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

Bernd Fischer*, Ralf Lämmel†, Vadim Zaytsev‡

* Electronics and Computer Science, University of Southampton, Southampton, United Kingdom

† Software Languages Team, Universität Koblenz-Landau, Koblenz, Germany

‡ Software Analysis and Transformation Team, Centrum Wiskunde en Informatica, Amsterdam, The Netherlands

**Abstract.** There exist a number of software engineering scenarios that essentially involve equivalence or correspondence assertions for some of the context-free grammars in the scenarios. For instance, when applying grammar transformations during parser development—be it for the sake of disambiguation or grammar-class compliance—one would like to preserve the generated language. Even though equivalence is generally undecidable for context-free grammars, we have developed an automated approach that is practically useful in revealing evidence of nonequivalence of grammars and discovering correspondence mappings for grammar nonterminals. Our approach is based on systematic test data generation and parsing. We discuss two studies that show how the approach is used in comparing grammars of open source Java parsers as well as grammars from the course work for a compiler construction class.

**Keywords:** grammar-based testing, test data generation, coverage criteria, grammar equivalence, parsing, compiler construction, course work

## 1 Introduction

The paper is concerned with the automated comparison of context-free grammars based on test data generated from a grammar. The goal is here to reveal evidence, if any, for grammar nonequivalence, and to suggest a correspondence mapping between the nonterminals of the compared grammars. If no evidence of grammar nonequivalence is found, then this status may support an assertion of grammar equivalence (against the odds of undecidability). We develop a corresponding approach for grammar comparison which we demonstrate with two studies. The resulting infrastructure and both studies in grammar comparison are available online.[1]

The following grammar comparison scenarios exemplify the relevance of the presented work.

[1] http://slps.sourceforge.net/testmatch

***Grammar comparison scenarios***

- ◇ *Parser implementation*: The implementor of a parser may start from the "readable" grammar in a language manual and then transform it so that ambiguities or inefficiencies or grammar class violations are addressed. For instance, the (recovered) Cobol grammar from IBM's standard [9,13] requires substantial transformations before a quality parser is obtained. Grammar comparison can be used to shield this laborious process against errors.
- ◇ *Language documentation*: The documenter is supposed to provide a readable grammar for which it may be hard to establish though that it precisely represents the intended language. For instance, each version of the Java Language Specification contains a "more readable" and a "more implementable" grammar [4], and a substantial number of deviations have been identified by a complex and laborious process of grammar convergence [17]. Grammar comparison can be used to improve automation of this process.
- ◇ *Interoperability testing*: Suppose that there exist multiple grammars (in fact, front-ends) for the same (intended) language. Interoperability testing may be based on code reviews or manually developed test suites. Grammar comparison techniques can be used to test for interoperability more automatically and systematically even during mapping preparation phase.
- ◇ *Teaching language processing*: Compiler construction is a very established subject in computer science education and there are continuous efforts to improve and update corresponding courses [1,5,24,27]. However, the typical course involves laborious efforts—on the educator's side—some of which can be reduced with grammar comparison. For instance, the nonterminal names of student solutions can be automatically connected with a reference solution. Differences between the generated languages can be automatically identified.

***Contributions of the paper***

- ◇ We develop a framework for grammar-based test data generation and various related coverage criteria with associated and modularized generation algorithms. This results in a simple and integrated framework—when compared to previous work.
- ◇ We develop a grammar matching algorithm which uses a systematic classification scheme for the nonterminal correspondences between two grammars starting from accept/reject results obtained by "combinatorial" parsing: all mappings between nonterminals of the grammars are evaluated.
- ◇ We produce empirical evidence for the power of grammar-based test data generation in practical situations based on two complementary studies. Different coverage criteria are shown to make a contribution in this context.

***Roadmap of this paper:*** §2 presents a methodology for grammar comparison. §3 describes a set of coverage criteria and test data generation algorithms for use in grammar comparison. §4 reports on a grammar comparison study for Java grammars which concludes with a nonequivalence result in particular. §5 develops a matching

algorithm for nonterminals based on parser applications to test data. §6 reports on a grammar comparison study for a compiler construction class managing to match grammars of the course work. §7 discusses related work. §8 concludes the paper.

## 2 Methodology

Overall, the idea of test-based comparison of grammars may appear relatively straightforward. Nevertheless, a suitable methodology has to be set up.

***Asymmetric comparison.*** Given are two grammars $G$ and $G'$ which have been extracted from or can be turned uniformly into parsers (acceptors) $A$ and $A'$. Here we call $G$ the *reference grammar* and $G'$ the *grammar under test*. Accordingly, $G$ represents the intended language, and we want to support assertions of correctness and completeness for $G'$ relative to $G$. We say that $G'$ is *complete*, if $A'$ accepts all strings that $A$ accepts. We say that $G'$ is *correct*, if $A'$ rejects all strings that $A$ rejects. With test-based comparison we can attempt to find counterexamples. That is, we generate (positive) test cases from $G$ and apply $A'$ to them; rejection provides evidence of incompleteness of $G'$. We also generate (positive) test cases from $G'$ and apply $A$ to them; rejection provides evidence of incorrectness of $G'$.

***Symmetric comparison.*** In practice, we cannot always assume that one grammar is clearly a reference grammar. Instead, both grammars may simply compete with each other to appropriately capture an intended language. In this case, it does not make sense any longer to speak of correctness and completeness. One can still exercise both of the above-mentioned directions of test data generation and parser application, but what was called evidence of incompleteness or incorrectness previously simply reduces to evidence of nonequivalence. (A)symmetric comparison, as discussed here, is a form of *differential testing* [20].

***Non-context-free effects.*** When discussing (a)symmetric grammar comparison so far, we stipulated that a parser $A$ should precisely accept the language generated by the grammar $G$. Obviously, this is not necessarily true in practice. For instance, grammar-class restrictions or built-in ambiguity resolution strategies may imply that a generated parser rejects some part of the formal language. Also, parser descriptions may provide additional control that also goes beyond plain context-free grammars; see, for example, syntactic and semantic predicates in ANTLR. Further, a parser may rely on a designated lexer whose description may be incorporated into the grammar, but some aspects may be hard to model explicitly, e.g., whitespace handing. These and other differences between grammar and parser challenge the soundness of any grammar comparison approach. We encounter such effects in the case studies, but we defer a more general investigation of these effects to future work.

