UNIT-4

Syntax Testing

Syntax Testing is a type of black box testing technique which is used to examine the format and the grammar of the data inputs used in the software application, either external or input, which may be formally described in technical or established & specified notations such as BNF and could be used to design input validation tests.

Syntax testing is performed to verify and validate the both internal and external data input to the system, against the specified format, file format, database schema, protocol and other similar things. Generally, syntax tests are automated, as they involve the production of large number of tests.

Syntax testing consists of the following steps:

The methodology of syntax testing can be perceived and understood through following steps in the sequential order:

- The very first step of the syntax testing involves the identification of the target language or format.
- Thereafter, syntax of the language is defined, as specified in the formal notation such as **Backus-Naur form (BNF)**. As each and every input has some syntax, which may be formally pre-specified, undocumented, etc.
- The final step involves testing and debugging the syntax to ensure its completeness & consistency. Generally, the syntax is tested using two conditions as stated below.
  - Testing the normal condition using the covering set of paths of the syntax graph, for the minimum necessary requirements.
  - Testing the garbage condition*, using invalid set of input data.

Note*: Garbage condition is the method of testing the system's tolerance against the bad or dirty data. The condition is executed by providing dirty data (invalid data) to the system, which is being not supported by the specified format and grammar of the syntax.

Garbage in, Garbage out (GIGO)

GIGO (garbage in, garbage out) is a concept common to computer science and mathematics: the quality of output is determined by the quality of the input. So, for example, if a mathematical equation is improperly stated, the answer is unlikely to be correct. Similarly, if incorrect data is input to a program, the output is unlikely to be informative.

Applications of Syntax Testing

The following are the applications of syntax testing:

1. **Command-Driven Software:** This is the most obvious application and probably the most common. If a system is mostly command driven, then much of its testing can be organized under syntax testing.
2. **Menu-Driven Software:** The popular alternative to command-driven software is menu-driven software, in which actions are initiated by selecting from choices presented in a menu. However, menu-driven or not, there are still data fields to enter and those fields have a syntax against which syntax testing is effective. It's usually very easy, though, because the syntax tree tends to be shallow
3. **Macro Languages**: Many commercial software packages for PCs have a macro language, also called a scripting language. This is a programming language that can be used to automate repetitive operations. These languages are often implemented in commercial packages not just for the users' convenience but because they are a neat way to implement a lot of complicated features, such as mail-merge in a word processor. In their best form, these are full-blown, albeit specialized, programming languages with formal specifications (often in BNF or the equivalent). Non-commercial software or software that serves a narrow segment of an industry*also may have a macro language, but usually, it's not as clearly defined or implemented as the commercial (horizontal) packages. Such languages, if they are part of an application, warrant syntax testing.
4. **Communications:** All communications systems have an embedded language. It is the language of the format of messages. Even telephone exchanges use language to communicate with each other. But proper telephone numbers, local, long distance and international, have a very formal syntax called a number plan. Every message used to communicate must have a format (i.e., syntax) that must be parsed.
5. **Database Query Languages**: Any database system has a command language used to specify what is to be searched and what is to be retrieved. The simplest ones allow only one key. The more mature systems allow boolean searches based on user-supplied parameters, which we recognize as being predicates. Such predicates obviously have a formal syntax and semantics, and should, therefore, be treated to syntax testing.
6. **Compilers and Generated Parsers**: The one place syntax testing should not be used, especially dirty syntax testing, is to test a modern compiler. This might seem perverse and strange, but a tester should never repeat the tests previously done by another. Modern compilers have a parser, of course. But that parser is generated completely automatically by use of a parser generator given a formal definition in something like BNF. Lexical analyzers are also

generated automatically from formal specifications. From a testing viewpoint, there's not much that syntax testing, even when fully automated, can do to break such lexers and parsers. Before you spend a lot of effort on syntax testing (even if automated), look at the application and how it has been implemented. If a generated lexer and parser are used or planned, you're unlikely to have great success in using syntax testing; those kinds of bugs have been totally prevented. The only thing left to test is semantics, which is often done by another technique such as domain testing.

Backus-Naur notation (BNF)

Backus-Naur Form (BNF) is a notation technique used to describe recursively the syntax of

- programming languages
- document formats
- communication protocols and etc.

Syntax:

<symbol>::= exp1|exp2|exp3|……..

Where <symbol> is non-terminal, expression are sequences of terminals and non-terminals.

Example:

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

- "::=" means "is defined as", LHS can be replaced with RHS (some variants use ":=")
- "|" means "or"
- Angle brackets mean a nonterminal
- Symbols without angle brackets are terminals

Operators in BNF:

[ ]	Brackets enclose optional items.
{ }	Braces enclose items only one of which is required.
|	A vertical bar separates alternatives within brackets or braces.
+ or #	The character "+" or "#" preceding an element indicates one more occurrences
*	The character "*" preceding an element indicates zero more occurrences
. (dot)	Concatenation

Example:

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

<unsigned integer> ::= <digit> | <unsigned integer><digit>

<integer> ::= <unsigned integer> | + <unsigned integer> | −<unsigned integer>

<letter> ::= a | b | c | . . .

<identifier> ::= <letter> | <identifier><letter> | <identifier><digit>

Designed in the 1950–60s to define the syntax of the programming language ALGOL

Example

As an example, consider this possible BNF for a postal address:

```
<postal-address> ::= <name-part> <street-address> <zip-part>
<name-part> ::= <personal-part> <name-part>
<personal-part> ::= <initial> "." | <first-name>
<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>
<zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>
<opt-apt-num> ::= <apt-num> | ""
```

This translates into English as:

- A postal address consists of a name-part, followed by a street-address part, followed by a zip-code part.
- A name-part consists of a personal part followed by a name part (this rule illustrates the use of recursion in BNFs, covering the case of people who use multiple first and middle names and initials).
- A personal-part consists of either a first name or an initial followed by a dot.
- A street address consists of a house number, followed by a street name, followed by an optional apartment specified, followed by an end-of-line.
- A zip-part consists of a town-name, followed by a comma, followed by a state code, followed by a ZIP-code followed by an end-of-line.
- An opt-suffix-part consists of a suffix, such as "Sr.", "Jr." or a roman-numeral, or an empty string (i.e. nothing).
- An opt-apt-num consists of an apartment number or an empty string (i.e. nothing).

