Lab 4
Syntax-Directed Code Generation

Objective

During the previous lab, you have written your own evaluator of the Mu language. In this lab the objective is to generate *valid* LEIA codes from Mu programs:

- Generate 3-address code for the Mu language.
- Generate executable "dummy" LEIA from programs in Mu via two simple allocation algorithms.
- **Please follow instructions and COMMENT YOUR CODE!**

Student files are in the GIT repository.

Your work is due on Tomuss on Tuesday the 12th of December (before midnight).
Rename your TP05 directory into NameSurnameTP05 and send it as an archive (after make clean) without the tests in `tests/step1` and `tests/step2` directories (or use make tar). Your code must compile and sucessfully pass your own tests in `tests/mine`. Please also fill the provided `Readme` file.

Important remark From now on, we add some restrictions to our language:

- Variables are of type (signed) `int` or `bool` only (no float, no string, **no char**). Thus all values can be stored in regular registers or in one cell in memory. You can let your program crash if an other type of variable is provided.
- The only use of strings are inside `log` instructions:
  - `log("this is a message")` is valid but not `u="mymessage"`.
  - `log(x)` is also valid whevener *x* has a value (`int` or `bool`).

Structure of the code

- In `APICodeLEIA.py` we provide you with utility functions to encode 3-address LEIA instructions. Instruction classes are in `Instruction3A.py` and `Operands.py`. An Instruction is either a Comment, a Label, or a `Instru3A`; it has arguments which can be immediate numbers (of type `Immediate`), temporaries (of type `VirtualRegister`), regular registers (`Register`), offsets in memory (`Offset`).

- A LEIA program contains a list of instructions, and also a virtual register pool (temporary variables).

- In Section 4.1, you will use an instance of the `LEIAProg` in order to construct a list of such instructions via calls to `addInstructionXXX` methods. A call to the `printCode` method will dump this code into a text file.

- File `Allocation.py` is responsible of the allocation part. From a `LEIAProg` with temporaries (instructions formed with virtual registers), producing an actual LEIA program (instructions with regular registers or memory acesses) is done by:

  - First, compute an allocation for each temporary (in the current `LEIAProg` instance). In Section 4.2, we provide you with `LEIAProg.naive_alloc()` which computes such a (naive) allocation, you will have to design your own allocation function in Section 4.3.
  - For each instruction of the program, if the instruction contains a read or write access to a temporary, replace operands with the corresponding actual registers/memory location (and possibly add some instructions before and after). This is done by the use of the `LEIAProg.iter_instructions` iterator on instructions and `Allocations.replace_reg` methods. In Section 4.3 you will have to write such a "replacement" function.

- The file `Main.py` launches the chain: production of 3-address code with temporaries, allocation, replacement, print.

- The script `test_codegen.py` will help you to test your code. We will use it in Section 4.2.
- A `Readme.md` file to be completed progressively during the lab.

EXERCISE #1 ► LEIA **Simulator**
Git pull, then recompile the LEIA simulator in the `leia/simulateur/` directory and test its command-line version (we only need this!):

```
$ python3 assembleur/asm.py EX/test_print.s
$./simulateur/LEIA -q test_print.obj
this is a test of the print function. you should see 42 and 666 below:
42
666
```

If you encounter some issues with a graphical library, edit the (simulator) Makefile and comment the line:

```
WITHSDL=1  # comment to avoid SDL dependence.
```

4.1 Three-address code generation

In this section you have to implement the course rules (Figures 4.1 and 4.2) in order to produce LEIA code with temporaries.

Here is an example of the expected output of this part. From the following Mu program:

```
var a,n:int;
n=1;
a=7;
while (n<a) {
 n= n+1;
}
log(n);
```

the following code is supposed to be generated:

```
;;Automatically generated LEIA code, 2017
;;(stat  (assignment n = (expr (atom 1)) ;))
.let  r2  1
copy r0 r2
;;(stat  (assignment a = (expr (atom 7)) ;))
.let  r3  7
copy r1 r3
;;(stat  (while_stat while (expr (atom ( (expr (expr (atom n)) < (expr (atom a))) )))  (stat_block {  (block (stat  (
      assignment n = (expr (expr (atom n)) + (expr (atom 1))) ;)))  }) ))
l_while_begin_0:
snif r0  slt  r1
jump l_cond_neg_1
letl  r4  1
jump l_cond_end_1
l_cond_neg_1:
letl  r4  0
l_cond_end_1:
snif  r4  eq 1
jump l_while_end_0
;;(stat  (assignment n = (expr (expr (atom n)) + (expr (atom 1))) ;))
.let  r5  1
add r6 r0 r5
copy r0 r6
```

jump l_while_begin_0
l_while_end_0:
jump 0

EXERCISE #2 ► **3-address code generation**
In the archive, we provide you a main and an incomplete `MyMuCodeGenVisitor.py`. To test it, type

```
make FOO=tests/step1/test01.mu
```

and observe the generated code in `<samepath>/test01.s`[1]. You now have to implement the 3-address code generation rules seen in the course. Code and test incrementally [2]:
- the printing instruction `log` for scalar variables (chars and strings are optional) (we recall that there is a native `print` instruction in the LEIA assembly).
- numerical expressions without variables (constants are expected to hold on 16 bits).
- then (numerical) assignments and expressions with variables; PowExpr and MultiplicativeExpr are bonus, implement them only if after everything else is working.

