Lab 5
Code generation with smart IRs

Objective

- Construct the CFG.
- Compute live ranges, construct the interference graph.
- Allocate registers and produce final code.

During the previous lab, you wrote a dummy code generator for the Mu language. In this lab the objective is to generate a more efficient SARUMAN code. **You will extend your previous code, in the same directory. People in advance are encouraged to keep their current code, students with more difficulties will be provided a working 3 address code generation Visitor on Tuesday, 21/01/2019, 6pm.**

Installations We're going to use graphviz for visualization. If it is not already installed (e.g. on your personal machine), install it, for instance with:

```
apt-get install graphviz graphviz-dev
```

You may have to install the following PYTHON packages:

```
pip3 install --user networkx
pip3 install --user graphviz
pip3 install --user pygraphviz \
  --install-option="--include-path=/usr/include/graphviz" \
  --install-option="--library-path=/usr/lib/graphviz/"
```

If the last command errors out complaining about a missing `Python.h`, run:

```
apt-get install python3-dev
```

and then relaunch the command `pip3 install ...` On the university machines, you might have to update existing already installed packages:

```
pip3 install --user --upgrade networkx graphviz pygraphviz
```

5.1 CFG construction

In class we have presented CFGs with maximal basic blocks. In this lab we will implement CFGs with minimal basic blocks that is CFG with one node per line of code/instruction (even comments).

EXERCISE #1 ▶ **CFG By hand**
What are the expected result of the CFG construction from the 3-address code of Lab5 for each of these programs ?

```
var n,u,v:int;
n=6;
u=12;
v=n+u;
log(v);
```

```
var x,y:int;
x=2;
if (x < 4)
   x=4;
else
   x=5;
log(x)
```

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

EXERCISE #2 ▶ **CFG Construction**

The `APISaruman` is able to deal with CFGs. `Instructions` have a list of predecessors (`self._in`) and successors (`self._out`) and a `SARUMANProg` contains the initial control point (`self._start`) from which we can traverse the graph. This feature allows us to easily construct the CFG of a program.

We give you the construction for all idioms. Each time your Visitor creates a new SARUMAN instruction, the CFG updates itself automatically: when adding an instruction, it creates an edge between the last instruction (`self._end`) and the instruction to be added. **Edit 22/1 8H : There was an issue with the student version of `APISaruman.py`, the CondJUMP as well as JUMP are not working right now. This will be updated in the git today, but a quick edit makes the job:**

Listing 5.1: 'APISaruman.py'

```
def addInstructionCondJUMP(self, label, op1, c, op2):
[...]
# the last 3 instructions should be:
      self.add_instruction(ins)
      self.add_edge(ins, label)
      return ins

 def addInstructionJUMP(self, label):
 [...]
 # the last 3 instructions should be:
      self.add_instruction(i, linkwithsucc=False)
      self.add_edge(i, label)
      return i
```

In this exercise, you only have to understand (look at the API!) and test the provided code.

The smart code generation called by `Main.py` has a second optional argument that enables it to print the CFG as a dot file. Verify that it is the case, if not, edit and modify it:

```
    elif reg_alloc == "smart":
      prog.do_smart_alloc(basename, True) # True to print CFGs
```

The file is printed as `<name>.dot.pdf` in the same directory as the source file.

Now you can launch:

```
python3 Main.py --reg-alloc smart /path/to/example.mu
```

1. Test for lists of assignments (for instance `testdataflow/df01.mu`) You should see a chain of blocks.
2. Same for boolean expressions, and tests.
3. Same for while loops ....
4. Propose appropriate examples and draw nice pictures!

5.2 Liveness analysis and Interference graph

For the liveness analysis, we recall the notations. A variable at the left-hand side of an assignement is *killed* by the block. A variable whose value is used in this block (before any assignment) is *generated*.