Note that many things (such as the format of a first-name, apartment specifier, ZIP-code, and Roman numeral) are left unspecified here. If necessary, they may be described using additional BNF rules.

Why BNF?

• Using a BNF specification is an easy way to design format-validation test cases.

• It is also an easy way for designers to organize their work.

• You should not begin to design tests until you are able to distinguish incorrect data from correct data.

Test Case Generation

• There are three possible kinds of incorrect actions:

- Recognizer does not recognize a good string.
- Recognizer accepts a bad string.
- Recognizer crashes during attempt to recognize a string.

• Even small BNF specifications lead to many good strings and far more bad strings.

• There is neither time nor need to test all strings.

Testing case design Strategy

• Create one error at a time, while keeping all other components of the input string correct.

• Once a complete set of tests has been specified for single errors, do the same for double errors, then triple, errors, ...

• Focus on one level at a time and keep the level above and below as correct as you can.

Sources of Syntax

• Designer-Tester Cooperation
• Manuals
• Help Screens
• Design Documents
• Prototypes
• Programmer Interviews
• Experimental (hacking)

Dangers of Syntax Test Design

• It’s easy to forget the “normal” cases.
• Don’t go overboard with combinations:

  - Syntax testing is easy compared to structural testing.

- Don't ignore structural testing because you are thorough in syntax testing.
- Knowing a program's design may help you eliminate cases without sacrificing the thoroughness of the testing process

Syntax Testing Drivers

• Build a driver program that automatically sequences through a set of test cases usually stored as data.

• Do not try to build the "garbage" strings automatically because you will be going down a diverging infinite sequence of syntax testing.

Design Automation:

Primitive Method

• Use a word processor to specify a covering set of correct input strings.

• Using search/replace, replace correct substrings with incorrect ones.

• Using the syntax definition graph as a guide, generate all single-error cases.

• Do same for double errors, triple errors.

Random String Generators

• Easy to do, but useless.

• Random strings get recognized as invalid too soon.

• The probability of hitting vulnerable points is too low because there are simply too many "garbage" strings.

Productivity, Training, Effectiveness

• Syntax testing is a great confidence builder for people who have never designed tests.

• A testing trainee can easily produce 20-30 test cases per hour after a bit of training.

• Syntax testing is an excellent way of convincing a novice tester that:

- Testing is often an infinite process.
- A tester's problem knows which tests to ignore.

Ad-lib Testing

• Ad-lib testing is futile and doesn't prove anything.

• Most of the ad-lib tests will be input strings with format violations.

• Ad-lib testers will try good strings that they think are bad ones!

• If ad-lib tests are able to prove something, then the system is so buggy that it deserves to be thrown out!

Logic-based testing

Logic-based testing is structural testing when it's applied to structure (e.g., control flow graph of an implementation); it is functional testing when it is applied to a specification. In logic-based testing we focus on the truth values of control flow predicates. Logic-based testers design tests from logical expressions that appear in software artifacts such as source code, design models, and requirements specifications.

Programmers and logic:

- "Logic" is one of the most often used words in programmer's vocabularies but one of their least used techniques.
- Boolean algebra is being the simplest form of logic.
- Boolean algebra is to logic as arithmetic is to mathematics. Without it, the tester or programmer is cut off from many test and design techniques and tools that incorporate those techniques.

Hardware Logic Testing:

- Logic has been the primary tool of hardware logic designers.
- Many test methods developed for hardware logic can be adapted to software logic testing. Because hardware testing automation is 10 to 15 years ahead of software testing automation.
- So hardware testing methods and its associated theory is a perfect opportunity for software testing methods.

Specification Systems and Languages:

- The trouble with specifications is that they're hard to express.
- Boolean algebra (also known as the sentential calculus) is the most basic of all logic systems.
- Higher-order logic systems are needed and used for formal specifications.
- Much of logical analysis can be and is embedded in tools. But these tools incorporate methods to simplify, transform, and check specifications, and the methods are to a large extent based on Boolean algebra.

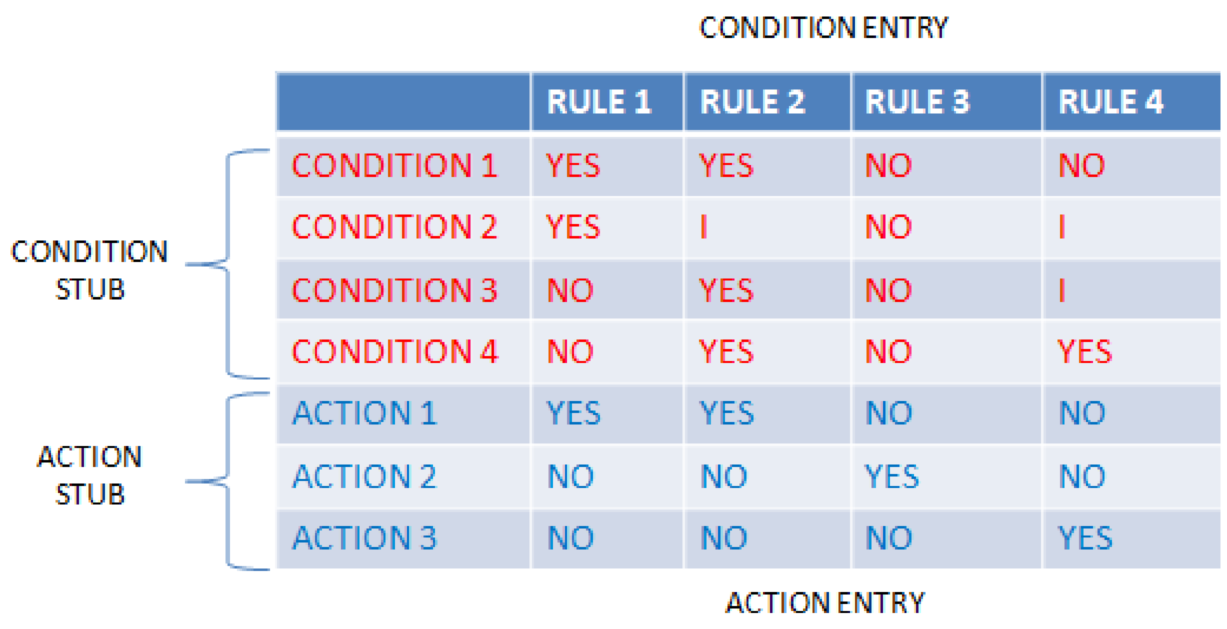
KNOWLEDGE BASED SYSTEM:

