Lab 4
Control Flow Graph

Objective

- Understand the CFG construction and linearization.
- This is due on **`https://etudes.ens-lyon.fr`** (**NO EMAIL PLEASE**), before **2025-10-14 23:59**. More instructions in section 4.5.

4.1 Preliminaries

Student files are in the Git repository. We use the graphviz visualization tool, that you need to install if you did not already do it for Lab 2.

```
sudo apt install graphviz
python3 -m pip install graphviz
```

Make sure your Git repository is up-to-date, using `git pull`. The local offline documentation has been updated: try it by opening `cap-lab25/docs/html/index.html` in your favorite browser.

4.2 CFG construction

During class we presented Control Flow Graphs with maximal basic blocks. In this lab you will transform the linear code produced during the previous lab into a CFG, using the algorithm seen during the course.

EXERCISE #1 ► **CFG By hand**
What are the expected result of the CFG construction for the each of these programs?

Listing 4.1: `df01.c`

```
int n,u,v;
n=6;
u=12;
v=n+u;
print_int(v);
```

Listing 4.2: `df04.c`

```
int x;
x=2;
if (x < 4)
   x=4;
else
   x=5;
print_int(x)
```

Listing 4.3: `df05.c`

```
int x;
x=0;
while (x < 4){
   x=x+1;
}
```

EXERCISE #2 ► **Finding the leaders**
In the course on intermediate representations, we have defined the notion of *basic blocks* and *leaders*, which designate the indices of the instructions starting a block. We define the `find_leaders` procedure as taking the list of instructions and returning a list of leaders. The list of indices `leaders` should have the following properties:

- `leaders[i]` is the starting instruction of the i^{th} block.
- Each interval `leaders[i]` to `leaders[i+1]-1` delimits the instructions of a block.
- We have `leaders[0]=0` and `leaders[-1]=len(instructions)`.

Compute the leaders by hand on the following example.

```
subi temp2, temp2, 4
beq temp2, zero, lelse1
li temp4, 7
```

```
mv temp1, temp4
jump lendif1
lelse1:
addi temp3, temp2, 1
mv temp1, temp3
lendif1:
```

EXERCISE #3 ► **Completing the CFG Construction (file `CodeGen/BuildCFG.py`)**

The `Lib.CFG` module contains all the utilities related to Control Flow Graphs:

- the `Block` class, representing a basic block,
- the `CFG` class, representing a complete function in CFG form.

Additionally, the `Lib.Terminator` module contains definitons related to terminators, the final branching instructions of the blocks. Before writing any code, you should carefully read the documentation of these new modules.

The construction of the CFG is split into several pieces, mainly in `CodeGen/BuildCFG.py`.

- The `find_leaders` function returns a list of all the leaders.
- The `separate_with_leaders` function breaks the code into pre-chunks according to the list of leaders.
- The `prepare_chunk` function takes a pre-chunk and extracts its initial label and last jump (if any) from the other statements; it also check the rest of the statements are without any label or jump.
- The `jump2terminator` function converts the final jump of a chunk into a terminator.
- The rest of the `build_cfg` function uses the initial labels, inner instructions, and final jumps to build the actual CFG blocks, and add edges between the blocks based on the terminators.

You have to complete the procedures `find_leaders` and `prepare_chunks`. The procedure `find_leaders` is currently incomplete and always return the list `[0,len(instructions)]`. You can assume that for each label that appears in the list of instructions, there is an instruction somewhere that jumps to it; this makes the code significantly simpler and just as correct.

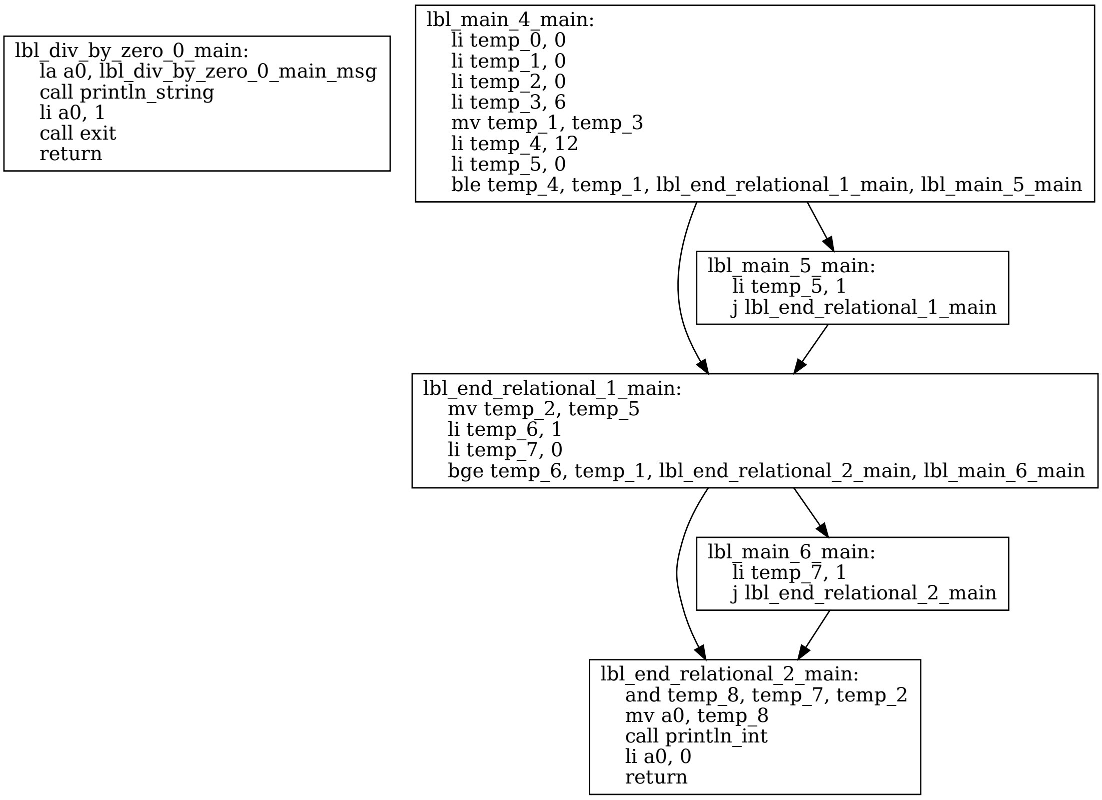
In `prepare_chunks`, you have to replace the two `raise NotImplementedError()` by an extraction of the first instruction if it is a label (otherwise, create a fresh label identifying this block with `fdata.fresh_label`) and an extraction of the last instruction if it is a jump (otherwise do nothing).

