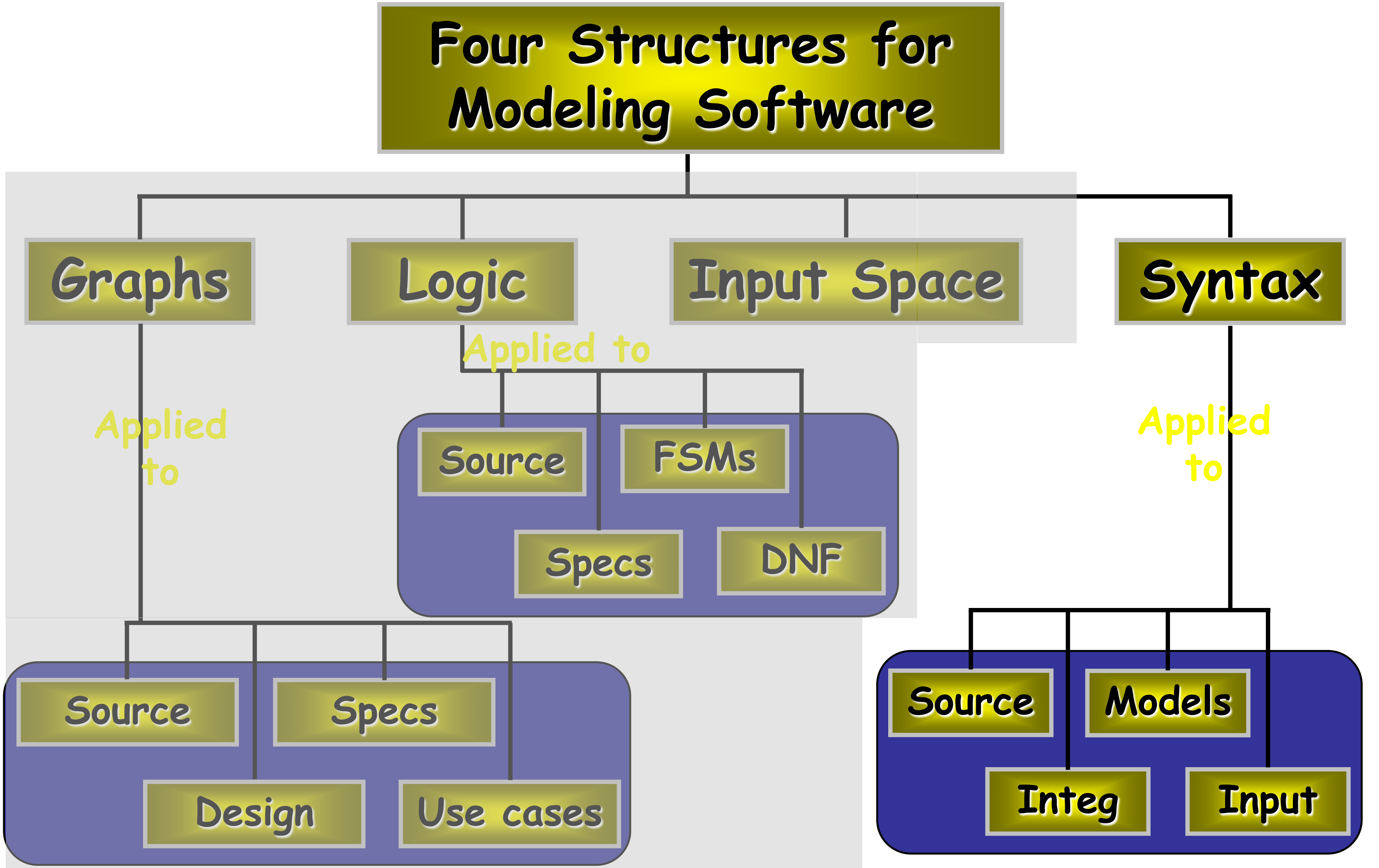
Software Engineering Testing

Syntax-based Testing

Syntax Coverage



Using the Syntax to Generate Tests

- Lots of software artifacts follow <u>strict syntax</u> rules
- The syntax is often expressed as some sort of <u>grammar</u> such as Backus–Naur Form (BNF)
- <u>Syntactic descriptions</u> can come from many sources
  - Programs
  - Integration elements
  - Design documents
  - Input descriptions
- Tests are created with <u>two general goals</u>
  - <u>Cover</u> the syntax in some way
  - <u>Violate</u> the syntax (invalid tests)

