Lab 5
Syntax-Directed Code Generation

Objective

- Generate 3-address code for the Mu language.
- Generate executable "dummy" LEIA from programs in Mu via 2 simple allocation algorithms.

Student files are in the GIT repository. Start by installing the networkx and graphviz modules if necessary:

```
pip3 install networkx graphviz --user
```

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. You will have 2 sessions for that.
Your work is due on Thursday the 2nd of November at noon.
Rename your TP05 directory into NameSurnameTP05 and send it as an archive (after make clean). Your code must compile and sucessfully pass your own tests. Please also provide a Readme that explains: what is working or not, what kind of tests you made, any comments on your implementation that can help understanding your code.

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 you 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`).

  This feature will not be cost you much if it is not implemented

Structure of the code

- In `APICodeLEIA.py` we provide you with utility functions to encode 3-address LEIA instructions. 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`). In Section 5.1, you will have 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 5.2, we provide you with `LEIAProg.naive_alloc()` which computes such a (naive) allocation, you will have to design your own allocation function in Section 5.3.
  - 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 5.3 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 5.2

5.1 Three-address code

In this section you have to implement the course rules (Figures 5.1 and 5.2) in order to produce LEIA code with temporaries. The LEIA documentation can be found in Appendix A.

EXERCISE #1 ▶ **Manual translation**
Give a (LEIA) three-address code for the following Mu program:

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

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/test01.mu
```

and observe the generated code in `../tests/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` (we recall that there is a `print` native instruction in the LEIA assembly).
- numerical expressions without variables (constants are expected to hold on 16 bits).
- then (numerical) assignements and expressions with variables; PowExpr and MultiplicativeExpr are bonus, implement them only if after everything else is working.
- then boolean expressions: compute 1 (true) or 0 (false) in the destination register;
- while;
- if then else. **Be careful with nested ifs and their labels!** To help you, you can use the class `CodeGenContext` provided in the `APICodeLEIA`. It allows you to remember where you are inside imbricated ifs. Your code will then look like:

```
def visitIfStat(...):
    if_ctx = CodeGenContext() # Create context
    self.ctx_stack.append(if_ctx) # and push it on stack
    if_ctx.end_label = ... # fill-in the context with
                           # what you want
    ... self.visit(...) ...
    self.ctx_stack.pop() # Remove the context from stack
```

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(R0, R0, 0)) # dummy instruction
self._prog.addLabel(lend_if)
```

[1] We generated LEIA comments with Mu statements for debug.
[2] Using files in the `TP04/ex/` and `TP04/ex/stud-ok/` directories. All the test files you use will have to be in your archive.

5.2 Testing with the trivial allocator

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 `alloc_naive(src_prog)` implementation in `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. Uncomment the following lines in `Main.py`:

   ```
   if naive_alloc:
       alloc_naive(prog)
   ```

2. Test with:

   ```
   python3 test_codegen.py
   ```

   This script tests all files in the `test/` directory:

   - 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:

     ```
     log 1 < 2;
     log 1 < 1;
     log 1 > 2;
     log 1 > 1;
     # EXPECTED
     # 1
     # 0
     # 0
     # 0
     ```

     is a great test case for the comparison operator.
   - In any cases, it compares the actual output after assembling and simulating to the output given by your evaluator of the Mu Language (Lab 4). **If your evaluator is buggy, you can decide either to correct your bugs or to comment appropriate lines in the python scripts.**

5.3 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, which 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`).

[3] Be careful, this allocator crashes if there is more than 8 temporaries !
[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 #4 ▶ **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 5.1: '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 #5 ▶ **Implement**
Now you are on your own to implement this code generation. Here are the main steps (less than 50 locs of PYTHON):

1. Implement a `alloc_to_mem(self)` method in `APICodeLEIA.py`. This method only maps each temporary ("virtual register") to a new offset in memory.
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.

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

EXERCISE #6 ▶ **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 #7 ▶ **Bonus**
Implement the 3-address code for ParExpr 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 5.1: 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 *S*1 else *S*2	`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 5.2: 3@ Code generation for Statements

Appendix A
LEIA Assembly Documentation (ISA)

Source:

- ISA: Florent de Dinechin, Nicolas Louvet, Antoine Plet, for ASR1, ENSL, 2016.
- Simulator: Pierre Oechsel and Guillaume Duboc, L3 students at ENSL, 2016.

A.1 Installing the simulator and getting started

To get the LEIA assembler and simulator, follow instructions of the first Lab (git pull on the course lab repository).

A.2 The LEIA architecture

Here is an example of LEIA assembly code for 2017:

```
  letl r0 17     ; initialisation of  a register
loop:
  wmem r13 [r0]  ; write in memory
  rmem r13 [r2]  ; read in memory
  add r0 r10 r11 ; add
  snif r0 eq 3   ; test : if r0 = 3 skip next instruction
  jump loop  ; equivalent to jump -3, and this is a comment
  xor r0 r0 -1
```

Memory, Registers The memory is shared into words of 16 bits, with address of size 16 bits (from $(0000)_H$ to $(FFFF)_H$).

