CITS5501 Software Testing and Quality Assurance
Semester 1, 2020
Week 9 Workshop – Grammars and syntax-based testing

Reading

It is strongly suggested you complete the recommended readings for weeks 1-8 *before* attempting this lab/workshop.

Grammars

Take a look at the documentation page for Docker which explains the command-line arguments that can be given to the Docker executable:
https://docs.docker.com/engine/reference/commandline/cli/

The page shows a typical way of documenting a command-line program. The syntax used is a little like BNF, except that non-terminals are usually written in ALL CAPS, square brackets ("`[ ]`") are used to indicate optional elements, and an ellipsis ("`...`") means some element can be repeated.

The following grammar represents a simplified, very small subset of these command-line arguments:

```
<invocation> ::= "docker " ( "--verbose " )? <subcommand> <image>
<subcommand> ::= "pull " | "run " | "build "
<image>      ::= "debian" | "ubuntu" | "fedora"
```

The question mark indicates an optional element.

In your browser, visit the **BNF Playground** webpage at https://bnfplayground.pauliankline.com, and type the grammar into the box labelled "Enter your BNF (or EBNF) below".

Click the button labelled "Compile BNF". This checks that the grammar follows the rules for BNF, analyses it, and converts it into data structures which can be used to build *recognizers* (programs which check a string, and see if it belongs to the language defined by the grammar) and *generators* (programs which produce random strings belonging to the language).

In the box labelled "Test a string here!", type:

```
docker pull debian
```

You'll see that the box is initially red (indicating the string is *not* recognized as being in the language we have defined), and then turns green (once it *is* recognized).

Exercise:

- Try the string "docker pull alpine", and note that it is not recognized.
- Amend the grammar so it *is* recognized.
  (You might want to clear the "Test a string here!" box first – otherwise, when you make changes to the grammar, it may show up as invalid and display errors. Once you've amended the grammar, you need to re-compile.)

Amend the last line of the grammar so it reads:

```
<image>      ::= "debian" | "ubuntu" | "fedora" | "alpine"
```

Generators

Now try hitting the button labelled "Generate random `<invocation>`" several times, and see what strings are produced.

Note that the BNF differs slightly from the less formal version we have seen in class, in that it requires spaces be explicitly inserted.

Exercise:

- Remove the spaces, and try generating random strings – are they what you would expect?

Now put the original grammar back in and compile it again.

Exercise:

- When selecting which of several alternatives to use, the generator chooses one randomly. How would you alter the grammar so that the image "`ubuntu`" is chosen twice as often as the others?

Alter the last line of the original grammar, which was:

```
<image>      ::= "debian" | "ubuntu" | "fedora"
```

Currently, "`debian`" and "`fedora`" occur once each; if we want "`ubuntu`" to be selected twice as often (i.e. two-thirds of the time), we should replace "`ubuntu`" with:

```
"ubuntu" | "ubuntu" | "ubuntu" | "ubuntu"
```

to get

```
<image>      ::= "debian" | "ubuntu" | "ubuntu" | "ubuntu" | "ubuntu" |
    ↪  "fedora"
```

Testing grammars

Recall that when testing something that can be represented as a grammar, there are different levels of *coverage* we might aim to achieve.

Exercise:

a. Looking at the original grammar – could we test this grammar *exhaustively*? Why or why not? If so, how many tests would be required?
b. Can all grammars be tested exhaustively? Why or why not?

a. Yes, we could. It defines a *finite* language – that is, a language containing a finite set of strings.

   How many tests would be required? One for each string in the language.

   There are 3 options for "image", and 3 options for "subcommand", and two options for "–verbose" (either present, or not.)

   So the total number of tests is $3 \times 3 \times 2$ or 18.

b. No, they cannot. Some grammars define languages with an infinite number of strings.

   For instance

```
<list> ::= "END" | "0" <list>
```

   defines an infinitely large language, consisting of any number (zero or more) of instances of the string "0", then the string "END".

Exercise:

- Looking at the original grammar – how many tests would we need to write if we wanted to get the following sorts of coverage for this grammar?

  a. terminal coverage
  b. production coverage

a. The number of terminals is the number of "raw" strings in the grammar. They are:

   - `"docker "`
   - `"--verbose "`
   - `"pull "`
   - `"run "`
   - `"build "`
   - `"debian"`
   - `"ubuntu"`
   - `"fedora"`

   So, eight tests.

b. A *production*, as we define it in class, is one of the *alternatives* within a rule.

   (An optional element counts as two alternatives – it's either present or not.)

   We can just count the number of alternatives in each rule:

   ```
   <invocation> ::= "docker " ( "--verbose " )? <subcommand> <image>
   ```

   - two alternatives (verbose is present or not)

   ```
   <subcommand> ::= "pull " | "run " | "build "
   ```

   - three alternatives, "pull" or "run" or "build"

   ```
   <image>      ::= "debian" | "ubuntu" | "fedora"
   ```

   - three alternatives, "debian" or "ubuntu" or "fedora"

   So there are $2 + 3 + 3$ productions, and thus 8 tests required.

Exercise:

- A *recognizer* for a language can be regarded as a program which takes in a string, and gives back a boolean saying whether the string is in the language or not.
- Suppose you wanted to write a test ensuring the "build" production can be recognized. What would a test case look like which does this?

We might define the test case as follows:

- Test value: the string “docker build debian”
- Invoking the recognizer: run the recognizer on the test value.
- Expected result: The recognizer should return “true”.

Exercise:

- Consider the following grammar:

  ```
  <list> ::= "1" | "0" <list>
  ```

  Try entering it into the BNF playground, generating some random strings, and seeing what strings it recognizes.

  a. Give an example of a string containing just zeroes and ones which is *not* in the language defined by the grammar.
  b. Can the grammar be tested exhaustively? Explain why or why not.

a. The string “`10`” is not in the language – all strings in the language *end* with 1.
b. It cannot – it represents all strings consisting of one or more zeroes, then a 1. It is infinite in size.