

Advances in Grammar Mining and Testing

Andreas Zeller
CISPA / Saarland University

Saarbrücken







Scientific excellence in fundamental research
50,000,000 €/year • 500+ researchers

Fuzzing

Random Testing at the System Level

[;x1-GPZ+wcckc]; ,N9J+?#6^6\ e?]9lu2_%'4GX"0VUB[E/r
~fApu6b8<{ %siq8Zh.6{V,hr?; {Ti.r3PIxMMMv6{xS^+'Hq!Ax B"YXRS@!
Kd6;wtAMeffWM(`|J_<1~o}z3K(CCzRH JIIvHz> *. \>JrlU32~eGP?
lR=bF3+;y\$3lodQ**<B89!5" W2fK*vE7v{ ')KC-i,c{<[~m!]o;{. ' }Gj\ (X}**
EtYetrbpbY@aGZ1{P!AZU7x#4(Rtn!q4nCwqol^y6}0|
Ko=*JK~;zMKV=9Nai:wxu{J&UV#HaU)*BiC<),`+t*gka<W=Z.
%T5WGHZpI30D< Pq>&]BS6R&j ?#tP7iaV} - }` \ ?[_ [Z^LBMPG-
FKj '\xwuZ1=Q`^`5,\$N\$Q@[!CuRzJ2D|vBy!^zkhdf3C5PAkR?V hn|
3='i2Qx]D\$qs40`1@fevnG'2\11Vf3piU37@55ap\zIyl"'f,
\$ee,J4Gw:cgNKLie3nx9(`efSlg6#[K" @WjhZ}r[Scun&sBCS,T[/
vY'pduwgzDlVNy7'rnzxNwI)(ynBa>%|b`;`9fG]P_0hdG~\$@6
3]KAeEnQ7lU)3Pn,0)G/6N-wyzj/MTd#A;r

Fuzzing

Random Testing at the System Level



Fuzzer

UNIX utilities

“ab’d&gfdffff”

grep • sh • sed ...

25%–33%

Grammar Fuzzing

- Suppose you want to test a *parser* – to compile and execute a program
- To get deep into the program, you need *syntactically correct inputs*



Parser

LangFuzz (2012)



- Fuzz tester for JavaScript and other languages
- Uses a full-fledged *grammar* to generate inputs
- Uses grammar to *parse existing inputs*

JavaScript Grammar

If Statement

IfStatement^{full} ⇒

| **if** ParenthesizedExpression Statement^{full}
| **if** ParenthesizedExpression Statement^{noShortIf} **else** Statement^{full}

IfStatement^{noShortIf} ⇒ **if** ParenthesizedExpression Statement^{noShortIf} **else** Statement^{noShortIf}

Switch Statement

SwitchStatement ⇒

| **switch** ParenthesizedExpression { }
| **switch** ParenthesizedExpression { CaseGroups LastCaseGroup }

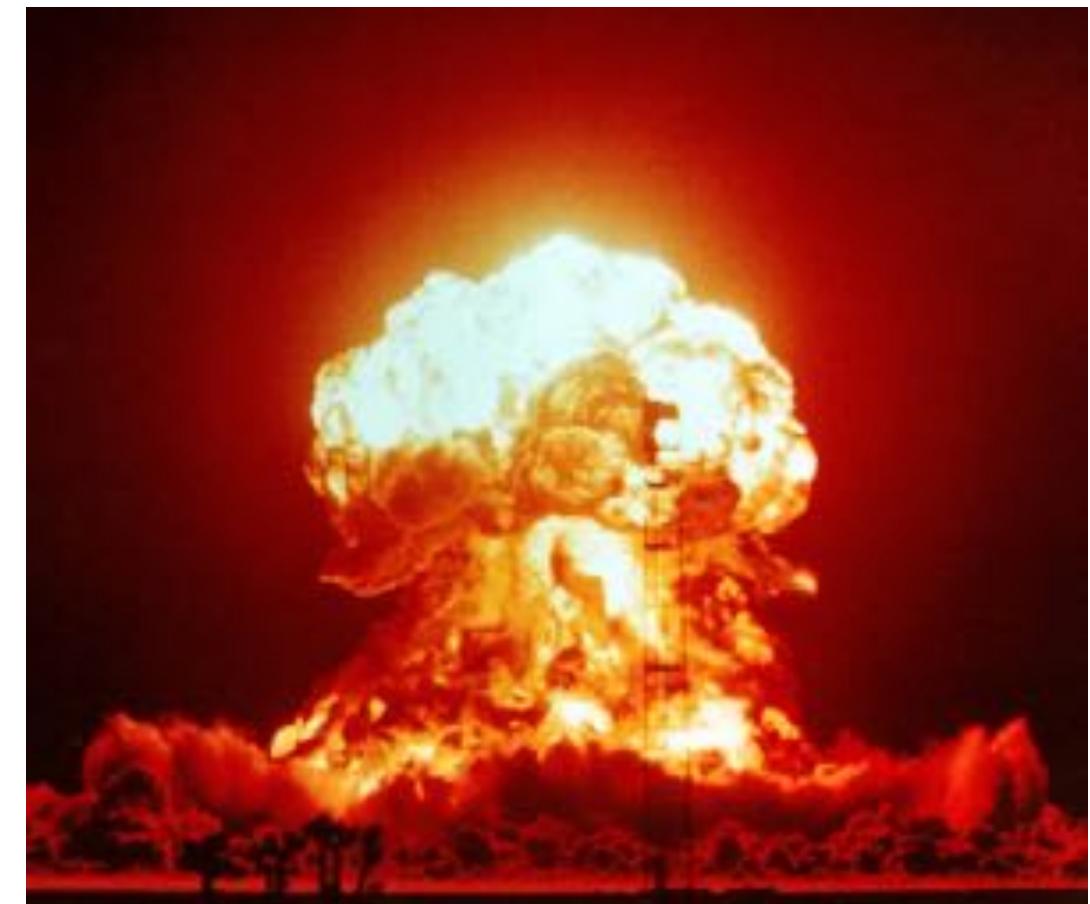
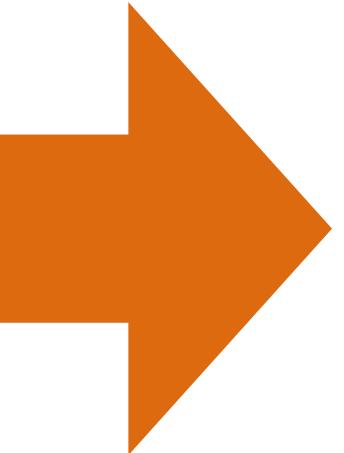
CaseGroups ⇒

