<ch8 toc ch10>

# Chapter 9
# SYNTAX TESTING

## 1. SYNOPSIS

System inputs must be validated. Internal and external inputs conform to formats, which can usually be expressed in **Backus-Naur form—**a specification form that can be mechanically converted into more input-data validation tests than anyone could want to execute.

## 2. WHY, WHAT, AND HOW

### 2.1. Garbage

"Garbage-in equals garbage-out" is one of the worst cop-outs ever invented by the computer industry. We know when to use that one! When our program screws up and people are hurt. An investigation is launched and it's discovered that an operator made a mistake, the wrong tape was mounted, or the source data were inconsistent, or something like that. That's the time to put on the guru's mantle, shake your head, disclaim guilt, and mutter, "What do you expect? Garbage-in equals garbage-out."

Can we say that to the families of the airliner crash victims? Will you offer that excuse for the failure of the intensive care unit's monitoring system? How about a nuclear reactor meltdown, a supertanker run aground, or a war? GIGO is no explanation for anything except our failure to install good data-validation checks, or worse, our failure to test the system's tolerance for bad data. Garbage shouldn't get in—not in the first place or in the last place. Every system must contend with a bewildering array of internal and external garbage, and if you don't think the world is hostile, how do you plan to cope with alpha particles?

### 2.2. Casual and Malicious Users

Systems that interface with the public must be especially robust and consequently must have prolific input-validation checks. It's not that the users of automatic teller machines, say, are willfully hostile, but that there are so many of them—so many of them and so few of us. It's the million-monkey phenomenon: a million monkeys sit at a million typewriters for a million years and eventually one of them will type *Hamlet*. The more users, the less they know, the likelier that eventually, on pure chance, someone will hit every spot at which the system's vulnerable to bad inputs.

There are malicious users in every population—infuriating people who delight in doing strange things to our systems. Years ago they'd pound the sides of vending machines for free sodas. Their sons and daughters invented the "blue box" for getting free telephone calls. Now they're tired of probing the nuances of their video games and they're out to attack computers. They're out to get *you*. Some of them are programmers. They're persistent and systematic. A few hours of attack by one of *them* is worse than years of ordinary use and bugs found by chance. And there are so many of them; so many of them and so few of us.

Then there's crime. It's estimated that computer criminals (using mostly hokey inputs) are raking in hundreds of millions of dollars annually. A criminal can do it with a laptop computer from a telephone booth in Arkansas. Every piece of bad data accepted by a system—every crash-causing input

sequence—is a chink in the system's armor that smart criminals can use to penetrate, corrupt, and eventually suborn the system for their own purposes. And don't think the system's too complicated for them. They have your listings, and your documentation, and the data dictionary, and whatever else they need.* There aren't many of them, but they're smart, motivated, and possibly organized.

---

*Accountants are the most successful embezzlers—why wouldn't the "best" computer criminal be a programmer?

---

## 2.3. Operators

Roger and I were talking about operators and the nasty things they can do, and the scenarios were getting farfetched. Who'd think of mounting a tape with a write-ring installed, writing a few blocks, stopping, opening the transport's door, dismounting the tape reel without unloading the buffers, removing the ring, remounting the tape without telling the system, and then attempting to write a new block? The malice we ascribed to the operators was embarrassing. I said to Roger, the designer most concerned with the impact of operator shenanigans, "What the operators have done to these systems in the past is bad enough—just imagine how they'd act if they knew how we talked about them."

To which he snapped, "If they knew how we talked about them, they'd probably act the way we expect them to!"

I'm not against operators and I don't intend to put them down. They're our final defense against our latent bugs. Too often they manage, by intuition, common sense, and brilliance, to snatch a mere catastrophe from the jaws of annihilation. Operators make mistakes—and when they do, it can be serious. It's right that they probe the system's defenses, catalog its weaknesses and prepare themselves for the eventualities we didn't think of.

## 2.4. The Internal World

Big systems have to contend not only with a hostile external environment but also a hostile internal environment. Malice doesn't play a role here, but oversight, miscommunication, and chance can be just as deadly. Any big system is subdivided into loosely coupled subsystems and consequently, there are many internal interfaces. Each interface presents another opportunity for data corruption and may require explicit internal-data validation. Furthermore, hardware can fail in bizarre ways that will cause it to pump streams of bad data into memory, across channels, and so on. Another piece of software may fail and do the same. And then there're always alpha particles.

## 2.5. What to Do

Input validation is the first line of defense against a hostile world. Good designers design their system so that it just doesn't accept garbage—good testers subject systems to the most creative garbage possible. Input-tolerance testing is usually done as part of system testing, such as in a formal feature test or in a final acceptance test, so it's usually done by independent testers.

This kind of testing and test design is more fun than any other kind I know of: it's great therapy and they pay you for it. My family and pets loved it when I was doing these tests; after I was through kicking and stomping the programmers around, there wasn't a mean bone left in my body.

But to be really diabolical takes organization, structure, discipline, and method. Taking random potshots and waiting for inspirations with which to victimize the programmer won't do the job. Syntax testing is a primary toot of dirty testing, and method beats sadism every time.

## 2.6. Applications and Hidden Languages

Opportunities for applying syntax testing abound in most systems because most systems have **hidden languages**. A **hidden language** is a programming language that hasn't been recognized as such. Remember the Third Law (Chapter 2, Section 3.4.1.): *Code Migrates to Data*. One of the ways this happens is by taking control information and storing it as data (in lists or tables, say) and then interpreting that data as statements in an unrecognized and undeclared, internal, high-level language. Syntax testing is used to validate and break the explicit or (usually) implicit parser of that language. The troubles with these hidden languages are: there's no formal definition of syntax; the syntax itself is often buggy; and parsing is inexorably intertwined with processing. The key to exploiting syntax testing is to learn how to recognize hidden languages. Here are some examples:

**1.** User and operator commands are obvious examples of languages. Don't think that it doesn't pay to use syntax-testing methods because you only have a few commands. I've found it useful for only one or two dozen commands. For mainframe systems there are system operator commands and also a big set of user OS commands. For PC or any other application that can be used interactively, there are application-specific command sets.
**2.** The counterpart to operator and user command languages for batch processing are job control languages: either at the operating system level (e.g., JCL) or application-specific.
**3.** In Chapter 4, Section 5.2, I wrote about transaction-control languages. Reread that section for this application.
**4.** A system-wide interprocess-communication convention has been established. Isn't that a minilanguage? A precompilation preprocessor has been implemented to verify that the convention is followed: use syntax-testing thinking to test it.
**5.** An offline database generator package is used to create the database. It has a lot of fields to look at and many rules to follow—and more syntax to check.
**6.** Any internal format used for interprocess communications that does not consist of simple, fixed fields should be treated to a syntax test—for example, project or application calling sequence conventions, or a macro library.
**7.** Almost any communication protocol has, in part, a command language and/or formats that deserve syntax testing. Even something as simple as using a telephone can be tested by syntax testing methods.
**8.** A complicated application may have several hidden languages: an external language for user commands and an internal language, not apparent to the user, out of which the applications are built. The internal languages could be subtle and difficult to recognize. For example, a language could consist of a pattern of calls to worker subroutines. A deep call tree with a big common subroutine library can be viewed as a **syntax graph** (see below). A tip-off for syntax testing is that there are a few high-level routines with subroutine or function names in the call and a big common subroutine library. When you see that kind of a call tree, think of syntax testing.

I wouldn't use syntax-testing methods against a modern compiler. The way modern compiler construction is automated almost guarantees that syntax testing won't be effective. By extension, if the hidden language is out in the open and implemented as a real language, then syntax testing will probably fail—not because the tests won't be valid but because they won't reveal enough bugs to warrant the effort. Syntax testing is a shotgun method that depends on creating many test cases. Although any one case is unlikely to reveal a bug, the fact that many cases are used and that they are very easy to design makes the method effective. It's almost impossible for the kinds of bugs that syntax testing exposes to

remain by the time (typically, in system testing) that syntax testing is used if it is used against a modern compiler. Also, for most programming languages there's no need to design these tests because you can buy a test oracle for the compiler at a far lower cost than you can design the tests.

