Comparison of Context-free Grammars Based on Parsing Generated Test Data

Bernd Fischer & Ralf Lämmel & Vadim Zaytsev

2011

Grammar nonequivalence

- ✓ Undecidable.
- ✓ Can we cheat?
- ✓ Converge grammars semi-automatically.
- ✓ Perform model synchronisation.
- ✓ …
- ✓ Grammar-based test generation!

Resources

- ✓ This talk & slides
- ✓ SLE pre-proceedings
- ✓ Pending SLE post-proceedings

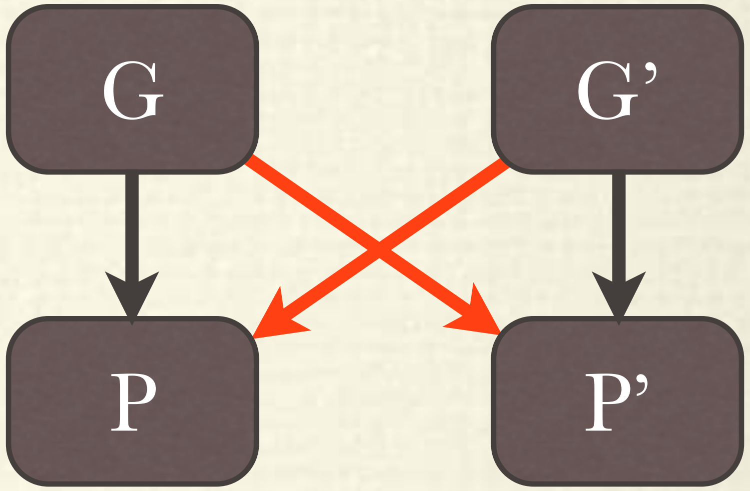
- http://softlang.uni-koblenz.de/testmatch
- http://slps.sourceforge.net/testmatch
- http://slps.sourceforge.net/tank/#tescol
- http://grammarware.net/text/2011/testmatch.pdf
- http://grammarware.net/slides/2011/testmatch-sle.pdf
- http://grammarware.net/bib/TestMatch2011.bib

Language comparison

- ✓ Implementing a parser from documentation (e.g., COBOL parser from the IBM manual)
- ✓ Creating/validating/fixing documentation (e.g., JLS and their “readable” & “implementable”)
- ✓ Grammarware interoperability (e.g., grammar-based protocol verification)
- ✓ Teaching ~~compiler construction~~ language processing (e.g., reducing the teacher’s effort; clone detection)

Methodology

- ✓ Asymmetric comparison:
  - ✓ Reference grammar vs. parser under test
- ✓ Symmetric comparison:
  - ✓ Differential testing
- ✓ Systematic test data generation
  - ✓ Controlled combinatorial coverage
- ✓ Larger sets of smaller test data items
- ✓ Nonterminal matching
- ✓ Non-context-free effects



Test data generation (1/4)

```
grammar(Ps)
    ⇐
        maplist(prod,Ps).

prod(p(L,N,X))
    ⇐
        mapopt(atom,L),atom(N), expr(X).

expr(true).
expr(t(T)) ⇐ atom(T).
expr(n(N)) ⇐ atom(N).
expr(','(Xs)) ⇐ maplist(expr,Xs).
expr(';'(Xs)) ⇐ maplist(expr,Xs).
expr('?'(X)) ⇐ expr(X).
expr('*'(X)) ⇐ expr(X).
expr('+'(X)) ⇐ expr(X).
```

```
tree(true).
tree(t(T)) ⇐ atom(T).
tree(n(P,T)) ⇐ prod(P).
tree(','(Ts)) ⇐ maplist(tree,Ts).
tree(';'(X,T)) ⇐ expr(X), tree(T).
tree('?'(Ts)) ⇐ mapopt(tree(Ts).
tree('*'(Ts)) ⇐ maplist(tree,Ts).
tree('+'(Ts)) ⇐ maplist1(tree,Ts).
```

Test data generation (2/4)

```
mark(C,p(L,N,X1),p(L,N,X2)) ⇐
    mark(C,X1,X2).

mark(uc,n(N),{n(N)}).
mark(bc,';'(Xs),{';'(Xs)}).
mark(bc,'?'(X),{'?'(X)}).
mark(bc,'*'(X),{'*'(X)}).
mark(bc,'+'(X),{'+'(X)}).

mark(C,'?'(X1),'?'(X2)) ⇐
    mark(C,X1,X2).
mark(C,'*'(X1),'*'(X2)) ⇐
    mark(C,X1,X2).
mark(C,'+'(X1),'+'(X2)) ⇐
    mark(C,X1,X2).

mark(C,','(Xs1),','(Xs2)) ⇐
    append(Xs1a,[X1|Xs1b],Xs1),
    append(Xs1a,[X2|Xs1b],Xs2),
    mark(C,X1,X2).

mark(C,';'(Xs1),';'(Xs2)) ⇐
    append(Xs1a,[X1|Xs1b],Xs1),
    append(Xs1a,[X2|Xs1b],Xs2),
    mark(C,X1,X2).
```

Marked productions are essentially marked expressions.