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \text{final} \\ \bigcup\{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} & \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

The sets are initialised to $\emptyset$ and computed iteratively, until reaching a fixpoint.
From now on, you have to modify `APISaruman.py`

EXERCISE #3 ▶ **Liveness Analysis, Initialisation**

Initialise the Gen(B) and Kill(B) for each kind of instruction (add, let, ...). This corresponds to the numerous TODOs `ADD GEN KILL INIT IF REQUIRED`.
We give you an example for the print instruction. Be careful to properly handle the following cases:

ADD temp1 temp1 12

and

LETI temp1 42

To test/debug this initialisation, you should uncomment the following lines in `APISaruman.py` (function `do_smart_alloc`):

```
        if debug:
        self.printGenKill()
```

As an example, here is the expected initialisation for `testsdataflow/df04.mu`, obtained by:

```
python3 Main.py --reg-alloc smart /path/to/df04.mu
```

```
instr 0: comment
gen: {}
kill: {}

instr 1: leti temp_2 2
gen: {}
kill: {temp_2}

instr 2: let temp_1 temp_2
gen: {temp_2}
kill: {temp_1}

instr 3: comment
gen: {}
kill: {}

instr 6: leti temp_3 4
gen: {}
kill: {temp_3}

instr 8: leti temp_4 0
gen: {}
kill: {temp_4}

instr 9: cond_jump lbl_end_relational_2
                   temp_1 sge temp_3
gen: {temp_1,temp_3}
kill: {}

instr 10: leti temp_4 1
gen: {}
kill: {temp_4}

instr 7: lbl_end_relational_2
gen: {}
kill: {}

instr 11: cond_jump lbl_end_cond_1
                    temp_4 eq 0
gen: {temp_4}
kill: {}

instr 12: leti temp_5 4
gen: {}
kill: {temp_5}

instr 13: let temp_1 temp_5
gen: {temp_5}
kill: {temp_1}

instr 14: jump lbl_end_if_0
gen: {}
kill: {}

instr 5: lbl_end_cond_1
gen: {}
kill: {}

instr 15: leti temp_6 5
gen: {}
kill: {temp_6}

instr 16: let temp_1 temp_6
gen: {temp_6}
kill: {temp_1}

instr 4: lbl_end_if_0
gen: {}
kill: {}
```

EXERCISE #4 ▶ Liveness Analysis, fixpoint. (Only test!)

We implemented for you the fixpoint iteration as a method (`doDataflow`) in `APISaruman.py` "while it is not finished, store the old values, do an iteration, decide if its finished". The `doDataflow` program method makes calls to `do_dataflow_onestep` instruction methods. The result (live in, live out sets of variables, are stored in `mapin` and `mapout` member sets of the `SARUMANProg` class).

All you have to do in this exercice is to check that the results that are obtained with with analysis are correct at least for the examples of the `testsdataflow/` directory.

To do so, you should first uncomment these two lines (same file):

```
mapin, mapout = self.doDataflow()
if debug:
    self.printMapInOut()
```

As an example, here is the expected output for `testsdataflow/df04.mu`: [1]

```
In: {0: {}, 1: {}, 2: {temp_2}, 3: {temp_1}, 4: {}, 5: {},
     6: {temp_1}, 7: {temp_4}, 8: {temp_3,temp_1},
     9: {temp_3,temp_1,temp_4}, 10: {}, 11: {temp_4},
     12: {}, 13: {temp_5}, 14: {}, 15: {}, 16: {temp_6}, 17: {}}
Out: {0: {}, 1: {temp_2}, 2: {temp_1}, 3: {temp_1}, 4: {}, 5: {},
      6: {temp_3,temp_1}, 7: {temp_4}, 8: {temp_3,temp_1,temp_4},
      9: {temp_4}, 10: {temp_4}, 11: {},
      12: {temp_5}, 13: {}, 14: {}, 15: {temp_6}, 16: {}, 17: {}}
```

EXERCISE #5 ▶ 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 a block (an instruction) b and $x, y \in LV_{out}(b)$
- OR There exist a block b such that $x \in LV_{out}(b)$ and y is defined in the block
- OR the converse.

