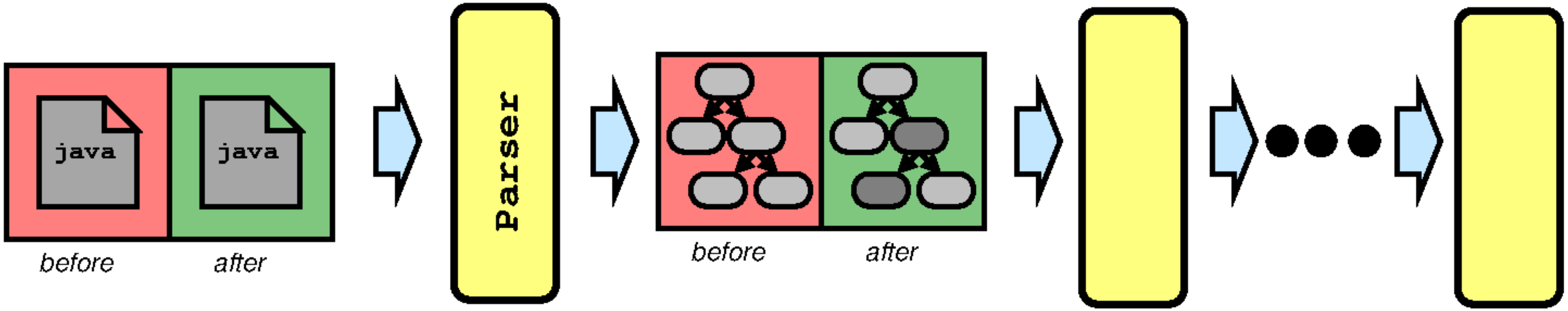
Incremental Compilation

```
class Child {
 private Parent link;

 public Child(Parent l){
   this.link = l;
 }

 public String getText() {
   return "Child";
}}
```

```
class Child {
 private Parent ptr;

 public Child(Parent l){
   this.ptr = l;
 }

 public String getText() {
   return "Child";
}}
```

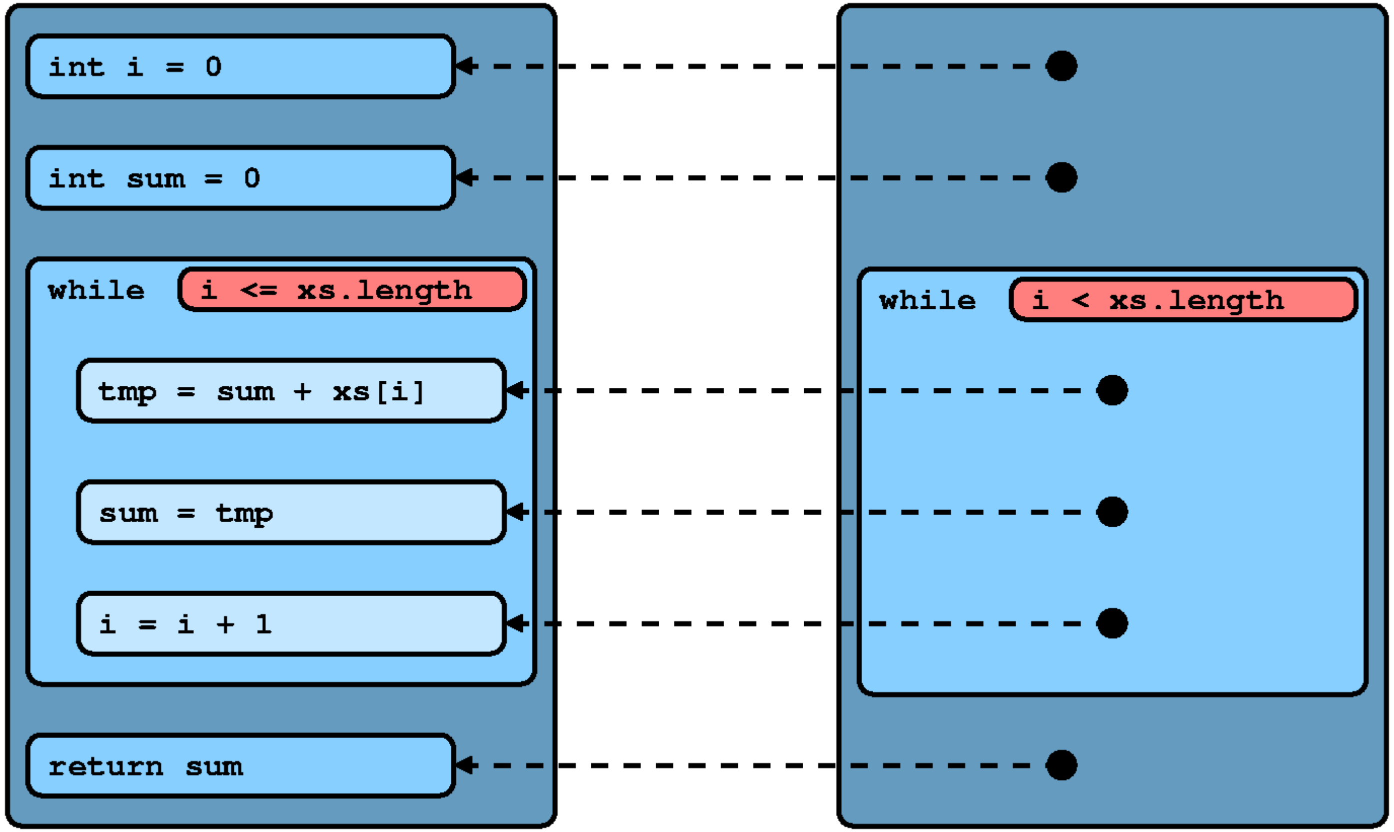
Q) *What goes through our pipeline now?*

