Grammar-Based Testing for Little Languages

An Experience Report with Student Compilers

Phillip van Heerden
Stellenbosch University
Stellenbosch, South Africa
18962378@sun.ac.za

Moeketsi Raselimo
Stellenbosch University
Stellenbosch, South Africa
22374604@sun.ac.za

Konstantinos Sagonas
Uppsala University
Uppsala, Sweden
kostis@it.uu.se

Bernd Fischer
Stellenbosch University
Stellenbosch, South Africa
bfischer@sun.ac.za

Abstract

We report on our experience in using various grammar-based test suite generation methods to test 61 single-pass compilers that undergraduate students submitted for the practical project of a computer architecture course.

We show that (1) all test suites constructed systematically following different grammar coverage criteria fall far behind the instructor's test suite in achieved code coverage, in the number of triggered semantic errors, and in detected failures and crashes; (2) a medium-sized positive random test suite triggers more crashes than the instructor's test suite, but achieves lower code coverage and triggers fewer non-crashing errors; and (3) a combination of the systematic and random test suites performs as well or better than the instructor's test suite in all aspects and identifies errors or crashes in every single submission.

We then develop a light-weight extension of the basic grammar-based testing framework to capture contextual constraints, by encoding scoping and typing information as "semantic mark-up tokens" in the grammar rules. These mark-up tokens are interpreted by a small generic core engine when the tests are rendered, and tests with a syntactic structure that cannot be completed into a valid program by choosing appropriate identifiers are discarded. We formalize individual error models by overwriting individual mark-up tokens, and generate tests that are guaranteed to break specific contextual properties of the language. We show that a fully automatically generated random test suite with 15 error models achieves roughly the same coverage as the instructor's test suite, and outperforms it in the number of triggered semantic errors and detected failures and crashes. Moreover, all failing tests indicate real errors, and we have detected errors even in the instructor's reference implementation.

CCS Concepts: • **Software and its engineering** → *Parsers*; **Software testing and debugging**.

Keywords: Structure-aware fuzzing, semantic fuzzing, property-based testing, random testing.

ACM Reference Format:
Phillip van Heerden, Moeketsi Raselimo, Konstantinos Sagonas, and Bernd Fischer. 2020. Grammar-Based Testing for Little Languages: An Experience Report with Student Compilers. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3426425.3426946

1 Introduction

Compilers for "little languages" [2] such as domain-specific languages or languages used in compiler courses still need testing. Since little languages typically lack a large user base, the construction of appropriate test suites falls back to their developers, which puts a large burden on them.

Such test suites can also be generated automatically from a context-free grammar (CFG) for the language. This is the basic tenet of grammar-based testing and fuzzing. However, fuzzing is primarily aimed at finding crashes and not functional errors, and grammars play only a minor role in (functional) software testing [11]. It is therefore unclear to what extent grammar-based testing is suitable for little languages.

In this paper, we address this issue. We report on our experience in using a variety of test suites automatically generated from CFGs to test simple compilers written by a cohort of 61 undergraduate students, and compare their performance (measured both in achieved code coverage and number of failures and crashes triggered) against the instructor's evaluation and marking test suite.

SLE '20, November 16–17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8176-5/20/11…$15.00
https://doi.org/10.1145/3426425.3426946

The ultimate question behind our research is whether we can automatically generate from a grammar a test suite of the same quality as a carefully hand-crafted one. We first focus on *purely syntactic* test suites where we only have a *syntactic oracle*, i.e., only know whether the tests are syntactically valid or not, and experimentally investigate how well different purely syntactic test suite constructions perform. In particular, we address two research questions:

RQ1: Do purely syntactic test suites have the same quality as a hand-crafted test suite?

RQ2: Do purely syntactic test suites systematically constructed by the cover algorithm achieve better system coverage and find more errors than purely syntactic random test suites of similar size?

To address RQ1 and RQ2, we generate test suites using a generic cover algorithm with different grammar coverage criteria, including the new *deriv*-criterion that we introduce in this paper (Definition 2.1). In Section 4, we show that none of these test suites achieves code coverage above 68%, which is considered to be insufficient in software engineering, and that individually they all fall far behind the instructor's test suite in the achieved code coverage and the numbers of triggered semantic errors and detected failures and crashes. However, a combined positive/negative test suite triggers failures in and crashes every single submission, and thus outperforms the instructor's test suite in that respect.

In the same section, we show that random test suites of the same size outperform systematically generated test suites for all but the smallest sizes. We also show that larger random test suites push code coverage up to 75% and outperform the instructor's test suite in the number of crashes triggered, but underperform it in the number of non-crashing errors. Finally, a combination of the systematic and random test suites performs in all aspects as well as or better than the instructor's test suite.

We then address the main limitation of syntactic test suites, the lack of a more precise *semantic oracle*. We can thus use these test suites only for syntax and *crash testing*. Any *functional testing* requires tests that either conform to the contextual constraints of the language—principally, its scoping and typing rules—or break them in a well-defined way. We address this issue with two further research questions:

RQ3: How do we encode contextual constraints into the grammar and integrate them with test suite generation to create semantic test suites?

RQ4: How do we generate tests with specific failure modes?

In Section 5, we develop a light-weight extension of the basic grammar-based testing framework to capture contextual constraints, by encoding scoping and typing information as "semantic mark-up tokens" in the grammar rules. These mark-up tokens are interpreted by a small generic core engine when the tests are rendered, and tests with a syntactic structure that cannot be completed into a valid program by choosing appropriate identifiers are discarded. We formalize individual error models by overwriting individual mark-up tokens, and generate tests that are guaranteed to break specific contextual properties of the language. We show that a fully automatically generated random test suite with 15 error models achieves roughly the same coverage as the instructor's test suite, and outperforms it in the number of detected failures and crashes. Moreover, all failing tests indicate real errors, and we have detected errors in the instructor's reference implementation.

2 Grammar-Based Test Suite Construction

A context-free *grammar* is a four-tuple $G = (N, T, P, S)$ with $N \cap T = \emptyset$, $V = N \cup T$, $P \subset N \times V^*$, and $S \in N$. We use $A, B, C, \ldots$ for non-terminals N, $a, b, c, \ldots$ for terminals T, X, Y, Z for *grammar symbols* V, w, x, y, z for *words* over T^*, and $\alpha, \beta, \gamma, \ldots$ for *sentential forms* over V^*, with ε denoting the empty string. We write $A \to \gamma$ for a rule $(A, \gamma) \in P$. We use $\alpha A \beta \Rightarrow \alpha\gamma\beta$ to denote that $\alpha A \beta$ produces $\alpha\gamma\beta$ by application of the rule $A \to \gamma \in P$ and use $\Rightarrow^*$ for its reflexive-transitive closure.

For our evaluation, we use test suites that satisfy several different grammar coverage criteria such as *rule* and *cdrc* coverage [18]. We use both positive (i.e., syntactically correct) and negative tests; the latter are guaranteed to contain exactly one syntax error [27].

For the construction of positive tests we use a generic cover algorithm that follows the approaches proposed by Fischer et al. [9] and Havrikov and Zeller [11]. Its basic idea is to (i) iterate over all symbols $X \in V$, (ii) embed X, i.e., compute a minimal derivation $S \Rightarrow^* \alpha X \omega$, (iii) cover X, i.e., compute a set of minimal derivations $X \Rightarrow^* \gamma$ that conform to the criterion, and (iv) convert the sentential form into a word, i.e., compute a minimal derivation $\alpha\gamma\omega \Rightarrow^* w$ where each non-terminal A in $\alpha\gamma\omega$ is replaced by its minimal yield w_A. Note that this algorithm is by construction biased towards very short tests because it uses of minimal derivations in all steps. Note further that different covering derivations may still lead to the same test case.

We use the standard coverage criteria *symbol*, *rule*, and *cdrc* [18] as well as their extension to k-step derivations (or *k-path coverage* [11]) as arguments to the cover algorithm. In addition, we also use *derivable pair coverage*, which can intuitively be thought of as a fixpoint version of k-step coverage, since it covers the shortest derivation between any two symbols irrespectively of its length.

Definition 2.1 (Derivable Pair Coverage)**.** Let $G = (N, T, P, S)$ be a grammar and $V = N \cup T$, A word w covers a pair $(X, Y) \in V \times V$ if $S \Rightarrow^* \alpha X \beta \Rightarrow^* \alpha\gamma Y \delta\beta \Rightarrow^* w$. $TS \subseteq L(G)$ satisfies derivable pair coverage (w.r.t. G) if each pair (X, Y) with $X \Rightarrow^* \mu Y \nu$ for some μ, ν is covered by a word $w \in TS$.

Finally, we also generate tests according to a simultaneous derivation relation $\Rrightarrow$, where $X_1 \ldots X_n \Rrightarrow \gamma_1 \ldots \gamma_n$ if

$X_i \to \gamma_i \in P$ for all $X_i \in N$ and $\gamma_i = X_i$ for all $X_i \in T$. We also denote its k-fold repetition $\Rrightarrow^k$ by bfs_k because it amounts to k "breadth-first rounds" of rule applications. In combination with the cover algorithm sketched above, bfs_k explores deeper derivations than a normal breadth-first search, because separate "bfs waves" are started from each non-terminal symbol.

