Assignment 2: Compiler for $\mathcal{L}_{Var}$

(Deadline: 22.11.2021 09:00)

Exercise 1: Remove Complex Operands Pass

Implement the `remove_complex_operands` pass in `compiler.py`, creating auxiliary functions for each non-terminal in the grammar, i.e., `rco_exp` and `rco_stmt`. This pass translates $\mathcal{L}_{Var}$ to $\mathcal{L}^{mon}_{Var}$, shown below.

$$
\begin{array}{rcl}
atm & ::= & \texttt{Constant(}int\texttt{)} \mid \texttt{Name(}var\texttt{)} \\
exp & ::= & atm \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{UnaryOp(USub(),}atm\texttt{)} \mid \texttt{BinOp(}atm\texttt{,Add(),}atm\texttt{)} \\
 & \mid & \texttt{BinOp(}atm\texttt{,Sub(),}atm\texttt{)} \\
stmt & ::= & \texttt{Expr(Call(Name('print'),[}atm\texttt{]))} \mid \texttt{Expr(}exp\texttt{)} \\
 & \mid & \texttt{Assign([Name(}var\texttt{)], }exp\texttt{)} \\
\mathcal{L}^{mon}_{\mathsf{Var}} & ::= & \texttt{Module(}stmt^*\texttt{)}
\end{array}
$$

Example (more in the book):

```
x = 42 + -10
print(x + 10)
```

⇒

```
tmp_0 = -10
x = 42 + tmp_0
tmp_1 = x + 10
print(tmp_1)
```

Exercise 2: Select Instructions Pass

Implement the `select_instructions` pass in `compiler.py`. This pass translates $\mathcal{L}^{mon}_{Var}$ to $x86_{Var}$, shown below. We recommend implementing an auxiliary function named `select_stmt` for the *stmt* non-terminal of $\mathcal{L}^{mon}_{Var}$.

$$
\begin{array}{rcl}
reg & ::= & \texttt{'rsp'} \mid \texttt{'rbp'} \mid \texttt{'rax'} \mid \texttt{'rbx'} \mid \texttt{'rcx'} \mid \texttt{'rdx'} \mid \texttt{'rsi'} \mid \texttt{'rdi'} \mid \\
 & & \texttt{'r8'} \mid \texttt{'r9'} \mid \texttt{'r10'} \mid \texttt{'r11'} \mid \texttt{'r12'} \mid \texttt{'r13'} \mid \texttt{'r14'} \mid \texttt{'r15'} \\
arg & ::= & \texttt{Immediate(}int\texttt{)} \mid \texttt{Reg(}reg\texttt{)} \mid \texttt{Deref(}reg\texttt{,}int\texttt{)} \mid \texttt{Variable(}id\texttt{)} \\
instr & ::= & \texttt{Instr('addq',[}arg\texttt{,}arg\texttt{])} \mid \texttt{Instr('subq',[}arg\texttt{,}arg\texttt{])} \\
 & \mid & \texttt{Instr('movq',[}arg\texttt{,}arg\texttt{])} \mid \texttt{Instr('negq',[}arg\texttt{])} \\
 & \mid & \texttt{Instr('pushq',[}arg\texttt{])} \mid \texttt{Instr('popq',[}arg\texttt{])} \\
 & \mid & \texttt{Callq(}label\texttt{,}int\texttt{)} \mid \texttt{Retq()} \mid \texttt{Jump(}label\texttt{)} \\
x86_{Var} & ::= & \texttt{X86Program(}instr^*\texttt{)}
\end{array}
$$

Please use the `label_name` function provided in `utils.py` for generating labels for `Callq(`*label*`,`*int*`)`. An example translation (more in the book) where arg_1 and arg_2 are the translations of atm_1 and atm_2 respectively:

var = atm_1 + atm_2 ⇒ `movq` arg_1, *var*
`addq` arg_2, *var*

Function calls to `input_int` and `print` can be translated as follows (functions `read_int` and `print_int` are provided through the x86 interpreter):

var = `input_int();` ⇒ `callq read_int`
`movq %rax,` *var*

`print(`*atm*`)`	⇒	`movq` *arg*`, %rdi` `callq print_int`

Exercise 3: Assign Homes Pass

Implement the `assign_homes` pass in `compiler.py`, defining auxiliary functions for each of the non-terminals in the $x86_{Var}$ grammar. We recommend that the auxiliary functions take an extra parameter that maps variable names to homes (stack locations for now). This pass translates $x86_{Var}$ to $x86_{Int}$ shown below, i.e. removes program variables.

reg	::=	`'rsp'` | `'rbp'` | `'rax'` | `'rbx'` | `'rcx'` | `'rdx'` | `'rsi'` | `'rdi'` | `'r8'` | `'r9'` | `'r10'` | `'r11'` | `'r12'` | `'r13'` | `'r14'` | `'r15'`
arg	::=	`Immediate(`*int*`)` | `Reg(`*reg*`)` | `Deref(`*reg*`,`*int*`)`
instr	::=	`Instr('addq',[`*arg*`,`*arg*`])` | `Instr('subq',[`*arg*`,`*arg*`])`
	|	`Instr('movq',[`*arg*`,`*arg*`])` | `Instr('negq',[`*arg*`])`
	|	`Instr('pushq',[`*arg*`])` | `Instr('popq',[`*arg*`])`
	|	`Callq(`*label*`,`*int*`)` | `Retq()` | `Jump(`*label*`)`
$\text{x86}_{\mathsf{Int}}$	::=	`X86Program(`*instr**`)`

An example:

```
movq $42, a
movq a, b
movq b, %rax
```

⇒

```
movq $42, -8(%rbp)
movq -8(%rbp), -16(%rbp)
movq -16(%rbp), %rax
```

In the process of assigning variables to stack locations, it is convenient for you to compute and store the size of the frame (in bytes) in the field `stack_space` of the `X86Program` node, which is needed later to generate the conclusion of the `main` procedure. The x86-64 standard requires the frame size to be a multiple of 16 bytes.

Exercise 4: Patch Instructions Pass

Implement the `patch_instructions` pass in `compiler.py`. The x86 ISA allows only one argument of an instruction to be a memory reference. In this pass, the output of the `assign_homes` pass is modified to satisfy this requirement. You may use the `rax` register for this purpose. An example:

```
movq -8(%rbp), -16(%rbp)
```

⇒

```
movq -8(%rbp), %rax
movq %rax, -16(%rbp)
```

Exercise 5: Prelude and Conclusion Pass

Implement the `prelude_and_conclusion` pass in `compiler.py`. The prelude has the following structure (x is the required stack space as calculated in the assign homes pass):

```
pushq  %rbp
movq   %rsp, %rbp
subq  x, %rsp
```

The conclusion has the following structure:

```
addq    x, %rsp
popq    %rbp
retq
```

For more information, refer to section 2.2 of the book.

Exercise 6: Further Testing (optional)

You should now be able to run `run_tests.py` and get feedback on your implementation. If you want to create more tests, you can create them in the `tests/var` directory (cf. stub documentation).

If you want to test your compiler in detail, you can set the variable `text_x86` in the function `compile_and_test` inside `utils.py` to `True`. You will then get feedback on each of your passes.

Windows/MAC users: Please copy the newest version of `utils.py` from this repository `https://github.com/Compiler-Construction-Uni-Freiburg/assignment-2-stub` into your stub.