The LEIA has 16 generalistic registers. Only R15[1] is reserved for the routine return address. They are also specific 16 bits registers: PC (*Program Counter*), IR (*Instruction Register*).

Constants: `leth` and `letl` These expressions provide ways to initialize registers. The constant is encoded in the bits 0 to 7. For the `letl` instruction, bit 7 (sign bit) of the constant is replicated into the bits 8 to 15 of the destination register. Thus:

```
letl r0 xx
```

stores the constant xx in register r0, provided xx between -128 and 127. The `leth` instruction stores the 8 bit constant in the bits 8 to 15 of the destination register, the other bits being unchanged. Thus:

```
letl r0 2
leth r0 3
```

stores in r0 the constant $2+3*2^8 = 770$. The LEIA assembler tool provides a macro:

```
.let r0 770
```

to generate these two instructions automatically.

[1] registers are indifferently in capital letters or in lower case.

Table A.1: All LEIA instructions

15	14	13	12	mnemonic	class	description	ext(i)
0	0	0	0	`wmem`	wmem	write to memory	
0	0	0	1	`add`	ALU	addition	$z(i)$
0	0	1	0	`sub`	ALU	subtraction	$z(i)$
0	0	1	1	`snif`	snif	skip next if	$s(i)$
0	1	0	0	`and`	ALU	logical bitwise and	$s(i)$
0	1	0	1	`or`	ALU	logical bitwise or	$s(i)$
0	1	1	0	`xor`	ALU	logical bitwise xor	$s(i)$
0	1	1	1	`lsl`	ALU	logical shift left	$z(i)$
1	0	0	0	`lsr`	ALU	logical shift right	$z(i)$
1	0	0	1	`asr`	ALU	arithmetic shift right	$z(i)$
1	0	1	0	`call`	call	sub-routine call	
1	0	1	1	`jump` `return`	jump	relative jump if `offset` $\neq 1$ return from call if `offset` = 1	
1	1	0	0	`letl`	letl	8-bit constant to Rd, sign-extended	
1	1	0	1	`leth`	leth	8-bit constant to high half of Rd	
1	1	1	0	`print`	print	print or refresh	
1	1	1	1	`rmem` `copy`	rmem	read from memory if i=0 register-to-register copy if i=1	

notation	meaning
d	a 3 or 4 bit number that specifies the destination register
i	a 4-bit number (bits 4 to 7 of the instruction word), the number of the first operand register
j	a 4-bit number

Table A.2: Notations

Arithmetical and logical instructions Arithmetical and logical instructions have 3 operands:

```
add r1 r0 3     ; add immediate
add r1 r2 r1    ; add registers
```

The first operand is the destination register, and the two remaining operands are sources: either two registers (if the bit 11 is 0) or a register and an immediate constant j of 4 bits (if the bit 11 is 1). Because of the restricted number of bits to describe the first operand, the **destination register can only be one of the first eight registers** (from r0 to r7). If a constant is used then it is extended into a 16 bit constant before the operation. This is documented in the last column of table A.1:

- $z(j)$ means that j is extended with zeros. In other words j is interpreted as a *positive integer*.
- $s(j)$ means that the bit 3 (sign bit) of j is replicated into bits 4 to 15: j is interpreted as a *signed integer* and is transformed into a 16 bits integer of the same value.

Thus the result of the instruction:

```
add r1 r0 -1
```

is not really what is expected. The constant $j = -1$ is encoded as `1111`, extended as $z(j)$ =`0000000000001111`, thus the sum should be done with the 31 constant. **The assembler tool throws an error in that case:**

```
instruction add: Number, Not in bound: [0, 15]
```

Branching Let `a` be the instruction's address, and `c` the integer encoded in the bits 0 to 11 of the instruction's word. The `call` instruction makes a copy of `a` + 1 into r_{15} then executes `pc` ← `c` × 16. **Thus procedures should have addresses that are multiple of 16.**

The `jump` instruction considers the constant `c` as a signed integer (thus between -2048 and 2047) and executes `pc` ← `a` + `c` except if `c` = 1, in which case it executes `pc` ← r_{15}. In this case we can use the mnemonic `return`.