Incremental Compilation



- **Parser**. Now accepts *source delta*
- **Semantic Analysis**. Now accepts *AST delta*

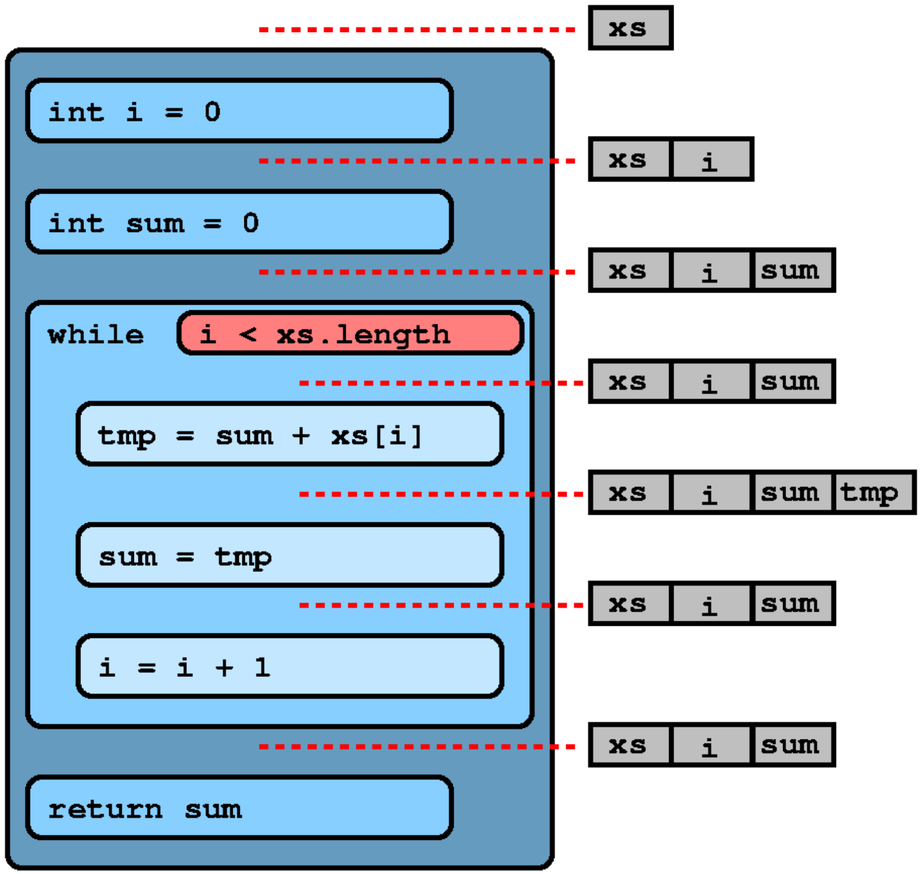
Incremental Compilation



Before *After*

- **Incremental Update.** Parser produces a *tree delta*...

Incremental Semantic Analysis



- **Incremental Update.** Invalidate affected nodes and *restart*

Whiley

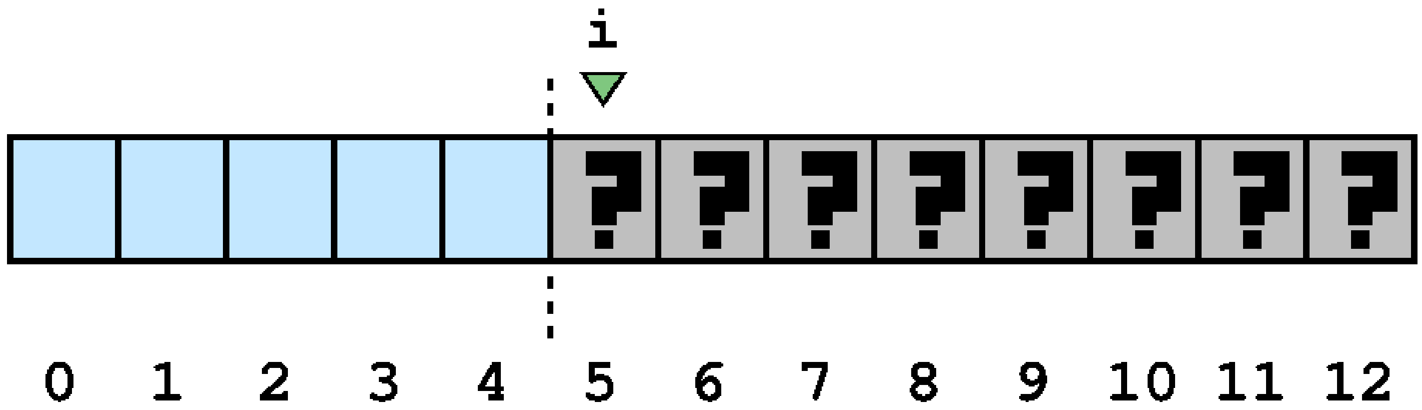
Whiley: Overview

```
function max(int x, int y) -> (int z)
// result must be one of the arguments
ensures x == z || y == z
// result must be greater-or-equal than arguments
ensures x <= z && y <= z:
    ...
```