As common in grammar-based fuzzing, we also use random derivations. More specifically, we construct a random subset of the $\Rrightarrow^k$ derivations, which allows us to explore longer derivations than full bfs_k. After k iterations, we replace the unexpanded non-terminals with their minimal yield, as in the generic cover algorithm. Note that this setup may introduce some bias, in particular towards the rules used in the yield construction. In addition, we use a modified derivation construction where a proposed rule is only applied with a user-defined rule-specific probability; if it is rejected, the non-terminal remains as is, but may be expanded in the next iteration.

For the construction of syntactically negative tests, we use token- and rule-mutation algorithms [27]. We extended token mutation to use arbitrary n-grams rather than poisoned pairs (i.e., bigrams).

We generally do not address lexical aspects in the test suites and, unless stated otherwise, the generated programs do not contain comments or whitespace other than a single space between tokens to prevent accidental token pasting, and all structured tokens are expanded into their shortest instance (e.g., all `id`s are `a`).

3 Context and Experimental Setup

For our evaluation, we used the compilers for a small imperative language that second-year computer science students developed as practical project in a mandatory computer architecture course at Stellenbosch University. The project ran throughout the entire semester and accounted for 25% of the overall course marks. We could not access individual project marks, but the average course mark of the 61 students in our cohort is 80%. The instructor has taught the course with the same general set-up for more than five years, and has made only minor changes to the syntax and semantics of the example languages.

The course uses compilers to illustrate low-level programming concepts like memory allocation and addressing, byte code and assembly code, *etc.* The students must therefore use the C programming language. Since this is their first exposure at university to C, they typically struggle with this course. Below, we give some details about the setup.

Language. Figure 1 shows the EBNF grammar of the AMPL anguage used for the project. Terminal symbols are typeset in **`bold typewriter`** and non-terminals in *italics*; structured tokens are specified using a lex-like notation and typeset in `normal typewriter`.

prog → **`program`** `id` **`:`** *fdef** **`main`** **`:`** *body*
fdef → `id` **`:`** **`takes`** *vdecl* (**`;`** *vdecl*)*
`returns` (*type* | **`nothing`**) *body*
vdecl → `id` (**`,`** `id`)* **`:`** *type*
type → (**`boolean`** | **`integer`**) **`array`** ?
body → *decls*? *stmts* **`end`**
decls → *vdecl* (**`;`** *vdecl*)*
stmts → **`chillax`** | *stmt* (**`;`** *stmt*)*
stmt → *assign* | *call* | *return* | *input* | *output* | *case* | *loop*
assign → **`let`** `id` (**`[`** *simple* **`]`**)? **`=`** (*expr* | **`array`** *simple*)
call → **`do`** `id` **`(`** *expr* (**`,`** *expr*)* **`)`**
return → **`pop`** *expr*?
input → **`input`** `id` (**`[`** *simple* **`]`**)?
output → **`output`** (`string` | *expr*) (**`.`** (`string` | *expr*))*
case → **`when`** **`case`** *expr* **`:`** *stmts* **`end`**
(**`case`** *expr* **`:`** *stmts* **`end`**)*
(**`otherwise`** **`:`** *stmts* **`end`**)?
loop → **`while`** *expr* **`:`** *stmts* **`end`**
expr → *simple* (*relop simple*)?
relop → **`=`** | **`>=`** | **`>`** | **`<=`** | **`<`** | **`/=`**
simple → **`-`**? *term* (*addop term*)*
addop → **`-`** | **`or`** | **`+`**
term → *factor* (*mulop factor*)*
mulop → **`and`** | **`/`** | **`*`** | **`rem`**
factor → `id` (**`[`** *simple* **`]`** | **`(`** *expr* (**`,`** *expr*)* **`)`**)?
| `num` | **`(`** *expr* **`)`** | **`not`** *factor* | **`true`** | **`false`**
`id` → `[a-zA-Z_][a-zA-Z_0-9]*`
`num` → `[0-9][0-9]*`
`string` → `"([a-zA-Z_0-9 !#-/:-?] | \" | \t | \n | \\)*"`

Figure 1. AMPL grammar as specified by the instructor.

AMPL has only the basic imperative control-flow constructs (albeit in an idiosyncratic syntax), including procedure and function definitions and calls, an elementary but strict type system with two base types (without implicit type coercions) and one-dimensional arrays over these, and a rudimentary visibility system. There is only a single namespace, and multiple definitions with different types are not allowed. Procedures (resp. functions) can only be declared at the top-level and ahead of global variables. Procedures and global variables are in the same global scope; parameters and procedure-local variables are in the same local scope, and local names cannot shadow global names.

Task. The overall goal of the project is to implement in C a compiler from AMPL into Java bytecode suitable for the JASMIN bytecode assembler (http://jasmin.sourceforge.net/).

The students were provided with a 60-page language specification document that contains the CFG for AMPL in EBNF. The contextual constraints, in particular the scoping and typing rules, are specified informally and illustrated with examples.

The project prescribed a rigid approach and environment. The students were required to implement from scratch a one-pass, recursive descent compiler that aborts on the first error it encounters, irrespective of whether this is at the lexical,

Syntax errors:

```
(a)  <token> expected, but found <token>
(b)  type expected, but found <token>
(c)  statement expected, but found <token>
(d)  factor expected, but found <token>
(e)  array allocation or expression expected, but found <token>
(f)  expression or string expected, but found <token>
```

Scope errors:

```
(g)  multiple definition of <id>
(h)  the identifier <id> is unknown
```

Type errors:

```
(i)  <id> is not a function
(j)  <id> is not a procedure
(k)  <id> is not a variable
(l)  <id> is not an array
(m)  <op> is an illegal array operation
(n)  scalar expected instead of <id>
(o)  incompatible types for array allocation
(p)  incompatible types for array index of <id>
(q)  incompatible types for array size of <id>
(r)  incompatible types for assignment to <id>
(s)  incompatible types for case guard
(t)  incompatible types for while guard
(u)  incompatible types for operator <op>
(v)  incompatible types for unary minus
(w)  incompatible types for not
(x)  incompatible types for parameter <n> of <id>
(y)  too few arguments for call of <id>
(z)  too many arguments for call of <id>
```

Figure 2. AMPL error messages as specified by instructor.

syntactic, or semantic level. Scaffolding code (in particular error handlers) and architectural skeletons were given by the instructor. Figure 2 lists all expected error messages. Their use was enforced by an automated build system and marking script, so that the project submissions all have the same standard architecture: i) a main module containing the parse functions with the corresponding semantic actions; ii) a scanner; iii) a symbol table; and iv) a code generator that emits the bytecodes.

Subject Programs. The overall course enrollment was 94 students. We removed 33 submissions that did not build or run in the prescribed environment. The experimental basis for our evaluation is therefore 61. The average size of each student submission is approximately 1300 lines of C code.

Baseline Test Suite. We compare our automatically generated test suites (see Section 4 and Section 5 for details) against the test suite manually constructed by the instructor. This test suite contains 48 tests with syntax errors and 182 syntactically valid tests, of which 103 satisfy all contextual constraints. It includes all tests used for marking, but was not released to the students before submission. It was developed over several years, adapting the tests to the annual syntax and semantics variations.

Data Collection. We used the instructor's build script to compile the submissions with GCC 8.3.0 and used gcov 8.3.0 to collect coverage information for each individual submission; we report the aggregate coverage over test suites, not over individual tests. The AMPL compiler runtimes varied over the different submissions and test suites, but are on average between 0.1 and 0.2 seconds per test. Using test suites with several thousands of tests therefore poses no problem, but much larger test suites clearly induce a heavy compuational load. We performed the experiments in a Docker container to ensure consistent results. We also collected the error outputs to measure "semantic" coverage and to identify which part of the implementation causes errors. We manually inspected outputs not conforming to the required error message format in Fig. 2 and corrected for near misses; we took all other messages as evidence of a system crash.

4 Syntactic Testing

We call a test suite construction method *syntactic* if the only guaranteed claim that its tests allow is whether they are syntactically valid or not, i.e., if it embodies a *syntactic oracle*. In this section, we experimentally investigate how well different purely syntactic test suites perform. The results of our experiments are summarized in Table 1 and Fig. 3; we discuss them in more detail below.

4.1 Systematic Coverage-Based Positive Test Suites

Grammar-based testing and fuzzing makes a high-level assumption, namely that better *input space* coverage gives better *system* coverage [11]. We look at this assumption and investigate how the positive test suites (i.e., containing only syntactically correct tests) perform, both relative to each other and compared to the positive tests from the instructor's test suite (denoted as $instr^+$). We therefore address two refinements of RQ1 (see Section 1):

RQ1a: What improvement in system-wide code coverage and error detection do we get for larger systematic coverage of the grammar?

RQ1b: Which systematic grammar coverage criteria give the best results relative to the size of the test suite?

Test Suites. We generated test suites using the generic cover algorithm of Section 2 with different criteria. We use k-step coverage for $0 \leq k \leq 4$ (denoted by *symbol*, *rule*, *cdrc*, $step_3$, and $step_4$, respectively), derivable pair coverage (Definition 2.1) denoted by *deriv*, and k-level full breadth-first search coverage for $k = 2$ and $k = 3$ (denoted by bfs_2 and bfs_3, respectively). *lexical* denotes the full two-level bfs coverage over a grammar version where the regular expressions were translated into BNF rules as well; this generates more varied identifiers and strings (e.g., using all escape sequences), but only in their fixed embedding positions, as determined by the cover algorithm. For this test suite, we also generate comments in the test programs, making it the only one to exercise these parts of the scanner.

