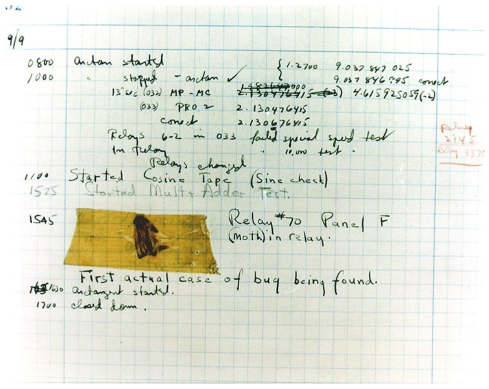
1 Testing Programs

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][2], 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 1.1. Software is everywhere, and errors therein



9/9

0800 Antan started
1000 " stopped - antan ✓ {1.2700 9.037 847 025
9.037 846 995 conect
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (033) 4.615925059(-2)
(033) PRO 2 2.130476415
conect 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 11.000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 antangent started.
1700 closed down.

Figure 1.1: The first computer bug, caught by Grace Hopper, U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

have a huge economic impact on our society and can threaten lives[3].

The ISO/IEC organizations have developed standards for software testing[4]. To illustrate basic concepts of software quality, consider a hypothetical route planning system. Essential factors of its quality are:

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?

[1] https://en.wikipedia.org/wiki/Software_bug

[2] http://edison.rutgers.edu/NamesSearch/DocImage.php3?DocId=LB003487

[3] https://en.wikipedia.org/wiki/List_of_software_bugs

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

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

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: 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: 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: 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.

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.

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

Problem 1.1

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 1.1.

1.1 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

Listing 1.1 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"
```

is called *function 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:

1. Decide which units to test: The program shown in Listing 1.1 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 1.2 it is shown that the branch points have been given a comment and a number.

Listing 1.2 date2DayAnnotated.fsx:
In white-box testing, the branch points are identified.

```
// Unit: januaryFirstDay
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

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

// Unit: date2Day
let date2Day d m y =
  let dayPrefix =
    ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri";
   "Satur"]
  (* WB: 1 *)
  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"
```

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 as shown in Table 1.1. The impossible cases have been intentionally blank, e.g., it is not possible for $j < 0$ and $j > n$ for some positive value n.

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)`	-	-

Table 1.1: Unit test

4. Write a program that tests all these cases and checks the output, see Listing 1.3.

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.

Listing 1.3 date2DayWhiteTest.fsx:
The tests identified by white-box analysis. The program from Listing 1.2 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
```

1.2 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 as shown in Table 1.2.

4. Write a program executing the tests, as shown in Listing 1.4 and 1.5. 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.**

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

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

Table 1.2: Black-box testing

sensible output in many cases. It is, however, in no way a guarantee that the program is error free.

Listing 1.4 date2DayBlackTest.fsx:
The tests identified by black-box analysis. The program from Listing 1.2 has been omitted for brevity.

```
let testCases = [
  ("A complete week",
   [(1, 1, 2016, "Friday");
    (2, 1, 2016, "Saturday");
    (3, 1, 2016, "Sunday");
    (4, 1, 2016, "Monday");
    (5, 1, 2016, "Tuesday");
    (6, 1, 2016, "Wednesday");
    (7, 1, 2016, "Thursday");]);
  ("Across boundaries",
   [(31, 12, 2014, "Wednesday");
    (1, 1, 2015, "Thursday");
    (30, 9, 2017, "Saturday");
    (1, 10, 2017, "Sunday")]);
  ("Across Feburary boundary",
   [(28, 2, 2016, "Sunday");
    (29, 2, 2016, "Monday");
    (1, 3, 2016, "Tuesday");
    (28, 2, 2017, "Tuesday");
    (1, 3, 2017, "Wednesday")]);
  ("Leap years",
   [(1, 3, 2015, "Sunday");
    (1, 3, 2012, "Thursday");
    (1, 3, 2000, "Wednesday");
    (1, 3, 2100, "Monday")]);
  ]

printfn "Black-box testing of date2Day.fsx"
for i = 0 to testCases.Length - 1 do
  let (setName, testSet) = testCases.[i]
  printfn "  %d. %s" (i+1) setName
  for j = 0 to testSet.Length - 1 do
    let (d, m, y, expected) = testSet.[j]
    let day = date2Day d m y
    printfn "    test %d - %b" (j+1) (day = expected)
```