***Nonterminal matching.*** When discussing (a)symmetric grammar comparison so far, we focused on confidence for equivalence or evidence for nonequivalence. As some of the introductory scenarios indicated, one may want to go beyond (non)equivalence and aim at nonterminal matching. This generalization is useful for *understanding* grammars and for preparing an effective *mapping* between derivation trees of the compared grammars, if needed. The key idea here is to use data sets indexed by nonterminals so that acceptance/rejection can be tested per nonterminal which eventually allows to match nonterminals from the two grammars when they accept each other test data sets better than for any other combination of nonterminals. For practicality's sake, it is important to support nonterminal matching even for grammars that are not fully equivalent.

***Stochastic vs. systematic test data generation.*** As we discuss in [15], prior art in grammar-based testing focuses on *stochastic test data generation* (e.g., [19,25]). The canonical approach is to annotate a grammar with probabilistic weights on the productions and other hints. A test data set is then generated using probabilistic production selection and potentially further heuristics. Stochastic approaches have been successfully applied to practical problems. One conceptual challenge with stochastic approaches is that they require some amount of configuration to achieve coverage. For instance, recursive nonterminals in grammars imply a need for appropriate probabilistic weights so that divergence is avoided. This needs to be done carefully to avoid, in turn, insufficient coverage. In the present paper, we leverage systematic test data generation, by which we mean that test data sets are generated by effective enumeration methods for the coverage criteria of interest. These methods do not require any configuration. Also, these methods imply minimality of the test data sets in both an intuitive and a formal sense.

***Larger sets of smaller test data items.*** Starting with Purdom's seminal work [22], there is the question of how to trade off size of test data *set* vs. size of test data *items*. For instance, when attempting to cover all productions of a grammar, one may generate a smaller test data set with each item covering as many additional productions as possible (thereby implying larger items); instead, one may also generate a larger test data set with each item covering as few individual productions as possible (thereby implying smaller items). In the present paper, without loss of generality, we adopt the latter principle which is well in line with general (unit) testing advice. We also refer to [20] for support of this principle.

## 3 Test data generation

Based on previous work on grammar-based test data generation [7,14,15,18,22,25], we develop a generation framework which accumulates a number of coverage criteria and associated generation algorithms in a modular manner. We have specified all ingredients in a declarative logic program of which we show excerpts

below. (The complete specification, which also includes some optimizations, is available online; see the footnote on the first page.)

### 3.1 Grammars and trees

Generation algorithms process a grammar and generate trees. We represent grammars as lists of productions. A production is a triple $p(L, N, X)$ consisting of an optional label $L$, a left-hand side nonterminal $N$, and a right-hand side expression $X$. *expr/1* specifies the allowed expression forms for BNF and EBNF, using functors *true* for $\varepsilon$, $t$ for terminals, $n$ for nonterminals, ',' for sequences, ';' for choices, '?' for optional parts, '*' and '+' for repetitions. The structure of trees follows exactly the one of grammars, and hence all functors are overloaded to represent trees as well as grammars. We refer to Figure 1 for details.[2] Grammar fragments are included into trees for origin tracking; see $n$ and ';' on the right of the figure.

```
grammar(Ps)                          prod(p(L,N,X))
   ⇐ maplist(prod,Ps).                  ⇐ mapopt(atom,L), atom(N), expr(X).

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

**Fig. 1.** Logic programming-based specification of grammars and trees.

### 3.2 Coverage criteria

Suppose that $S$ is a set of derivation trees for a given grammar $G$. We say that $S$ achieves *trivial coverage* (TC), if $S$ is not empty; $S$ achieves *nonterminal coverage* (NC), if $S$ exercises each nonterminal of $G$ at least once; $S$ achieves *production coverage* (PC), if $S$ exercises each production of $G$ at least once; $S$ achieves *branch coverage* (BC), if $S$ exercises each branch for each occurrence of ';', '?', '*', '+' at least once; $S$ achieves *unfolding coverage* (UC), if $S$ exercises each production of each right-hand side nonterminal occurrence at least once.

[2] The definitions leverage higher-order predicates *maplist/2*, *maplist1/2*, *mapopt/2* for applying unary predicates to arbitrary lists, to lists with at least one element, or to lists of zero or one elements, respectively.

For backward compatibility with preexisting terminology [14], we also give the name *context-dependent branch coverage* (CDBC) to the combination of BC and UC.

Trivial, nonterminal and production coverage presumably do not require further formal clarification. BC and UC require the notion of a *focus*, i.e., the right-hand side non-terminal that is expanded ("varied") next. The predicate *mark/3* in Figure 2 precisely enumerates all possible foci for branch and unfolding coverage in an expression (or an entire production). In the case of BC, all expressions that involve a form of choice are foci. In the case of UC, all expressions that denote a nonterminal occurrence are foci. In *mark(C,X1,X2)*, *C* is the name of the coverage criterion (*bc* or *uc*), *X1* is the original expression, *X2* is *X1* updated so that one subterm contains a focus that is marked by enclosing it in {...}.

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

Marked productions are essentially marked expressions.

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

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

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

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