«empty»
| CaseGroups CaseGroup

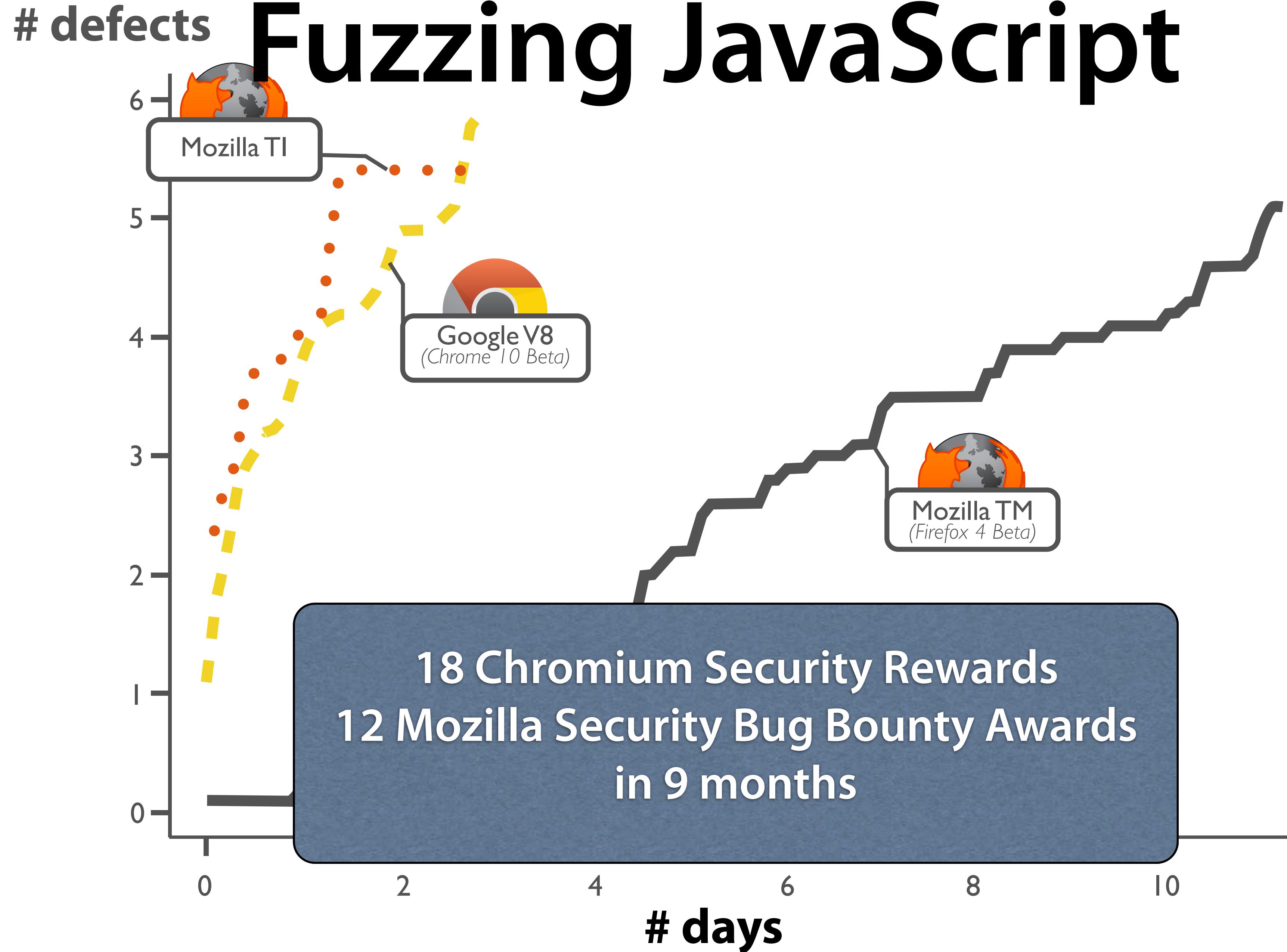
CaseGroup ⇒ CaseGuards BlockStatementsPrefix

A Generated Input

```
1 var haystack = "foo";  
2 var re_text = "^foo";  
3 haystack += "x";  
4 re_text += "(x)";  
5 var re = new RegExp(re_text);  
6 re.test(haystack);  
7 RegExp.input = Number();  
8 print(RegExp.$1);
```



Fuzzing JavaScript



Learning Grammars

If Statement

IfStatement^{full} \Rightarrow

if *ParenthesizedExpress*

 | **if** *ParenthesizedExpress*

IfStatement^{noShortIf} \Rightarrow **if** *Pa*

Switch Statement

SwitchStatement \Rightarrow

switch *ParenthesizedEx*

 | **switch** *ParenthesizedEx*

CaseGroups \Rightarrow

 «empty»

 | *CaseGroups CaseGroup*

CaseGroup \Rightarrow *CaseGuards BlockStatementsPrefix*



else *Statement^{noShortIf}*

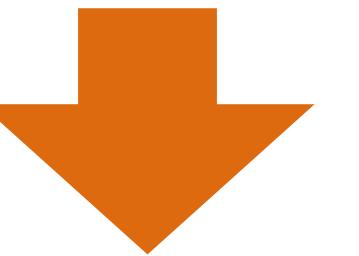
p }

Learning Grammars

- Let us characterize program behavior via its **input/output language**
- Assume I/O is a stream of characters (symbols)
- Assume we can characterize this stream via a **formal language** – regular expressions, grammars
- We want to **learn** such a language from the program

Learning Grammars

`http://user:pass@www.google.com:80/path`



Program

Learning Grammars

`http://user:pass@www.google.com:80/path`

`http`

– protocol

Learning Grammars

http://user:pass@www.google.com:80/path

http – protocol

www.google.com – host name

Learning Grammars

`http://user:pass@www.google.com:80/path`

`http` – protocol

`www.google.com` – host name

`80` – port

Learning Grammars

http://user:pass@www.google.com:80/path

http – protocol

www.google.com – host name

80 – port

user pass – login

Learning Grammars

http://user:pass@www.google.com:80/path

http – protocol

www.google.com – host name

80 – port

user pass – login

path – page request

Learning Grammars

http://user:pass@www.google.com:80/path

http	– protocol
www.google.com	– host name
80	– port
user pass	– login
path	– page request
:// : @ : /	– terminals

Learning Grammars

http://user:pass@www.google.com:80/path

http

– protocol

www.google.com

– host name

80

– port

user pass

– login

path

– page request

:// : @ : /

– terminals



processed in
different
functions

stored in
different
variables

Tracking Input

We *track input characters* throughout program execution:

1. *Dynamic tainting* labels all characters read (and derived values) with their origin
2. *Recognizing inputs* checks string variables whether they hold input fragments (simpler)

Grammar Inference

- Start with grammar $\$START ::= \textit{input}$

$\$START ::= \textcolor{blue}{\texttt{http://user:pass@www.google.com:80/path#ref}}$

Grammar Inference

- For each $(var, value)$ we find during execution, where $value$ is a substring of $input$:
 1. Replace all occurrences of $value$ by $\$VAR$
 2. Add a new rule $\$VAR ::= value$

```
$START ::= http://user:pass@www.google.com:80/path#ref
```

```
fragment = 'ref'  
url = '/path'  
path = '/path'  
scheme = 'http'  
netloc = 'user:pass@www.google.com:80'
```

Grammar Inference

- For each $(var, value)$ we find during execution,
where $value$ is a substring of $input$:
 1. Replace all occurrences of $value$ by $\$VAR$
 2. Add a new rule $\$VAR ::= value$

```
$START ::= http://$NETLOC/path#ref  
$NETLOC ::= user:pass@www.google.com:80
```

```
fragment = 'ref'  
url = '/path'  
path = '/path'  
scheme = 'http'
```

Grammar Inference

- For each $(var, value)$ we find during execution, where $value$ is a substring of $input$:
 1. Replace all occurrences of $value$ by $\$VAR$
 2. Add a new rule $\$VAR ::= value$

```
$START ::= $SCHEME://$NETLOC/path#ref
$NETLOC ::= user:pass@www.google.com:80
$SCHEME ::= http
```

```
fragment = 'ref'
url = '/path'
path = '/path'
```

Grammar Inference

- For each $(var, value)$ we find during execution,
where $value$ is a substring of $input$:
 1. Replace all occurrences of $value$ by $\$VAR$
 2. Add a new rule $\$VAR ::= value$

```
$START ::= $SCHEME://$NETLOC$path#ref
$NETLOC ::= user:pass@www.google.com:80
$SCHEME ::= http
$PATH ::= /path
```

```
fragment = 'ref'
url = '/path'
```

Grammar Inference

- For each $(var, value)$ we find during execution, where $value$ is a substring of $input$:
 1. Replace all occurrences of $value$ by $\$VAR$
 2. Add a new rule $\$VAR ::= value$

```
$START ::= $SCHEME://$NETLOC$PATH#$FRAGMENT
$NETLOC ::= user:pass@www.google.com:80
$SCHEME ::= http
$PATH ::= /path
$FRAGMENT ::= ref
```

```
url = '/path'
```

Grammar Inference

- For each $(var, value)$ we find during execution,
where $value$ is a substring of $input$:
 1. Replace all occurrences of $value$ by $\$VAR$
 2. Add a new rule $\$VAR ::= value$

```
$START ::= $SCHEME://$NETLOC$PATH#$FRAGMENT
$NETLOC ::= user:pass@www.google.com:80
$SCHEME ::= http
$PATH ::= $URL
$FRAGMENT ::= ref
$URL ::= /path
```

Demo

AUTOGRAM

AUTOGRAM: a grammar miner for Java programs

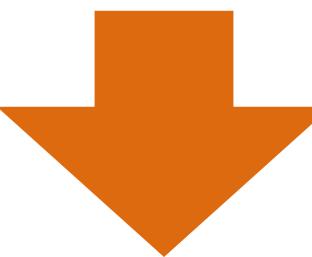
Uses *active learning* to infer

- repetitions
- optional parts
- common elements (numbers, identifiers...)