- The **knowledge-based system** (also expert system, or "artificial intelligence" system) has become the programming construct of choice for many applications that were once considered very difficult.
- Knowledge-based systems incorporate knowledge from a knowledge domain such as medicine, law, or civil engineering into a database. The data can then be queried and interacted with to provide solutions to problems in that domain.

- One implementation of knowledge-based systems is to incorporate the expert's knowledge into a set of rules. The user can then provide data and ask questions based on that data.
- The user's data is processed through the rule base to yield conclusions (tentative or definite) and requests for more data. The processing is done by a program called the **inference engine**.
- Understanding knowledge-based systems and their validation problems requires an understanding of formal logic.

Decision Table Testing:

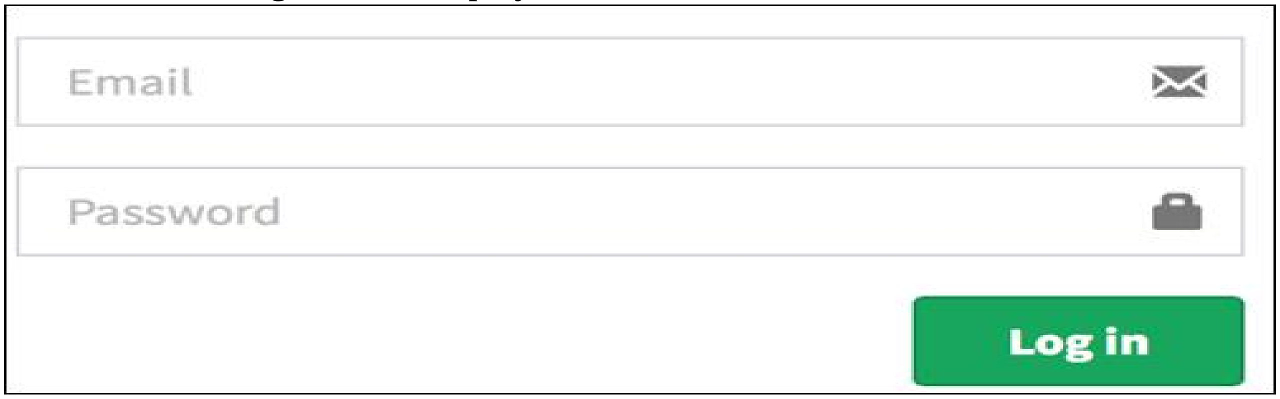
- Decision table testing is a software testing technique used to test system behavior for different input combinations.
- This is a systematic approach where the different input combinations and their corresponding system behavior (Output) are captured in a tabular form.
- That is why it is also called as a Cause-Effect table where Cause and effects are captured for better test coverage.
- A Decision Table is a tabular representation of inputs versus rules/cases/test conditions. Let's learn with an example.
- Decision table consists of four areas called
  1 Condition stub
  2 Condition entry
  3 Action stub
  4 Action entry
- Each column of the table is a **rule** that specifies the conditions under which the actions named in the action stub will take place.
- The **condition stub** is a list of names of conditions.
- A rule specifies whether a condition should or should not be met for the rule to be satisfied. "YES" means that the condition must be met, "NO" means that the condition must not be met, and "I" means that the condition plays no part in the rule, or it is immaterial to that rule.
- The **action stub** names the actions the routine will take or initiate if the rule is satisfied.
- If the action entry is "YES", the action will take place; if "NO", the action will not take place.


CONDITION ENTRY
CONDITION STUB
ACTION STUB
ACTION ENTRY

	RULE 1	RULE 2	RULE 3	RULE 4
CONDITION 1	YES	YES	NO	NO
CONDITION 2	YES	I	NO	I
CONDITION 3	NO	YES	NO	I
CONDITION 4	NO	YES	NO	YES
ACTION 1	YES	YES	NO	NO
ACTION 2	NO	NO	YES	NO
ACTION 3	NO	NO	NO	YES

Example 1: How to make Decision Base Table for Login Screen

- Let's create a decision table for a login screen.
- The condition is simple if the user provides correct username and password the user will be redirected to the homepage. If any of the input is wrong, an error message will be displayed.



	Rule 1	Rule 2	Rule 3	Rule 4
Username	**INVALID**	**VALID**	**INVALID**	**VALID**
Password	**INVALID**	**INVALID**	**VALID**	**VALID**

Action 1 (Error Message)	**YES**	**YES**	**YES**	**NO**
Action 2 (HOME SCREEN)	**NO**	**NO**	**NO**	**YES**

Decision Table Processors

- Decision tables can be automatically translated into code and as such are a higher order language.
- The decision table translator checks the source decision table for consistency and completeness and fills in any required default rules.
- Decision tables as a source language have the virtue of clarity, direct correspondence to specifications, and maintainability.