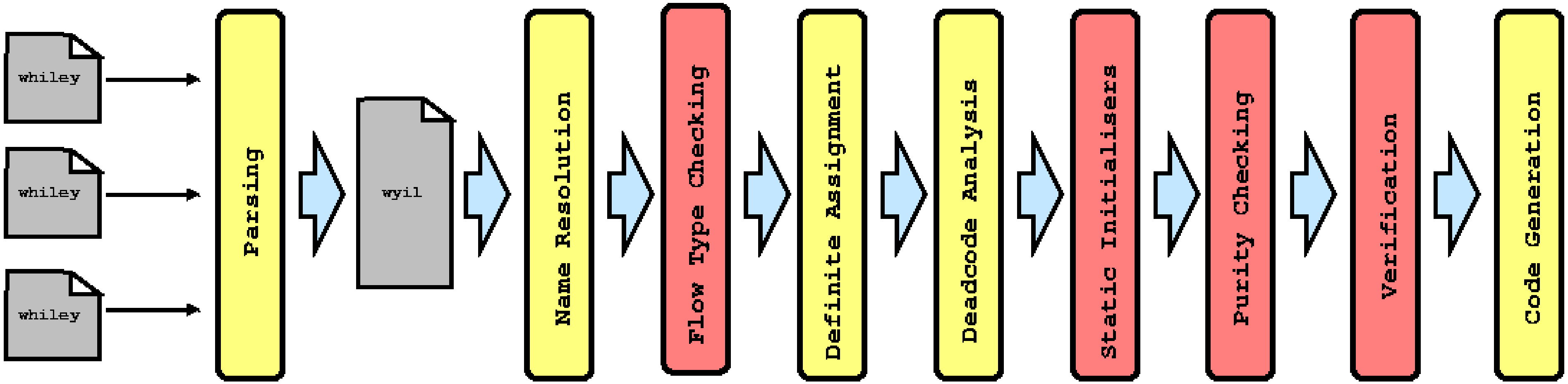
- A language designed specifically to simplify **verifying software**
- Several trade offs e.g. **performance for verifiability**
  - *Unbounded Arithmetic, value semantics, etc*
- **Goal**: to statically verify functions meet their specifications

Whiley: Demo!

"Given an array, find the index of a given item."



Whiley: Compiler Pipeline



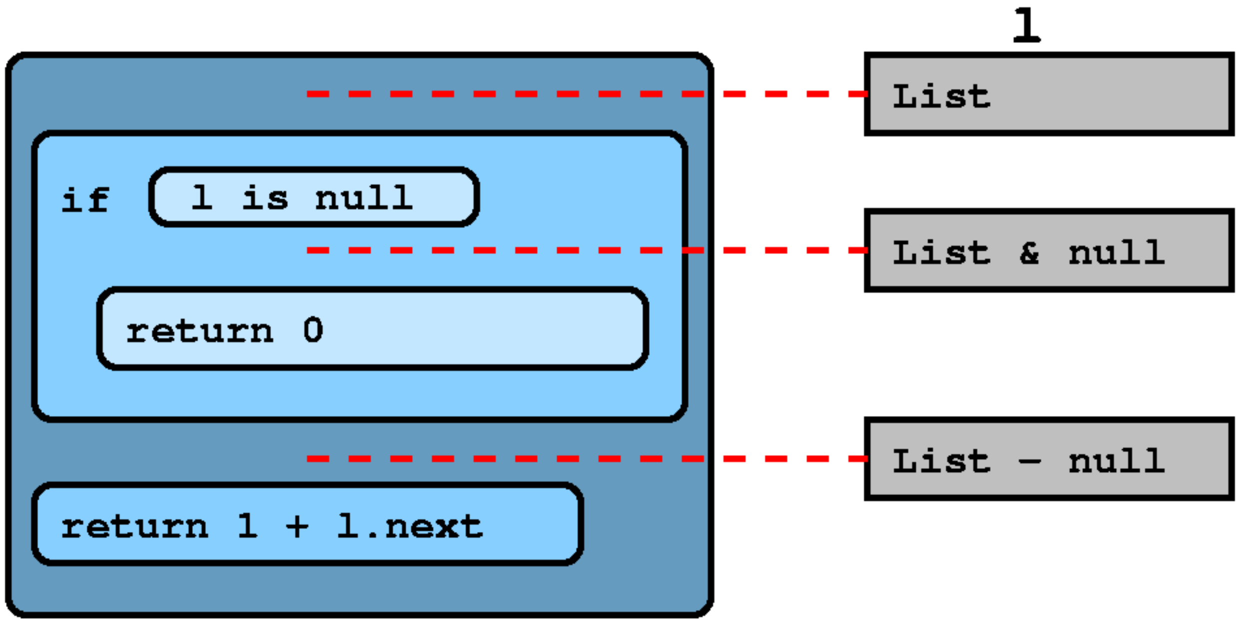
- Purity checking to ensure `functions` are **pure**
- Static initialiser checking to ensure **acyclic initialiser graph**
- Verification begins with **Verification Condition Generation**

Whiley: Flow Typing

```
type List is null | { List next, int data }

function length(List l) -> (int r):
    //
    if l is null:
        return 0
    //
    return 1 + length(l.next)
```

- Flow typing is a **flow-sensitive** activity

Whiley: Flow Typing



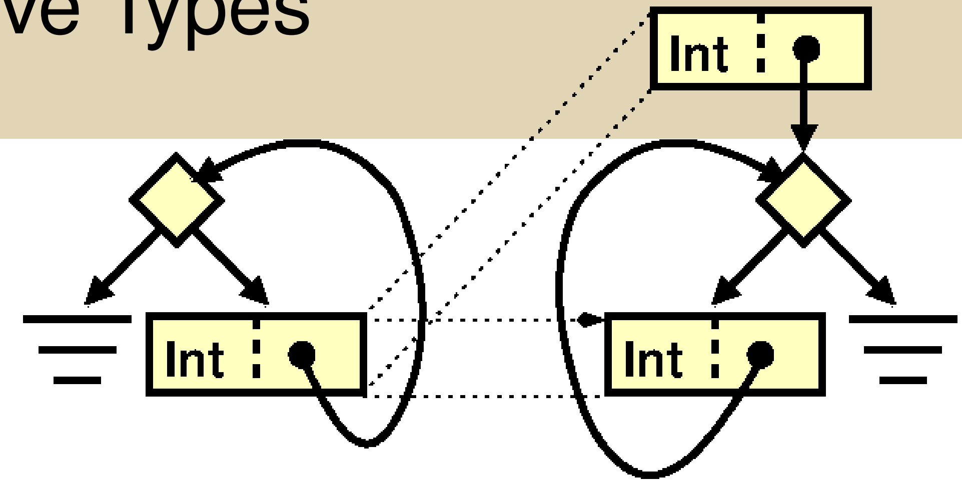
- Determines type for **each** variable at **every** point
- Flow typing is therefore **more expensive**...

