The goal of this homework is two-fold:

1. To help you understand how passes are implemented and run in llvm,
2. To asses your understanding by introducing the second project.

Sources / Inspiration Here are some of the articles used for this homework:

- https://blog.regehr.org/archives/1603: The article used in class, a great, detailled explanation of how multiple passes are run on a small program.
- https://www.cs.cmu.edu/afs/cs/academic/class/15745-s14/public/lectures/L6-LLVM-Part2.pdf: A good introduction to llvm passes, contain a brief list of "useful passes" and some high-level insights on llvm's structure.
- https://www.incredibuild.com/blog/compiling-with-clang-optimization-flags: A good description of llvm's levels of optimization, with some good examples.
- https://www.intel.com/content/www/us/en/developer/articles/technical/optimizing-llvm-code-generation-for-data-analytics.html: Contains a detailled discussion on vectorization optimization.
- http://jonathan2251.github.io/lbt/opt.html: Discusses how to perform optimization "accross files". Insightful, but outside the scope of this homework.
- https://llvm.org/docs/LoopTerminology.html: A great, detailled, explanation of how loops are treated by llvm.

Disclaimer: llvm evolves *a lot*, and even if some resources are extremely useful as a lead, they may be outdated and in dire need of adaptation to work with your verison of llvm!

1 LLVM's Passes

1.1 List and Organization

You can use

- `opt -help`,
- `opt -print-passes`
- or https://llvm.org/docs/Passes.html.

To access the list of passes on your installation (for the first two commands) or in the latest version of llvm (for the on-line documentation). Observe that there are roughly 3 types of passes:

- Analysis Passes,
- Transform Passes,
- Utility Passes,

that are further sub-divided into "Loop analyses", "Function passes", "Module passes", etc.

1.2 Demo

Consider the (very uselessly convoluted) program demo.c:

```
int demo(int n) {
  int x, y;
  for (int i = 12; i < n - 1; i += 2) {
    x = 1 + 2;
    y = x + 1 + 2;
  }
  if (x > y) {
    return 0;
  } else {
    return 1;
  }
}
```

1. Start by reading the code carefully and explaining at a high-level what it does, and which value it will return.
2. Compile this program with a high level of optimization, using

   ```
   clang demo.c -S -emit-llvm -Os -o demo_optim.ll
   ```

   and observe the output.
3. Compile this program with a low level of (more or less "no") optimization, using

   ```
   clang demo.c -S -emit-llvm -O0 -o demo_non_optim.ll
   ```

   and observe the output.
4. Use

   ```
   opt -Os demo_non_optim.ll -time-passes -mem2reg -instcount -stats -S
   ```

   to observe which passes are run on the demo intermediate representation at level "s" of optimization. Explain what each flag does in brief terms.
5. Open demo_non_optim.ll, and remove the "optnone" attribute from the function attributes, and run again the previous command. Notice the difference in the number of passes run.

1.3 Brief Digression

What happened is that `-O0` is interpreted by clang as "do not run any optimization *ever*", hence the "optnone" attribute added to our function attributes, that prevents most passes from being run on that particular function. Instead of editing that file by hand every time we want to re-allow optimization, we can construct the intermediate representation using

```
clang demo.c -S -emit-llvm -O0 -Xclang -disable-O0-optnone -o demo_non_optim.ll
```

so that the "optnone" attribute is not added[1].

Sources: https://stackoverflow.com/a/56564938, https://stackoverflow.com/a/49093344, https://stackoverflow.com/q/46513801, https://groups.google.com/g/llvm-dev/c/jrbvGDeqstk, https://stackoverflow.com/a/62853265.

1.4 First Example: Analysis

Let us run the "-dot-cfg" pass that "creates 'dot' file that represent the CFG of the analyzed program." on our example.

1. Make sure you have the intermediate representation of our file. If needed, re-run

   ```
   clang demo.c -S -emit-llvm -O0 -o demo_non_optim.ll -Xclang -disable-O0-optnone
   ```

2. Make sure the "optnone" attribute*s* (they are, normally, twice such attributes) are *not* present in the .ll file. Remove them if they are.
3. Run

   ```
   opt -dot-cfg -S demo_non_optim.ll
   ```

   to create a ".test.dot" file.
4. Since files starting with a dot are by default hidden, you may not "see" it with your window explorer, but you should be able to open it with any text editor.
5. "Reading" the dot file is nice, but "seeing" it is even better! Normally, a simple

   ```
   dot -Tpng .test.dot -o test.png
   ```

   should give you Figure 1. Isn't that neat? If you obtained an error message about dot not being found, simply install the graphviz package using e.g. `sudo apt-get install graphviz`.