Note that in particular the test suites from covering "short" k-step derivations (i.e., $k \leq 2$) are very small: even *cdrc* has fewer tests (79) than the grammar has rules (89), despite the fact that it requires the application of each rule in each context. Moreover, due to way the cover algorithm works,

Table 1. Results of syntactic tests. M denotes the number of tests, length their average length in tokens. Line and branch coverage are measured by gcov; error is the average number of different semantic (for positive testsuites) resp. syntax (for negative testsuites) errors triggered. Pass and Failures denote the number of submissions passing all tests in the suite respectively failing on at least one test. For positive tests, front refers to the call of the error handler with a syntax error message (see Fig. 2); for negative tests, it refers to situations where no syntactic or semantic errors (which can pre-empt syntactic errors due to the single-pass implementation) are called, crash refers to an unspecified system crash. Note that submissions can produce different failures over the individual tests in a suite; in these cases, they are counted towards all failure categories, and pass and front/crash failure numbers do not add up to 61. Revealing Tests denotes the average relative number of tests causing at least one failure in the respective category; front and crash are as before, while table refers to test failures due to calls from the symbol table module to the error handler with scope or type error messages.

	Size		Coverage			Pass	Failures		Revealing Tests (%)		
Testsuite	M	length	line (%)	branch (%)	error (#)		front	crash	front	table	crash
symbol	37	10.1	60.7	46.3	1.1	18	8	39	0.9	12.3	19.9
rule	46	10.8	62.9	48.6	1.1	18	11	39	1.2	15.5	20.1
cdrc	79	11.8	63.9	50.3	1.1	17	12	41	1.5	19.9	20.7
$step_3$	180	13.6	65.2	51.5	1.1	15	18	42	1.9	24.7	21.1
$step_4$	427	14.6	66.4	53.3	1.3	15	19	43	2.7	26.7	21.6
deriv	449	14.3	66.7	55.6	1.3	13	20	44	3.1	26.3	22.3
bfs_2	148	12.2	64.2	50.8	1.1	16	14	41	1.6	23.1	20.7
bfs_3	4571	16.0	67.3	53.6	1.5	13	20	43	1.6	31.8	24.1
lexical	447	9.0	67.0	55.5	1.2	16	14	41	0.5	6.4	19.1
$instr^+$	182	33.8	75.6	67.3	3.2	6	50	49	3.7	13.4	22.9
$DL_2(rule)$	41874	12.1	62.1	55.1	6.6	15	44	42	90.0	5.5	4.3
$DL_3(rule)$	43350	12.2	62.2	55.2	6.6	15	44	42	90.0	5.5	4.3
rule-mutation	9971	12.6	61.8	53.4	6.9	16	45	42	80.0	12.7	7.2
$instr^-$	48	14.7	56.1	45.9	6.4	24	36	34	90.0	1.4	8.4
combined	10867	12.5	74.5	67.6	7.0	0	55	46	73.6	13.0	8.3
instr	230	29.8	78.7	71.6	9.0	5	51	50	22.0	10.7	19.8
$random_{symbol}$	37	10.1	60.9	47.4	0.8	17.0	8.0	40.0	0.8	10.6	19.3
$random_{rule}$	46	10.8	63.2	49.6	0.8	17.6	10.4	40.0	0.9	12.1	20.8
$random_{cdrc}$	79	11.8	64.9	52.5	1.1	17.0	12.4	41.0	1.1	16.2	21.2
$random_{step_3}$	180	13.6	66.2	53.7	0.8	14.0	18.0	43.5	1.6	21.0	21.6
$random_{step_4}$	427	14.6	67.7	55.9	0.8	13.8	19.4	43.4	2.4	23.7	22.2
$random_{deriv}$	449	14.3	67.7	58.0	0.8	13.0	20.2	44.0	2.8	25.0	23.0
$random_{bfs_2}$	148	12.2	65.0	52.8	1.1	16.0	14.2	41.0	1.4	20.9	21.0
$random_{bfs_3}$	4571	16.0	67.9	55.7	1.1	13.0	22.0	44.0	2.3	31.2	24.4
$random_{500}$	7.5	86.2	47.4	34.8	0.9	16.6	9.6	41.4	4.3	29.5	42.4
$random_{5000}$	72.5	69.9	68.6	56.1	1.3	11.8	20.8	46.8	5.6	28.3	40.9
$random_{50000}$	648.1	77.2	72.6	63.9	1.8	9.8	22.0	49.2	5.3	28.0	41.5
$random_{500000}$	6531.4	76.6	75.0	68.6	3.0	7.4	24.0	51.6	5.2	27.7	41.2
combined/$random_{500000}$	17410.3	36.5	78.6	73.4	8.2	0.0	54.8	52.0	47.8	18.5	20.7

these test suites contain mostly short programs. For example, `program a : main : chillax end`, which is the shortest possible AMPL program, already covers seven terminals and four non-terminals for *symbol*, and five rules for *rule*.

Longer derivations achieve better system coverage. Our experiments confirm the high-level assumption both for the narrower k-step and the wider k-level bfs derivation schemes. The first block of Table 1 shows an increase in the system-wide code coverage (both at line and branch level) with an increase in the derivation length k. For example, *symbol* (i.e., $step_0$) has a noticeably lower line (60.7%) and branch coverage (46.3%) than $step_4$ (66.4% and 54.8%, respectively).

The boxplots in Fig. 3 show overlaps in the distributions but the increases are in most cases statistically significant as pairwise paired t-tests over all test suites show.[1] This is hardly surprising, as $step_k$ and bfs_k are both contained in $step_{k+1}$ and bfs_{k+1}, respectively. The only exceptions are between $step_4$ and *deriv* and between *lexical* and $step_4$, *deriv*, and bfs_3, respectively, where we cannot reject the null hypothesis for line coverage. In the former case, however, branch coverage *is* statistically significantly higher for *deriv* than

[1]The shape of the boxplots in Fig. 3 indicates that we can approximate the code coverage measurements sufficiently well with a normal distribution. We consider one test suite as baseline and the other as treatment, and pair up the respective results over the underlying student cohort. The null hypothesis is that both coverage metrics are drawn from the same distribution; its rejection means that the observed mean coverage values are statistically significantly different. We consider this at significance levels of $p < 0.05$, as usual.

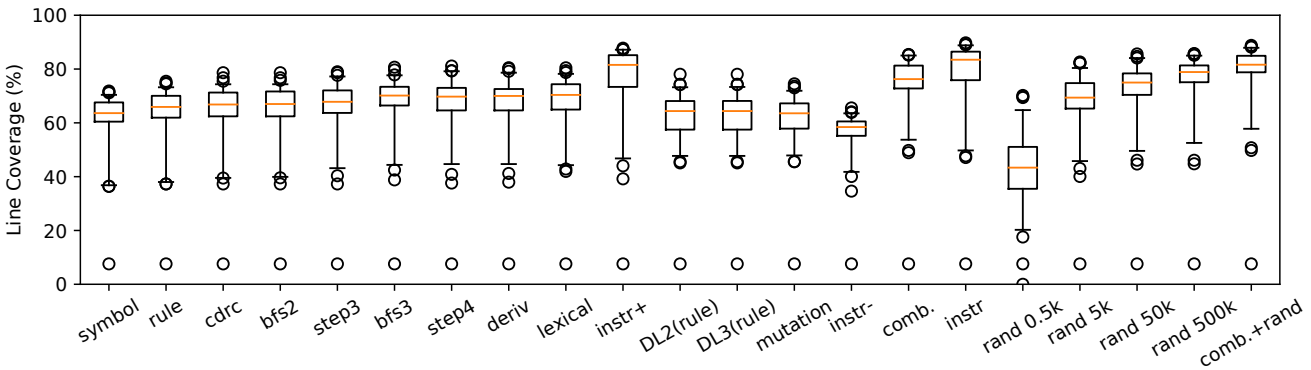

Figure 3. Boxplots for line coverage for syntactic test suites (excluding systematic test suites with random token instances). Each box shows Q3/Q1 interquartile range of the line coverages, i.e. the upper end of the box shows the 75th percentile (where the line coverage is better than the indicated value in 75% of the student compilers) while its lower end corresponds to the 25th percentile. The line across each box indicates the median and the "whiskers" extend from the 95th to the 5th percentile. Outliers are shown as circles; the bottom outlier across all test suites reflects the coverage from the same student's submission.

for $step_4$, indicating that the few longer derivations explored by *deriv* exercise more system behaviors rather than simply a larger part of the system's code base. The latter cases indicate that *lexical* indeed exercises different aspects of the system—specifically, its line coverage of the scanner module is as expected widely above (by approx. 15%-points) that of every other test suite.

Note, however, that line coverage tops out around 68%. This falls short of the 70-80% code coverage range that is considered to be an indication of a good test suite. Stated more clearly: from a software engineering point of view, the systematic syntactic tests are insufficient for testing.

Similarly to the code coverage numbers, the numbers of submissions with detected failures in the front-end (i.e., errors in the implementation of the parser) or with crashes go up, and consequently, the number of submissions passing all tests in the respective suites goes down from 18 for *symbol* to 13 for *deriv* and bfs_3. However, the number of semantic errors exercised remains consistently low across all test suites and is only half of that of $instr^+$. The systematic tests mostly exercise the scope errors (g) and (h) in Fig. 2.

Surprisingly (or perhaps not—these are student implementations after all), up to a third of the submissions fail with an unexpected syntax error for at least one test in the test suites. The test suites from the standard criteria (i.e, *symbol*, *rule* and *cdrc*) over shorter derivations perform much worse than those over longer derivations, not only because the former are smaller, but also because the individual tests are much less likely to reveal any failure: for *symbol*, only 0.9% of the $61 \times 37 = 2257$ runs of all submissions over all tests led to an unexpected syntax error, while 3.1% of the corresponding $61 \times 449 = 27389$ *deriv*-runs did.