Whiley: Flow Typing

```
function indexOf(int[] items, int item) -> (int|null r)
// If integer value returned, must be index of item
ensures r is int ==> items[r] == item
// No element before integer r matches item
ensures r is int ==> all { k in 0..r | items[k] != item }
// If null returned, no matching item
ensures r is null ==> all { k in 0..|items| | items[k] != item }:
    //
    int i = 0
    //
    while i < |items|
    where i >= 0 && i <= |items|
    where all { j in 0..i | items[j] != item }:
        if items[i] == item:
            return i
        i = i + 1
    //
    return null
```

- Flow typing in **expressions** is useful!!

Whiley: Structural Recursive Types



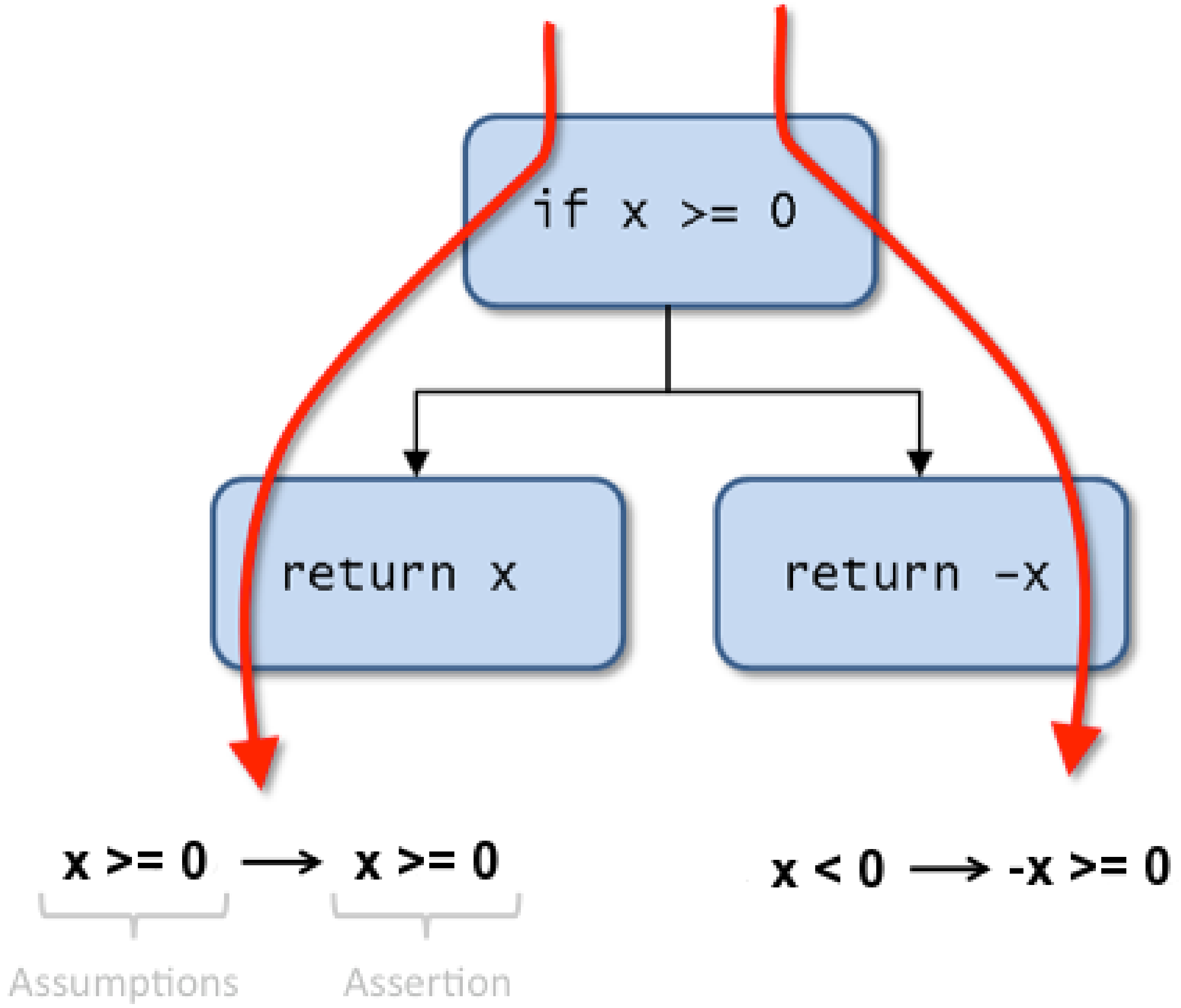

```
type List is null | { List next, int data }
type NonEmptyList is { List next, int data }

function append(List l, NonEmptyList r) -> List:
    if l is null:
        return r
    else:
        l.next = append(l.next, r)
        return l
```

Whiley: Verification

```
function abs(int x) -> (int r)
// Either x or its negation returned
ensures (r == x) || (r == -x)
// return value cannot be negative
ensures r >= 0:
    //
    if x >= 0:
        return x
    else:
        return -x
```

- For this example, **2** verification conditions generated

Whiley: Verification Condition Generation



Whiley: Assertion Language

> *"the best and most far-reaching single design decision we made in the implementation of the Spec# verifier was to introduce the intermediate language Boogie in between the Spec# program and the formulas sent to the theorem prover."*
>
> –*Barnett* et al.

