CITS5501 Software Testing and Quality Assurance
Semester 1, 2020

Syntax-based testing

Command-line programs often take a range of arguments and options.

For instance, we might have a command `delete_file`, that takes a file to delete, and a *flag* (boolean option) stating whether the removal is *recursive* (i.e., if the file is actually a directory, then remove it and all contents).

Documentation will typically show the way the command can be called as follows:

```
delete_file [--recursive] FILE_NAME
```

The brackets mean something is optional. A pipe ("`|`") is used to separate alternatives. Words in capital letters usually mean a string representing, e.g., a file, directory, URL, etc.

If we are testing the program, we can write invocations of the program as following a *grammar*.

Exercise 1

How can we describe the grammar for the `delete_file` program invocations, using the BNF syntax we've seen?

Assume we have a terminal symbol ***`file_name`*** we don't have to define.

Here are several possible solutions.

solution 1:
Define a non-terminal symbol `<recursive_option>`, which is either the exact string `--recursive`, or the empty string.

```
<delete_file_invocation> ::= "delete_file"  <recursive_option> <file_name>

<recursive_option> ::= "--recursive" | ""
```

solution 2

Don't use any non-terminals besides `<delete_file_invocation>`; instead just define two different "branches" in `<delete_file_invocation>` (i.e. two distinct *productions*), one with the string `--recursive`, and one without.

```
<delete_file_invocation> ::= ("delete_file" "--recursive" <file_name>)
                           | ("delete_file"  <file_name>)
```

not technically a solution

Technically, the following is not a BNF specification of a grammar:

```
<delete_file_grammer> :: = "delete_file" ("--recursive" | "") FILE_NAME
```

Why not? Because it uses parentheses for grouping – and our definition of what can go in BNF specifications in the lecture notes did *not* include parentheses – only terminal symbols, non-terminal symbols, and the bits of syntax `::=` and `|`.

However, parentheses (and a few other bits of syntax, useful for things like being able to represent repetition easily) turn out to be very convenient in practice, and form part of what's called *Extended* Backus-Naur Form (Wikipedia article here).

So to go strictly by what we said in the lectures, you should probably stick to plain BNF.

Exercise 2

(a) Could we write an exhaustive set of tests for this syntax? (b) What sort of coverage would that give us?

(b) What is a set of tests that would give us production coverage?

a. Yes, we easily could. Many BNF-specified grammars (like the one for “numbers” in the lecture slides) represent an extremely large or infinite set of strings, so can’t be tested exhaustively.

But here – *given* that we make the assumption that `<file_name>` is a special sort of terminal – then as far as the theory of syntax-based testing goes, yes, we can write an exhaustive set of tests.

b. It would give us:

- Terminal Symbol Coverage (TSC), because we’ll have used all terminal symbols;
- Production Coverage (PDC), because we’ll have tried all alternative productions; and
- Derivation Coverage (DC), because we’ll have tried all the strings the grammar produces.

c. We aren’t told exactly what happens if `delete_file` is called on a directory without the `--recursive` option being specified, so let’s make the following assumption:

Assumption: If `delete_file` is called on a directory, and the `--recursive` option has not been specified, an error message will be displayed.

We should also pin down what all the inputs are to a test, when specifying it, so we’ll also assume that the string `test_file` constitutes our special terminal, and that it is the name of a directory containing two files.

- Test 1
  Description: Test `delete_file` with the `--recursive` option
  Test inputs:
  - A directory `test_file` in the working directory containing two files
  - The parameters “`--recursive file_name`” passed to the program `delete_file`
  Expected output: `file_name` plus the files within it are deleted
- Test 2
  Description: Test `delete_file` without the `--recursive` option
  Test inputs:
  - A directory `test_file` in the working directory containing two files
  - The parameters “`file_name`” passed to the program `delete_file`
  Expected output: An error message is displayed

Questions asked in the workshop

Q. Do we need to break down "`file_name`" further when answering the questions?
A. No, we've explicitly assumed that `file_name` is a special sort of terminal symbol – and in a BNF specification, a terminal symbol *can't* be broken down any further. (That's why it's called "terminal".)

Recall that whenever we make a model of a system – a state diagram, a control-flow graph, or something else – we are necessarily simplifying (ignoring some features of the real system). And we are allowed to decide what balance to strike between making the model *simple* (but less accurate) and *realistic* (and thus, more complicated, and usually more difficult to understand and test). For instance, when constructing a control-flow graph, we might choose to ignore exceptions being thrown.

In this case: it's useful for our purposes to make the simplifying assumption that `file_name` is a terminal. But there are other assumptions we could make, and other ways we could model the `delete_file` program.

Further reading

Parsing command-line options for a program is a very common task. Even when working on a web-based project, often programmers will make use of their own custom-written programs and scripts which need to parse command-line options.

Some sample libraries that do command-line option-parsing are:

- `argparse` (for Python)
- `args4j` (for Java)
- the `OptionParser` class (for Ruby)
- `optparse-applicative` (for Haskell)

An advantage of using these libraries is that, in addition to *parsing* options given by a user on the command line, they can also *generate* help documentation – if the program is called improperly, it can print out a message about correct usage:

```
$ delete_file --recursive

Missing: FILE_NAME

Usage: delete_file [--recursive] FILE_NAME
  Delete a file or directory (possibly recursively)

Available options:
  --recursive              Remove recursively
  FILE_NAME                file to delete
```