## 2.7. The Graph We Cover

Look at the back of almost any Pascal reference manual, and you'll see several pages of graphs such as the one shown in Figure 9. 1. It is a graph because it has nodes joined by links. The nodes shown here are either circles or boxes. The links are arrows as usual. The circles enclose actual characters. The boxes refer to other parts of the syntax graph, which you can think of as subroutine calls. The meanings attached to this graph are slightly different than those we've used before:

1. An arrow or link means "is followed by."
2. A branch point at a node with two or more outlinks means "or."

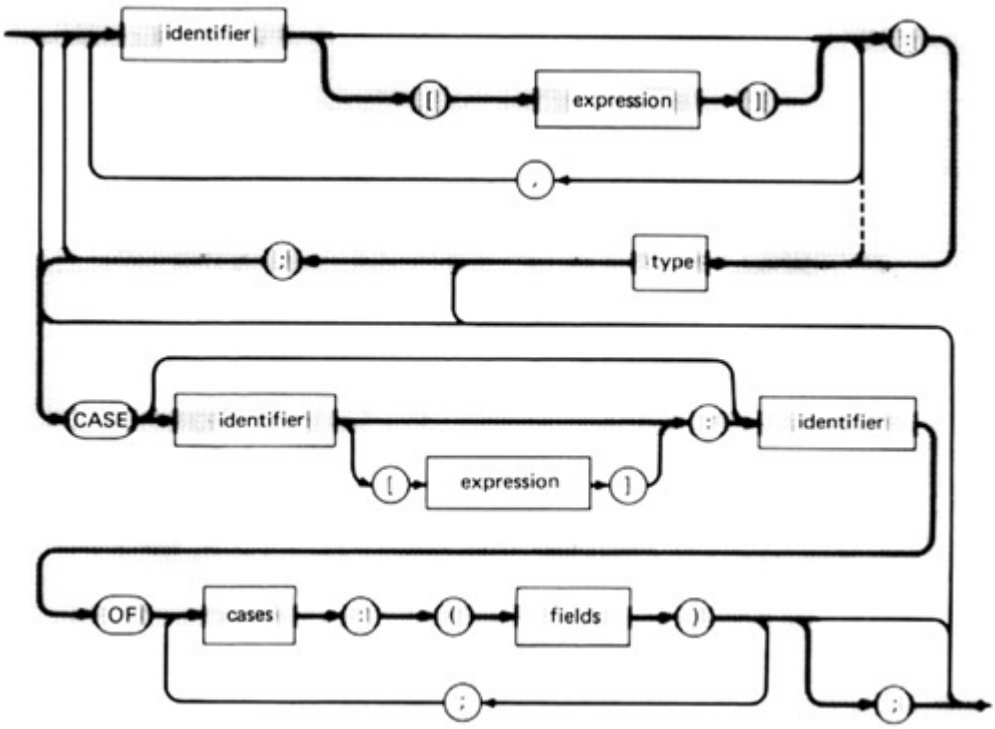


**Figure 9.1.** Pascal Fields Definition Graph. (Courtesy of Microsoft.)

We can interpret one path through this graph to get an example of "*fields*." The marked path is equivalent to:

"A *fields* (can) consist of an *identifier* followed by "[" followed by an *expression* followed by "]:" followed by a *type* ";CASE", another *identifier*, ": ", yet another *identifier*, "OF", *cases*, ":(" *fields*, and ending with ");".

**What do you do when you see a graph? COVER IT!** Do you know how to do syntax testing? Of course you do! Look at Figure 9. 1. You can test the normal cases of this syntax by using a covering set of paths through the graph. For each path, generate a *fields* that corresponds to that path. There are two explicit loops in the graph, one at the top that loops around *identifier* and another at the bottom following the "OF." There is an implicit loop that you must also test: note that *fields* calls itself recursively, I would start with a set of covering paths that didn't hit the recursion loop and only after that was satisfactory, hit the recursive cases.

In syntax testing, we must test the syntax graph with (at least) a covering set of test cases, but we usually go much further and also test with a set of cases that cannot be on the graph—the dirty cases. We generate these by methodically screwing up one circle, box, or link at a time. Look at the comma in the top loop: we can remove it or put something else in its place for two dirty test cases. We can also test a link that doesn't exist, such as following the comma with a *type* as indicated by the dotted arrow.

You also know the strategy for loops. The obvious cases are: not looping, looping once, looping twice, one less than the maximum, the maximum, and one more than the maximum. The not-looping case is often productive, especially if it is a syntactically valid but semantically obscure case. The cases near the maximum are especially productive. You know that there must be a maximum value to any loop and if you can't find what that value is, then you're likely to draw blood by attempting big values.

### 2.8. Overview

Syntax testing consists of the following steps:

1. Identify the target language or format (hidden or explicit).
2. Define the syntax (format) of the language, formally, in a convenient notation such as **Backus-Naur form (BNF).**
3. Test and debug the syntax to assure that it is complete and consistent and that it satisfies the intended semantics.
4. Normal condition testing consists of a covering set of input strings including critical loop values. The difficult part about normal case testing is predicting the outcome and verifying that the processing was correct. That's ordinary functional testing—i.e., semantics. Covering the syntax graph assures that all options have been tested. This is a minimum mandatory requirement with the analogous strengths and weaknesses of branch testing for control flowgraphs. It isn't "complete" syntax testing by any measure.
5. Syntax testing methods pay off best for dirty testing. Test design is a top-down process that consists of methodically identifying which component is to be cruddied-up and how.
6. Much of syntax test design can and should be automated by relatively simple means.
7. Test execution automation is essential for syntax testing because this method produces so many tests.

## 3. A GRAMMAR FOR FORMATS

### 3.1. Objectives

Every input has a syntax. That syntax may be formally specified or undocumented and "just understood," but it does exist. Data validation consists (in part) of checking the input for correct syntax. It's best when the syntax is defined in a formal language—best for the designer and the tester. Whether the designer creates the data-validation software from a formal specification or not is not important to the tester, but the tester needs a formal specification to create useful garbage. That specification is conveniently expressed in Backus-Naur form, which is very similar to regular expressions.

Regular expressions were introduced in Chapter 8 as an algebraic representation of all the paths in a graph. It's usually more convenient to deal with the algebraic version of graphs than with the pictorial version. Get comfortable with going back and forth between algebraic forms and pictorial forms for graphs and with talk about "covering a graph" even if there's no pictorial graph around. This isn't new to you because you worked with paths through an algebraic representation of a graph long before you heard about flowgraphs: what did you mean by "paths through code"?

## 3.2. BNF Notation (BACK59)

### 3.2.1. The Elements

Every input can be considered as if it were a string of characters. The software accepts valid strings and rejects invalid ones. If the software fails on a string, we've really got it. If it accepts an invalid string, then it's guilty of GIGO. There's nothing we can do about syntactically valid strings whose values are valid but wrong—that kind of garbage we have to accept. The syntax definition must be formal, starting with the most elementary parts, the characters themselves. Here's a sample definition:

```
alpha_characters ::= A/B/C/D/E/F/G/H/1/J/K/L/M/N/O/P/Q/
                     R/S/T/U/V/W/X/Y/Z
numerals         ::= 1/2/3/4/5/6/7/8/9
zero             ::= 0
signs            ::= !/#/$/%/&/*/(/)/-/+/=/;/:/"/'/,/./?
space            ::= sp
```

The left-hand side of the definitions is the name given to the collection of objects on the right-hand side. The string "::=" is interpreted as a single symbol that means "is defined as." The slash "/" means "or." We could have used the plus sign for that purpose as with regular expressions but that wouldn't be in keeping with the established conventions for BNF. We are using BNF to define a miniature language. The "::=" is part of the language in which we talk about the minitanguage, called the **metalanguage**. Spaces are always confusing because we can't display them on paper. We use *sp* to mean a space. The actual spaces on this page have no meaning. Similarly, an italicized (or underlined) symbol is used for any other single character that can't conveniently be printed, such as *null (nl), end-of-text (eot), clear-screen, carriage-return (cr), line-feed (lf), tab, shift-up (su), shift-down (sd), index, backspace (bs),* and so on. The underlined space, as in *alpha_characters*, is used as usual in programming languages to connect words that comprise a single object.