Whiley: Assertion Language

- Whiley compiler emits verification conditions in **assertion language**

```
assert:
  forall (int x):
    x >= 0 ==> x >= 0

assert:
  forall (int x):
    x < 0 ==> -x >= 0
```

- Verification conditions from `abs()` example shown above
- In principle, can hook up different **automatic theorem provers**

Whiley: Performance

Suite	Parsing	Type Checking	Semantic Analysis	Verification
Valid	474ms	592ms	711ms	55107ms
Average	28ms	24ms	29ms	73ms
Fib	26ms	21ms	24ms	51ms
GCD	30ms	24ms	27ms	117ms
Matrix	50ms	157ms	36ms	16031ms
Queens	83ms	146ms	43ms	8249ms
Regex	44ms	173ms	29ms	2567ms

- Valid test suite comprised **582** test cases
- Bench testsuite comprised **6** micro-benchmarks

Verification

Automated Theorem Proving

"These [decision] procedures have to be **highly efficient***, since the problems they solve are* **inherently hard***."*

– Kroenig and Strichman

"Automatic theorem provers (ATPs) based on the resolution principle ... have reached a **high degree of sophistication***. They can often find long proofs even for problems having* **thousands of axioms***"*

–Benzmuller et al.

- "Automated Theorem Provers are a **dark art** — just use Z3!"

Theorem Proving: Assertion Language

- Whiley compiler emits verification conditions in **assertion language**

```
define abs_ensures_0(int x, int r) is:
    (r == x) || (r == -x)

assert "postcondition not satisfied":
    forall(int x):
        if:
            x >= 0
        then:
            abs_ensures_0(x, x)
```

- Verification conditions from `abs()` example shown above
- In principle, can hook up different **automatic theorem provers**

Theorem Proving: Proofs

(1)	$\exists(\texttt{int x}).(\texttt{x} \geq 0 \wedge \texttt{x} < 0)$	
(2)	$\texttt{x}_1 < 0 \wedge \texttt{x}_1 \geq 0$	($\exists$-*elimination*, 1)
(3)	$\texttt{x}_1 \geq 0$	($\wedge$-*elimination*, 2)
(4)	$\texttt{x}_1 < 0$	($\wedge$-*elimination*, 2)
(5)	$0 < 0$	($\leq$-*closure*, 3 + 4)
(6)	$\perp$	(*simplification*, 5)

- Purpose-built **Automated Theorem Prover** developed
- Focus on **simplicity** rather than **scale**
- For example, not based on DPLL

Theorem Proving: $\vee$-Elimination

(1) $\exists(\texttt{int x}).((\texttt{x} = 0 \vee \texttt{x} > 0) \wedge \texttt{x} < 0)$

---

(2)	$(\texttt{x}_1 = 0 \vee \texttt{x}_1 > 0) \wedge \texttt{x}_1 < 0$	($\exists$-*elimination*, 1)
(3)	$(\texttt{x}_1 = 0 \vee \texttt{x}_1 > 0)$	($\wedge$-*elimination*, 2)
(4)	$\texttt{x}_1 < 0$	($\wedge$-*elimination*, 2)

(5)	$\texttt{x}_1 = 0$	($\vee$-*elimination*, 2)
(6)	$\texttt{x}_1 < \texttt{x}_1$	(*congruence*, 4 + 5)
(7)	$\bot$	(*simplification*, 6)

(8)	$\texttt{x}_1 > 0$	($\vee$-*elimination*, 2)
(9)	$0 < 0$	($\leq$-*closure*, 4 + 8)
(10)	$\bot$	(*simplification*, 5)

Theorem Proving: Proof Optimisation

(1) $\exists(\mathtt{int\,i}).\big((\mathtt{i} < 0) \wedge (\mathtt{i} == 0) \wedge (\mathtt{i} > 0)\big)$

---

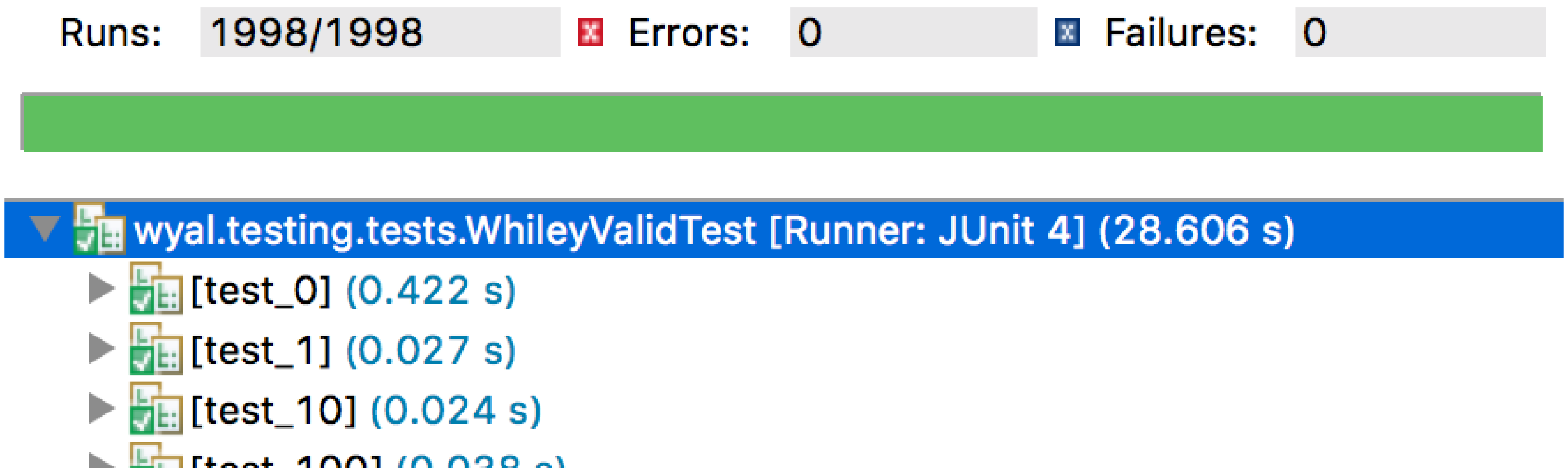
(2)	$(\mathtt{i}_1 < 0) \wedge (\mathtt{i}_1 == 0) \wedge (\mathtt{i}_1 > 0)$	($\exists$-*elimination*, 1)
(3)	$\mathtt{i}_1 < 0$	($\wedge$-*elimination*, 2)
(4)	$\mathtt{i}_1 == 0$	($\wedge$-*elimination*, 2)
(5)	$\mathtt{i}_1 > 0$	($\wedge$-*elimination*, 2)
(6)	$0 < 0$	(*congruence*, 3+4)
(7)	$\bot$	(*simplification*, 6)

- **Full Proof.** Reflects work done searching proof space by automated theorem prover.