A nonterminal occurrence provides a focus for unfolding coverage. The EBNF forms ';', '?', '*', '+' provide foci for branch coverage.

Foci for BC and UC may also be found by recursing into subexpressions.

Sequences and choices combine multiple expressions, and foci are found by considering one subexpression at the time.

Coverage criteria

- ✓ **Trivial** coverage: if the test data set is not empty.
- ✓ **Nonterminal** coverage: if each nonterminal is exercised at least once.
- ✓ **Production** coverage: if each production in the grammar is exercised at least once.
- ✓ **Branch** coverage: each branch of ?|*+
- ✓ **Unfolding** coverage: each production of each right hand side nonterminal occurrence
- ✓ **Context-dependent branch coverage**!

Test data generation (3/4)

$$\begin{array}{l} \mathit{vary}(G,\{n(N)\},n(P,T)) \Leftarrow \\ \quad \mathit{def}(G,N,Ps), \\ \quad \mathit{member}(P,Ps), \\ \quad P = p(\_,\_,X), \\ \quad \mathit{complete}(G,X,T). \end{array}$$

A nonterminal occurrence in focus is varied so that all productions are exercised. (The complete spec also deals with chain productions and top-level choices in a manner that increases variation in a reasonable sense.)

$$\begin{array}{l} \mathit{vary}(G,\{';'(Xs)\},';'(X,T)) \Leftarrow \\ \quad \mathit{member}(X,Xs), \\ \quad \mathit{complete}(G,X,T). \end{array}$$

A choice in focus is varied so that all branches are exercised.

$$\begin{array}{l} \mathit{vary}(\_,\{'?'(\_)\},'?'([])). \\ \mathit{vary}(G,\{'?'(X)\},'?'([T])) \Leftarrow \\ \quad \mathit{complete}(G,X,T). \\ \mathit{vary}(\_,\{'*'(\_)\},'*'([])). \\ \mathit{vary}(G,\{'*'(X)\},'*'([T])) \Leftarrow \\ \quad \mathit{complete}(G,X,T). \\ \mathit{vary}(G,\{'+'(X)\},'+'([T])) \Leftarrow \\ \quad \mathit{complete}(G,X,T). \\ \mathit{vary}(G,\{'+'(X)\},'+'([T1,T2])) \Leftarrow \\ \quad \mathit{complete}(G,X,T1), \\ \quad \mathit{complete}(G,X,T2). \end{array}$$

An optional expression and a '*' repetition in focus are varied so that the cases for no tree and one tree are exercised. A '+' repetition is varied so that the cases for sequences of length 1 and 2 are exercised.

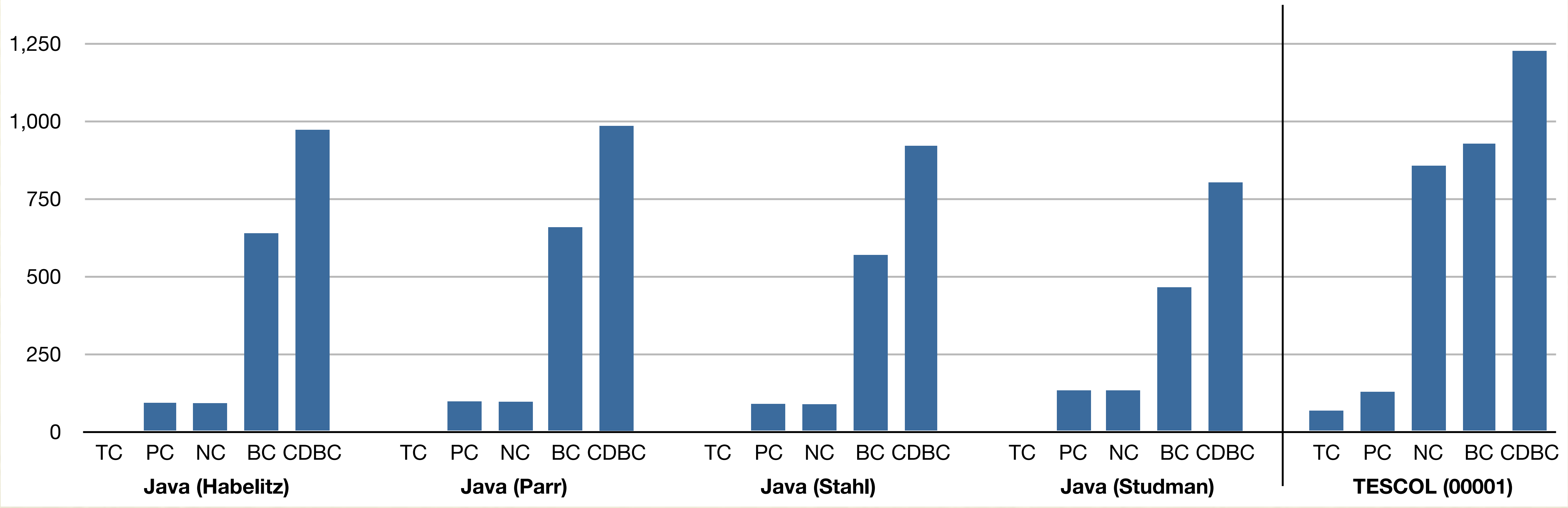
We omit all clauses for recursing into compound expressions; they mimic shortest completion but they are directed in a way that they reach the focus.