Why is Decision Table Testing is important?

- This testing technique becomes important when it is required to test different combination. It also helps in better test coverage for complex business logic.

Advantages of Decision Table Testing

- When the system behavior is different for different input and not same for a range of inputs, both equivalent partitioning, and boundary value analysis won't help, but decision table can be used.
- The representation is simple so that it can be easily interpreted and is used for development and business as well.
- This table will help to make effective combinations and can ensure a better coverage for testing
- Any complex business conditions can be easily turned into decision tables.
- In a case we are going for 100% coverage typically when the input combinations are low, this technique can ensure the coverage.

Disadvantages of Decision Table Testing

- The main disadvantage is that when the number of input increases the table will become more complex

Decision tables as a Basis for Test Case Design

- If a specification is given as a decision table, it follows that decision tables should be used for test case design.
- If a program's logic is implemented as a decision table, decision table should also be used as a basis for test design.
- It is not always possible or desirable to implement the program as a decision table because the program's logical behavior is only part of its behavior. The program interfaces with other programs, there are restrictions or the decision table language may not have needed features.
- The use of a decision table model to design tests is warranted when:
- The specification is given as a decision table or can be easily converted into one.
- The order in which the predicates are to be evaluated does not affect interpretation of the rules or the resulting action.
- Once a rule is satisfied and an action is selected, no other rule need be examined.
- If several actions can result from satisfying rule, the order in which the actions are executed doesn't matter.

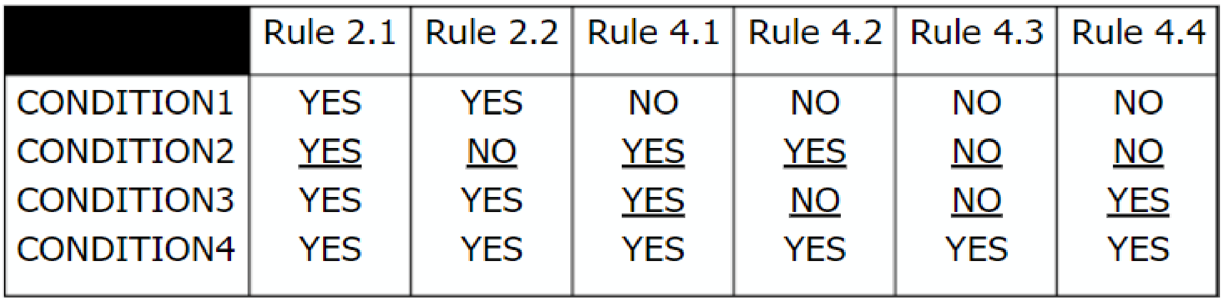
Expansion of Immaterial Cases

- Improperly specified immaterial entries (I) cause most decision-table contradictions.
- If a condition's truth value is immaterial in a rule, satisfying the rule doesnot depend on the condition. It doesn't mean that the case is impossible.
- For example,
- Rule 1: " if the persons are male and over 30, then they shall receive a 15% raise"
- Rule 2: "but if the persons are female, then they shall receive a 10% raise."
- The above rules state that age is material for a male's raise, but immaterial for determining a female's raise.

	RULE 1	RULE 2	RULE 3	RULE 4
CONDITION 1	**YES**	**YES**	**NO**	**NO**
CONDITION 2	**YES**	**I**	**NO**	**I**
CONDITION 3	**NO**	**YES**	**NO**	**I**
CONDITION 4	**NO**	**YES**	**NO**	**YES**
ACTION 1	**YES**	**YES**	**NO**	**NO**
ACTION 2	**NO**	**NO**	**YES**	**NO**
ACTION 3	**NO**	**NO**	**NO**	**YES**

Rule 2 in above table contains on I entry and therefore expands into two equivalent sub-rules. Rule 4 contains two I entries and therefore expands into four sub-rules. The expansion of rules 2 and 4 are shown in below table.

Rule 2 has been expanded by converting the I entry for condition 2 into a separate rule 2.1 for YES and 2.2 for NO. Similarly, condition 2 was expanded in rule 4 to yield intermediate rules 4.1, 4.2,4.3,4.4.



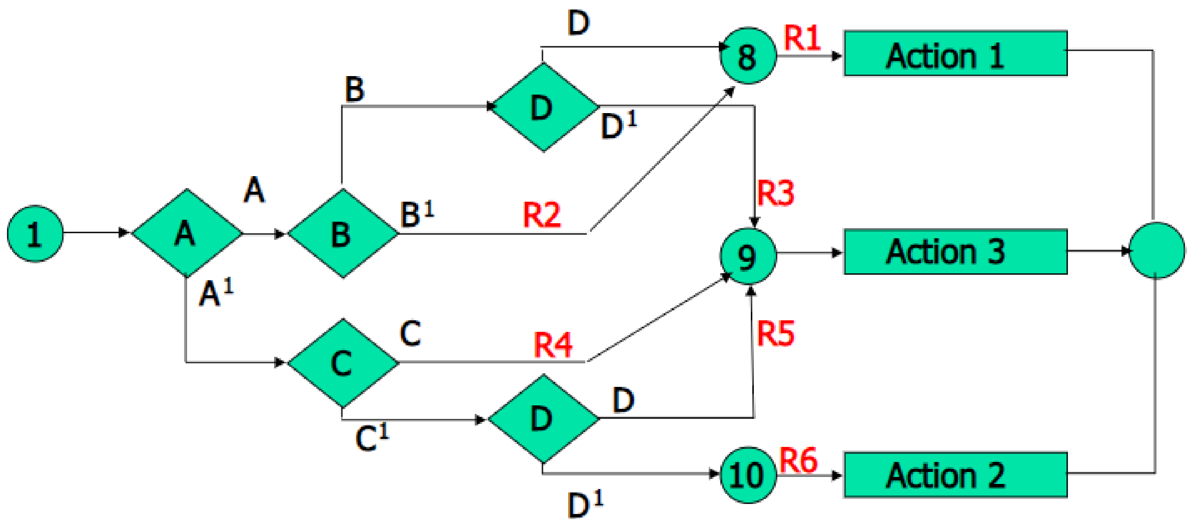
	Rule 2.1	Rule 2.2	Rule 4.1	Rule 4.2	Rule 4.3	Rule 4.4
CONDITION1	YES	YES	NO	NO	NO	NO
CONDITION2	YES	NO	YES	YES	NO	NO
CONDITION3	YES	YES	YES	NO	NO	YES
CONDITION4	YES	YES	YES	YES	YES	YES