```
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).
```

Sequences and choices combine multiple expressions, and foci are found by considering one subexpression at the time. (Marking is designed to be nondeterministic here.)

**Fig. 2.** Marking foci for branch and unfolding coverage.

The remarkable property of this uniform specification is that it facilitates effectively systematic test data generation for the coverage criteria *BC* and *UC* in the sense that a generation algorithm may simple iterate over the extension of the predicate and exercise all options for any marked focus.

### 3.3 Generation primitives

Generation algorithms for the five coverage criteria can be composed from a small set of primitives; one of which is the predicate *mark/3* described above. These are the remaining ones; we include mode annotations for the intended direction of usage.[3]

- *complete(+G,+X,−T)*: we follow the standard definition [18,22]; the tree $T$ is the shortest completion of expression $X$ according to grammar $G$.
- *mindepth(+G,+N,−D)*: the natural number $D$ is the minimum depth of derivation trees rooted by nonterminal $N$ according to grammar $G$ in terms of the nonterminal nodes on paths—this is the essential relationship for shortest completion and possibly further generation algorithms; it can be computed by a simple fixed point computation.
- *hole(+G,+N,+H,−T,−V)*: the tree $T$ is rooted in nonterminal $N$ with a "hole" for a derivation tree for nonterminal $H$ where the hole is accessible through the place holder (logical variable) $V$—the tree is the smallest one in the sense of the shortest path from $N$ to $H$ (in terms of nonterminal nodes) while using shortest completion everywhere else.
- *dist(+G,+$N_1$,+$N_2$,−D)*: the natural number $D$ is the (minimum) distance between nonterminals $N_1$ and $N_2$ in the sense of nonterminal nodes on paths in derivation trees from $N_1$ to $N_2$—this is the essential relationship for smallest trees with holes; it can be computed by a simple fixed point computation similar to *mindepth/3*.
- *vary(+G,+X,−T)*: the expression $X$ contains exactly one focus ($\{\ldots\}$) and trees $T$ are enumerated such that they are shortest completions overall, but all "immediate options" for the focus are exercised.

Figure 3 lists the specification of *vary/3*; it uses the primitive *complete/3* and a trivial relation *def(+G,?N,−Ps)* between grammars $G$, nonterminals $N$, and productions $Ps$ such that $N$ is a nonterminal defined by the grammar $G$ according to the productions $Ps$.

### 3.4 Generation algorithms

We are ready to define algorithms for the coverage criteria *TC*, *NC*, *PC*, *BC*, and *UC*. We leverage the primitives mentioned above. See Figure 4 for the specification of the algorithms. The simple specifications generate larger sets of smaller trees that achieve coverage in the intended manner, due to the focus of the primitives on minimality (i.e., shortest completion, smallest tree, etc) and the individual expansion of the foci via *mark/3*. For instance *uc/3* with arguments *uc(+G,?R,−T)* generates (by backtracking) derivation trees $T$ for nonterminals

[3] The modes "+" and "−" are used for (instantiated) input or (uninstantiated) output arguments, respectively. In principle, there is also the mode "?" for unconstrained arguments.

```
vary(G,{n(N)},n(P,T)) ⇐
    def(G,N,Ps),
    member(P,Ps),
    P = p(_,_,X),
    complete(G,X,T).
```

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.)

```
vary(G,{';'(Xs)},';'(X,T)) ⇐
    member(X,Xs),
    complete(G,X,T).
```

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

```
vary(_,{'?'(_)},'?'([])).
vary(G,{'?'(X)},'?'([T])) ⇐
    complete(G,X,T).
vary(_,{'*'(_)},'*'([])).
vary(G,{'*'(X)},'*'([T])) ⇐
    complete(G,X,T).
vary(G,{'+'(X)},'+'([T])) ⇐
    complete(G,X,T).
vary(G,{'+'(X)},'+'([T1,T2])) ⇐
    complete(G,X,T1),
    complete(G,X,T2).
```

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.

**Fig. 3.** Varying foci for branch and unfolding coverage.