At this step, the code generation is not finished, but we will do some allocation so that to be able to test properly. All examples in `tests/step1` directory should generate code without any error at this point:

```
for i in tests/step1/*.mu; do echo "file="$i; python3 Main.py $i > /dev/null; done
```

4.2 Testing with the trivial allocator, end of code generation.

The former code is not executable since it uses temporaries. We provide you with an allocation method which allocates temporaries in registers as long as possible, and fails if there is no available registers. The process takes as input the former 3-address code and transforms each instruction according to the allocation function.

EXERCISE #3 ► **Testing the trivial allocator**
Open, read, understand the `prog.naive_alloc()` implementation in `APICodeLEIA` and `Allocations.py` and how it is used to perform the actual LEIA code generation. Then, intensively test your former code generation with this allocator [3]:

1. Have a look at the `test_codegen.py` script: comment or uncomment files to test, and what to test.
2. Test with:

   ```
   python3 test_codegen.py
   ```

   This script tests all files in the `test/*` directories:

   - if the pragma `#EXPECTED` is present in the file, it compares the actual output after assembling and simulating with the list of expected values. For instance:

     ```
     var x,y:int;
     x = 42;
     log x
     y = x + 8;
     log(y)
     # EXPECTED
     # 42
     # 50
     ```

     is a great test case to test assignments.

[1] We generated LEIA comments with Mu statements for debug.
[2] Using files in the `TP04/tests/*` directories. All the test files you use will have to be in your archive.
[3] Be careful, this allocator crashes if there is more than 8 temporaries !

  - In any cases, it compares the actual output after assembling and simulating to the output given by your evaluator of the Mu Language (Lab 3). **If your evaluator is buggy, you can decide either to correct your bugs or to comment appropriate lines in the Python script.**

At this step, the tests should be ok for all files given in directory `tests/step1/`:

```
make tests
=========================== 8 passed in xx seconds ========
```

Now that we have a way to test our code generation for tiny Mu codes, we can come back to it.

EXERCISE #4 ► **End of 3-address code generation for Mu**
Implement the 3-address code generation rules:

- for boolean expressions and numerical comparison: compute 1 (true) or 0 (false) in the destination register;
- while loops;
- if then else. **Be careful with nested ifs and their labels!**.

At this point all the tests should be ok for all files in directory `tests/step2/` (You should modify the test script pathes). However these tests are not sufficient, you should add some other ones (in the directory `tests/mine/`).

About tests For tests (and boolean expressions), make sure you generate "conditional jumps" with:

```
self._prog.addInstructionCondJUMP(label, op1, cond, op2)
```

where `op1` (resp `op2`) is the left operand (resp right operand), ie a register or a value of the boolean condition (`Condition('eq')` for equality, for instance), and `label` is a label to jump to if the condition evaluates to true. Later on (while printing), this instruction will expand itself to a regular `snif`.

Be also careful to avoid "jump to next line" because the LEIA machine doesn't allow this instruction. You might have to add dummy instruction like this to ensure to jump at least two instructions below:

```
self._prog.addInstructionJUMP(lend_if)
self._prog.addInstructionSUB(R7, R7, 0)) # dummy instruction
self._prog.addLabel(lend_if)
```

About nested if-then-else There is an issue with nested ifs. Indeed, how can we remember where to jump after one CondBlock (in `visitCondBlock(self, ctx)`)? We propose to use a label stack called `self.ctx_stack`: each time we enter `visitIfStat`, we push the end label. This label is used in all `visitCondBlock` (at some point you have to insert a jump instruction to the `cond_if` label). At the end of the `visitIfStat` function this label is poped out.

4.3 LEIA **code with "all-stack" allocation of temporaries**

As the number of registers is only 16, we have to find a way to store the results elsewhere. In this particular lab, we will use the following solution:

- for a given expression/instruction rule, the generated code can use r_2 to r_5 registers instead of temporaries;
- but all values that are propagated from one rule to another (subexpressions, ...) must be stored in the stack, whose address will be stored in r_6 (as defined in LEIAProg.printCode).
- r_0 will be used to compute the actual addresses from the base register r_6.
- r_1 will be used to compute the value to store or as a destination register for the value to read.

