Learning to Program with F#

Jon Sporring

Department of Computer Science,
University of Copenhagen

2018-09-24 15:30:57+02:00

Contents

10 | Testing Programs

A software bug is an error in a computer program that causes it to produce an incorrect result or behave in an unintended manner. The term 'bug' was used by Thomas Edison in 1878[1], but made popular in computer science by Grace Hopper, who found a moth interfering with the electronic circuits of the Harward Mark II electromechanical computer and coined the term *bug* for errors in computer programs. The original bug is shown in Figure 10.1. Software is everywhere, and errors therein have a huge economic impact on our society and can threaten lives[2].

· bug

The ISO/IEC organizations have developed standards for software testing[3]. To illustrate basic concepts of software quality, consider a hypothetical route planning system. Essential factors of its quality are:

· functionality

Functionality: Does the software compile and run without internal errors. Does it solve the problem it was intended to solve? E.g., does the route planning software find a suitable route from point a to b?

· reliability

Reliability: Does the software work reliably over time? E.g., does the route planning software work when there are internet dropouts?

· usability

Usability: Is the software easy and intuitive to use by humans? E.g., is it easy to enter addresses and alternative routes in the software's interface?

· efficiency

[1] `https://en.wikipedia.org/wiki/Software_bug`, possibly `http://edison.rutgers.edu/NamesSearch/DocImage.php3?DocId=LB003487`

[2] `https://en.wikipedia.org/wiki/List_of_software_bugs`

[3] ISO/IEC 9126, International standard for the evaluation of software quality, December 19, 1991, later replaced by ISO/IEC 25010:2011

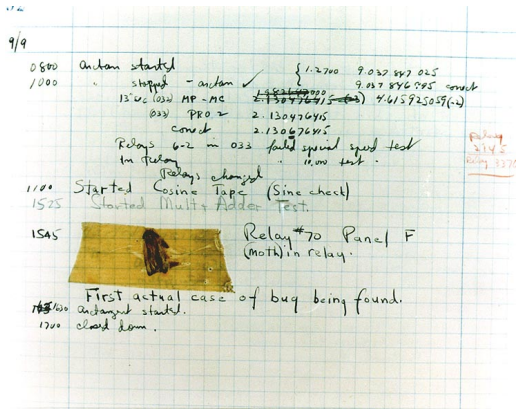

Figure 10.1: The first computer bug, caught by Grace Hopper, U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

Efficiency: How many computer and human resources does the software require? E.g., does it take milliseconds or hours to find a requested route? Can the software run on a mobile platform with limited computer speed and memory?

· maintainability

Maintainability: In case of the discovery of new bugs, is it easy to test and correct the software? Is it easy to extend the software with new functionality? E.g., is it easy to update the map with updated roadmaps and new information? Can the system be improved to work both for car drivers and bicyclists?

· portability

Portability: Is it easy to port the software to new systems such as new server architecture and screen sizes? E.g., if the routing software originally was written for IOS devices, will it be easy to port to Android systems?

The above-mentioned concepts are ordered based on the requirements of the system. Functionality and reliability are perhaps the most important concepts, since if the software does not solve the specified problem, then the software design process has failed. However, many times the problem definition will evolve along with the software development process. But as a bare minimum, the software should run without internal errors and not crash under a well-defined set of circumstances. Furthermore, it is often the case that software designed for the general public requires a lot of attention to the usability of the software, since in many cases non-experts are expected to be able to use the software with little or no prior training. On the other hand, software used internally in companies will be used by a small number of people who become experts in using the software, and it is often less important that the software is easy to understand by non-experts. An example is text processing software like Microsoft Word versus Gnu Emacs and LaTeX. Word is designed to be used by non-experts for small documents such as letters and notes and relies heavily on interfacing with the system using click-interaction. On the other hand, Emacs and LaTeX are for experts for longer and professionally typeset documents and relies heavily on keyboard shortcuts and text-codes for typesetting document entities.

The purpose of *software testing* is to find bugs. When errors are found, then we engage in *debugging*, which is the process of diagnosing and correcting bugs. Once we have a failed software test, i.e., one that does not find any bugs, then we have strengthened our belief in the software, but it is important to note that software testing and debugging rarely removes all bugs, and with each correction or change of software there is a fair risk new bugs being introduced. It is not exceptional that the testing-software is as large as the software being tested.

