Breaking Parsers: Mutation-Based Generation of Programs with Guaranteed Syntax Errors

SOLVED!

Moeketsi Raselimo, Jan Taljaard, and **Bernd Fischer**
Stellenbosch University
South Africa

UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

Generating Well-Formed Programs for Fuzzing and Testing

Bernd Fischer
Stellenbosch University
South Africa

UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

Grammar-Based Testing

Test suite construction:

$$
\begin{array}{lll}
\mathit{prog} & \rightarrow & \texttt{module}\ \mathit{prio}\ \texttt{id} = \mathit{block}\ . \\
\mathit{prio} & \rightarrow & \texttt{[}\ \texttt{num}\ \texttt{]} \\
\mathit{block} & \rightarrow & \texttt{begin}\ (\mathit{decl}\ \texttt{;})^*\ (\mathit{stmt}\ \texttt{;})^*\ \texttt{end} \\
\mathit{decl} & \rightarrow & \texttt{var}\ \texttt{id}\ :\ \mathit{type} \\
\mathit{type} & \rightarrow & \texttt{bool} \mid \texttt{int} \\
\mathit{stmt} & \rightarrow & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt}\ (\texttt{else}\ \mathit{stmt})? \mid \\
 & & \texttt{while}\ \mathit{expr}\ \texttt{do}\ \mathit{stmt} \mid \texttt{id} = \mathit{expr} \mid \mathit{block} \\
\mathit{expr} & \rightarrow & \mathit{expr} = \mathit{expr} \mid \mathit{expr} + \mathit{expr} \mid \texttt{(}\ \mathit{expr}\ \texttt{)} \mid \texttt{id} \mid \texttt{num}
\end{array}
$$

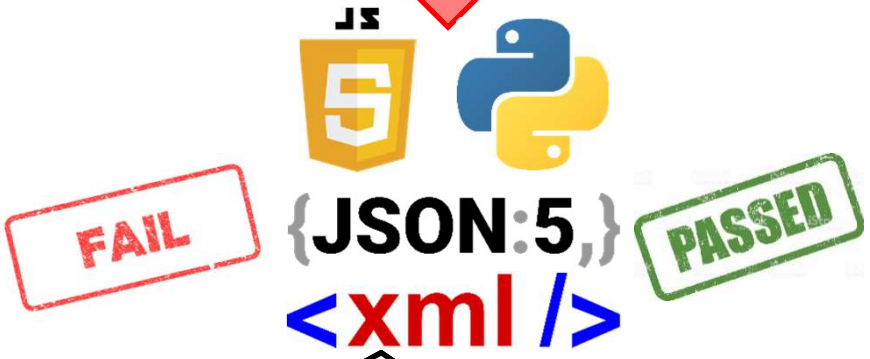
grammar *G*

sentence generation

test suite $TS \subseteq L(G)$

```
module[1] x = begin begin end; end.
module[2] y = begin end.
module[3] z = begin x = (y); end.
module[1] z = begin x = x + y; end.
module[2] x = begin y = z; end.
module[3] z = begin x = z = y; end.
module[1] y = begin y = 1; end.
module[2] y = begin if x then begin end; end.
module[3] y = begin var x : bool; end.
module[2] z = begin var z : int; end.
module[1] x = begin while x do begin end; end.
...
```

execution



Testing:

- some test fails $\Rightarrow L(G) \not\subseteq L(U)$
  - since $TS \subseteq L(G)$
- what about contextual constraints?

Scoping and Typing in CFGs

Usual solution: attributed grammars

- destroys conceptual simplicity
- provides unconstrained escape

Elegant solution: domain-specific mark-up languages (NaBL)

- destroys conceptual simplicity

Pragmatic solution (hack): mark-up tokens

- provides unconstrained escape