In summary, and as answer to RQ1a, we can conclude that larger input space coverage through longer or wider derivations improves line and branch coverage from 60.7% to 67.3% and from 46.3% to 55.5%, respectively, and substantially increases the number of submissions in which syntax errors (from 8 to 20 out of 61 submissions) and crashes (from 39 to 44) are uncovered.

Wider derivations are not better than narrower derivations. Table 1 shows that test suites based on the k-level *bfs* derivations perform better than those based on the corresponding narrower k-step derivations, but paired t-tests show that the difference is not statistically significant in this case. Moreover, the size of the *bfs* test suites follows the expected exponential growth (e.g., bfs_4 already contains more than eight million tests, and we were not able to complete its evaluation), but the larger size does not correspond to a much better coverage of the semantic errors. Similarly, the k-level *bfs* test suites find syntax errors or crashes only in few additional submissions, and pass a couple of fewer students entirely. This indicates that the additional tests in the k-level *bfs* test suites mostly exercise the same behaviors than those already exercised by the corresponding narrow k-step test suites.

The instructor's test suite outperforms systematic positive test suites. The syntactically correct instructor test suite $instr^+$ achieves a substantially and statistically significantly higher line (75.6% vs. 60.7%–67.3%) and branch (67.3% vs. 46.3%–55.6%) coverage than any of the systematically generated positive test suites.

$instr^+$ also covers the error space much better: on average, submissions exercise 3.2 different semantic errors over this suite, achieving 2×–3× the coverage induced by the grammar-based test suites. $instr^+$ detects syntax errors in 50 of the 61 submissions (i.e., at 2.5×–6× the ratio of the automatically generated test suite), with 3.7% of the test runs failing with an unexpected syntax error. Overall, it uncovers errors and crashes in more than 90% of the student submissions, letting only six of them pass.

Summary. In our experiments, the automatically generated positive syntactic test suites perform substantially and statistically significantly worse than the instructor's hand-crafted suite, even when we focus on their ability to uncover syntax errors rather than on the achieved code coverage.

When no hand-crafted test suite is available, grammar-based test suites can be used as a basis for testing and validation tasks. In our case, they achieve 60%–68% line coverage, and uncover syntax errors in 8–20 and crashes in 39–44 of 61 submissions. Test suites based on longer and more complex derivations perform better, but grow quickly and come with a higher test execution cost. The challenge is to define a criterion that reduces these costs but still achieves good system coverage. Our results show that *deriv* achieves the highest code coverage and triggers the highest number of failures and crashes of all grammar-based test suites, with a moderate number of tests. It therefore has good potential as low cost and high coverage criterion, thus answering RQ1b.

4.2 Negative Test Suites

Next, we investigate how the negative test suites (i.e., containing only syntactically invalid tests) perform. For their construction we use token- and rule-mutation algorithms [27]. We apply the Damerau-Levenshtein mutations with n-grams of length $n = 2$ and $n = 3$ to the positive *rule* test suite (denoted by $DL_2(rule)$ and $DL_3(rule)$, respectively). We compare these to the 48 negative tests from the instructor's test suite (denoted as $instr^-$). Note that for the negative tests the Pass column in Table 1 shows the number of submissions that correctly reported a syntax error.

The second block of Table 1 shows that the mutation-based negative test suites achieve higher line, branch, and error coverage than the manually constructed negative test suite of the instructor. $instr^-$ passes more (24) submissions, but also crashes fewer submissions compared to the mutation-based test suites. Note that the number of different errors reported on average is actually *higher* than the number of different syntax error messages given in the specification. This indicates that many students did not follow the instructions and reported errors wrongly.

Despite its smaller size, $instr^-$ contains test cases that are targeted towards certain parts of the compiler (which explains the high number of passing submissions), but overall the test cases have less syntactic variation than the mutation-based tests, which therefore achieve higher coverage and crash more submissions.

4.3 Combined Systematic Test Suites

Next, we merge the tests from *deriv* (for deep syntactic coverage), *lexical* (to further exercise lexical analysis), and *rule-mutation* (for negative test cases) and evaluate these as a single *combined* test suite (third block of Table 1). This combination achieves a statistically significant increase in code coverage over the individual test suites, and finds errors in all submissions, but it still fails to match *instr* on code coverage.

4.4 Random Test Suite Generation

Next, we investigate the performance of random positive test suites. We are interested in the question whether *systematic* or *randomized* grammar-based methods are better.

RQ2: Do purely syntactic test suites systematically constructed by the cover algorithm achieve better system coverage and find more errors than purely syntactic random test suites of similar size?

For this question, we investigate two different methods of constructing random programs. First, we generate the systematic test suites, as before, but use random instances for the structured tokens `id`, `num`, and `string`, rather than their minimal yield instances. This can be seen as a randomized version of skeletal program enumeration [31]. We denote these test suites by $\text{random}_{\text{symbol}}$ to $\text{random}_{\text{deriv}}$ in Table 1. Second, we generate test suites using random derivations, but to allow for a fair comparison with the systematically constructed test suites, we limit their size to a fixed "token budget" (i.e., total length). More specifically, we first generate a pool of 100000 random $\Rightarrow^{11}$-derivations; we do not eliminate duplicates, to boost the number of smaller tests. We then sample four test suites with token budgets $N = 500, 5000, 50000, 500000$ such that the random test suites are roughly the size of *rule*, twice that of $step_4$, bfs_3, and 10 times that of bfs_3, respectively. We use random instances for the structured tokens, which means that the test suites can contain structurally equivalent copies, with different names only. We denote these test suites by random_N in Table 1. Note that the average file sizes are much larger here than in the systematic and instructor's test suites. We repeat both random constructions five times and record the averages in Table 1.

For randomized skeletal program enumerations, code coverages are statistically indistinguishable from those of the systematic versions using fixed minimal token yields. Semantic error coverages actually go down (because the random names fail to trigger "multiple declaration" errors), and the number of submissions with failures remains largely unchanged.

The results of the random derivations are much better. The resulting test suites (excluding random_{500}) achieve a statistically significantly higher code coverage in comparison to test suites of the same size constructed following any of the systematic criteria, cover more semantic errors, and trigger more failures and crashes. In particular, they break through the line coverage barrier of the systematic test suites, and random_{500000} also achieves a semantic error coverage almost matching that of $instr^+$, although it uncovers syntax errors in fewer submissions.

We can therefore state in answer to RQ2 that random grammar-based test suites perform better than systematically constructed *positive* test suites, and achieve code coverage on par with *combined* positive and negative test suites, but trigger more crashes than the combination.

4.5 Combined Systematic/Random Test Suites

Finally, we merge the *combined* syntactic tests with the random tests in random_{500000}. This combination performs consistently well and on par with the instructor's test suite: it achieves about the same line coverage, better branch coverage, covers almost as many errors, and triggers failures and crashes in about the same number of submissions, eventually finding errors in every single submission. Both approaches (i.e., systematic and random test suite generation) complement each other well: the combined code coverage is noticeably higher than that of either individual test suite, the random tests trigger many crashes, while the systematic tests ensure high error coverage and uncover syntax errors in every single submission.

We can therefore give a positive answer to RQ1: we can indeed construct grammar-based test suites that have the same performance as a hand-crafted test suite. Their only shortcoming—which is ultimately a limitation of the purely syntactical approach to test suite generation— is the lack of a *semantic oracle*, which we address in the following section.

5 Testing Contextual Constraints

The results so far show the main limitation of syntactic test suites: while we can achieve adequate code coverage through a combination of systematic and random test suites, the lack of a more precise *semantic oracle* means that we can use these test suites only for syntax and *crash testing*. Any *functional testing* (with concomitant better error detection capabilities) requires tests that either conform to the contextual constraints of the language (principally its scoping and typing rules), or break them in a well-defined way. Our last two questions address this issue:

RQ3: How do we encode contextual constraints into the grammar and integrate them with test suite generation to create semantic test suites?

RQ4: How do we generate tests with specific failure modes?

In order to construct such tests, we need to integrate a semantic oracle *with*—but not necessarily *into*—the test suite construction. In the following, we describe and evaluate such an integration via a simple generate-and-test schema with a "dumb" generator and a lightweight tester. We first randomly generate syntactically valid program skeletons (see Section 4.4), and then test whether we can complete them into programs that satisfy the required properties (or break them in a controlled way) by choosing appropriate values for the identifier tokens; skeletons that fail this property check are discarded.

Our key idea is to use a lightweight extension of CFGs to capture the required contextual constraints, rather than relying on an attribute grammar framework (such as RAGs [12] or Silver [28]) or even a full-fledged DSL such as NaBL [17]. More specifically, we encode scoping and typing information with "semantic mark-up tokens" in the rules, for example

fdef → `id_dcl` **`:`** enter
`takes` *vdecl* (**`;`** *vdecl*)*
`returns` (*type* | **`nothing`**)
body leave
...
factor → `id_ref` (**`[`** *simple* **`]`** | **`(`** *expr* (**`,`** *expr*)* **`)`**)? | ...

Here, mark-up tokens such as enter or leave are typeset in sans serif; note that we use the notation `id_dcl` (resp. `id_ref`) to add the semantic mark-up dcl (resp. ref) to the token `id`.