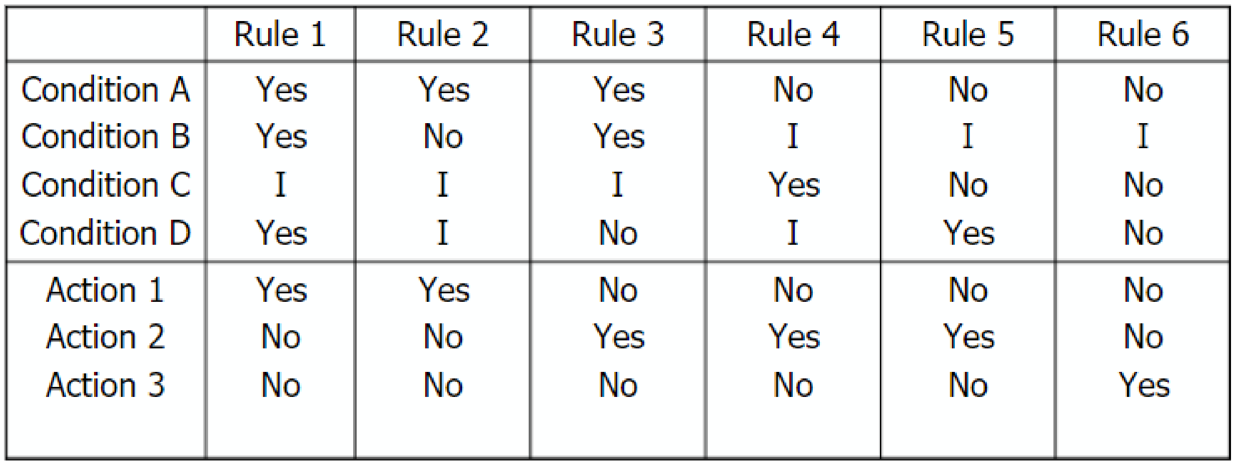
Test Case Design

- Test case design by decision tables begins with examining the specification's consistency and completeness.
- This is done by expanding all immaterial cases and checking the expanded tables.
- Once the specification have been verified, the objective of the test case is to show that the implementation provides the correct action for all combinations of predicate values.

Decision tables and Structure

- Decision tables can also be used to examine a program's structure.





	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6
Condition A	Yes	Yes	Yes	No	No	No
Condition B	Yes	No	Yes	I	I	I
Condition C	I	I	I	Yes	No	No
Condition D	Yes	I	No	I	Yes	No
Action 1	Yes	Yes	No	No	No	No
Action 2	No	No	Yes	Yes	Yes	No
Action 3	No	No	No	No	No	Yes

The decision table corresponding to the previous decision tree

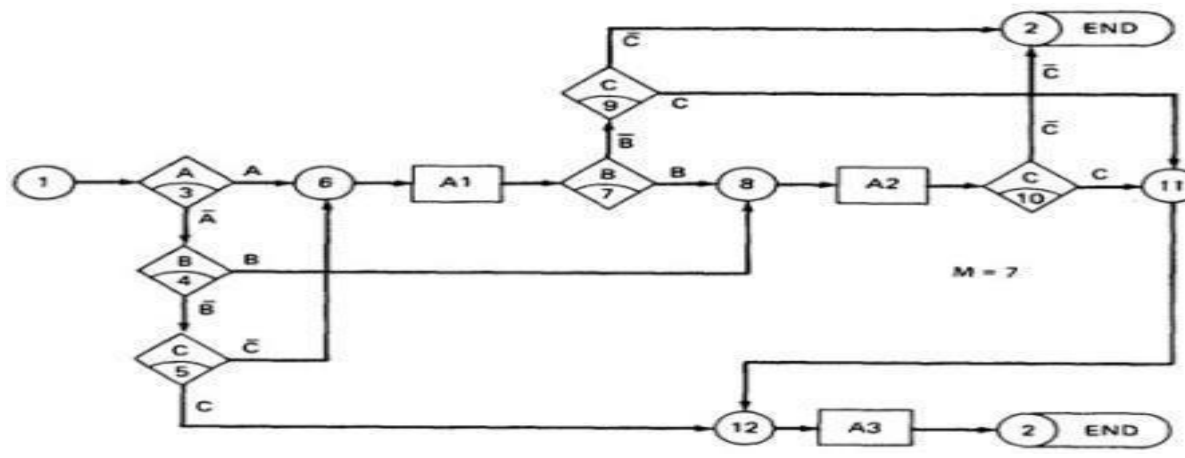
PATH EXPRESSIONS:

- Logic-based testing is structural testing when it's applied to structure (e.g., control flow graph of an implementation); it's functional testing when it's applied to a specification.
- In logic-based testing we focus on the truth values of control flow predicates.
- A **predicate** is implemented as a process whose outcome is a truth-functional value.
- For our purpose, logic-based testing is restricted to binary predicates.
- We start by generating path expressions by path tracing as in Unit V, but this time, our purpose is to convert the path expressions into boolean algebra, using the predicates' truth values (e.g., A and $\overline{A}$) as weights.

