1 Small x86-64 assembly exercise (50%)

The goal of this assignment is to get some familiarity with x86-64 assembly language, by manually compiling small C programs.

An assembly code is written in a file with suffix `.s` and looks like this:

```
      .text
      .globl main
main:
      ...
      mov  $0, %rax       # exit code
      ret
      .data
      ...
```

You can compile and run such a program as follows:

```
gcc -g file.s -o file
./file
```

(Add the option `-no-pie` if you use `gcc` version 5 or later.) When needed, you can use `gdb` to execute your program step by step. Use the following commands

```
gdb ./file
(gdb) break main
(gdb) run
```

and then execute one step with command `step`. More information in this tutorial.

This page by Andrew Tolmach provides some information to write/debug x86-64 assembly code. These notes on x86-64 programming are really useful.

1.1 Printing using `printf`

Question 1.1 Compile the following C program:

```c
#include <stdio.h>

int main() {
    printf("n = %d\n", 42);
    return 0;
}
```

To call the library function `printf`, we pass its first argument (the format string) in register `%rdi` and its second argument (here the integer 42) in register `$rsi`, as specified by the calling conventions. We must also set register `%rax` to zero before calling `printf`, since it is a variadic function (in that case, `%rax` indicates the number of arguments passed in vector registers — here none).

The format string must be declared in the data segment (`.data`) using the directive `.string` that adds a trailing 0-character.

1.2 Arithmetic expressions

Question 1.2 Write assembler programs to evaluate and display results of the following arithmetic expressions:

- `4 + 6`
- `21 * 2`
- `4 + 7 / 2`
- `3 - 6 * (10 / 5)`

The expected results are

```
10
42
7
-9
```

To display an integer, you can use the solution of Exercise 1.1.

1.3 Boolean expressions

Question 1.3 Taking the convention that the integer `0` represents the Boolean value *false* and any other integer represents the value *true*, write assembly programs to evaluate and display the results of the following expressions (you must display `true` or `false` in the case of a Boolean result):

- `true && false`
- `if 3 <> 4 then 10 * 2 else 14`
- `2 = 3 || 4 <= 2 * 3`

The expected results are

```
false
20
true
```

It will be useful to write a `print_bool` function to display a boolean.

1.4 Global variables

Question 1.4 Write an assembly program that evaluates the following three instructions:

```
let x = 2
let y = x * x
print (y + x)
```

The variables `x` and `y` will be allocated in the data segment. The expected outcome is `6`.

1.5 Local variables

Question 1.5 Write an assembly program that evaluates the following program:

```
print (let x = 3 in x * x)
print (let x = 3 in (let y = x + x in x * y) + (let z = x + 3 in z / z))
```

We will allocate the variables `x`, `y` and `z` in the stack. The expected result is

```
9
19
```

2 Compilation of a mini-language (50%)

The purpose of this exercise is to produce a compiler for a mini-language of arithmetic expressions, called ARITH in this followed, towards the x86-64 assembler. A programming language ARITH is composed of a suite of instructions, which are either the introduction of a global variable with the syntax

```
set <ident> = <expr>
```

or the display of the value of an expression with the syntax

```
print <expr>
```

Here, `<ident>` denotes a variable name and `<expr>` an arithmetic expression. Arithmetic expressions can be constructed from integer constants, variables, addition, subtraction, multiplication, division, negation, parentheses, and a `let in` construct introducing a local variable. More formally, the syntax of arithmetic expressions is thus as follows:

```
<expr> ::= <integer constant>
         | <ident>
         | ( <expr> )
         | <expr> + <expr>
         | <expr> - <expr>
         | <expr> * <expr>
         | <expr> / <expr>
         | - <expr>
         | let <ident> = <expr> in <expr>
```

Here is an example of a program in the ARITH language:

```
set x = 1 + 2 + 3*4
print (let y = 10 in x + y)
```

Variable names are composed of letters and numbers and cannot begin with a number. The words `set`, `print`, `let` and `in` are reserved, i.e. they cannot be used as variable names. Operator precedence is as usual and the `let in` construct has the lowest precedence.

