

Targeted Automated Testing Using Constraint Logic Programming

Kyle Dewey
Advisor: Ben Hardekopf



Targeted Automated Testing Using Constraint Logic Programming

Testing

Goal is to test some piece of software in the hopes of finding bugs before users do

Targeted Automated Testing Using Constraint Logic Programming

Automated

No user intervention necessary once we
start running things

Targeted Automated Testing Using Constraint Logic Programming

Targeted

Not completely random; trying to create specific inputs which act as good tests

Targeted Automated Testing Using **Constraint Logic Programming**

Constraint Logic Programming

99% Prolog, plus some other nice features

Outline

- Background
- Research problem
- Applications
 - Data Structure Generation
 - Generating JavaScript Programs with Known Runtime Behaviors
 - Testing Rust's Typechecker
 - Testing SMT Solvers
- Conclusion

Outline

- **Background**
- Research problem
- Applications
 - Data Structure Generation
 - Generating JavaScript Programs with Known Runtime Behaviors
 - Testing Rust's Typechecker
 - Testing SMT Solvers
- Conclusion

Automated Testing

Motivation

- Writing correct software is hard
- Writing tests is time-consuming
- CPU cycles are cheap

Background: Differential Testing

- Idea: generate an input via some process
- Run input on different implementations
- If implementations disagree on result, bug has been found

Input Generator

Input
Generator

Generates

```
function foo() { ... }  
...  
bar();
```

Test Input

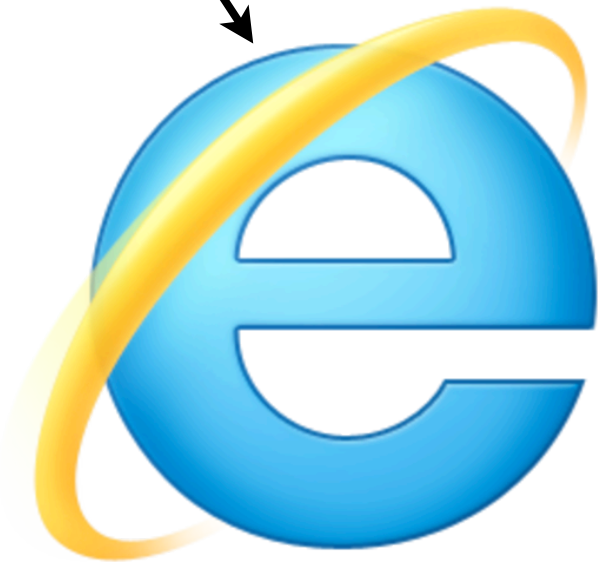
Input
Generator

Generates

```
function foo() { ... }  
...  
bar();
```

Test Input

Executed on



Input
Generator

Generates

```
function foo() { ... }  
...  
bar();
```

Test Input

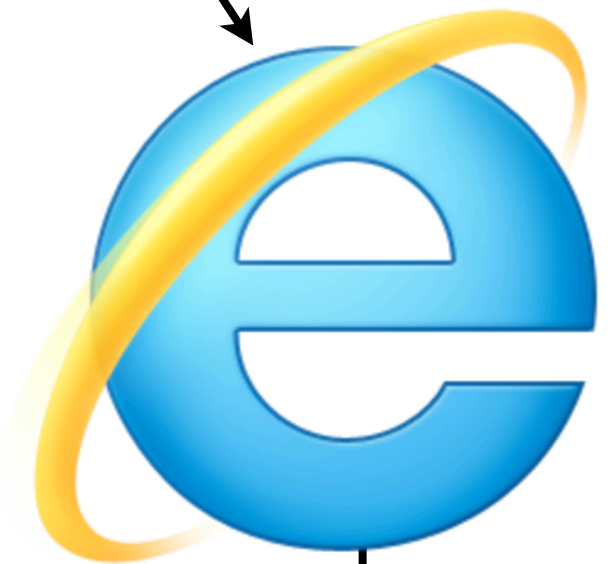
Executed on



↓
42



↓
42



↓ Produce
53

Input
Generator

Generates

```
function foo() { ... }  
...  
bar();
```

Test Input

Executed on

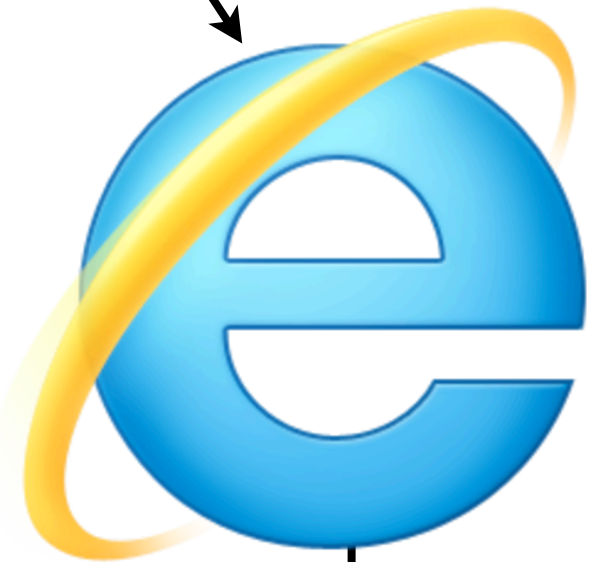


42



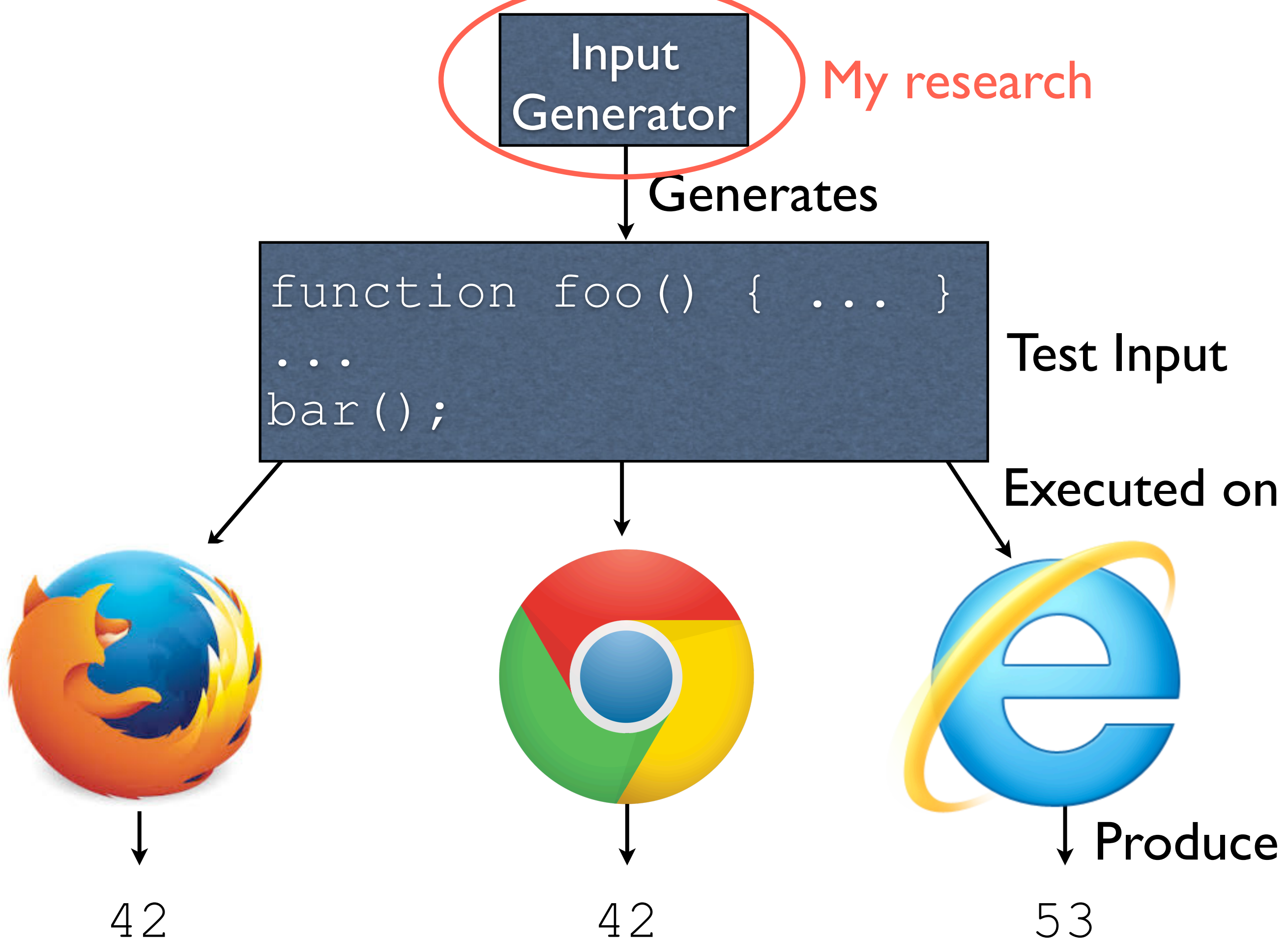
42

Mismatch: bug



53

Produce



Outline

- Background
- **Research problem**
- Applications
 - Data Structure Generation
 - Generating JavaScript Programs with Known Runtime Behaviors
 - Testing Rust's Typechecker
 - Testing SMT Solvers
- Conclusion

Consider the JavaScript snippet from before

```
function foo() { ... }  
...  
bar();
```

Any random input for “...” forms a possible test

```
function foo() { qw }  
asdf  
bar();
```

Any random input for “...” forms a possible test

```
function foo() { qw }  
asdf  
bar();
```

