Lab 6
Intermediate Representations

Objective

While the direct code production is quick, the resulting program yields poor performances. In order to provide a common base for optimisation, code selection, scheduling, memory allocation, ...we will work on some intermediate representation of our program. The objectives of this lab is to build such intermediate forms.

6.1 Given code

- **Register.h/.cpp** replace maintenance of temporaries. We assume that we possess an infinite number of registers and we will take care of their actual allocation (on the stack or inside one of the 4 free registers) at code generation.
- **BitVector.h/.cpp** is simply a class to manipulate bit vectors.
- **Code.h** contains an abstract class. It gathers methods common to different classes that implements intermediate representations (`PseudoCode`, `BasicBlock`, ...).
- **PseudoCode.h** represents a single pseudo-instruction. Remark the implementation of the different abstract methods of the `Code` class. The first group of instructions (line 15) corresponds to instructions seen in the lecture. Next groups implements specialized ones (over reserved registers: SP, ARP). Finally, remark the pretty-constructors, alike the ones generating the Digmips code in the previous lab.
- **Cfg.h** implements a control flow graph. Every node of this graph contains a `Code` object (thus, either a `PseudoCode`, or a `BasicBlock`). `live-in[i]` contains the temporaries that are live *just before* executing the `i` node. `live-out[i]` contains the temporaries that are live *just after* the execution of the `i` node. Remark the pretty-constructors, that add one pseudo-code instruction to a global CFG (`cfg` variable, defined inside parser.ypp, line 53).

6.2 Control-flow graph

EXERCISE #1 ▶ **CFG build - `Cfg.cpp`**
Given the following "pseudo-asm" code :

extests.asm

```
1     r0 = 1
      r1 = 0
      r2 = 10
loop :
      if (r1 < r2) goto end_loop
6     r1 = r1 + r0
      r3 = 1
      r4 = r3 + r0
      goto loop
end_loop :
11    r4 = r4 + r0
```

- Draw the CFG on paper.

- In `main.cpp`, use the CFG API (calls to `add,cjump`) to build the CFG datastructure representing this code.
  At the end, do not forget to call the method `fix_succc` to compute the non-trivial edges on your cfg, otherwise it will be linear.
- Display the CFG. A call to `cfg->print_dot(cout)` produces a `dot` description on the standard output.[1].

EXERCISE #2 ► **Live ranges - Basic Blocks**

On the same code:
- Compute live-ranges for each temporary with the classic fixpoint algorithm (by hand).
- Use then `do_liveness()` method of `Cfg.cpp` to check your results (use the method `cfg->fix_succ()` just before), then display the resulting CFG.
- Extract the basic blocks using the `extract_basic_blocks()` method of `Cfg.cpp`. The extraction has to be done after the computation of the live ranges. Display the resulting CFG.

6.3 Data flow analyses on CFGs

We already saw an example of data flow analysis (live-in). Here is another example. Recall the data-flow equations for available expressions :

$$AE_{entry}(\ell) = \begin{cases} \emptyset & \text{if } \ell = init \\ \bigcap\{AE_{exit}(\ell') | (\ell', \ell) \in flow(G)\} & \end{cases}$$

$$AE_{exit}(\ell) = \left(AE_{entry}(\ell) \backslash kill_{AE}(\ell)\right) \cup gen_{AE}(\ell);$$

where:
- $AE_{entry}(\ell)$ (resp. $AE_{exit}(\ell)$) denotes the available expressions at the entry (resp. exit) of block ℓ;
- $flow(G)$ denotes the graph transition relation;
- $kill_{AE}(\ell)$ denotes expressions that are killed in the block. For instance, any expression containing b is killed by $b := ....$
- $gen_{AE}(\ell)$ denotes expressions that are generated (and not killed) in the block. $x := a + b$ generates the expression $a + b$.

EXERCISE #3 ► **Available expressions. On paper.**
Consider the following program:

```
x:=a+b;
y:=a*b;
while(y>a+b) do
   a:=a+a;
   x:=a+b;
done
```

- Draw the CFG.
- Compute the available expressions for all blocks.
- Optimize the code: replace useless recomputations of arithmetic expressions.

6.4 DAG generation

The class **Dag.cpp** implements a direct acyclic graph between pseudo-instructions. `node_reg[tmp]` is the root node of the expression computed inside the `temp` temporary. `node_def[thenode]` is the list of temporaries that contains the results computed until node `thenode`.

[1] Use redirection to produce the dot file: `./digcc >cfg.dot`, then produce a pdf calling `dot -Tpdf cfg.dot > cfg.pdf`

EXERCISE #4 ► **Using DAG API**
Open `Dag.cpp` and review the constructor. It executes redundancies elimination inside blocks, and also do some simplifications such as $0+x \rightarrow x$, constant propagation... For each instruction `r = r' op r"`, we examine if `r'` and `r"` are associated to existing nodes, and if those nodes have a common ancestor n, that executes *op*. If yes, $node\_reg[r] := n$. Similar rules exist for the other kinds of operator.

- In `main.cpp`, build the DAG for node 2[2].
- (main) Display this DAG with `dag->print_dot(cout);`.

6.5 Production of intermediate code

The file **parser.ypp** is modified to create a new CFG for every function (line 812). Remark the use of the pretty-constructors of `Cfg.h` inside the translation rules. For a given function, we compute live ranges (line 827), then extract the basic blocks (line 830) and finally produce a DAG for each basic block (line 834 and following).

EXERCISE #5 ► **Syntax-based CFG construction + code production**
In `main.cpp`, comment your additions and uncomment the call to the parser. Compile, then look at the generated code for the given test files. Try it with the example files of last lab (exassign, for and while examples). Write adequate tests for function calls. Some of the simplifactions done before must be visible.

[2] With the instruction: `Dag* dag = new Dag((BasicBlock*)cfg_bb->node[2]);`, where `cfg_bb` is the CFG with basic blocks built during exercise 6.2