### 3.2.2. BNF Operators

The operators are the same as those used in path expressions and regular expressions: "or," concatenate, (which doesn't need a special symbol), "×", and "+". Exponents, such as $A^n$, have the same meaning as before—*n* repetitions of the strings denoted by the letter A. Syntax is defined in BNF as a set of definitions. Each definition may in turn refer to other definitions or to itself in such a way that eventually it gets down to the characters that form the input string. Here's an example:

```
word ::= alpha_character alpha_character / numeral sp numeral
```

I've defined an input string called *word* as a pair of *alpha_characters* or a pair of *numerals* separated by a space. Here are examples of *words* and non*words*, by this definition:

```
words : AB, DE, XY, 3 sp 4, 6 sp 7, 9 sp 9, 1 sp 2
nonwords : AAA, A sp A1, A), 11, 111, WORD, NOT sp WORD, +
```

There are 722 possible *words* in this format and an infinite number of non*words*. If the strings are restricted to four characters, there are more than a million non*words*. The designer wants to detect and accept words and reject non*words*; the tester wants to generate non*words* and force the program to deal with them.

### 3.2.3. Repetitions

As before, *object*$^{1\text{-}3}$ means one to three *objects*, *object** means zero or more repetitions of *object* without limit, and *object*+ means one or more repetitions of object. Neither the star (*) nor the plus (+) can legitimately appear in any syntax because both symbols mean a possibly infinite number of repetitions. That can't be done in finite memory or in the real world. The software must have some means to limit repetitions. It can be done by an explicit test associated with every + or * operator, in which case you should replace the operator with a number. Another way to limit the repetitions is by placing a global limit on the length of any string. The limit then applies to all commands and it may be difficult to predict what the actual limit is for any specific command. You test this kind of limit by maximum-length strings. Yet another way to implement limits is to limit a common resource such as stack or array size. Again, the limits for a specific command may be unpredictable because it is a global limit rather than a format-specific limit. The way to test this situation is with many repetitions of short strings. For example, using as many as possible minimum-length *identifiers* and not including *expression* in the first loop of Figure 9.1: "ID1, ID2, ID3, ID4,.... ID999: *type*" will do the job.

One of the signs of weak software is the ease with which you can destroy it by overloading its repetition-limiting mechanisms. If the mechanism doesn't exist, you can probably scribble all over the stack or code—crash-crash, tinkle-tinkle, goody-goody.

### 3.2.4. Examples

This is an example and not a real definition of a telephone number:

```
special_digit    ::= 1/2/5
zero             ::= 0
other_digit      ::= 3/4/6/7/8/9
ordinary_digit   ::= special digit / zero / other_digit
exchange_part    ::= other_digit2 ordinary_digit
number_part      ::= ordinary_digit4
phone-number     ::= exchange_part number-part
```

According to this definition, the following are *phone-numbers*,

```
3469900, 9904567, 3300000
```

and these are not:

```
5551212, 5510000, 123, 8, ABCDEFG, 572-5580, 886-0144.
```

Another example:

```
operator_command ::= mnemonic field_unit[1-8] +
```

An *operator_command* consists of a *mnemonic* followed by one to eight *field_units* and a plus sign.

```
field_unit   ::= field delimiter
mnemonic     ::= first_part second_part
delimiter    ::= sp / , / . / $ / ×sp[1-42]
field        ::= numeral / alpha / mixed / control
first-part   ::= a_vowel a_consonant
second_part  ::= b_consonant alpha
a_vowel      ::= A/E/I/O/U
a_consonant  ::= B/D/F/G/H/J/K/L/M/N/P/Q/R/S/T/V/X/Z
b_consonant  ::= B/G/X/Y/Z/W/M/R/C
```

```
alpha         ::= a-vowel / a consonant b_consonant
numeral       ::= 1/2/3/4/5/6/7/8/9/0
control       ::= $/×/%/sp/@
mixed         ::= control alpha control /
                  control numeral control /
                  control control control
```

Here are some valid *operator_commands*:

```
ABXW A. B. C. 7. +
UTMA W sp sp sp sp +
```

While the following are not *operator-commands*:

```
ABC sp +
A sp BCDEFGHIJKLMNOPQR sp47 +
```

The telephone number example and the operator command example are different. The telephone number started with recognizable symbols and constructed the more complicated components from them—a bottom-up definition. The command example started at the top and worked down to the real characters—a top-down definition. These two ways of defining things are equivalent—it's only a matter of the order in which the definition lines are printed. The top-down order is generally more useful and it's the usual form for language design. Looking at the definition from the top down leads you to some tests and looking from the bottom up can lead to different tests.

As a final notational convenience, it's sometimes useful to enclose an expression in parentheses to reduce the number of steps in the definition. For example, the definition step for *field_unit* could have been simplified as follows:

```
operator_command ::= mnemonic (field delimiter)1-8 +
```

This is fine if the syntax doesn't use parentheses that can confuse you in the definitions; otherwise use some other bracket symbols such as < and >. BNF notation can also be expanded to define optional fields, conditional fields, and the like. In most realistic formats of any complexity, you won't be able to get everything expressed in this notation—nor is it essential that you do so; additional narrative descriptions may be needed.

# 4. TEST CASE GENERATION

## 4.1. Generators, Recognizers, and Approach

A data-validation routine is designed to recognize strings that have been explicitly or implicitly defined in accordance with an input syntax. It either accepts the string, because it is recognized as valid, or rejects it and takes appropriate action. The routine is said to be a **string recognizer**. Conversely, the tester attempts to generate strings and is said to be a **string generator**. There are three possible kinds of incorrect actions:

1. The recognizer does not recognize a good string.
2. It accepts a bad string.
3. It may accept or reject a good string or a bad string, but in so doing, it fails.

Even small specifications lead to many good strings and far more bad strings. There is neither time nor

need to test them all. String errors can be categorized as follows:

1. *High-Level Syntax Errors*—The strings have violations at the topmost level in a top-down BNF syntax specification.
2. *Intermediate-Level Syntax Errors*—Syntax errors at any level other than the top or bottom.
3. *Field-Syntax Errors*—Syntax errors associated with an individual field, where a **field** is defined as a string of characters that has no subsidiary syntax specification other than the identification of characters that compose it. A field is the lowest level at which it is productive to think in terms of syntax testing.
4. *Delimiter Errors*—Violation of the rules governing the placement and the type of characters that must appear as separators between fields.
5. *Syntax—Value Errors*—When the syntax of one field depends on values of other fields, there is a possibility of an interaction between a field-value error and a syntax error—for example, when the con-tents of a control field dictate the syntax of subsequent fields. This is a messy business that permits no reasonable approach. It needs syntax testing combined with domain testing, but it's better to redesign the syntax.
6. *State-Dependency Errors*—The permissible syntax and/or field values is conditional on the state of the system or the routine. A command used for start-up, say, may not be allowed when the system is running. If state behavior is extensive, consider state testing (Chapter 11).

Errors in the values of the fields or the relation between field values are domain errors and should be tested accordingly.

## 4.2. Test Case Design

### 4.2.1. Strategy

The strategy is to create one error at a time, while keeping all other components of the input string correct; that is, in the absence of the single error, the string would have been accepted. Once a complete set of tests has been specified for single errors, do the same for double errors and then triple errors. However, if there are of the order of $N$ single-error cases, there will be of the order of $N^2$ double-error and $N^3$ triple-error cases. Once past the single errors, it takes a lot of judicious pruning to keep the number of tests reasonable. This is almost impossible to do without looking at the implementation details.

### 4.2.2. Top, Intermediate, and Field-Level Syntax Errors

Say that the topmost syntax level is defined as:

```
item ::= atype / btype / ctype / dtype etype
```

Here are some obvious test cases:

1. *Do It Wrong*—Use an element that is correct at some other lower syntax level, but not at this level.
2. *Use a Wrong Combination.* The last element is a combination of two other elements in a specified order. Mess up the order and combine the wrong things:

```
dtype atype / btype etype / etype dtype / etype etype / dtype dtype
```

3. *Don't Do Enough*—For example,

```
dtype / etype
```

4. *Don't Do Nothing*. No input, just the end-of-command signal or carriage return. Amazing how many bugs you catch this way.
5. *Do Too Much*—For example:

```
atype btype ctype dtype etype / atype atype atype /
dtype etype atype / dtype etype etype / dtype etype[128]
```

Focus on one level at a time and keep the level above and below as correct as you can. It may help to draw a definition graph; we'll use the telephone number example (see Figure 9.2).