- **Pruned Proof.** For easier reading, should eliminate unused facts which were explored.

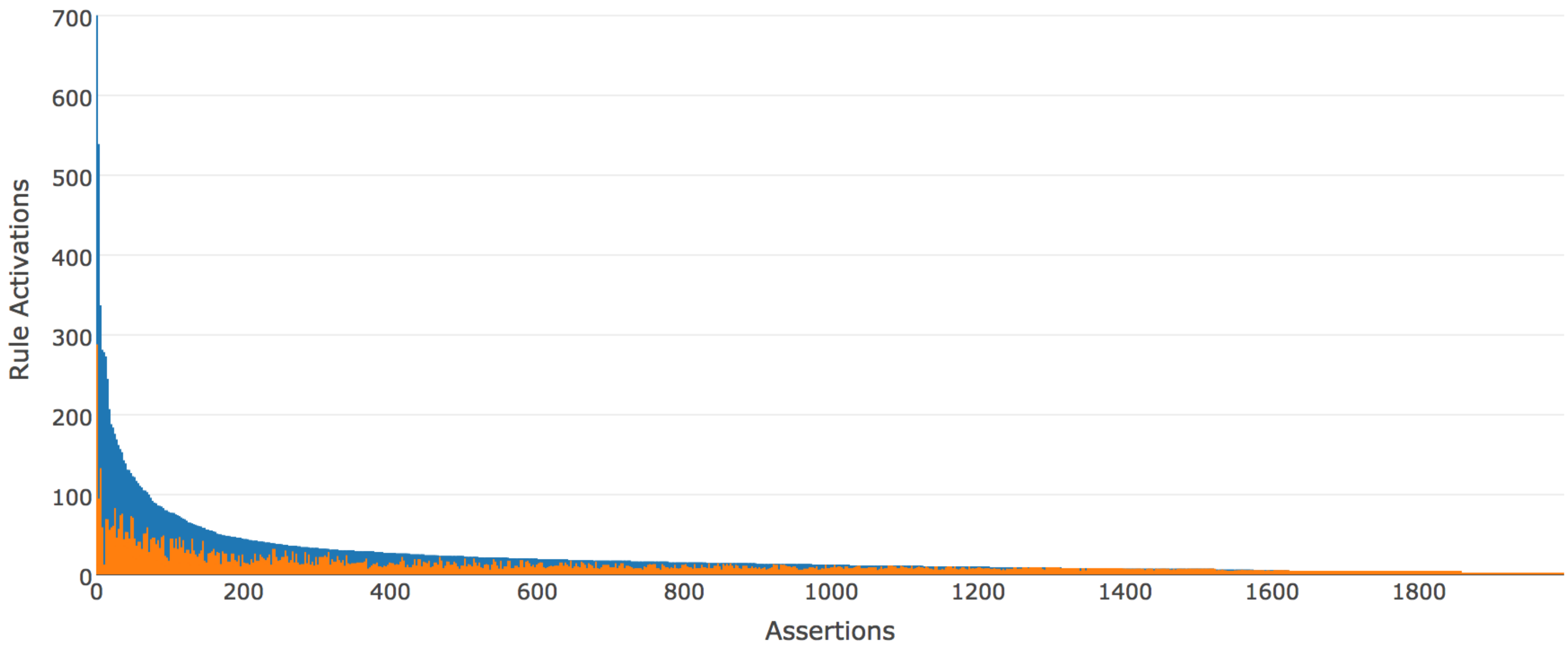
Q) *how big are these proofs?*

Theorem Proving: Data Set

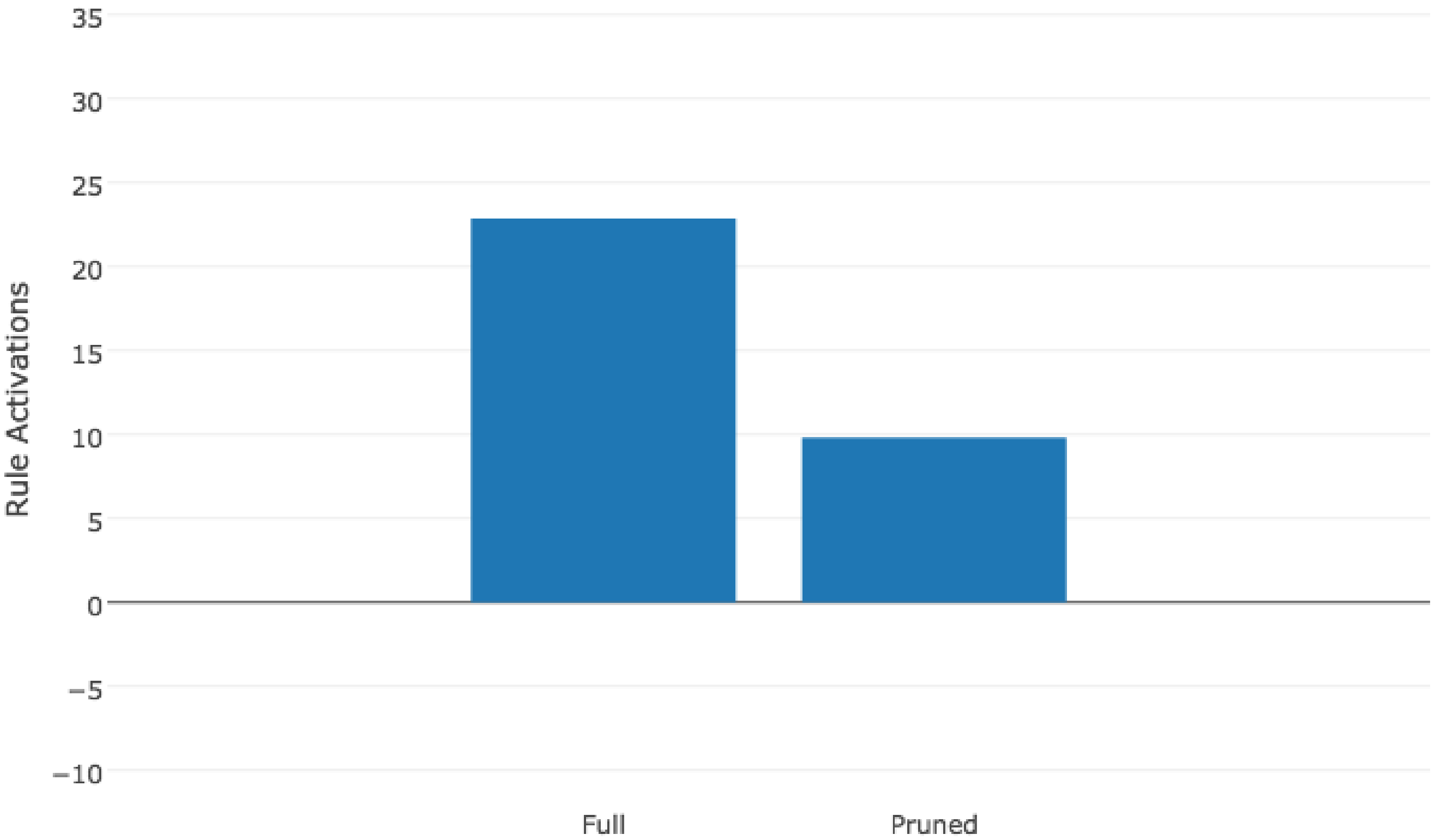


- Whiley Compiler has (approx) 540 valid and 287 invalid **test cases**
- Each test case is **single Whiley file** (either correct or not)
- From this, generated **1998 valid assertions** and **91 invalid assertions**

Theorem Proving: Experimental Results I



Theorem Proving: Experimental Results II



http://whiley.org

@WhileyDave
http://github.com/Whiley

Theorem Proving: Counterexample Generation?