Listing 1.5: Output from Listing 1.4.

```
$ 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
```

1.3 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. 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 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*, that is, replacing 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 section introduce *Trace by hand* as a technique to simulate the execution of a program by hand. In the following section, tracing will refer to the Trace by hand method.

To understand the method of Tracing by hand, we will consider 3 imperative programs of gradually increasing complexity: a program using function call, a program including a `for`-loop, and a program with dynamic scope. In **??** we give a fourth example using recursion, a concept to be introduced in **??**.

Tracing may seem tedious in the beginning, but in conjunction with strategically placed debugging `printfn` statements, it is a very valuable tool for debugging.

1.3.1 Tracing Function Calls

Consider the program in Listing 1.6. The program calls `testScope 2.0`, and by running the program, we see that the return-value is `6.0` and not `8.0`, as we had expected. Hence, we will use tracing to understand the result.

Tracing a program by hand means that we simulate its execution and, as part of that, keep track of the bindings, assignments closures, scopes, and input and output of the program. To do this, we need to consider the concept of *environments*.

Environments describe bindings available to the program at the present scope and at a particular time and place in the code. There is always an outer environment, called E_0, and each time we call a function or create a scope, we create a new environment.

Listing 1.6 lexicalScopeTracing.fsx:
Example of lexical scope and closure environment.

```
1 let testScope x =
2   let a = 3.0
3   let f z = a * z
4   let a = 4.0
5   f x
6 printfn "%A" (testScope 2.0)
```

```
1 $ fsharpc --nologo lexicalScopeTracing.fsx
2 $ mono lexicalScopeTracing.exe
3 6.0
```

Only one environment can be active at a time, and it is updated as we simulate the execution of code with new bindings and temporary evaluations of expressions. Once a scope is closed, then its environment is deleted and a return-value is transported to its enclosing environment. In tracing, we note return-values explicitely. Likewise, output from, e.g., `printfn` is reported with a special notation.

To trace code, we make a table with 4 columns: Step, Line, Environment, and Bindings and evaluations. The Step column enumerates the steps performed. The Line column contains the program-line treated in a step *where* the present environment is updated. The Environment contains the name of the present environment, and Bindings . . . shows *what* in the environment is updated.

The code in Listing 1.6 contains a function definition and a call, hence, the first lines of our table looks like,

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope $= \big((x), \text{testScope-body}, ()\big)$
2	6	E_0	testScope $2.0 = ?$

The elements of the table is to be understood as follows. Step 0 initializes the outer environment. In order for us to remember that the environment is empty, we write the unit value "`()`". Reading the code from top to bottom, the first nonempty and non-comment line we meet is line 1, hence, in Step 1, we update the enviroment with the binding of a function to the name `testScope`. Since functions are values in F#, we note their bindings by their closures: a tuple of argument names, the function-body, and the values lexically available at the place of binding. See **??** for more information on closures. Following the function-binding, the `printfn` statement is called in line 6 to print the result `testScope 2.0`. However, before we can produce any output, we must first evaluate `testScope 2.0`. Since we do not yet know what this function evaluates to, in Step 2 we simply write the call with a question mark. The call causes the creation of a new environment, and we continue our table as follows,

Step	Line	Env.	Bindings and evaluations
3	1	E_1	$\big((x = 2.0), \text{testScope-body}, ()\big)$

