# Lab 5

# Smart IRs, part B: Register allocation

## Objectives

- Compute live ranges and use them to 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 **2025-11-12 23:59**. More instructions in section 5.5.

**During this Lab, you will modify the following files `RegAlloc/SmartAllocator.py`, `RegAlloc/LivenessSSA.py`, `RegAlloc/SequentializeMoves.py` and `RegAlloc/ExitSSA.py`.**

You already used `NaiveAllocator` and `AllInMemAllocator` in lab 4 (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 information 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()`, that gives the main steps of the allocation: compute liveness, build the conflict graph, color this graph, and finally modify the 3 address code to get the final executable.

Finally, we will adapt SSA exit to the smart allocator, because $\phi$ nodes imply parallel moves that have to be handled specially (with *windmills*, as seen in the course).

## 5.1 Check your previous lab

To begin this lab, you need to finish the implementation of the previous one. Make sure it is working correctly with `make test-lab4 MODE=codegen-ssa` 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. This is done in the `RegAlloc/LivenessSSA.py` file. 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 complete 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:
    livein_at_instruction(S, v)
conflicts_on_phis()

livein_at_instruction(S, v):
  if S is a phi node and v is used in S:
    Bpred = predecessor of B associated to v in S
    liveout_at_block(Bpred, v)
  else 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)

liveout_at_block(B, v):
  if v was not propagated in B:
    S = last instruction of B
    liveout_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()` must ensure this is the case by marking these variables as alive in the $\phi$ nodes. It is called after the program above (see the `run` method).

This algorithm is partially implemented in `RegAlloc/LivenessSSA.py`. In particular, the `run` method initializes and populates the `liveout` dictionary. We recall that variables defined (resp. used) by an instruction `instr` 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 procedures `liveout_at_instruction`, `livein_at_instruction`, `liveout_at_block` and `conflict_on_phis` to implement the pseudo-code above.

Carefully check your results are correct at least with the examples of the `CodeGen/tests/provided/dataflow/` directory. As an example, here is a possible output for `dataflow/df04.c`, obtained with the command

```
python3 MiniCC.py --mode=codegen-ssa --reg-alloc smart --debug CodeGen/tests/provided/dataflow/df04.c
```

(temp numbering may differ):

```
Liveout: [
Block lbl_div_by_zero_0_main: {...}
Block lbl_end_if_1_main: {
 "temp_15 = ϕ({else_2_main: temp_19})": {temp_15},
 "temp_16 = ϕ({else_2_main: temp_20, main_7_main: temp_22})": {temp_15,temp_16},
 "temp_17 = ϕ({main_7_main: temp_21})": {temp_15,temp_17,temp_16},
 "# Return at end of function:": {},
 "li a0, 0": {},
 "return": {}}
Block lbl_else_2_main: {
 "li temp_19, 5": {temp_19},
 "mv temp_20, temp_19": {temp_20,temp_19},
 "j lbl_end_if_1_main": {temp_20,temp_19}}
Block lbl_main_7_main: {
 "li temp_21, 4": {temp_21},
 "mv temp_22, temp_21": {temp_22,temp_21},
 "j lbl_end_if_1_main": {temp_22,temp_21}}
Block lbl_end_relational_3_main: {
 "temp_14 = ϕ({main_6_main: temp_23, main_5_main: temp_13})": {temp_14},
 "beq temp_14, zero, lbl_else_2_main, lbl_main_7_main": {}}
Block lbl_main_6_main: {
 "li temp_23, 1": {temp_23},
 "j lbl_end_relational_3_main": {temp_23}}
Block lbl_main_5_main: {
 "li temp_8, 0": {},
 "li temp_9, 0": {},
 "# (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))) ) (stat_block (stat (assignment x = (expr
     (atom 4))) ;)) else (stat_block (stat (assignment x = (expr (atom 5))) ;))))": {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, lbl_main_6_main": {temp_13}}
]
```

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 given by the function `SmartAllocator.build_interference_graph`.

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 $i$ in a block $b$ and $x, y \in$ `liveout[b, i]`
- OR there exist an instruction $i$ in a block $b$ such that $x \in$ `liveout[b, i]` and $y$ is defined in the instruction $i$.