TOP LEVEL

1. Do nothing.
2. An *exchange_part* by itself.
3. A *number_part* by itself.
4. Two *exchange_parts*.
5. Two *number_parts.*
6. An *exchange_part* and *two number_parts*.
7. Two *exchange_parts* and one *number_part*.

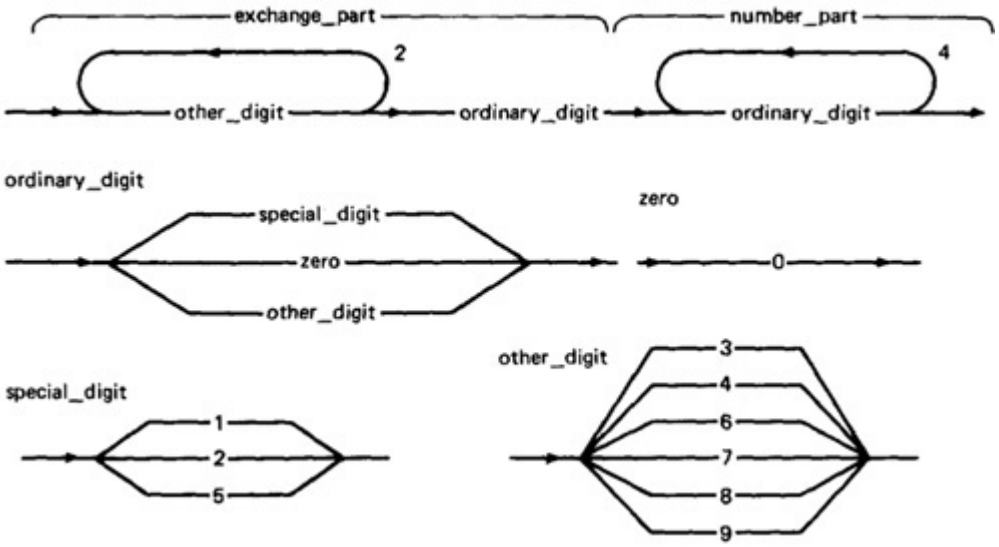


**Figure 9.2.** Phone_number Definition Graph.

NEXT LEVEL

8. Bad *exchange_part*.
   - **8.1—do** nothing—covered by test 3 above.
   - **8.2—no** *other_digit*.
   - **8.3—two** *special_ digits*.
   - **8.3—three** *special_digits*.
   - **8.4—et** cetera.
9. Bad *number_part*.
   - **9.1—not** enough *digits*.
   - **9.2—too** many *digits*.
   - **9.3—et** cetera.

THIRD LEVEL

**10.** Bad *ordinary_digit*.
- **10.1** —not a digit—use an alphabetic.
- **10.2** —not a digit—use a control character.
- **10.3** —not a digit—use a delimiter.
- **10.4** —not a digit—leave it out.
- **10.5** —et cetera.

FOURTH LEVEL

**11.** Bad *other_digit*—as with bad *ordinary_digit*.
**12.** Bad *special_digit*—as with bad *ordinary_digit*.
**13.** Et cetera.

Check the levels above and below as you generate cases. Not everything generated by this procedure produces bad cases, and the procedure may lead to the same test case by two different paths. The corruption of one element could lead to a correct but different string. Such tests are useful because logic errors in the string recognizer might miscategorize the string. Similarly, if a test case (either good or bad) can be generated via two different paths, it is an especially good case because there is a potential for confusion in the routine. I like test cases that are difficult to design and difficult to recognize as either good or bad because if I'm confused, it's likely that the designer will also be confused. It's not that designers are dumb and testers smart, but designers have much more to do than testers. To design and execute syntax-validation tests takes 5% to 10% of the effort needed to design, code, test, validate, and integrate a syntax-validation routine. The designer has 10 to 20 times as much work to do as does the tester. Given equal competence, if the tester gets confused with comparatively little to do, it's likely that the overloaded designer will be more confused by the same case.

Now look at Figure 9.3. I generated this syntax graph from Figure 9.2 by inserting the subsidiary definitions and simplifying by using regular expression rules. The result is much simpler. It can obviously be covered by one test for the normal path and there are far fewer dirty tests:

**1.** Start with a *special_digit.*
**2.** Start with a *zero.*
**3.** Only one *other_digit* before a *zero.*
**4.** Only one *other_digit* before a *special-digit.*
**5.** Not enough digits.
**6.** Too many digits.
**7.** Selected nondigits.

You could get lavish and try starting with two *zeros*, two *special_digits, zero* followed by *special_digit*, and *special_digit* followed by *zero*, thereby hitting all the double-error cases of interest; but there's not much more you can do to mess up this simplified graph. Should you do it? No! The implementation will tend to follow the definition and so will the bugs.

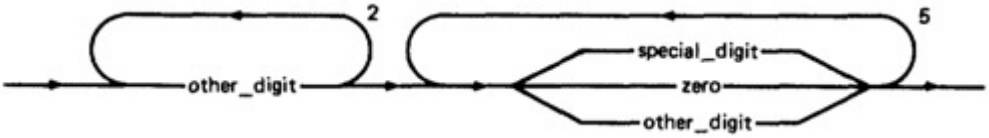


**Figure 9.3.** Phone-number Graph Simplified.

Therefore, there's a richer possibility for variations of bad strings to which the simplified version is not vulnerable. Don't expect to find opportunities for such simplifications in the syntax of mature programming languages—the language designers usually do as much simplification as makes sense before the syntax is released. For formats, operator commands, and hidden languages, there are many such opportunities. The lesson to be learned from this is that you should always simplify the syntax graph if you can. The implementation will be simpler, it will take fewer tests to cover the normal cases, and there will be fewer meaningful dirty tests.

#### 4.2.3. Delimiter Errors

**Delimiters** are characters or strings placed between two fields to denote where one ends and the other begins. Delimiter problems are an excellent source of test cases. Therefore, it pays to identify the delimiters and the rules governing their syntax.

1. *Missing Delimiter*—This kind of error causes adjacent fields to merge. This may result in a different, but valid, field or may be covered by another kind of syntax error.
2. *Wrong Delimiter*—It's nice when several different delimiters are used and there are rules that specify which can be used where. Mix them up and use them in the wrong places.
3. *Not a Delimiter*—There are some characters or strings that are not delimiters but could be put into that position. Note the possibility of changing adjacent field types as a result.
4. *Too Many Delimiters*—The number of delimiters appearing at a field boundary may be variable. This is typical for spaces, which can serve as delimiters. If the number of delimiters is specified as 1 to $N$, it pays to try 0, 1, 2, $N$ - 1, $N$, $N$ + 1, and also an absurd number of delimiters, such as 127, 128, 255, 256, 1024, and so on.
5. *Paired Delimiters*—These delimiters represent another juicy source of test cases. Parentheses are the archetypal paired delimiters. There could be several kinds of paired delimiters within a syntax. If paired delimiters can nest, as in "( ()() )", there are a whole set of new evils to perpetrate. For example, "BEGIN...BEGIN...END", "BEGIN...END...END". Nested paired delimiters provide opportunities for matching ambiguities. For example, "((()(()))" has a matching ambiguity and it's not clear where the missing parenthesis belongs.
6. *Tolerant Delimiters*—The delimiter may be optional or several alternate formats may be acceptable. In communications systems, for example, the start of message is defined as ZCZC, but many systems allow any one of the four characters to be corrupted. Therefore, #CZC, Z#ZC, ZC#C, and ZCZ# (where "#" denotes any character) are all acceptable; there are many nice confusing ways to put in a bad character here:
    a. A blank.
    b. Z or C in the wrong place—CCZC, ZZZC, ZCCC, ZCZZ (catches them every time!).
    c. Something off-the-wall—especially important control characters in some other context.

Tolerance is most often provided for delimiters but can also be provided for individual fields and higher levels of syntax. It's a sword of many edges—more than two for sure—all of them dangerous. Syntax tolerance is provided for user and operator convenience and in the interest of making the system humane and robust. But it also makes the format-validation design job and testing format-validation designs more complicated. Format tolerance is sophisticated and takes sophisticated designs to implement and, consequently, many more and more complicated tests to validate. If you can't do the whole job from design to thorough validation, there's no point in providing the tolerance. Most users and operators prefer a solid system with rigid syntax rules to a system with tolerant rules that don't always work.

