Lab 5
Register allocation with smart IRs

Objective

- Compute live ranges, construct the interference graph.
- Allocate registers and produce final "smart" code.
- Labs 5a and 5b are due **on `https://etudes.ens-lyon.fr` (NO EMAIL PLEASE)**, before **2021-11-15 23:59**. More instructions in section 5.4.

During this Lab, you will modify the files `SmartAllocation.py` and `LivenessSSA.py`. You already used `NaiveAllocator` and `AllInMemAllocator` in the previous lab (the mapping from temporary to physical register or memory location was provided to you, and you had to modify the 3 address code to take this mapping into account). We first complete `LivenessSSA`, which computes liveness information on SSA programs. Next, we implement `SmartAllocator` which uses the liveness informations to map temporaries to physical registers in an optimized way, and uses memory (i.e. spilling) only when necessary. Read the body of `SmartAllocator.prepare()` and `SmartAllocator.rewriteCode()`, that gives the main steps of the allocation: conflict graph, graph colouring, and finally 3 address code modification to get the final executable.

5.1 Check your previous lab

To begin this lab, you need to finish the implementation of the previous one (apart from the bonus questions). Make sure it is working correctly with `make tests-pyright` (to check for typing mistakes) and `make tests-notsmart SSA=1` (to run the test suite).

5.2 Liveness analysis and Interference graph

To build the interference graph and proceed with the allocation, we need the liveness analysis. We use two pieces of information at each instruction: *live in*, marking variables which are alive before the instruction, and *live out*, for the variables alive after the instruction. We will store only the `liveout` information for each instruction.

The liveness algorithm proceeds by starting from each *use* of a variable and then propagating liveness backward until it reaches a definition. The recursion is bound by blocks to ensure termination. Here is the completed pseudo-code for the algorithm seen during the course.

```
For each statement S in the program:
  OUT[S] = {}
For each variable v in the program:
  For each statement S that uses v:
    if S is a ϕ containing (B,v):
      liveout_at_block(B,v)
    else:
      livein_at_instruction(S, v)
conflicts_on_phis()

liveout_at_block(B, v):
  if v was not propagated in B:
    S = last instruction of B
    liveout_at_instruction(S, v)

livein_at_instruction(S, v):
```

```
  if at the beginning of block B:
    for each Bpred in pred(B):
      liveout_at_block(Bpred, v)
  else:
    Spred = pred(S):
    liveout_at_instruction(Spred, v)

liveout_at_instruction(S, v):
  OUT[S] = OUT[S] ∪ {v}
  if S does not define v:
    livein_at_instruction(S, v)
```

Recall that we always generate move instructions for phi nodes. This means that all variables newly introduced by phi instructions have to be in conflicts with one-another. The method `conflicts_on_phis()` ensures this is the case by marking these variables as alive. It is called after the program above.

This algorithm is partially implemented in `TP05/LivenessSSA.py`. In particular, the `run` function initializes and populates the `liveout` dictionary. We recall that variables defined (resp. used) by an instruction are available through `instr.defined()` (resp. `instr.used()`).

Exercise #1 ► Liveness Analysis on SSA

This exercise is the most important of the Lab!

Complete the `liveout_at_block`, `livein_at_instruction` and `liveout_at_instruction` procedures to implement the pseudocode above.

Carefully check that your results are correct at least with the examples of the `dataflow/` directory. As an example, here is a possible output for `dataflow/df04.c`, obtained with the command `python3 MiniCC.py --reg-alloc smart --ssa --debug TP05/tests/provided/dataflow/df04.c` (temp numbering may differ):

```
Out: {...,
 "# (stat (assignment x = (expr (atom 2))) ;)": {},
 "li temp_10, 2": {temp_10},
 "mv temp_11, temp_10": {temp_11},
 "# (stat (if_stat if ( (expr (expr (atom x)) < (expr (atom 4))) ) ... else ...))": {temp_11},
 "li temp_12, 4": {temp_12,temp_11},
 "li temp_13, 0": {temp_13,temp_12,temp_11},
 "bge temp_11, temp_12, lbl_end_relational_3_main": {temp_13},
 "li temp_14, 1": {temp_14},
 "temp_15 = φ({main_5_main: temp_14, main_4_main: temp_13})": {temp_15},
 "beq temp_15, zero, lbl_else_2_main": {},
 "li temp_22, 4": {temp_22},
 "mv temp_23, temp_22": {temp_23,temp_22},
 "j lbl_end_if_1_main": {temp_23,temp_22},
 "li temp_16, 5": {temp_16},
 "mv temp_17, temp_16": {temp_17,temp_16},
 "temp_18 = φ({else_2_main: temp_16, main_6_main: None})": {temp_18},
 "temp_19 = φ({else_2_main: None, main_6_main: temp_22})": {temp_18,temp_19},
 "temp_20 = φ({else_2_main: temp_17, main_6_main: temp_23})": {temp_18,temp_19,temp_20},
 "# Return at end of function:": {},
 ...}
```

Exercise #2 ► Interference graph

The interference graph contains an edge (x, y) if temporaries x and y are in conflict. It is built using the liveness information in the function `SmartAllocator.build_interference_graph` and the `interfere` method.

