

Syntax-Based Testing

We are going to completely switch gears now. We will see two applications of context-free grammars:

1. input-space grammars: create inputs (both valid and invalid)
2. program-based grammars: modify programs (mutation testing)

Mutation testing. The basic idea behind mutation testing is to improve your test suites by creating modified programs (*mutants*) which force your test suites to include test cases which verify certain specific behaviours of the original programs by killing the mutants. We'll talk about this in more detail in a week or two.

Generating Inputs: Regular Expressions and Grammars

Consider the following Perl regular expression for Visa numbers:

$$\sim^4[0-9]\{12\}(\?:[0-9]\{3\})?\$$$

Idea: generate “valid” tests from regexps and invalid tests by mutating the grammar/regexp. (Why did I put valid in quotes? What is the fundamental limitation of regexps?)

Instead, we can use grammars to generate inputs (including sequences of input events).

Typical grammar fragment:

```
mult_exp  =  unary_exp | mult_exp STAR unary_arith_exp | mult_exp DIV unary_arith_exp;
unary_exp =  quant_exp | unary_exp DOT INT | unary_exp up;
          ⋮
start    =  header?declaration*
```

Using Grammars. Two ways you can use input grammars for software testing and maintenance:

- recognizer: can include them in a program to validate inputs;
- generator: can create program inputs for testing.

Generators start with the start production and replace nonterminals with their right-hand sides to get (eventually) strings belonging to the input languages. Typically you would generate inputs until you have enough, either randomly sampling from the input space or systematically generating inputs of larger and larger sizes.

Another Grammar.

```
roll      =  action*
action    =  dep | deb
dep       =  "deposit" account amount
deb       =  "debit" account amount
account   =  digit { 3 }
amount    =  "$" digit+ "." digit { 2 }
digit     =  ["0" - "9"]
```

Examples of valid strings.

Note: creating a grammar for a system that doesn't have one, but should, is a useful QA exercise. Using this grammar at runtime to validate inputs can improve software reliability, although it makes tests generated from the grammar less useful.

Some Grammar Mutation Operators.

- Nonterminal Replacement; e.g.
 $\text{dep} = \text{"deposit" account amount} \implies \text{dep} = \text{"deposit" amount amount}$

(Use your judgement to replace nonterminals with similar nonterminals.)

- Terminal Replacement; e.g.
 $\text{amount} = \text{"\$" digit⁺ "." digit { 2 }} \implies \text{amount} = \text{"\$" digit⁺ "$" digit { 2 }}$

- Terminal and Nonterminal Deletion; e.g.
`dep = "deposit" account amount \implies dep = "deposit" amount`
- Terminal and Nonterminal Duplication; e.g.
`dep = "deposit" account amount \implies dep = "deposit" account account amount`

Using grammar mutation operators.

1. mutate grammar, generate (invalid) inputs; or,
2. use correct grammar, but mis-derive a rule once—gives “closer” inputs (since you only miss once.)

Why test invalid inputs? Bill Sempf (@sempf), on Twitter: “QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.”

If you’re lucky, your program accepts strings (or events) described by a regexp or a grammar. But you might not use a parser or regexp engine. Generating using the regexp or grammar helps detect deviations, both right now and in the future.

What we’ll do is to mutate the grammars and generate test strings from the mutated grammars.

Some notes:

- Can generate strings still in the grammar even after mutation.
- Recall that we aren’t talking about semantic checks.
- Some programs accept only a subset of a specified larger language, e.g. Blogger HTML comments. Then testing intersection is useful.