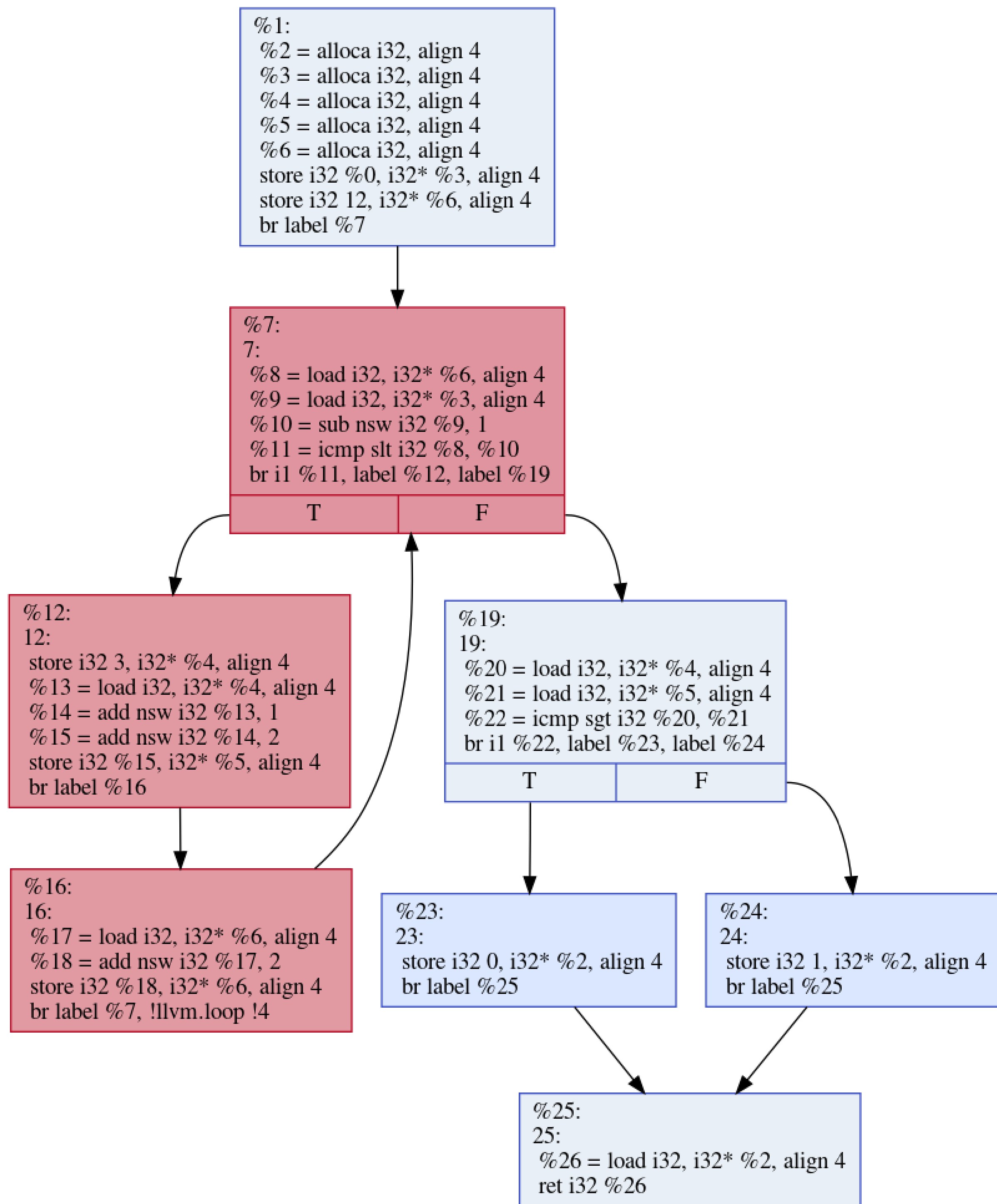

Figure 1: Control Flow Graph for a Simple "test" function

While visualizing this representation is not useful for the compiler *per se*, the CFG is used e.g. when running simplifyCFG that uses this representation to simplify the program's structure.

Sources: `https://stackoverflow.com/a/67578423`, `https://stackoverflow.com/a/1494527`.

