

CITS5501 Software Testing and Quality Assurance

Syntax-based testing

Unit coordinator: Arran Stewart

Grammars, syntax and language

Developers use grammars and syntax all the time, though they may not realize it.

Whenever we see a requirement like “a date should be in the format YYYY-MM-DD”, we’re making use of a grammar (though only very informally expressed).

Analysing a date

If some requirement says that a parameter to a function, or an item in a database, should be “in the format YYYY-MM-DD”, what it (usually) means, but more explicitly stated is:

- Any of the Y’s, M’s or D’s can be replaced by a digit in the range 0–9 – if you provide a date that can’t be generated in such a fashion, we might say you’ve provided a *syntactically ill-formed* date.
- There are other rules about validity (e.g. if the first “M” is replaced by a 1, then the second “M” can only be in the range 0–2), but we usually don’t consider those to be *syntax* errors.
 - Dates which violate these rules are usually said to violate the *semantics* of dates, or *semantic constraints*

Grammars

Grammars just give us a way of formally specifying what things are and are not syntactically correct.

Every grammar defines what is called a *language* (though not always a very interesting one) – a set of acceptable strings.

A grammar for the date might look like this:

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |  
           "7" | "8" | "9"
```

```
<date>  ::= <digit> <digit> <digit> <digit> "-"  
           <digit> <digit>  "-"  
           <digit> <digit>
```

Grammars

The following grammar is *equivalent* to the previous one – in that they define the exact same set of strings – but provides a few hints as to the *semantics* of bits of the string (and is probably a bit easier to read).

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |  
           "7" | "8" | "9"
```

```
<year>  ::= <digit> <digit> <digit> <digit>
```

```
<month> ::= <digit> <digit>
```

```
<day>   ::= <digit> <digit>
```

```
<date>  ::= <year> "-" <month> "-" <day>
```

Notation

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |  
           "7" | "8" | "9"  
<year>  ::= <digit> <digit> <digit> <digit>  
<month> ::= <digit> <digit>  
<day>   ::= <digit> <digit>  
<date>  ::= <year> "-" <month> "-" <day>
```

The notation is a simplified form of what is called **BNF** (Backus-Naur Form).

The following symbols are used in this notation:

We read “`::=`” as “is defined as” or “can be expanded to”, and “`|`” as “or”.

So the first line says, “A ‘digit’ is defined as being either the string”0“, or the string”1“, or ...”

(These symbols are sometimes called “meta-syntactic symbols”, meaning symbols used to define a syntax.)

Notation

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |  
           "7" | "8" | "9"  
<year>   ::= <digit> <digit> <digit> <digit>  
<month>  ::= <digit> <digit>  
<day>    ::= <digit> <digit>  
<date>   ::= <year> "-" <month> "-" <day>
```

The things in strings are called *terminal symbols* – they are the equivalent of “words” in our language.

They are like atoms, in that they are the smallest, indivisible parts of our language.

In our case, the terminals are all strings containing a single digit.

Notation

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |  
           "7" | "8" | "9"  
<year>  ::= <digit> <digit> <digit> <digit>  
<month> ::= <digit> <digit>  
<day>   ::= <digit> <digit>  
<date>  ::= <year> "-" <month> "-" <day>
```

The things between angle brackets are called *non-terminal* symbols.

The above grammar contains five *rules* (also called “productions”, in the textbook).

In the sorts of grammar we will consider,¹ every rule is of the form:

non-terminal “::=” sequence of terminals and non-terminals

¹Called *context-free grammars* or CNFs (see https://en.wikipedia.org/wiki/Context-free_grammar). There are other formalisms more and less expressive than CNFs.

Notation

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |  
           "7" | "8" | "9"  
<year>  ::= <digit> <digit> <digit> <digit>  
<month> ::= <digit> <digit>  
<day>   ::= <digit> <digit>  
<date>  ::= <year> "-" <month> "-" <day>
```

To be precise – at its simplest, the right-hand side (RHS) of a rule will be a sequence of terminals and non-terminals.

But we can also insert:

- bars to indicate “or” (alternatives)
- an asterisk (called the “Kleene star”) to indicate “zero or more of the preceding thing”
- a plus sign to indicate “one or more of the preceding thing”
- a range of numbers (e.g. “3–4”) to indicate a number of possible instances of the preceding thing.

And we can use parentheses to group things.