· software testing
· debugging

In this chapter, we will focus on two approaches to software testing which emphasize functionality: *white-box* and *black-box testing*. An important concept in this context is *unit testing*, where the program is considered in smaller pieces, called units, and for which accompanying programs for testing can be made which test these units automatically. Black-box testing considers the problem formulation and the program interface, and can typically be written early in the software design phase. In contrast, white-box testing considers the program text, and thus requires the program to be available. Thus, there is a tendency for black-box test programs to be more stable, while white-box testing typically is developed incrementally alongside the software development.

· white-box testing
· black-box testing
· unit testing

To illustrate software testing, we'll start with a problem:

Problem 10.1

Given any date in the Gregorian calendar, calculate the day of the week.

Facts about dates in the Gregorian calendar are:

- Combinations of dates and weekdays repeat themselves every 400 years.
- The typical length of the months January, February, ... follow the knuckle rule, i.e., January belongs to the index knuckle, February to the space between the index and the middle finger, and August restarts or starts on the other hand. All knuckle months have 31 days, all spacing months have 30 days except February, which has 29 days on leap years and 28 days all other years.
- A leap year is a multiple of 4, except if it is also a multiple of 100 but not of 400.

Many solutions to the problem have been discovered, and here we will base our program on Gauss' method, which is based on integer division and calculates the weekday of the 1st of January of a given year. For any other date, we will count our way through the weeks from the previous 1st of January. The algorithm relies on an enumeration of weekdays starting with Sunday = 0, Monday = 1, ..., and Saturday = 6. Our proposed solution is shown in Listing 10.1.[4]

Listing 10.1 date2Day.fsx:
A function that can calculate day-of-week from any date in the Gregorian calendar.

```
let januaryFirstDay (y : int) =
  let a = (y - 1) % 4
  let b = (y - 1) % 100
  let c = (y - 1) % 400
  (1 + 5 * a + 4 * b + 6 * c) % 7

let rec sum (lst : int list) j =
  if 0 <= j && j < lst.Length then
    lst.[0] + sum lst.[1..] (j - 1)
  else
    0

let date2Day d m y =
  let dayPrefix =
    ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
  let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 =
  0)) then 29 else 28
  let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30; 31;
  30; 31]
  let dayOne = januaryFirstDay y
  let daysSince = (sum daysInMonth (m - 2)) + d - 1
  let weekday = (dayOne + daysSince) % 7;
  dayPrefix.[weekday] + "day"
```

10.1 White-box Testing

White-box testing considers the text of a program. The degree to which the text of the program is covered in the test is called the *coverage*. Since our program is small, we have the opportunity to ensure that all functions are called at least once, which is called *function*

· white-box testing
· coverage

[4] Jon: **This example relies on lists, which has not been introduced yet.**

coverage, and we will also be able to test every branching in the program, which is called *branching coverage*. If both are fulfilled, we say that we have *statement coverage*. The procedure is as follows:

- function coverage
- branching coverage
- statement coverage

1. Decide which units to test: The program shown in Listing 10.1 has 3 functions, and we will consider these each as a unit, but we might as well just have chosen `date2Day` as a single unit. The important part is that the union of units must cover the whole program text, and since `date2Day` calls both `januaryFirstDay` and `sum`, designing test cases for the latter two is superfluous. However, we may have to do this anyway when debugging, and we may choose at a later point to use these functions separately, and in both cases, we will be able to reuse the testing of the smaller units.

2. Identify branching points: The function `januaryFirstDay` has no branching function, `sum` has one, and depending on the input values, two paths through the code may be used, and `date2Day` has one where the number of days in February is decided. Note that in order to test this, our test-date must be March 1 or later. In this example, there are only examples of `if`-branch points, but they may as well be loops and pattern matching expressions. In the Listing 10.2 it is shown that the branch points have been given a comment and a number.