This means that we are going to execute the code in testScope-body. The function was called with 2.0 as argument, causing $x = 2.0$. Hence, the only binding available at the start of this environment is to the name `x`. In the testScope-body, we make 3 further bindings and a function call. First to `a`, then to `f`, then to another `a`, which will overshadow the previous binding, and finally we call `f`. Thus, our table is updated as follows,

Step	Line	Env.	Bindings and evaluations
4	2	E_1	$a = 3.0$
5	3	E_1	$\text{f} = \big((z), \text{a * z}, (a = 3.0, x = 2.0)\big)$
6	4	E_1	$a = 4.0$
7	5	E_1	$\text{f}\ x = ?$

Note that by lexical scope, the closure of `f` includes everything above its binding in E_1, and therefore we add $a = 3.0$ and $x = 2.0$ to the environment element in its closure. This has consequences for the following call to `f` in line 5, which creates a new environment based on `f`'s closure and the value of its arguments. The value of `x` in Step 7 is found by looking in the previous steps for the last binding to the name `x` in E_1, which occurs in Step 3. Note that the binding to a name `x` in Step 5 is an internal binding in the closure of `f` and is irrelevant here. Hence, we continue the table as,

Step	Line	Env.	Bindings and evaluations
8	3	E_2	$\big((z = 2.0), \text{a * z}, (a = 3.0, x = 2.0)\big)$

Executing the body of `f`, we initially have 3 bindings available: `z = 2.0`, `a = 3.0`, and `x = 2.0`. Thus, to evaluate the expression `a * z`, we use these bindings and write,

Step	Line	Env.	Bindings and evaluations
9	3	E_2	$a * z = 6.0$
10	3	E_2	return $= 6.0$

The 'return'-word is used to remind us that this is the value to replace the question mark with in Step 7. Here we will make a mental note and not physically replace the question mark with the calculated value. If you are ever in doubt which call is connected with which return value, seek upwards in the table from the return statement for the first question mark. Now we delete E_2 and return to the enclosing environment, E_1. Here the function call was the last expression, hence the return-value from `testScope` will be equal to the return-value from `f`, and we write,

Step	Line	Env.	Bindings and evaluations
11	3	E_1	return = 6.0

Similarly, we delete E_1 and return to the question mark in Step 2, which is replaced by the value 6.0. We can now finish the `printfn` statement and produce the output,

Step	Line	Env.	Bindings and evaluations
12	6	E_0	output = "6.0\n"

The return-value of a `printfn` statement is `()`, and since this line is the last of our program, we return `()` and end the program:

Step	Line	Env.	Bindings and evaluations
13	6	E_0	return = ()

The full table is shown for completeness in Table 1.3. Hence, we conclude that the

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope = $\big((x), \text{testScope-body}, ()\big)$
2	6	E_0	testScope 2.0 = ?
3	1	E_1	$\big((x = 2.0), \text{testScope-body}, ()\big)$
4	2	E_1	$a = 3.0$
5	3	E_1	f = $\big((z), \text{a * z}, (a = 3.0, x = 2.0)\big)$
6	4	E_1	$a = 4.0$
7	5	E_1	f x = ?
8	3	E_2	$\big((z = 2.0), \text{a * z}, (a = 3.0, x = 2.0)\big)$
9	3	E_2	$a * z = 6.0$
10	3	E_2	return = 6.0
11	3	E_1	return = 6.0
12	6	E_0	output = "6.0\n"
13	6	E_0	return = ()

Table 1.3: The complete table produced while tracing the program in Listing 1.6 by hand.

program outputs the value `6.0`, since the function `f` uses the first binding of `a = 3.0`, and this is because the binding of `f` to the expression `a * z` creates a closure with a lexical scope. Thus, in spite that there is an overshadowing value of `a`, when `f` is called, this binding is ignored in the body of `f`. To correct this, we update the code as shown in Listing 1.7.

Listing 1.7 lexicalScopeTracingCorrected.fsx:
Tracing the code in Listing 1.6 by hand produced the table in Table 1.3, and to get the desired output, we correct the code as shown here.

