# Lab Work – Playing with Dot[1]

Name: ______________________________ ID: ______________

1. This question refers to the program below[2]:

```
#include <stdio.h>
int main(int argc, char** argv) {
  char* s = "My string";
  if (argc % 2) {
    s[0] = 'm';
  }
  printf("[%s]\n", s);
}
```

   Assuming the program is stored in a file `file1.c`, you can compile it with the following command:

```
$> clang file1.c -o file1
```

   (a) What happens if you run the program in these different ways below?

      i. `$> ./file1; echo $?`

      ii. `$> ./file1 a; echo $?`

   (b) Why one of the executions terminates with an error? What is wrong with the program above? Perhaps you would like to generate its assembly version, e.g., `clang -S file.c -o file.s`. Try to imagine how the string pointed by `a` is stored in memory.

   (c) Experiment compiling the program with the following command line:

```
$> clang -fwritable-strings file1.c -o file1
```

      i. Do you obtain the same error as before?

      ii. What can you guess about the flag `fwritable-strings`?

[1] The material necessary for this assignment is available at `http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/lab/exercises/PlayingDot.tgz`

[2] If you do not want to type the program, the examples are usually available in the course's web page.

2. Dot is a format to describe graphs, which is used by many tools. Nowadays, several compilers use dot as a standard output format, which helps in program debugging and understanding. LLVM uses dot in a number of situations. In this exercise, we will take a look into the *Control Flow Graph* of a program, which LLVM outputs as a dot graph. Consider, as an example, the program below:

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

#define BUF_SIZE 80

int main(int argc, char** argv) {
  int i;
  int index = 0;
  char *buf = (char*)malloc(BUF_SIZE);
  for (i = 0; i < argc; i++) {
    int j;
    int lim = strlen(argv[i]);
    for (j = 0; j < lim; j++) {
      if (index < BUF_SIZE - 1) {
        buf[index] = argv[i][j];
        index++;
      } else {
        break;
      }
    }
  }
  buf[index] = '\0';
  printf("%s\n", buf);
}
```

(a) In what follows, lets us assume that the source code of the program above is stored in a file `file2.c`. Execute the following commands[3]:

```
$> clang -c -emit-llvm file2.c  -o file2.bc
$> opt -view-cfg file2.bc
```

(b) A control flow graph is made of *basic blocks*. Which criterion determines the beginning and the end of a basic block?

(c) Our example program has several basic blocks with only one instruction, e.g., `br label %XX`. Such basic blocks contain only a jump, but no payload, i.e., instructions that perform actual computation. Why does LLVM creates this "empty" basic blocks?

[3] If you can open displays in your environment, `view-cfg` should give you a window produced via `dotty` or `graphviz`. Otherwise, it will produce a dot file in a temporary folder. You can copy that file, and open it locally, using, for instance, `dot -Tpdf file.dot -o file.pdf; evince file.pdf`

(d) Try preprocessing the file, with the following commands:

```
$> opt -instnamer file2.bc -o file2.new.bc
$> opt -view-cfg file2.new.bc
```

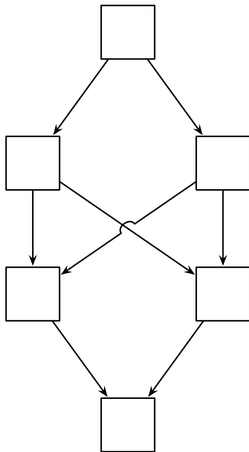
What is the difference between the CFGs of `file2.bc` and `file2.new.bc`? What does the flag `instnamer` do?

(e) Now, try preprocessing `file2.new.bc` with the following command:

```
$> opt -mem2reg file2.new.bc -o file2.reg.bc
$> opt -view-cfg file2.reg.bc
```

What is the difference between `file2.new.bc` and `file2.reg.bc`? What does the `mem2reg` flag do?

(f) Below we see a graph pattern called *The Butterfly*. Usually the CFG of structured programs do not have this pattern. Could you code a simple C program that contains it? Feel free to use any command available in the syntax of C.



3. Control flow graphs that we built out of C programs usually have the *Single-Entry, Single-Exit* (SESE) property. SESE regions are usually called *hammock* graphs.

**definition 0.1 (Hammock Graph - Ferrante'87)** *If $G$ is a CFG, then a hammock $H$ is an induced subgraph of $G$, so that $H$ contains a node $V$ called the entry, and $G - H$ contains a node $W$, with the following properties:*

- *all edges from $G - H$ to $H$ go to $V$.*
- *all edges from $H$ to $G - H$ go to $W$.*

We can use LLVM to visualize the SESE regions in the CFG of a program. To this end, lets consider the program below:

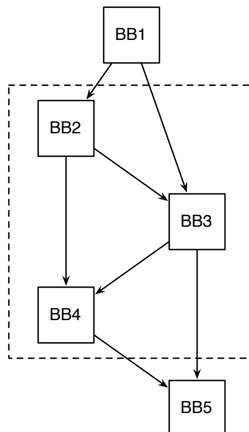
```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 0;
  do {
    char* p;
    for (p = argv[i]; *p != '\0'; p++) {
      if (*p != '\n') {
        printf("%c", *p);
      }
    }
    printf("\n");
    i++;
  } while (i < argc);
  return 0;
}
```

(a) Use opt to visualize the hammock regions in this program. Assuming your code is in `file3.c`, you can perform this visualization with the following commands:

```
$> clang -c -emit-llvm file3.c -o file3.bc
$> opt -view-regions file3.bc
```

(b) Lets assume that every *branch* in the CFG is the starting point $V$ of a hammock region. In this case, what do you think is the algorithm used to find these regions? You may try to visualize regions for different programs, in order to determine a more precise algorithm. If you feel like using the right names, search for the notion of *post-dominance*.

(c) It is possible to create programs containing branches which are not the starting point of hammock regions. The butterfly of the previous question is an example. Another example is given below:



In this case, the block BB2 starts a region which is not hammock. In this exercise, you must create a program that gives origin to a non-hammock region. **You cannot use the command goto**.

4. Now, let's raise the level of our toils a little bit. Instead of playing with bytecodes, let's take a look into the source code of a program. To this end, use the command below to see the AST of our first example:

```
$> clang34 -cc1 -ast-view file1.c
```

If all works well, you probably are seeing something similar to the graph in Figure 1.

   (a) When is it better to perform analyses and optimizations in the high-level representation of the program, i.e., on its AST?

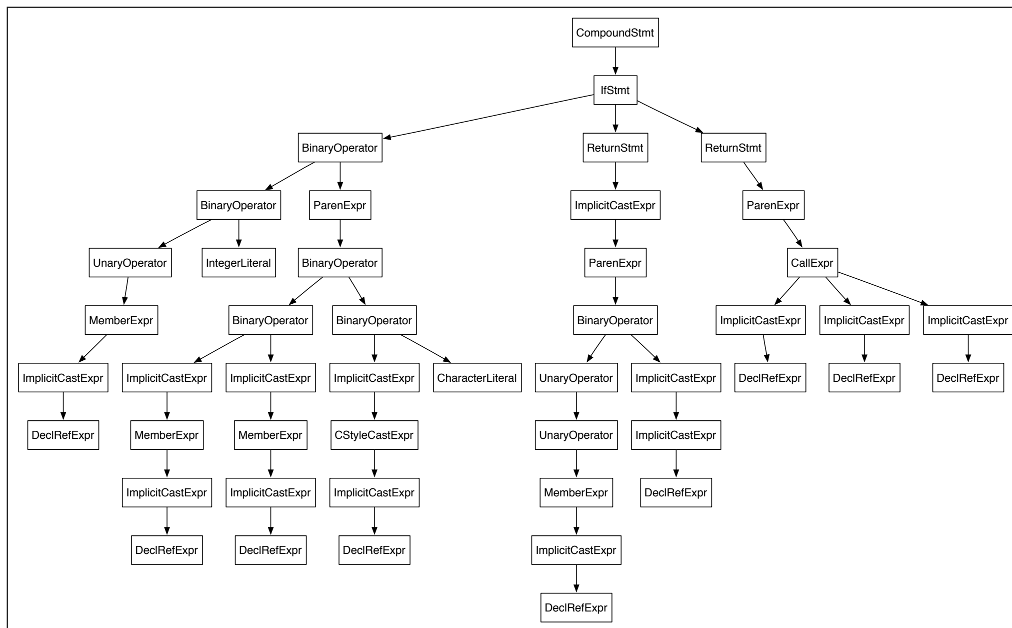



Figure 1: Abstract Syntax Tree.