...but there is no telling if this will be a *good* test

Any random input for “...” forms a possible test

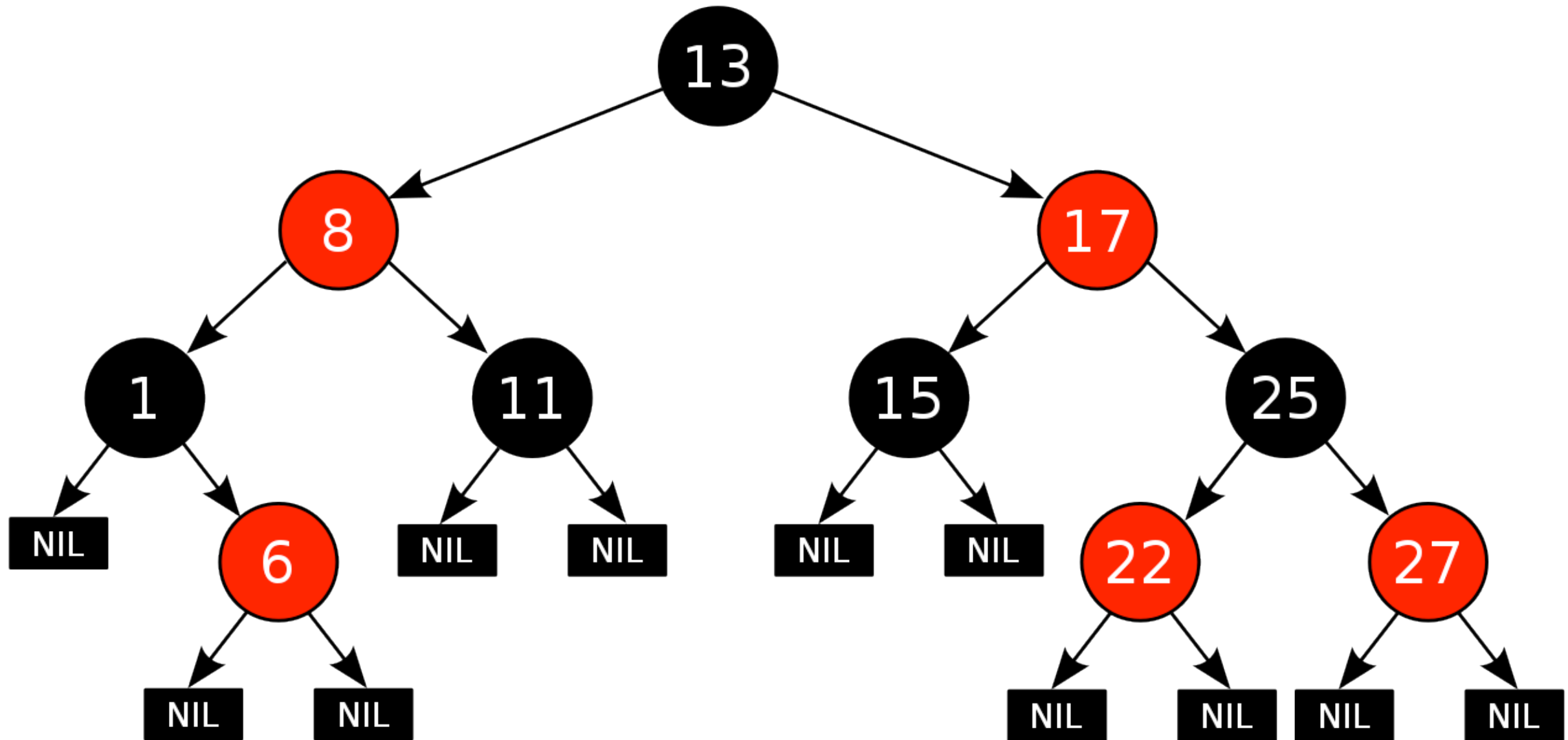
```
function foo() { qw }  
asdf  
bar();
```

...but there is no telling if this will be a *good* test

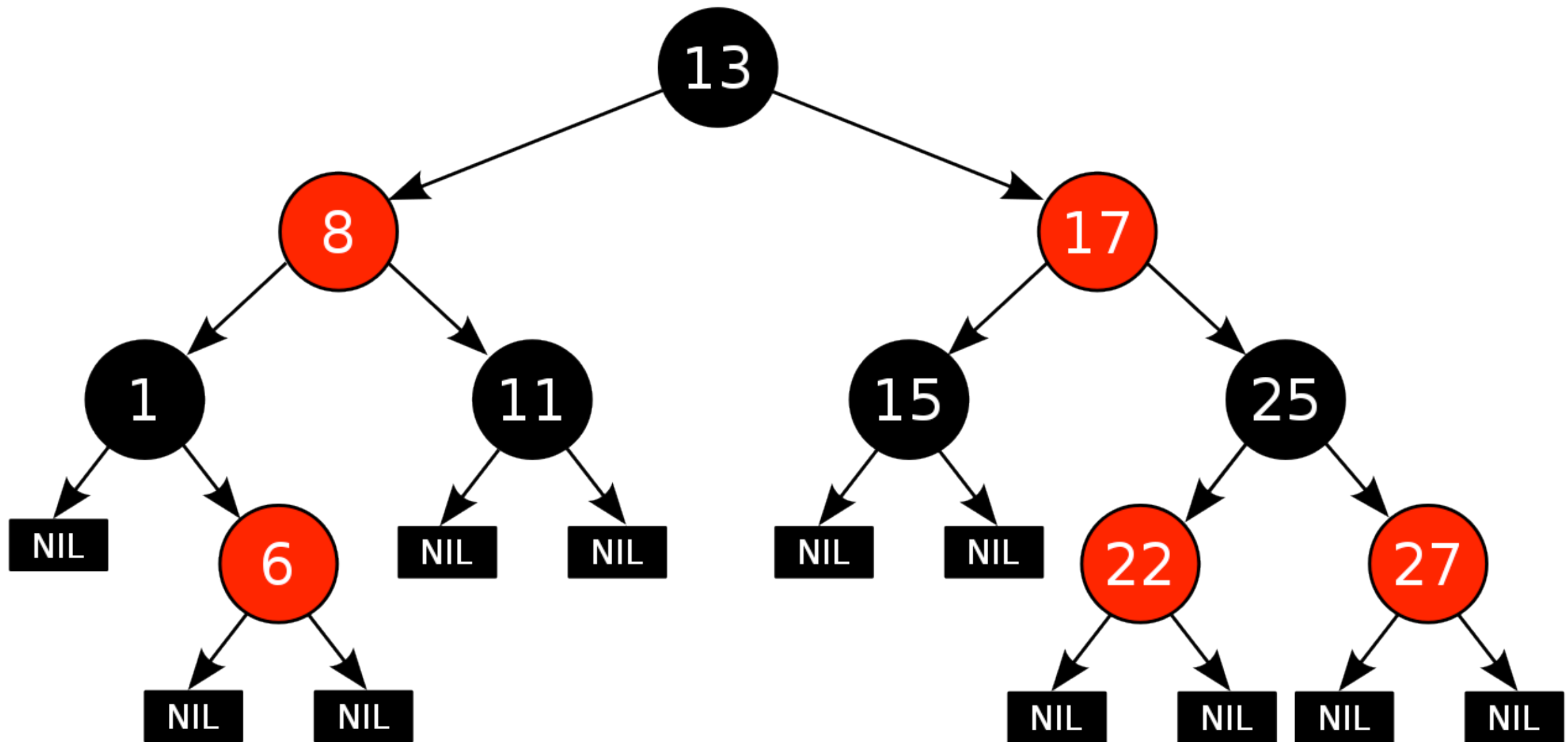
(In this case, at best a test of variable lookup)

More complex example: a library that manipulates red-black trees

More complex example: a library that manipulates red-black trees

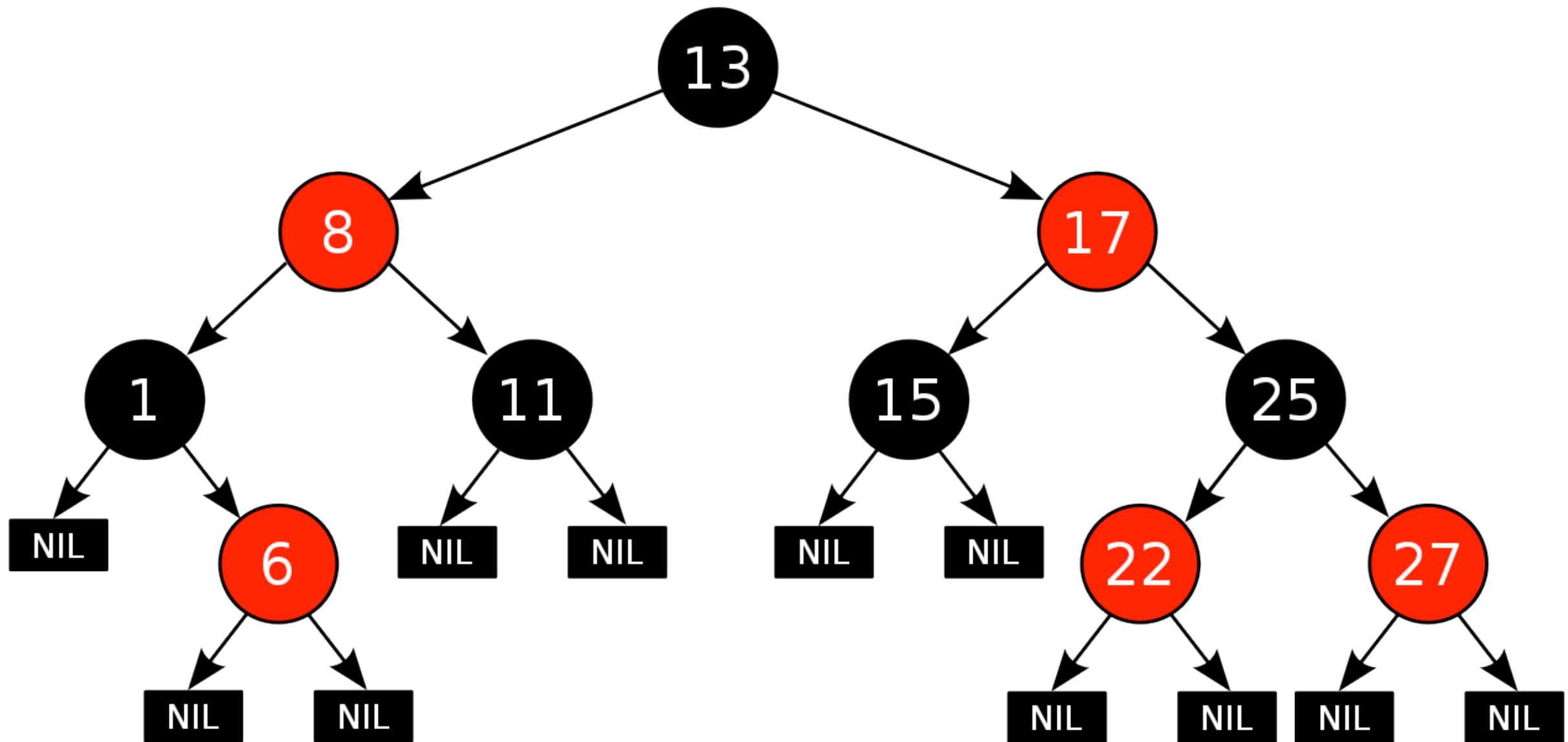


More complex example: a library that manipulates red-black trees



Intuitively, we need at least a significant portion of tests which consist of valid red-black trees