```
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).
```

**Fig. 4.** Enumeration of test data achieving coverage.

$R$ from grammar $G$. It is important to notice that the predicates of Figure 4 iterate over all possible nonterminals for the root $R$ of the generated trees (as-

suming $R$ is left uninstantiated). This implies that we can generate test data sets that are indexed by the nonterminals of the grammar; see again §2.

Let us pick one generation algorithm for discussion. For instance, predicate *pc/3* enumerates trees achieving $PC$ as follows. The first clause of *pc/3* models the case that we want to cover a production $P$ of the rooting nonterminal $R$, in which case we simply apply shortest completion to $P$. The second clause of *pc/3* models the case that we want to cover a production of some nonterminal $H$ that is only reachable through a nonempty path starting from the rooting nonterminal $R$, in which case we create a tree with a hole for nonterminal $H$ to be filled by recursive invocation of *pc/3*.

## 4 Grammar nonequivalence study: Java 5

In this study, we apply symmetric grammar comparison to four different grammars, in fact, parsers of the Java programming language. That is, we generate test data for all the grammars, and each test case from each of the test sets is then fed into each of the parsers. In this manner, we discover differences between the languages generated by the four grammars. (All involved grammars and tools are available online; see the footnote on the first page.)

### 4.1 Grammar sources

In previous work, we have extracted Java grammars from the Java Language Specification (JLS) [4], with many inconsistencies and irregularities reported in [17]. These Java grammars appear to be a good target for grammar comparison, yet a significant grammar recovery effort would be needed to make those grammars executable. In fact, this recovery process would involve judgment calls that possibly bring the executable grammar further away from the JLS.

It turns out though that several handmade, executable adaptations of the JLS already exist and are deployed in practice. Thus, in the current work we acquired four operational grammars for J2SE 5.0 ("Java 5") from four widely used ANTLR sources, distributed under the BSD license. The underlying ANTLR-based parser descriptions strive to cover the same language; they were developed independently from one another by different grammar engineers, based on their experience, style and understanding of the JLS [4]:

| | Technology | Author | Year | PROD | VAR | TERM |
|---|---|---|---|---|---|---|
| **Habelitz** | ANTLR3[4] | Dieter Habelitz[5] | 2008 | 397 | 226 | 166 |
| **Parr** | ANTLR3 | Terence Parr[6] | 2006 | 425 | 151 | 157 |
| **Stahl** | ANTLR2[7] | Michael Stahl[8] | 2004 | 262 | 155 | 167 |
| **Studman** | ANTLR2 | Michael Studman[9] | 2004 | 267 | 161 | 168 |

[4] http://www.antlr.org

[5] http://www.antlr.org/grammar/1207932239307/Java1_5Grammars/Java.g

[6] http://www.antlr.org/grammar/1152141644268/Java.g

[7] http://www.antlr2.org

[8] http://www.antlr.org/grammar/1093454600181/java15-grammar.zip

[9] http://www.antlr.org/grammar/1090713067533/java15.g

PROD, VAR and TERM values in the table refer to simple grammar metrics [21] of the number of top alternatives in grammar production rules, the number of nonterminal symbols, and the number of terminal symbols. We have developed a simple infrastructure for driving a set of ANTLR-based parsers including aspects of parser generation and selecting the appropriate ANTLR version.

### 4.2 Grammar extraction

Based on previous work on grammar convergence [16], we were able to extract the context-free grammars from the ANTLR-based parser description. That is, we developed a designated extractor, using the Rascal [11] meta-programming language, so that the following ANTLR constructs are abstracted away:

- 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`, ...

Also some minor notational features like character class negation (`~`) or range operator (`..`) needed to be translated into basic context-free grammar notation. Tokens defined as terminals were merged with the normal grammar rules. By doing so, we are able to fit most of the grammar knowledge in our infrastructure without focusing on idiosyncratic details. An abstracted grammar differs from the original in terms of the accepted language, and these effects are yet to be fully studied; see §2.

### 4.3 Test set generation

Using the algorithm and the infrastructure described in §3, we generated test data for (only) the start symbols of each of the Java grammars. Figure 5 reports on the amount of test data. As an exercise in studying the effectiveness of the different coverage criteria, we explicitly divided test data based on the coverage criteria, and ultimately found out that the CDBC set contains the largest number of test cases and usually includes TC, PC, NC and BC sets.

Trivial coverage only involves one test case (rooted in the start symbol). One may expect that the shortest completions of all grammars are mutually accepted by the parsers. The test sets for production and nonterminal coverage yield the same test sets because of ANTLR-implied[10] and author-specific grammar style. The way *BC* and *UC* (and hence *CDBC*) are defined, the corresponding test sets need not to imply *PC* and *NC*, but, in practice, the implication holds. Hence, for the rest of the paper, we use test sets of CDBC for drawing actual conclusions on grammar comparison.