BOOLEAN ALGEBRA:

STEPS:

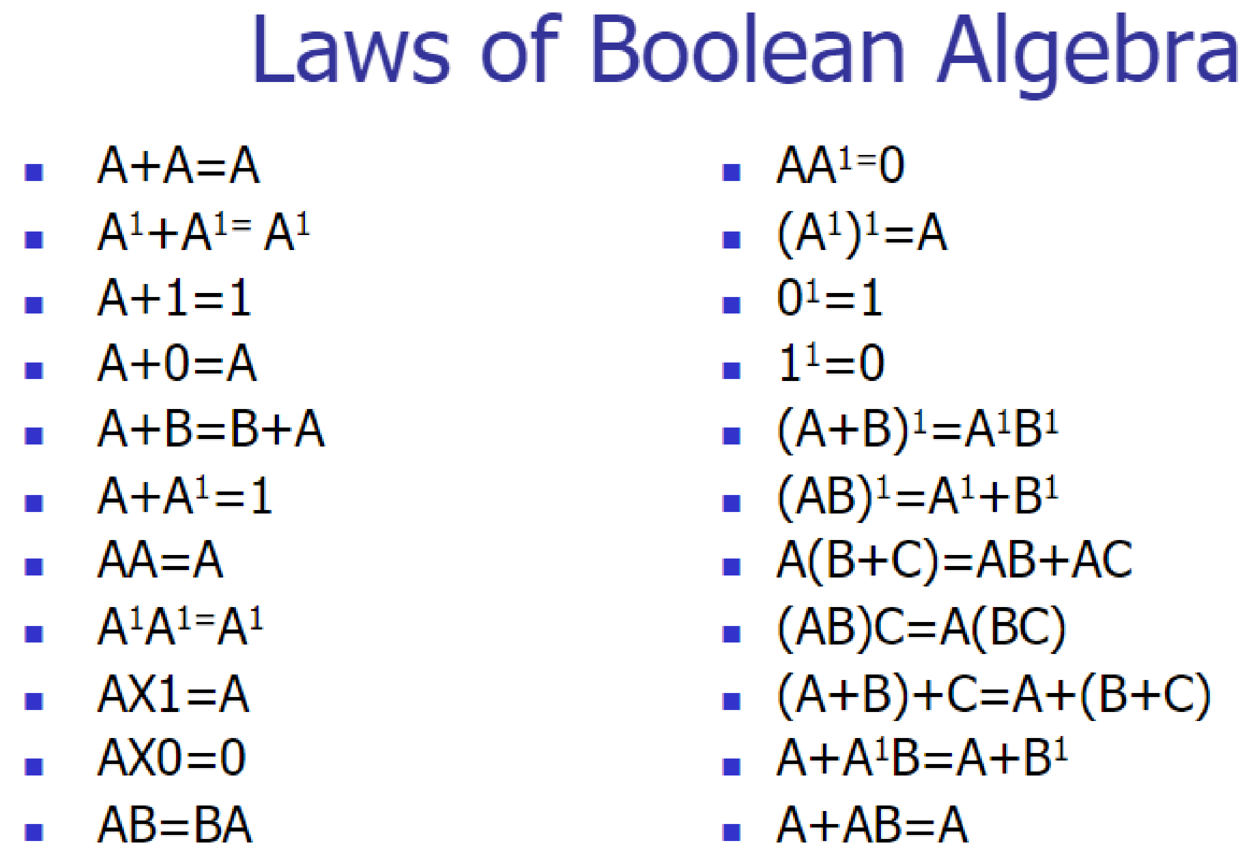
1. Label each decision with an uppercase letter that represents the truth value of the predicate. The YES or TRUE branch is labeled with a letter (say A) and the NO or FALSE branch with the same letter overscored (say $\overline{A}$).
2. The truth value of a path is the product of the individual labels. Concatenation or products mean "AND". For example, the straight-through path of Figure 6.5, which goes via nodes 3, 6, 7, 8, 10, 11, 12, and 2, has a truth value of ABC. The path via nodes 3, 6, 7, 9 and 2 has a value of $AB\overline{C}$.



3. If two or more paths merge at a node, the fact is expressed by use of a plus sign (+) which means "OR".

Using this convention, the truth-functional values for several of the nodes can be expressed in terms of segments from previous nodes. Use the node name to identify the point.

$$\mathbf{N6=A+\bar{A}\bar{B}C}$$
$$\mathbf{N8=(N6)B+\bar{A}B=A\bar{B}+\bar{A}\bar{B}C\bar{B}+AB}$$
$$\mathbf{N11=(N8)C+(N\bar{6})BC}$$
$$\mathbf{N12=N11+\bar{A}\bar{B}C}$$
$$\mathbf{N2=N12+(N8)\bar{C}+(N\bar{6})\bar{B}C}$$



Laws of Boolean Algebra

- $A+A=A$
- $A^1+A^1= A^1$
- $A+1=1$
- $A+0=A$
- $A+B=B+A$
- $A+A^1=1$
- $AA=A$
- $A^1A^1=A^1$
- $AX1=A$
- $AX0=0$
- $AB=BA$
- $AA^1=0$
- $(A^1)^1=A$
- $0^1=1$
- $1^1=0$
- $(A+B)^1=A^1B^1$
- $(AB)^1=A^1+B^1$
- $A(B+C)=AB+AC$
- $(AB)C=A(BC)$
- $(A+B)+C=A+(B+C)$
- $A+A^1B=A+B^1$
- $A+AB=A$

KV-Charts

- The Boolean algebraic expressions are used to determine which cases are interesting and which combination of predicate values should be used to reach which node.
- If you had deal with expressions in four or five or six variables, you could get bogged down in the algebra and make as many errors in the designing test cases as there are bug in the routine you’re tesing.
- The karnaugh – Veitch chart reduces boolean algebraic manipulations to graphical trivia.