To understand why the last case is 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
```

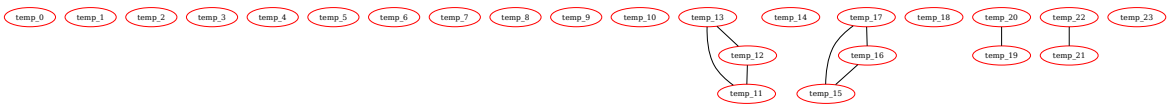
where $x$ is not alive after the `x=1` statement, however $x$ is in conflict with $y$ since we generate the code for `x=1` while $y$ is alive[1].

From the result of the previous exercise, the code in `LivenessSSA` builds a `self._liveness._liveout` field of type `Dict[tuple[Block, Statement], Set[Temporary]]` that gives the set of temporaries that are live after a given instruction. Use this data to construct the interference graph of your program (this is done by the function `build_interference_graph`).

You need to iterate over each instruction, and look at which temporaries are in conflict at this place according to the liveness analysis (approximately 10 lines of code).

The library contains an undirected graph API in `Lib/Graphes.py` for that. Use the `--graphs` option and relevant tests to validate your code.

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 $\phi$ 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.

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

Some help:

- `GP_REGS` is an array of registers available for the register allocation.
- An element of type Register can be obtained from a given register color with the helper function `GP_REGS[coloringreg[xxx]]`, where `coloringreg` is the graph coloring returned by the `.color()` function, and for Offset you have the method `self._fdata.fresh_offset()` that returns a fresh one (see the documentation).
- The easiest way to build `alloc_dict` is probably to iterate over all the temporaries of the program, and for each temporary check the corresponding color to associate it to the right register or memory location in `alloc_dict`.
- To launch `MiniCC.py` with the smart allocator, use `--reg-alloc smart`. To run the test suite, use `make test-smart MODE=codegen-ssa`.

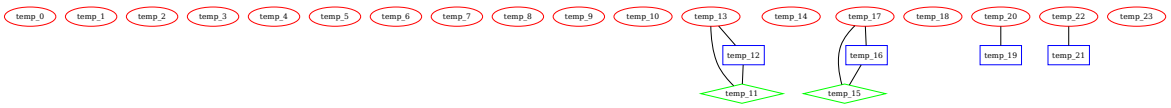
EXERCISE #3 ▶ **Smart Register Allocation: implement!**

In this exercice, you have first to complete method `SmartAllocator.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` 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 `Lib/Graphes` class to allocate registers or offsets in `smart_alloc()`. The allocation is followed by statement rewriting, like in previous lab. You need to implement it in `SmartAllocator.py` (`replace`): 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, using `--reg-alloc smart`.

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, from lab 5a, you added a block containing one move per relevant $\phi$ instruction. All the assignments in a set of $\phi$s are supposed to be executed "in parallel". In the cases of the naive and all-in-mem allocators, everything worked fine because there was no register/memory location reuse at all. Now that we are potentially allocating several (non-interfering) temporaries to the same register, we need to be more careful.

Consider a block starting with the following $\phi$ 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 locations 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 `SequentializeMoves.py` will allow to deal with these issues. You have to complete the implementation of `sequentialize_moves` which takes a set of moves `parallel_moves` *between Register or Offset*, and returns a correct sequence of instructions, using an extra register `tmp` to implement swaps if necessary. This corresponds to the lecture you had about windmills.

Then, you have to write the function `generate_smart_move` which takes a destination and a source that might be Register or Offset, and returns a sequence of instructions implementing the assignment.

Finally, rewrite `generate_moves_from_phis` in `ExitSSA.py` to use these new functions (or write another function to do so, with an explicit name like `generate_moves_from_phis_smart`, and use the `is_smart` parameter of the `exit_ssa` function to use the latter when allocating with the Smart Allocator). **To be able to call your function from `SequentializeMoves.py`, you have to add at the beginning of `ExitSSA.py` the following:**

```
from RegAlloc.SequentializeMoves import sequentialize_moves
```

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

2. Write the implementation of `sequentialize_moves`, to sequentialize a set of parallel moves given by $\phi$ nodes. It proceeds in two steps: 1) Generate moves from all the leaves 2) Once there are no more leaves, only cycles remain. Handle them using the extra register provided. Here is the pseudo-code for the part of this function you have to complete (the code provided buids the graph from the parallel moves given, and transforms the list of moves into actual RISCV instructions).

   ```
   sequentialize_moves(G)=
       moves = []
       For each vars v without successors in G:
           For src in pred(v):
               moves := moves + (v, src)
           G := G \ {v}
       For each cycle in G:
           previous := tmp
           for v in reversed(cycle):
               moves := moves + (previous, v)
               previous := v
           moves := moves + (previous, tmp)
       return moves
   ```

3. Write the implementation of `generate_smart_move`. Be mindful of the four potential cases: either the source and destination are both Register, both Offset, the first is a Register and the second an Offset, or the opposite.

4. 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 Register or Offset in the pairs) to provide to `sequentialize_moves`, which generates the desired list of instructions.

EXERCISE #5 ▶ **Massive tests**

Test your implementation on all test files you have. For that purpose,
`make test-codegen MODE=codegen-ssa`
runs your compiler on the whole test suite, while
`make test-smart MODE=codegen-ssa`
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`, `--dom-graphs` and `--ssa-graphs` are not given. **In particular, we do not want any pdf file to be opened when we will use `make test-codegen MODE=codegen-ssa` 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 test-codegen MODE=codegen-ssa`.

## 5.4 Extensions

EXERCISE #6 ▶ **Optimising swaps**
Improve the algorithm of `sequentialize_moves` to not use a supplementary register for cycles of size 1, as well as for cycles of size 2 between registers only.

## 5.5 Final delivery

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

EXERCISE #7 ▶ **Archive**
Labs 5a and 5b are due on the course's webpage on

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

Python code and C test cases will be graded. **Late deliveries will get a penalty of 1 point per hour. We will thoroughly check your code for plagiarism.**

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, any extension you implemented, and known bugs you could not solve.