Grammar Coverage Criteria

- Software engineering makes practical use of automata theory in several ways
  - Programming languages defined in BNF
  - Program behavior described as finite state machines
  - Allowable inputs defined by grammars
- A simple regular expression:

(*G s n* | *B t n*)*

'*' is *closure* operator, zero or more occurrences

'|' is *choice*, either one can be used

- **Any sequence of "*G s n*" and "*B t n*"**
- **'*G*' and '*B*' could be commands, methods, or events**
- **'*s*', '*t*', and '*n*' could represent arguments, parameters, or values**
- **'*s*', '*t*', and '*n*' could be literals or a set of values**

Test Cases from Grammar

- A string that satisfies the derivation rules is said to be "*in the grammar*"
- A test case is a sequence of strings that satisfy the regular expression
- Suppose 's', 't' and 'n' are numbers

G 18 08.01.90

B 14 06.27.94

G 13 11.21.94

B 05 01.09.03

Could be one test with four parts,

four separate tests, . . .

BNF Grammars

Stream ::= action*

Start symbol

action ::= actG | actB

Non-terminals

actG ::= “G” s n

actB ::= “B” t n

Production rule

s ::= digit$^{1\text{-}3}$

Terminals

t ::= digit$^{1\text{-}3}$

n ::= digit^{2} “.” digit^{2} “.” digit^{2}

**digit ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” |
“7” | “8” | “9”**

Using Grammars

Stream ::= action action *

::= actG action*

::= G s n action*

::= G digit$^{1\text{-}3}$ digit^{2} . digit^{2} . digit^{2} action*

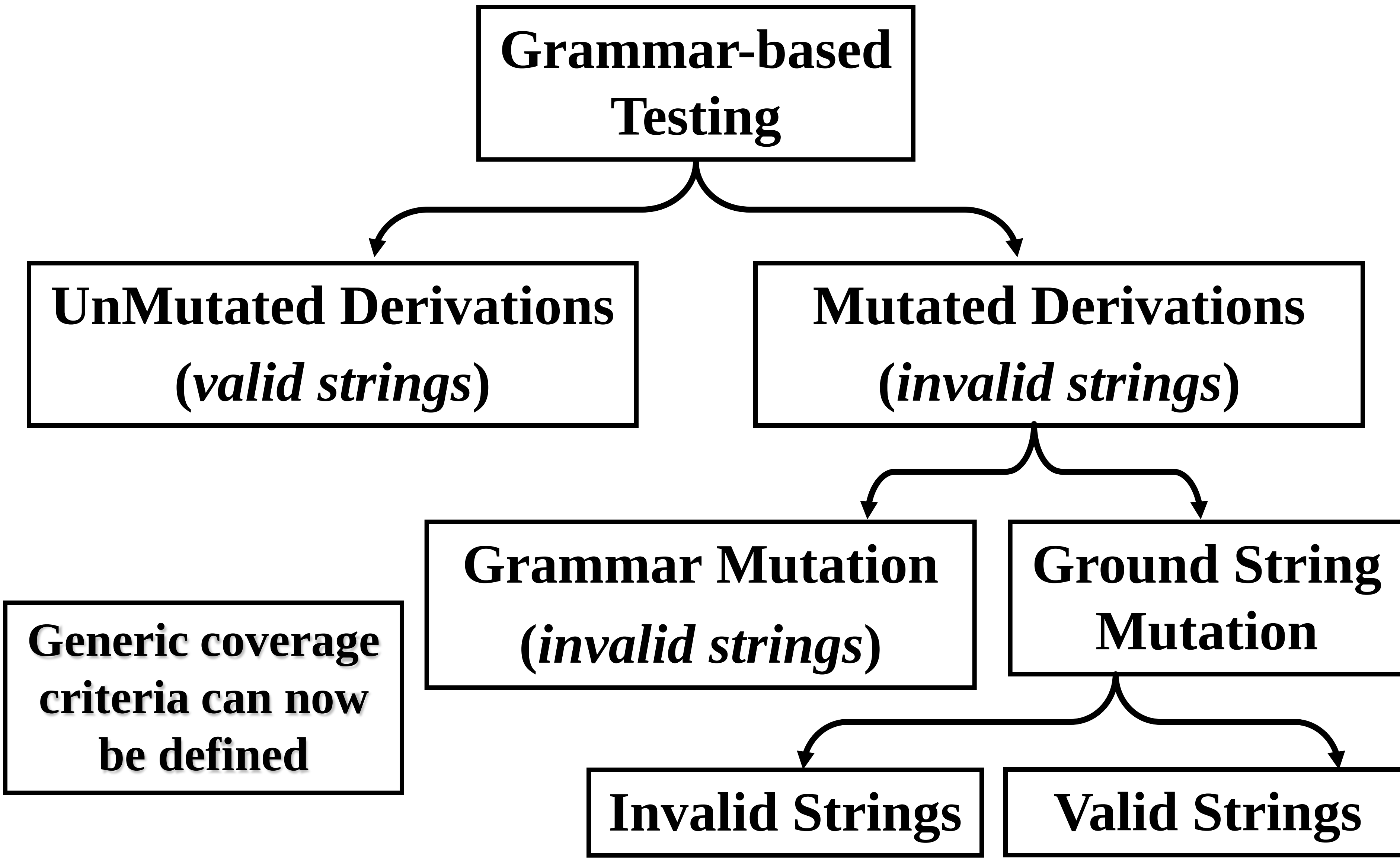
::= G digitdigit digitdigit.digitdigit.digitdigit action*