[10] For instance, definitions of nonterminals in ANTLR have exactly one production because choices are used instead of multiple productions.

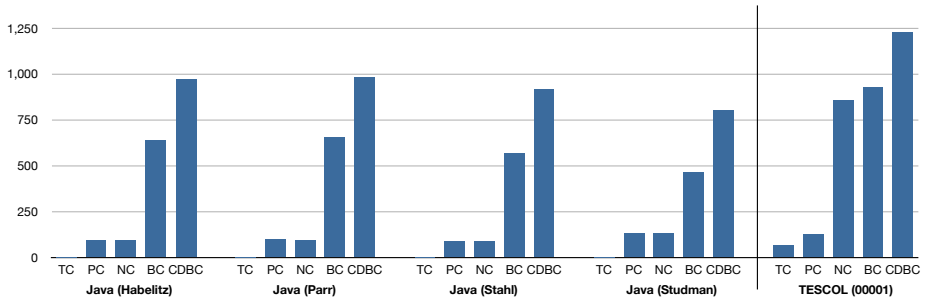


**Fig. 5. Test set sizes.** Amount of test data generated to satisfy trivial, production, nonterminal, branch and context-dependent branch coverage criteria. For comparison, we also show test set sizes for a much smaller grammar of the study in §6 in which case test sets were generated for all nonterminals as opposed to only the start symbol of the Java grammars.

### 4.4 Results

Figure 6 reports on the degree of observed nonequivalence during testing. The blue dots represent acceptance rate for each of the criteria-driven subsets, while the green block behind them reports on all test data together. Let us first examine the diagonal plots which are expected to be equal to 100%, not just close to it. Namely, consider one of the test cases generated from but not parseable with the Habelitz grammar:

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

According to the extracted grammar, switch block labels are defined by a nillable nonterminal aptly called `switchBlockLabels`:

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

However, the original parser specification contained an AST rewriting rule:

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

This rule raises an exception if an attempt is made to rewrite an empty tree, and the unhandled exception is then treated as a failure to parse code. Since the context-free part allows `switchBlockLabels` to be $\varepsilon$, generated test data

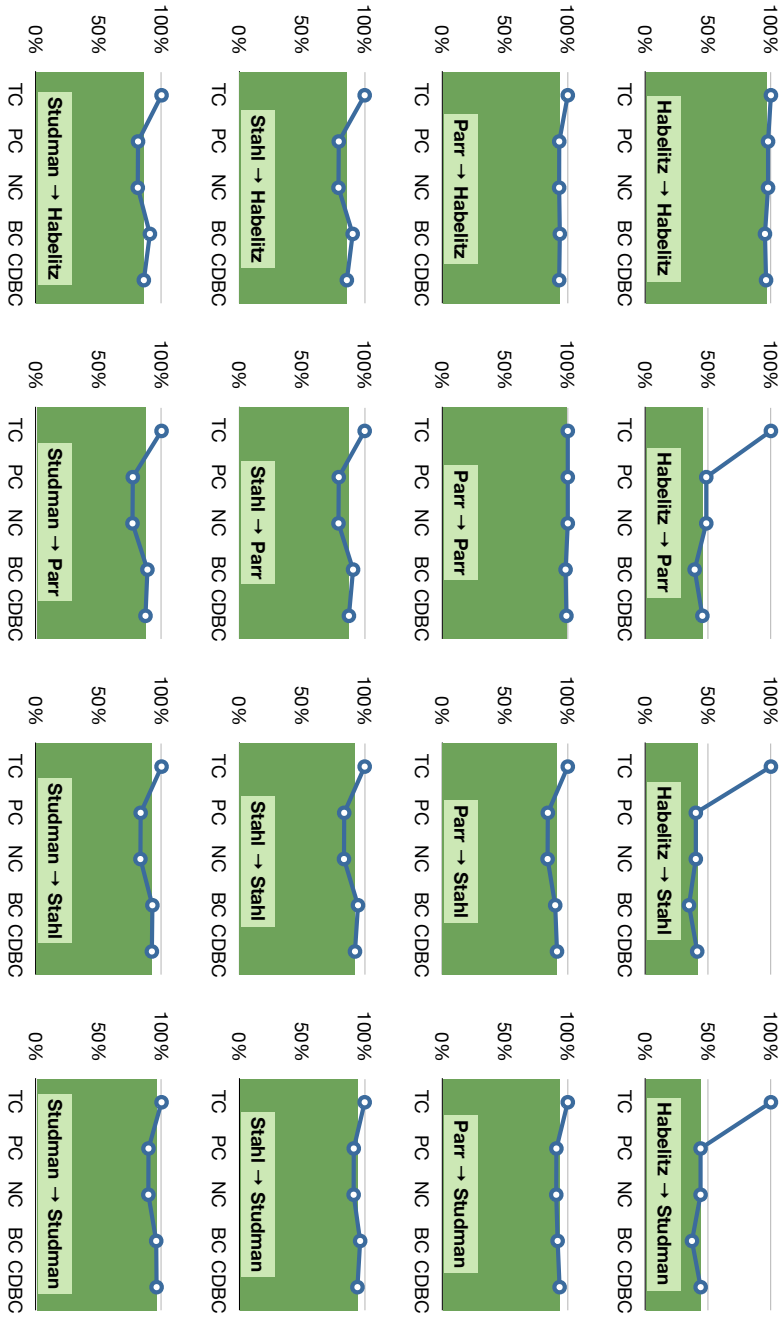


**Fig. 6. Testing Java grammars and parsers.** The Habelitz grammar is apparently much more permissive than the rest. All parsers accept almost all test cases generated from their corresponding grammars (diagonal plots).

explores the option, but the idiosyncrasy with which its structure was originally defined, leads to false nonequivalence reports. It is also worth mentioning that the grammar with the highest self-acceptance rate (99%) is Parr, which was designed by the creator of the ANTLR notation.

From the non-diagonal plots of Figure 6 one can see that the Parr, Stahl and Studman grammars are rather close to one another, but the Habelitz grammar is much more permissive. Indeed, manual cursory examination of the failing test cases shows that the Habelitz parser accepts, among other things:

- `class a < a extends a {}`, `class a < a >> {}`, `class a < a >>> {}` (the piece of grammar dealing with angle brackets is annotated with a "dirty trick" comment)
- `native class a { }` ("native" is a modifier for a method, not for a class)
- `@ a ( ++ 0 )` (annotation followed by neither class nor package declaration)

The last mentioned example is responsible for most of the failures. In fact, the only place we were able to spot where the Habelitz grammar is more restrictive than the rest is enumeration definitions (it does not allow for empty enumerations).

## 5 Matching algorithm

Nonterminals of two given grammars are to be matched. We assume that the grammars are executable in that corresponding parsers are available. In fact, the grammars may have been extracted from the parsers—as discussed above. We start from a test set indexed by nonterminals of one grammar. We apply the parser of the other grammar to the indexed test set while also varying the start symbol so that all nonterminals are exercised. For each parser run with one test case, we get a positive response (meaning that this particular test case has been accepted as valid according to a particular nonterminal) or a negative one (meaning that a parse error occurred, AST building failed, a syntactic or semantic predicate did not hold, etc.).

We can group these results into triples {reference nonterminal, nonterminal under test, percentage of successfully parsed test data}. Such a relation, when displayed in table form with reference nonterminals as rows and nonterminals under test as columns, and when sorted alphabetically, looks like Figure 7 (left). Cells with 0% successes are left blank, up to 25% are yellow, below 75% are blue, up to 99% are green and exactly 100% successes are red.

The results are processed further by making actual matches between nonterminals. First, **universal**$(\cdot, y)$ matches are made by removing nonterminals under test that accept all test data generated by more than 75% of the reference nonterminals. Then, different rules for matching are attempted exhaustively. Each single match is recorded and the matched nonterminals are removed from further checks for the rest of the matching loop. There are the following rules for matches; these options are attempted in the given order for each matching step:

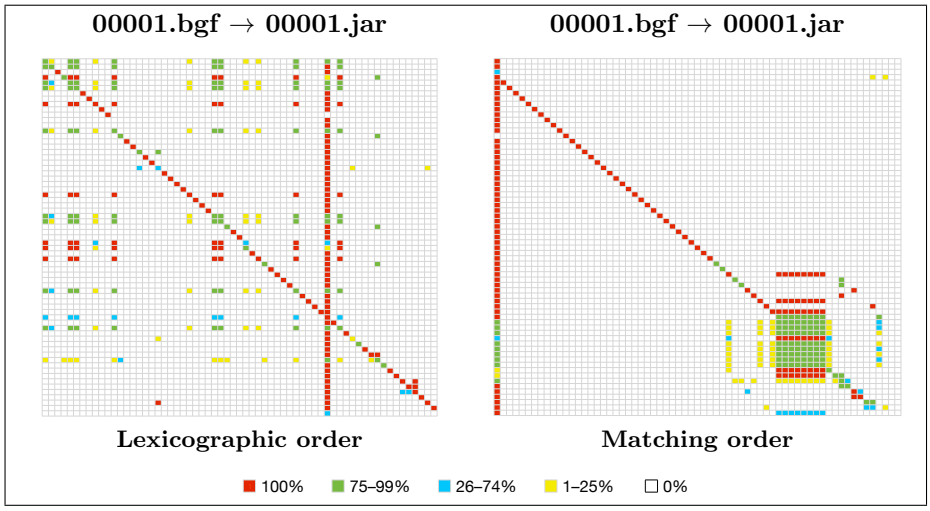