Table A.3: Encoding per instruction class

class	action	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ALU reg	$r_d \leftarrow r_i$ op r_j	opcode				0	d			i				j			
ALU imm4	$r_d \leftarrow r_i$ op ext(j)	opcode				1	d			i				j			
snif	skip next if	0	0	1	1	$c/\overline{r}$	condition			i				j			
letl	$r_d \leftarrow s(b)$	1	1	0	0	d				b							
leth	$r_d[15..8] \leftarrow b$	1	1	0	1	d				b							
call	jump to the routine	1	0	1	0	c											
jump	jump	1	0	1	1	c											
return	return to calling routine	1	0	1	1	0											1
wmem	mem$[r_j] \leftarrow r_i$	0	0	0	0	0	0	0	0	i				j			
rmem	$r_d \leftarrow$ mem$[r_j]$	1	1	1	1	d				0	0	0	0	j			
copy	$r_d \leftarrow r_j$	1	1	1	1	d				0	0	0	1	j			
print reg	print (the numerical content of) r_i	1	1	1	0	0	1	0	0	0				i			
print char	print c	1	1	1	0	1	0	0	0	$ascii(c)$							
refresh	wait	1	1	1	0	0											

Tests: `snif` "skip next if" The `snif op1 <condition> op2` instruction deactivates the next instruction if the condition is true. Operands 1 and 2 are encoded like in the ALU instructions. In particular the second operand can be an immediate constant, which sign will be extended. The condition is encoded thanks to the following table:

10	9	8	mnemonic	description
0	0	0	`eq`	equal, op1 = op2
0	0	1	`neq`	not equal, op1 ≠ op2
0	1	0	`sgt`	signed greater than, op1 > op2, two's complement
0	1	1	`slt`	signed smaller than, op1 < op2, two's complement
1	0	0	`gt`	op1 > op2, unsigned
1	0	1	`ge`	op1 ≥ op2, unsigned
1	1	0	`lt`	op1 < op2, unsigned
1	1	1	`le`	op1 ≤ op2, unsigned

Let us illustrate the difference between `sgt` et `gt`: if R_0 contains 0, then:

```
snif r0 gt -1
```

is false, but

```
snif r0 sgt -1
```

is true. In fact, the −1 constant is extended as `ffff` (hexa), which is interpreted as 65535 by `gt`, and -1 by `sgt`.

Memory accesses The memory address is always specified in the r_j register encoded in bits 0 to 3. The instruction `rmem rd [rj]` copies in the destination register (coded in bits 8 to 11) the content of the memory at address r_j. The instruction `wmem ri [rj]` copies the content of the register r_i (coded in bits 4 to 7) in the memory cell whose address is stored inside `rj`.

Register management Some registers cannot be used with arithmetic and logical instructions, yet it is possible to use them to store a result thanks to the `copy` instruction. This instruction is also usefull before function calls to quickly save registers that are known to be used by the function.

Print Two examples of use of the native print instruction:

```
print r1     ; prints the content of r1 (numerical value)
print 'z'    ; prints the character 'z'
```

Assembly directives A bit more of syntax:

- The assembly begins at address 0.
- Labels can be used for jumps. **Warning, for the compiler to work properly, do not type anything else than the label on its line, followed by a colon ':'.**
- The keyword `.word xxxx` reserves a memory cell initialized to the 16 bit constant xxxx.
- The keyword `.reserve xxxx` reserves n memory cells initialized to 0.
- The keyword `.string "Hello"` reserves 6 memory cells and store the ascii numbers corresponding to all the characters of the message (ending it with a Null character).
- The keyword `.align16` pads memory cells in order for the next line to be at an address multiple of 16.
- The macro `.let r3 585` stores the constant 585 in register 3 (see paragraph A.2)
- The macro `.set r3 label` loads the address corresponding to label onto r3. For instance, the following program:

```
  .set r0 foo
foo:
        .word 42
```

  is assembled into:

```
c002 ; letl r0 2 (because 42 is stored at line 2)
d000 ; leth r0 0
002a ; the 42 constant
```

From Lab 5 we will be using a stack. The address of its top will be stored in r_7 and we will use the following macros:

- The macro `.push ri` that pushes the content of the r_i register into the memory. It is equivalent to:

```
sub r7 r7 1
wmem ri [r7] ;
```

- The macro `.pop ri` that does the converse:

```
rmem ri [r7]
add r7 r7 1
```

A.3 Help to encode constants

hex to binary

a	b	c	d	e	f
1010	1011	1100	1101	1110	1111

2's complement Let us code $n = (-3)_{10}$ in 2's complement on 6 bits, with the recipe: "code -n in base 2, then negate bitwise, then add one". First, 3 is encoded as `000011` on 6 bits. Its negation is `111100`, thus $(-3)_{10} = 111101_{\bar{2}}$.

A.4 The graphical library

Coordinates of the screen start on the bottom left corner of the screen $({}_{(0,0)} \uparrow^{x}_{\rightarrow} {}_{y})$

- `clearscr`: does what it is supposed to do. Uses register r_1.
- `putstr`: puts a string on the screen at coordinates (r_1, r_2) ; the string address is in register r3 ; if r_4 is not 1 then refresh between each letter. Uses registers 1, 2, 3, 6, 14, 15 and those of `putchar`. An example can be found in Lab 1.
- `putchar`: puts a char on the screen at coordinates (r_1, r_2). Uses registers r1 to r6. An example can be found in Lab 1.
- `refresh`: refreshes the screen.