Notation – asterisk

An example: the following is a fairly typical way of defining valid *identifiers* in many programming languages:

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |  
           "7" | "8" | "9"  
<letter> ::= "a" | "b" | ...  
<underscore> ::= "_"  
<identifier> ::= ( <letter> | <underscore> )  
                  ( <letter> | <underscore> | <digit> )*
```

This means, “An identifier always starts with a letter or underscore; it is followed by any number (possibly zero) of characters drawn from the set of letters, digits and the underscore character”.

Use in development

Much of the software we rely on makes use of grammars (though not always explicitly).

Whenever we *validate* entries into web forms or databases, we are often defining a syntax to do so. (One common way is to use what are called *regexes* – we will discuss them more later.)

Use in development

Command-line programs often take arguments – sometimes adhering to very complex rules, as we saw in the first lecture:

[illegible]

Use in development

For very simple programs, we might analyse the arguments “by hand”.

For complex programs – we typically use a *command-line argument parser* to work out whether a user has supplied a valid set of arguments (and what we should do with them).

Use in development

Grammars are used to define whether something is a valid

- email
- HTML page
- email address

and many other formats.

Use in development

Often, it will be useful to define what are called “domain-specific languages” (DSLs) which describe entities in a domain and things to do with them – e.g. Makefiles are an example of this.

Syntaxes are typically used to define such languages.

Use in development

And of course, every programming language is defined by a grammar or syntax – when we violate the syntax, the compiler tells us we've committed a “syntax error”.

Syntactically well-formed Java class:

```
class MyClass { }
```

Syntactically ill-formed:

```
class { MyClass }
```


Questions

- Can we describe binary formats, as well as text?
 - Yes, though BNF is not especially suited to describing binary formats.
 - BNF works well for things in textual format (including the source code of programming language files, HTML documents, JSON documents, and so on).
 - For data in binary format (for instance, TCP packets or JPEG files), a commonly-used formalism is [ASN.1](#) (“Abstract Syntax Notation One”).
 - We won’t be examining ASN.1 in detail, but similar considerations apply.

Using the Syntax to Generate Tests

- Syntactic descriptions can be obtained from many sources:
 - program source code
 - design documents
 - input descriptions (e.g. file formats, network message formats, etc)
- Tests are created with two general goals
 - Cover the syntax in some way
 - Violate the syntax (invalid tests)

Using the Syntax to Generate Tests

- Should we apply the techniques we see in this lecture to *every* example of syntactic validation / use of grammars?
- Usually not – we will usually focus on areas of high risk (e.g. that are easy to get wrong, or have bad impacts when we get them wrong).
- Parsing command-line arguments is sufficiently important that we should probably test it.

An example of syntax-generated tests

- *Mutation-based fuzzers* use a body of inputs, and generate new ones (some valid, some invalid) by repeatedly mutating existing inputs
- Often the fuzzers aim to *crash* the program (get it to exit unexpectedly, and/or, in the case of memory-unsafe languages like C and C++, violate memory integrity).
- e.g. We could start with a set of valid PNG files, and use a mutation-based fuzzer to produce many variants of these
- Often we'll want to be sure that our software handles any sort of input *gracefully* – regardless of whether the input is valid or invalid, the program should give some sort of “proper” result (even if that is just an error message). It *shouldn't* (usually) go into an erroneous state.

Example – arithmetic expressions

Another example – we'll define a language to represent simple arithmetic expressions.

Some strings will be *valid* in our language (like “ $(3 + 2) - 5$ ”) and some will not (like “ $3++-()$ ”).

Our *terminal symbols* will consist of the numerals 0-9, and the symbols “ $+ - ()$ ”.

Example – arithmetic expressions

As before, we define a digit:

$$\langle \text{digit} \rangle ::= \text{"0"} \mid \text{"1"} \mid \text{"2"} \mid \text{"3"} \mid \text{"4"} \mid \text{"5"} \mid \text{"6"} \mid \text{"7"} \mid \text{"8"} \mid \text{"9"}$$

Example – arithmetic expressions

And we can say, “An *expression* is either a digit, or, a smaller expression plus some other smaller expression.”

`<expression> ::= <digit> | <expression> "+" <expression>`

Example – arithmetic expressions

Our whole grammar:

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |  
           "7" | "8" | "9"
```

```
<expression> ::= <digit>  
                | <expression> "+" <expression>  
                | <expression> "-" <expression>  
                | "(" <expression> ")"
```


More on BNF grammars

When we specify a grammar, there will normally be a *start symbol*, representing the “top level” of whatever construct we’re specifying.