Preamble To help you build this compiler, we provide its basic structure (as a set of OCAML files `arithc.tar.gz`) that you can download in Teams. Once this archive is uncompressed with

```
tar zxvf arithc.tar.gz
```

you get an `arithc/` directory containing the following files:

`ast.ml`	arbitrary syntax of ARITH (completed)
`lexer.mll`	lexical analyzer (completed)
`parser.mly`	parser (completed)
`x86_64.mli, x86_64.ml`	for writing x86-64 code (completed)
`compile.ml`	the compilation itself (to be completed)
`arithc.ml`	the main program (completed)
`Makefile/dune`	to automate compilation (completed)

The provided code compiles (type `make`, which will launch a build) but it is incomplete: the assembly code produced is empty. You must complete the `compile.ml` file.

The program expects an ARITH file with the suffix `.exp`. When you do `make`, the program is launched on the `test.exp` file, which has the effect of producing a `test.s` file containing the assembly code, then the commands

```
    gcc -g -no-pie test.s -o test.out
    ./test.out
```

are launched. To debug, we can examine the content of `test.s` and if necessary use a debugger like `gdb` with the command `gdb ./test.out` then the step-by-step mode with `step`).

Note to MACOS users: you must modify the line `let mangle = mangle_none` in the provided `x86_64.ml` file, to replace it with `let mangle = mangle_leading_underscore`. You must also replace `let lab = abslab` with `let lab = rellab`.

Scheme of compilation We will carry out a simple compilation using the stack to store intermediate values (i.e. the values of subexpressions). Remember that an integer value takes up 8 bytes in memory. We can allocate 8 bytes on the stack by subtracting 8 from the value of `%rsp` or by using the `pushq` instruction.

Global variables will be allocated in the data segment (assembler `.data` directive; here it corresponds to the data field of type `X86_64.program`).

Local variables will be allocated on the stack. The space needed for all local variables will be allocated at program startup (by an appropriate subtraction on `%rsp`), after saving `%rbp`. The `%rbp` register will be positioned so as to point to the top of the space reserved for local variables.

```
         +-------------+
         | adr. retour |
         +-------------+
         | ancien %rbp |
 %rbp -> +-------------+  ^
         | var. locale |  |
         | var. locale |  |  frame_size (multiple de 16)
         | ...         |  |
 %rsp -> +-------------+  v
```

So any reference to a local variable will be relative to `%rbp`, with an offset of `-8, -16`, etc., depending on the variable.

Warning: before calling a library function like `printf`, the stack must be aligned to 16 bytes. Once the value of `frame_size` is determined, we therefore ensure that it is a multiple of 16 (see the code provided).

Exercises to do You have to read carefully the code in `compile.ml`. The parts to be completed, marked `(* to be completed *)` are the following:

1. The `compile_expr` function that compiles an arithmetic expression `e` into a sequence of x86-64 instructions whose effect is to place the value of `e` on the top of the stack. This function is defined using a local recursive function `comprec` that takes as arguments:
   - a parameter `env` of type `StrMap.t`: it is a dictionary indicating for each local variable its position on the stack (relative to `%rbp`);
   - a parameter `next` of type `int`: indicates the first free location for a local variable (relative to `%rbp`);
   - the expression to compile, on which a pattern matching is performed.
2. The `compile_instr` function that compiles an Arith instruction into a sequence of x86-64 instructions. In both cases (`set x = e` or `print e`), we must start by compiling `e`, then we find the value of `e` on the top of the stack (do not forget to `pop`).
3. The `compile_program` function that applies `compile_instr` to all the instructions of the program and adds code:
   - before, in particular to allocate space for local variables and set `%rbp`;
   - after, to restore the stack and terminate the program with `ret`.

Indications: we can proceed construction by construction, testing each time, in the following order:

1. constant expression `Cst`, instruction `Print` and exit with `ret`;
2. arithmetic operations (`Binop` constructor);
3. global variables (`Var` and `Set` constructors);
4. local variables (`Letin` and `Var` constructors).

We will finally test with the `test.exp` file (also provided), the result of which should be as follows:

```
60
50
0
10
55
60
20
43
```

==

Optional question To use a little less stack space, we can improve this compilation scheme a bit so that the result of `compile_expr` ends up in the `%rax` register rather than on top of the stack. In this way, only the results of left-hand side subexpressions end up on the stack.

==