::= G 18 08.01.90 action*

...

- Recognizer : Given a string (or test), is the string in the grammar ?
  - This is called parsing
  - Tools exist to support parsing
  - Programs can use them for input validation
- Generator : Given a grammar, derive strings in the grammar

Mutation as Grammar-Based Testing



- The most common and straightforward use every terminal and every production at least once

Terminal Symbol Coverage (TSC) : TR contains each terminal symbol *t* in the grammar *G*.

Production Coverage (PC) : TR contains each production *p* in the grammar *G*.

- **PC subsumes TSC**
- **Grammars and graphs are interchangeable**
- **Other graph-based coverage criteria could be defined on grammar**
  - **But have not**

- A related criterion is the impractical one of deriving all possible strings

Derivation Coverage (DC) : TR contains every possible string that can be derived from the grammar *G*.

- **The number of TSC tests is bound by the number of terminal symbols**
  - **13 in the stream grammar**
- **The number of PC tests is bound by the number of productions**
  - **18 in the stream grammar**
- **The number of DC tests depends on the details of the grammar**
  - **2,000,000,000 in the stream grammar !**
- **All TSC, PC and DC tests are in the grammar … how about tests that are NOT in the grammar ?**

- Grammars describe both valid and invalid strings
- Both types can be produced as mutants
- 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 ?

General View

mutation operators

We are performing mutation analysis whenever we

- **use well defined rules**
- **defined on syntactic descriptions**
- **to make systematic changes**
- **to the syntax or to objects developed from the syntax**

grammars

Applied universally or according to empirically verified distributions

grammar

ground strings

- Ground string: A string in the grammar
  - The term “ground” is used as a reference to algebraic ground terms
- 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

- The key to mutation testing is the design of the mutation operators
  - Well designed operators lead to powerful testing
- Sometimes mutant strings are based on ground strings
- Sometimes they are derived directly from the grammar
  - Ground strings are used for valid tests
  - Invalid tests do not need ground strings

Valid Mutants

Ground Strings	Mutants
G 18 08.01.90	*B 18 08.01.90*
B 14 06.27.94	*B 45 06.27.94*

Invalid Mutants

13 18 08.01.90
B 134 06.27.1

Questions About Mutation

- Should more than one operator be applied at the same time ?
  - Should a mutated string contain one mutated element or several?
  - Almost certainly not – multiple mutations can interfere with each other
  - Extensive experience with program-based mutation indicates not