#### 4.2.4. Field-Value Errors

Field-value errors are clearly a domain-testing issue, and domain testing is where it's at. Whether you

choose to implement field-value errors in the context of syntax testing or the other way around (i.e., syntax testing under domain testing) or whether you choose to implement the two methods as separate test suites depends on which aspect dominates. Syntax-testing methods will usually wear out more quickly than will domain testing. For that reason, it pays to separate domain and syntax tests into different suites. You may not be able to separate the two test types because of (unfortunately) context-dependent syntax—either field values whose domain depends on syntax or syntax that depends on field values (ugh!). Here's a reminder of what to look for: boundary values and near-boundary values, excluded values, binary values for integers, and values vulnerable to semantic type changes and representation changes.

#### 4.2.5. Context-Dependent Syntax Errors

Components of the syntax may be interrelated and may be related by field values of other fields. The first field could be a code that specifies the syntax of subsequent fields. As an example:

```
command       ::= pilot_field syntax_option
pilot_field   ::= 1/2/3/4/5/6/7/8/9
syntax_option ::= option1 / option2 / option3 / . . .
```

The specification further states that option!] must be preceded by "1" as the value of the *pilot-field*. Actually, it would have been better had the specification be written as:

```
command ::= 1 option1 / 2 option2 / 3 option3 / . . .
```

but that's not always easy to do. The test cases to use are clearly invalid combinations of syntactically valid field values and syntactically valid options. If you can rewrite the specification, as in the above example, to avoid such field-value dependencies, then it's better you do so; but if so doing means vast increases in formality, which could be handled more clearly with a side-bar notation that specifies the relation, then it's better to stick to the original form of the specification. The objective is to make things as clear as possible to the designer and to yourself, and excessive formality can destroy clarity just as easily as modest formality can enhance it.

#### 4.2.6. State-Dependency Errors

The string or field value that may be acceptable at one instant may not be acceptable at the next because validity depends on the transaction's or the system's state. As an example, say that the operator's command-input protocol requires confirmation of all commands. After every command the system expects either an acknowledgment or a cancellation, but not another command. A valid command at that point should be rejected, even though it is completely correct in all other respects. As another example, the system may permit, as part of a start-up procedure, commands that are forbidden after the system has been started, or it may forbid otherwise normal commands during a start-up procedure. A classical example occurs in communications systems. The start of message sequence ZCZC is allowed tolerance (see page 301) when it occurs at the beginning of a message. However, because the system has to handle Polish language words such as: "zczalny", "jeszcze", "deszcz", and "zczotka" (BEIZ79), the rules state that any subsequent start-of-message sequence that occurs prior to a correct end-of-message sequence (NNNN) must be intolerant; the format is changed at that point and only an exact "ZCZC" will be accepted.

I divide state-dependency errors into "simple" and "complicated." The simple ones are those that can be described by at most two states. All the rest are complicated and are best handled by the methods of Chapter 11. The simple ones take two format or two field-value specifications—and require at worst

double the work.

## 4.3. Sources of Syntax

### 4.3.1. General

Where do you get the syntax? Here's another paradox for you. If the syntax is served up in a nice, neat, package, then syntax-testing methods probably won't be effective and if syntax testing is effective, you'll have to dig out and formally define the syntax yourself. Where do you get the syntax?

Ideally, it comes to you previously defined, formally defined, in BNF or an equivalent, equally convenient notation.* That's the case for common programming languages, command languages written by and for programmers, and languages and formats defined by a formal standard.

---

*I'll use the term " BNF specification" in the sequel to mean actual BNF or specifications in any other metalanguage that can be used for the same purpose. Many other metalanguages have been developed, but BNF is the most popular and it has the advantage of being almost the same as regular expressions.

---

### 4.3.2. Designer-Tester Cooperation and Design Specifications

If there is no BNF specification, I try to get the designers to create one—at least the first version of one. Realistically, though, if a BNF specification does not exist, the designers will have to create a document that can be easily converted into one or what is she designing to? If you get the designer to create the first version of the BNF specification, you may find that it is neither consistent nor complete. Test design begins with requests for clarification of that preliminary specification. Many serious bugs can be avoided this way. Do it yourself if you can't get the designers to create the first version of the BNF specification. It doesn't really matter whether it's complete or correct, as long as it's down on paper and formal. Present your specification version to the designers and say that tests will be defined accordingly. There may be objections, but the result should be a reasonably correct version in short order.

Using a BNF specification is the easiest way to design format-validation test cases. It's also the easiest way for designers to organize their work, but sadly they don't always realize that. You can't begin to design tests unless you agree on what is right or wrong. If you try to design tests without a formal specification, you'll find that you're throwing cases out, both good and bad, as the designers change the rules in response to the cases you show them. If you can't get agreement on syntax early in the project, put off syntax test design and concentrate on some other area. Alternatively, and more productively, participate in the design under the guise of getting a specification tied down. You'll prevent lots of bugs that way.

It can boomerang, though. I pushed for a BNF specification of operator commands on one project. The commands had been adapted from a previous system in the same family whose formats were clearly specified but not in BNF. This designer fell in love with BNF and created a monster that was more complicated than a combination of Ada and COBOL—and mostly wrong. To make matters worse, his first version of the operator's manual was written in top-down BNF, so operators had to plow through several levels of abstract syntax to determine which keys to hit. Good human engineering will dictate simple, clean, easy-to-understand syntax for user and operator interfaces. Similarly, internal formats for interprocess communications should also be simple. There's usually a topmost syntax level, several field options, and a few subfields. Recursive definitions are rare (or should be). We do find useful recursive definitions in operating system command languages or data query languages for things such as sorting,

data object specifications, and searching; but in general, recursive definitions are rare and more likely to be a syntax-specification error than a real requirement. Be suspicious if the syntax is so complicated that it looks like a new programming language. That's not a reasonable thing to expect users or operator to employ.

#### 4.3.3. Manuals as Sources

Manuals, such as instruction manuals, reference manuals, and operator manuals are the obvious place to start for command languages if there isn't a formal syntax document and you can't get designers to do the job for you. The syntax in manuals may be fairly close to a formal syntax definition. Manuals are good sources because more often than not, we're dealing with a maintenance situation, rehosting, or a rewrite of an old application. But manuals can be mushy because the manual writer tries to convey complicated syntactic issues in a language that is "easy for those dumb users and operators."*

---

*Another war story about my favorite bad software vendor, Coddler Incorporated—a pseudonym invented to protect me from lawsuits. They had a word processing programming language that was putatively designed for use by word processing operators. The language's syntax description in the manual was sketchy and wrong. It's the only programming language that ever defeated me because after weeks of trying I still couldn't write more than 20 statements without a syntax error. When I asked them for complete documentation of the language's syntax, I was told that, as a mere user, I wasn't entitled to such "proprietary" information. We dragged the syntax out of that evil box experimentally and discovered so many context and state dependencies that it was clear to us that they had broken new grounds in language misdesign. What was even worse, when they confronted you with a syntax error, it was presented by reference to the tokenized version of the source after compilation rather than to the source—you guessed it, the token table was also "proprietary." We dragged that out experimentally also and discovered even more state dependencies. Syntax testing that garbage dump was like using hydrogen bombs against a house of cards. The ironic thing about this experience was that we were trying to use this "language" and its "processor" to automate the design of syntax tests for the system we were testing.

---

#### 4.3.4. Help Screens and Systems

Putting user information such as command syntax into HELP systems and on-line tutorial is becoming more commonplace, especially for PC software because it's cheaper to install a few hundred K of HELP material on a floppy than it is to print a few hundred pages of instruction manual. You may find the undocumented syntax on these screens. If you have both manuals and help systems, compare them and find out which one is correct.

#### Data Dictionary and Other Design Documents

For internal hidden languages, your most likely source of syntax is the data dictionary and other design documents. Also look at internal interface standards, style manuals, and design practice memos. Common subroutine and macro library documents are also good sources. Obviously you can't expect designers to hand you the syntax of a language whose existence they don't even recognize.