Following the convention that r_6 always stores the "begining of stack address", pushing[4] the content of r_1 in the stack will be done following the steps:

- compute a new offset (call to the `new_offset` method of the class `LEIAProg`).

[4] Please do not use the assembly macros push and pop that do not follow our conventions!

- generate the following instructions:

```
SUB r0 r6 <valueofoffset>
WMEM r1 [r0]
```

Be careful with the size of the offsetvalue!

EXERCISE #5 ▸ **Manual translation**
Complete the expected output for the following two statements (15 lines of LEIA code):

```
var x,y:int;
x=4;
y=12+x
```

Listing 4.1: 'all in mem alloc for test00b.mu'

```
;;Automatically generated LEIA code, 2017
;; "All-in-memory allocation" version
;stack management
.set r6 stack

        ;; (stat (assignment x = (expr (atom 4)) ;))
        ;; .let temp_2 4
        .LET r1 4
        SUB r0 r6 2
        WMEM r1 [r0]
        ;; end .let temp_2 4
        ;; copy temp_1 temp_2
        SUB r0 r6 2
        RMEM r1 [r0]
        COPY r1 r1
        SUB r0 r6 3
        WMEM r1 [r0]
        ;; end copy temp_1 temp_2
        ;; (stat (assignment y = (expr (expr (atom 12)) + (expr (atom x))) ;))

        ;;  <complete here>

        ;; ...


;;postlude
jump 0
.align16
stackend:
.reserve 42
stack:
```

EXERCISE #6 ▸ **Implement**
Now you are on your own to implement this code generation. Here are the main steps (less than 50 locs of PYTHON):

1. We have implemented for you an `alloc_to_mem(self)` method in `APICodeLEIA.py`. This method only maps each temporary ("virtual register") to a new offset in memory (in a PYTHON `dict`), then iterates the `replace_mem` function on all instructions of the three adress program to perform the actual allocation.

2. In `Allocations.py`, implement a `replace_mem(old_i)` that takes as input a "3-address with temporaries" LEIA code and outputs a list of instructions as a replacement. For instance, each time we access a source operand, we have to load it from memory before, thus the `replace_mem` should contains lines like:

```
after.append(Instru3A('SUB', R0, R6, offset))
after.append(Instru3A('WMEM', R1, Indirect(R0)))
```

The files you generate have to be tested with the LEIA simulator with the same script as ebfore.

EXERCISE #7 ► **Bonus**
Implement an hybrid version that allocates temporaries (virtual registers) in actual registers as long as possible, then in memory. You can use all r_0 to r_{15} registers, but be careful to avoid conflicts!

EXERCISE #8 ► **Bonus**
Implement the 3-address code for PowExpr and MultiplicativeExpr.

c	`dr <-newTemp()`
	`code.add(InstructionLETL(dr, c))`
	`return dr`

x	`#get the place associated to x.`
	`regval<-getTemp(x)`
	`return regval`

e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionADD(dr, t1, t2))`
	`return dr`

e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionSUB(dr, t1, t2))`
	`return dr`

true	`dr <-newTemp()`
	`code.add(InstructionLETL(dr, 1))`
	`return dr`

$e_1 < e_2$	`dr <- newTemp()`
	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`endrel <- newLabel()`
	`code.add(InstructionLET(dr, 0))`
	`#if t1>=t2 jump to endrel`
	`code.add(InstructionCondJUMP(endrel, t1, ">=" , t2)`
	`code.add(InstructionLET(dr, 1))`
	`code.addLabel(endrel)`
	`return dr`

Figure 4.1: 3@ Code generation for numerical or Boolean expressions (t1 and t2 are already defined)

x = e	`  dr <- GenCodeExpr(e)`
	`#a code to compute e has been generated`
	`  if x has a location loc:`
	`      code.add(instructionCOPY(loc,dr))`
	`  else:`
	`      storeLocation(x,dr)`

S1; S2	`#concat codes`
	`   GenCodeSmt(S1)`
	`   GenCodeSmt(S2)`

`if` b `then` $S1$ `else` $S2$	`lelse,lendif <-newLabels()`
	`t1 <- GenCodeExpr(b)`
	`#if the condition is false, jump to else`
	`code.add(InstructionCondJUMP(lelse, t1, "=", 0))`
	`GenCodeSmt(S1) #then`
	`code.add(InstructionJUMP(lendif))`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) #else`
	`code.addLabel(lendif)`

`while` b `do` S `done`	`ltest,lendwhile <-newLabels()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`code.add(InstructionCondJUMP(lendwhile, t1, "=", 0))`
	`GenCodeSmt(S)                       #execute S`
	`code.add(InstructionJUMP(ltest))    #and jump to the test`
	`code.addLabel(lendwhile)            #else it is done.`

Figure 4.2: 3@ Code generation for Statements