> *"Most bugs have small counter examples"*
>
> *-Jackson'06*

Theorem Proving: Counterexample Generation

- **Approach.** Use brute force generation with a “small world” (e.g. integers in range $\langle -5 \ldots 5 \rangle$, array lengths $\langle 0 \ldots 2 \rangle$, etc).

```
forall(int i, int[] arr):
    (arr[i] >= 0) ==> (i == |arr|)
```

- **Example.** For above, generate models `i=0,arr=[]`, `i=0,arr=[0]`, `i=1,arr=[0]`, etc.

- **Problems.** E.g. *uninterpreted functions* and *undefined behaviour*?

```
forall(int[] xs):
    xs[0] > 0
```

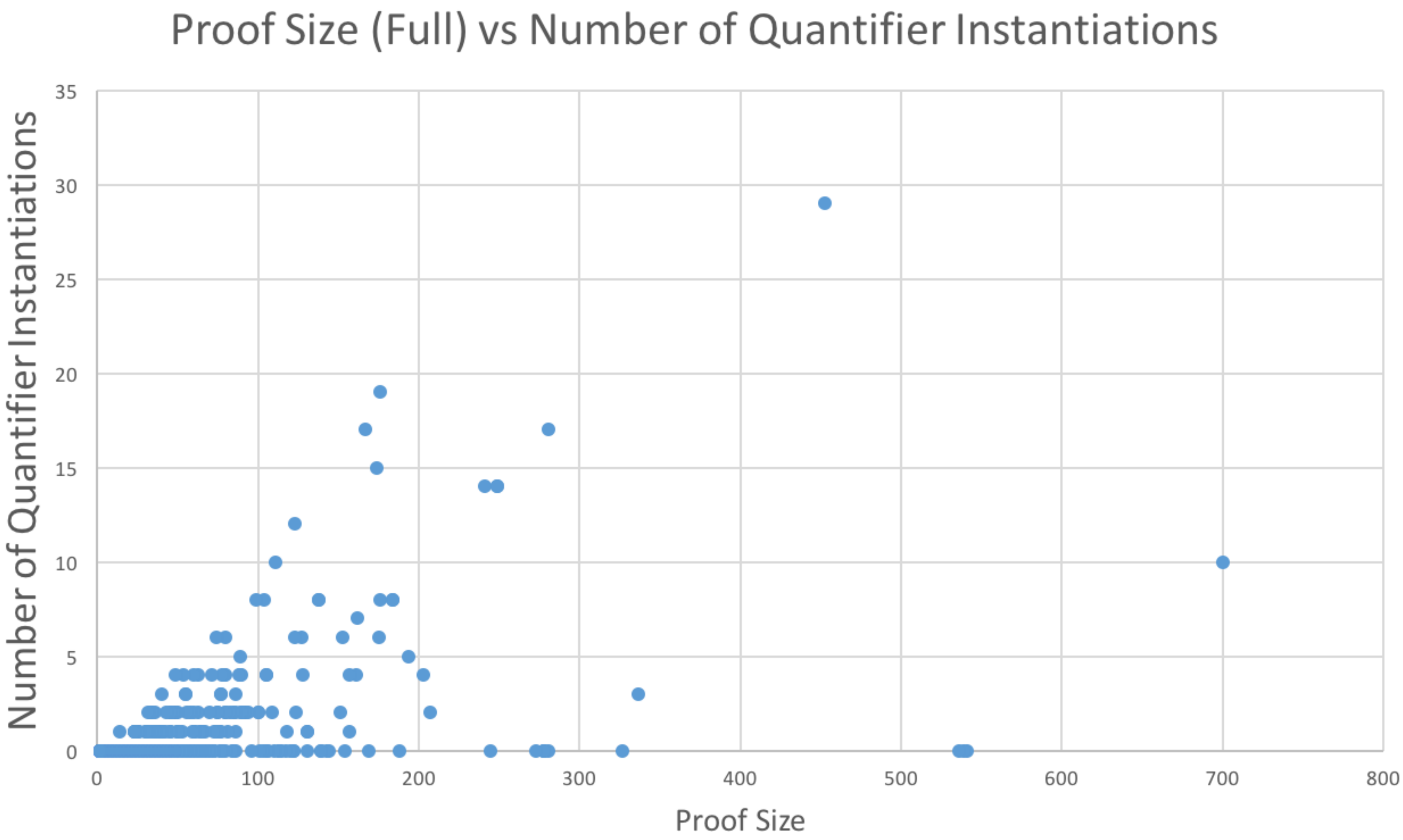
Theorem Proving: Counterexample Generation