- e.g. for some programming language:

```
<program_file> ::= <import_statements><declarations><definitions>
```

- Each possible rewriting (i.e., each alternative) of a non-terminal is often called a *production*.

Use of grammars

- Grammars can be used to build *recognizers* (programs which decide whether a string is in the grammar – i.e., parsers) and also *generators*, which derive strings of symbols.

Coverage criteria

- If we're developing tests based on syntax ...
- The most straightforward coverage criterion:
use every terminal and every production rule at least once

Terminal Symbol Coverage (TSC) Test requirements contain each terminal symbol t in the grammar G .

Production Coverage (PDC) Test requirements contain each production p in the grammar G .

Coverage criteria (cont'd)

- Production coverage subsumes terminal symbol coverage; if we've used every production, we've also used every terminal.

Coverage criteria – an impractical one

- We could aim to cover all possible strings

Derivation Coverage (DC) Test requirements contain every possible string that can be derived from the grammar G .

- But except in special cases, this will be impractical

Bounds on coverage

- Example grammar:

`<integer> ::= <digit>|<integer><digit>`

`<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |
"7" | "8" | "9"`

- The number of tests to get TS coverage is bounded by the number of terminal symbols (ten, here)
- To get production coverage, that depends on the number of productions (here: 2 for the first rule, 10 for the second – so, 12)
- Whereas the number of strings that can be generated – needed for derivation coverage – is actually infinite.
 - (likewise for, say, the set of all possible Java programs)
- Even for finite grammars (e.g. some file formats), DC will usually require an infeasibly large number of tests

Data structures

- Typically, for any format we specify syntactically (like JPEG, GIF etc.), we'll have an accompanying data structure that mirrors that the structure of the syntax, in order to manipulate in-memory objects representing that format.
- E.g. see the [JpegImageData](#) class from the Apache Commons Imaging library for Java, or the [png_struct_def](#) for the libpng C library.

Data structures

- But even for data structures which *don't* represent something necessarily stored in binary or textual format, we can consider them as having a syntax-like structure.
- For instance, what is a linked list? In Java-like syntax:

```
class node<V> {  
    V value;  
    node<V> next;  
};
```

It is *either*:

(1) a null pointer, or (2) a value prepended to a list.

Data structures

- Note that this seems quite similar in structure to our grammar for integers.
- Let's consider linked lists of booleans
- Suppose we write the null pointer, an empty list, as "`[]`", and nodes containing boolean values as "`T`" and "`F`". and represent prepending as a colon, ":"
- Then we can actually write a BNF grammar for linked lists.

“Linked list” grammar

`<bool> ::= "T" | "F"`

`<list> ::= "[]" | <bool>":"<list>`

“Linked list” grammar

`<bool> ::= "T" | "F"`

`<list> ::= "[" | <bool>":"<list>`

- So `[]` is a list, as is `T:[]`, and `F:T:[]`.
- `[:] : []` is not a valid list, nor is `[:] : T`.

Data structures

- What about a linked list, where the value type V is, say, another class, `Person`:

```
class Person {  
    PersonID  personID;  
    bool  isStaff;  
    int  age;  
}
```

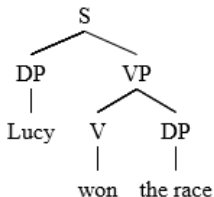
- Which in turn refers to the `PersonID` class.

Data structures

- What grammars and data structures have in common is that they both have terminals (in data structures, these are atomic or “primitive” values that we cannot, or choose not to, break down any further), and they define aggregate structures in terms of simpler structures, in a potentially recursive way.
- So we can use much the same principles to see if our tests of them have good coverage, or to generate them randomly, etc.

Trees

- We can draw a *tree* structure for an expression adhering to some particular syntax called a *parse tree*.²



(Here, “S” stands for “sentence”, “VP” for “verb phrase”, “V” for “verb”, “DP” for “determiner phrase” – basically something that picks out a particular entity.)

²Image from https://commons.wikimedia.org/wiki/File:Precedent_example_1_decl_sent.png

Trees

- The parse tree shows what productions should be followed to parse (or alternatively, to generate) a particular string.
- In practice, the trees formed by *data structures* (as opposed to grammars) are of a slightly different sort – they are *abstract syntax trees* rather than parse trees – but we will not be too concerned with the details.