Höschele, Zeller: "Mining Input Grammars from Dynamic Taints", ASE 2016

URLs

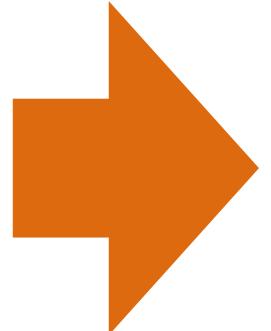
`http://user:password@www.google.com:80/command?foo=bar&lorem=ipsum#fragment`
`http://www.guardian.co.uk/sports/worldcup#results`
`ftp://bob:12345@ftp.example.com/oss/debian7.iso`



```
URL ::= PROTOCOL '://' AUTHORITY PATH ['?' QUERY] ['#' REF]
AUTHORITY ::= [USERINFO '@'] HOST [':' PORT]
PROTOCOL ::= 'http' | 'ftp'
USERINFO ::= /[a-z]+:[a-z]+/
HOST ::= /[a-z.]+/
PORT ::= '80'
PATH ::= /\[/a-z0-9.\/]*/
QUERY ::= 'foo=bar&lorem=ipsum'
REF ::= /[a-z]+/
```

INI Files

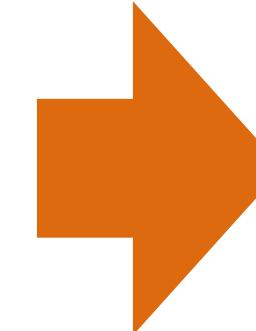
```
[Application]
Version = 0.5
WorkingDir = /tmp/mydir/
[User]
User = Bob
Password = 12345
```



```
INI ::= LINE+
LINE ::= SECTION_LINE '\r'
      | OPTION_LINE ['\r']
SECTION_LINE ::= '[' KEY ']'
OPTION_LINE ::= KEY ' = ' VALUE
KEY ::= /[a-zA-Z]*/
VALUE ::= /[a-zA-Z0-9\/]/
```

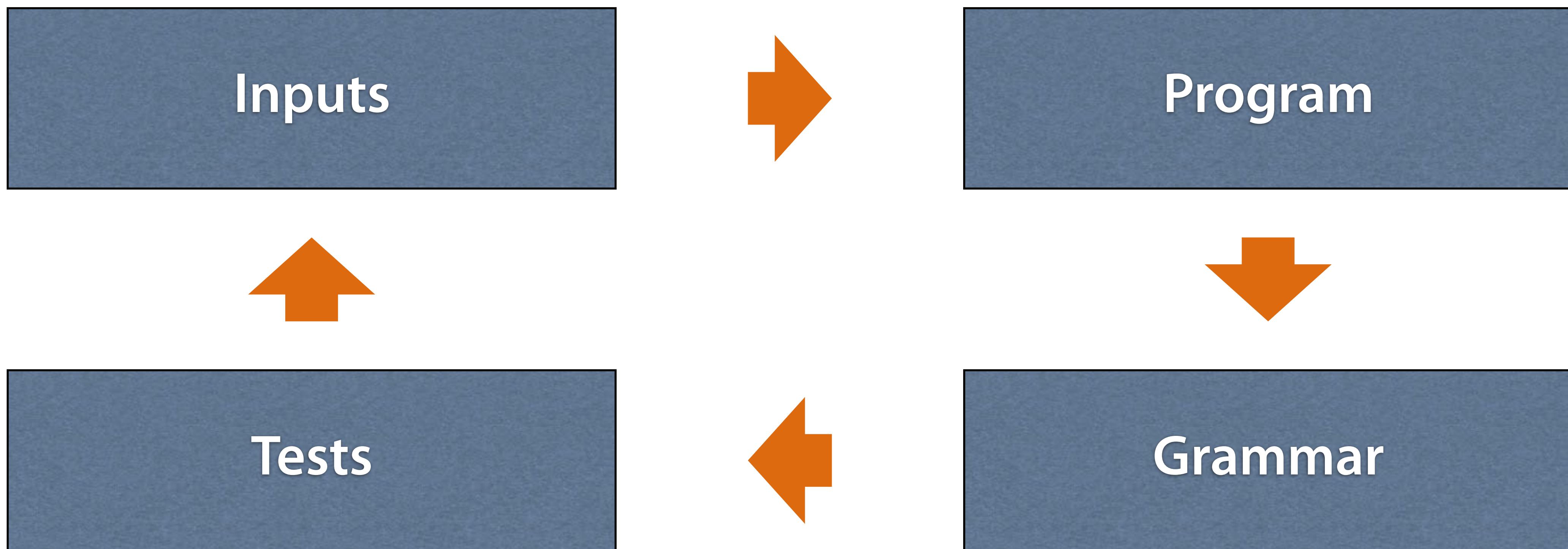
JSON Input

```
{  
  "v": true,  
  "x": 25,  
  "y": -36,  
  ...  
}
```



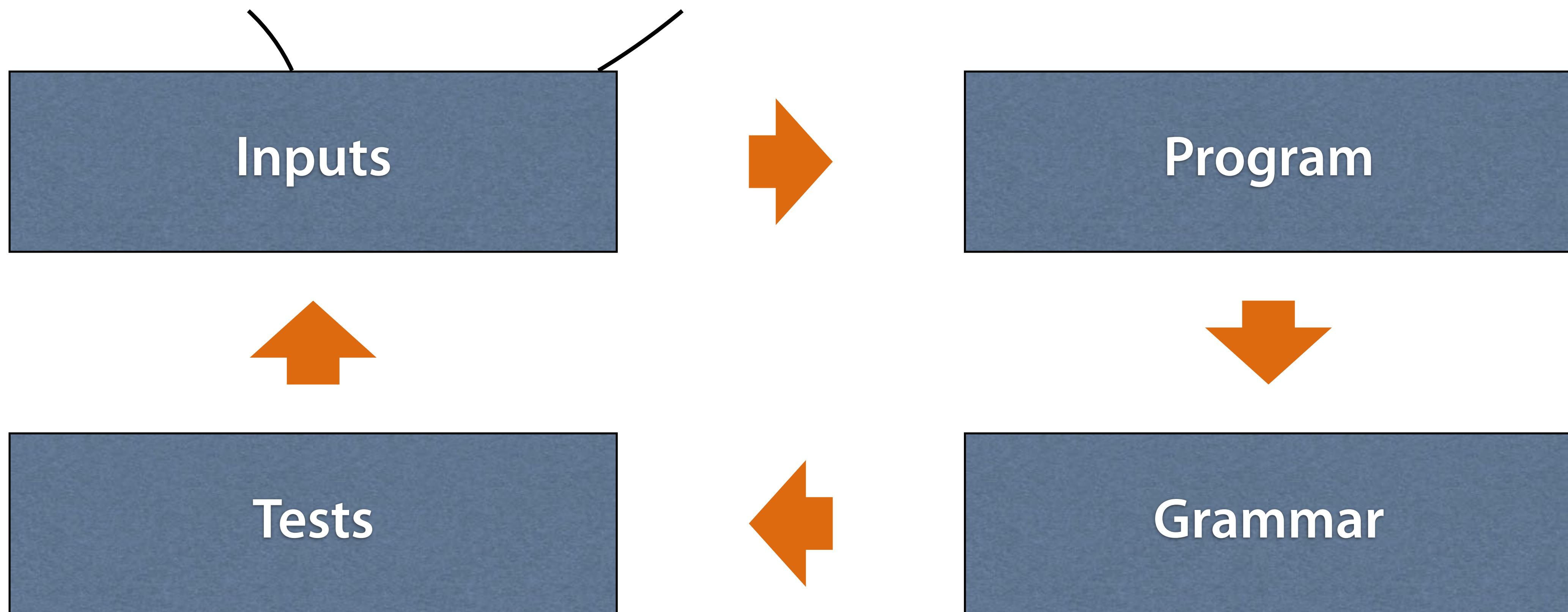
```
JSON ::= VALUE  
VALUE ::= JSONOBJECT | ARRAY | STRINGVALUE |  
       TRUE | FALSE | NULL | NUMBER  
TRUE ::= 'true'  
FALSE ::= 'false'  
NULL ::= 'null'  
NUMBER ::= '-' /[0-9]+/  
STRINGVALUE ::= '"' INTERNALSTRING '"'  
INTERNALSTRING ::= /[a-zA-Z0-9 ]+/  
ARRAY ::= '['  
        [VALUE ',' VALUE]+]  
      ']'  
JSONOBJECT ::= '{'  
           [STRINGVALUE ':' VALUE  
            ',', STRINGVALUE ':' VALUE]  
           '+']  
         '}'
```