- Should every possible application of a mutation operator be considered ?
  - Necessary with program-based mutation

- Mutation operators exist for several languages
  - Several programming languages (*Fortran, Lisp, Ada, C, C++, Java*)
  - Specification languages (*SMV, Z, Object-Z, algebraic specs*)
  - Modeling languages (*Statecharts, activity diagrams*)
  - Input grammars (*XML, SQL, HTML*)

Killing Mutants

- When ground strings are mutated to create valid strings, the hope is to exhibit different behavior from the ground string

- This is normally used when the grammars are programming languages, the strings are programs, and the ground strings are pre-existing programs

- Killing Mutants : Given a mutant $m \in M$ for a derivation D and a test t, t is said to kill m if and only if the output of t on D is different from the output of t on m

- The derivation D may be represented by the list of productions or by the final string

- Coverage is defined in terms of killing mutants.

Mutation Coverage (MC) : For each $m \in M$, TR 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**

- When creating invalid strings, we just apply the operators
- This results in two simple criteria
- It makes sense to either use every operator once or every production once

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 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.

Example

Grammar

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

Ground String

G 18 08.01.90

B 14 06.27.94

Mutation Operators

- *Exchange actG and actB*
- *Replace digits with other digits*

Mutants using MPC

B 18 08.01.90	*G 14 06.27.94*
G 28 08.01.90	*B 11 06.27.94*
G 38 08.01.90	*B 13 06.27.94*
G 48 08.01.90	*B 15 06.27.94*
G 58 08.01.90	*B 16 06.27.94*
…	…

Mutants using MOC

B 18 08.01.90

B 19 06.27.94

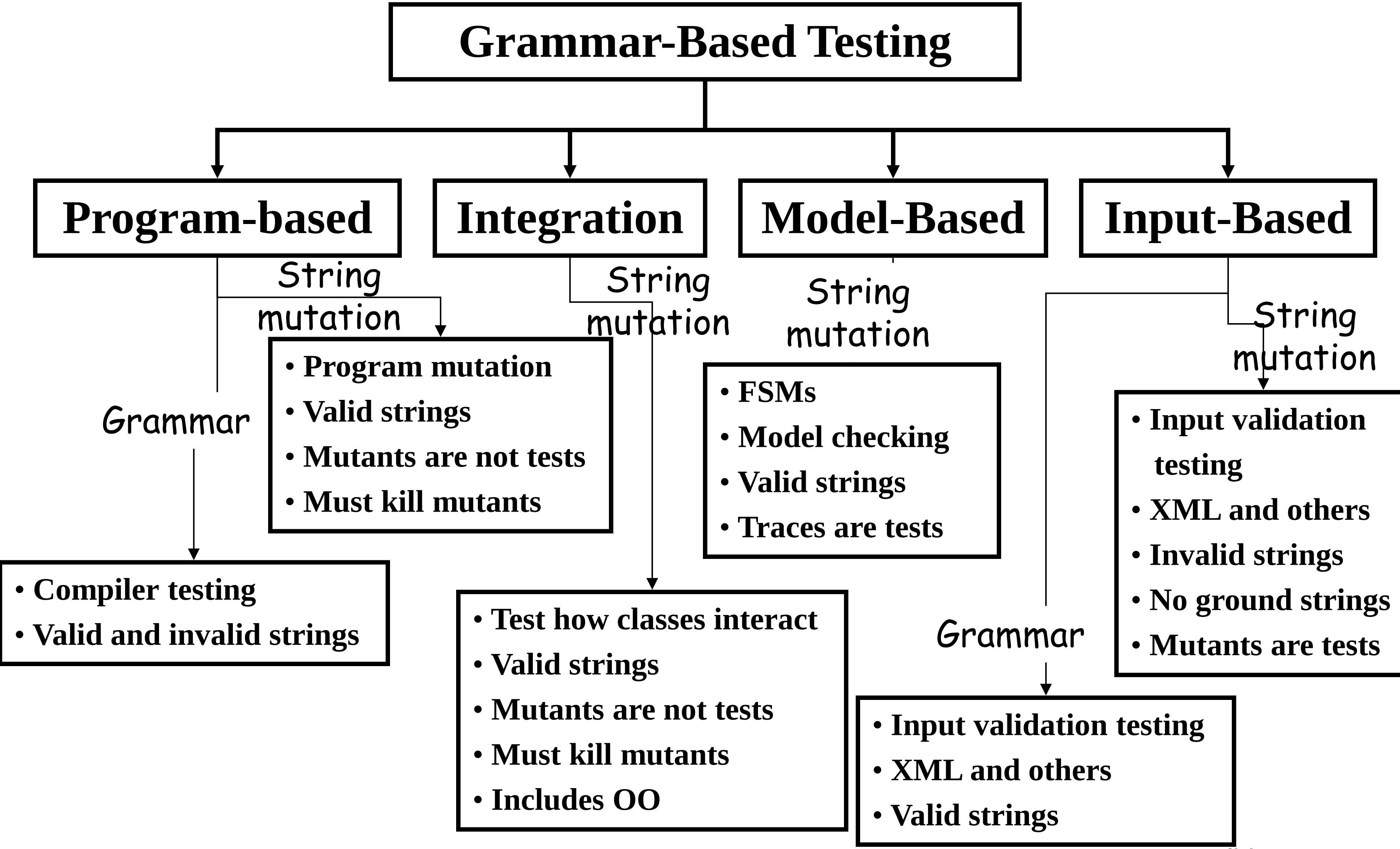
- The number of test requirements for mutation depends on two things
  - The syntax of the artifact being mutated
  - The mutation operators