For the two last cases, consider the following list of instructions:

```
y=2
x=1
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[2].

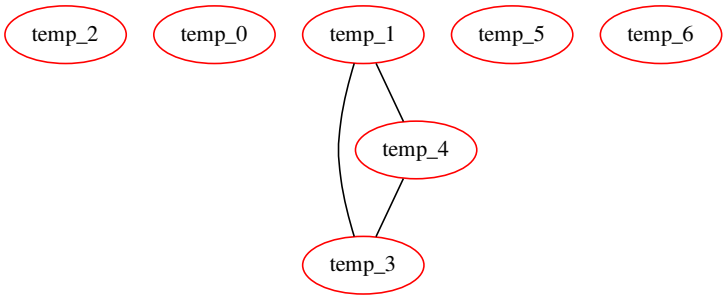
From the result of the previous exercise, construct the interference graph (complete the `doInterfGraph` function) of your program (each time a pair of temporaries are in conflict, add an edge between them). We give you a non-oriented graph API (`LibGraph.py`) for that. Use the `print_dot` method and relevant tests to validate your code.

In this exercise, we care about correctness more than complexity. It is OK 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 the conflict graph that should be obtained for `df04.mu` (uncomment some lines, command line as usual):

[1] Here $r7$ is the register we use to encode nop instructions, we can ignore it during the dataflow analysis (but not during code generation!)

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



5.3 Register allocation and code production

Instead of the iterative algorithm of the course, we will implement the following algorithm for k register allocation [3]:

- Color the interference graph with $k-2$ colors.
- All the other variables will be allocated on the stack. To compute the offset from the stack pointer (sp), recolor the subgraph of remaining variables with an infinite number of colors.

Then the memory allocation:

- For non-spilled variable: replace the temporary with its associated color/register.
- For spilled variables: do the same as "all in mem" in Lab 4!

update 22/1 4pm : an element of type Register can be obtained from a given register color with the helper function `GP_REGS[coloringreg[xxx]]`, and for offsets you have a constructor `Offset(SP, xxx)`. Grep is your friend.

update 23/1 6pm : be careful with types when dealing with the graph. As the comment in APISaruman.py states, `self._igraph` contains only elements of type string, while the `alloc_dict` map given to `self._pool.set_reg_allocation()` must have `Temporary` objects as keys. There is no easy way to retrieve a `Temporary` object from its name, but it is easy to get the name as a string from a `Temporary`: just use `str()`. 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`.

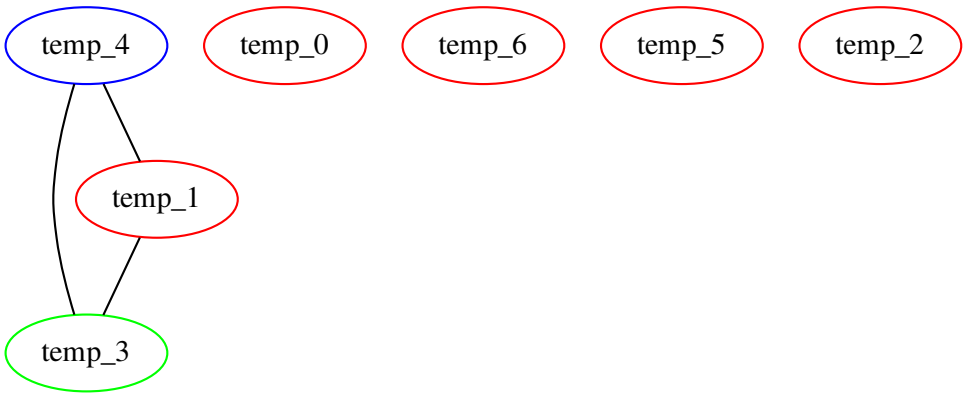
EXERCISE #6 ▶ **Register Allocation**

Use the algorithm (with k=8) and the coloration method of the `LibGraphes` class to allocate registers (or a place in memory). For that you have to complete the program method `smart_alloc`. Comments will help you design this (non trivial) function. The allocation itself is done by statement rewriting, like in previous lab. You need to implement it in `Allocations.py` (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.mu` example, the graph coloring succeeds with:

[3] $k = 8$ this year!



EXERCISE #7 ▶ **Massive tests**
Comment out all the print dot instructions, debug, ... and test on all test files you have.