The mark-up tokens are interpreted by a small generic core engine when the token streams for the generated tests are rendered. Some tokens are "pretty-printed" purely for their side-effect in the core engine and actually rendered as empty strings (e.g., enter creates and enters a new scope in the symbol table but leaves no trace in the actual program) while others have a side-effect but also return a textual representation (e.g., `id_dcl` adds a declaration for a fresh name to the symbol table's current scope and returns the name).

The interpretation of a mark-up token can also fail (e.g., `id_ref` trying to look up an identifier in an empty symbol table). In this case, it is impossible to complete the skeleton into a valid program (under the interpretation of the contextual constraints given by the mark-up tokens), and the skeleton is discarded as a test.

Finally, the integration requires some minimal glue code that implements the contextual constraints using the core engine's API, and a mediator between the grammar and this glue code that maps the mark-up tokens to the corresponding functionality. For example, the mediator mapping

`enter` ⇝ `scope.enter(uni, single, transparent)`

maps the mark-up token `enter` to the API-function `enter`, with arguments capturing AMPL's scoping model; see below for details.

This separation of concerns between the semantic oracle and the test suite construction has several advantages. First, it allows us to reuse the grammar-based testing framework *without any changes*. We can therefore use this to drive exploration of the semantic space, e.g., to generate all skeletons that cover all minimal paths between `id_decl` and `id_ref`. However, we leave such specialized coverage policies for future work, and simply use random program skeletons for our evaluation. Second, we can reuse semantic core functionalities such as symbol tables. Third, we can easily change the underlying contextual constraints by deliberately overwriting some of the mappings in the mediator. We use this to implement concise error models that are guaranteed to produce tests that break specific contextual properties of

```
// Declarations
program → initialize enter
              program id : funcdef* main : body
          leave cleanup
fdef    → id_dcl : enter
              takes vdecl ( ; vdecl)*
              returns ( type | nothing )
          body leave
varseq  → id_dcl ( , id_dcl)* : type
// References
assign  → let id_ref ([ simple ])? = ( expr | array simple)
call    → id_ref ( expr ( , expr)* )
input   → input id_ref ([ simple ])?
factor  → id_ref ([ simple ] | ( expr ( , expr)* ))? | ...
// Mappings
initialize ⇝ scope.initialize()
cleanup    ⇝ scope.cleanup()
enter      ⇝ scope.enter(uni, single, transparent)
leave      ⇝ scope.leave()
id_dcl     ⇝ scope.declare(uni)
id_ref     ⇝ scope.lookup(uni, visible)
```

Figure 4. AMPL rules with scoping tokens and mappings.

the language. For example, by changing the mapping from `id_ref ⇝ scope.lookup(uni, visible)`, which returns an arbitrary identifier that is visible at the current location, to `id_ref ⇝ scope.fresh()`, which generates a fresh identifier, we generate AMPL programs for which compilers must report the error "*the identifier <id> is unknown*".

5.1 Testing Scoping Rules

We now illustrate our approach in more detail and use semantic mark-up tokens to specify AMPL's scoping rules. We also demonstrate the use of error models. Figure 4 shows all rules that require scoping mark-up tokens, split into the rules where `id`'s are declared and where they are referenced, and the mediator mappings.

As indicated in Section 3, AMPL's scoping model is very simple. It has a single `universal` namespace and at most two nested scopes, since functions can only be declared at the top-level and declarations cannot shadow each other. Hence, there are only two rules where we need to mark-up tokens to handle scoping; see Fig. 4. At the *program* rule, we `enter` and `leave` the global scope at start resp. end of the program; the mapping for `enter` calls the `scope.enter` API-function with arguments that specify scopes which only allow a `single` declaration for each name, and have a `transparent` boundary with any nested scopes and do not allow overwriting of any declarations. At the *fdef* rule, we `enter` the local scope after the function name and `leave` it after the function body; this ensures that the function's name is still added (via the mark-up token) to the global scope.

Variables and parameters are both declared at the *varseq* rule, via the `id_dcl` mark-up token. Note that the `enter` and `leave` mark-up tokens ensure that the declarations are entered into the correct scope. Since we are generating test cases, the actual name is not fixed at this point; the mapping for `id_dcl` simply uses the `scope.fresh()` API function to generate an unused random identifier. Note also that program name is not adorned with a mark-up token, because it is not actually entered into the symbol table; in fact, we uncovered this inconsistency in the AMPL specification through our testing. At references, we use the function `scope.lookup(uni, visible)` mapped from the mark-up token `id_ref` to simply pick a random name that is visible in the current scope.

The simplicity of the mediator mappings is actually a virtue. Since AMPL is a little language with a simple scoping model, we can map its definition directly into API-calls. More complicated languages could require more effort.

Error Models. The scoping mark-up tokens described above guarantee that any syntactic skeleton which is successfully completed into a test case is correct with respect to AMPL's scoping model (but not necessarily its typing rules; see Section 5.2). Hence, any student submission that produces any scope error (in Fig. 2, messages (g) or (h)) must contain an error. We can therefore use these tests to identify both types of scoping errors. However, obviously, this does not catch all errors, e.g., it would not catch errors in a compiler not implementing any symbol table at all.

We can improve our testing with tests that are syntactically correct but contain a single, well-defined violation of a contextual rule. We can construct such *negative contextual* test suites easily within our framework, simply by overwriting the mappings for individual mark-up tokens. For example, the following four mappings model different possible errors in the symbol table:

```
id_dcl  ⇝e  scope.lookup(uni, current)
id_dcl  ⇝e  scope.lookup(uni, outer)
id_ref  ⇝e  scope.fresh()
id_ref  ⇝e  scope.lookup(uni, stale)
```

The first error model tests the handling of multiple definitions (Fig. 2(g)), by returning an identifier that is already declared in the current scope rather than declaring a fresh one (as the base mapping does; see Fig. 4). Note, however, that this would fail on the first `id_dcl` mark-up token that it encounters (since the current scope is still empty), and never produce a test case. We therefore use a generic error function to wrap this into a try/catch-construction that falls back to the base mapping if the mapping in the error model fails; we denote this by $\rightsquigarrow_e$.[2] Hence, by using the base mapping for the first `id_dcl` mark-up token and the error model for the second, the skeleton `program id : main: id_dcl , id_dcl : boolean; chillax end` can be successfully completed into the test

[2]The actual implementation also maintains error counts and locations, and provides more flexibility, allowing the specification of additional guards and a separate base mapping.

case **program** a **: main:** x **,** x **: boolean; chillax end** that contains the expected duplicate *variable* declaration for x.

The second error model also tests for duplicate declarations, but it looks for an existing declaration in an `outer` (rather than the `current`) scope. Given AMPL's scoping rules, this produces test cases where a parameter or local variable shadows a function declaration.

Similarly, we can generate test cases with references to undeclared variables (Fig. 2(h)), either by returning a `fresh` and therefore guaranteed undeclared name, or by looking up a `stale` name that has gone out of scope.

We emphasize that our goal here is only to demonstrate how an adequate symbol table implementation can be integrated into grammar-based test suite generation to enforce and to break the scoping rules for a little language, not to introduce a new general scoping modeling language.

5.2 Testing Typing Rules

We now extend our approach from handling scopes to handling types. Our overall approach remains the same: we generate a program skeleton that contains semantic mark-up tokens, and then interpret it from left to right, choosing names that ensure that the syntactically correct skeleton also obeys the contextual constraints.

However, the structure of the interpretation changes, and we now execute a type checker over the program. We use additional mark-up tokens or *type annotations* ::type to express in the rules that a grammar symbol, or more precisely its yield, must have the given type under the type information stored in the symbol table. Types are either AMPL's basic scalar types (rendered as bool and int), nothing, or generic scalar and array types, with any representing any type and something representing any type but nothing. A compatibility relation specified as part of the type system allows us to match the types, for example bool against any or something.

We need to refactor the rules slightly if the syntactic structure can interfere with the contextual constraints. For example, we need to split the original "untyped" rule *term* → *factor* (*mulop factor*)* into three different variants with the right type annotations:

$$\begin{array}{lll} \textit{term}{::}\mathsf{bool} & \rightarrow \textit{factor}{::}\mathsf{bool}\ (\textbf{and}\ \textit{factor}{::}\mathsf{bool})^+ \\ \textit{term}{::}\mathsf{int} & \rightarrow \textit{factor}{::}\mathsf{int}\ ((\ \textbf{/}\ \mid\ \textbf{*}\ )\ \textit{factor}{::}\mathsf{int})^+ \\ \textit{term} & \rightarrow \textit{factor} \end{array}$$

Note that this destroys the LL(1)-property of the grammar, but this does not matter since we are generating programs, not parsing them. The first variant says that we can generate a *term* of type bool if we generate a list of at least two *factor*s also of type bool that are separated by **and**-tokens. The last variant remains untyped because it is generic: the type of the *term* is the same as the type of the *factor*.