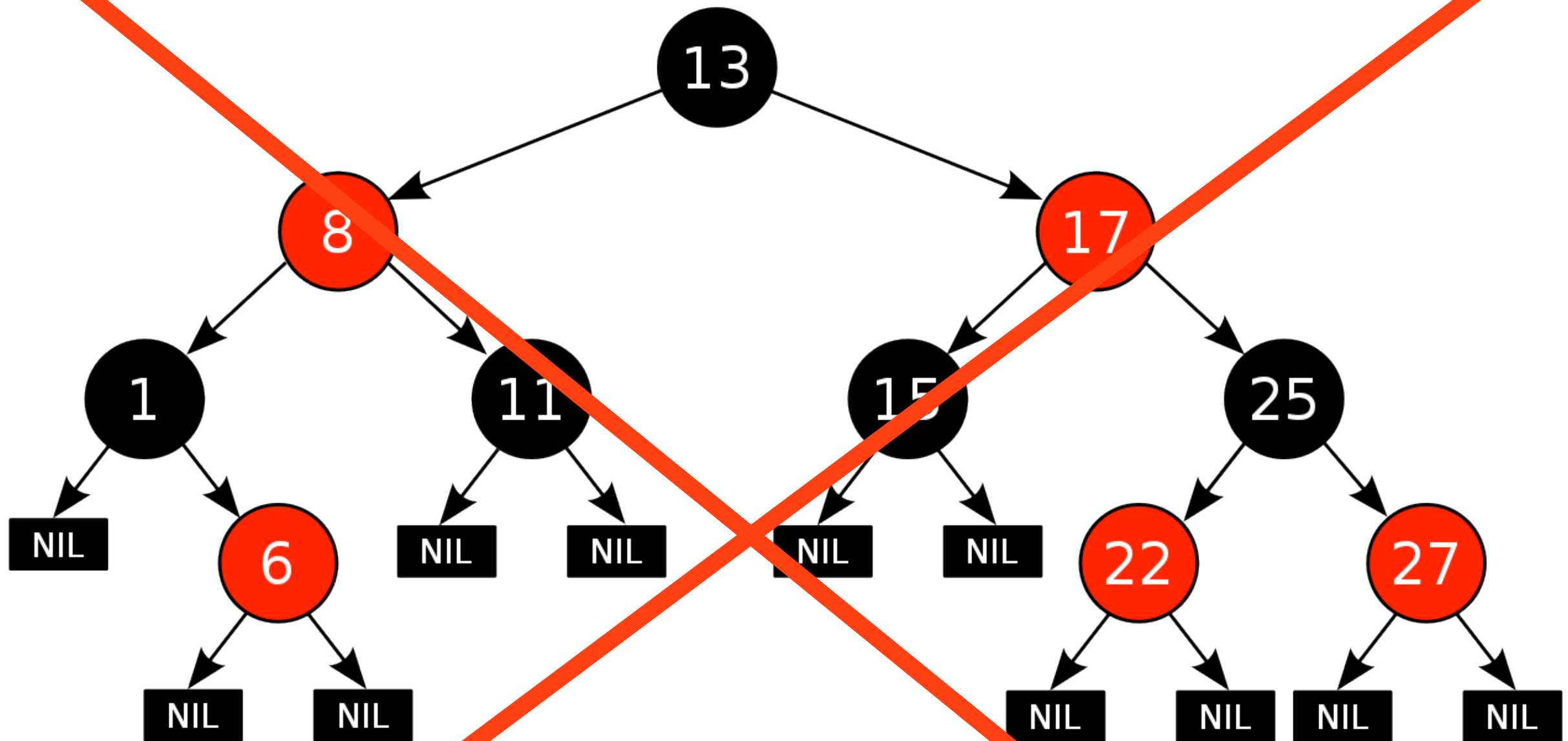
More complex example: a library that manipulates red-black trees



Intuitively, we need at least a significant portion of tests which consist of valid red-black trees

Hard problem!

More complex example: a library that manipulates red-black trees



Intuitively, we need at least a significant portion of tests which consist of valid red-black trees

Still too trivial!

Too trivial?

- Arbitrary red-black trees aren't bad tests, but they aren't good tests, either
- More interesting: valid red-black trees of depth $\leq D$, containing values between 0 and K , **which will rebalance on the insertion of value v**

Too trivial?

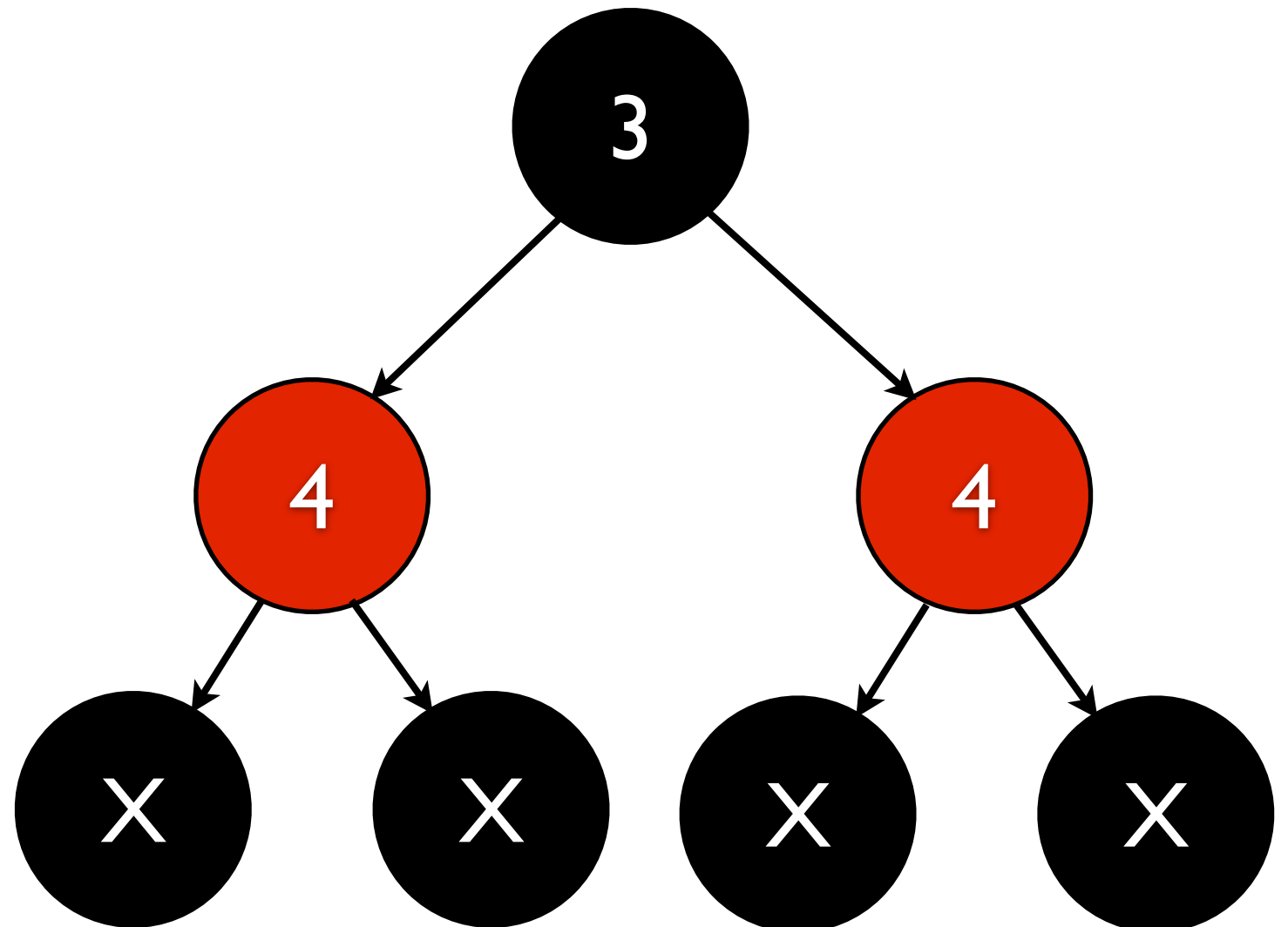
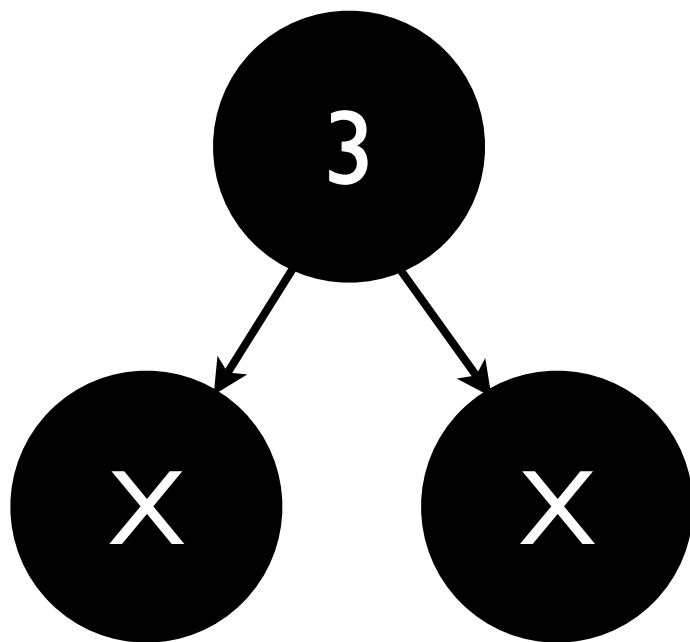
- Arbitrary red-black trees aren't bad tests, but they aren't good tests, either
- More interesting: valid red-black trees of depth $\leq D$, containing values between 0 and K , **which will rebalance on the insertion of value v**
 - Tests are much more specific
 - Targeted tests for finding bugs in insertion and rebalancing
 - Significantly more difficult
 - Unique to my work

Key Insights (I)

- Valid test inputs can be described as solutions to systems of logical constraints