We recall that two temporaries x, y are in conflict if they are simultaneously alive after a given instruction, which means:

- There exists an instruction i and $x, y \in$ `liveout[i]`
- OR There exist an instruction i such that $x \in$ `liveout[i]` and y is defined in the instruction
- OR the converse.

To understand why the last two cases are needed, consider the following list of instructions:

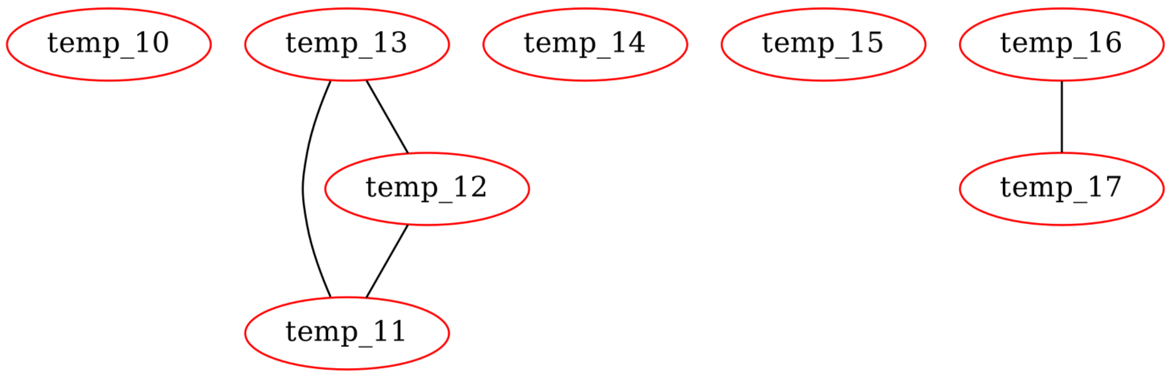
```
y=2
x=1 // Can x and y be mapped to the same place? Obviously not.
z=y+1
```

where x is not alive after the `x=1` statement, however x is in conflict with y since we generate the code for `x=1` while y is alive[1].

From the result of the previous exercise, construct the interference graph (the job is done by function `build_interference_graph`) of your program. You need to iterate over each pair of temporaries, and call `interfere` (that you also need to implement). We give you a undirected graph API (`TP05/LibGraphes.py`) for that. Use the `print_dot` method and relevant tests to validate your code.

In this exercise, we care about correctness more than complexity. It is sufficient to write an $O(n^3)$ algorithm (for each t_1, for each t_2, for each control point c, check whether t_1 and t_2 have a conflict).

As an example, here is part of the conflict graph that should be obtained for `df04.c` (temp ordering and numbering may differ):



5.3 Register allocation and code production

We will implement the following algorithm for register allocation:

- Color the interference graph with an infinite number of colors, using the first ones as much as possible.
- The first `len(GP_REGS)` colors will be mapped to registers.
- All the other variables will be allocated on the stack. For each color, we use a memory location according to their coloring number.
- To add the moves corresponding to ϕ nodes when exiting SSA form, we may have to load or store instead of doing moves between registers, according to whether the source and target are mapped to registers or memory locations.

Then the 3 address code modification:

- For non-spilled variable: replace the temporary with its associated color/register, as we did for the naive allocator.
- For spilled variables: add `ld` / `sd` statements as needed and replace the temporary with one of `s1`, `s2`, `s3` as we did for the "all in mem" allocator.

Some help:

- `GP_REGS` is an array of registers available for the register allocator.

[1] Another solution consists in eliminating dead code before generating the interference graph.

- An element of type Register can be obtained from a given register color with the helper function `GP_REGS[coloringreg[xxx]]`, where `coloringreg` is graph coloring returned by the `.color()` function, and for offsets you have the method `self._function_code.new_offset()` that returns a fresh one (all in `Operands.py`).
- The easiest way to build `alloc_dict` is probably to iterate over all the temporaries of the program (available in `self._pool._all_temps`), and for each temporary check the corresponding color to associate it to the right register or memory location in `alloc_dict`.

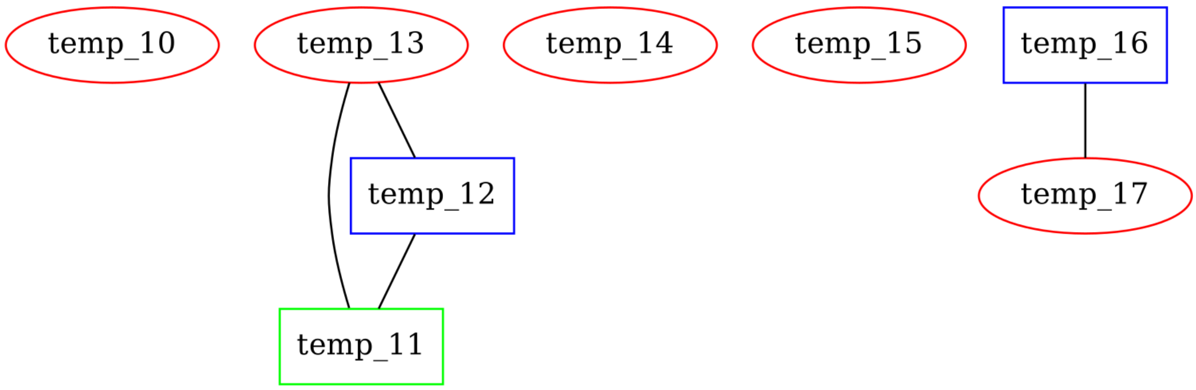
EXERCISE #3 ► Smart Register Allocation: implement!