Testing with Mined Grammars

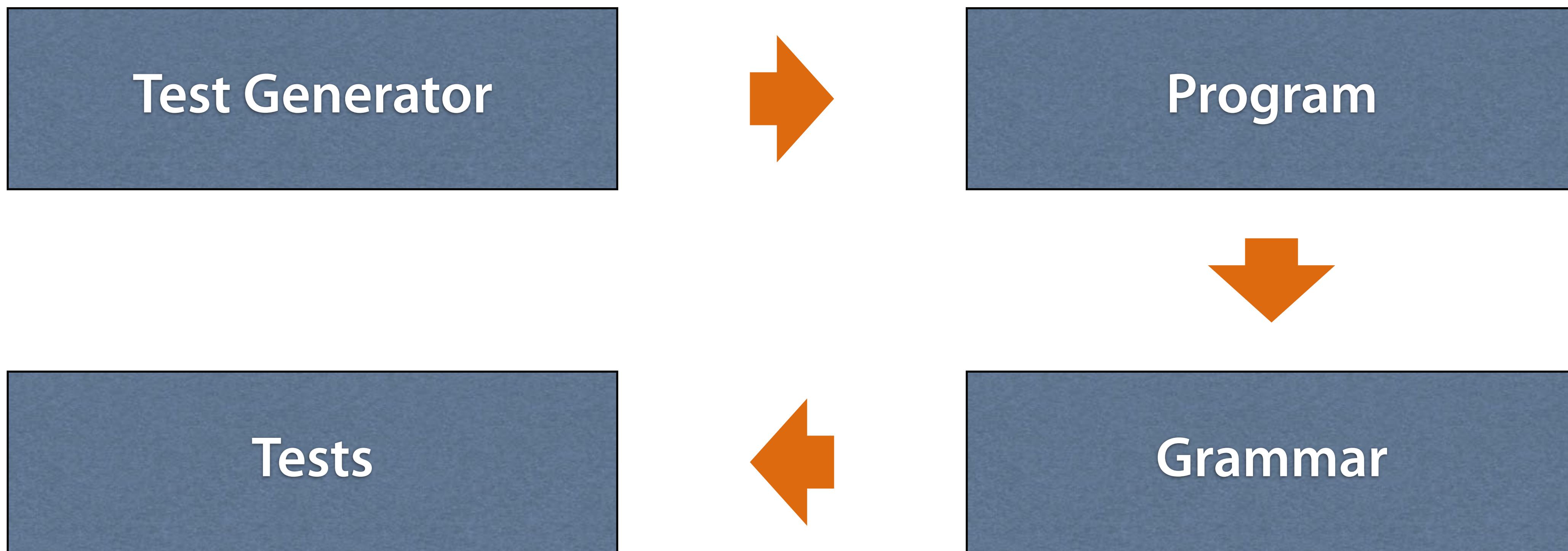


Testing with Mined Grammars

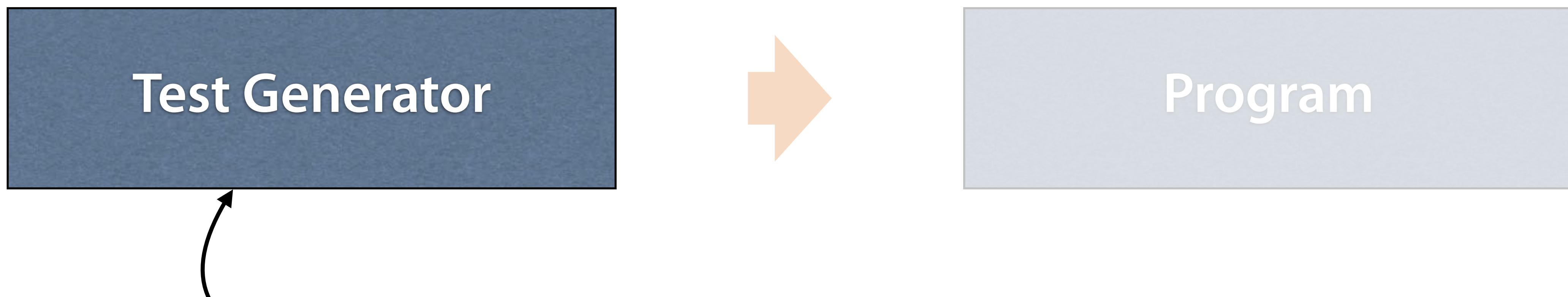
may not be available introduce bias



Sample-Free Grammar Learning

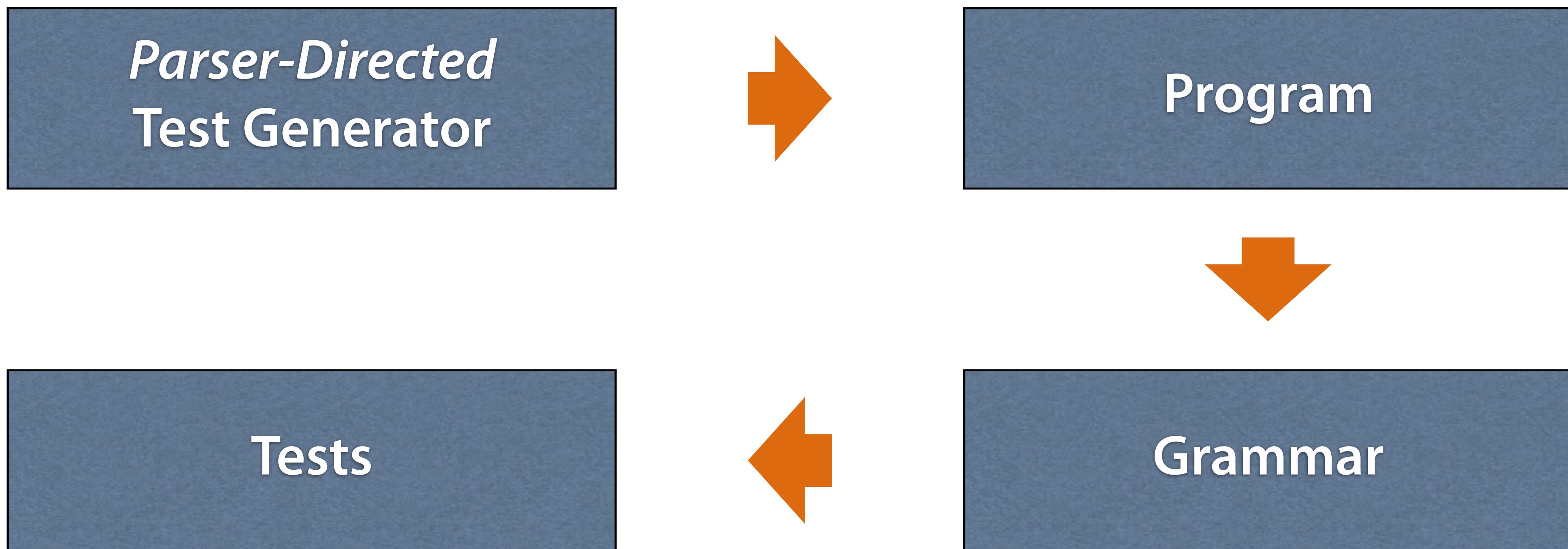


Sample-Free Grammar Learning

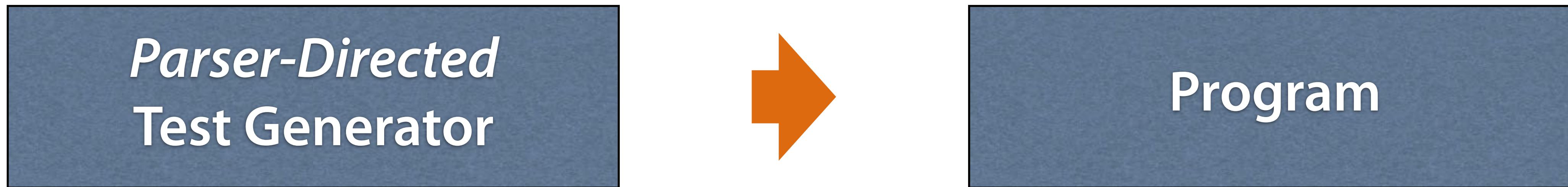


But this is what we want to build in the first place!