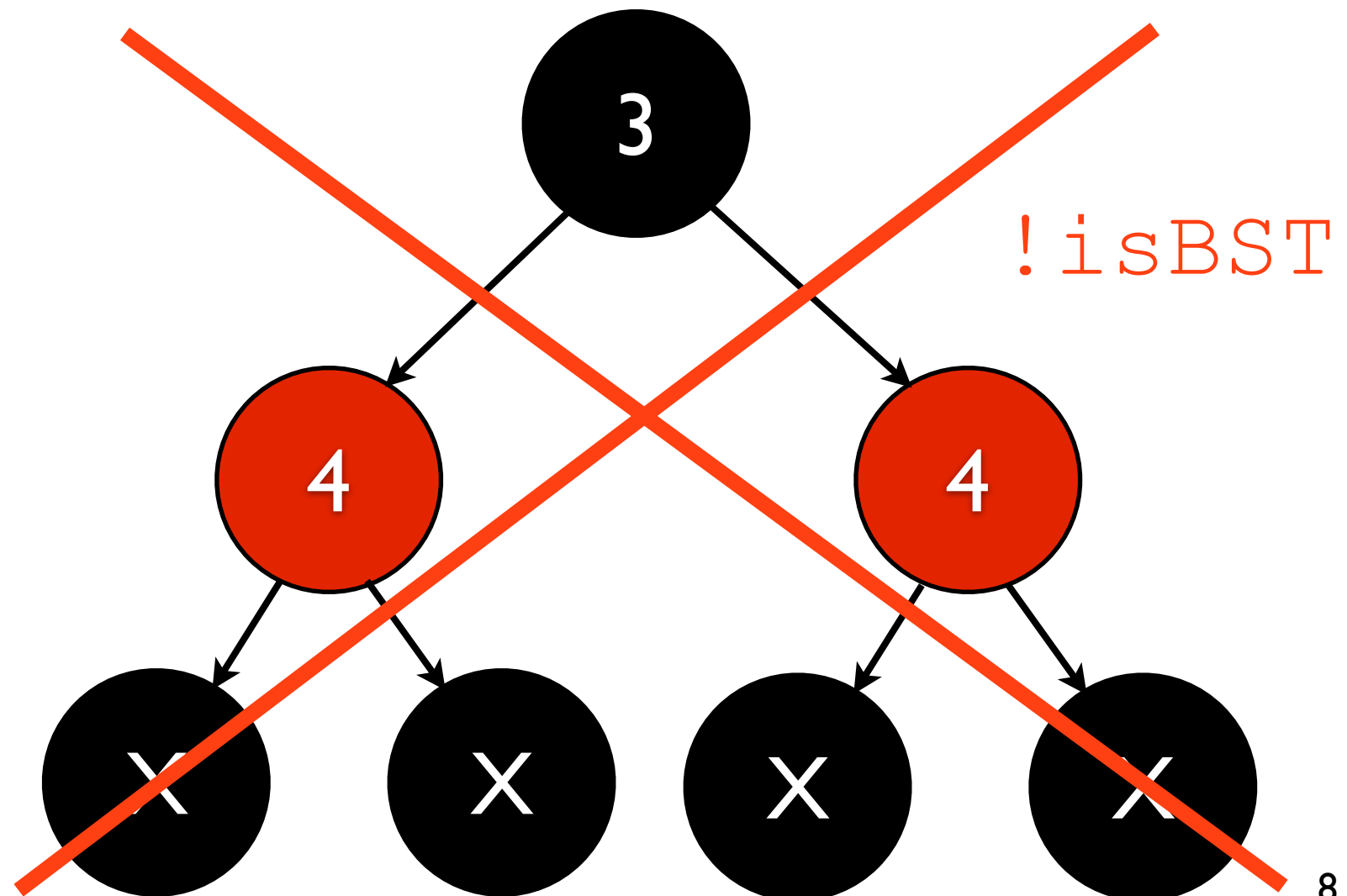
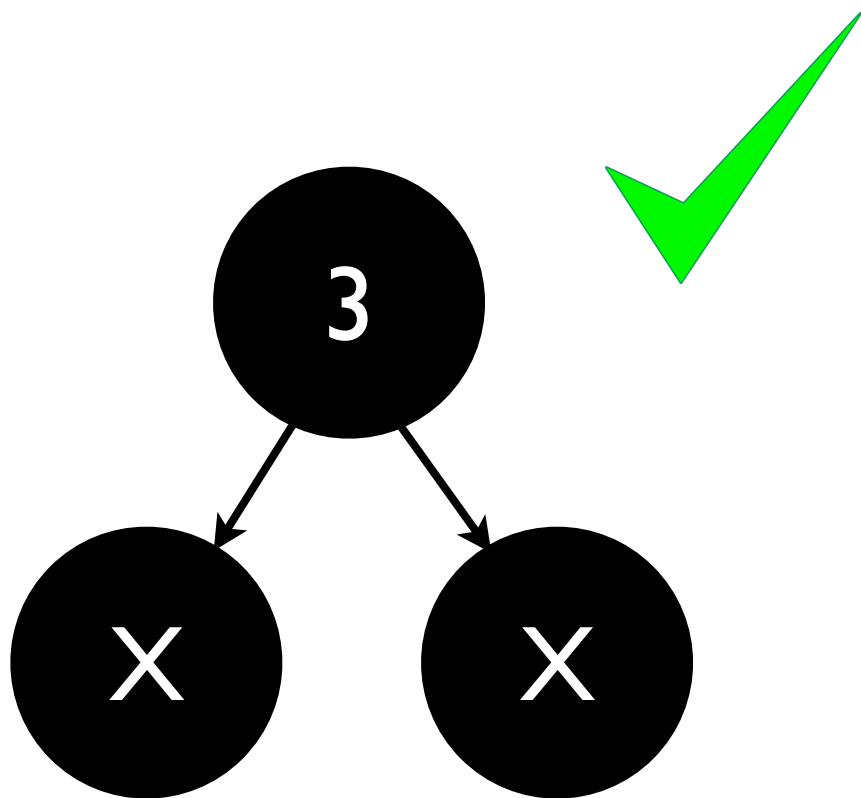
Key Insights (I)

- Valid test inputs can be described as solutions to systems of logical constraints



Key Insights (I)

- Valid test inputs can be described as solutions to systems of logical constraints



Key Insights (2)

- A variety of search strategies can be used to explore the space of solutions to these logical constraints

Key Insights (2)

- A variety of search strategies can be used to explore the space of solutions to these logical constraints
 - Search strategies can be **independent** of constraints
 - Search strategies are **useful** because there tend to be many, even infinitely many, solutions
 - E.g., depth-first search, random, etc.

Enter Constraint Logic Programming

- Constraint Logic Programming (CLP) is Prolog integrated with arithmetic constraint solvers
- CLP overall is viewable as a solver of logical constraints, with fine-grained control over how the constraints are solved
- Therefore, we can specify test input constraints in CLP, and use existing CLP engines to generate corresponding inputs

Digression: Why CLP?

- Why **not** SMT solvers?
 - $x > y \wedge y < z$
 - Not designed for getting *all* solutions, only *one* solution; getting all ranges from practically to actually impossible
 - Slow (testing faster than generation)
 - No / minimal control over search

Z3

Digression: Why CLP?

- Why **not** a custom constraint solver?
 - Lots of engineering needed to make it fast, which is unrelated to the test generation problem
 - Very easy to accidentally reimplement CLP

Digression: Why CLP?

- Why **not** a custom constraint solver?
 - Lots of engineering needed to make it fast, which is unrelated to the test generation problem
 - Very easy to accidentally reimplement CLP
- Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.
- Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in udit. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pages 225–234, New York, NY, USA, 2010. ACM.
- Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pages 619–630, New York, NY, USA, 2015. ACM.
- Burke Fetscher, Koen Claessen, Michal Palka, John Hughes, and Robert Bruce Findler. Making random judgements: Automatically generating well-typed terms from the definition of a type-system. ESOP 2015.

CLP Example: Binary Trees

- Binary trees, **not** *binary search* trees
 - Consist of leaves and internal nodes
 - Both are associated with one value
 - Constraint: for each element value E ,
 $\text{Min} \leq E \leq \text{Max}$
-

CLP Example: Binary Trees

- Binary trees, **not** *binary search* trees
 - Consist of leaves and internal nodes
 - Both are associated with one value
 - Constraint: for each element value E ,
 $Min \leq E \leq Max$
-

CLP Example: Binary Trees

- Binary trees, **not** *binary search* trees
 - Consist of leaves and internal nodes
 - Both are associated with one value
 - Constraint: for each element value E ,
 $Min \leq E \leq Max$
-

```
inBounds (E, Min, Max) :-  
    Min #=< E,  
    E #=< Max.
```

CLP Example: Binary Trees

- Binary trees, **not** *binary search* trees
- Consist of leaves and internal nodes
 - Both are associated with one value
- Constraint: for each element value E ,
 $Min \leq E \leq Max$

Clause - comparable to function definition

```
inBounds (E, Min, Max) :-  
    Min #=< E,  
    E #=< Max.
```


CLP Example: Binary Trees

- Binary trees, **not** *binary search* trees
- Consist of leaves and internal nodes
 - Both are associated with one value
- Constraint: for each element value E ,
 $Min \leq E \leq Max$

Clause head - comparable to function signature

```
inBounds (E, Min, Max) :-
```

```
Min #=< E,
```

```
E #=< Max.
```

CLP Example: Binary Trees

- Binary trees, **not** *binary search* trees
 - Consist of leaves and internal nodes
 - Both are associated with one value
 - Constraint: for each element value E ,
 $Min \leq E \leq Max$
-

`inBounds (E, Min, Max) :-`

`Min #=< E,`
`E #=< Max.`

Clause body - comparable to function body

CLP Example: Binary Trees

- Binary trees, **not** *binary search* trees
- Consist of leaves and internal nodes
 - Both are associated with one value
- Constraint: for each element value E ,
 $Min \leq E \leq Max$

`inBounds (E, Min, Max) :-`

`Min #=< E,`

`E #=< Max.`

Reverse implication



CLP Example: Binary Trees

- Binary trees, **not** *binary search* trees
- Consist of leaves and internal nodes
 - Both are associated with one value
- Constraint: for each element value E ,
 $Min \leq E \leq Max$

`inBounds (E, Min, Max) :-`

`Min #=< E`

`E #=< Max.`

Conjunction

(\wedge)