1.5 Second Example: Optimization with Loop Rotation

Let us now work on an example of optimization (or, to be more precise, of "canonicalization" of the intermediate representation). Loop rotation is precisely documented at `https://llvm.org/docs/LoopTerminology.html#loop-terminology-loop-rotate` and easy to observe. Running

```
opt -S -loop-rotate -mem2reg -print-before=loop-rotate -print-after=loop-rotate
↪  demo_non_optim.ll
```

will display the intermediate representation *before* and *after* the "loop-rotate" optimization is performed, given next. We also present the control flow graph of this "rotated" loop test in Figure 2.

[1] Somehow, this seems to not always work as expected, so please double-check!

Before:

```
*** IR Dump Before LoopRotatePass on Loop at depth 1 containing:
 ↪ %7<header><exiting>,%12,%16<latch> ***

; Preheader:
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 12, i32* %6, align 4
  br label %7

; Loop:
7:                                                ; preds = %16, %1
  %8 = load i32, i32* %6, align 4
  %9 = load i32, i32* %3, align 4
  %10 = sub nsw i32 %9, 1
  %11 = icmp slt i32 %8, %10
  br i1 %11, label %12, label %19

12:                                               ; preds = %7
  store i32 3, i32* %4, align 4
  %13 = load i32, i32* %4, align 4
  %14 = add nsw i32 %13, 1
  %15 = add nsw i32 %14, 2
  store i32 %15, i32* %5, align 4
  br label %16

16:                                               ; preds = %12
  %17 = load i32, i32* %6, align 4
  %18 = add nsw i32 %17, 2
  store i32 %18, i32* %6, align 4
  br label %7, !llvm.loop !4

; Exit blocks
19:                                               ; preds = %7
  %20 = load i32, i32* %4, align 4
  %21 = load i32, i32* %5, align 4
  %22 = icmp sgt i32 %20, %21
  br i1 %22, label %23, label %24
```

After:

```
*** IR Dump After LoopRotatePass on Loop at depth 1 containing:
↪ %7<header><exiting>,%12,%16<latch> ***

; Preheader:
.lr.ph:                                           ; preds = %1
  br label %11

; Loop:
11:                                               ; preds = %.lr.ph, %15
  store i32 3, i32* %4, align 4
  %12 = load i32, i32* %4, align 4
  %13 = add nsw i32 %12, 1
  %14 = add nsw i32 %13, 2
  store i32 %14, i32* %5, align 4
  br label %15

15:                                               ; preds = %11
  %16 = load i32, i32* %6, align 4
  %17 = add nsw i32 %16, 2
  store i32 %17, i32* %6, align 4
  %18 = load i32, i32* %6, align 4
  %19 = load i32, i32* %3, align 4
  %20 = sub nsw i32 %19, 1
  %21 = icmp slt i32 %18, %20
  br i1 %21, label %11, label %._crit_edge, !llvm.loop !4

; Exit blocks
._crit_edge:                                      ; preds = %15
  br label %22
```

2 Project #2

This project aims at improving your understanding of llvm's passes. As llvm is an evolving and complex technology, it can be frustrating and more difficult than you expect, but it is my hope that it will engage you in "playing" with llvm in a deep, non-trivial manner. I reserve the right to grade this project to up to 60 points if you either

- Study multiple passes,
- Observe and details the dependencies of passes,
- Invent particularly clever examples to illustrate in a straightforward manner how a pass work.

The deliverable must be a textual document including, clearly indicated, your name, the pass(es) you are interested in, and the sources you used. Feel free to share .ll or .c (or any type of source, really) files separately on top of including the relevant aspects in your document. Use proper grammar and syntax, but the format does not matter: a simple .txt file would do just fine.

The suggested workflow is the following:

1. Pick a pass (refer to the first section for ways to obtain lists of passes),
2. Explain its "theory":
   (a) What does it do? Does it simply enable other passes (if yes, which ones?), actually "improve" the representation with some respect, something else entirely?
   (b) Why is it correct?
   (c) What are the conditions for its application?
3. Write two programs (feel free to share the source code – which can be in C or any other language supported by llvm – and / or the intermediate representation):