Sample-Free Grammar Learning



Dynamic Checks

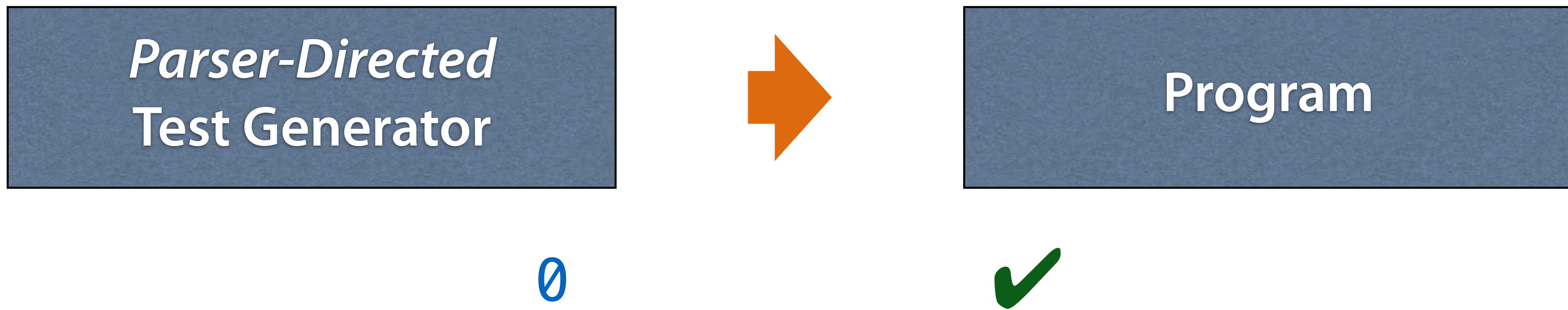


xyzzy

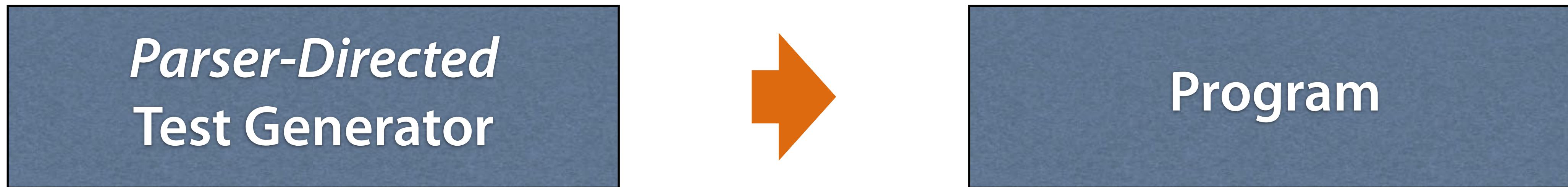
✗

- checks for digit
- checks for "true"/"false"
- checks for ""
- checks for '['
- checks for '{'

Dynamic Checks



Dynamic Checks

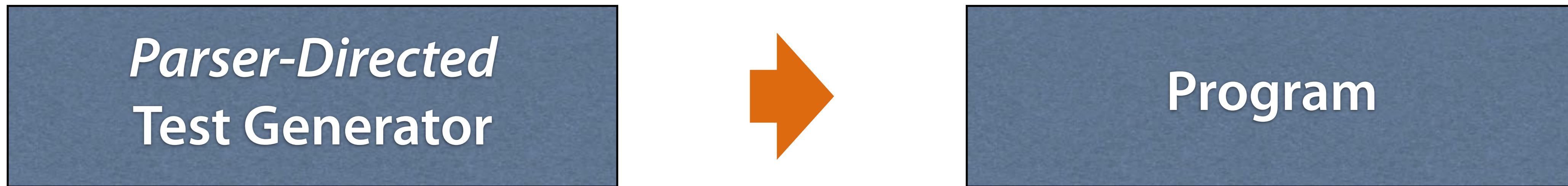


0

✓

- checks for digit
- checks for "true"/"false"
- checks for ""
- checks for '['
- checks for '{'

Learning Behavior

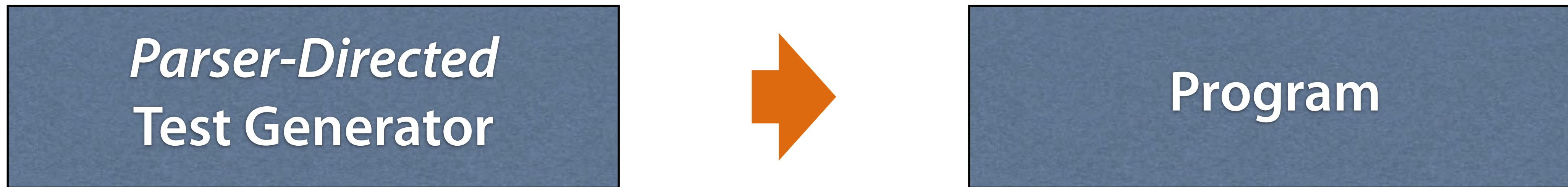


0

✓

- checks for digit
- checks for "true"/"false"
- checks for ""
- checks for '['
- checks for '{'

Dynamic Checks

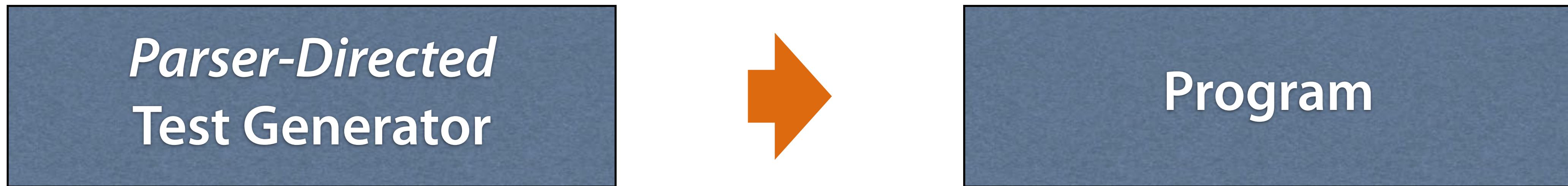


0

✓

- checks for digit
- checks for "true"/"false"
- checks for ""
- checks for '['
- checks for '{'