CLP Example: Binary Trees

- Binary trees, **not** *binary search* trees
 - Consist of leaves and internal nodes
 - Both are associated with one value
 - Constraint: for each element value E ,
 $Min \leq E \leq Max$
-

```
inBounds (E, Min, Max) :-  
    Min #=< E,  
    E #=< Max.
```

Arithmetic \leq over symbolic variables

CLP Example: Binary Trees

- Binary trees, **not** *binary search* trees
 - Consist of leaves and internal nodes
 - Both are associated with one value
 - **Constraint: for each element value E ,**
 $Min \leq E \leq Max$
-

```
inBounds (E, Min, Max) :-  
    Min #=< E,  
    E #=< Max.
```

End of clause

CLP Example: Binary Trees

- Binary trees, **not** binary search trees
- Consist of leaves and internal nodes
 - Both are associated with one value
- Constraint: for each element value E ,
 $Min \leq E \leq Max$

```
inBounds (Elem, Min, Max) :-  
  Min #=< Elem,  
  Elem #=< Max.
```

Logical meaning:

$\forall Min. \forall Elem. \forall Max.$

$inBounds(Elem, Min, Max) \Leftarrow$

$Min \leq Elem \wedge Elem \leq Max$

CLP Example: Binary Trees

- Binary trees, **not** *binary search* trees
 - Consist of leaves and internal nodes
 - Both are associated with one value
 - Constraint: for each element value E ,
 $\text{Min} \leq E \leq \text{Max}$
-

CLP Example: Binary Trees

CLP Example: Binary Trees

```
inBounds (Elem, Min, Max) :-  
    Min #=< Elem,  
    Elem #=< Max.
```

CLP Example: Binary Trees

```
inBounds (Elem, Min, Max) :-  
    Min #=< Elem,  
    Elem #=< Max.
```

```
tree (leaf (Elem), Min, Max) :-  
    inBounds (Elem, Min, Max).
```

CLP Example: Binary Trees

```
inBounds (Elem, Min, Max) :-  
    Min #=< Elem,  
    Elem #=< Max.
```

```
tree(leaf(Elem), Min, Max) :-  
    inBounds (Elem, Min, Max).
```

CLP Example: Binary Trees

```
inBounds (Elem, Min, Max) :-  
    Min #=< Elem,  
    Elem #=< Max.
```

```
tree (leaf (Elem), Min, Max) :-  
    inBounds (Elem, Min, Max).
```

CLP Example: Binary Trees

```
inBounds (Elem, Min, Max) :-  
    Min #=< Elem,  
    Elem #=< Max.
```

```
tree (leaf (Elem), Min, Max) :-  
    inBounds (Elem, Min, Max).  
tree (node (Left, Elem, Right),  
    Min, Max) :-  
    inBounds (Elem, Min, Max),  
    tree (Left, Min, Max),  
    tree (Right, Min, Max).
```

CLP Example: Binary Trees

```
inBounds (Elem, Min, Max) :-  
    Min #=< Elem,  
    Elem #=< Max.
```

```
tree (leaf (Elem), Min, Max) :-  
    inBounds (Elem, Min, Max).
```

```
tree (node (Left, Elem, Right),  
    Min, Max) :-
```

```
    inBounds (Elem, Min, Max),  
    tree (Left, Min, Max),  
    tree (Right, Min, Max).
```

CLP Example: Binary Trees

```
inBounds (Elem, Min, Max) :-  
    Min #=< Elem,  
    Elem #=< Max.
```

```
tree (leaf (Elem), Min, Max) :-  
    inBounds (Elem, Min, Max).  
tree (node (Left, Elem, Right),  
    Min, Max) :-  
    inBounds (Elem, Min, Max),  
    tree (Left, Min, Max),  
    tree (Right, Min, Max).
```


CLP Example: Binary Trees

```
inBounds (Elem, Min, Max) :-  
    Min #=< Elem,  
    Elem #=< Max.
```

```
tree (leaf (Elem), Min, Max) :-  
    inBounds (Elem, Min, Max).  
tree (node (Left, Elem, Right),  
    Min, Max) :-  
    inBounds (Elem, Min, Max),  
    tree (Left, Min, Max),  
    tree (Right, Min, Max).
```

CLP Example: Binary Trees

```
inBounds (Elem, Min, Max) :-  
    Min #=< Elem,  
    Elem #=< Max.
```

```
tree (leaf (Elem), Min, Max) :-  
    inBounds (Elem, Min, Max).  
tree (node (Left, Elem, Right),  
    Min, Max) :-  
    inBounds (Elem, Min, Max),  
    tree (Left, Min, Max),  
    tree (Right, Min, Max).
```

CLP Example: Binary Trees

- Generating valid trees can be done like so:

```
?- tree(Tree, 0, 3),  
   writeln(Tree),  
   fail.
```

CLP Example: Binary Trees

- Generating valid trees can be done like so:

```
?- tree(Tree, 0, 3),  
    writeln(Tree),  
    fail.
```

Query

CLP Example: Binary Trees

- Generating valid trees can be done like so:

```
?- tree(Tree, 0, 3),  
   writeln(Tree),  
   fail.
```

Generate a tree with `Min = 0`
and `Max = 3`; bind it to `Tree`

CLP Example: Binary Trees

- Generating valid trees can be done like so:

```
?- tree(Tree, 0, 3),  
   writeln(Tree),  
   fail.
```

Write out the tree

CLP Example: Binary Trees

- Generating valid trees can be done like so:

```
?- tree(Tree, 0, 3),  
   writeln(Tree),  
   fail.
```

Trigger backtracking to occur to generate another tree.

Intuitively, `Tree` is nondeterministically bound to all possible trees.

Outline

- Background
- Research problem
- **Applications**
 - **Data Structure Generation**
 - Generating JavaScript Programs with Known Runtime Behaviors
 - Testing Rust's Typechecker
 - Testing SMT Solvers
- Conclusion

Data Structure (DS) Generation

- Applied CLP to the generation of complex data structures, along with particular *variants of interest* for testing
 - Variants form a strict subset of the space, and each DS had its own variant
- Most of the data structures were novel to our work, along with all of the variants
 - Intentionally wanted to push the limit

Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees Covered in related work
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
 - Red-black trees
 - Array-based heaps (priority queues)
 - ANI images (via grammars)
 - Skip lists
 - Splay trees
 - B-trees
- Novel to our work

Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Example Variant (Same as Previously Described)

- Valid red-black trees of depth $\leq D$, containing values between 0 and K , **which will rebalance on the insertion of value v**

Example Variant (Same as Previously Described)

- Valid red-black trees of depth $\leq D$, containing values between 0 and K , **which will rebalance on the insertion of value v**

```
isBST(Tree1)  $\wedge$   
isRedBlackTree(Tree1)  $\wedge$   
callsRebalance(  
    insert(Tree1, v, Tree2))
```

Example Variant (Same as Previously Described)

- Valid red-black trees of depth $\leq D$, containing values between 0 and K , **which will rebalance on the insertion of value v**

```
isBST(Tree1)  $\wedge$   
isRedBlackTree(Tree1)  $\wedge$   
callsRebalance(  
    insert(Tree1, v, Tree2))
```

Example Variant (Same as Previously Described)

- Valid red-black trees of depth $\leq D$, containing values between 0 and K , **which will rebalance on the insertion of value v**

```
isBST(Tree1)  $\wedge$   
isRedBlackTree(Tree1)  $\wedge$   
callsRebalance(  
    insert(Tree1, v, Tree2))
```

Example Variant (Same as Previously Described)

- Valid red-black trees of depth $\leq D$, containing values between 0 and K , **which will rebalance on the insertion of value v**

`isBST(Tree1) \wedge`

`isRedBlackTree(Tree1) \wedge`

`callsRebalance(
 insert(Tree1, v, Tree2))`

Example Variant (Same as Previously Described)

- Valid red-black trees of depth $\leq D$, containing values between 0 and K , **which will rebalance on the insertion of value v**

Requires reasoning about both rebalancing and insertion;

`isBST(Tree1) \wedge intuitively large state space`

`isRedBlackTree(Tree1) \wedge`

`callsRebalance(
insert(Tree1, v, Tree2))`

Example Variant (Same as Previously Described)

- Valid red-black trees of depth $\leq D$, containing values between 0 and K , **which will rebalance on the insertion of value v**

Naive approach: generate all possible trees, and filter those

`isBST(Tree1) \wedge that are related via insert`

`isRedBlackTree(Tree1) \wedge`

`callsRebalance(`

`insert(Tree1, v , Tree2))`

Evaluation

- Tested all aforementioned data structures and their special variants on Korat, UDITA, and CLP (using GNU Prolog)
- Measured **how quickly** all data structures within certain bounds (small, medium, large) could be generated, with a 30 minute timeout
 - Quicker generation means more time testing and less time generating

Seconds
(lower is
better)

Small Bounds

Korat UDITA CLP

1800.00000

1350.00025

900.00050

450.00075

0.00100

Lists

Red-Black

Heaps

Image

Skip

Splay

B-Trees



Seconds
(lower is
better)