Listing 10.2 date2DayAnnotated.fsx:
In white-box testing, the branch points are identified.

```
1  // Unit: januaryFirstDay
2  let januaryFirstDay (y : int) =
3    let a = (y - 1) % 4
4    let b = (y - 1) % 100
5    let c = (y - 1) % 400
6    (1 + 5 * a + 4 * b + 6 * c) % 7
7
8  // Unit: sum
9  let rec sum (lst : int list) j =
10   (* WB: 1 *)
11   if 0 <= j && j < lst.Length then
12     lst.[0] + sum lst.[1..] (j - 1)
13   else
14     0
15
16 // Unit: date2Day
17 let date2Day d m y =
18   let dayPrefix =
19     ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri";
    "Satur"]
20   (* WB: 1 *)
21   let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400
    = 0)) then 29 else 28
22   let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30;
    31; 30; 31]
23   let dayOne = januaryFirstDay y
24   let daysSince = (sum daysInMonth (m - 2)) + d - 1
25   let weekday = (dayOne + daysSince) % 7;
26   dayPrefix.[weekday] + "day"
```

3. For each unit, produce an input set that tests each branch: In our example, the branch points depend on a Boolean expression, and for good measure, we are going

to test each term that can lead to branching. Using 't' and 'f' for `true` and `false`, we thus write,

Unit	Branch	Condition	Input	Expected output
`januaryFirstDay`	0	-	`2016`	`5`
`sum`	1	`0 <= j && j < lst.Length`		
	1a	`t && t`	`[1; 2; 3] 1`	`3`
	1b	`f && t`	`[1; 2; 3] -1`	`0`
	1c	`t && f`	`[1; 2; 3] 10`	`0`
	1d	`f && f`	-	-
`date2Day`	1	`(y % 4 = 0) && ((y % 100 <> 0) || (y % 400 = 0))`		
	-	`t && (t || t)`	-	-
	1a	`t && (t || f)`	`8 9 2016`	`Thursday`
	1b	`t && (f || t)`	`8 9 2000`	`Friday`
	1c	`t && (f || f)`	`8 9 2100`	`Wednesday`
	-	`f && (t || t)`	-	-
	1d	`f && (t || f)`	`8 9 2015`	`Tuesday`
	-	`f && (f || t)`	-	-
	-	`f && (f || f)`	-	-

The impossible cases have been intentionally blank, e.g., it is not possible for $j < 0$ and $j > n$ for some positive value n.

4. Write a program that tests all these cases and checks the output, e.g.,

Listing 10.3 date2DayWhiteTest.fsx:
The tests identified by white-box analysis. The program from Listing 10.2 has been omitted for brevity.

```
printfn "White-box testing of date2Day.fsx"
printfn "  Unit: januaryFirstDay"
printfn "    Branch: 0 - %b" (januaryFirstDay 2016 = 5)

printfn "  Unit: sum"
printfn "    Branch: 1a - %b" (sum [1; 2; 3] 1 = 3)
printfn "    Branch: 1b - %b" (sum [1; 2; 3] -1 = 0)
printfn "    Branch: 1c - %b" (sum [1; 2; 3] 10 = 0)

printfn "  Unit: date2Day"
printfn "    Branch: 1a - %b" (date2Day 8 9 2016 =
  "Thursday")
printfn "    Branch: 1b - %b" (date2Day 8 9 2000 =
  "Friday")
printfn "    Branch: 1c - %b" (date2Day 8 9 2100 =
  "Wednesday")
printfn "    Branch: 1d - %b" (date2Day 8 9 2015 =
  "Tuesday")
```

```
$ fsharpc --nologo date2DayWhiteTest.fsx && mono
  date2DayWhiteTest.exe
White-box testing of date2Day.fsx
  Unit: januaryFirstDay
    Branch: 0 - true
  Unit: sum
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
  Unit: date2Day
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
    Branch: 1d - true
```

Notice that the output of the tests is organized such that they are enumerated per unit, hence we can rearrange as we like and still uniquely refer to a unit's test. Also, the output of the test program produces a list of tests that should return true or success or a similar positively loaded word, but without further or only little detail, such that we at a glance can identify any test that produced unexpected results.

After the white-box testing has failed to find errors in the program, we have some confidence in the program, since we have run every line at least once. It is, however, in no way a guarantee that the program is error free, which is why white-box testing is often accompanied with black-box testing to be described next.

