Lab 6
Code generation with smart IRs

Objective

- Construct a CFG, and 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 LEIA code. You will have 2 sessions for that.

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-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/"
```

6.1 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**
We adapted `APICodeLEIA` to be able to deal with CFGs. Now `Instructions` have a list of predecessors (`self._in`) and successors (`self._out`) and a `LEIAProg` contains the initial control point (`self._start`) from which we can traverse the graph. This new feature allows us to easily construct the CFG of a program.

Constructing the graph consists in minor modifications of the code generation visitor you made in Lab 5. We give you the construction for all idioms except for the `while` loop **in the visitor code (one TODO)**.

In order to print the CFG, `Main.py` already contains a call to the (`printDot`) function that generates a dot file from the CFG. This `dot` file and its corresponding `pdf` file will be generated next to the `mu` input file.

Uncomment the "print Dot" line in `Main.py` file and:

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. Encode the construction for `while` loops, and draw nice pictures!

6.2 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 `APICodeLEIA.py`

EXERCISE #3 ▶ **Liveness Analysis, Initialisation**
Initialise the $Gen(B)$ and $Kill(B)$ for each kind of instruction (add, let, ...). This corresponds to the 11 TODOs `ADD GEN KILL INIT IF REQUIRED`.
We give you an example for the print instruction. Be careful to properly handle the following cases:

```
1 add temp1 temp1 12
```

and

```
.let temp1 42
```

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

```
pc = 0
gen: {}
kill: {}

pc = 1
gen: {}
kill: {temp_2}

pc = 2
gen: {temp_2}
kill: {temp_1}

pc = 3
gen: {}
kill: {}

pc = 6
gen: {}
kill: {temp_3}

pc = 8
gen: {}
kill: {temp_4}

pc = 9
gen: {temp_1,temp_3}
kill: {}

pc = 10
gen: {}
kill: {temp_4}

pc = 7
gen: {}
kill: {}

pc = 11
gen: {temp_4}
kill: {}

pc = 12
gen: {}
kill: {temp_5}

pc = 13
gen: {temp_5}
kill: {temp_1}

pc = 14
gen: {}
kill: {}

pc = 5
gen: {}
kill: {}

pc = 15
gen: {}
kill: {temp_6}

pc = 16
gen: {temp_6}
kill: {temp_1}

pc = 17
gen: {r7}
kill: {r7}

pc = 4
gen: {}
kill: {}
```

The exercise that follows is the most important of the Lab

EXERCISE #4 ► **Liveness Analysis, fixpoint (**)**

Implement the fixpoint iteration as a method (`doDataflow`) in `APICodeLEIA.py` "while it is not finished, store the old values, do an iteration, decide if its finished". The `doDataflow` program method should make calls to `do_dataflow_onestep` instruction methods (which is given). To perform set comparison you can have a look there:

`https://docs.python.org/3/library/stdtypes.html?highlight=set#set`

Carefully check that your results are correct at least with the examples of the `testsdataflow/` directory. As an example, here is the expected output for `testsdataflow/df04.mu`: [1]

```
In: {0: {}, 1: {}, 2: {temp_2}, 3: {temp_1}, 4: {}, 5: {r7},
6: {temp_1}, 7: {r7,temp_4}, 8: {temp_3,temp_1},
9:{r7,temp_3,temp_4,temp_1}, 10: {r7}, 11: {r7,temp_4}, 12: {},
13:{temp_5}, 14: {}, 15: {r7}, 16: {temp_6,r7}, 17: {r7}}