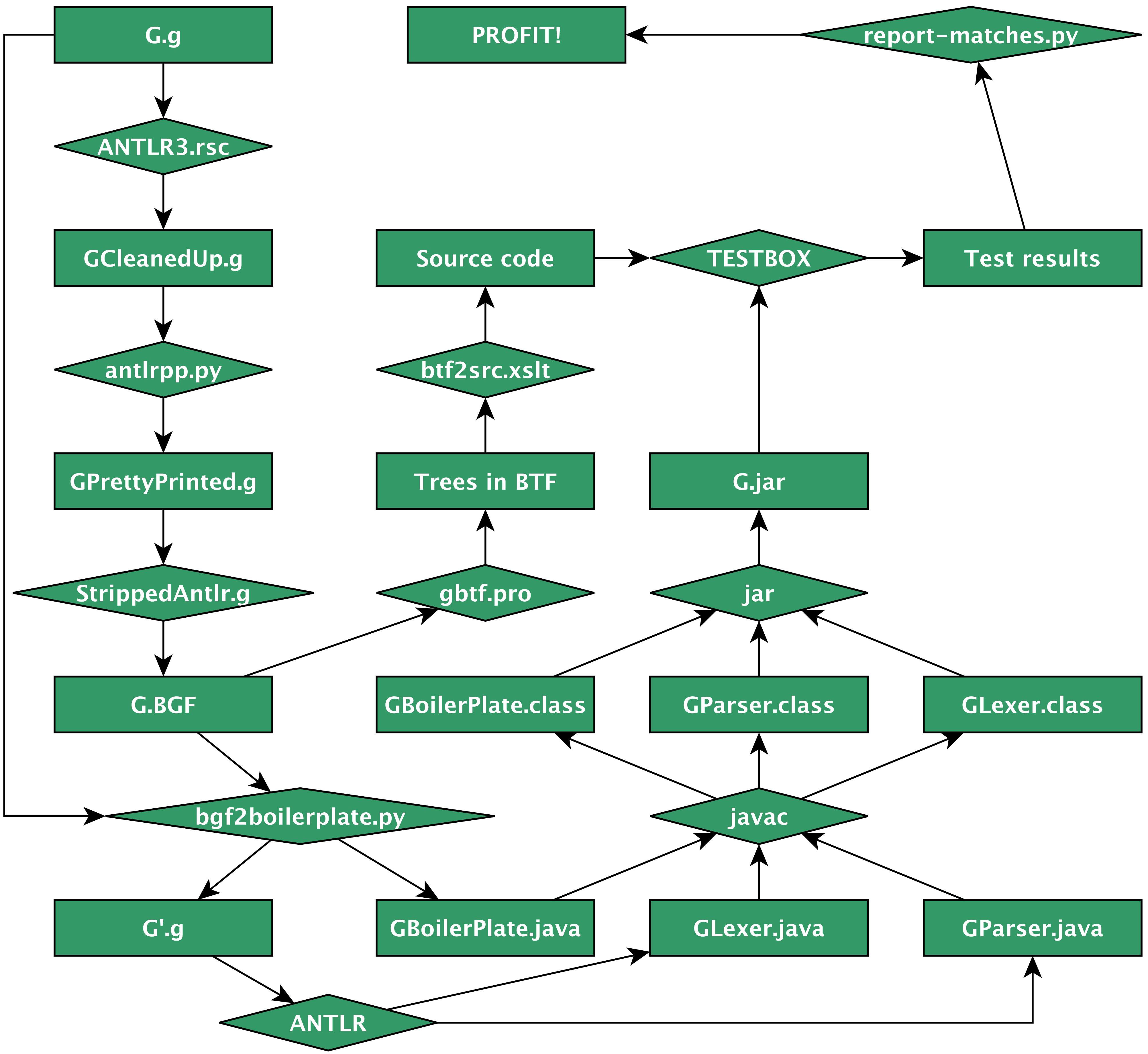
Test data generation (4/4)

```
tc(G,R,T)
   ⇐ def(G,R,_), complete(G,n(R),T).
nc(G,R,T)
   ⇐ def(G,R,_), dist(G,R,H,_), hole(G,n(R),H,T,V), complete(G,n(H),V).
pc(G,R,T)
   ⇐ def(G,R,Ps), member(P,Ps), complete(G,P,T).
pc(G,R,T)
   ⇐ def(G,R,_), dist(G,R,H,_), hole(G,n(R),H,T,V), pc(G,H,V).
bc(G,R,T)
   ⇐ cdbc(bc,G,R,T).
uc(G,R,T)
   ⇐ cdbc(uc,G,R,T).
cdbc(C,G,R,T)
   ⇐ def(G,R,Ps), member(P,Ps), mark(C,P,F), vary(G,F,T).
cdbc(C,G,R,T)
   ⇐ def(G,R,_), dist(G,R,H,_), hole(G,n(R),H,T,V), cdbc(C,G,H,V).
```