- Mutation testing is very difficult to apply by hand

- Mutation testing is very effective – considered the “gold standard” of testing

- Mutation testing is often used to evaluate other criteria

Instantiating Grammar-Based Testing



Applying Syntax-based Testing to Programs

Applying Syntax-based Testing to Programs

- Syntax-based criteria originated with programs and have been used most with programs
- BNF criteria are most commonly used to test compilers
- Mutation testing criteria are most commonly used for unit testing and integration testing of classes

Instantiating Grammar-Based Testing

Grammar-Based Testing

- **Program-based**
  - Grammar
    - Compiler testing
    - Valid and invalid strings
  - String mutation
    - Program mutation
    - Valid strings
    - Mutants are not tests
    - Must kill mutants
- **Integration**
  - String mutation
    - Test how classes interact
    - Valid strings
    - Mutants are not tests
    - Must kill mutants
    - Includes OO
- **Model-Based**
  - String mutation
    - FSMs
    - Model checking
    - Valid strings
    - Traces are tests
- **Input-Based**
  - Grammar
    - Input validation testing
    - XML and others
    - Valid strings
  - String mutation
    - Input validation testing
    - XML and others
    - Invalid strings
    - No ground strings
    - Mutants are tests

BNF Testing for Compilers

- Testing compilers is very complicated
  - Millions of correct programs !
  - Compilers must recognize and reject incorrect programs
- BNF criteria can be used to generate programs to test all language features that compilers must process
- This is a very specialized application and not discussed in detail

- The original and most widely known application of syntax-based testing is to modify programs
- Operators modify a ground string (program under test) to create mutant programs
- Mutant programs must compile correctly (valid strings)
- Mutants are not tests, but used to find tests
- Once mutants are defined, tests must be found to cause mutants to fail when executed
- This is called “killing mutants”

Killing Mutants

Given a mutant $m \in M$ for a ground string program P and a test t, t is said to <u>kill</u> m if and only if the output of t on P is different from the output of t on m.