**Fig. 7. Visualized nonterminal matching.** In every color matrix, each row represents a producing nonterminal and each column denotes an accepting nonterminal. On the left color matrix, nonterminals (i.e., rows and columns) are sorted alphabetically; on the right one, in the order of matching.

**void**$(x, \cdot)$ all nonterminals under test accept less than 25% for $x$'s test data.
**perfect**$(x, y)$ $x$ generates test data which can always be parsed by $y$ and never by any other nonterminal, and $y$ also exclusively accepts only $x$'s test data;
**nearlyPerfect**$(x, y)$ $x$ generates test data of which more than 75% can be parsed by $y$ and never by any other nonterminal, and $y$ also exclusively accepts only $x$'s test data;
**exclusive**$(x, y)$ $x$ generates test data which is best parsed by $y$ at more than 75%, and $y$ exclusively accepts only $x$'s test data;
**probable**$(x, y)$ $x$ generates test data which is parsed only by $y$, and acceptance rate is at least 25%;
**block**$(x_i, y_i)$ all $x_i$ yield test data that is well accepted ($> 75\%$) by all $y_i$;
**probableBlock**$(x_i, y_i)$ all $x_i$ yield test data accepted at $> 25\%$ by all $y_i$;
**maximum**$(x, y)$ of all candidates, $y$ has the highest acceptance rate.

If any nonterminals are left once the above rules have been exhausted, then that rest is assumed to match **none**$(x, \cdot)$. If rows and columns of the relation are resorted in the order of matching, we can see a picture like the one on Figure 7 (right). There we see a **universal** match being made, followed by a long series of **perfect** and then **nearly perfect** matches, several **exclusive** matches, a big **block** match and some less reliable matches at the end of the process.

## 6 Nonterminal matching study: course work

TESCOL (TESt COmpiler Language) is an artificial small programming language used by the first coauthor in a compiler engineering course. A TESCOL

program contains a list of semicolon-separated declarations and a single statement. The program starts with the keyword `trolley`, followed by a constant identifier, the keyword `contains`, and the declarations. The statement is separated from the declarations by the keyword `checkout` and followed by a semicolon, the mandatory `done` and another semicolon. There are also some contextual restrictions: global naming scheme, non-recursive procedures, declarations preceding uses, etc.

A class of students was asked to implement TESCOL in ANTLR, resulting in a codebase of 83 grammars claiming to conform to the same language specification. The following actions were part of the preparations of the TESCOL grammar base:

- ANTLR3 grammars were recovered from the submitted tarballs;
- The grammars were extracted as described in §4.2;
- We generated boilerplate Java code for passing a file name and a nonterminal name as parameters for parser runs;
- The code produced by ANTLR from the grammar was compiled together with the boilerplate code to form a JAR;
- The filenames were obfuscated to avoid disclosing students' identities.

In this way we were able to obtain 32 pairs, each consisting of a valid context-free grammar and a runnable JAR with a parser. Each of the remaining 51 grammars contained small errors in the ANTLR productions or the expected interaction protocol which prevented their automated processing in the study of this paper. Each of the working grammars was used to generate test data for all nonterminals it contained. Such a test data set for one grammar consisted of around 1000 test cases (min. 599, max. 1354), distributed among coverage criteria as shown in Figure 5 (right). One test data set took around 5 hours to test against all 2300 nonterminals of available 32 candidate grammars on an Intel Core i7 machine with a 2.80GHz CPU; see also Table 3. The results reported in this paper refer specifically to one test data set for the reference grammar nicknamed 00001, fed into all of the available parsers. The choice of 00001 over other TESCOL grammars was purely incidental.

TESCOL grammars are considerably smaller than Java grammars, having on average four times less top alternatives, three times less nonterminal symbols and almost half less terminals (compare with the values in the table on page 9):

| | PROD | VAR | TERM |
|---|---|---|---|
| **Minimum** | 69 | 54 | 101 |
| **Average** | 85 | 67 | 104 |
| **Maximum** | 126 | 83 | 120 |

Let us return to Figure 7, which we already used for illustration of nonterminal matching. In fact, the two matrices in the figure represent matches of the reference grammar against its own parser. The only universal match is with a nonterminal called `token`, which serves error handling. Void matches for

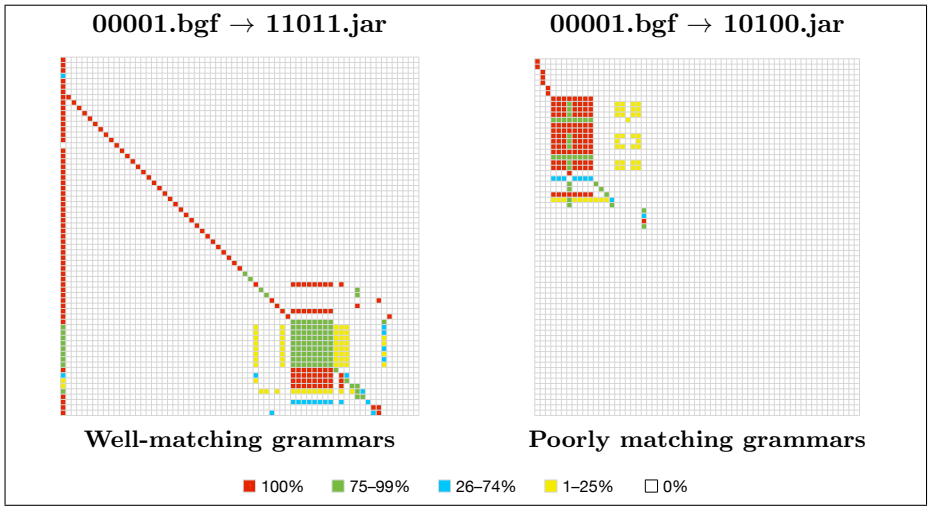


**Fig. 8. Visualized nonterminal matching.** A good match between languages can be seen on the left; a considerably worse one on the right.

`comment`, `COMMENT` and `WS` (whitespace) make sense because of the way how a parser handles, in fact, skips such lexical categories. However, a void match for `procDec` is suspicious; when investigated, we see the same problem encountered earlier in §4.4: a `RewriteEmptyStreamException`.

Nominal inspection of all 50 singular matches shows that they are correct. There are also two group matches: one correct (comprising `expr`, `multExpr`, `compExpr`, `andExpr`, etc, closely related nonterminals from one grammatical level) and one incorrect (`constDec` and `declarations` with themselves). The incorrectness of the latter is a direct consequence of the problem with `procDec`.

Figure 8 shows two more examples of nonterminal matching which we will discuss very briefly. The one on the left is well-matched, with a couple of groups and many perfect matches, most of which could not have been inferred from nominal matching: `MULTI` with `ARITH-MUL`, `NEQ` with `COND-NONEQUAL`, `grstatement` with `statement-group`, etc. The one on the right is matching rather poorly, with 41 nonterminals matching **void** or **none** and the rest being in **block**s.