Small Bounds

Korat UDITA CLP

UDITA Is Extremely Slow

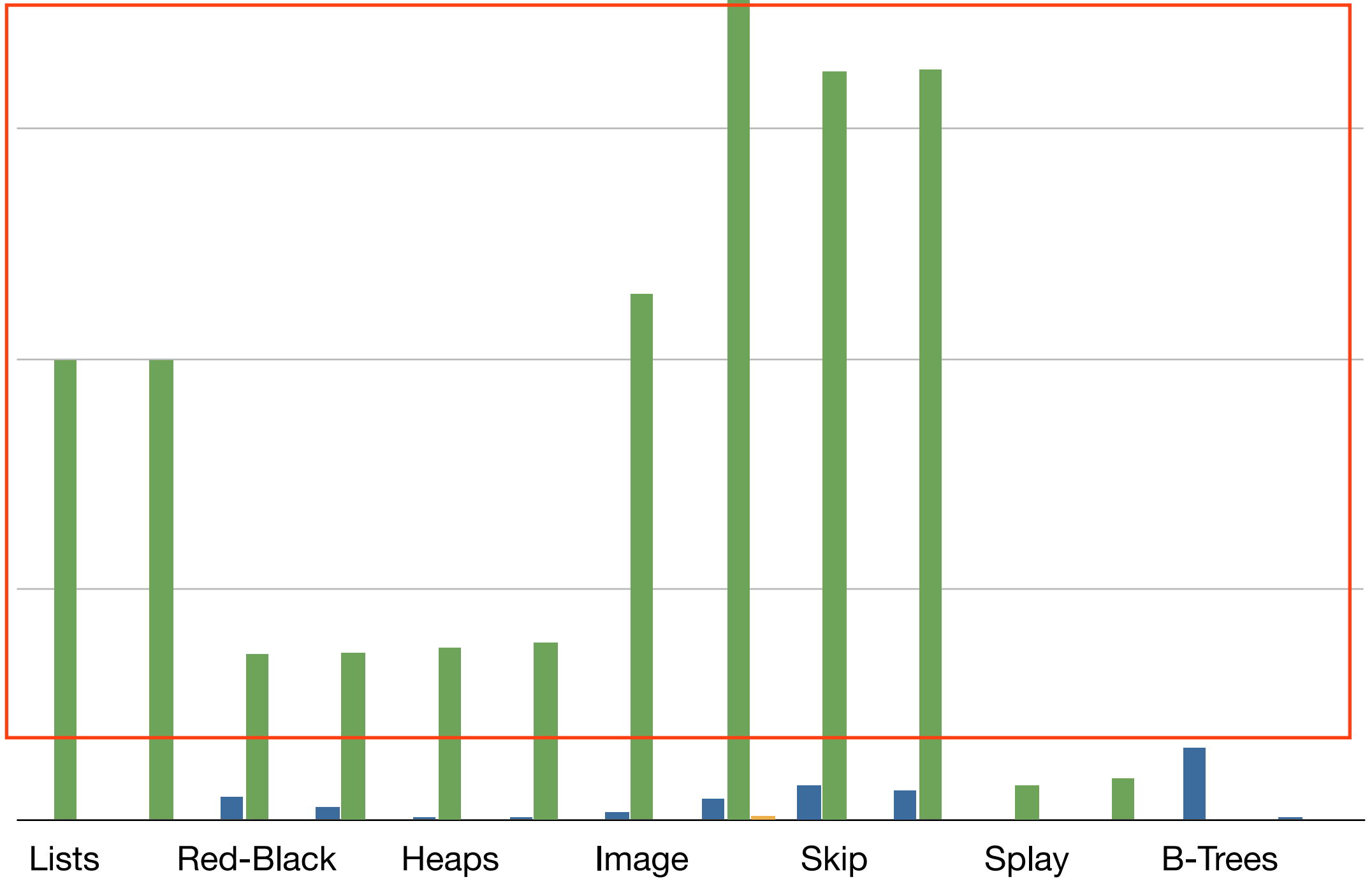
1800.00000

1350.00025

900.00050

450.00075

0.00100

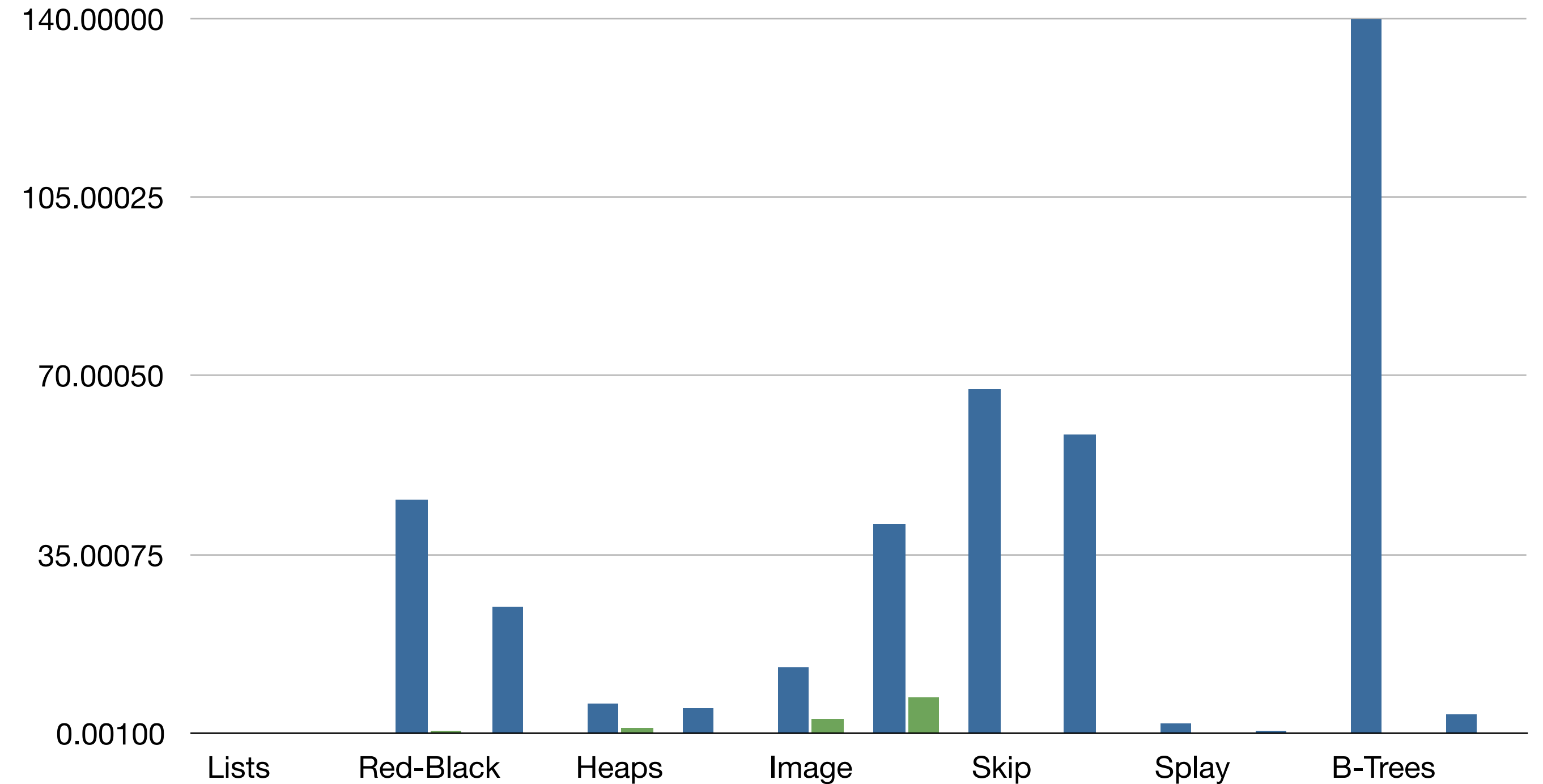


Seconds
(lower is
better)

Small Bounds

Korat

CLP



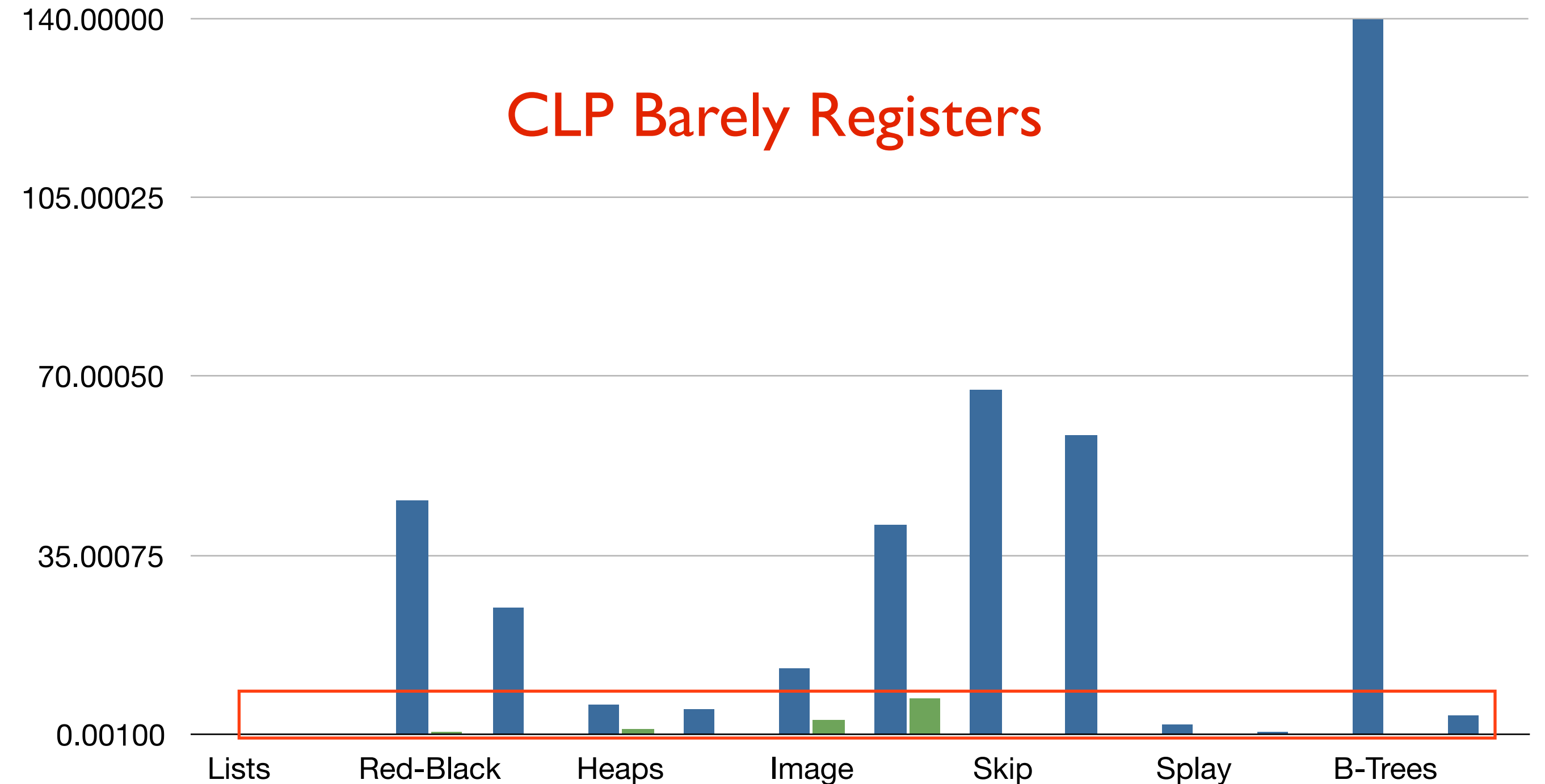
Seconds
(lower is
better)