- If mutation operators are designed well, the resulting tests will be very powerful
- Different operators must be defined for different programming languages and goals
- Testers can keep adding tests until all mutants have been killed
  - <u>Dead mutant</u> : A test case has killed it
  - <u>Stillborn mutant</u> : Syntactically illegal
  - <u>Trivial mutant</u> : Almost every test can kill it
  - <u>Equivalent mutant</u> : No test can kill it (equivalent to original program)

Original Method

```
int Min (int A, int B)
{
      int minVal;
      minVal = A;
      if (B < A)
      {
          minVal = B;
       }
       return (minVal);
} // end Min
```

With Embedded Mutants

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
∆ 1   minVal = B;
        if (B < A)
∆ 2   if (B > A)
∆ 3   if (B < minVal)
        {
                minVal = B;
∆ 4           Bomb ();
∆ 5           minVal = A;
∆ 6           minVal = failOnZero (B);
        }
        return (minVal);
} // end Min
```

Replace one variable with another

Changes operator

Immediate runtime failure … if reached

Immediate runtime failure if B==0 else does nothing

6 mutants

Each represents a separate program

Mutation Coverage (MC) : For each $m \in M$, TR contains exactly one requirement, to kill m.

- **The RIP model from chapter 1:**
  - **Reachability : The test causes the faulty statement to be reached (in mutation – the mutated statement)**
  - **Infection : The test causes the faulty statement to result in an incorrect state**
  - **Propagation : The incorrect state propagates to incorrect output**
- **The RIP model leads to two variants of mutation coverage …**

- **1) Strongly Killing Mutants:**

  Given a mutant $m \in M$ for a program P and a test t, t is said to <u>strongly kill</u> m if and only if the <u>output</u> of t on P is different from the output of t on m

- **2) Weakly Killing Mutants:**

  Given a mutant $m \in M$ that modifies a location l in a program P, and a test t, t is said to <u>weakly kill</u> m if and only if the <u>state</u> of the execution of P on t is different from the state of the execution of m immediately on t after l

  - **<u>Weakly killing</u> satisfies reachability and infection, but not propagation**

<u>Weak Mutation Coverage (WMC)</u> : For each $m \in M$, TR contains exactly one requirement, to weakly kill m.

- **"Weak mutation" is so named because it is <u>easier to kill</u> mutants under this assumption**
- **Weak mutation also requires <u>less analysis</u>**
- **Some mutants can be killed under weak mutation but not under strong mutation (<u>no propagation</u>)**
- **In practice, there is <u>little difference</u>**

Weak Mutation Example

- Mutant 1 in the Min( ) example is:

```
      minVal = A;
Δ 1  minVal = B;
       if (B < A)
           minVal = B;
```

- **The complete test specification to kill mutant 1:**

- **Reachability** : ***true*** **// Always get to that statement**
- **Infection** : $A \neq B$
- **Propagation**: $(B < A) = false$ **// Skip the next assignment**
- **Full Test Specification** : $true \wedge (A \neq B) \wedge ((B < A) = false)$

  $\equiv (A \neq B) \wedge (B \geq A)$

  $\equiv (B > A)$

- **(A = 5, B = 7) will weakly kill mutant 1, but not strongly**

Equivalent Mutation Example

```
      minVal = A;
      if (B < A)
Δ 3   if (B < minVal)
```

- Mutant 3 in the Min() example is equivalent:

- **The infection condition is “(B < A) != (B < minVal)”**

- **However, the previous statement was “minVal = A”**
  - **Substituing, we get: “(B < A) != (B < A)”**

- **<u>Thus no input can kill this mutant</u>**

Strong Versus Weak Mutation