10.2 Black-box Testing

In black-box testing, the program is considered a black box, and no knowledge is required about how a particular problem is solved. In fact, it is often useful not to have that

knowledge at all. It is rarely possible to test all input to a program, so in black-box testing, the solution is tested for typical and extreme cases based on knowledge of the problem. The procedure is as follows:

1. Decide on the interface to use: It is useful to have an agreement with the software developers about what interface is to be used, e.g., in our case, the software developer has made a function `date2Day d m y` where `d`, `m`, and `y` are integers specifying the day, month, and year.

2. Make an overall description of the tests to be performed and their purpose:

   1 a consecutive week, to ensure that all weekdays are properly returned

   2 two set of consecutive days across boundaries that may cause problems: across a new year, and across a regular month boundary.

   3 a set of consecutive days across February-March boundaries for a leap and non-leap year

   4 four dates after February in a non-leap year, a non-multiple-of-100 leap year, a multiple-of-100-but-not-of-400 non-leap year, and a multiple-of-400 leap year.

   Given no information about the program's text, there are other dates that one could consider as likely candidates for errors, but the above is judged to be a fair coverage.

3. Choose a specific set of input and expected output relations on the tabular form:

Test number	Input	Expected output
1a	1 1 2016	Friday
1b	2 1 2016	Saturday
1c	3 1 2016	Sunday
1d	4 1 2016	Monday
1e	5 1 2016	Tuesday
1f	6 1 2016	Wednesday
1g	7 1 2016	Thursday
2a	31 12 2014	Wednesday
2b	1 1 2015	Thursday
2c	30 9 2017	Saturday
2d	1 10 2017	Sunday
3a	28 2 2016	Sunday
3b	29 2 2016	Monday
3c	1 3 2016	Tuesday
3d	28 2 2017	Tuesday
3e	1 3 2017	Wednesday
4a	1 3 2015	Sunday
4b	1 3 2012	Thursday
4c	1 3 2000	Wednesday
4d	1 3 2100	Monday

4. Write a program executing the tests, as shown in Listing 10.4 and 10.5.

Listing 10.4 date2DayBlackTest.fsx:
The tests identified by black-box analysis. The program from Listing 10.2 has been omitted for brevity.

```
28 let testCases = [
29   ("A complete week",
30    [(1, 1, 2016, "Friday");
31     (2, 1, 2016, "Saturday");
32     (3, 1, 2016, "Sunday");
33     (4, 1, 2016, "Monday");
34     (5, 1, 2016, "Tuesday");
35     (6, 1, 2016, "Wednesday");
36     (7, 1, 2016, "Thursday");]);
37   ("Across boundaries",
38    [(31, 12, 2014, "Wednesday");
39     (1, 1, 2015, "Thursday");
40     (30, 9, 2017, "Saturday");
41     (1, 10, 2017, "Sunday")]);
42   ("Across Feburary boundary",
43    [(28, 2, 2016, "Sunday");
44     (29, 2, 2016, "Monday");
45     (1, 3, 2016, "Tuesday");
46     (28, 2, 2017, "Tuesday");
47     (1, 3, 2017, "Wednesday")]);
48   ("Leap years",
49    [(1, 3, 2015, "Sunday");
50     (1, 3, 2012, "Thursday");
51     (1, 3, 2000, "Wednesday");
52     (1, 3, 2100, "Monday")]);
53   ]
54
55 printfn "Black-box testing of date2Day.fsx"
56 for i = 0 to testCases.Length - 1 do
57   let (setName, testSet) = testCases.[i]
58   printfn "  %d. %s" (i+1) setName
59   for j = 0 to testSet.Length - 1 do
60     let (d, m, y, expected) = testSet.[j]
61     let day = date2Day d m y
62     printfn "    test %d - %b" (j+1) (day = expected)
```

Listing 10.5: Output from Listing 10.4.

```
$ fsharpc --nologo date2DayBlackTest.fsx && mono
   date2DayBlackTest.exe
Black-box testing of date2Day.fsx
  1. A complete week
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
    test 6 - true
    test 7 - true
  2. Across boundaries
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
  3. Across Feburary boundary
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
  4. Leap years
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
```

Notice how the program has been made such that it is almost a direct copy of the table produced in the previous step.