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

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

1.3.2 Tracing Loops

Consider the program in Listing 1.8. The program includes a function for printing

Listing 1.8 printSquares.fsx:
Print the squares of a sequence of positive integers.

```fsharp
let N = 3
let printSquares n =
  for i = 1 to n do
    let p = i * i
    printfn "%d: %d" i p

printSquares N
```

```
$ fsharpc --nologo printSquares.fsx && mono
   printSquares.exe
1: 1
2: 4
3: 9
```

the sequence of the first N squares of integers. It uses a for-loop with a counting value. F# creates a new environment each time the loop body is executed. Thus, to trace this program, we mentally *unfold* the loop as shown in Listing 1.9. The unfolding contains 3 new scopes lines 3–7, lines 8–12, and lines 13–17 corresponding to the 3 times, the loop is repeated, and each scope starts by binding the counting value to the name i.

**Listing 1.9 printSquaresUnfold.fsx:
An unfolded version of Listing 1.8.**

```
let N = 3
let printSquaresUnfold n =
  (
    let i = 1
    let p = i * i
    printfn "%d: %d" i p
  )
  (
    let i = 2
    let p = i * i
    printfn "%d: %d" i p
  )
  (
    let i = 3
    let p = i * i
    printfn "%d: %d" i p
  )

printSquaresUnfold N
```

```
$ fsharpc --nologo printSquaresUnfold.fsx && mono
  printSquaresUnfold.exe
1: 1
2: 4
3: 9
```

In the rest of this section, we will refer to the code in Listing 1.8. The first rows in our tracing-table looks as follows:

Step	Line	Env.	Bindings and evaluations
0	-	E_0	$()$
1	1	E_0	$N = 3$
2	2	E_0	$\text{printSquares} = \big((n), \text{printSquares-body}, (N = 3)\big)$
3	7	E_0	$\text{printSquares } N = ?$

Note that due to the lexical scope rule, the closure of `printSquares` includes `N` in its environment element. Calling `printSquares N` causes the creation of a new environment,

Step	Line	Env.	Bindings and evaluations
4	2	E_1	$\big((n = 3), \text{printSquares-body}, (N = 3)\big)$

The first statement of printSquares-body is the `for`-loop. As our unfolding in Listing 1.9 demonstrated, each time the loop-body is executed, a new scope is created with a new binding to `i`. Reusing the notation from closures, we write

Step	Line	Env.	Bindings and evaluations
5	3	E_1	$\text{for} \cdots = ?$

and create a new environment as if it had been a function,

Step	Line	Env.	Bindings and evaluations
6	3	E_2	$\big((i = 1), \text{for-body}, (n = 3, N = 3)\big)$

As for functions, this denotes the bindings available at beginning of the execution of the `for`-body. The first line in the `for`-body is the binding of the value of an expression to `p`. The expression is `i*i`, and to calculate its value, we look in the `for`-loop's pseudo-closure where we find the $i = 1$ binding. Hence,

Step	Line	Env.	Bindings and evaluations
7	4	E_2	$i * i = 1$
8	4	E_2	$p = 1$

The final step in the for-body is the `printfn`-statement. Its arguments we get from the updated, active environment E_2 and write,

Step	Line	Env.	Bindings and evaluations
9	5	E_2	output = "1 : 1\n"

At this point, the `for`-loop has reached its last line, E_2 is deleted, we create a new environment with the counter variable increased by 1, and repeat. Hence,

Step	Line	Env.	Bindings and evaluations
10	3	E_3	$\big((i = 2), \text{for-body}, (n = 3, N = 3)\big)$
11	4	E_3	$i * i = 4$
12	4	E_3	$p = 4$
13	5	E_3	output = "2 : 4\n"

Again, we delete E_3, create E_4 where i is incremented, and repeat,

Step	Line	Env.	Bindings and evaluations
14	3	E_4	$\big((i = 3), \text{for-body}, (n = 3, N = 3)\big)$
15	4	E_4	$i * i = 9$
16	4	E_4	$p = 9$
17	5	E_4	output = "3 : 9\n"

