Homework Sheet 7, Ex. 2

Büchler P., Lohmann N.J

Institute for Software Systems

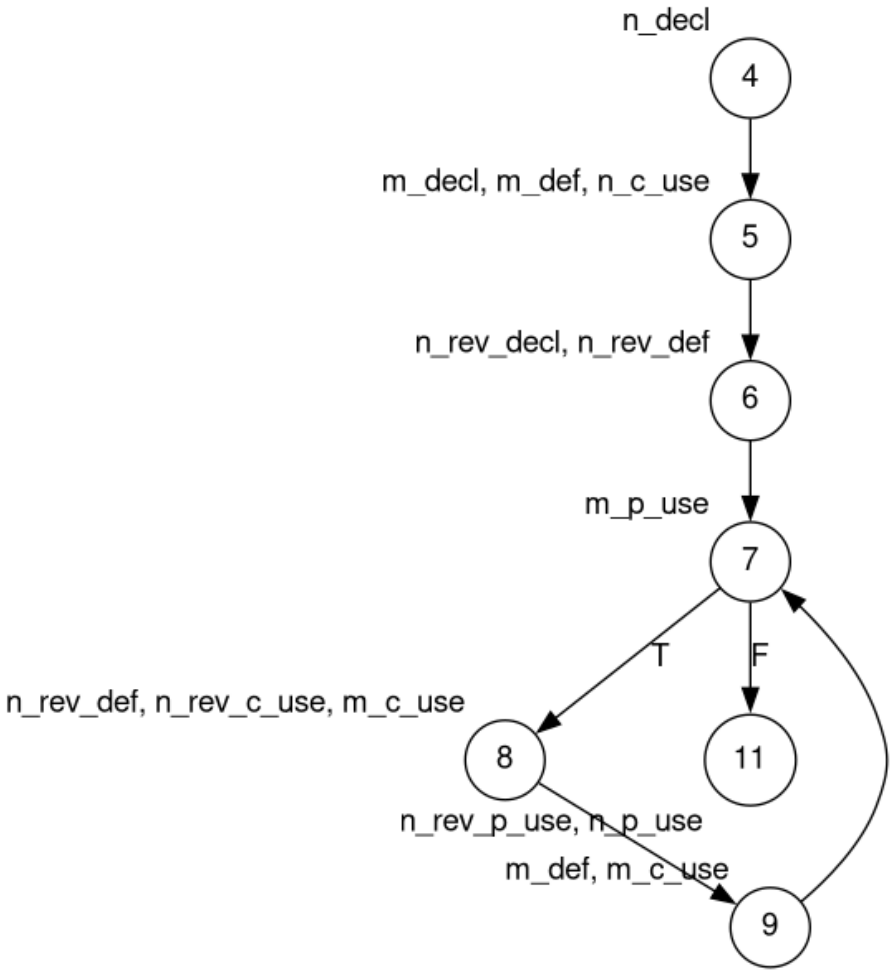
June 29, 2023

Contents

Palindrome

```
1 #include <stdio.h>
2 #include <stdbool.h>
3
4  bool is_palindrome(unsigned int n) {
5    unsigned int m = n;
6    unsigned int n_rev = 0;
7    while (m > 0) {
8        n_rev = (n_rev*10) + (m % 10);
9        m = m / 10;
10   }
11   return n_rev == n;
12 }
```

Data Flow Graph



DU-Paths (n)

- 5-11

DU-Paths (m)

- 5-6-7
- 5-6-7-8-9
- 9-7-8

DU-Paths n_rev

- 6-7-8
- 6-7-11
- 8-9-7
- 8-9-7-11

Test Suite

Test	#1
Test Path	4-5-6-7-11
Coverage	4-11, 5-6-7, 6-7-11
Input	n = 0
Output	true

Test	#2
Test Path	4-5-6-7-8-9-7-8-9-7-11
Coverage	4-11, 5-6-7, 5-6-7-8-9, 8-9-7, 8-9-7-11, 9-7-8
Input	n = 11
Output	true

Comparison Data Flow vs. Branch coverage

For 100% multiple-condition coverage, $c = 1 \implies 2^c = 2$
Matches exactly the number for covering all DU-paths. This doesn't have to be the case always. The last statement (returning the bool) is unusual as it doesn't change the branching of the program, but very much its behavior.