A black-box test is a statement of what a solution should fulfill for a given problem. Hence, **it is a good idea to make a black-box test early in the software design phase, in order to clarify the requirements for the code to be developed and take an outside view of the code prior to developing it.** Advice

After the black-box testing has failed to find errors in the program, we have some confidence in the program, since, from a user's perspective, the program produces sensible output in many cases. It is, however, in no way a guarantee that the program is error free.

10.3 Debugging by Tracing

Once an error has been found by testing, the *debugging* phase starts. The cause of a bug can either be that the chosen algorithm is the wrong one for the job, or the implementation of it has an error. In the debugging process, we have to keep an open mind and not rely on assumptions, since assumptions tend to blind the reader of a text. A frequent source of errors is that the state of a program is different than expected, e.g., because the calculation performed is different than intended, or that the return of a library function is different than expected. The most important tool for debugging is *simplification*. This is similar to white-box testing, but where the units tested are very small. E.g., the suspected piece of code could be broken down into smaller functions or code snippets, which are given · debugging

well-defined input, and, e.g., use `printfn` statements to obtain the output of the code snippet. Another related technique is to use *mockup code*, which replaces parts of the code with code that produces safe and relevant results. If the bug is not obvious, then more rigorous techniques must be used, such as *tracing*. Some development interfaces have a built-in tracing system, e.g., `fsharpi` will print inferred types and some binding values. However, often the source of a bug is due to a misunderstanding of the flow of data through a program execution, and we will in the following introduce *hand tracing* as a technique to simulate the execution of a program by hand.

· mockup code
· tracing
· hand tracing

Consider the program in Listing 10.6. [5]

Listing 10.6 gcd.fsx:
The greatest common divisor of 2 integers.

```
1  let rec gcd a b =
2    if a < b then
3      gcd b a
4    elif b > 0 then
5      gcd b (a % b)
6    else
7      a
8
9  let a = 10
10 let b = 15
11 printfn "gcd %d %d = %d" a b (gcd a b)
```

```
1  $ fsharpc --nologo gcd.fsx && mono gcd.exe
2  gcd 10 15 = 5
```

The program includes a function for calculating the greatest common divisor of 2 integers, and calls this function with the numbers 10 and 15. Hand tracing this program means that we simulate its execution and, as part of that, keep track of the bindings, assignments and input and output of the program. To do this, we need to consider code snippets' *environments*. E.g., to hand trace the above program, we start by noting the outer environment, called E_0 for short. In line 1, the `gcd` identifier is bound to a function, hence we write:

· environment

$$E_0 : \\ \quad \text{gcd} \rightarrow \big((a, b), \text{gcd-body}, \varnothing\big)$$

Function bindings like this one are noted as closures, which are triplets of the form $\big(\text{arguments}, \text{expression}, \text{environment}\big)$. The symbol $\varnothing$ denotes the empty environment. The closure is everything needed for the expression to be calculated. Here, we wrote gcd-body to denote everything after the equal sign in the function binding. Next, F# executes line 9 and 10, and we update our environment to reflect the bindings as,

$$E_0 : \\ \quad \text{gcd} \rightarrow \big((a, b), \text{gcd-body}, \varnothing\big) \\ \quad a \rightarrow 10 \\ \quad b \rightarrow 15$$

In line 11, the function is evaluated. This initiates a new environment E_1, and we update

[5] Jon: **This program uses recursion, which has not been introduced yet.**

our trace as,

$$
\begin{aligned}
&E_0 : \\
&\quad \text{gcd} \to \big((a, b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line 11: gcd a b} \to ? \\
&E_1 : \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

where the new environment is noted to have gotten its argument names `a` and `b` bound to the values `10` and `15` respectively, and where the return of the function to environment E_0 is yet unknown, so it is noted as a question mark. In line 2, the comparison `a < b` is checked, and since we are in environment E_1, this is the same as checking `10 < 15`, which is true, so the program executes line 3. Hence, we initiate a new environment E_2 and update our trace as,

$$
\begin{aligned}
&E_0 : \\
&\quad \text{gcd} \to \big((a, b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line 11: gcd a b} \to ? \\
&E_1 : \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 3: gcd b a} \to ? \\
&E_2 : \big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

where in the new environment, `a` and `b` are bound to the values `15` and `10` respectively. In E_2, `10 < 15` is false, so the program evaluates `b > 0`, which is true, hence line 5 is executed. This calls `gcd` once again, but with new arguments. Since `a % b` is parenthesized, it is evaluated before `gcd` is called.