The core engine maintains the type annotations on a stack so that the top of the stack tracks the currently expected type.[3] The annotated rules as shown above are therefore pre-processed so that the right mark-up tokens are produced in the right order: a type annotation ::type on a rule's head is translated into a mark-up token pop(type) that is inserted before the rule body, while the same annotation in a rule's body is translated into a mark-up token push(type) that is inserted before the annotated symbol. As an example, from the annotated rules

$$\begin{array}{lll} \textit{loop} & \rightarrow \textbf{while}\ \textit{expr}{::}\mathsf{bool}\ \textbf{:}\ \textit{stmts}\ \textbf{end} \\ \textit{factor}{::}\mathsf{bool} & \rightarrow \textbf{not}\ \textit{factor}{::}\mathsf{bool} \end{array}$$

and the corresponding generic rules for *expr*, *simple*, *term*, and *factor* (see Fig. 5 in Appendix A.1 for the full AMPL grammar with the type annotations and mark-up tokens) we produce the fragment

while push(bool) pop(bool) **not** push(bool) id_ref(var) **:**
chillax end

When the core engine interprets a push(type) mark-up token, it simply pushes this on the stack. When it interprets a pop(type) mark-up token, it checks that the expected type matches the top of stack; if the types match (under the specified compatibility relation), the top is popped off the stack, otherwise the skeleton cannot be made type-correct and is discarded. Hence, the push(bool) and pop(bool) after the **while** cancel out, as expected, and the second push(bool) leaves a bool on top of the stack when id_ref(var) mark-up token is interpreted.

At this point, the interpretation of the scoping and the typing mark-up tokens intersect: id_ref(var) is mapped to a `lookup`-call for a currently `visible` variable whose declared type matches the type on the top of the stack; if this succeeds, the type is "consumed" and popped off the stack. This can be formulated concisely in the mediator mapping

id_ref(var)⇝scope.lookup(uni, visible, var, type.pop())

However, this requires the corresponding type information for an identifier to be added to the symbol table. This happens through the rule

varseq → mark id_dcl(var) (**,** id_dcl(var))* **:** *type* upd_vars

Here, mark first pushes a marker on the stack; each id_dcl(var) then uses `scope.declare` to create and declare a variable with an incomplete symbol table entry (i.e., without associated type information), and pushes a var marker on type stack. upd_vars pops off the correct type left there by the *type*-rule and unwinds the stack until it hits the marker. On the way it updates the type information in all incomplete symbol table entries it encounters, and replaces the var markers by the type, to leave on the stack the right information for upd_fun.

[3] In fact, the symbol table and type stack APIs are working on the same underlying stack; this makes the integration between scoping and typing easier, e.g., in the *varseq*- or *fdef*-rules, see below.

Table 2. Results of semantic tests. The interpretations of the columns Size and Failures remain as in Table 1, but Failures now includes semantic errors, due to the stronger semantic oracle. Coverage lists the different error messages (Fig. 2) triggered by tests in the corresponding test suite. The different error models are explained in Appendix A.

	Size		Coverage			Failures		
Error Model	M	length	line (%)	branch (%)	errors	front	table	crash
None	10780	40.2	72.0	61.0	*(x)	18	59	45
Redeclaration in scope	9903	41.8	64.3	52.3	(g)	15	43	41
Undeclared identifier	432	42.5	52.8	40.5	(h)	7	45	43
Reference out of scope	668	56.0	51.5	38.9	(h)	6	55	43
Variable instead of function	54	69.0	51.9	38.4	(i),(j)	13	45	39
Function instead of variable	917	65.6	68.3	55.3	(k)-(m),(p)-(t),(w),(x)	20	61	51
Scalar in array assign	88	59.6	64.2	53.2	(l)	9	50	39
Not an array	98	61.4	63.8	48.4	(l),(r)	9	53	36
Array instead of scalar	254	64.9	61.2	47.1	(m),(p),(q),(s),(t)	12	61	43
Array instead of scalar (generic)	901	48.6	65.5	52.0	(n)	10	53	44
Boolean instead of integer	194	62.9	68.6	55.9	(p),(q),(u),(v)	12	49	41
Mismatched types in assignment	26	72.8	57.5	43.7	(r)	7	56	36
Integer instead of boolean	339	61.2	67.4	54.6	(s)-(u),(w)	13	50	41
Mismatch types in function parameters	13	74.9	49.1	36.1	(x)	5	47	33
Too few arguments	68	69.3	57.0	43.5	(y)	16	49	38
Too many arguments	12	49.4	42.5	30.5	(z)	9	50	35
Combined	24659	43.6	75.0	66.1	(g)-(n),(p)-(z)	22	59	51

In general, the type stack allows us to store and flexibly manipulate the contextual dependencies while we traverse the skeleton in single left-to-right pass, e.g., to handle the argument alignment between function declarations and function and procedure calls:

$$\mathit{factor} \longrightarrow \texttt{id\_ref(fun)}\ \texttt{(}\ \mathit{expr}\ (\ \texttt{,}\ \mathit{expr})^*\ \texttt{)}\ \texttt{unmark}$$

Here, `id_ref(fun)` looks up a visible function that matches the target type on top of the stack, and pushes the marker and then the argument types (in reverse order) on the stack. Each *expr* then finds its own corresponding target type on top of the stack; if it encounters the marker, the skeleton has too few actual arguments to match the formal parameters, and is discarded. Likewise, if unmark does *not* see the marker, the call has too many arguments for it to be valid and the skeleton is discarded as well.

All successfully completed skeletons satisfy the contextual constraints—now both scoping and typing.

Error Models. As with the scoping model, we can now easily inject a wide variety of errors into the generated tests, simply by overwriting the mappings for individual mark-up tokens. For example, we can trigger type errors at control flow predicates or boolean operators by pushing an int rather than the required bool on the type stack, using a simple mapping push(bool) $\leadsto_e$ push(int). Similarly, we can use arrays where scalars are required (push(scalar) $\leadsto_e$ push(array)), or vice versa, or test that functions are not called as procedures (push(nothing) $\leadsto_e$ push(something)). Note that these error models are in some sense cross-cutting concerns, since for example push(bool) $\leadsto_e$ push(int) can apply at any syntactic position that requires a bool value. In order to target errors associated with specific syntactic positions, e.g., "incompatible types for not" (see Fig. 2(w)), we would need to refine the mark-up tokens, or perhaps even use a different style of error models where we overwrite entire grammar rules; however, we leave this for future work.

Overall, we formulated in about four hours 15 different error models for scoping and typing errors. Most of them are equally straightforward as the examples above, since we only changed the arguments of the push mark-up tokens to a different, incompatible value. This did not require any deeper insights into the language specification, since AMPL's type system is simple. However, injecting errors to trigger the "incompatible types for parameters" and "too few / many arguments" messages Fig. 2(x), (y), and (z) required some more involved stack manipulations. An informal description of all error models, including examples of corresponding tests is given in Appendix A.3.

The entire implementation, including the generic symbol table, the AMPL-specific glue code, and all mediators (both for correct code and for error models), comprises less than 300 lines of code, not counting comments and empty lines.

Experimental Results. We derived a test suite with type-correct programs, as well as one for each error model, from 100 000 skeletons randomly generated at depth 11 using rule probabilities to bias shorter declaration and argument lists, and then using the encoding of the contextual constraints resp. the error models to complete them into programs. This took about 90 seconds per test suite on a standard laptop, roughly evenly split between the random skeleton construction and the completion; we observed only little variation between the different error models.

Table 2 shows the characteristics of the generated test suites. The number of skeletons that could be typed according the semantics of AMPL varies between the error models,

but we do not consider this to be a limiting factor as test suite generation is inexpensive. While the code coverage of the semantically correct tests (i.e., no error model) is slightly lower than the code coverage of the best syntactic test suites (see Table 1), the code coverage of the test suites under the different error models drops substantially. This is hardly surprising, since they are substantially smaller. However, what is more important is that with our stronger semantic oracle, the test suites find incorrect implementations of the contextual constraints in all of the student compilers. Additionally, we find three bugs in the instructor implementation (these are shown in Appendix A.2), including an erroneous call to error function (x) when parsing a type correct program.

6 Threats to Validity

Our observations are based on a single experiment. However, our cohort size is large enough to allow statistically valid conclusions. We believe that the experimental set-up reflects a common teaching situation, and that our observations can be generalized to similar educational contexts. We also believe that they could also be generalized to compilers for domain-specific languages since they are often of a similar size and complexity as AMPL. However, all subject programs are implemented as one-pass compilers, so we do not face the usual "phase bottleneck" where code in one phase is only exercised if the test complies with all constraints checked in prior phases. Overall coverage values may thus be substantially lower for modern multi-pass compilers, although coverage over the syntactic and semantic analyses phases should remain similar. The subject programs all used given scaffolding code and architectural skeletons, which could systematically skew results in either direction. Finally, note that the subject programs were implemented by students, which may make it easy to find errors in these systems. This may not be the case in production compilers but code coverage numbers should generalize, all other things remaining equal.

Another threat to validity is the lack of comparison to existing tools in our discussion of Section 4. However, we did fuzz the instructor's compiler using AFL [30], both using its default blind mode, and with a dictionary of tokens constructed from the instructor's test suites. AFL plateaus at 73.6% line coverage, and provides no stronger oracle than our syntactic test suites.

We mitigated against the usual internal validity threats of human error, human bias and human performance by automating experiments and using well-established tools for code coverage and statistical evaluation.

7 Related Work

Grammar-Based Test Suite Construction. Purdom's seminal paper [26] proposed an algorithm that systematically constructs a minimal number of sentences but ensures that all grammar rules are applied. However, we do not use Purdom's algorithm in our evaluation, because of the complexity of the individual test cases. Context-dependent rule coverage [18] is considered the standard grammar coverage criterion. We use other coverage criteria such as *symbol*, *rule*, their extensions to k-step and k-level simultaneous derivations and a fix-point derivable pair coverage (Definition 2.1).