#### 4.3.6. Prototypes

If there's a prototype, then it's likely to embody much of the user interface and command language syntax you need. This source will become more useful in the future as prototyping gains popularity. But remember that a prototype doesn't really have to work, so what you get could be incomplete or wrong.

#### 4.3.7. Programmer Interviews

The second most expensive way to get user and operator command syntax is to drag the information out of the implementing programmer's head by interviews. I would do it only after I had exhausted all other sources. If you're forced to do this as your only source, then syntax testing may be pointless because a low-level programmer is making user interface or system architecture decisions and the project's probably aground on the reef—it just hasn't sunk yet. Syntax testing is then just a cruel way to demonstrate that fact—pitiful.

#### 4.3.8. Experimental

The most expensive possible way to get the syntax is by experimenting with the running program. Think back to the times you've had to use a new system without an instruction manual and of how difficult it was to work out even a few simple commands—now think of how much work that can be for an entire set of commands; but for dirty old code, it's sometimes the only way. You got it. Take what you know of the syntax and express it in BNF. Apply syntax testing to that trial syntax and see what gets accepted, what gets rejected, and what causes crashes and data loss. You'll have a few surprises that will cause you to change your syntax graph. Change it and start over again until either the money runs out or the program's been scrapped. Looking at the code may help, but it often doesn't because, as often as not, parsing, format validation, domain validation, and processing are hopelessly intertwined and splattered across many components and many levels. It's usually pretty tender software. If you have to live with it, think in terms of putting a proper format validation front end on such junk (see Section 6 below) and avoid the testing altogether.

### 4.4. Ambiguities and Contradictions

Unless it's the syntax of a programming language or a communication format or it's derived from a previous system or from some other source that's been in use for a long time, it's unlikely that the syntax of the formats you're testing will be correct the first time you test it. There will be valid cases that are rejected and other cases, valid or otherwise, for which the action is unpredictable. I mean fundamental errors in the syntax itself and not in the routines that analyze the format. If you have to create the format syntax in order to design tests, you are in danger of creating the format you want rather than the one that's being implemented. That's not necessarily bad if what you want is a simpler, more reliable, and easier format to use, implement, and test.

Ambiguities are easy to spot—there's a dangling branch on the definition tree. That is, something appears on the right-hand side of a definition, but there's no definition with that term on the left-hand side. An obvious contradiction occurs when there are two or more definitions for the same term. As soon as you permit recursive definitions, state dependencies, and context-dependent syntax, the game's over for easy ambiguity and contradiction spotting—in fact, the problem's known to be unsolvable. Approaches to detecting ambiguities and contradictions in the general case is a language-validation problem and beyond the scope of this book by a wide margin. I'm assuming that we're dealing only with the simpler and more obvious ambiguities and contradictions that become apparent when the format is set down formally (e.g., written in BNF) for the first time.

The point about syntactic ambiguities and contradictions (as I've said several times before) is that although a specification can have them, a program, because it is deterministic, is always unambiguous and consistent. Therefore, without looking at the code, even before the code's been designed, you know that there *must* be bugs in the implementation. Take advantage of every ambiguity and contradiction you detect in the format to push the format's design into something that has fewer exception conditions, fewer state dependencies, fewer field correlations, and fewer variations. Keep in close contact with the format's designer, who is often also the designer of the format-analysis routine. Maintain a constant pressure of weird cases, interactions, and combinations. Whenever you see an opportunity to simplify

the format, communicate that observation to the format's designer: he'll have less to design and code, you'll have less to test, and the user will thank you both for a better system. It's true that flaws in the syntax may require a more elaborate syntax and a more complicated implementation. However, my experience has been that the number of instances in which the syntax can be simplified outnumber by about 10 to 1 the instances in which it's necessary to complicate it.

### 4.5. Where Did the Good Guys Go?

Syntax test design is like a lot of other things that are hard to stop once you've started. A little practice with this technique and you find that the most innocuous format leads to hundreds of tests; but there are dangers to this kind of test design.

**1.** *It's Easy to Forget the Normal Cases*—I've done it often. You get so entangled in creative garbage that you forget that the system must also be subjected to good inputs. I've made it a practice to check every test area explicitly for the normal case. Covering the syntax definition graph does it.
**2.** *Don't Go Overboard with Combinations*—It takes iron nerves to do this. You've done all the single-error cases, and in your mind you know exactly how to create the double—and higher-error cases. And there are so many of them that you can create an impressive mound of test cases in short order. "How can the test miss anything if I've tried 1000 input format errors?," you think. Remind yourself that any one strategy is inherently limited to discovering certain types of bugs. Remind yourself that those $N^2$ double-error cases and $N^3$ triple-error cases may be no more effective than trying every value from 1 to 1023 in testing a loop. Don't let the test become top-heavy with syntax tests at the expense of everything else just because syntax tests are so easy to design.
**3.** *Don't Ignore Structure*—Just because you can design thousands of test cases without looking at the code that handles those cases doesn't mean you should do it that way. Knowing the program's design may help you eliminate cases wholesale without sacrificing the integrity and thoroughness of the test. As an example, say that operator-command keywords are validated by a general-purpose preprocessor routine. The rest of the input character string is passed to the appropriate handler for that operator command only after the keyword has been validated. There would be no point to designing test cases that deal with the interaction of keyword errors, the delimiter between the keyword and the first field, and format errors in the first field. You don't have to know a whole lot about the implementation. Often, just knowing what parts of the format are handled by which routines is enough to avoid designing a lot of impressive but useless error combinations that won't prove a thing. The bug that could creep across that kind of interface would be so exotic that you would have to design it. If it takes several hours of work to postulate and "design" a bug that a test case is supposed to catch, you can safely consider that test case as too improbable to worry about—certainly in the context of syntax testing.
**4.** *There's More than One Kind of Test*—Did you forget that you designed path tests and domain tests—that there are state tests to design (Chapter 11), data-flow tests (Chapter 5), or logic-based tests (Chapter 10)? Each model of the system's behavior leads to tests designed from a different point of view, but many of these tests overlap. Although redundant tests are harmless, they cost money and little is learned from them.
**5.** *Don't Make More of the Syntax Than There Is*—You can increase or decrease the scope of the syntax by falsely making it more or less tolerant than it really is. This may lead to the false classification of some good input strings as bad and vice versa—not a terrible problem, because if there is confusion in your mind, there may be confusion in the designer's mind. At worst, you'll have to reclassify the outcome of some cases from "accept" to "reject," or vice versa.
**6.** *Don't Forget the Pesticide Paradox*—Syntax tests wear out fast. Programmers who don't change their design style after being mauled by syntax testing can probably be thrashed by any

testing technique, now and forever. However they do it, by elegant methods or by brute force, good programmers will eventually become immune to syntax testing.

# 5. IMPLEMENTATION AND APPLICATION

## 5.1. Execution Automation

### 5.1.1. General

Syntax testing, more than any other technique I know, forces us into test execution automation because it's so easy to design so many tests (even by hand) and because design automation is also easy. Syntax testing is a shotgun method which—like all shotgun methods—is effective only if there are a lot of pellets in your cartridge. How many ducks will you bring down if you have to throw the pellets up one at a time?

An automation platform is a prerequisite to execution automation. The typical dedicated (dumb) terminal is next to useless. Today, the box of choice is a PC with a hard disc and general-purpose terminal emulator software such as CROSSTALK MK-4 (CROS89). If you've still got 37xx or VTxxx terminals, or teleprinters, or even cardwallopers around and someone wants to foist them off on you as test platforms, resist—violently. Dumb terminals are hardly better than paper tape and teleprinters.

### 5.1.2. Manual Execution

Manual execution? Don't! Even primitive automation methods such as putting test cases on paper tape (see the first edition) was better than doing it manually. I found that the only way it could be done by hand was to use three persons, as in the following scenario. If that doesn't convince you to automate, then you're into compulsive masochism.

Use three persons to do it. The one at the terminal should be the most fumble-fingered person in the test group. The one with the test sheet should be almost illiterate. The illiterate calls out one character at a time, using her fingers to point to it, and moving her lips as she reads. The fumble-fingered typist scans the keyboard (it helps if he's very myopic) and finally finds it.

"A" the illiterate calls out.

