# Calico Documentation

***Release 1.0***

**H. Turgut Uyar**

**Jun 28, 2018**

# Contents

Calico is a utility for checking command-line programs in terms of their input and output. It checks whether a program generates the correct output when given some inputs. It was developed to evaluate simple programming assignments in an introductory programming course.

**PyPI** https://pypi.python.org/pypi/calico/

**Repository** https://github.com/itublg/calico

**Documentation** https://calico.readthedocs.io/

Calico has been tested with Python 3.6+. You can install the latest version from PyPI:

```
pip install calico
```

Calico uses pexpect for interacting with the program it is checking. The file that specifies the inputs and outputs for the checks is in YAML format.

CHAPTER 1

# Contents

## 1.1 Tutorial

Let's assume that you want to use Calico for checking a programming assignment. The assignment is to write a C program that will get the radius of a circle from the user, calculate its area and print the result. A simple implementation could be the following code, saved in the file `circle.c`:

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    float radius, area;

    printf("Enter radius of circle: ");
    scanf("%f", &radius);
    area = 3.14159 * radius * radius;
    printf("Area: %f\n", area);
    return 0;
}
```

### 1.1.1 Basics

To run Calico, we have to write a test specification file. This specification consists of a sequence of stages where a stage is a test or a setup operation. For each stage, the basic information that needs to be supplied is what command to run and what it should return. For example, if we want to check whether this source file can be compiled, we can write the following specification:

```
- compile:
    run: gcc -c circle.c -o circle.o
    exit: 0
```

This specification states that there is only one stage. The name of the stage is "compile". The command is to run the source code through the gcc compiler, and the compiler should exit with the status code 0 (success).

Save this specification in the file `circle.yaml` in the same directory as the `circle.c` file. Note that the run command requires that the C file has to be in the current directory. Now, if you run `calico circle.yaml`, you see the following output:

```
compile ................................. PASSED
Grade: 0 / 0
```

This stage will create a `circle.o` file in the current directory as a result of the run command.

If we want to do grading, we can assign points to stages. If a stage with points passes, Calico will print those points in the report. Also note that we can leave out the `exit: 0` clause because successful completion is the default desired outcome:

```
- compile:
    run: gcc -c circle.c -o circle.o
    points: 10
```

In this case, the output will be:

```
compile ................................. 10 / 10
Grade: 10 / 10
```

### 1.1.2 Blockers

As our next step, let's check whether the compiled object file can be linked. We add a second stage to our specification:

```
- compile:
    run: gcc -c circle.c -o circle.o
    points: 10

- link:
    run: gcc circle.o -o circle
    points: 20
```

The stages are executed in order and the output is:

```
compile ................................. 10 / 10
link .................................... 20 / 20
Grade: 30 / 30
```

However, it doesn't make sense to try to the link the object file if the source file could not be compiled and the object file was not generated. If a stage is marked as a blocker, all subsequent stages will be cancelled if that stage fails.

```
- compile:
    run: gcc -c circle.c -o circle.o
    blocker: true
    points: 10

- link:
    run: gcc circle.o -o circle
    points: 20
```

If you introduce an error into the C code and run Calico again, you'll see:

```
compile ................................. 0 / 10
Grade: 0 / 30
```

As seen in the example, Calico will count the points of failed and cancelled stages in the total points calculation.

### 1.1.3 Interacting with the program

If the compile and link stages are successful, we'll have an executable (in the file `circle` as a result of the link command) that we can run for I/O checking. So let's write a stage to test whether it produces the correct output for a simple case:

```
- compile:
    run: gcc -c circle.c -o circle.o
    blocker: true

- link:
    run: gcc circle.o -o circle
    blocker: true

- case_1:
    run: ./circle
    script:
      - expect: "Enter radius(.*?):\s+"
      - send: "1"
      - expect: "Area: 3.14(\d*)\r\n"
      - expect: _EOF_
    points: 10
```

First of all, note the changes in the compile and link stages. Both of these stages are blockers and we have decided not to assign points to them. To describe the interaction with a program, we supply a script, which is a sequence of expect/send operations. An expect operation expects the given output from the program and a send operation sends a user input to the program. Expected output is given as a regular expression and user input is given as a simple string.

In the example, the script first expects a prompt for entering the radius, then sends the string "1" (as if the user typed it in). Next, it expects that the program prints a message that contains the correct area for that input. Finally it expects to program to terminate without requiring further user input.[1] Running Calico now prints:

```
compile ................................. PASSED
link .................................... PASSED
case_1 .................................. 10 / 10
Grade: 10 / 10
```

A stage that doesn't have a script is assumed to be non-interactive and it consists of a single step where it expects the program to terminate.

Say that if the user types in a negative radius value we want to program to exit with a failure code. For that, we can use the exit status setting:

```
- case_negative:
    run: ./circle
    script:
      - expect: "Enter radius(.*?):\s+"
      - send: "-1"
      - expect: _EOF_
    exit: 1
    points: 30
```

To make that test pass, the C file can be modified as follows:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    float radius, area;
```

(continues on next page)

[1] `_EOF_` is a marker for end-of-file and expecting `_EOF_` means expecting program termination.