Practice

But how does this actually look like? After all our Data Flow graphs are fortunately made for us humans and hence subjective!, but a compiler needs to be precise. DU-paths (commonly known as Def-Use chains) are too complex to generate but go in the right direction. The next step are Static Single-Assignment form (SSA) derived from the CFG.

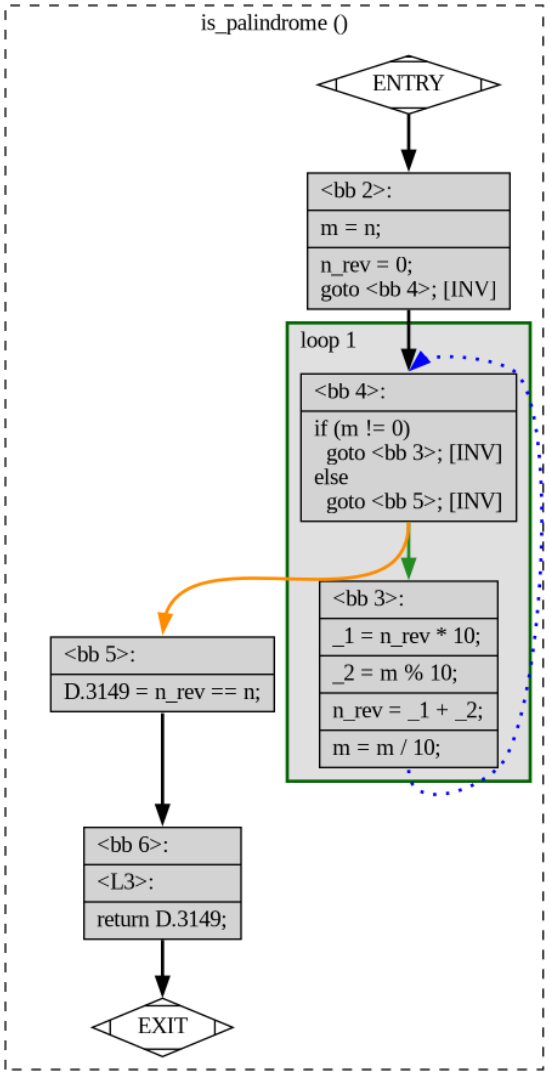
One can generate quiet helpful information that the compiler *gcc* $\geq$ 4.0 uses via

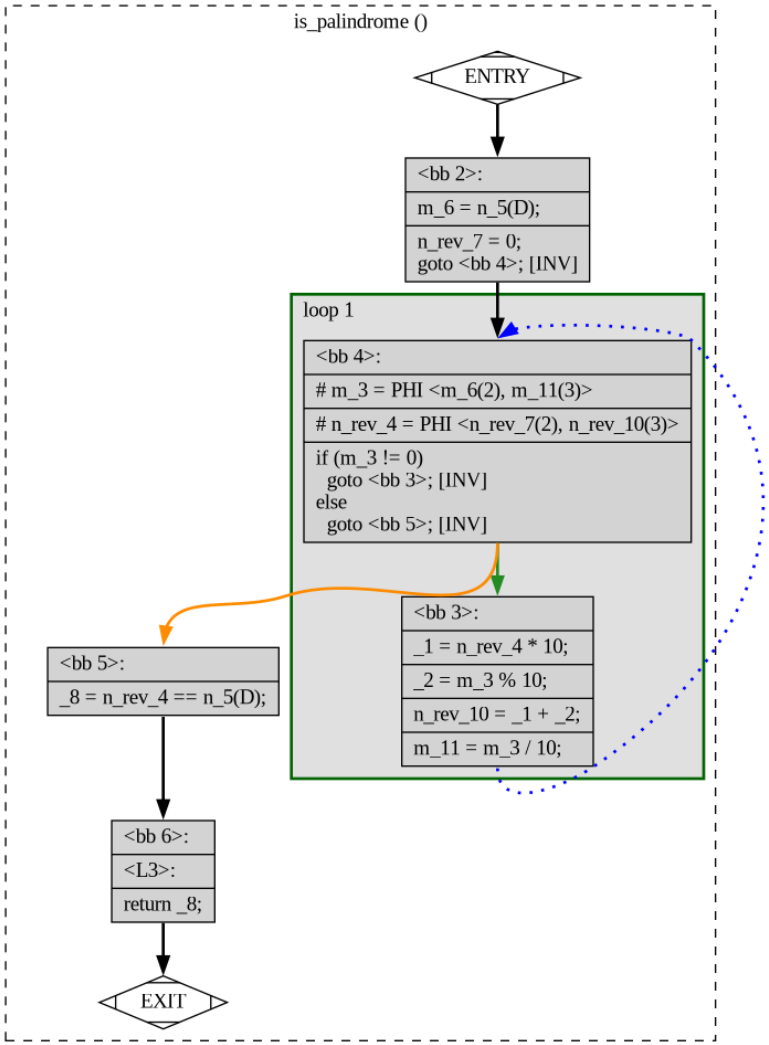
```
gcc palindrome.c -fdump-tree-all-graph
```