One Variable Map

A

	0	1	
0	0	0	The function is never true

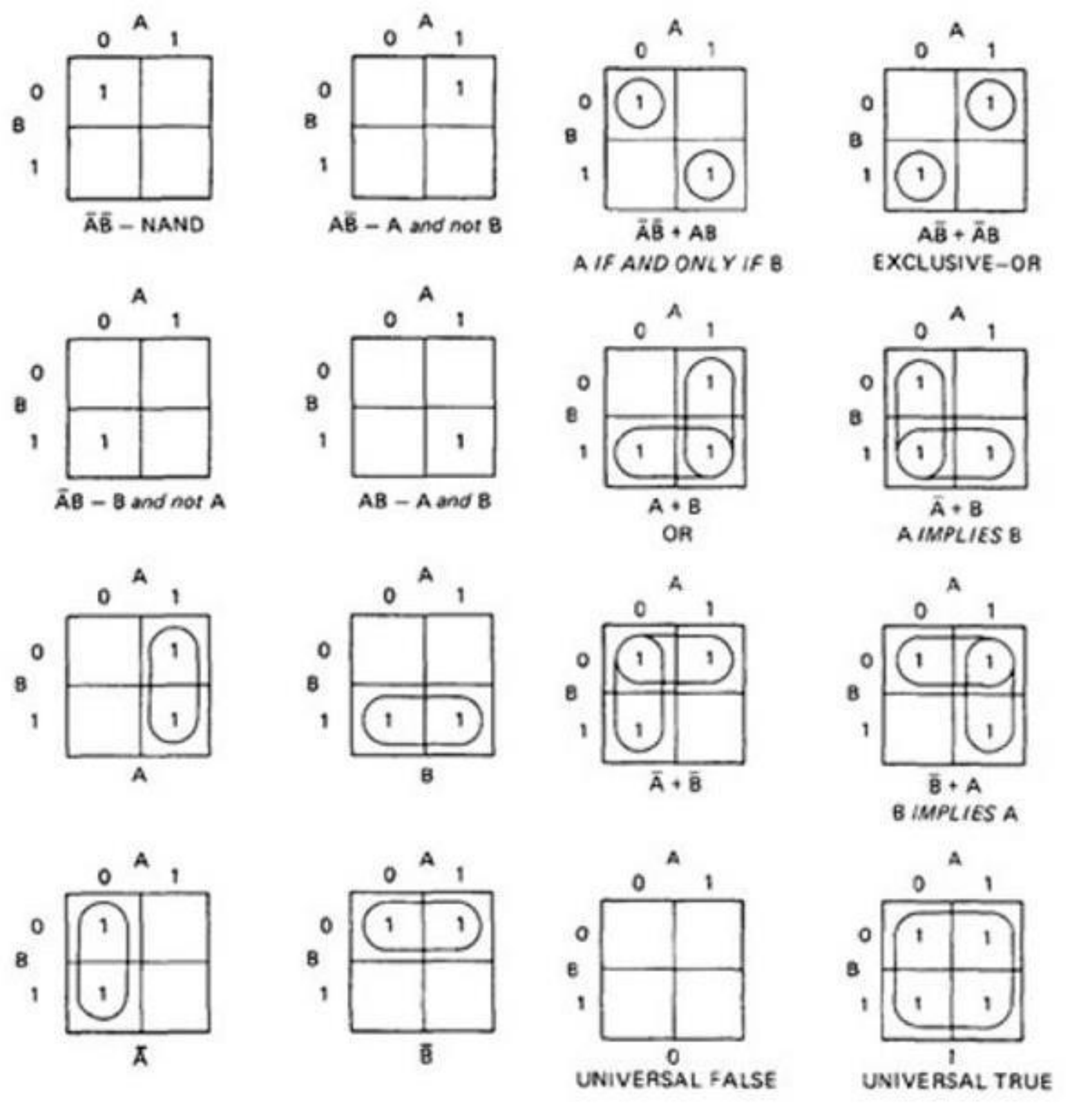
	0	1	
A	0	1	The function is true when A is true

	0	1	
A^1	1	0	The function is true when A is false

	0	1	
1	1	1	The function is always true

- The charts show all possible truth values that the variable A can have.
- A "1" means the variable’s value is "1" or TRUE. A "0" means that the variable's value is 0 or FALSE.
- The entry in the box (0 or 1) specifies whether the function that the chart represents is true or false for that value of the variable.
- We usually do not explicitly put in 0 entries but specify only the conditions under which the function is true

TWO VARIABLES:

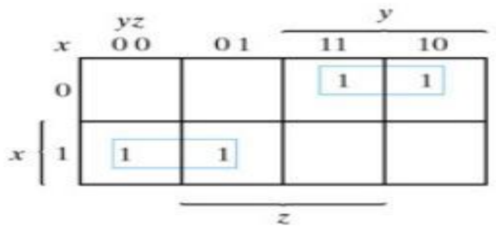


- Each box corresponds to the combination of values of the variables for the row and column of that box.
- A pair may be adjacent either horizontally or vertically but not diagonally.
- Any variable that changes in either the horizontal or vertical direction does not appear in the expression.
- In the fifth chart, the B variable changes from 0 to 1 going down the column, and because the A variable's value for the column is 1, the chart is equivalent to a simple A.