Small Bounds

Korat

CLP

CLP Barely Registers



Medium Bounds

- UDITA times out on everything
- Korat times out on 5 / 14 experiments
- CLP is generally $\sim 30\times$ - $1,000\times$ faster
- For B-trees, Korat and UDITA both timeout, but CLP completes within **a single millisecond**, ultimately thanks to the **capability to control search**
 - Internally, they took the naive strategy

Large Bounds

- Korat and UDITA timeout on everything
- Depending on the data structure, CLP takes between ~70 seconds and just under 30 minutes

Outline

- Background
- Research problem
- **Applications**
 - Data Structure Generation
 - **Generating JavaScript Programs with Known Runtime Behaviors**
 - Testing Rust's Typechecker
 - Testing SMT Solvers
- Conclusion

Generating Programs

- Bulk of existing literature is focused on *stochastic grammars*
 - Randomly walk over a language's grammar, producing syntactically valid programs as a result

Generating Programs

- Bulk of existing literature is focused on *stochastic grammars*
 - Randomly walk over a language's grammar, producing syntactically valid programs as a result

$$e \in \textit{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$$

Generating Programs

- Bulk of existing literature is focused on *stochastic grammars*
 - Randomly walk over a language's grammar, producing syntactically valid programs as a result

$$e \in \mathit{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$$

Generating Programs

- Bulk of existing literature is focused on *stochastic grammars*
 - Randomly walk over a language's grammar, producing syntactically valid programs as a result

$$e \in \textit{ArithExp} ::= \boxed{n \in \mathbb{N}} \mid e_1 + e_2$$

Generating Programs

- Bulk of existing literature is focused on *stochastic grammars*
 - Randomly walk over a language's grammar, producing syntactically valid programs as a result

$e \in \textit{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$

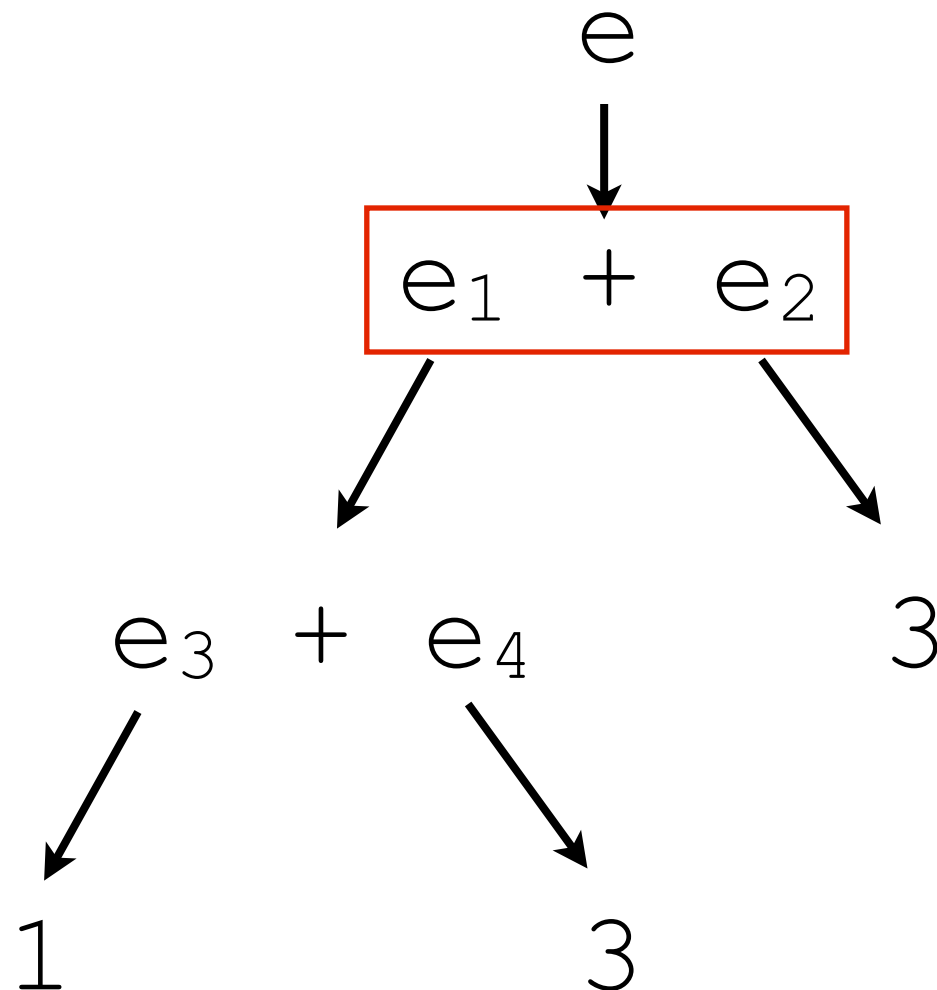
Generating Programs

- Bulk of existing literature is focused on *stochastic grammars*
 - Randomly walk over a language's grammar, producing syntactically valid programs as a result

$$e \in \textit{ArithExp} ::= n \in \mathbb{N}^{0.6} \mid e_1 + e_2^{0.4}$$

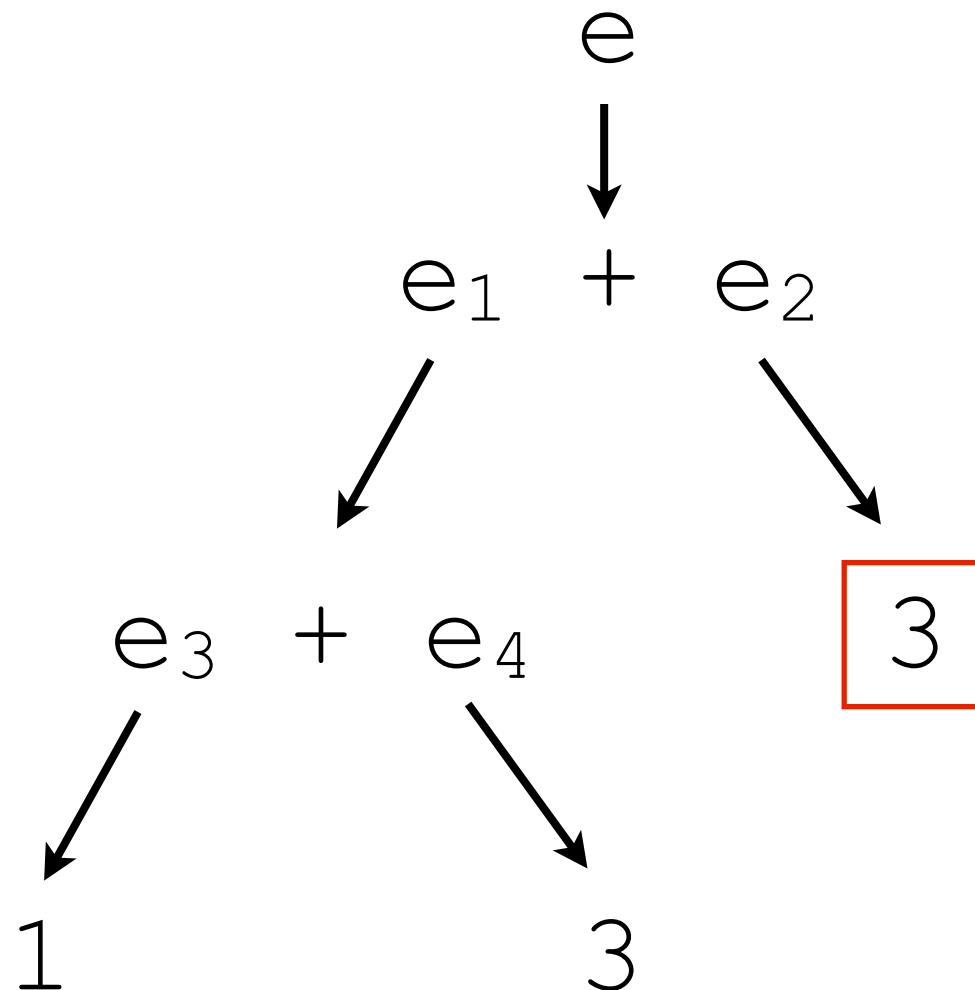
Example Derivation

$$e \in \textit{ArithExp} ::= n \in \mathbb{N}^{0.6} \mid \boxed{e_1 + e_2}^{0.4}$$



Example Derivation

$$e \in \textit{ArithExp} ::= \boxed{n \in \mathbb{N}}^{0.6} \mid e_1 + e_2^{0.4}$$



Problems with Stochastic Grammars

- All you get is syntactic validity
 - No idea what programs do
 - Programs are not generally well-typed
 - Difficult to test particular components (e.g., specifically code generation)
 - Only configuration is by tuning probabilities

Enter CLP

- Generating syntactically valid programs is easy...

Syntactic Validity

$e \in \textit{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$

Syntactic Validity

$$e \in \textit{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$$

Syntactic Validity

$e \in \textit{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$

```
arithExp (num (N) ) :-  
  INTMIN #=< N,  
  N #=< INTMAX.
```

Syntactic Validity

$e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$

`arithExp (num (N)) :-`
`INTMIN #=< N,`
`N #=< INTMAX.`

Syntactic Validity

$e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$

`arithExp (num (N)) :-
 INTMIN #=< N,
 N #=< INTMAX.`

Syntactic Validity

$e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$

```
arithExp (num (N) ) :-  
  INTMIN #=< N,  
  N #=< INTMAX.
```

Syntactic Validity

$e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$

```
arithExp (num (N) ) :-  
  INTMIN #=< N,  
  N #=< INTMAX.
```

Syntactic Validity

$e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$

```
arithExp (num (N) ) :-  
  INTMIN #=< N,  
  N #=< INTMAX.
```

```
arithExp (add (E1, E2) ) :-  
  arithExp (E1) ,  
  arithExp (E2) .
```

Syntactic Validity

$e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$

```
arithExp (num (N) ) :-  
  INTMIN #=< N,  
  N #=< INTMAX.
```

```
arithExp (add (E1, E2) ) :-  
  arithExp (E1) ,  
  arithExp (E2) .
```

Syntactic Validity

$e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$

```
arithExp (num (N) ) :-  
  INTMIN #=< N,  
  N #=< INTMAX.
```

```
arithExp (add (E1, E2) ) :-  
  arithExp (E1) ,  
  arithExp (E2) .
```

Syntactic Validity

$e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$

```
arithExp (num (N) ) :-  
  INTMIN #=< N,  
  N #=< INTMAX.
```

```
arithExp (add (E1, E2) ) :-  
  arithExp (E1),  
  arithExp (E2).
```