Hence, we update our trace as,

$$
\begin{aligned}
&E_0 : \\
&\quad \text{gcd} \to \big((a, b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line 11: gcd a b} \to ? \\
&E_1 : \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 3: gcd b a} \to ? \\
&E_2 : \big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \to 5 \\
&\quad \text{line 5: gcd b (a \% b)} \to ? \\
&E_3 : \big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

Again we fall through to line 5, evaluate the remainder operator and initiate a new envi-

ronment,

$$
\begin{aligned}
&E_0: \\
&\quad \text{gcd} \to \big((a, b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line 11: gcd a b} \to ? \\
&E_1: \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 3: gcd b a} \to ? \\
&E_2: \big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \to 5 \\
&\quad \text{line 5: gcd b (a \% b)} \to ? \\
&E_3: \big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \to 0 \\
&\quad \text{line 5: gcd b (a \% b)} \to ? \\
&E_4: \big((a \to 5, b \to 0), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

This time both `a < b` and `b > 0` are false, so we fall through to line 7, and `gcd` from E_4 returns its value of `a`, which is `5`, so we scratch E_4 and change the question mark in E_3 to `5`,

$$
\begin{aligned}
&E_0: \\
&\quad \text{gcd} \to \big((a, b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line 11: gcd a b} \to ? \\
&E_1: \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 3: gcd b a} \to ? \\
&E_2: \big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \to 5 \\
&\quad \text{line 5: gcd b (a \% b)} \to ? \\
&E_3: \big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \to 0 \\
&\quad \text{line 5: gcd b (a \% b)} \to \cancel{?}\, 5 \\
&\cancel{E_4}: \big((a \to 5, b \to 0), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

Now, line 5 in E_3 is also a return point of `gcd`, hence we scratch E_3 and change the question

mark in E_2 to 5,

$$
\begin{aligned}
&E_0: \\
&\quad \text{gcd} \to \big((a, b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line 11: gcd a b} \to ? \\
&E_1: \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 3: gcd b a} \to ? \\
&E_2: \big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \to 5 \\
&\quad \text{line 5: gcd b (a \% b)} \to \not{?}\, 5 \\
&\not{E_3}: \big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \to 0 \\
&\quad \text{line 5: gcd b (a \% b)} \to \not{?}\, 5 \\
&\not{E_4}: \big((a \to 5, b \to 0), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

and likewise, for E_2 and E_1,

$$
\begin{aligned}
&E_0: \\
&\quad \text{gcd} \to \big((a, b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line 11: gcd a b} \to \not{?}\, 5 \\
&\not{E_1}: \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 3: gcd b a} \to \not{?}\, 5 \\
&\not{E_2}: \big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \to 5 \\
&\quad \text{line 5: gcd b (a \% b)} \to \not{?}\, 5 \\
&\not{E_3}: \big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \to 0 \\
&\quad \text{line 5: gcd b (a \% b)} \to \not{?}\, 5 \\
&\not{E_4}: \big((a \to 5, b \to 0), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

Now we are able to continue the program in environment E_0 with the `printfn` statement,

and we write,

$$
\begin{aligned}
&E_0: \\
&\quad \text{gcd} \rightarrow \big((a, b), \text{gcd-body}, \varnothing\big) \\
&\quad a \rightarrow 10 \\
&\quad b \rightarrow 15 \\
&\quad \text{line 11: gcd a b} \rightarrow \not{?}\, 5 \\
&\quad \text{line 11: stdout} \rightarrow \text{"gcd 10 15 = 5"} \\
&\not{E_1}: \big((a \rightarrow 10, b \rightarrow 15), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 3: gcd b a} \rightarrow \not{?}\, 5 \\
&\not{E_2}: \big((a \rightarrow 15, b \rightarrow 10), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \rightarrow 5 \\
&\quad \text{line 5: gcd b (a \% b)} \rightarrow \not{?}\, 5 \\
&\not{E_3}: \big((a \rightarrow 10, b \rightarrow 5), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \rightarrow 0 \\
&\quad \text{line 5: gcd b (a \% b)} \rightarrow \not{?}\, 5 \\
&\not{E_4}: \big((a \rightarrow 5, b \rightarrow 0), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

which completes the hand tracing of `gcd.fsx`.

