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 have written a dummy code generator for the Mu language. In this lab the objective is to generate a more efficient LC3 code. You will have 3 sessions for that. Your code is due by email to your **two** teaching assistants on **December, 9th** (code, readme, testfiles, makefile and scripts if any).

First download the archive from the course website.

Installations You may have to install the following PYTHON packages:

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

and also (on your machines):

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

6.1 CFG Construction and liveness analysis

EXERCISE #1 ► **CFG**

We give you an API for the CFG construction. Contrarily to the course, a block will be a unique LC-3 instruction (or a label). Adapt the visitor of the previous lab (the first one) to construct the CFG of your program as follows:

```
#self._prog.addInstructionNOT(dr,reg)
self._cfg.append(BlockNOT(dr,reg))
```

In the visitProgRule, you should have the following instructions:

```
self._cfg = CFG(BlockPROG())
self.visit(ctx.block())
```

and `Main.py` already contains a call to the function that prints a dot file from the CFG (A `dot` file and its corresponding `pdf` file must be generated next to the `mu` input file)

1. First, implement and test for lists of assignments. You should see a chain of blocks.
2. For branchs, loops, it is a bit more complicated...

To do that, you will need to proceed as in the following example:

```
# We have a branch!
blockBRn = self._cfg.append(BlockBR("n",labelfalse))

# We create the true and false branches
blockTrue = blockBRn.append(BlockLabel(labeltrue)) # TRUE case comes
    first
```

```
blockFalse = blockBRn.append(BlockLabel(labelfalse))

# TRUE case:
self._cfg.setEnd(blockTrue) # The end of the CFG now points to
   blockTrue
self._cfg.append(...)
endTrue = self._cfg.append(BlockGOTO(labelend)) # When done, we jump to
    labelend

# FALSE case:
self._cfg.setEnd(blockFalse)
endFalse = self._cfg.append(...)

# Finally, we merge the branches
blockLabelend = BlockLABEL(labelend) # Must be the last block created
self._cfg.setEnd(endTrue).append(blockLabelend)
self._cfg.setEnd(endFalse).append(blockLabelend)
```

Additionally, you must respect the following rules (due to how code generation works using the CFG):

- **The BR instruction jumps to the false case (in this example), so we must do the true case FIRST**
- **The BlockLABEL(labelend) must be the LAST block created (due to internal id incrementation)**

Note that the end of the CFG is now blockLabelend. **You have to make a demo of this construction to your teaching assistants on Thursday, Nov, 24th**.

Your demo should at least work on the following three programs:

```
x=1;
y=2+x;
z=x+y;
x=7;
```

```
x=2;
if (x < 4)
   x=4;
else
   x=5;
y=x+1;
```

```
x=0;
while (x < 4){
   x=x+1;
}
y=x+3;
z=y+x;
```

EXERCISE #2 ▸ Liveness Analysis

For the liveness analysis, in the `CFG.py` file we give you a function that performs one iteration of the Dataflow analysis for liveness. You have to:

- Initialise the $Gen(B)$ and $Kill(B)$ for each block (statement or comment). Be careful to properly handle the following cases:

  ```
  ADD temp1 temp1 12
  ```

  and

  ```
  AND temp1 temp1 0
  ```

- Implement the fixpoint iteration as a method in `Cfg.py`“while it is not finished, store the old values, do an iteration, decide if its finished”.

To test the liveness analysis, you'll have to invent relevant tests.

EXERCISE #3 ▸ Interference graph

We recall that two temporaries 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[1].

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 (`LibGraphes.py`) for that. Use the `print_dot` method and relevant tests to validate your code. *You may have to change a bit the CFG API to collect all the temporaries during its construction..*

6.2 Register allocation and code production

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

- Color the graph with $k-3$ colors ($r0$ to $r4$).
- 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 code generation:

- For non-spilled variable: replace the temporary with its associated color/register.
- For a spilled variable (say, $temp5$ here):
  `ADD temp6 temp1 temp5`
  becomes (we use $r5$ and $r7$ to make load and stores for spilled variables):
  `LDR R5 R6 #-dec`
  `ADD alloc(temp6) R5 alloc(temp5)`
  (this is why we need to color with $k-3$ registers).

EXERCISE #4 ► **Register Allocation**
Use the algorithm (with k=8) and the coloration method of the `LibGraphes` class to allocate registers (or a place in memory). Validate your allocation on tiny test files that do not need more than 5 physical registers.

EXERCISE #5 ► **Final Code Generation**
We are nearly done! Modify the `CFG` print method to be able to replace temporaries with their new place, and test your generated asm files.

6.3 Bonus: to go further

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

EXERCISE #6 ► **Optimise the test process!**
Use the LC3 command line generator and scripts to perform your tests:

`https://highered.mheducation.com/sites/0072467509/student_view0/lc-3_simulator.html`

You can get inspiration from this webpage:

`https://www.cs.colostate.edu/~fsieker/misc/lc3.html`

EXERCISE #7 ► **Big constants**
Find a way to handle numerical constants that are two big to be stored in 5 bits.

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

EXERCISE #8 ► **Chains**

Find a way to handle log instructions:

- First, constant chains that will be stored in memory.
  ```
  LEA R0, mychain ; in R0 only
  PUTS  #print
  ...
  mychain: .STRINGZ "Hello"
  ```
  prints “Hello.”.
- Then, numerical values computed in a given register (you may have to store it somewhere).
- And finally all log instructions.
- If you want to print a char, you must store its (ASCII) value in the $R0$ register and use the `OUT` 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 ► **Register Allocation by iterative coloring**

Algorithm 1: Register allocation – a less naive version

Algorithm `Allocate`(CFG, k)
  Build interference graph G
  Color(G,$k-2$) $\rightsquigarrow$ Alloc, $G = G_{ref} \cup G_m$
  while $G_m \neq \emptyset$ **do**
    spill ← spill $\cup\, G_m$
    Generate the code with spill code
    Build interference graph G'
    Colour(G',k) $\rightsquigarrow$ $G' = G'_{reg} \cup G_m$
  end

The previous allocation freezes two registers, this we color with k-2 registers. What if we need k registers? Show that it can happen. Use the iterative algorithm 1 to produce the code

EXERCISE #11 ► **Multiplication**

Implement a multiplication routine, and produce the code for the multiplication that calls this routine.