(continued from previous page)

```
    printf("Enter radius of circle: ");
    scanf("%f", &radius);

    if (radius < 0) {
        fprintf(stderr, "Negative radius values are not allowed.\n");
        exit(1);
    }

    area = 3.14159 * radius * radius;
    printf("Area: %f\n", area);
    return 0;
}
```

### 1.1.4 Debug mode

Running in debug mode will show you what's going on between Calico and the tested program. Type the command `calico --debug circle.yaml` and you get the following output:

```
starting test compile
running command: gcc -c circle.c -o circle.o
  expecting: _EOF_
  received: _EOF_
exit status: 0 (expected 0)
compile ................................... PASSED
starting test link
running command: gcc circle.o -o circle
  expecting: _EOF_
  received: _EOF_
exit status: 0 (expected 0)
link ...................................... PASSED
starting test case_1
running command: ./circle
  expecting: "Enter radius(.*?):\s+"
  received: "Enter radius of circle: "
  sending: 1
  expecting: "Area: 3.14(\d*)\r\n"
  received: "Area: 3.141590
"
  expecting: _EOF_
  received: _EOF_
exit status: 0 (expected 0)
case_1 .................................... 10 / 10
Grade: 10 / 10
```

### 1.1.5 Timeouts

It's possible that the tested program goes into an infinite loop or takes too long to respond. For such cases, we would like to limit the amount of time Calico should wait. Expect steps can have timeout comments that make this possible:

```
- case_1:
    run: ./circle
    script:
      - expect: "Enter radius(.*?):\s+"
      - send: "1"
      - expect: "Area: 3.14(\d*)\r\n"      # timeout: 2
```

(continues on next page)

(continued from previous page)

```
      - expect: _EOF_
    exit: 0
    points: 10
```

In this example, after sending the user input, Calico will wait 2 seconds for the program to print the area. If the program doesn't respond in that time, the stage will fail. To test it, add a sleep statement to the C code and run Calico in debug mode:

```
starting test case_1
running command: ./circle
  expecting: "Enter radius(.*?):\s+"
  received: "Enter radius of circle: "
  sending: "1"
  expecting (2s): "Area: 3.14(\d*)\r\n"
  received: ""
FAILED: Timeout exceeded.
case_1 ..................................... 0 / 10
```

Run commands can also have timeout comments if the stage doesn't have a script. In that case Calico will expect the program to terminate within that time frame. If the stage has a script, the timeout comment for the run command will be ignored. Timeout comments for other items such as send steps also have no effect.

### 1.1.6 Hidden stages

To make sure that there are no object and executable files left over from earlier runs, let's add an initialization stage to delete these generated files. But we don't want this stage to be included in the report, so we mark it as not visible:

```
- init:
    run: rm -f circle.o circle
    visible: false

- compile:
    ...
```

Running Calico will leave out the "init" stage:

```
compile ..................................... PASSED
link ........................................ PASSED
case_1 ...................................... 10 / 10
case_negative ............................... 30 / 30
Grade: 40 / 40
```

Running in debug mode will include the details about the hidden stages.

### 1.1.7 Defining variables

In scripts, in order not to repeatedly type the same data -such as expected prompts- you can define variables in the special `_define` section:

```
- _define:
    vars:
      prompt: "Enter radius(.*?):\\s+"

- case_1:
    run: ./circle
    script:
      - expect: %(prompt)s
```

(continues on next page)

(continued from previous page)

```
      - send: "1"
      ...

- case_2:
    run: ./circle
    script:
      - expect: %(prompt)s
      - send: "0"
      ...
```

**Note:** To make the specification file shorter, you can use the following shortcuts for the keywords: `r` for `run`, `e` for `expect`, `s` for `send`, `x` for `exit`, `b` for `blocker`, `v` for `visible`, `p` for `points`.

### 1.1.8 Jailing tests

To prevent the tested program from damaging the system, Calico runs the stages in a restricted environment, if possible. For this to work, the "fakechroot" command has to be in the path of executables. If this command is available, stages that have names starting with "case_" will be jailed to the directory in which they are run and cannot access files in upper directories.

CHAPTER 2

# Indices and tables

- genindex
- search