Grammar equivalence study: Java

Codename	Tech	Author	year	PROD	VAR	TERM	…
Habelitz	ANTLR3	Dieter Habelitz	2008	397	226	166	…
Parr	ANTLR3	Terence Parr	2006	425	151	157	…
Stahl	ANTLR2	Michael Stahl	2004	262	155	167	…
Studman	ANTLR2	Michael Studman	2004	267	161	168	…




G.g
PROFIT!
report-matches.py
ANTLR3.rsc
GCleanedUp.g
Source code
TESTBOX
Test results
antlrpp.py
btf2src.xslt
GPrettyPrinted.g
Trees in BTF
G.jar
StrippedAntlr.g
gbtf.pro
jar
G.BGF
GBoilerPlate.class
GParser.class
GLexer.class
bgf2boilerplate.py
javac
G'.g
GBoilerPlate.java
GLexer.java
GParser.java
ANTLR

Grammar extraction

- ✓ Semantic actions — {...}
- ✓ Rule arguments — [...]
- ✓ Semantic predicates — {...}?
- ✓ Syntactic predicates — (...)=>
- ✓ Rewriting rules — -> ^(...)
- ✓ Return types of the rules — returns ...
- ✓ Specific sections — options, @header, @members, @rulecatch, …
- ✓ Rule modifiers — options, scope, @after, @init, …
- ✓ Class negation (~), range operator (..), etc

Results (example)

class a { { switch (++ this) { } } }

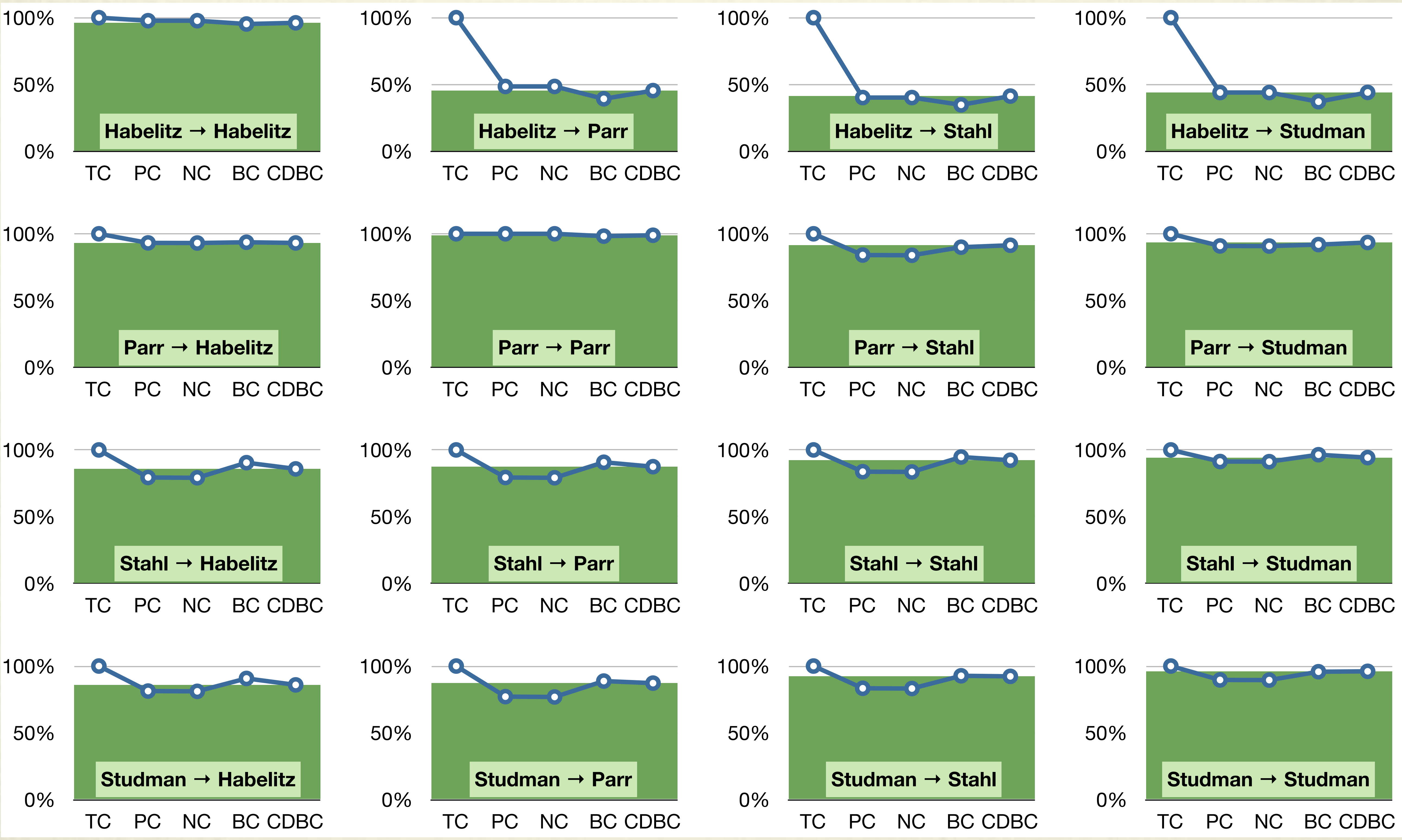
```
switchBlockLabels:
  switchCaseLabels switchDefaultLabel?
    switchCaseLabels
switchDefaultLabel:
  DEFAULT COLON blockStatement*
switchCaseLabels:
  switchCaseLabel*
```