Out: {0: {}, 1: {temp_2}, 2: {temp_1}, 3: {temp_1}, 4: {},
5: {r7}, 6:{temp_3,temp_1}, 7: {r7,temp_4},
8: {temp_3,temp_4,temp_1}, 9:{r7,temp_4}, 10: {r7,temp_4},
11: {r7}, 12: {temp_5}, 13: {}, 14: {}, 15: {temp_6,r7},
 16: {r7}, 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
```

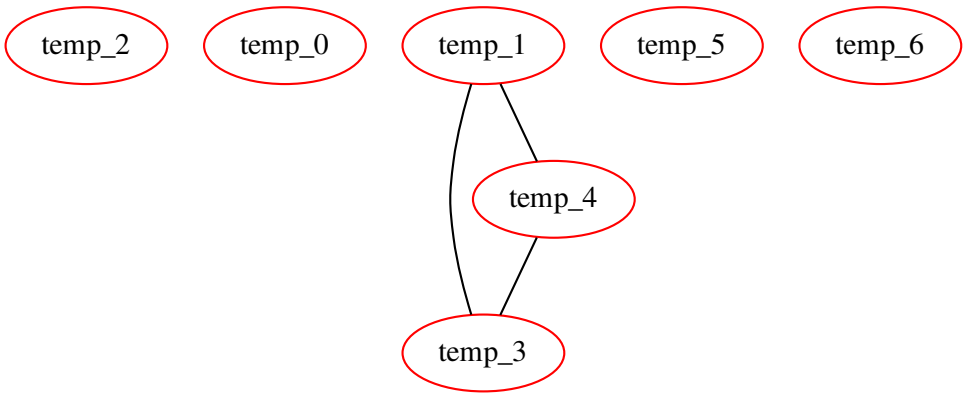
where x is not alive after the `x=1` statement, however x is in conflict with y since we generate the code for `x=1` while y is alive[2].

From the result of the previous exercise, construct the interference graph 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.

As an example, here is the conflict graph that should be obtained for `df04.mu`

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

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



6.3 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 $k-3$ colors.
- All the other variables will be allocated on the stack. To compute the offset from the stack pointer ($r6$), recolor the subgraph of remaining variables with an infinite number of colors.

Then the memory allocation:

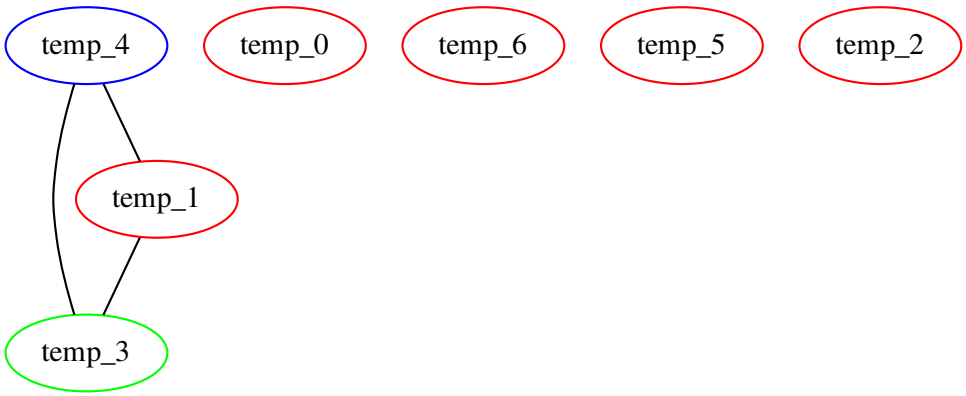
- For non-spilled variable: replace the temporary with its associated color/register.
- For spilled variables: do the same as in Lab 5!

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` and make a call to `self._pool.smart_alloc(coloring, ...)`. The allocation itself is done by statement rewriting, like in Lab5. You do not have to modify `Allocations.py`.

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.

6.4 Bonus: to go further

If you have time, you can choose among the following improvements for your compiler.

EXERCISE #8 ▶ **Chains**
Find a way to handle 'non int' log instructions:

- First, constant chains that will be stored in memory.

```
call clearscr
.let r4 1
.let r0 0x0000
.let r1 10
.let r2 95
.set r3 HELLO
call putstr                    ; putstr code is in lib.s
refresh

jump 0

HELLO:
        .string "Hello"

#include lib.s
```

  prints "Hello." in the graphical interface of the simulator (use "fast" option).
- Then, numerical values computed in a given register (you may have to store its value somewhere in the memory).
- And finally all log instructions.
- If you want to print a char, you must store its (ASCII) value in the r_3 register and use the `print_char` system call to print it.

EXERCISE #9 ▶ **Constant propagation**
Design and implement a "constant propagation" dataflow algorithm. Design new examples to test your optimisation.

EXERCISE #10 ▸ **Multiplication**
Implement a multiplication routine, and produce the code for the multiplication that calls this routine.