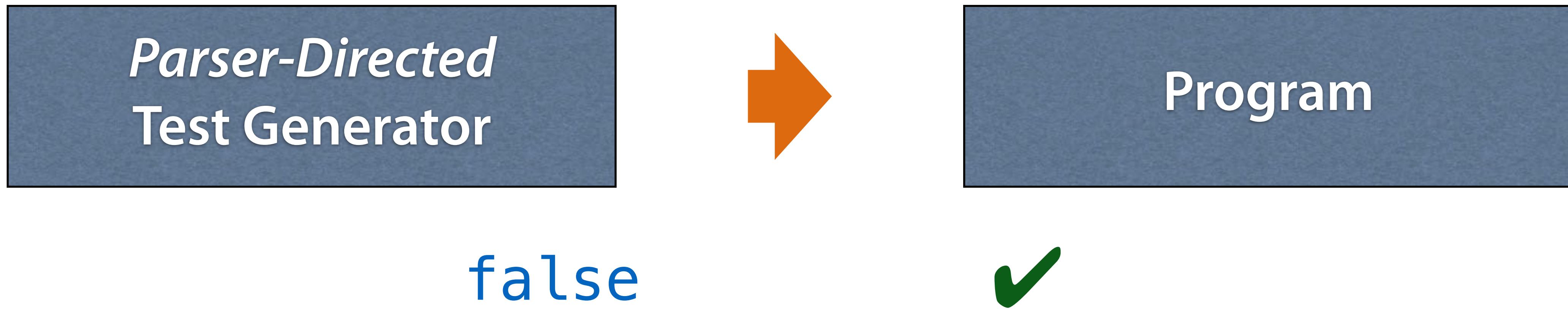
Dynamic Checks



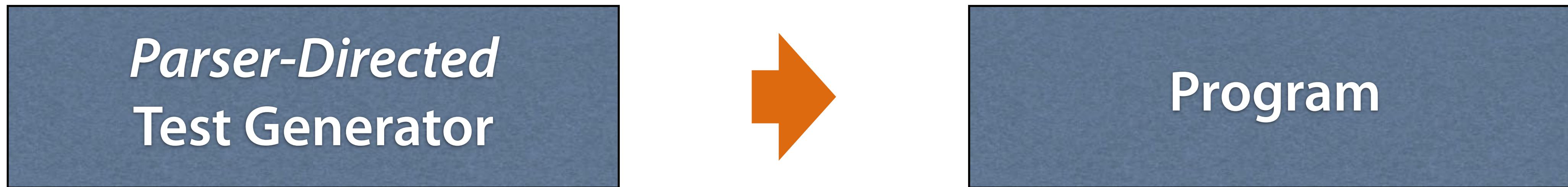
true



Dynamic Checks



Dynamic Checks

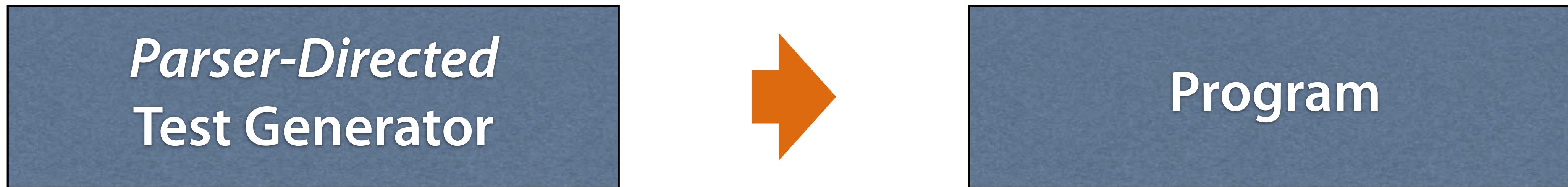


false



- checks for digit
- checks for "true"/"false"
- checks for ""
- checks for '['
- checks for '{'

Dynamic Checks



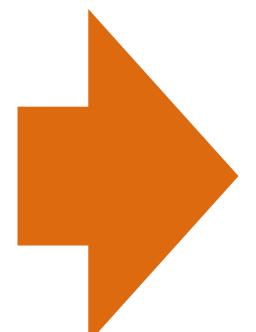
false



- checks for digit
- checks for "true"/"false"
- checks for ""
- checks for '['
- checks for '{'

Dynamic Checks

*Parser-Directed
Test Generator*



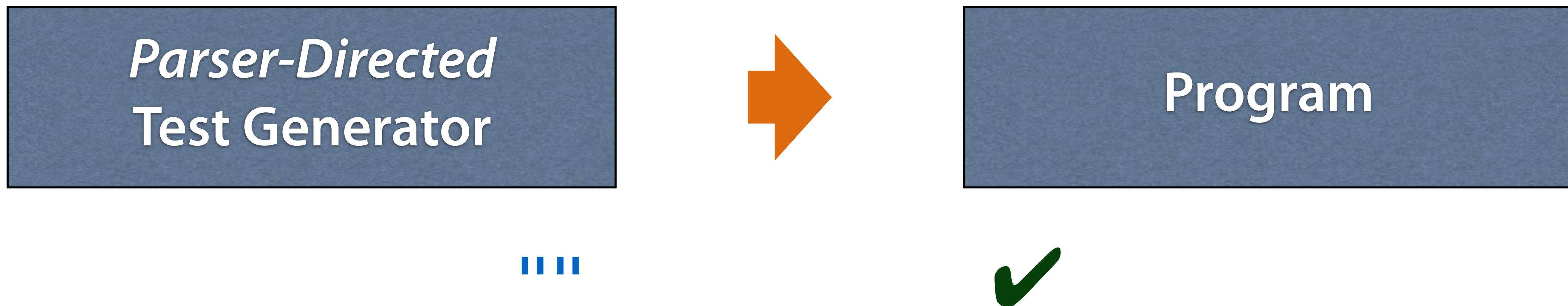
Program

"

X

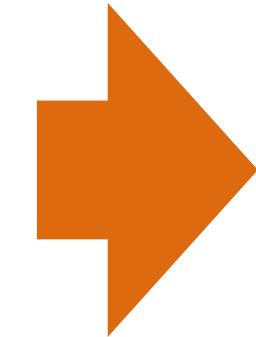
- checks for ""
- checks for '\'
- checks for character

Dynamic Checks



Learning JSON

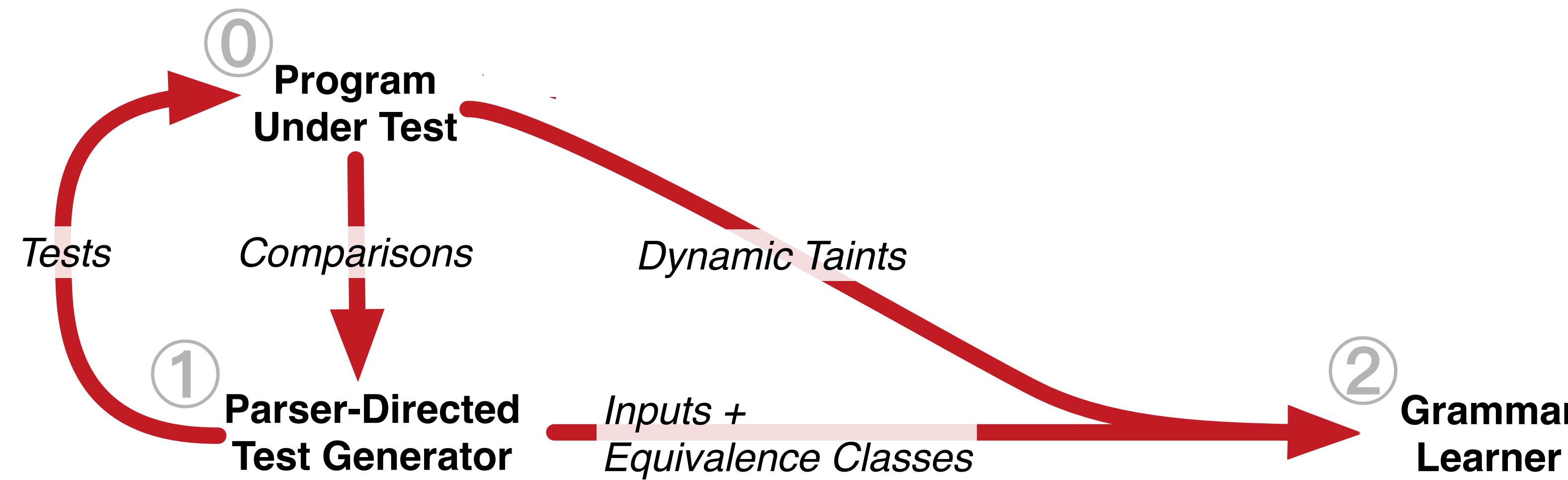
*Parser-Directed
Test Generator*



```
JSON ::= VALUE
VALUE ::= JSONOBJECT | ARRAY | STRINGVALUE |
         TRUE | FALSE | NULL | NUMBER
TRUE ::= 'true'
FALSE ::= 'false'
NULL ::= 'null'
NUMBER ::= '-' / [0-9]+ /
STRINGVALUE ::= '"' INTERNALSTRING '"'
INTERNALSTRING ::= /[a-zA-Z0-9 ]+/
ARRAY ::= '['
        [VALUE ',' VALUE] +
        ']'
JSONOBJECT ::= '{'
        [STRINGVALUE ':' VALUE
         ',', STRINGVALUE ':' VALUE]
        +
        '}'
```