Generators

- Suppose we had the grammar:

`<Sentence> ::= <NounPhrase><Predicate>`

`<NounPhrase> ::= "Alice" | "Bob" | "the hacker"`

`<Predicate> ::= <Verb><NounPhrase>`

`<Verb> ::= "hires" | "defeats"`

- Then we can see that “Alice hires Bob” and “Bob defeats the hacker” are valid strings in the language this grammar defines (modulo some whitespace).
- And we can see how we could easily generate random valid sentences that conform to these rules.
- Being able to generate things that follow a syntax-like structure is extremely useful for testing.

Generators – network traffic

- We can use it to create traffic generators, for instance – we could generate random valid **TCP traffic** with which to test a router.
- TCP packets follow a syntax-like structure, so it's fairly straightforward to generate them randomly.
A TCP packet consists of: 2 bytes representing a source port (0 through 65535), 2 bytes representing a destination port, then 4 bytes representing a “sequence number”, then ... (see the TCP specification for detailed rules).
- Not all the validity rules for a TCP packet can be expressed in a syntactical way – for instance, it contains a checksum towards the end, which is calculated based on previous information – but quite a bit can.
- This is very handy for “stress” or “load” or “performance” testing – generating large amounts of data, and seeing how our system performs under the load.

Generators – http traffic

- HTTP requests for web pages also follow a syntax, so we could easily generate random HTTP traffic (for instance, to stress-test a web-server, and see how it performs under high load).
- The full syntax for HTTP requests is larger than this,³ but the start of a simplified version of it would look something like:

```
<request> ::= <GETrequest> | <POSTrequest>
<GETrequest> ::= "GET" <space> <URI> <space> <HTTPversion>
               <lineend> <getheaders> <getbody>
...

```

(i.e., HTTP requests are either GET or POST requests, and GET requests start with the keyword GET then a space, then a URI, and so on...)

³See IETF RFC 2616,

Generators – http traffic

- The vast majority of randomly generated HTTP requests would not be for valid URIs, and would result in 404 errors.
- If we wanted to generate, not just random HTTP requests, but requests that actually hit part of a website, we can add in additional constraints to ensure that happens.
- (E.g. We might start by only generating URLs that begin with `https://myblog.github.io/`, if we were testing a blog site hosted on GitHub.)

Generators

- Likewise, HTML and XML documents, JSON, and many other formats all follow syntactical rules, so we can randomly generate them.
- Likewise for custom formats we may come up with.
 - e.g. If we were writing a word processor, we might want to be able generate very large random documents in our word-processor format, to see how our program holds up.

Generators

- For common formats, there are often already data generators with many capabilities:
 - Tools for constructing and generating network traffic: [Ostinato](#), [Scapy Traffic Generator](#), [flowgrind](#), [jtg](#) ... see this [list](#) for many more.
 - HTTP request generators: see for example [httpperf](#)
 - Random bitmap generators: see for example [random.org](#)
- If not, it is perfectly possible to write our own.

Generators and data structures

Things to note when generating data structures:

- In languages with pointers or references, it may be possible to have data structures that contain *cycles*, meaning they are no longer trees but graphs.
- For instance, we could have two linked list nodes A and B, and make A's next reference point to B, and B's point to A. (A cyclic linked list.)
- It's still possible to generate random data of that sort, but doing so takes us beyond our current scope.

More complex rules for validity

- There may be rules for validity of a format (like the existence of checksums) that can't be captured by a grammar.
- This is frequently the case, actually. BNF lets us describe what are known as “context-free” grammars, and a specification for a format may include requirements that are impossible or inconvenient to specify using BNF.
 - e.g. In a valid Java program, variables have to be declared before they are used; it's an error to assign a string literal to an int; and many other rules.
- We may be able to use simple calculations to generate or verify those.
(e.g. to verify or generate a checksum)
- Or we may have to apply more complex rules – these are outside the scope of this unit.

Using generators for testing

- Generating random, valid values is useful for performance testing, as just described – but it is also useful for *property-based testing*, which we will see more of later.
- What is property-based testing? It's a sort of (usually randomized) testing which checks that invariants about functions hold.