"A" the typist responds when he's got his finger on the key. He presses it and snatches it away in fear that it will bite him.

"Plus" the reader shouts.

"No, dammit!" the third person, the referee, interrupts (the only one in the group who acts as if she had brains).

The idiot typist looks for the "DAMMIT" key.* . . .

---

*Don't snicker. Ask your friends who work in PC software customer service how many times they've had inquiries from panicked novices who couldn't find the "ANYKEY" key—as in ". . . then hit any key."

---

So I'm exaggerating: but it's very hard to get intelligent humans to do stupid things with consistency. Syntax testing is dominated by stupid input errors that you've carefully designed.

#### 5.1.3. Capture/Replay

See Chapter 13 for a more detailed discussion of capture/replay systems. **A capture/replay** system captures your keystrokes and stuff sent to the screen and stores them for later execution. However you've designed your syntax tests, execute them the first time through a capture/replay system if that's the only kind of execution automation you can manage. These systems (at least the acceptable ones) have a built-in editor or can pass the test data to a word processor for editing. That way, even if your first execution is faulty, you'll be able to correct it.

#### 5.1.4. Drivers

Build or buy a **driver—**a program that automatically sequences through a set of test cases usually stored as data. Don't build the bad strings (especially) as code in an ordinary programming language because you'll be going down a diverging infinite sequence of test testing.

#### 5.1.5. Scripting Languages

A **scripting language** is a language used to write test scripts. CASL (CROS89, FURG89) is nice scripting language because it can be used to emulate any interface, work from strings stored as data, provide smart comparisons for test outcome validation, editing, and capture/replay.

### 5.2. Design Automation

#### 5.2.1. General

Syntax testing is a good place to begin a test design automation effort because it's so easy and has such a high, immediate payoff. It's about the only test design automation area in which you can count on a payback the first time out.

#### 5.2.2. Primitive Methods

You can do design automation with a word processor. If you don't have that, will you settle for a copying machine and a bottle of white-out? Design a covering set of correct input strings. If you want to, because you have to produce paper documentation for every test case, bracket your test strings with control sequences such as "$$$XXX" so that you'll be able to extract them later on. Let's say you're doing operator commands. Pick any command and reproduce the test sheet as often as you need to cover all the bad cases for that command. Then, using the word processor's search-and-replace feature, replace the correct substring with the chosen bad substring. If you use the syntax definition graph as a guide, you'll see how to generate all the single-error cases by judicious uses of search-and-replace commands. Once you have the single-error cases done, go on to the double errors if you don't already have more cases than you can handle. With double errors you have to examine each case to be sure that it is still an error case rather than a correct case—similarly for triple and higher errors. If you're starting with a capture/replay system, then you can do the editing either in the system's own editor or with a word processor. It's really more difficult to describe than to do. Think about how you might automate syntax test design with just a copying machine and a hardy typist: then graduate to a word processor. If you understand these primitive methods, then you'll understand how to automate much of syntax test design.

#### 5.2.3. Scripting Languages

A scripting language and processor such as CASL has the features needed to automate the replacement of good substrings by bad ones on the fly. You can use random number generators to select which incorrect, single, character will be used in any spot. Similarly for replacing incorrect keywords by correct ones and for deciding whether or not to delete mandatory fields. You can play all kinds of game with this, but remember that you'll not be able to predict which produced strings are right or wrong. This is a good approach to use if your main purpose is to stress the software rather than to validate the format validation software. If you want to do it right, whatever language you do it in, you have to get more sophisticated.

#### 5.2.4. Random String Generators

Why not just use a random number generator to generate completely random strings? Two reasons: random strings get recognized as invalid too soon, and even a weak front end will catch most bad strings. The probability of hitting vulnerable points is too low, just as it was for random inputs in domain testing—there are too many bad strings in the world. A random string generator is very easy to build. You only have to be careful about where you put string terminators such as carriage returns. Throw the dice for the string length and then pack random characters (except string terminators) in *front* of the terminator until you've reached the required length. Easy but useless. Even with full automation and running at night, this technique caught almost nothing of value.

#### 5.2.5. Getting Sophisticated

Getting sophisticated means building an anti-parser. It's about as complicated as a simple compiler. The language it compiles is BNF, and instead of producing output code it produces structured garbage. I'll assume that you know the rudiments of how a compiler works—if not, this section is beyond you.

As with a compiler, you begin with the lexical scan and build a symbol table. The symbols are single characters, keywords, and left-hand sides of definitions. Keep the three lists separate and replace the source symbols with numerical tokens. Note that each definition string points to one or more other definition strings, characters, or keywords—i.e., to other tokens in the symbol table. There are two ways to screw up—bad tokens and bad pointers.

Start with a covering set of correct test cases. This can be done by hand or by trial and error with random number generators or by using flow-analyzer techniques such as are used in path test generators. Given a good string, you now scan the definition tree by using a tree search procedure. At every node in the subtree corresponding to your good test case you can decide whether you're going to use an incorrect token or an incorrect branch to a subsidiary definition. Use random numbers to replace individual characters, keywords, or pointers to other definitions. Double errors work the same way except that they use the single-error strings as a seed. Similarly, triple-error cases are built on using the double-error cases as a seed. They grow fast.

Another way to look at automated syntax test generation is to view the normal cases as path test generation over BNF as the source language. The errored cases are equivalent to creating **mutations** (BUDD81, WHIT87) of the source "code. " There is no sensitization problem because all paths are achievable.

If you've read this far, then you know that you can't guarantee bad strings, even for single-error cases because that's a known unsolvable problem. Double errors increase the probability of correct strings because of error cancellations. The only (imperfect) way to sort the good from the bad is to use the BNF specification as data to a parser generator and then use the generated parser to sort for you—it can't be perfect but it should do for simple operator commands. What's the point of generating test strings and

using an automatically created parser to sort the good from the bad? If you've got such a parser, use *it* instead of the code you're testing? If we were dealing with entirely new code and a new command language, it would be better to generate the parser and avoid the testing. Using the generated parser as above is useful if it's an older system under maintenance and your objective is to build a big syntax-testing suite where one didn't exist before.

### 5.3. Productivity, Training, and Effectiveness

I used syntax test design as basic training for persons new to a test group. With very little effort they can churn out hundreds of good tests. It's a great confidence builder for people who have never done formal test design before and who may be intimidated by the prospect of subjecting a senior designer's masterpiece to a barrage of calculated heartburn. With nothing more sophisticated for automation than a word processor and a copying machine, a testing trainee can usually produce twenty to thirty fully documented test cases *per hour* after a few days of training.

Syntax testing is also an excellent way of convincing a novice tester that testing is infinite and that the tester's problem is not generating tests but knowing which ones to cull. When my trainees told me that they had run out of ideas, it was time to teach them syntax testing. I would always ask them to produce *all* single-, double-, and triple-error cases for a few well-chosen operator commands. Think about it.

### 5.4. Ad-Lib Tests

Whenever you run a formal system test there's always someone in the crowd who wants to try ad-lib tests. And almost always, the kind of test they want to ad-lib is an input-syntax error test. I used to object to adlibbing, because it didn't prove anything—I thought. It doesn't prove anything substantive about the system, assuming you've done a good job of testing—which is why I used to object to it. It may save time to object to ad-lib tests, but it's not politic. Allowing the ad-lib tests demonstrates that you have confidence in the system and your test. Because a systemwide functional demonstration should have been through a dry run in advance, the actual test execution is largely ceremonial (or should be) and the ad-libbers are part of the ceremony, just as hecklers are part of the ball game—it adds color to the scene.

You should never object if the system's final recipient has cooked up a set of tests of his own. If they're carefully constructed, and well documented, and all the rest, you should welcome yet another independent assault on the system's integrity. Ad-lib tests aren't like that. The customer has a hotshot operator who's earned a reputation for crashing any system in under 2 minutes, and she's itching to get her mitts on yours. There's no prepared set of tests, so you know it's going to be ad-libbed. Agree to the ad-libbing, but only after all other tests have been done. Here's what happens:

**1.** Most of the ad-lib tests will be input strings with format violations, and the system will reject them—as it should.
**2.** Most of the rest are good strings that look bad. The system accepts the strings and does as it was told to do, but the ad-lib tester doesn't recognize it. It will take a lot of explanation to satisfy the customer that it was a cockpit error.
**3.** A few seemingly good strings will be correctly rejected because of a correlation problem between two field values or a state dependency. These situations will also take a lot of explanation.
**4.** At least once, the ad-lib tester will shout "Aha!" and claim that the system was wrong. It will take days to dig out the documentation that shows that the way the system behaves for that case is precisely the way the customer insisted that it behave—over the designer's objections.
**5.** Another time the ad-lib tester will shout "Aha!" but, because the inputs weren't documented

and because nonprinting characters were used, it won't be possible to reproduce the effect. The ad-lib tester will be forever convinced that the system has a flaw.