All in One

PYGMALION prototype for Python programs



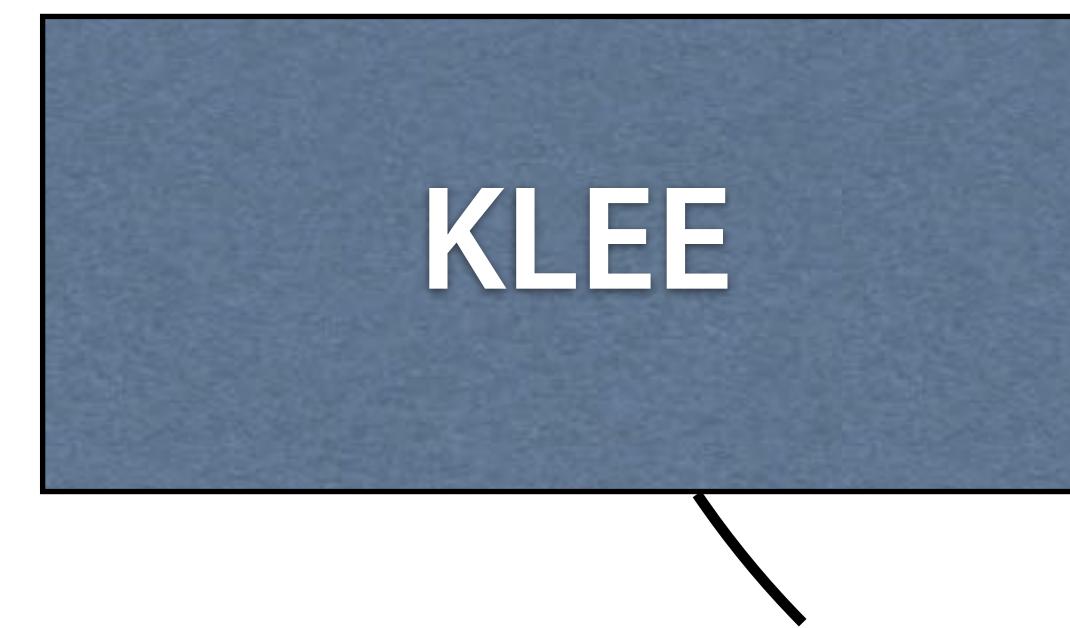
Gopinath, Mathis, Höschele, Kampmann, Zeller: "Sample-Free Learning of Input Grammars"

@AndreasZeller

Initial Evaluation

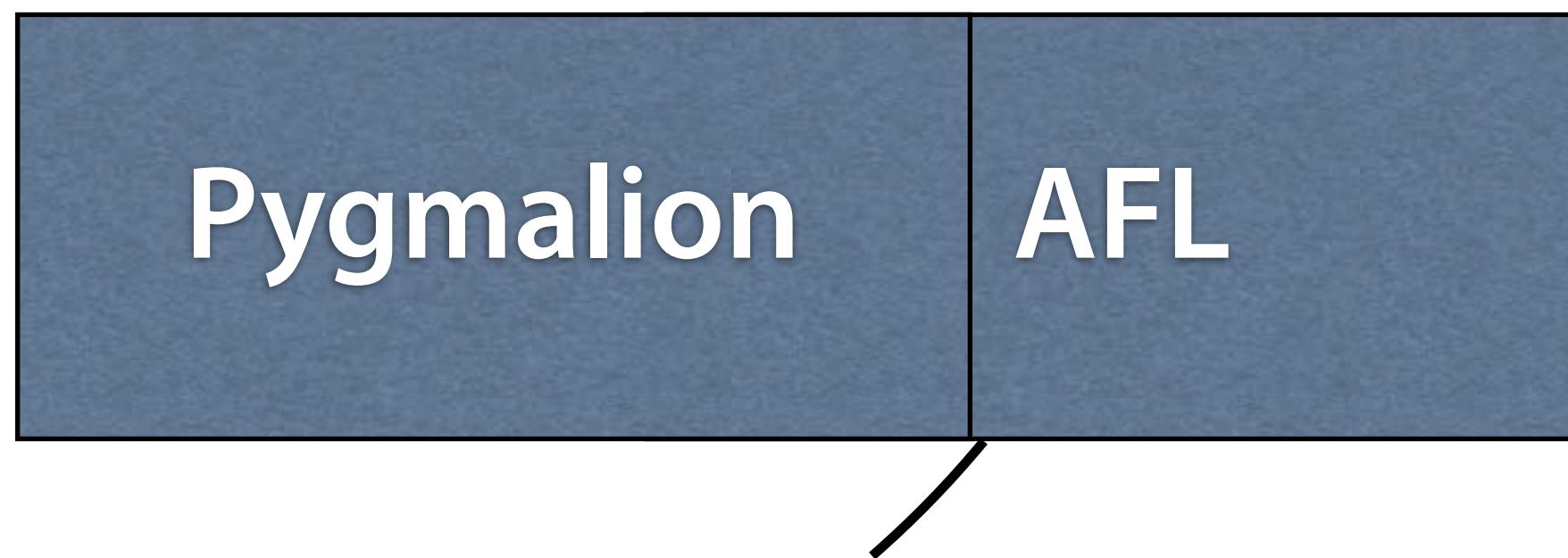


*mutation-based
test generator*



*constraint-based
test generator*

Initial Evaluation

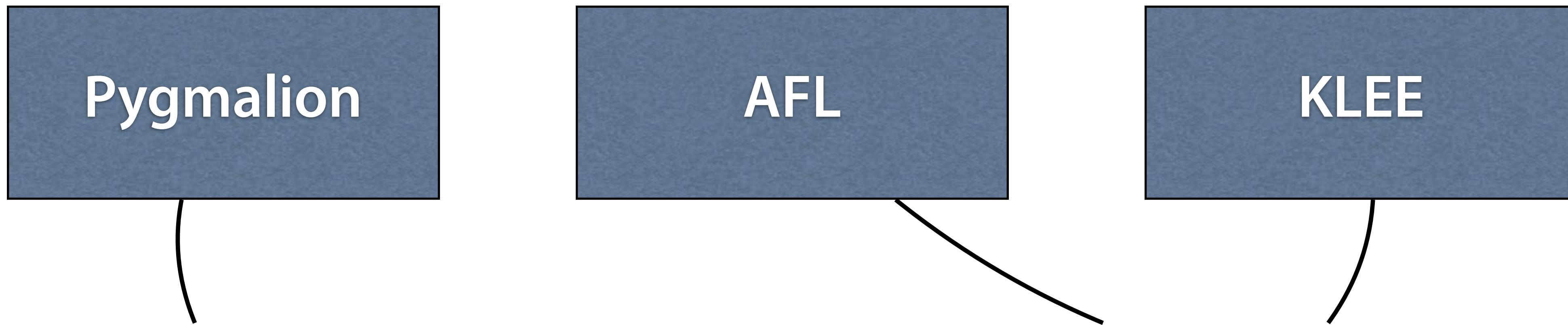


*mutation-based
test generator*



*constraint-based
test generator*

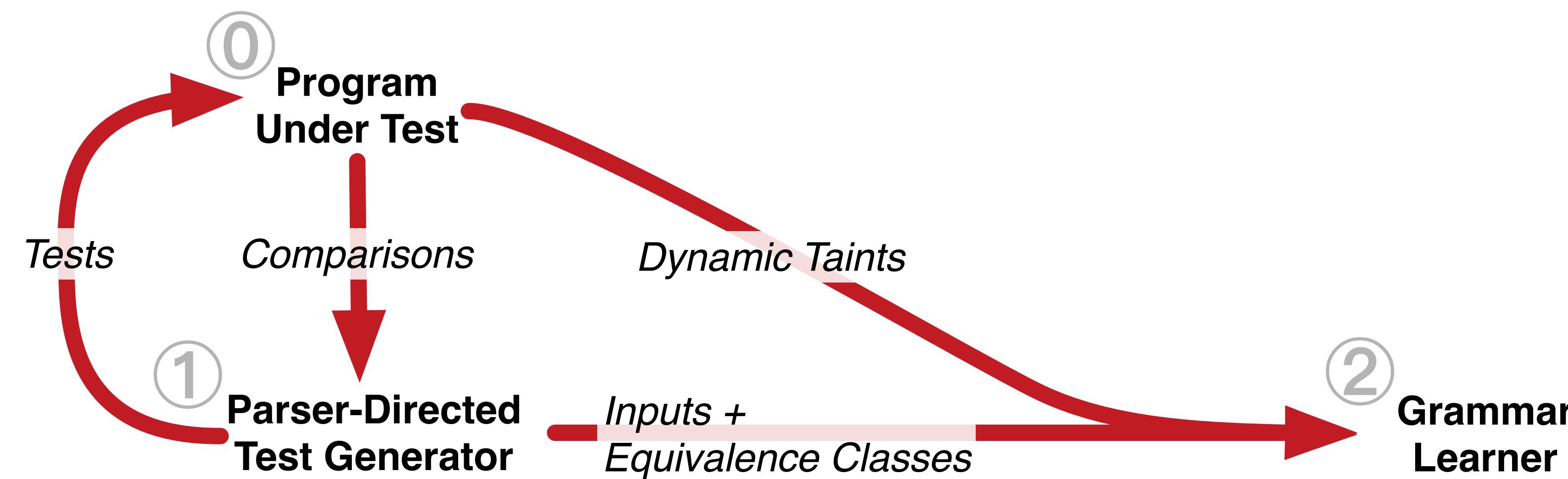
Initial Evaluation



No seed inputs for any tool

Testing with Inferred Grammars

PYGMALION prototype for Python programs



Perfect coverage, much faster than AFL, much better structure than KLEE

Gopinath, Mathis, Höschele, Kampmann, Zeller: "Sample-Free Learning of Input Grammars"