Grammar-based test generation has focused almost exclusively on generation of positive test suites until recent work that proposed two mutation-based algorithms [27] that guarantee programs with single well-defined errors. We also use extensions to token mutation that look at n-grams instead of poisoned pairs.

Random Program Generation. Random program generation for testing purposes goes back at least to Hanford [10]'s work in 1970. More recent random generation approaches [3, 13, 15, 19, 22, 23, 25] typically use a large number of control parameters (e.g., rule probabilities, traversal orders, symbol and rule counts, length, depth, and balance restrictions, rule guards, and many others) to ensure that the derivation process terminates, and that the generated programs have certain characteristics. Similar techniques are employed by grammar-based fuzzers such as langfuzz [14].

QuickCheck-style [5], property-based testing techniques and tools have also been used to generate programs, either directly from using types as generators [24], from grammars or from other kinds of rules [1, 20]. One of the main challenges has been to generate programs/terms that are not completely random but satisfy certain (semantic) preconditions, e.g., are type or semantically correct. For example, Fetscher et al. [8] present a generic method for randomly generating well-typed expressions starting from a specification of a typing judgment in PLT Redex and using a specialized solver that employs randomness to find many different valid derivations of the judgment form. They then use these random terms to falsify semantic model conjectures.

In the area of random testing of compilers, of special note is Csmith [29], a well-engineered, highly effective tool for generating C programs that avoid undefined or unspecified behavior by construction.

More recent works propose methods specifically aiming to find test programs which result in difficult-to-find miscompilations [21], or that generate test programs which are more likely to be bug-revealing and diverse [4].

Generation of Semantically (In)Valid Programs. Kifetew et al. [16] use stochastic context-free grammars with semantic annotations on grammar rules similar to our type annotations to ensure that the generated sentences respect the semantic constraints of the language. They combine the stochastic grammar with genetic programming (using code coverage as the fitness function) in order to generate test suites. However, no investigation is made into the generation of semantically incorrect programs.

Dewey et al. [6] depart from using stochastic context-free grammars as generators and instead express the grammar as predicates in a constraint logic programming system. They then use that method to test the type checker of the Rust programming language [7].

8 Summary and Lessons Learned

Little languages still need large test suites for proper testing. Here we reported on our experience in using grammar-based test suite construction methods to test simple compilers written by a cohort of 61 undergraduate students.

We first focused on *purely syntactic* test suites where we only know whether a test is syntactically valid or not. We compared different systematic and random test suite construction methods against each other and against the course instructor's hand-crafted evaluation and marking suite. The main lessons we learned from this experiment are: (i) Systematic test suites achieve a code coverage below 68%, which is typically considered to be insufficient for testing. They also all fall far behind the instructor's test suite in the number of detected failures and crashes, even in the student's implementation of the syntax, and cover fewer semantic errors. (ii) Random test suites of the same size outperform systematic test suites for all but the smallest sizes. Larger random test suites push code coverage up to 75%, and trigger more crashes but fewer non-crashing errors than the instructor's test suite. (iii) A combination of different positive and negative systematic and random test suites performs in all aspects as well as or better than the instructor's test suite, and triggers failures and crashes in every single student submission.

We then addressed the lack of a more precise *semantic oracle* that limit syntactic test suites to syntax and *crash testing*. We developed a light-weight extension of the basic grammar-based testing framework where we encode scoping and typing information as "semantic mark-up tokens" in the grammar rules. These mark-up tokens are interpreted by a small generic core engine when the tests are rendered, and tests that cannot be completed into a valid program by choosing appropriate identifiers are discarded. This enables the generation of tests that either conform to the contextual constraints of the language, or break them in a well-defined way, and allows us to use the test suites for *functional testing*. The main lessons we learned here are: (i) The separation of concerns between the semantic oracle and the test suite simplified the both implementation of the core engine and the formulation of target-specific glue code. (ii) Most mark-up is straightforward to add: we only needed to refactor few rules (mostly in the expression syntax) to separate out different mark-ups for shared syntax. (iii) We can concisely formulate very targeted error models to test different aspects of the contextual constraints. The effort for this is also low: we wrote in about four hours 15 different error models for scoping and typing errors that trigger almost all semantic errors specified in the AMPL documentation. This approach allowed us to find errors even in the instructor's reference implementation.

We are currently investigating how we can extend our approach to more complex scoping and typing models used in modern programming languages, e.g., Rust. We are also interested in more advanced methods to develop error models, e.g., by deriving them automatically from the scope specifications. Here, the use of a higher-level name-binding DSL such as NaBL may be attractive.

We currently do not check for the correct name resolution: if there are multiple declarations of a name in multiple enclosing outer scopes, how do we test that the implementation picks the right one? This may require the generation of tests with run-time oracles.

Acknowledgments

This work is funded in part by the NRF under Grant 113364 and by a collaboration grant from SASUF (the South Africa – Sweden Univerity Forum). The support of the DSI-NRF Centre of Excellence in Mathematical and Statistical Sciences (CoE-MaSS) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the CoE. The third author acknowledges the support of his research from grant #621-2017-04812 by the Swedish Research Council, and by the Swedish Foundation for Strategic Research through the aSSIsT project.

References

[1] Bernhard K. Aichernig and Richard Schumi. 2019. Property-based testing of web services by deriving properties from business-rule models. *Software and Systems Modeling* 18, 2 (2019), 889–911. https://doi.org/10.1007/s10270-017-0647-0

[2] Jon Louis Bentley. 1986. Little Languages. *Commun. ACM* 29, 8 (1986), 711–721. https://doi.org/10.1145/6424.315691

[3] David L. Bird and Carlos Urias Munoz. 1983. Automatic Generation of Random Self-Checking Test Cases. *IBM Syst. J.* 22, 3 (1983), 229–245. https://doi.org/10.1147/sj.223.0229

[4] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-Guided Configuration Diversification for Compiler Test-Program Generation. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. IEEE, 305–316. https://doi.org/10.1109/ASE.2019.00037

[5] Koen Claessen and John Hughes. 2000. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/1988042.1988046

[6] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language fuzzing using constraint logic programming. In *ACM/IEEE International Conference on Automated Software Engineering (ASE 2014)*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, New York, NY, USA, 725–730. https://doi.org/10.1145/2642937.2642963

[7] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, Myra B. Cohen, Lars

Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 482–493. https://doi.org/10.1109/ASE.2015.65

[8] Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Proceedings (LNCS, Vol. 9032)*, Jan Vitek (Ed.). Springer, 383–405. https://doi.org/10.1007/978-3-662-46669-8_16

[9] Bernd Fischer, Ralf Lämmel, and Vadim Zaytsev. 2011. Comparison of Context-Free Grammars Based on Parsing Generated Test Data. In *Software Language Engineering - 4th International Conference, (SLE 2011), Revised Selected Papers (LNCS, Vol. 6940)*, Anthony M. Sloane and Uwe Aßmann (Eds.). Springer, 324–343. https://doi.org/10.1007/978-3-642-28830-2_18

[10] Kenneth V. Hanford. 1970. Automatic Generation of Test Cases. *IBM Syst. J.* 9, 4 (1970), 242–257. https://doi.org/10.1147/sj.94.0242

[11] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. IEEE, 189–199. https://doi.org/10.1109/ASE.2019.00027

[12] Görel Hedin. 2000. Reference Attributed Grammars. *Informatica (Slovenia)* 24, 3 (2000).

[13] Daniel Hoffman, David Ly-Gagnon, Paul A. Strooper, and Hong-Yi Wang. 2011. Grammar-based test generation with YouGen. *Softw. Pract. Exp.* 41, 4 (2011), 427–447. https://doi.org/10.1002/spe.1017

[14] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium*. 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[15] William Homer and Richard Schooler. 1989. Independent Testing of Compiler Phases Using a Test Case Generator. *Softw. Pract. Exp.* 19, 1 (1989), 53–62. https://doi.org/10.1002/spe.4380190106

[16] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. 2017. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. *Empirical Software Engineering* 22, 2 (2017), 928–961. https://doi.org/10.1007/s10664-015-9422-4

[17] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. 2012. Declarative Name Binding and Scope Rules. In *Software Language Engineering, 5th International Conference, SLE 2012, Revised Selected Papers (LNCS, Vol. 7745)*, Krzysztof Czarnecki and Görel Hedin (Eds.). Springer, 311–331. https://doi.org/10.1007/978-3-642-36089-3_18

[18] Ralf Lämmel. 2001. Grammar Testing. In *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Proceedings (LNCS, Vol. 2029)*, Heinrich Hußmann (Ed.). Springer, 201–216. https://doi.org/10.1007/3-540-45314-8_15

[19] Ralf Lämmel and Wolfram Schulte. 2006. Controllable Combinatorial Coverage in Grammar-Based Testing. In *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference (TestCom 2006), Proceedings (LNCS, Vol. 3964)*, M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko (Eds.). Springer, 19–38. https://doi.org/10.1007/11754008_2

[20] Leonidas Lampropoulos and Konstantinos Sagonas. 2012. Automatic WSDL-guided Test Case Generation for PropEr Testing of Web Services. In *8th International Workshop on Automated Specification and Verification of Web Systems*. 3–16. https://doi.org/10.4204/EPTCS.98.3

[21] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

[22] Peter M. Maurer. 1990. Generating Test Data with Enhanced Context-Free Grammars. *IEEE Softw.* 7, 4 (1990), 50–55. https://doi.org/10.1109/52.56422

[23] Peter M. Maurer. 1992. The Design and Implementation of a Grammar-based Data Generator. *Softw. Pract. Exp.* 22, 3 (1992), 223–244. https://doi.org/10.1002/spe.4380220303

[24] Manolis Papadakis and Konstantinos Sagonas. 2011. A PropEr Integration of Types and Function Specifications with Property-based Testing. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/2034654.2034663

[25] A. J. Payne. 1978. A Formalised Technique for Expressing Compiler Exercisers. *SIGPLAN Not.* 13, 1 (Jan. 1978), 59–69. https://doi.org/10.1145/953428.953435

[26] Paul Purdom. 1972. A Sentence Generator for Testing Parsers. *BIT Numerical Mathematics* 12 (Sept. 1972), 366–375. Issue 3. https://doi.org/10.1007/BF01932308

[27] Moeketsi Raselimo, Jan Taljaard, and Bernd Fischer. 2019. Breaking parsers: mutation-based generation of programs with guaranteed syntax errors. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2019)*, Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira (Eds.). ACM, 83–87. https://doi.org/10.1145/3357766.3359542

[28] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Sci. Comput. Program.* 75, 1-2 (2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

[29] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, Mary W. Hall and David A. Padua (Eds.). ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[30] Michał Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/

[31] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, Albert Cohen and Martin T. Vechev (Eds.). ACM, New York, NY, USA, 347–361. https://doi.org/10.1145/3062341.3062379

A Appendix

A.1 Typed AMPL Grammar

Figure 5 shows the full AMPL grammar with the type annotation mark-up tokens.