Results (example)

class a { { switch (++ this) { } } }

```
switchBlockLabels
 :  switchCaseLabels switchDefaultLabel? switchCaseLabels
-> ^(SWITCH_BLOCK_LABEL_LIST switchCaseLabels switchDefaultLabel? switchCaseLabels) ;
```

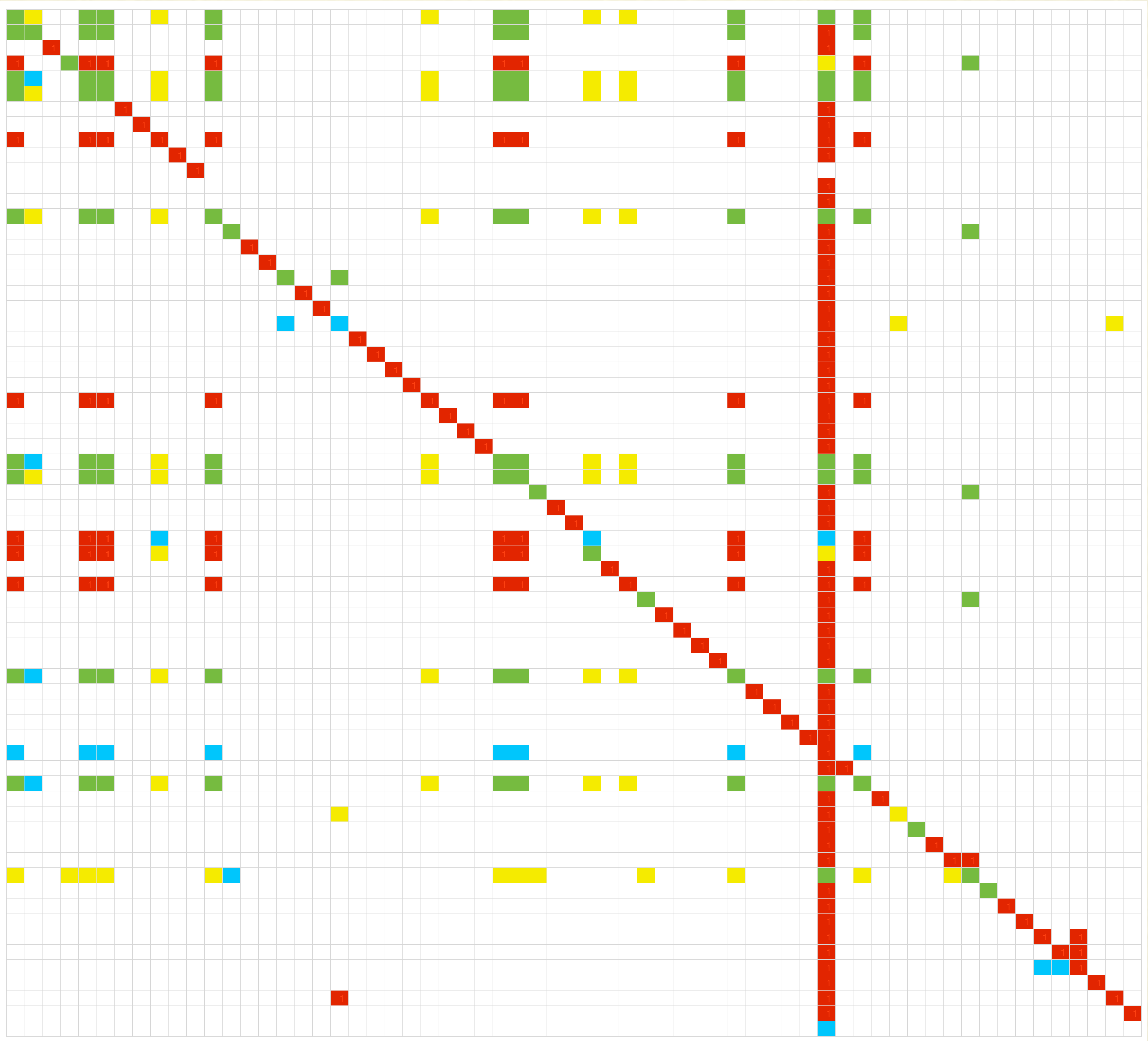
Grammar equivalence study: Java



Name matching study: TESCOL

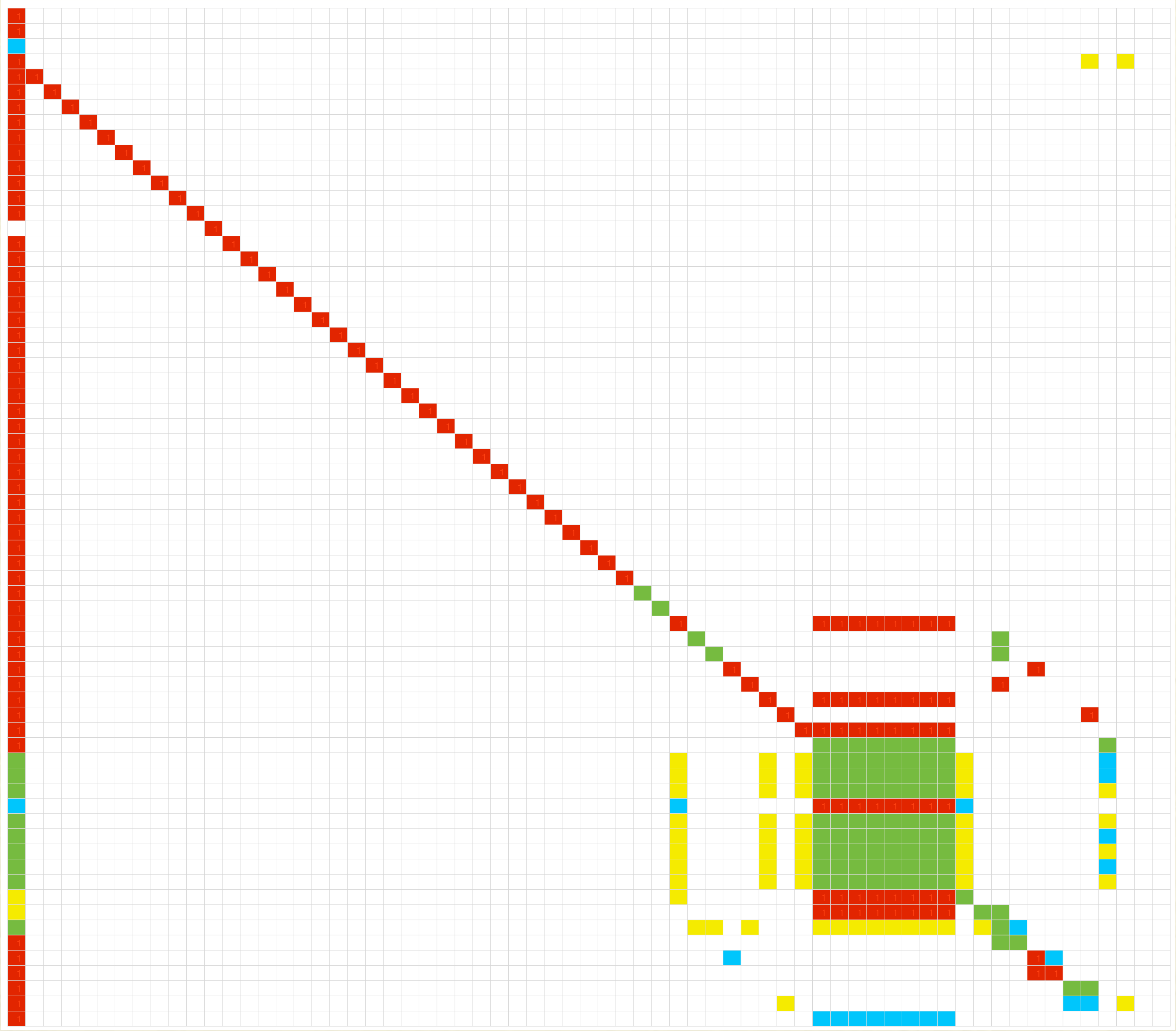
Codename	Tech	Author	year	PROD	VAR	TERM	…
00000	ANTLR3	[obfuscated]	2010	126	74	107	…
00001	ANTLR3	[obfuscated]	2010	79	67	107	…
00010	ANTLR3	[obfuscated]	2010	101	73	108	…
00011	ANTLR3	[obfuscated]	2010	84	63	107	…
00100	ANTLR3	[obfuscated]	2010	93	76	108	…
00101	ANTLR3	[obfuscated]	2010	94	76	107	…
00110	ANTLR3	[obfuscated]	2010	92	75	120	…
00111	ANTLR3	[obfuscated]	2010	84	71	108	…
01000	ANTLR3	[obfuscated]	2010	85	67	107	…
…	…	…	…	…	…	…	…