Reminder: at any point, you can run `make test-pyright` to check your code for typing errors.

You can test your code by specifying the `--mode codegen-cfg` option to `MiniCC.py`. Furthermore, when adding the option `--graphs` to `MiniCC.py`, the graph of the CFG will be printed as a PDF file `<name>.dot.pdf` (using the tool “dot”), in the same directory as the source file `<name>.c` and opened automatically. You can try it with:

```
python3 MiniCC.py --mode codegen-cfg --reg-alloc none --graphs /path/to/example.c
```

For example, the CFG for `df02.c` should look like:



The isolated block in this example corresponds to code related to division by zero. This particular block is added independently from code generation.

<ins>EXERCISE #4</ins> ► **Check and test your CFGs**

Check examples with:

1. Straight code (for instance `CodeGen/tests/provided/dataflow/df01.c`)
2. Boolean expressions, tests and if statements
3. While loops
4. Your own tests

If available, use `--reg-alloc all-in-mem` to obtain executable code from the CFG.

Note that register allocation does not affect the CFG printed by `--graphs`, as it is output before register allocation. In the rest of the lab, your compiler will do the allocation on the CFG, and all the tests from the previous lab should still pass.

To run the test suite with the CFG, you can run `make test-lab4 MODE=codegen-cfg`. This will run all tests in CFG mode with both the naive and all-in-memory allocators.

If you get errors with the all-in-mem allocator but not with the naïve one, you probably overlooked something in the implementation during lab 4a, and we invite you to triple-check the subject. In particular, check that you did not add useless `ld` or `sd` instructions.

4.3 Optimized CFG linearization

<ins>EXERCISE #5</ins> ► **CFG Linearization**

Before emitting instructions, a control flow graph needs to be *linearized*, i.e., turned back into a linear sequence of instructions. This is done in the `LinearizeCFG.py` file by the `linearize` function.

The current method is not very efficient: it emits jump instructions at the end of each block to link to the next block, even if it is immediately next. This results in the assembly instructions on the left below, even though the instructions on the right are sufficient (and more efficient). Furthermore, the current implementation doesn't try to re-order the block to minimize the number of jumps. This will be particularly problematic after the next lab on SSA form, which will add new blocks to our CFG.

Listing 4.4: Code with extra jump

```
lbl_end_relational_3_main: ld s1, -144(fp) beq s1, zero,
lbl_end_while_2_main j lbl_main_6_main
 lbl_main_6_main: li s2, 1
sd s2, -152(fp) ...
```

Listing 4.5: Code without extra jump

```
  lbl_end_relational_3_main: ld s1, -144(fp) beq s1, zero,
  lbl_end_while_2_main lbl_main_6_main: li s2, 1 sd s2, -152(fp)
  ...
```

1. Inspect the assembly of simple programs after CFG linearization. Try to linearize them by hand to minimize the number of jumps.
2. Improve the `linearize` function in `CodeGen/LinearizeCFG.py` to avoid jumps to blocks that are immediately following.

4.4 Extension

EXERCISE #6 ▶ **Block reordering**
How would you choose a better order to linearize the blocks? Explain your reasoning in the readme file, and try to implement it in the function `LinearizeCFG.ordered_blocks_list`.

4.5 Delivery

EXERCISE #7 ▶ **Readme**
Complete the `README-codegen.md` from lab 4a with any relevant information about your lab 4b.

EXERCISE #8 ▶ **Archive**
Type `make tar` in the `MiniC` folder to obtain the archive `MYNAME.tgz` to send (change your name in the Makefile before!).

Labs 4a and 4b (working together) are due on the course's webpage

`https://etudes.ens-lyon.fr/`

To get a perfect grade for lab 4, you need to implement everything perfectly, plus a syntax extension from lab 4a or the block reordering extension from lab 4b.

Your archive must also contain your tests (TESTS!) in the `CodeGen/tests/students` folder, with complete coverage of the code in `CodeGen/`. We expect tests with clear and explicit names that are relevant for what we implemented in lab 4. The command `make test-lab4 MODE=codegen-cfg` must work with your implementation; if some of the tests you have fail, please report the corresponding bugs in your readme.