We have condensed the results of matching all grammars with the reference grammar in Figure 9, where matches are counted based on their type. **Universal**, **void** and **none** belong to a group of usually unwanted matches since they fail to provide any information to the grammar engineer. On the other end, **block** and **probable block** matches give some information which requires more sophisticated heuristics or human interpretation. The remaining matches are singular: one reference nonterminal matches with one nonterminal under test. As it becomes apparent from the diagram, **perfect**, **nearly perfect**, **exclusive**, **probable** and **maximum** matches cover the majority of reference

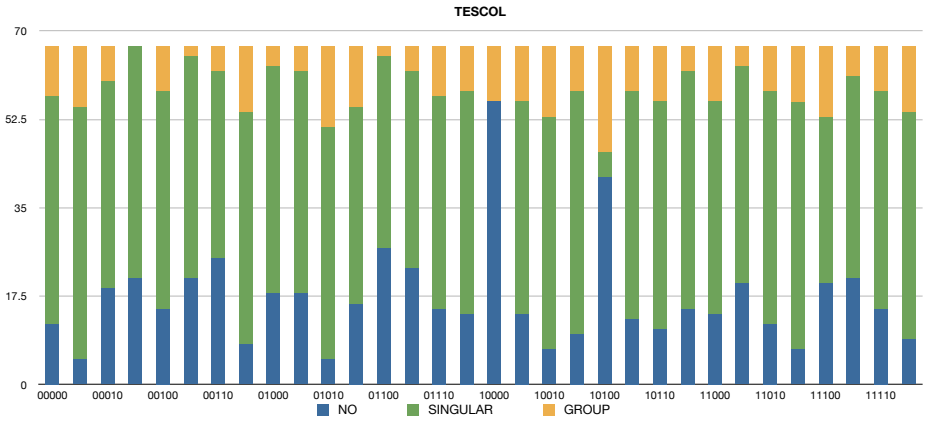


**Fig. 9. TESCOL nonterminal matching.** Blue (dark grey) bar parts denote nonterminals that did not match anything (universal, void, none); green (grey) denotes nonterminals for which a match was found (perfect, nearly perfect, exclusive, probable, maximum); yellow (light grey) is for nonterminals which were matched in a group (block, probable block).

nonterminals. Group matches also provide useful and adequate results. Hence, nonterminal matching is successful in the context of the study.

## 7 Related Work

§2 already provided some general background on the established topic of grammar-based testing; we refer to [3,7,8,12,14,15,18,19,20,25] for extensive discussion of methods and applications of grammar-based testing. Our work is original in so far that we are the first to actually use grammar-based testing for the comparison of grammars. Usually, grammar-based testing is used to test parsers or compilers.

In both studies in §4 and §6, we have noticed imperfect self-matching and explained reasons for it. One of the ways to improve on this issue would be to take into account the constraints expressed by the parser specification. There are related methods of extending grammar-based testing to attribute grammars [6,10].

In our current development, we do not yet leverage any sort of negative test data generation. There are grammar-based testing scenarios that clearly benefit from inclusion of negative test cases [29]. For instance, a parser for which no grammar-based parser description is available can only be tested for *completeness* with regard to reference grammar with positive test cases whereas testing for *correctness* would require negative test cases. In our comparison-based context of the present paper, negative test data is "less important" because evidence of both non-completeness and non-correctness can be found with the help of positive test cases that are obtained from the compared grammars; see again §2.

Grammar nonequivalence is a well-known undecidable problem. One related problem is the status of a grammar to be ambiguous (or not). Some sort of testing has been successfully applied though in this context [2]. Another related problem is grammar-class/non-ambiguity preservation under composition. While context-free grammars can always be combined together to form new context-free grammars, smaller subclasses related to specific parsing technology (or to the requirement of non-ambiguity) usually do not exhibit this property. Several attempts to provide painless language modularity are known, such as Kiama [26], Silver/Copper [28], language boxes methodology [23], etc. Grammar comparison-like methods may be potentially useful in supporting safe composition.

## 8 Conclusion

We have developed and demonstrated an approach to grammar comparison which relies on systematic grammar-based test data generation and parsing. We have shown, in particular, that the approach can be used for revealing differences between substantially large grammars and for matching many grammars. We conclude with a discussion of future work.

The results of nonterminal matching turn out to be useful based on our nominal inspection. Further research is needed to see how the information that is derived from nonterminal matching can be usefully consumed by grammar engineers for different scenarios. For instance, someone who likes to converge two grammars may need to turn the matches into appropriate transformations.

We already mentioned the possibility of generating negative test cases. In theory, more evidence can be found by applying parsers to negative test cases. Whether or not this evidence makes a difference in practical scenarios like ours is an open question.

There is also the related question whether we can improve precision of matching by generating larger test sets for more demanding coverage criteria. While it may lead to bad scalability to universally replace CDBC by a more demanding criterion, a more selective approach could be scalable enough: generate more test data when about to match a block; see §5.

Our implementation leaves much room for optimization. As apparent from Table 3, the generation phase is not a problem: it is required only once, and takes only a few minutes. However, our current infrastructure for parser execution loops over test cases such that the parser is run separately for each test case, causing excessive overhead with loading and unloading in the JVM. The computation of the results of the present paper relied on parallelism/distribution.[11]

### Acknowledgments

Bernd Fischer was supported by EPSRC grant EP/F052669/1.
Robert van Liere (CWI, Amsterdam) provided expert advice on visualization.

[11] We used several machines at the CWI SWAT department. The estimated, sequential time to run all TESCOL-based test data against all parsers is 300 days.