THREE VARIABLES:

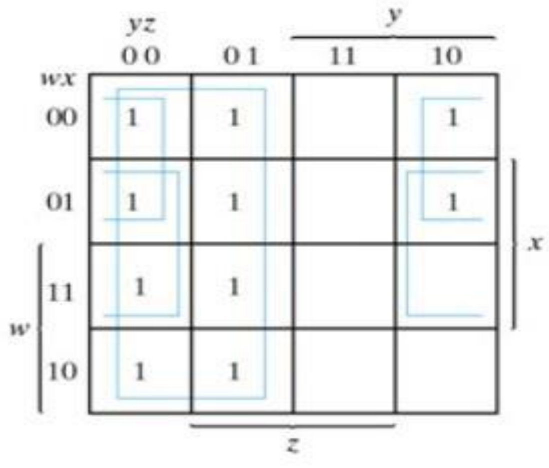
- KV charts for three variables are shown below.
- As before, each box represents an elementary term of three variables with a bar appearing or not appearing according to whether the row-column heading for that box is 0 or 1.
- A three-variable chart can have groupings of 1, 2, 4, and 8 boxes.
- A few examples will illustrate the principles:

Three variable map-example



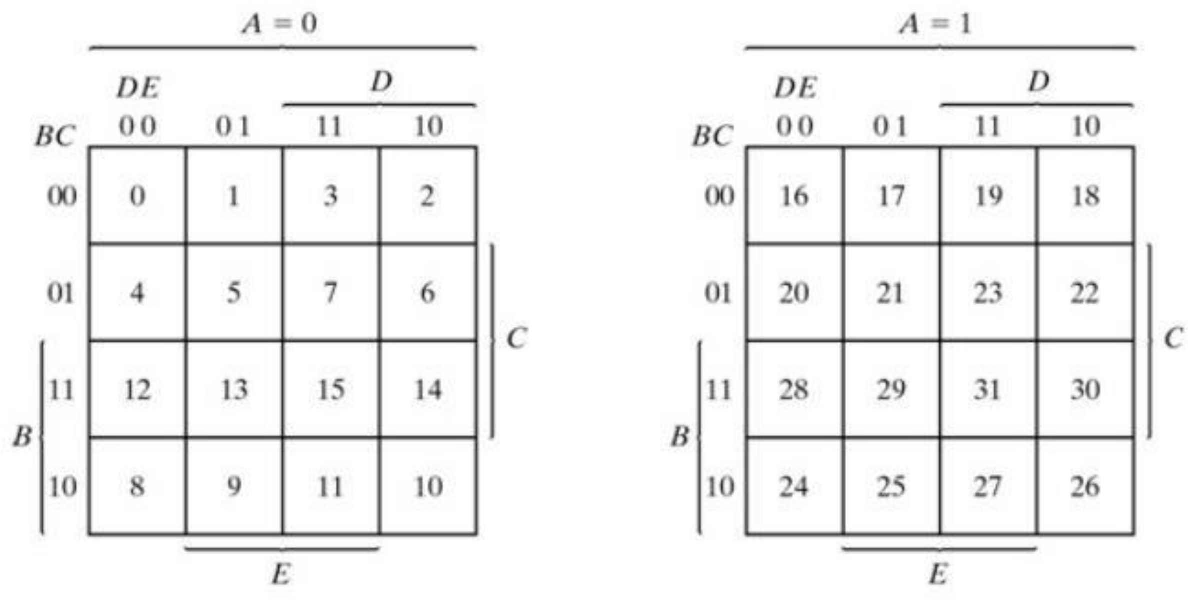
$F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$

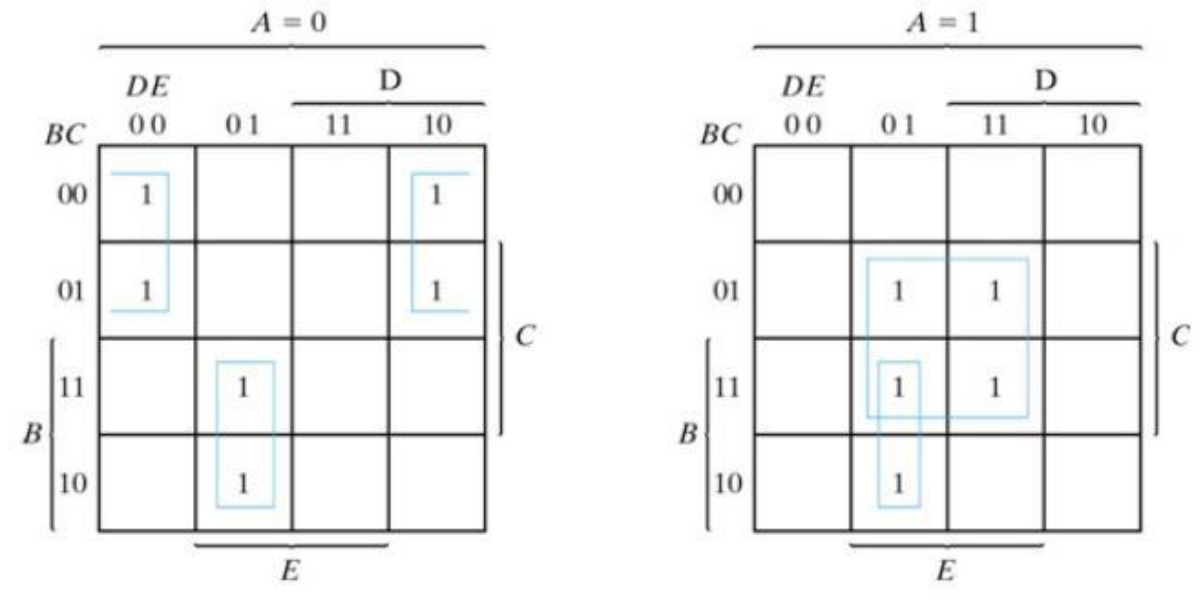
Four variable map-example



$F(w, x, y, z)$
$= \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14) = y' + w'z' + xz'$

Five variable map-example





$F = A'B'E' + BD'E + ACE$