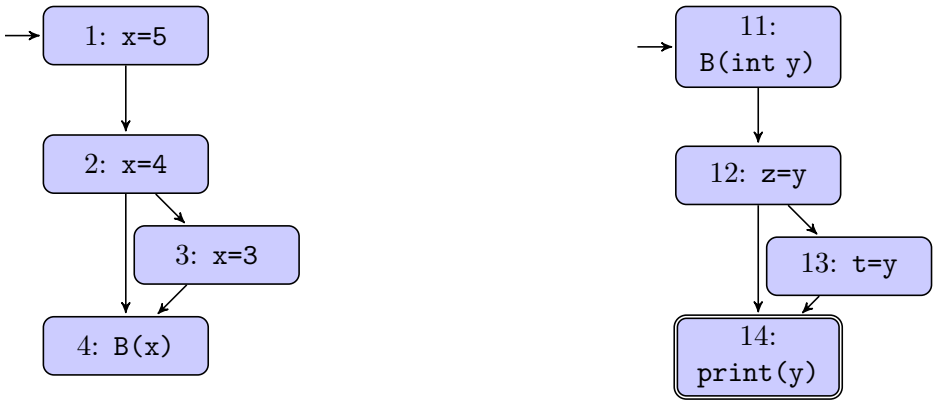
Software Testing, Quality Assurance and Maintenance Winter 2015

Lecture 17 — February 11, 2015

Patrick Lam *version 1*

We want to define du-pairs between callers and callees. Here's an example.



We'll say that the *last-defs* are 2, 3; the *first-use* is 12. We formally define these notions:

Definition 1 (Last-def). *The set of nodes N that define a variable* `x` *for which there is a def-clear path from a node $n \in N$ through a call to a use in the other unit.*

Definition 2 (First-use). *The set of nodes N that have uses of* `y` *and for which there exists a path that is def-clear and use-clear from the entry point (if the use is in the callee) or the callsite (if the use is in the caller) to the nodes N.*

We need the following side definition, analogous to that of def-clear:

Definition 3 *A path $p = [n_1, \ldots, n_j]$ is* use-clear *with respect to v if for every $n_k \in p$, where $k \neq 1$ and $k \neq j$, then v is not in use(n_k).*

In other words, the last-def is the definition that goes through the call or return, and the first-use picks up that definition.

Here are two more examples:

```
x = 14;   // last-def
y = g(x);
print(y); // first-use

int g(a) {
  print(a); // first-use
  b = 24;   // last-def
  return b;
}
```

```
main() {
  x = 1;
  x = 2; // last-def
  if (...) { x = 3; /* last-def */ }
  B(x);
}
B(int y) {
  if (...) {
    Z = y; // first-use
  } else {
    T = y; // first-use
  }
  print(y);
}
```

One can create pairs of triples linking last-defs and first-uses; characterize a last-def or a first-use by the method name, the variable name, and the statement, and then link them.

Tests can, of course, go beyond just testing the first-use. Our first-use and last-def definitions, however, make testing slightly more tractable. We could, of course, carry out full inter-procedural data-flow, i.e. covering all du-pairs, but this would be more expensive.

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.

Generating Inputs: Regular Expressions and Grammars

Consider the following Perl regular expression for Visa numbers:

```
^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:

$$
\begin{array}{rcl}
\text{mult\_exp} & = & \text{unary\_exp} \mid \text{mult\_exp } \texttt{STAR} \text{ unary\_arith\_exp} \mid \text{mult\_exp } \texttt{DIV} \text{ unary\_arith\_exp;} \\
\text{unary\_exp} & = & \text{quant\_exp} \mid \text{unary\_exp } \texttt{DOT INT} \mid \text{unary\_exp up;} \\
 & \vdots & \\
\text{start} & = & \text{header}^{?}\text{declaration}^{*}
\end{array}
$$

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.

We specify three coverage criteria for inputs with respect to a grammar G.

Criterion 1 Terminal Symbol Coverage *(TSC). TR contains each terminal of grammar G.*

Criterion 2 Production Coverage *(PDC). TR contains each production of grammar G.*

Criterion 3 Derivation Coverage *(DC). TR contains every possible string derivable from G.*

PDC subsumes TSC. DC often generates infinite test sets, and even if you limit to fixed-length strings, you still have huge numbers of inputs.

Another Grammar.

$$
\begin{array}{rcl}
\text{roll} & = & \text{action}^{*} \\
\text{action} & = & \text{dep} \mid \text{deb} \\
\text{dep} & = & \texttt{"deposit"} \text{ account amount} \\
\text{deb} & = & \texttt{"debit"} \text{ account amount} \\
\text{account} & = & \text{digit } \{ \ 3 \ \} \\
\text{amount} & = & \texttt{"\$"} \text{ digit}^{+} \texttt{"."} \text{ digit } \{ \ 2 \ \} \\
\text{digit} & = & [\text{"0"} - \text{"9"}]
\end{array}
$$

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.
  dep = "deposit" account amount $\Longrightarrow$ dep = "deposit" amount amount

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

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

- Terminal and Nonterminal Deletion; e.g.
  dep = "deposit" account amount $\Longrightarrow$ dep = "deposit" amount

- Terminal and Nonterminal Duplication; e.g.
  dep = "deposit" account amount $\Longrightarrow$ 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.

As you saw in Assignment 1, it's the easiest thing to overlook invalid inputs. Yet they may lead to undefined behaviour.

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

Some notes:

- Book claims we don't have much experience using grammar-based operators.
- 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.