F# uses the lexical scope, which implies that besides function arguments, we also at times need to consider the environment at the place of writing. Consider the program in Listing 10.7.

Listing 10.7 lexicalScopeTracing.fsx: Example of lexical scope and closure environment.

```
let testScope x =
  let a = 3.0
  let f z = a * z
  let a = 4.0
  f x
printfn "%A" (testScope 2.0)
```

```
$ fsharpc --nologo lexicalScopeTracing.fsx
$ mono lexicalScopeTracing.exe
6.0
```

To hand trace this, we start by creating the outer environment, define the closure for `testScope`, and reach line 6,

$$
\begin{aligned}
&E_0: \\
&\quad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big) \\
&\quad \text{line 6: testScope 2.0} \rightarrow ?
\end{aligned}
$$

We create a new environment for `testScope` and note the bindings,

$$
\begin{aligned}
&E_0 : \\
&\quad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big) \\
&\quad \text{line 6: testScope 2.0} \rightarrow ? \\
&E_1 : \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big) \\
&\quad a \rightarrow 3.0 \\
&\quad \text{f} \rightarrow \big(z, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)\big) \\
&\quad a \rightarrow 4.0
\end{aligned}
$$

Since we are working with lexical scope, `a` is noted twice, and its interpretation is by lexical order. Hence, the environment for the closure of `f` is everything above in E_1, so we add $a \rightarrow 3.0$ and $x \rightarrow 2.0$. In line 5, `f` is called, so we create an environment based on its closure,

$$
\begin{aligned}
&E_0 : \\
&\quad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big) \\
&\quad \text{line 6: testScope 2.0} \rightarrow ? \\
&E_1 : \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big) \\
&\quad a \rightarrow 3.0 \\
&\quad \text{f} \rightarrow \big(z, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)\big) \\
&\quad a \rightarrow 4.0 \\
&\quad \text{line 5: f x} \rightarrow ? \\
&E_2 : \big(z \rightarrow 2.0, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0\big)
\end{aligned}
$$

The expression in the environment E_2 evaluates to `6.0`, and unraveling the scopes we get,

$$
\begin{aligned}
&E_0 : \\
&\quad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big) \\
&\quad \text{line 6: testScope 2.0} \rightarrow \not{?}\, 6.0 \\
&\quad \text{line 6: stdout} \rightarrow \text{"6.0"} \\
&\not{E_1} : \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big) \\
&\quad a \rightarrow 3.0 \\
&\quad \text{f} \rightarrow \big(z, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)\big) \\
&\quad a \rightarrow 4.0 \\
&\quad \text{line 5: f x} \rightarrow \not{?}\, 6.0 \\
&\not{E_2} : \big(z \rightarrow 2.0, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)\big)
\end{aligned}
$$

For mutable binding, the scope is dynamic. For this, we need the concept of storage. Consider the program in Listing 10.8.

Listing 10.8 dynamicScopeTracing.fsx: Example of lexical scope and closure environment.

```
let testScope x =
  let mutable a = 3.0
  let f z = a * z
  a <- 4.0
  f x
printfn "%A" (testScope 2.0)
```

```
$ fsharpc --nologo dynamicScopeTracing.fsx
$ mono dynamicScopeTracing.exe
8.0
```

We add a storage area to our hand tracing, e.g., line 6,

$$
\begin{aligned}
&\text{Store}: \\
&\quad E_0: \\
&\qquad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big) \\
&\qquad \text{line 6: testScope } 2.0 \rightarrow ?
\end{aligned}
$$