**6.** There may be one problem, typically related to an interpretation of a specification ambiguity, whose resolution will probably be trivial.

This may be harsh to the ad-lib testers of the world, but such testing proves little or nothing if the system is good, if it's been properly tested from unit on up, and if there has been good quality control. If ad-lib tests do prove something, then the system's so shaky and buggy that it deserves the worst that can be thrown at it.

## 6. TESTABILITY TIPS

### 6.1. The Tip

Here's the whole testability tip:

1. Bring the hidden languages out of the closet.
2. Define the syntax, formally, in BNF.
3. Simplify the syntax definition graph.
4. Build a parser.

I'll quell the objections to the last step by pointing out that building a minicompiler is a typical senior-year computer science project these days. I find that interesting because although most computer science majors build a compiler once in their lifetime, they'll never have a chance to build a real compiler once they're out in the world—we just don't need that many programming language compilers. So drag out the old notebooks to remember how it was done and if you learned your programming before compiler building was an undergraduate exercise, get one of the kids to do it for you.

### 6.2. Compiler Overview

This overview is superficial and intended only to illustrate testability issues. Compilation consists of three main steps: **lexical analysis**, **parsing,** and **code production.** In our context, we deal most often not with a compiler as such, but with an **interpreter;** but if we're testing hidden languages, then indeed we may be interested in a compiler. The main difference between an interpreter and compiler is that an interpreter works one statement at a time and does the equivalent of code production on the fly.

1. *Lexical Analysis*—The lexical analysis phase accomplishes the following:
   **a.** The analyzer knows enough about parsing to identify individual fields, where we define a **field** as a linguistic element that uses no lower-level definitions. That is, a field is defined solely in terms of primitive elements such as characters.
   **b.** Identifies interfield separators or delimiters.
   **c.** Classifies the field (e.g., integer, string, operator, keyword, variable names, program labels). Some fields, such as numbers or strings, may be translated at this point.
   **d.** New variable names and program labels are put into a **symbol table** and replaced by a pointer to that table. If a variable name or program label is already in the table, its appearance in the code is replaced by the pointer. The pointer is an example of a **token.**
   **e. Keywords** (e.g., STOP, IF, ELSE) are also replaced by tokens as are single-character operators. Numbers and strings are also put in a table and replaced by pointers.
   **f.** Delimiters are eliminated where possible, such as interfield delimiters. If the language permits multiple statements per line and there are statement delimiters, statements will be

separated so that subsequent processing will be done one statement at a time. Similarly, multiline statements are combined and thereafter treated as a single string.

**g.** The output of the lexical analysis phase is the partially translated version of the source in which all linguistic components have been replaced by tokens. The act of replacing these components by tokens is called **tokenizing**.

**2.** *Parsing*—Parsing is done on tokenized strings. There are many different strategies used for parsing, and they depend on the kind of statement, the language, and the compiler's objectives. There is also a vast literature on the subject. For general information you can start with LEER84 or MAG184. From our point of view, the validation aspect of parsing consists of showing that the string to be parsed corresponds to a path in the syntax graph. The output of the parser is a tree with the statement identifier at the top, primitive elements (e.g., characters and keywords) at the bottom, and with intermediate nodes corresponding to definitions that were traversed along the path through the syntax graph.

**3.** *Code Production*—Code production consists of scanning the above tree (bottom-up, say) in such a way as to assure that all objects needed are available when they are needed and then replacing the tokens with sequences of instructions that accomplish what the tokens signify. From our point of view and our typical use of syntax testing, the equivalent to code production is a call to a worker subroutine or transfer of control to the appropriate program point.

## 6.3. Typical Software

Unlike the above operation with its clear separation of lexical analysis, parsing, and production, the typical software for (operator command, say) syntax validation and command processing follows what I like to call:

```
(lex-a-little + parse_a_little + process_a_little)*
```

Because the three aspects of command interpretation are hopelessly intermixed, a single bug can involve all three aspects. In other words, the ground for bugs is much more fertile.

## 6.4. Separation of Phases

Separation of the three phases means that it is virtually impossible for a bug to involve, say, the interaction between processing, say, and parsing. We can handle the lexical testing by low-level syntax testing based on one field at a time and be confident that we can ignore lexical-level field interactions, except where field delimiters are involved. Similarly, because most delimiters are eliminated during lexical analysis, we don't have to bother with combinations of syntax errors and long delimiter strings. Lexical-parsing separation means that test strings with combined lexical and syntax errors will not be productive. Parsing-processing separation means that we can separate domain testing from syntax testing. Domain analysis is the first stage of processing and follows parsing, and it is therefore independent of syntax. The bottom line of phase separation is the wholesale elimination of possible double-error and higher-order vulnerabilities and therefore the need to even consider such cases.

In addition to more robust software that's easier to test, there's a payoff in maintenance. The lexical definitions, the syntax, and the equivalent of code that points to working subroutines that do the actual processing can all be stored in tables rather than as code. Separation means separate maintenance. If a processing routine is wrong, there's no need to change the lexical analyzer or parser. If a new command is to be added, chances are that only the parser and keyword table will be affected. Similarly for enhancing existing commands.

What's the price? Possibly more memory, possibly more processing time, but probably neither. The ad

hoc *lex_a_little*, *parse_a_little* code is a jumbled mess that often contains a lot of Code redundancy and wasted reprocessing. My own experience has been that in every instance where we replaced an old-style format analyzer and processor with an explicit lex analyzer and parser, even though we had planned on more time and more space, to our surprise the new software was tighter and faster.

### 6.5. Prerequisites

The language must be decent enough so that it is possible to do lexical analysis before parsing and parsing before processing. That means that it is possible to pull out the tokens in a single left-to-right pass over the string and to do it independently of any other string or statement in the language. The kind of thing that can't be handled for example, are formats such as: "If the first character of the string is an alpha, then every fourth character following it is a token delimiter and the last token is a symbol; but if the first character is numeric, only spaces and parentheses are delimiters and any contiguous string of alphas unbroken by a delimiter is a symbol."—hopeless.

The above is an example of a **context-dependent language**. Languages with more virtuous properties are called **context-free**. We're not dealing with general purpose programming languages here but with simpler minilanguages such as human-interface languages and internal hidden languages. The only excuse for context dependencies is that they were inherited. If it's an internal language then it can, and should, be changed to remove such dependencies. If it's a human-interface language, then the context dependencies must be ripped out because humans can't really deal with such command structures.

## 7. SUMMARY

**1.** Syntax testing begins with a validated format specification. That may be half the work of designing syntax tests.
**2.** Express the syntax in a formal language such as BNF.
**3.** Simplify the syntax definition graph before you design.
**4.** Design syntax tests level by level from top to bottom making only one error at a time, one level at a time, leaving everything else as correct as possible.
**5.** Test the valid cases by (at least) covering the definition graph.
**6.** Concentrate on delimiters, especially paired delimiters, and delimiter errors that could cause syntactic ambiguities.
**7.** Stress all BNF exponent (loop) values as for loop testing.
**8.** Test field-value errors and state dependencies by domain testing and state testing, as appropriate.
**9.** Cut multiple-error tests sharply at two and three simultaneous errors. Look for the bugs along the baseboards and in the corners.
**10.** Take advantage of the design to simplify the test and vice versa.
**11.** Don't forget the valid cases.
**12.** Document copiously and automate, automate, automate—use capture/replay systems and editors to create the tests and build or buy drivers to run them (Chapter 13).
**13.** Give the ad-lib testers the attention they crave and deserve but remember that they can probably be replaced by a random string generator and aren't much more useful.

AND

**14.** Don't forget to cover the valid cases.

<ch8 toc ch10>