Most importantly

```
palindrome.c.cfg
```

```
palindrome.c.ssa
```


is_palindrome ()
ENTRY
<bb 2>:
m = n;
n_rev = 0;
goto <bb 4>; [INV]
loop 1
<bb 4>:
if (m != 0)
goto <bb 3>; [INV]
else
goto <bb 5>; [INV]
<bb 3>:
_1 = n_rev * 10;
_2 = m % 10;
n_rev = _1 + _2;
m = m / 10;
<bb 5>:
D.3149 = n_rev == n;
<bb 6>:
<L3>:
return D.3149;
EXIT


is_palindrome ()
ENTRY
<bb 2>:
m_6 = n_5(D);
n_rev_7 = 0;
goto <bb 4>; [INV]
loop 1
<bb 4>:
# m_3 = PHI <m_6(2), m_11(3)>
# n_rev_4 = PHI <n_rev_7(2), n_rev_10(3)>
if (m_3 != 0)
goto <bb 3>; [INV]
else
goto <bb 5>; [INV]
<bb 3>:
_1 = n_rev_4 * 10;
_2 = m_3 % 10;
n_rev_10 = _1 + _2;
m_11 = m_3 / 10;
<bb 5>:
_8 = n_rev_4 == n_5(D);
<bb 6>:
<L3>:
return _8;
EXIT

Coverage

How about the statement/decision coverage? Really dependent on the languages ecosystem and most are happy with statement coverage. For c there is gcov available. Running

```
#include <stdbool.h>
#include <assert.h>

bool is_palindrome(unsigned int n) {
    unsigned int m = n;
    unsigned int n_rev = 0;
    while (m > 0) {
        n_rev = (n_rev*10) + (m % 10);
        m = m / 10;
    }
    return n_rev == n;
}

int main() {
    assert(is_palindrome(0) == true);
    assert(is_palindrome(11) == true);
    return 0;
}
```

```
$ gcc -Wall -fprofile-arcs -ftest-coverage palindrome_test.c
$ ./a.out
$ gcov -c palindrome_test.c
```

```
File 'palindrome_test.c'
Lines executed:100.00% of 11
Branches executed:100.00% of 6
Taken at least once:66.67% of 6
Calls executed:50.00% of 4
Creating 'palindrome_test.c.gcov'

Lines executed:100.00% of 11


function is_palindrome called 2 returned 100% blocks executed 100%
        2:    4:bool is_palindrome(unsigned int n) {
        2:    5:    unsigned int m = n;
        2:    6:    unsigned int n_rev = 0;
        4:    7:    while (m > 0) {
branch  0 taken 2
branch  1 taken 2 (fallthrough)
        2:    8:        n_rev = (n_rev*10) + (m % 10);
        2:    9:        m = m / 10;
        -:   10:    }
        2:   11:    return n_rev == n;
        -:   12:}
        -:   13:
```

The End

Thank you for your attention!