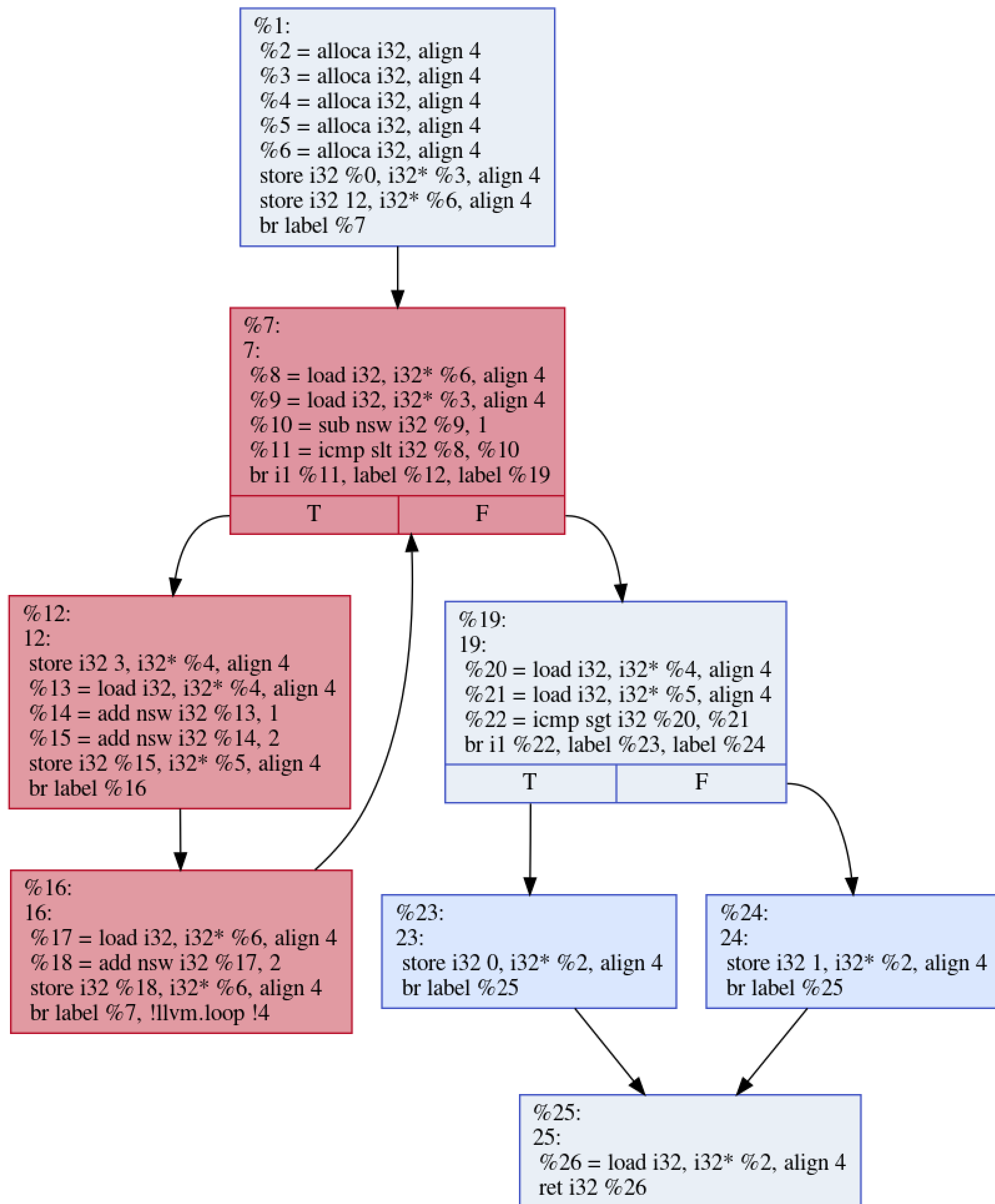

Figure 2: Control Flow Graph for a Simple “rotated” “test” function

- If the pass you have selected transforms the intermediate representation,
  (a) One program where the pass is applied and performs a transformation,
  (b) One program where the pass cannot be applied, along with the explanation of why it cannot be applied,
- If the pass you have selected does not transform the intermediate representation:
  (a) One program along with the output from the pass,
  (b) One program that is a further optimization of the previous program, along with the output from the pass.

To get started, you can e.g. use the following loop1.c code:

```
/*
 * Compiler Writing - CSCI 4800 / CSCI 6800
 * Fall 2021
 * Clément Aubert
 * File under https://creativecommons.org/licenses/by/4.0/
 */

#include <stdio.h>

#include <stdlib.h>

/*
 * This small program serves to illustrate some
 * of llvm's loop optimization and analysis.
 * Code inspired from
 * https://llvm.org/docs/LoopTerminology.html
 */

int main() {
  int c = 0;
  int x1, x2;
  for (int i = 0; i < 10; i++) {
    c = 0;
    if (c)
      x1 = 5;
    else
      x1 = 15;
  }

}
```

Compile it, using

```
clang loop1.c -S -emit-llvm -O0 -Xclang -disable-O0-optnone -o loop1_non_optim.ll
```

and then get detailled information of what e.g. the lcim pass does using

```
opt -licm loop1_non_optim.ll -print-after=licm -time-passes -mem2reg  -S -stats
```