| | generate | | | | | unparse | run | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Test set | 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 | — | |

**Table 3. Performance.** Time (in minutes, seconds and, if necessary, hours) to generate test data, unparse it (turn parse trees to source code), and run. Generation was measured separately for satisfying trivial, production, nonterminal, branch and context-dependent branch coverage criteria.

## References


1. Aho, A.V.: Teaching the Compilers Course. SIGCSE Bull. 40, 6–8 (Nov 2008)
2. Basten, H.J.S.: Tracking Down the Origins of Ambiguity in Context-free Grammars. In: Proceedings of the 7th International colloquium conference on Theoretical aspects of computing. pp. 76–90. ICTAC'10, Springer-Verlag, Berlin, Heidelberg (2010)
3. Burgess, C.J.: The Automated Generation of Test Cases for Compilers. Software Testing, Verification and Reliability 4(2), 81–99 (Jun 1994)
4. Gosling, J., Joy, B., Steele, G.L., Bracha, G.: The Java Language Specification. Addison-Wesley, third edn. (2005), all versions of the JLS are available at http://java.sun.com/docs/books/jls
5. Griswold, W.G.: Teaching Software Engineering in a Compiler Project Course. Journal on Educational Resources in Computing 2 (Dec 2002)
6. Harm, J., Lämmel, R.: Two-dimensional Approximation Coverage. Informatica 24(3) (2000)
7. Hennessy, M., Power, J.F.: Analysing the Effectiveness of Rule-coverage as a Reduction Criterion for Test Suites of Grammar-based Software. Empirical Software Engineering 13, 343–368 (August 2008)
8. Hoffman, D., Wang, H.Y., Chang, M., Ly-Gagnon, D., Sobotkiewicz, L., Strooper, P.: Two Case Studies in Grammar-based Test Generation. Journal of Systems and Software 83, 2369–2378 (December 2010)
9. IBM Corporation: VS COBOL II Application Programming Language Reference, 4th edn. (1993), Publication number GC26-4047-07
10. Kastens, U.: Studie zur Erzeugung von Testprogrammen für Übersetzer. Bericht 12/80, Institut für Informatik II, University Karlsruhe (1980)
11. Klint, P., van der Storm, T., Vinju, J.: EASY Meta-programming with Rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) Post-proceedings of GTTSE 2009. LNCS, vol. 6491, pp. 222–289. Springer-Verlag (January 2011)
12. Kossatchev, A.S., Posypkin, M.A.: Survey of Compiler Testing Methods. Programming and Computing Software 31, 10–19 (January 2005)

13. Lämmel, R., Verhoef, C.: VS COBOL II grammar Version 1.0.4 (1999), available at: http://www.cs.vu.nl/grammarware/browsable/vs-cobol-ii/
14. Lämmel, R.: Grammar Testing. In: Hussmann, H. (ed.) Proceedings of Fundamental Approaches to Software Engineering (FASE'01). LNCS, vol. 2029, pp. 201–216. Springer-Verlag (2001)
15. Lämmel, R., Schulte, W.: Controllable Combinatorial Coverage in Grammar-Based Testing. In: Uyar, U., Fecko, M., Duale, A. (eds.) Proceedings of the 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems (TestCom'06). LNCS, vol. 3964, pp. 19–38. Springer Verlag (2006)
16. Lämmel, R., Zaytsev, V.: An Introduction to Grammar Convergence. In: Proceedings of iFM. LNCS, vol. 5423, pp. 246–260. Springer (2009)
17. Lämmel, R., Zaytsev, V.: Recovering Grammar Relationships for the Java Language Specification. Software Quality Journal 19(2), 333–378 (2011)
18. Malloy, B.A., Power, J.F.: An Interpretation of Purdom's Algorithm for Automatic Generation of Test Cases. In: In 1st Annual International Conference on Computer and Information Science. pp. 3–5 (2001)
19. Maurer, P.: Generating Test Data with Enhanced Context-free Grammars. IEEE Software 7(4), 50–56 (1990)
20. McKeeman, W.M.: Differential Testing for Software. Digital Technical Journal of Digital Equipment Corporation 10(1), 100–107 (1998)
21. Power, J.F., Malloy, B.A.: A Metrics Suite for Grammar-based Software. Journal of Software Maintenance and Evolution: Research and Practice 16, 405–426 (November 2004)
22. Purdom, P.: A Sentence Generator for Testing Parsers. BIT 12(3), 366–375 (1972)
23. Renggli, L., Denker, M., Nierstrasz, O.: Language Boxes: Bending the Host Language with Modular Language Changes. In: van den Brand, M., Gaševic, D., Gray, J. (eds.) Software Language Engineering, LNCS, vol. 5969, pp. 274–293. Springer Berlin / Heidelberg (2010)
24. Schwartzbach, M.I.: Design Choices in a Compiler Course or How to Make Undergraduates Love Formal Notation. In: Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction. pp. 1–15. CC'08/ETAPS'08, Springer-Verlag (2008)
25. Sirer, E.G., Bershad, B.N.: Using Production Grammars in Software Testing. SIGPLAN Notices 35, 1–13 (December 1999)
26. Sloane, A.M., Kats, L.C.L., Visser, E.: A Pure Object-Oriented Embedding of Attribute Grammars. In: Ekman, T., Vinju, J. (eds.) Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009). Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers (2009)
27. Waite, W.M.: The Compiler Course in Today's Curriculum: Three Strategies. In: Proceedings of the 37th SIGCSE technical symposium on Computer science education. pp. 87–91. SIGCSE '06, ACM (2006)
28. van Wyk, E., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute Grammar-based Language Extensions for Java. In: European Conference on Object Oriented Programming (ECOOP). LNCS, vol. 4609. Springer Verlag (2007)
29. Zelenov, S., Zelenova, S.: Automated Generation of Positive and Negative Tests for Parsers. In: Grieskamp, W., Weise, C. (eds.) Formal Approaches to Software Testing, LNCS, vol. 3997, pp. 187–202. Springer Berlin / Heidelberg (2006)