Finally, incrementing i would mean that $i > n$, hence the `for`-loop ends and as all statements returns `()`

Step	Line	Env.	Bindings and evaluations
18	3	E_4	return = ()

At this point, the environment E_4 is deleted, and we return to the enclosing environment E_1 and the statement or expression following Step 5. Since the `for`-loop is the last expression in the `printSquares` function, its return value is that of the `for`-loop,

Step	Line	Env.	Bindings and evaluations
19	3	E_1	return = ()

Returning to Step 3 and environment E_0, we have now calculated the return-value of `printSquares N` to be `()`, and since this line is the last of our program, we return `()` and end the program:

Step	Line	Env.	Bindings and evaluations
20	3	E_0	return = ()

1.3.3 Tracing Mutable Values

For mutable bindings, the scope is dynamic. For this, we need the concept of storage. Consider the program in Listing 1.10. To trace the dynamic behavior of this program,

**Listing 1.10 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 second table to our hand tracing, which is initially empty and has the columns Step and Value to hold the Step number when the value was updated and the value stored. For Listing 1.10, the firsts 4 steps thus look like,

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope $= \big((x), \text{body}, ()\big)$
2	6	E_0	testScope 2.0 = ?
3	1	E_1	$\big((x = 2.0), \text{body}, ()\big)$

Step	Value
0	-

The mutable binding in line 2 creates an internal name and a dynamic storage location. The name `a` will be bound to a reference value, which we call α_1, and which is a unique name shared between the two tables:

Step	Line	Env.	Bindings and evaluations
4	2	E_1	$a = \alpha_1$

Step	Value
4	$\alpha_1 = 3.0$

The following closure of `f` uses the reference-name instead of its value,

Step	Line	Env.	Bindings and evaluations
5	3	E_1	$f = \big((z), a * z, (x = 2.0, a = \alpha_1)\big)$

Step	Value
4	$\alpha_1 = 3.0$

In line 4, the value in the storage is updated by the assignment operator, which we denote as,

Step	Line	Env.	Bindings and evaluations
6	4	E_1	a <- 4.0

Step	Value
6	$\alpha_1 = 4.0$

Hence, when we evaluate the function `f`, its closure looks up the value of `a` by following the reference and finding the new value:

Step	Line	Env.	Bindings and evaluations
7	5	E_1	f $x = ?$
8	5	E_2	$\big((z = 2.0), a * z, (x = 2.0, a = \alpha_1)\big)$
9	5	E_2	$a * z = 8.0$
10	5	E_2	return = 8.0
10	5	E_1	return = 8.0
11	6	E_0	output = "8.0\n"
12	6	E_0	return = ()

Step	Value
6	$\alpha_1 = 4.0$

For reference, the complete pair of tables is shown in Table 1.4. By comparing this to the value-bindings in Listing 1.6, we see that the mutable values give rise to a different result due to the difference between lexical and dynamic scope.

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope $= \big((x), \text{body}, ()\big)$
2	6	E_0	testScope 2.0 = ?
3	1	E_1	$\big((x = 2.0), \text{body}, ()\big)$
4	2	E_1	$a = \alpha_1$
5	3	E_1	$f = \big((z), a * z, (x = 2.0, a = \alpha_1)\big)$
6	4	E_1	a <- 4.0
7	5	E_1	f x = ?
8	5	E_2	$\big((z = 2.0), a * z, (x = 2.0, a = \alpha_1)\big)$
9	5	E_2	$a * z = 8.0$
10	5	E_2	return = 8.0
10	5	E_1	return = 8.0
11	6	E_0	output = “8.0\n”
12	6	E_0	return = ()

Step	Value
0	-
4	$\alpha_1 = 3.0$
6	$\alpha_1 = 4.0$

Table 1.4: The complete table produced while tracing the program in Listing 1.10 by hand.