In this exercice, you have first to complete method `smart_alloc()` to perform an allocation based on a graph coloring. The purpose of this method is to allocate a physical register or a memory location for each temporary in the program. Next, you will have to complete the function `replace_smart()` that replaces the temporary operands of a given instruction according to the allocation computed by `smart_alloc()`.

Use the algorithm and the coloration method of the `LibGraphes` class to allocate registers (or a memory location) in `smart_alloc()`. Comments will help you design this (non trivial) function. The allocation is followed by statement rewriting, like in previous lab. You need to implement it in `SmartAllocation.py` (`replace_smart`): it is very similar to the previous lab's version, but you have to deal with both memory locations and registers in the same function.

Validate your allocation on tiny well chosen test files (especially tests that augment the register pressure) and all the benchmarks of the previous lab. We adapted the previous script for that.

On the `df04.c` example, the graph coloring succeeds and the part shown above becomes:



Each color+shape pair indicates a different location. Temp numbering and coloring may be different in your output.

EXERCISE #4 ► SSA exit for allocated registers

In your previous implementation of SSA exit, you added a block containing one move per relevant ϕ instruction. All the assignments in a set of ϕs are supposed to be executed “in parallel”. Initially, these moves were done on temporaries, which are guaranteed to be assigned only once by the SSA form and thus allow to emit the moves in an arbitrary order. Now that we are allocating the registers, we need to be more careful.

Consider a block starting with the following ϕ instructions:

```
temp_1 = φ(temp_5, ...)
temp_2 = φ(temp_6, ...)
temp_3 = φ(temp_7, ...)
temp_4 = φ(temp_8, ...)
```

and the following allocation:

```
{temp_8: s4, temp_5: s5, temp_7: s6, temp_6: -8(FP),
 temp_4: s4, temp_1: -8(FP), temp_3: s5, temp_2: s6}
```

Clearly, there is a cycle in the assignments. A naive implementation of the moves would result in incorrect code. Furthermore, one of the location is in memory: A simple move instruction will not work.

The solution of the first problem is to find a correct order of moves that accounts for cycles and use an extra register to implement said cycles. The second problem can be solved by replacing the standard `mv` instruction by stores and loads as appropriate when one of the operand is in memory. Two functions in `SmartAllocation.py` allow to deal with these issues. `sequentialize_moves` takes an extra register and a set of moves *on registers or memory locations*, and return a correct sequence of instructions, using the extra register to implement swaps if necessary. `generate_smart_move` takes a destination and a source that might be registers or memories and return a sequence of instruction implementing the assignment.

1. Write on paper the sequence of instructions to implement the example above after allocation and SSA exit.

2. Complete the implementation of `generate_smart_move`. Be mindful of the 4 potential cases: either the source and destination are both registers, both memory, one is register and the other memory, or the opposite.

3. Rewrite your original implementation of `generate_moves_from_phis` to implement SSA exit with allocation. First generate a set of `(dest,src)` moves (there can be both registers or memory locations in the pairs) to provide to `sequentialize_moves`, then use `generate_smart_move` to generate the desired list of instructions.

EXERCISE #5 ► **Massive tests**

Test your implementation on all test files you have. For that purpose, `make tests SSA=1` runs your compiler on the whole test suite, while `make tests-smart SSA=1` only runs it on tests from lab 5 (this is less thorough but faster).

Make sure no debug output (`print_dot`...) is printed when options `--debug`, `--graphs` and `--ssa-graphs` are not given. **In particular, we do not want any pdf file to be opened when we will use `make tests SSA=1` on your delivered code.**

Do not forget to check that your test suite has a good coverage of the files relevant to Lab 5. To see detailed information on coverage, open `htmlcov/index.html` in your web browser after a run of `make tests SSA=1`.

5.4 Final delivery

We recall that your work is **personal** and code copy is **strictly forbidden**.

EXERCISE #6 ► **Archive**

Labs 5a and 5b are due on the course's webpage

`https://etudes.ens-lyon.fr/course/view.php?id=4814`

Python code and C testcases will be graded. **Late deliveries will get a heavy penalty, and deliveries more than one hour late will not be accepted.**

Type `make tar` to obtain the archive to send (change your name in the Makefile before!). Your archive must also contain tests (TESTS!) and a (minimal) `README-SSA.md` with your name, the functionality of the code, how to use it, your design choices if any, the bonus you implemented, and known bugs.