```
1     boolean isEven (int X)
2     {
3          if (X < 0)
4              X = 0 - X;
△ 4            X = 0;
5          if (float) (X/2) == ((float) X) / 2.0
6              return (true);
7          else
8              return (false);
9          }
```

<u>Reachability</u> : X < 0

<u>Infection</u> : X != 0

(X = -6) will kill mutant 4 under <u>weak mutation</u>

<u>Propagation</u> :

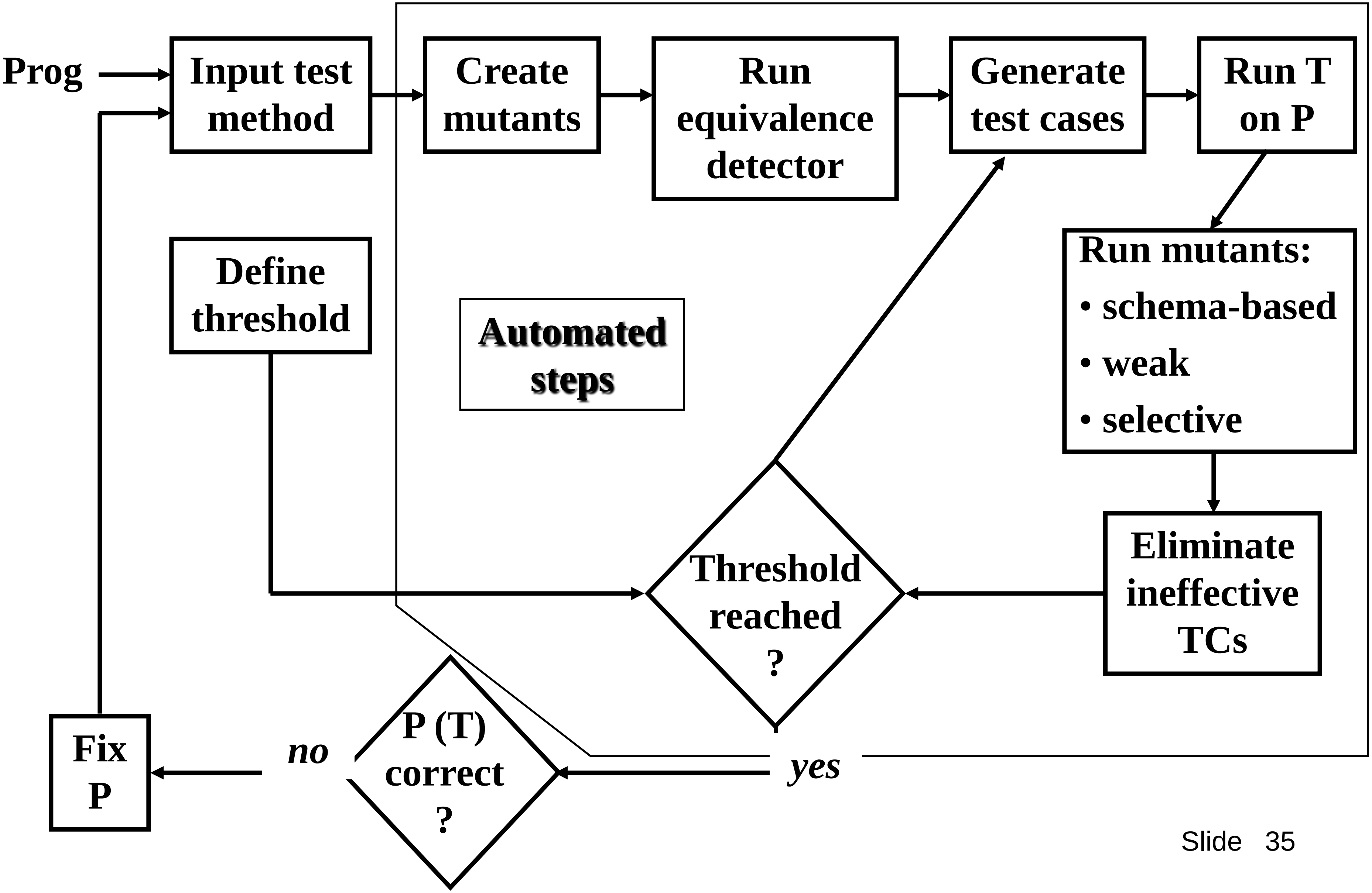
((float) ((0-X)/2) == ((float) 0-X) / 2.0)

!= ((float) (0/2) == ((float) 0) / 2.0)

That is, X is <u>not</u> even …

Thus (X = -6) does <u>*not*</u> kill the mutant under strong mutation

Testing Programs with Mutation



<u>Fundamental Premise of Mutation Testing</u>

If the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault

- This is not an absolute!
- The mutants guide the tester to a very effective set of tests
- A very challenging problem :
  - Find a <u>fault</u> and a set of <u>mutation-adequate tests</u> that do <u>not</u> find the fault
- Of course, this depends on the mutation operators …

- At the method level, mutation operators for different programming languages are similar
- Mutation operators do one of two things:
  - Mimic typical programmer mistakes (incorrect variable name)
  - Encourage common test heuristics (cause expressions to be 0)
- Researchers design lots of operators, then experimentally *select* the most useful

Effective Mutation Operators

If tests that are created specifically to kill mutants created by a collection of mutation operators *O* = {*o1*, *o2*, …} also kill mutants created by all remaining mutation operators with very high probability, then *O* defines an effective set of mutation operators

Mutation Operators for Java

1. ABS — Absolute Value Insertion:

Each arithmetic expression (and subexpression) is modified by the functions *abs()*, *negAbs()*, and *failOnZero()*.

2. AOR — Arithmetic Operator Replacement:

Each occurrence of one of the arithmetic operators +, —,*, ⁄, and % is replaced by each of the other operators. In addition, each is replaced by the special mutation operators *leftOp*, and *rightOp*.

3. ROR — Relational Operator Replacement:

Each occurrence of one of the relational operators (<, ≤, >, ≥, =, ≠) is replaced by each of the other operators and by falseOp and trueOp.

Mutation Operators for Java (2)

4. COR — Conditional Operator Replacement:

Each occurrence of one of the logical operators (and - &&, or - || , and with no conditional evaluation - &, or with no conditional evaluation - |, not equivalent - ^) is replaced by each of the other operators; in addition, each is replaced by falseOp, trueOp, leftOp, and rightOp.