Beyond Syntax

- For example, programs which evaluate to a particular value
 - A semantic property
- Involves writing an *evaluator* for the language in CLP

Expressions that Evaluate to 7

```
eval(num(N), N).  
eval(add(E1, E2), N) :-  
    eval(E1, N1),  
    eval(E2, N2),  
    N #= N1 + N2.  
  
% same arithExp from before  
evalsto7(E) :-  
    arithExp(E),  
    eval(E, 7).
```

Expressions that Evaluate to 7

```
eval(num(N), N).
```

```
eval(add(E1, E2), N) :-
```

```
    eval(E1, N1),
```

```
    eval(E2, N2),
```

```
    N #= N1 + N2.
```

```
% same arithExp from before
```

```
evalsto7(E) :-
```

```
    arithExp(E),
```

```
    eval(E, 7).
```

Expressions that Evaluate to 7

```
eval(num(N), N).
```

```
eval(add(E1, E2), N) :-
```

```
    eval(E1, N1),
```

```
    eval(E2, N2),
```

```
    N #= N1 + N2.
```

```
% same arithExp from before
```

```
evalsto7(E) :-
```

```
    arithExp(E),
```

```
    eval(E, 7).
```

Expressions that Evaluate to 7

```
eval(num(N), N).
```

```
eval(add(E1, E2), N) :-
```

```
    eval(E1, N1),
```

```
    eval(E2, N2),
```

```
    N #= N1 + N2.
```

```
% same arithExp from before
```

```
evalsto7(E) :-
```

```
    arithExp(E),
```

```
    eval(E, 7).
```

Expressions that Evaluate to 7

```
eval(num(N), N).  
eval(add(E1, E2), N) :-  
    eval(E1, N1),  
    eval(E2, N2),  
    N #= N1 + N2.  
  
% same arithExp from before  
evalsto7(E) :-  
    arithExp(E),  
    eval(E, 7).
```

Expressions that Evaluate to 7

```
eval(num(N), N).  
eval(add(E1, E2), N) :-  
    eval(E1, N1),  
    eval(E2, N2),  
    N #= N1 + N2.  
  
% same arithExp from before  
evalsto7(E) :-  
    arithExp(E),  
    eval(E, 7).
```

Expressions that Evaluate to 7

```
eval(num(N), N).
```

```
eval(add(E1, E2), N) :-
```

```
    eval(E1, N1),
```

```
    eval(E2, N2),
```

```
    N #= N1 + N2.
```

```
% same arithExp from before
```

```
evalsto7(E) :-
```

```
    arithExp(E),
```

```
    eval(E, 7).
```

Expressions that Evaluate to 7

```
eval(num(N), N).
```

```
eval(add(E1, E2), N) :-
```

```
    eval(E1, N1),
```

```
    eval(E2, N2),
```

```
    N #= N1 + N2.
```

```
% same arithExp from before
```

```
evalsTo7(E) :-
```

```
    arithExp(E),
```

```
    eval(E, 7).
```


Expressions that Evaluate to 7

```
eval(num(N), N).
```

```
eval(add(E1, E2), N) :-
```

```
    eval(E1, N1),
```

```
    eval(E2, N2),
```

```
    N #= N1 + N2.
```

```
% same arithExp from before
```

```
evalsTo7(E) :-
```

```
    arithExp(E),
```

```
    eval(E, 7).
```

Scaling Up

- This central idea was applied to generating JavaScript programs with known runtime behaviors
- Compared the generation rate to that of a finely-tuned stochastic grammar designed for the same thing
 - Stochastic grammar probabilities were tuned to try to generate programs with certain behaviors

Scaling Up

- For JavaScript, generate programs which:
 - avoid dereferencing `null`: CLP ~3.8x faster
 - stress integer optimizations: CLP ~7.8x faster
 - utilize `with` and higher-order functions in problematic ways: CLP ~3.1 **million x** faster
 - utilize prototype-based inheritance: CLP **infinitely** faster (stochastic grammars never generated such a program within the five-minute timeframe)

Outline

- Background
- Research problem
- **Applications**
 - Data Structure Generation
 - Generating JavaScript Programs with Known Runtime Behaviors
 - **Testing Rust's Typechecker**
 - Testing SMT Solvers
- Conclusion

Why Rust?

- A real language with a rapidly growing user base (over 6,600 packages available currently)
- A sophisticated type system with important guarantees (e.g., memory safety without GC)
- No formal semantics, or even an informal specification
 - Worked closely with Rust development team

Why Rust's Typechecker?

- Well-typed Rust programs are memory safe
- If the typechecker fails to flag an ill-typed program, then there is a *silent loss* of memory safety guarantees
- Most complex language component at the time

**Unique Challenge: There
is Only One Rust**

Input
Generator

Generates

```
function foo() { ... }  
...  
bar();
```

Test Input

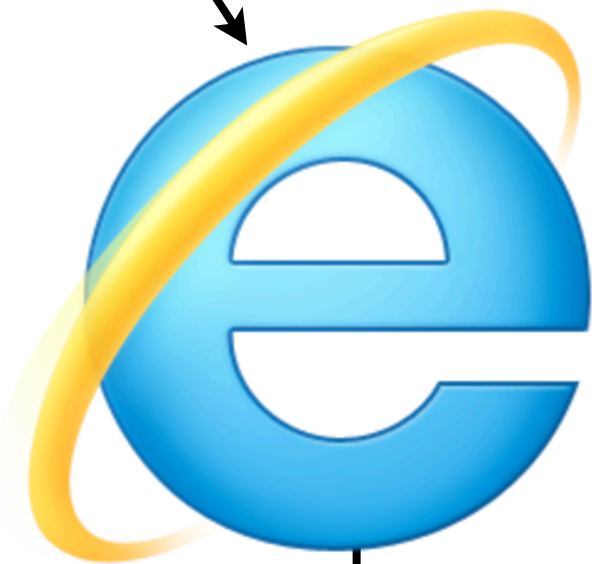
Executed on



42



42



53

Produce

Input
Generator

Generates

```
fn foo() { ... }  
fn bar() { ... }  
fn main() { ... }
```

Test Input

Executed on



well-typed

Input
Generator

Generates

```
fn foo() { ... }  
fn bar() { ... }  
fn main() { ... }
```

Test Input

Executed on



Nothing to compare to!

well-typed

Solution

- Generate tests which behave predictably ahead of time, and check that the underlying system agrees with the predication
- In other words, generate tests which we know to be well-typed or ill-typed, and make sure Rust agrees
- Requires understanding Rust's type system

Rust's Type System

- Rust has typeclasses, parametric polymorphism, generics, and **affine types** for guaranteeing memory safety statically
- Handling affine types properly requires:
 - Symbolic arithmetic constraints
 - Constraints on type variables

Rust's Type System

- Rust has typeclasses, parametric polymorphism, generics, and **affine types** for guaranteeing memory safety statically
 - Handling affine types properly requires:
 - Symbolic arithmetic constraints
 - Constraints on type variables
- Never before generated

Rust's Type System

- Rust has typeclasses, parametric polymorphism, generics, and **affine types** for guaranteeing memory safety statically
- Handling affine types properly requires:
 - Symbolic arithmetic constraints
 - Constraints on type variables **Unavailable in existing systems**

Rust's Type System

- Rust has typeclasses, parametric polymorphism, generics, and **affine types** for guaranteeing memory safety statically
- Handling affine types properly requires:
 - Symbolic arithmetic constraints
 - **Constraints on type variables**

Never attempted
before

Rust's Type System

- Rust has typeclasses, parametric polymorphism, generics, and **affine types** for guaranteeing memory safety statically
- Handling affine types properly requires:
 - Symbolic arithmetic constraints
 - Constraints on type variables

Altogether, must embed a specialized constraint solver for handling these features in CLP itself

Well-Typed for Testing

- For testing purposes, well-typed programs are not particularly interesting
 - If compiler rejects a well-typed program, the programmer gets annoyed
 - Types as analysis: rejection of a well-typed program is a precision issue

Ill-Typed for Testing

- More interesting for testing purposes: ill-typed programs
 - If a compiler accepts an ill-typed program, we get a silent loss of guarantees
 - For Rust, this means programs are not necessarily memory-safe, defeating the entire purpose of the language
 - Types as analysis: accepting an ill-typed program is a soundness issue

Generating Ill-Typed Programs

- Naive approach: generate syntactically valid programs and discard those that happen to be well-typed
 - Relatively efficient (most will be ill-typed)
 - Most programs are *obviously* ill-typed (multiple type errors; a typechecker need only spot one, so this masks bugs)

Generating Ill-Typed Programs

- Better approach: generate programs which are *almost* well-typed
- Intuitively, negate a single premise in a typing rule, leading to programs which are ill-typed by construction, but **only** with respect to the single negated premise
 - Results in highly targeted tests
- This idea is novel, and this was the first attempt to generate anything intentionally ill-typed

Rust Testing Results

- Able to generate ~2,300 programs **per second**
 - Versus ~2 per second compared to preexisting techniques on a simpler language CLP is over 1,100x **faster** for a significantly **more complex language**
- Found 18 issues; developers considered 14 of these bugs
 - Included one **specification level** bug

Outline

- Background
- Research problem
- **Applications**
 - Data Structure Generation
 - Generating JavaScript Programs with Known Runtime Behaviors
 - Testing Rust's Typechecker
 - **Testing SMT Solvers**
- Conclusion

SMT Solvers

- Used for solving constraints specified in the SMT-LIB language, in a similar vein as CLP
- Crucially important in *software verification*; that is, *proving* some code does the right thing
- Solvers can be buggy too
- Buggy solvers can mean faulty proofs

Testing Approach

- Lots of details
- Work is currently in submission
- Basic idea: test with well-typed SMT-LIB formulas which are known to be logically satisfiable or unsatisfiable ahead of time

Testing Results

- 24 bugs have been found across a number of solvers
- **Every solver tested had at least one correctness bug, including Z3**
- Included a **specification bug** which required communication with the standards committee

Outline

- Background
- Research problem
- Applications
 - Data Structure Generation
 - Generating JavaScript Programs with Known Runtime Behaviors
 - Testing Rust's Typechecker
 - Testing SMT Solvers
- **Conclusion**

Conclusion

- CLP is applicable to a number of test problems
- CLP is capable of generating very complex tests in a high-performance fashion
- No need to write lots of test generation code with CLP

Demo