Nonterminal matching

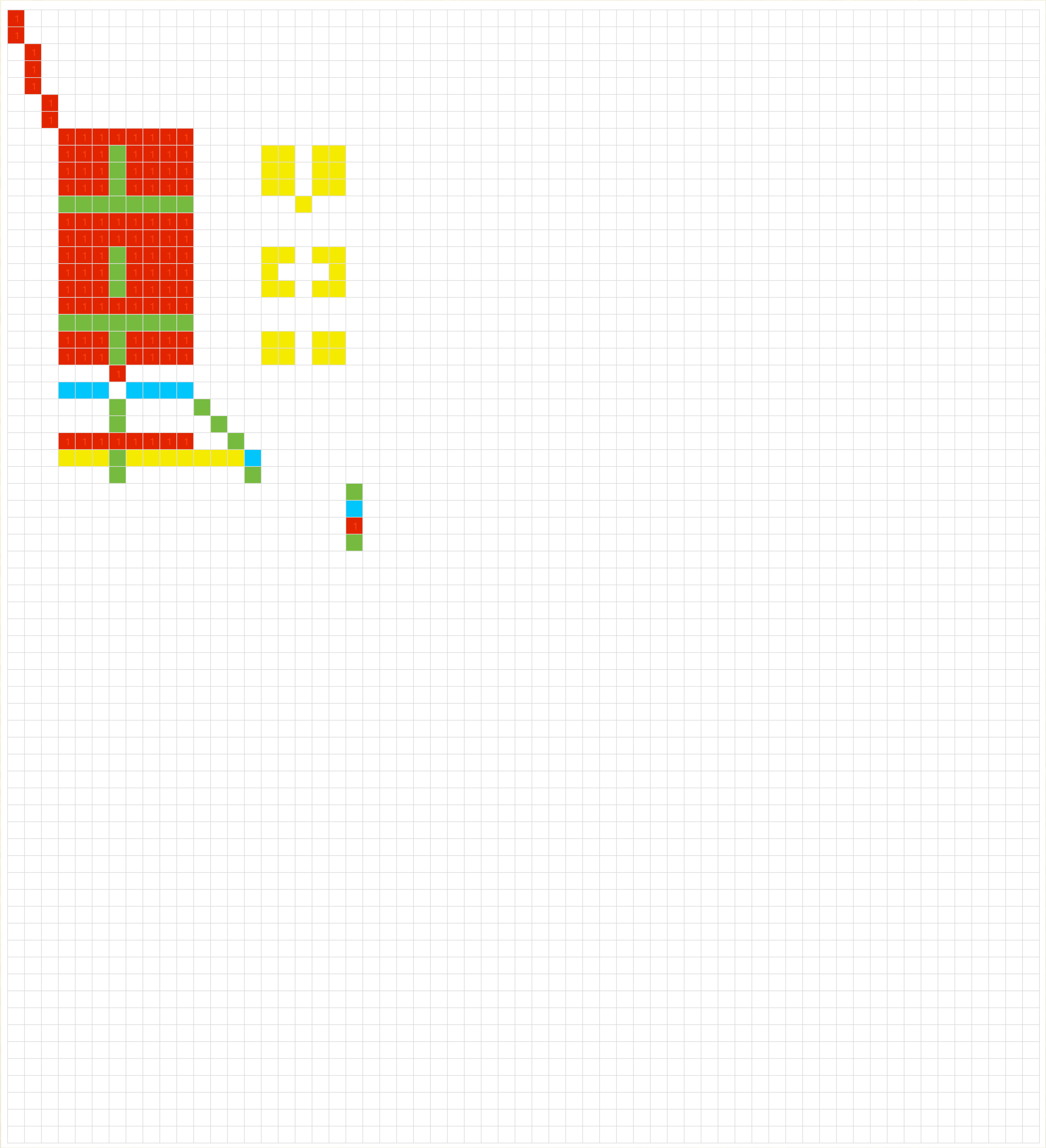


Nonterminal matching

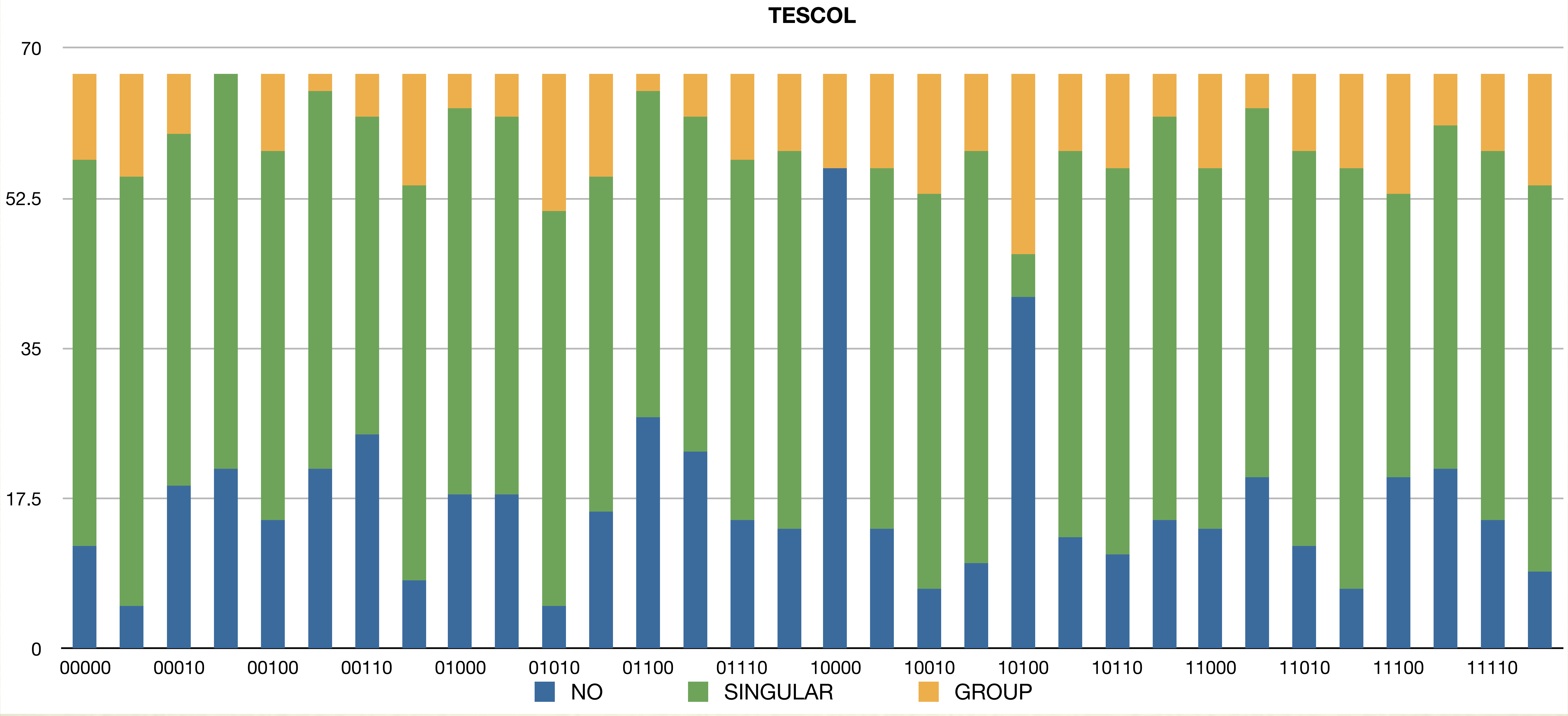
Nonterminal matching



Nonterminal matching



Nonterminal matching



Performance

Test set	generate					unparse	run			
	TC	PC	NC	BC	CDBC		Habelitz	Parr	Stahl	Studman
Habelitz	00:21	00:58	00:59	02:14	04:46	00:30	02:29	02:02	01:23	01:20
Parr	00:08	00:29	00:29	02:10	03:51	00:34	02:50	02:21	01:33	01:34
Stahl	00:08	00:35	00:35	02:45	05:01	00:39	03:02	02:34	01:40	01:39
Studman	00:09	00:38	00:39	02:59	05:12	00:37	03:05	02:35	01:41	01:41

	TC	PC	NC	BC	CDBC	unparse	00000	00001
00000	00:31	00:47	00:50	00:59	01:27	00:57	5:08:48	4:40:23
00001	00:05	00:14	00:51	01:12	01:53	01:47	5:41:22	5:10:36
...				...				
All TESCOL	02:21	08:44	27:21	34:21	59:19	17:32	—	

Conclusions

- ✓ Combinatorial grammar testing for matching languages
- ✓ Negative test cases? Larger test sets? Different criteria?
- ✓ Testing language modularity/integrability/extensibility
- ✓ Matches ⇒ suggestions ⇒ transformations
- ✓ Optimise performance
- ✓ Traceable abstraction, tighter coupled phases
- ✓ Integrate nonterminal matching in Grammar Lab