5. SOR — Shift Operator Replacement:

Each occurrence of one of the shift operators <<, >>, and >>> is replaced by each of the other operators. In addition, each is replaced by the special mutation operator leftOp.

6. LOR — Logical Operator Replacement:

Each occurrence of one of the logical operators (bitwise and - &, bitwise or - |, exclusive or - ^) is replaced by each of the other operators; in addition, each is replaced by leftOp and rightOp.

Mutation Operators for Java (3)

7. ASR — Assignment Operator Replacement:

Each occurrence of one of the assignment operators (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=) is replaced by each of the other operators.

8. UOI — Unary Operator Insertion:

Each unary operator (arithmetic +, arithmetic -, conditional !, logical ~) is inserted in front of each expression of the correct type.

9. UOD — Unary Operator Deletion:

Each unary operator (arithmetic +, arithmetic -, conditional !, logical~) is deleted.

Mutation Operators for Java (4)

10. SVR — Scalar Variable Replacement:

Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.

11. BSR — Bomb Statement Replacement:

Each statement is replaced by a special Bomb() function.

Subsumption of Other Criteria

- Mutation is widely considered the strongest test criterion
  – And most expensive !
- Mutation subsumes other criteria by including specific mutation operators
- Subsumption actually only makes sense for weak mutation – other criteria impose local requirements, like weak mutation
  – Node coverage
  – Edge coverage
  – Clause coverage
  – General active clause coverage
  – Correlated active clause coverage
  – All-defs data flow coverage

One more example - Grammar for Bank Example

- Consider a program that processes a sequence of deposits and debits to a bank

<u>Inputs</u>

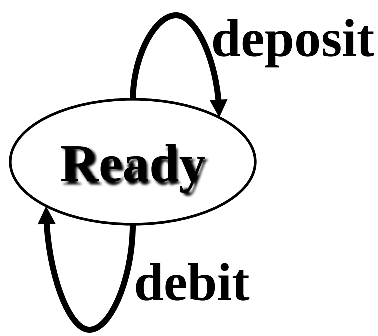
deposit 5306 $4.30

debit 0343 $4.14

deposit 5306 $7.29

<u>Initial Grammar</u>

(deposit account amount | debit account amount) *



FSM to represent the grammar

Grammar for Bank Example (contd.)

- Grammars are more expressive than regular expressions – they can capture more details

```
bank      ::=  action*
action    ::=  dep | deb
dep       ::=  "deposit" account amount
deb       ::=  "debit" account amount
account   ::=  digit^4
amount    ::=  "$" digit^+ "." digit^2
digit     ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
               "7" | "8" | "9"
```

References

- Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008