So when we generate environment E_1, the mutable binding is to a storage location,

$$
\begin{aligned}
&\text{Store}: \\
&\qquad \alpha_1 \rightarrow 3.0 \\
&\quad E_0: \\
&\qquad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big) \\
&\qquad \text{line 6: testScope } 2.0 \rightarrow ? \\
&\quad E_1: \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big) \\
&\qquad a \rightarrow \alpha_1
\end{aligned}
$$

which is assigned the value `3.0` at the definition of `a`. Now, the definition of `f` is using the storage location,

$$
\begin{aligned}
&\text{Store}: \\
&\qquad \alpha_1 \rightarrow 3.0 \\
&\quad E_0: \\
&\qquad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big) \\
&\qquad \text{line 6: testScope } 2.0 \rightarrow ? \\
&\quad E_1: \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big) \\
&\qquad a \rightarrow \alpha_1 \\
&\qquad \text{f} \rightarrow \big(z, \text{a * z}, (a \rightarrow \alpha_1, x \rightarrow 2.0)\big)
\end{aligned}
$$

and in line 4, it is the value in the storage which is updated,

$$
\begin{aligned}
&\text{Store}: \\
&\qquad \alpha_1 \to \cancel{3.0}\ 4.0 \\
&E_0: \\
&\qquad \text{testScope} \to \big(x, \text{testScope-body}, \varnothing\big) \\
&\qquad \text{line 6: testScope } 2.0 \to ? \\
&E_1: \big(x \to 2.0, \text{testScope-body}, \varnothing\big) \\
&\qquad a \to \alpha_1 \\
&\qquad \text{f} \to \big(z, \text{a * z}, (a \to \alpha_1, x \to 2.0)\big)
\end{aligned}
$$

Hence,

$$
\begin{aligned}
&\text{Store}: \\
&\qquad \alpha_1 \to \cancel{3.0}\ 4.0 \\
&E_0: \\
&\qquad \text{testScope} \to \big(x, \text{testScope-body}, \varnothing\big) \\
&\qquad \text{line 6: testScope } 2.0 \to ? \\
&E_1: \big(x \to 2.0, \text{testScope-body}, \varnothing\big) \\
&\qquad a \to \alpha_1 \\
&\qquad \text{f} \to \big(z, \text{a * z}, (a \to \alpha_1, x \to 2.0)\big) \\
&\qquad \text{line 5: f x} \to ? \\
&E_2: \big(z \to 2.0, \text{a * z}, (a \to \alpha_1, x \to 2.0)\big)
\end{aligned}
$$

and the return value from `f` evaluated in environment E_2 now reads the value `4.0` for `a` and returns `8.0`. Hence,

$$
\begin{aligned}
&\text{Store}: \\
&\qquad \alpha_1 \to \cancel{3.0}\ 4.0 \\
&E_0: \\
&\qquad \text{testScope} \to \big(x, \text{testScope-body}, \varnothing\big) \\
&\qquad \text{line 6: testScope } 2.0 \to \cancel{?}\ 8.0 \\
&\qquad \text{line 6: stdout} \to \text{"8.0"} \\
&\cancel{E_1}: \big(x \to 2.0, \text{testScope-body}, \varnothing\big) \\
&\qquad a \to \alpha_1 \\
&\qquad \text{f} \to \big(z, \text{a * z}, (a \to \alpha_1, x \to 2.0)\big) \\
&\qquad \text{line 5: f x} \to \cancel{?}\ 8.0 \\
&\cancel{E_2}: \big(z \to 2.0, \text{a * z}, (a \to \alpha_1, x \to 2.0)\big)
\end{aligned}
$$

By the above examples, it is seen that hand tracing can be used to study the flow of data through a program in detail. It may seem tedious in the beginning, but the care illustrated above is useful to ensure rigor in the analysis. Most will find that once accustomed to the method, the analysis can be performed rigorously but with less paperwork, and in conjunction with strategically placed debugging `printfn` statements, it is a very valuable tool for debugging.

Bibliography

[1] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345—-363, 1936.

[2] Ole-Johan Dahl and Kristen Nygaard. SIMULA a language for programming and description of discrete event systems. introduction and user's manual. Technical report, Norwegian Computing Center, 1967.

[3] European Computer Manufacturers Association (ECMA). Standard ecma-335, common language infrastructure (cli). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[4] International Organization for Standardization. Iso/iec 23271:2012, common language infrastructure (cli). `https://www.iso.org/standard/58046.html`.

[5] Object Management Group. Uml version 2.0. `http://www.omg.org/spec/UML/2.0/`.

[6] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.

[7] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[8] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. `http://worldpowersystems.com/projects/codes/X3.4-1963/`.

[9] George Pólya. *How to solve it*. Princeton University Press, 1945.

[10] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936.

Index