Test	**Counterexample**
test_11	`i=1`, `x=[0]`, i_1`=1`, i_2`=1`
test_102	`xs=[0], y=0, x=1`
test_129	x_1`={f:-1}`, `x={f:1}`
test_198	r_1`=[0]`, `r=[0,0]`, `i=0`, i_1`=1`, `ls=[0,0]`

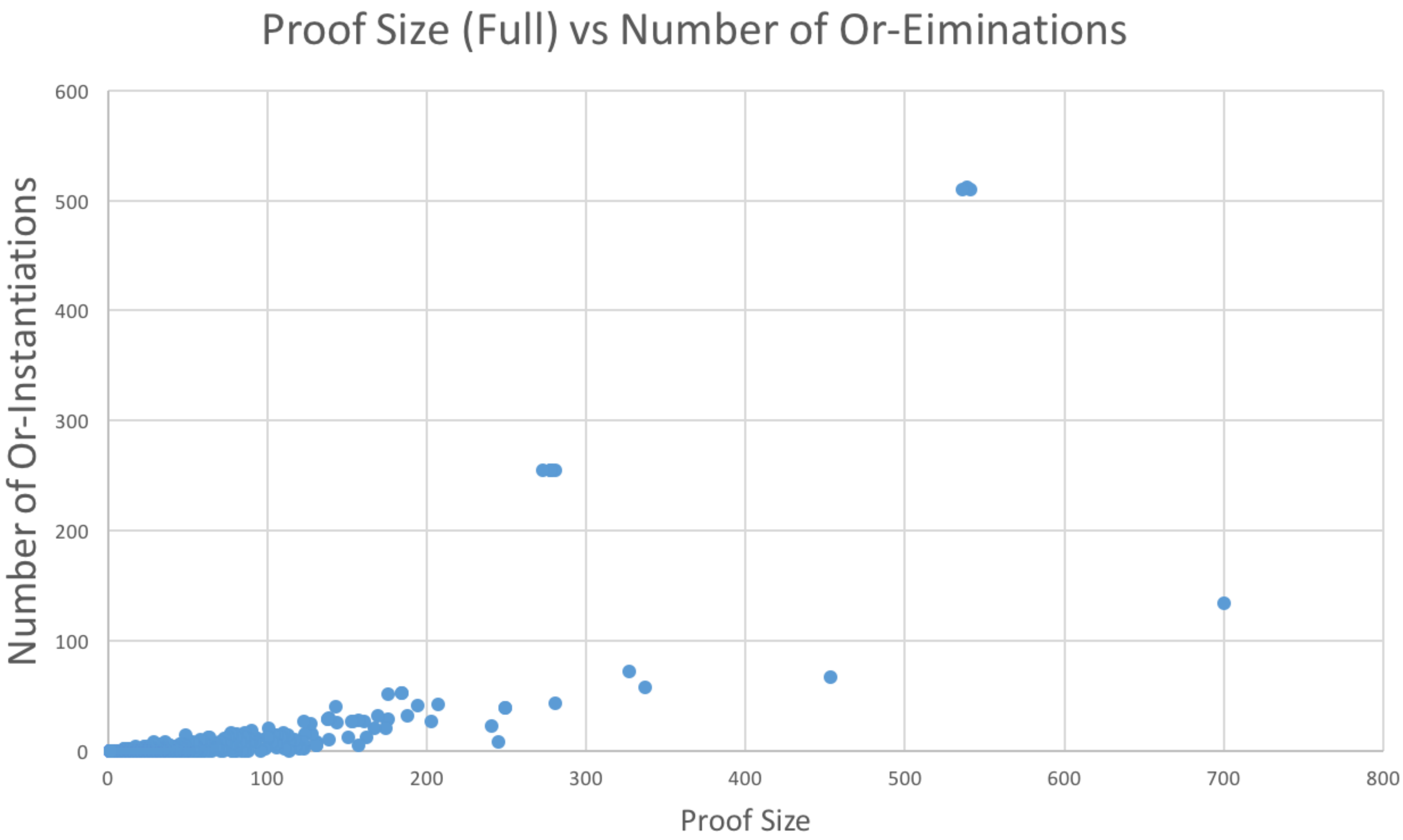
- Generated counterexamples for **75** / **91** invalid assertions!

Q) *What Causes a Large Proof?*

Theorem Proving: Experimental Results IV



Theorem Proving: Experimental Results V



Theorem Proving: Experimental Results VI