Property-based testing

- Consider the following method specification:
`List.remove(Object o)`: Search the list for elements which are equal to object `o` (using `.equals()`). If there are any, then the first such element is removed. Otherwise, the method does nothing.
- If L_{before} is the length of the list before we execute `remove()`, and L_{after} is the length of the list after we execute it, then the following invariant holds:
$$(L_{after} = L_{before}) \vee (L_{after} = L_{before} - 1)$$

Let's call this invariant Inv_1 , for short.
- It is certainly good practice to write tests for `remove()` based on Input Space Partitioning – e.g. constructing small lists that do or don't contain the element being searched for, and constructing test inputs based on that.

Property-based testing

- But if we can identify invariants like Inv_1 , that we think will *always* hold, then we can generate random data to improve our confidence that this is so.
- If our test framework generates a few thousand sample lists, and our invariant holds for all of them, we can be fairly confident that this theory about our method is true.
(We cannot be *certain* – we might have failed to generate a test case that exercises some particular fault – perhaps our method fails on extremely long lists, and we never generated those – but our confidence is definitely improved.)

Property-based testing

Testing frameworks that perform property-based testing include:

- [Hypothesis](#), for Python
- [QuickTheories](#), for Java
- [jsverify](#), for JavaScript
- [QuickCheck](#), the inspiration for most of the others, for Haskell
- ... Many more [listed](#) by David R. MacIver, the developer of Hypothesis.
- We will look at some of these testing frameworks in more detail.

Applications of syntax-based testing

- *Mutation-based fuzzers* use a body of inputs, and generate new ones (some valid, some invalid) by repeatedly mutating existing inputs
- e.g. We could start with a set of valid PNG files, and use a mutation-based fuzzer to produce many variants of these
- Often we'll want to be sure that our software handles any sort of input *gracefully* – accepting it if valid, but detecting the situation when input is invalid

Mutation testing

- Grammars describe both valid and invalid strings
- A *mutant* is a variation of a valid string
 - Mutants may be valid or invalid strings
- Mutation is based on “mutation operators” and “ground strings”

What is mutation ?

We are performing mutation analysis whenever we

- use well defined rules (i.e. operators)
- defined on syntactic descriptions (i.e. grammars)
- to make systematic changes
- to the syntax or to objects developed from the syntax
 - the objects are “ground strings”

Mutation testing – definitions

- Ground string: A string in the grammar
 - (The term “ground” basically means “not having any variables” – in this context, not having any non-terminals)
- Mutation operator: A rule that specifies syntactic variations of strings generated from a grammar
- Mutant: The result of one application of a mutation operator
 - A mutant is a string

Killing Mutants

- When ground strings are mutated to create valid strings, the hope is to exhibit different behavior from the ground string
- Killing Mutants : Given a mutant m for a derivation D and a test t , t is said to “kill” m iff the output of t on D is different from the output of t on m

Syntax-based coverage criteria – mutant coverage

- We can define a coverage criterion in terms of killing mutants:

Mutation Coverage (MC) For each mutant m , the test requirements contains exactly one requirement, to kill m .

- Coverage in mutation equates to number of mutants killed
- The amount of mutants killed is called the mutation score

Coverage criteria – creating invalid strings

- When creating invalid strings, two simple criteria –
- It makes sense to either use every operator once or every production once

Mutation Production Coverage (MPC) For each mutation operator, TR contains several requirements, to create one mutated string m that includes every production that can be mutated by that operator.

Mutation Operator Coverage (MOC) For each mutation operator, TR contains exactly one requirement, to create a mutated string m that is derived using the mutation operator.

Mutation example

A grammar:

```
Stream ::= action*  
action  ::= actG | actB  
actG    ::= "G" s n  
actB    ::= "B" t n  
s       ::= digit{1-3}  
t       ::= digit{1-3}  
n       ::= digit{2}   "." digit{2}   "." digit{2}  
digit   ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |  
           "7" | "8" | "9"
```

- Uses “*”, the “Kleene star”, to represent “zero or more”
- Uses braces to represent “n to m occurrences” or “n occurrences”

Mutation example (cont'd)

- A ground string:

G 23 08.01.90

B 19 06.27.94

Mutation example (cont'd)

- Some mutation operators:
 - Exchange actG with actB
 - replace digits with any other possible digit

Mutation example (cont'd)

- Using mutation operator coverage (MOC):

G 23 08.01.90

B 19 06.27.94

mutated to:

B 23 08.01.90

B 15 06.27.94

Mutation example (cont'd)

- Using mutation operator coverage (MOC):
 - `B 22 08.01.90 G 19 06.27.94`
 - `G 13 08.01.90 B 11 06.27.94`
 - `G 3 3 08.01.90 B 12 06.27.94`