CISPA
HELMHOLTZ CENTER FOR
INFORMATION SECURITY

Automated Debugging

WS 2022/2023

Prof. Dr. Andreas Zeller
Paul Zhu
Marius Smytzek

Exercise 2 (10 Points)

Due: 30. November 2022

Submit your solutions as a Zip file on your status page in the CMS.

We will provide you a structure to submit your solutions where each task has a dedicated file. You can add new files and scripts if you want, but you must not delete any provided ones. You can verify whether your submission is valid by calling python3.

```
python3 verify.py
```

The output provides an overview if a required file, variable, or function is missing and if a function pattern was altered. If you do not follow this structure or change it, we cannot evaluate your submission. A non evaluable exercise will result in 0 points, so make sure to verify your work before submitting it. Note that the script does not reveal if your solutions are correct.

Exercise 2-1: Can A Monkey use Our Debugger? (4 Points)

Monkey testing is a technique where the user tests the application or system by providing random inputs and checking the behavior, or seeing whether the application or system will crash. For interactive CLI tools such as debuggers, the user can provide arbitrary input -- imagine a monkey is now using our debugger and he can type anything from the command line -- so robustness should be a great concern.

Unfortunately, the `Debugger` introduced in this chapter is not robust enough (you may already realize it). In this exercise, let's improve the `Debugger` with the following user input checking:

1. for `break {arg}` and `delete {arg}` : report the error message `Expect a line number, but found '{arg}'` if the argument is not a valid Python integer, i.e., `int({arg})` raises `ValueError`
2. for `break {arg}` : report the error message `Line number {arg} out of bound ({start}-{end})` if the argument is outside the valid line numbers of the current function; also print the valid line number range (both inclusive)
3. for `assign {var}={value}` : report the error message `SyntaxError: '{var}' is not an identifier` if the input variable is not a valid Python identifier, i.e., `{var}.isidentifier()` evaluates to `False`
4. for `assign {var}={value}` : report a warning `Warning: a new variable '{var}' is created` if the input variable does not exist in the current frame

Override the corresponding methods (i.e., `break_command` , `delete_command` , and `assign_command`) of the `Debugger` class in `idb.py` . This Python file also comes with a convenient command line usage: `python3 idb.py <File>` -- executing the `debug_main()` function of `<File>` in the debugging mode.

To better demonstrate the desired behavior of your implementation, in the following, we provide a sample interaction on the test program `test.py` . Launch your debugger in command line: `python3 idb.py test.py` . The commands after the `(debugger)` prompt are input by the user.

```
Calling debug_main()
(debugger) break foo
Expect a line number, but found 'foo'
Breakpoints: set()
(debugger) delete foo
Expect a line number, but found 'foo'
```

```
Breakpoints: set()
(debugger) break -1
Line number -1 out of bound (1-10)
Breakpoints: set()
(debugger) step
2     def fib(n: int) -> int:
(debugger) step
                                        # fib = <function debug_main.<locals>.fib at
0x1067851b0>
9     x = fib(2)
(debugger) assign @ = 0
SyntaxError: '@' is not an identifier
(debugger) assign x=1
Warning: a new variable 'x' is created
(debugger) step
Calling fib(n = 2)
(debugger) assign n = 1
(debugger) list
   2>     def fib(n: int) -> int:
   3          if n == 0:
   4              return 0
   5          if n == 1:
   6              return 1
   7          return fib(n - 1) + fib(n - 2)
(debugger) break 8
Line number 8 out of bound (2-7)
Breakpoints: set()
(debugger) quit
```

Exercise 2-2: Where am I? (6 Points)

Knowing where the program has been executed so far is useful in debugging. Call stack is a data structure the keeps track of function calls. In this exercise you will continue extending the `Debugger` class (done in exercise 2-1) with the following commands (none of them take arguments):

1. `where` : print the stack trace (following the Python style, see the example below for the format)
2. `finish` : resume execution until the current function returns
3. `next` : resume execution until the next line (if the next line exists and is reachable in this execution) or the current function returns (if the current function returns at this line, including the case where this line is the last so the function has to return here)

Implement new methods (`where_command` , `finish_command` , and `next_command`) and other auxiliary methods/fields if necessary in class `Debugger` in file `idb.py` to achieve the above functionalities. Make sure you don't break anything already built for exercise 2-1. Hint: the Python interpreter internally maintains a call stack while running user programs, so you may find it helpful to reconstruct the call stack from frames (check the Python document for the `FrameType` class).

Again, we provide sample interactions on the test program `test.py` to better demonstrate the desired behavior of your implementation. The following one only steps through `debug_main()` :

```
Calling debug_main()
(debugger) next
2     def fib(n: int) -> int:
(debugger) next
                                        # fib = <function debug_main.<locals>.fib at
0x1003d9000>
9     x = fib(2)
(debugger) next
                                        # x = 1
10     fib(x)
(debugger) next
debug_main() returns None
(debugger) next
```

As you can see, the `next` command steps **over** function calls `fib(2)` and `fib(x)` , which behaves differently from `step` which will step **into** the called functions.

This one, however, steps into the call of `fib(2)` :

```
Calling debug_main()
(debugger) step
2     def fib(n: int) -> int:
(debugger) step
                                        # fib = <function debug_main.<locals>.fib at
0x106abd1b0>
9     x = fib(2)
(debugger) where
Traceback (most recent call last):
File "<absolute path to exercise_02>/test.py", line 9, in debug_main
  x = fib(2)
(debugger) step
Calling fib(n = 2)
(debugger) where
Traceback (most recent call last):
File "<absolute path to exercise_02>/test.py", line 9, in debug_main
  x = fib(2)
File "<absolute path to exercise_02>/test.py", line 2, in fib
  def fib(n: int) -> int:
(debugger)
```

The call stack, displayed by the `where` command, is changed before and after calling `fib(2)`. You should print the **most recent** call **last**. We provide a class `CallInfo` that encodes the information of such a call frame, and its `__repr__` method returns the formatted string you should output.

```
(debugger) finish
fib() returns 1
(debugger) where
Traceback (most recent call last):
File "<absolute path to exercise_02>/test.py", line 9, in debug_main
  x = fib(2)
File "<absolute path to exercise_02>/test.py", line 7, in fib
  return fib(n - 1) + fib(n - 2)
(debugger) step
                                        # fib = <function debug_main.<locals>.fib at
0x106abd1b0>, x = 1
10     fib(x)
(debugger) where
Traceback (most recent call last):
File "<absolute path to exercise_02>/test.py", line 10, in debug_main
  fib(x)
(debugger) finish
debug_main() returns None
(debugger) next
```

The `finish` command should resume the execution of the current function **until it returns**. For example, the first `finish` command resumes execution until `fib() returns 1`. But at this moment, the current call frame is yet **not** poped from the stack, but this will be done in the **next step** -- note the difference between the outputs of the two `where` commands. The second `finish` command resumes execution until `debug_main() returns None` because `debug_main` is the currently executed function.