This homework revolves around installing, testing and studying *the LLVM compiler infrastructure*. You are more than welcome to ask me directly if you are facing any difficulty, the instructions are written as if nothing was ever going to get wrong, ever.

1 Installing LLVM

Depending on your operating system, this process can be easy or … less easy. You can jump to the second section of this homework to first test if LLVM is already installed (something fairly likely if you are using a Mac), and come back to this section if it is not.

If you are adventurous, you can build LLVM from source no matter which operating system you are using, but this is longer and possibly more error-prone.

On Ubuntu or Debian: Simply visit `https://apt.llvm.org/` and follow the instructions. On most systems, a simple (as root)

```
bash -c "$(wget -O - https://apt.llvm.org/llvm.sh)"
```

should do.

On Windows: You can (in *supposed* increasing order of difficulty):

- use the Windows installer
- use the Chocolatey package manager to install the LLVM package, and then install some additional python packages,
- go through the (long) tutorial to install LLVM in Visual Studio.

If you want to integrate LLVM into visual studio, you can use the LLVM Compiler Toolchain Visual Studio extension. Note that if you have the WSL installed, installing LLVM from it may be the simplest option.

On MacOS: I would recommend using the homebrew formula. You can refer to this tutorial for more insights. I would also recommend using the full path (`/usr/local/opt/llvm/bin/clang`) instead of trying to replace your mac version of clang.

2 Testing LLVM

1. First, test if LLVM is installed *and* in the path. Open a command-line interface, and type `clang -v`, and if it does not work, try
   - `clang-13 -v`,
   - `/usr/local/opt/llvm/bin/clang -v` (on mac),
   - `C:\Tools\llvm\bin\Release\clang.EXE -v` (on windows)
   - or variations on those.
2. Then, follow the "Example with clang". Note that invoking `lli` (and `llvm-dis`, `llc`, and, later on, `opt` alike) can require to actually invoke `lli-13` or to similarly give the complete path to help your system finding the executable for lli, that directly executes programs from LLVM bitcode. When compiling the `s` file using gcc, you may have to add the `-no-pie` or `-fPIC` flag to produce the `hello.native` file, that you can execute using `./hello.native`.

3 What Just Happened? - Short Questions

Now, you need to understand what you just did. The following short questions should help you in this task. You can use `man clang` to open the manual of clang, or consult an on-line version, to find some answers.

1. In `clang hello.c -o hello`, what is the argument? What does the `-o` flag do?
2. In `clang -O3 -emit-llvm hello.c -c -o hello.bc`, what does `-O3` do?
3. Regarding the comment

   > The `-emit-llvm` option can be used with the `-S` or `-c` options to emit an LLVM `.ll` or `.bc` file (respectively) for the code.

Find the command to output a `hello.ll` file that should start with

```
; ModuleID = 'hello.c'
source_filename = "hello.c"
```

4. Re-read the paragraph that starts with "Beyond being implemented as a language,…" in Chris Lattner's LLVM article in The Architecture of Open Source Applications, and convert the `.bc` file into an `.ll` file. Then, compare the `.ll` file you obtained at this step with the file you obtained at the previous step: are they similar? What are their differences?

5. Similarly, convert the `.ll` file into an `.bc` file and compare the obtained file with the `.bc` file you produced following the tutorial: are they similar? What are their differences?

---

4 Longer Problems

Problem 1 Look back at Chris Lattner's LLVM article in The Architecture of Open Source Applications, and more particularly at the *C source code* shared in "11.3. LLVM's Code Representation: LLVM IR".

1. Copy-and-paste the code into a file called `additions.c`, and try to compile it with clang. Explain the error message that you obtained.
2. Obtain the `ll` version of this file using, e.g.,

   ```
   clang -o3 -emit-llvm additions.c -S -o additions.ll
   ```

   Was Chris Lattner completely honnest in his representation of LLVM's intermediate representation?
3. Use

   ```
   opt -S -O3 -aa -basic-aa -tbaa -licm additions.ll -o additions_opt.ll
   ```

   to obtain an optimized version of your `.ll` file, and compare the optimized and non-optimized versions: is this any closer to what Chris Lattner was discussing?
4. Now, pass the `-Os` flag to clang, using e.g.

   ```
   clang -Os -emit-llvm additions.c -S -o additions_simpl.ll
   ```

   and inspect the `.ll` file obtained. What is different about it?
5. Look in `clang` manual (using `man clang` or consulting an on-line version) for the ~10 different levels (!) of optimization, and make sure you understand what `-Os` does.

Problem 2 Save the following two programs as `c` files:

```
int main(){
    int a, b;
    if (1) a = 1;
    else a = 3;
}
```

and

```
int main(){
    int a, b;
    if (a > b) a = 1;
    else a = 3;
}
```

and compile them into their `.ll` forms, using `clang -S file.c -emit-llvm -o file.ll`. Pay a close attention to the way the `if` statement have been converted: what can you say about both files?