A.2 Bugs in Instructor Compiler

Bug #1: Unexpected Failure. The following type correct program produces an unexpected error message in the instructor compiler. The test was generated under the error model *None*.

```
program a:
n: takes e, o4 : boolean ;
         j, z : integer array
   returns nothing
  b50, gs81, k : integer array;
  x5w1 : boolean;
  d, q44, a : boolean array;
  pop;
  do n(o4 and x5w1 or x5w1, o4, gs81, b50)
end
main : chillax end
=> incompatible types (integer array and boolean)
   for parameter 3 of 'n'
```

program → initialize enter **program** id **:** *funcdef** **main :** *body* leave cleanup
fdef → id_dcl(fun) **:** enter **takes** *vdecl* (**;** *vdecl*)* **returns** (*type* | **nothing**::nothing) upd_fun *body* leave
type → **boolean**::bool | (**boolean array**)::bool_array | **integer**::int | (**integer array**)::int_array
varseq → mark id_dcl(var) (**,** id_dcl(var))* **:** *type* upd_vars
body → *vardecl*? clear *stmts* **end**
vardecl → *varseq* **;** (*varseq* **;**)*
stmts → **chillax** | *stmt* (**;** *stmt*)*
stmt → *assign* | *call* | *return* | *input* | *output* | *case* | *loop*
idexp → id_ref(var) | make_array id_ref(var) **[** *simple*::int **]**
assign → **let** ϵ::any dup idexp **=** *expr* | **let** id_ref(var)::array **=** **array** *simple*::int
call → **do** ϵ::nothing id_ref(fun) **(** *expr* (**,** *expr*)* **)** unmark
input → **input** idexp::scalar
output → **output** *elem* (**.** *elem*)*
elem → string | *expr*::scalar
return → **pop** in_proc | **pop** in_func *expr*
case → **when case** *expr*::bool **:** *stmts* **end** (**case** *expr*::bool **:** *stmts* **end**)* (**otherwise :** *stmts* **end**)?
loop → **while** *expr*::bool **:** *stmts* **end**
expr::bool → *simple*::int (**=** | **>=** | **>** | **<=** | **<** | **/=**) *simple*::int
expr → *simple*
simple::bool → *term*::bool ((**-** | **+**) *term*::bool)$^+$
simple::int → *term*::int (**or** *term*::int)$^+$
simple → *term*
term::bool → *factor*::bool (**and** *factor*::bool)$^+$
term::int → *factor*::int ((**/** | ***** | **rem**) *factor*::int)$^+$
term → *factor*
factor::bool → **not** *factor*::bool | **true** | **false**
factor::int → num
factor → id_ref(fun) **(** *expr* (**,** *expr*)* **)** unmark | **(** *expr* **)** | idexp

Figure 5. AMPL grammar with type annotations and mark-up tokens.

The procedure n is declared as taking two booleans, and two integer arrays as arguments, and returning nothing. The recursive do calls n with arguments of the correct types. However, the instructor implementation reports that the third argument (gs81) is not of the correct type and calls an error function.

Bug #2: Unexpected Failure. The following type correct program produces an undocumented error message in the instructor compiler. The test was generated under the error model *None*.

```
program a :
ua: takes f : boolean array
    returns nothing
  pop ua(f)
end
main :
  b : integer array ;
  pop
end
=> illegal procedure pop expression
```

The procedure ua is declared, taking one boolean array as an argument and returning nothing. The body of the procedure is a single pop statement called the result of calling ua recursively. The instructor compiler detects the attempt to pop a value from a procedure, and calls an undocumented error function.

Bug #3: Unexpected Pass. The following type incorrect program (generated under the error model *Function instead of variable*) is erroneously accepted by the instructor compiler.

```
program a:
o: takes c, x, s : boolean
   returns integer array
  y, br81a, b7, rn6aa : integer array;
  p : integer;
  qf, v : integer array;
  let br81a = o
end
main : chillax end
=> Generated a.class
```

The function o is declared, taking three boolean values as arguments, and returning an integer array. Some local variables are then declared, including br81a, an integer array. Under this error model, we expect to see an occurrence of a reference to a function where a reference to a variable was expected. This is the case in the assignment, `let br81a = o`, where a function is being assigned to an integer array.

A.3 Error Models

Array instead of scalar. This error model guarantees the use a reference to an array where a scalar value is expected, producing either an incompatible types or illegal array operation error messages.

```
program a:
main:
  p, n, b : integer array ;
  input n[(p)]
end
=> incompatible types (integer array and integer)
   for array index of 'n'
```

Array instead of scalar (generic). This error model guarantees the use of an array as an argument where a scalar was expected, producing only scalar expected error messages.

```
program a:
main:
  x: integer array;
  input x
end
=> scalar expected instead of 'x'
```

Scalar in array assignment. This error model guarantees the assignment of the result of an array initialisation to a scalar variable.

```
program a:
main:
  v : integer ;
  let  v = array v
end
=> 'v' is not an array
```

Boolean instead of integer. This error model guarantees that the use of a boolean variable where an integer is expected. The example shows the attempted initialization of an array using a boolean size argument.

```
program a:
main:
  s : boolean array ;
  input s[false]
end
=> incompatible types (boolean and integer)
   for array index of 's'
```

Function instead of variable. This error model guarantees the occurrence of a function or procedure where a variable was expected. The example shows an illegal assignment of a function to an integer array.

```
program a:
ji: takes s : boolean
     returns nothing
  input ji
end
main :
  pop
end
=> 'ji' is not a variable
```

Integer instead of boolean. This error model guarantees the occurrence of an integer where a boolean was expected. The example shows an illegal usage of an integer in a boolean expression.

```
program a:
main:
  output not  0
end
=> incompatible types (integer and boolean) for 'not'
```

Mismatched types in assignment. This error model guarantees the occurrence of an assignment where the types are mismatched. The example shows the illegal assignment of a boolean to an integer variable.

```
program a:
main:
  m : boolean ;
  ek2, k : integer ;
  z : integer ;
  let m = 2 rem k / z
end
=> incompatible types (integer and boolean)
   for assignment to 'm'
```

Mismatched types in function parameters. This error model guarantees the occurrence of a call to a function or procedure with the correct number of arguments, but where the types are mismatched. The example shows the illegal assignment of a boolean to an integer variable.

```
program a:
ts: takes nc : boolean
    returns nothing
  w : boolean array ;
  while nc or nc: do
    ts(w)
  end;
  do ts(nc)
end
main: chillax end
=> incompatible types (boolean array and boolean)
   for parameter 1 of 'ts'
```

Not an array. This error model guarantees the use of a non-array variable where an array is expected, such as indexing and array initialisation.

```
 program a:
 main:
   t, i, e : integer;
   input e[0]
 end
=> 'e' is not an array
```

Redeclaration in scope. This error model guarantees the declaration of name in a scope where that name is already visible.

```
 program a:
 main:
   e, e : boolean;
   pop
 end
=> multiple definition of 'e'
```

Reference out of scope. This error model guarantees a reference to a name that is not visible in the current scope.

```
 program a:
 k: takes m : integer
    returns nothing
   chillax
 end
 main :
   input m
 end
=> the identifier 'm' is unknown
```

Too few arguments. This error model guarantees a call to a function or procedure with fewer arguments than is required.

```
 program a:
 g: takes g2t, z3 : integer
     returns nothing
   do g(z3)
 end
 main : chillax end
=> too few arguments for call of 'g'
```

Too many arguments. This error model guarantees a call to a function or procedure with more arguments than is required.

```
 program a :
 x5s1a: takes i : boolean array
        returns nothing
   do x5s1a(i, i)
 end
 main: chillax end
=> too many arguments for call of 'x5s1a'
```

Undeclared identifier. This error model guarantees a reference to a name that has not been declared in any scope.

```
 program a:
 main:
   o : boolean ;
   input  g
 end
=> the identifier 'g' is unknown
```

Variable instead of function. This error model guarantees the use of a variable where a function is expected.

```
 program a:
 s: takes u : boolean
    returns nothing
   do u(u)
 end
 main : chillax end
=> 'u' is not a procedure
```