@AndreasZeller

Test Generation Compared

AFL

Low coverage, few valid inputs

KLEE

Top coverage, but flat inputs

Pygmalion

```
[false,[{"o":{$dYPrlj@?BR":397["+"]"SI+I4GzCW(C):-94"}}],  
[false,null]]
```

Top coverage, deep structure, control

Findings

- AFL is great for covering error-handling code
- Pygmalion and KLEE both achieve top coverage
- ~75% of inputs generated by Pygmalion are valid
- Pygmalion exercises far more input combinations
- Grammars give you control over what you want to test

Challenges

- **Implicit information flow**

Generated scanners and parsers



R. Gopinath



M. Höschele



B. Mathis



N. Havrikov

- **Context-sensitive features**

Binary formats, identifiers



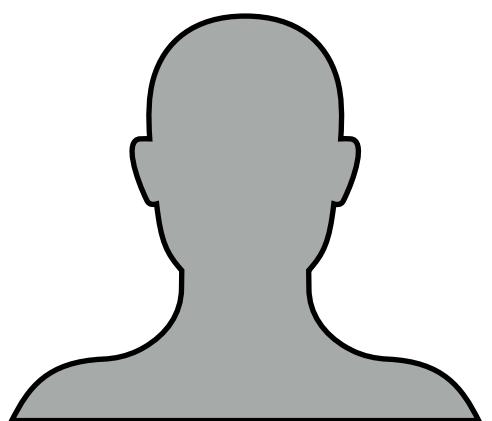
A. Kampmann



E. Soremekun



K. Jamrozik



M. Mera

- **Scale and applicability**

Port Pygmalion to C (2019)

- **Teach this!**

Book "Generating Software Tests"

Generating Software Tests

Breaking Software for Fun and Profit

by Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler

About this Book

Welcome to "Generating Software Tests"! Software has bugs, and catching bugs can involve lots of effort. This book addresses this problem by *automating* software testing, specifically by *generating tests automatically*. Recent years have seen the development of novel techniques that lead to dramatic improvements in test generation and software testing. They now are mature enough to be assembled in a book – even with executable code.

A Textbook for Paper, Screen, and Keyboard

You can use this book in three ways:

- You can **read chapters in your browser**. Check out the list of chapters in the menu above, or start right away with the [introduction to testing](#) or the [introduction to fuzzing](#). All code is available for download.
- You can **interact with chapters as Jupyter Notebooks** (beta). This allows you edit and extend the code, experimenting *live in your browser*. Simply select "Resources → Edit as Notebook" at the top of each chapter. [Try interacting with the introduction to fuzzing](#).
- You can **present chapters as slides**. This allows for presenting the material in lectures. Just select "Resources → View slides" at the top of each chapter. [Try viewing the slides for the introduction to fuzzing](#).

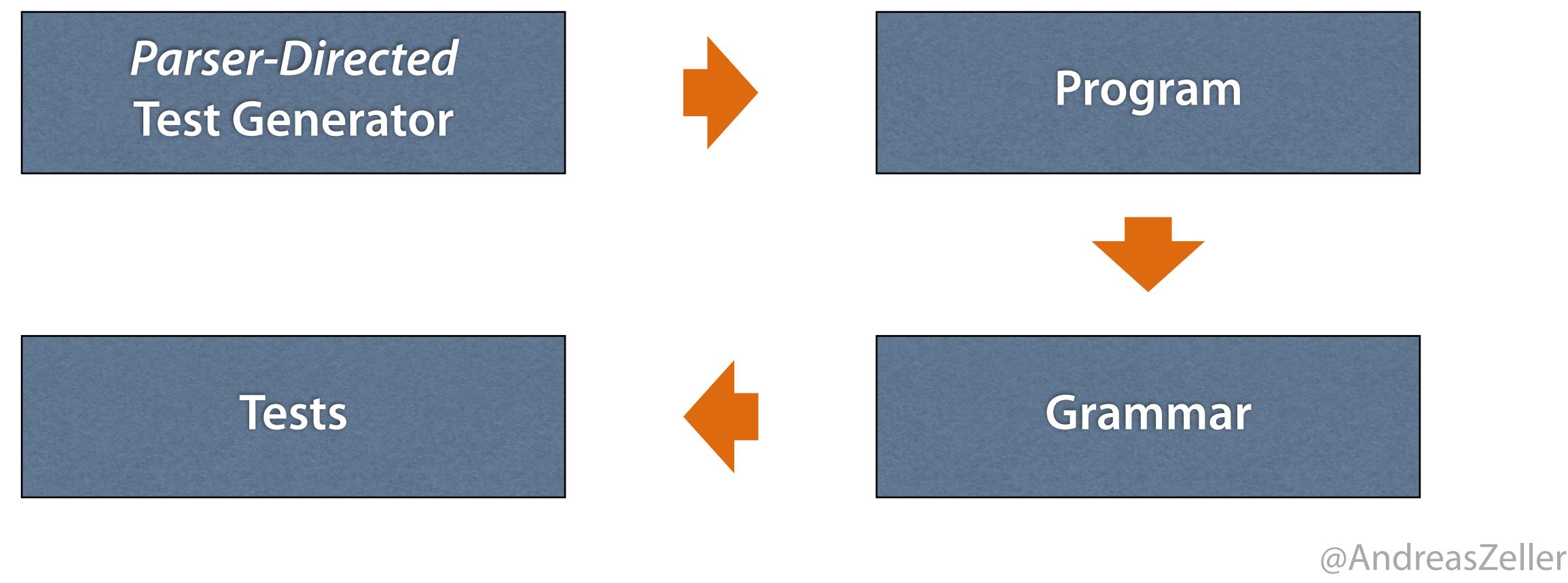
Who this Book is for

This work is designed as a *textbook* for a course in software testing; as *supplementary material* in a software testing or software engineering course; and as a *resource for software developers*. We cover random fuzzing, mutation-based fuzzing, grammar-based test generation, symbolic testing, and much more, illustrating all techniques with code examples that you can try out yourself.

Demo

Parser-Directed Testing

Learning Grammars



`http://user:password@www.google.com:80/command?foo=bar&lorem=ipsum#fragment`
`http://www.guardian.co.uk/sports/worldcup#results`
`ftp://bob:12345@ftp.example.com/oss/debian7.iso`

```
URL ::= PROTOCOL '://' AUTHORITY PATH ['?' QUERY] ['#' REF]
AUTHORITY ::= [USERINFO '@'] HOST [':' PORT]
PROTOCOL ::= 'http' | 'ftp'
USERINFO ::= /[a-z]+:[a-z]+/
HOST ::= /[a-z.]+/
PORT ::= '80'
PATH ::= /\[a-z0-9.\]/*/
QUERY ::= 'foo=bar&lorem=ipsum'
REF ::= /[a-z]+/
```

Höschele, Zeller: "Mining Input Grammars from Dynamic Taints", ASE 2016

@AndreasZeller

Test Generation Compared

Challenges

- **Implicit information flow**
generated scanners and parsers
 - **Context-sensitive features**
binary formats, identifiers
 - **Scale and Applicability**
port Pygmalion to C (this Fall)

