Assignment 4: Compiler for $\mathcal{L}_{\mathsf{If}}$

(Deadline: 10.01.2022 09:00)

Exercise 1: Shrink Pass

Implement the pass `shrink` in `compiler.py` to remove `and` and `or` from the language by translating them to `if` expressions in $\mathcal{L}_{\mathsf{If}}$ (see end of this assignment sheet for concrete and abstract syntax):

$$e_1 \texttt{ and } e_2 \quad \Rightarrow \quad e_2 \texttt{ if } e_1 \texttt{ else False}$$
$$e_1 \texttt{ or } e_2 \quad \Rightarrow \quad \texttt{True if } e_1 \texttt{ else } e_2$$

Exercise 2: Remove Complex Operands Pass

Add cases for Boolean constants, comparisons, `if` expressions and statements to `rco_exp` and `rco_stmt` functions in `compiler.py`. Regarding `if`, it is particularly important to **not** replace its condition with a temporary variable because that would interfere with the generation of high-quality output in the `explicate_control` pass.

We add a new language form, the `Let` expression (its abstract syntax can be found in `utils.py`), to aid in the translation of `if` expressions. When we recursively process the two branches of the `if`, we generate temporary variables and their initializing expressions (see example below). However, these expressions may contain side effects and should only be executed when the condition of the `if` is true (for the "then" branch) or false (for the "else" branch). The `Let` provides a way to initialize the temporary variables within the two branches of the `if` expression. In general, the `Let(`x`,`e_1`,`e_2`)` form assigns the result of e_1 to the variable x, and then evaluates e_2, which may reference x.

Example (of what we want to avoid):

```
(input_int() + 1) \
  if x > 0 else (42 + input_int())
```

$\Rightarrow$

```
tmp0 = input_int()
tmp1 = input_int()
(tmp0 + 1) \
  if x > 0 else (42 + tmp1)
```

Exercise 3: Explicate Control Pass

Implement the `explicate_control` pass (translating $\mathcal{L}_{\mathsf{if}}^{mon}$ to $\mathcal{C}_{\mathsf{If}}$, grammar found at the end of this assignment) using the following four auxiliary functions (as found in your assignment stub):

`explicate_effect` generates code for expressions as statements, so their result is ignored and only their side effects matter. It has three parameters: 1) the expression to be compiled, 2) the already-compiled code for this expression's *continuation*, that is, the list of statements that should execute after this expression, and 3) the dictionary of generated basic blocks. The `explicate_effect` function returns a list of $\mathcal{C}_{\mathsf{If}}$ statements and it may add to the dictionary of basic blocks. If the expression to be compiled is an `if` expression, we translate the two branches using `explicate_effect` and then translate the condition expression using `explicate_pred`, which generates code for the entire `if`.

`explicate_assign` generates code for expressions on the right-hand side of an assignment. It has four parameters: 1) the right-hand-side of the assignment, 2) the left-hand-side of the assignment (the variable), 3) the continuation, and 4) the dictionary of basic blocks. The `explicate_assign` function returns a list of $\mathcal{C}_{\mathsf{If}}$ statements and it may add to the dictionary of basic blocks.

`explicate_pred` generates code for an `if` expression or statement by analyzing the condition expression. It has four parameters: 1) the condition expression, 2) the generated statements for the "then" branch, 3) the generated statements for the "else" branch, and 4) the dictionary of basic blocks. The `explicate_pred` function returns a list of $\mathcal{C}_{\mathsf{If}}$ statements and it may add to the dictionary of basic blocks.

`explicate_stmt` generates code for statements. It has three parameters: 1) the statement to be compiled, 2) the code for its continuation, and 3) the dictionary of basic blocks. The `explicate_stmt` returns a list of statements and it may add to the dictionary of basic blocks.

Exercise 4: Select Instructions

Adapt the `select_instructions` pass (now translating $\mathcal{C}_{\mathsf{If}}$to $\text{x86}^{\mathsf{Var}}_{\mathsf{If}}$, see end of this assignment for grammar; note the new arg terminal `ByteReg`). Some useful examples:

`True` $\Rightarrow$ `1`   `False` $\Rightarrow$ `0`

var `= not` *var* $\Rightarrow$ `xorq $1,` *var*

var `= not` *atm* $\Rightarrow$ `movq` *arg*`,` *var*
`xorq $1,` *var*

var `= (`atm_1 `==` atm_2`)` $\Rightarrow$
`cmpq` arg_2`,` arg_1
`sete %al`
`movzbq %al,` *var*

`goto` ℓ $\Rightarrow$ `jmp` ℓ

```
if atm₁ == atm₂:
    goto ℓ₁
else:
    goto ℓ₂
```
$\Rightarrow$
```
cmpq arg₂, arg₁
je ℓ₁
jmp ℓ₂
```

Regarding the `return` statement, treat it as an assignment to the `rax` register followed by a jump a new block labelled `conclusion`, which you will fill later in the Prelude and Conclusion pass.

Exercise 5: Register Allocation

Register allocation for $\text{x86}^{\mathsf{Var}}_{\mathsf{If}}$requires you to perform liveness analysis on blocks separately. To perform liveness analysis on a block, we must know the live-after set for the last instruction in the block. If there are no successors (no jumps to other blocks), the live-after set is empty. If there are successors, their live-before set must be calculated first.

5.1: Control Flow Graph

Implement the function `cfg` in `register_allocation.py` to generate a control flow graph for a given dictionary of blocks. The control flow graph should be a directed graph where there is an edge from v to v' if there is a jump in block v to block v'.

In your stub, the result of this function is then transposed and topologically sorted to generate an order for performing liveness analysis for blocks.

5.2: Arg/Read/Write Locations

Extend your auxiliary functions for the new type of instructions found in $\text{x86}_{\mathsf{If}}^{\mathsf{Var}}$. You can omit `Jump` and `JumpIf`, they will be handled in 5.3.

5.3: Liveness Analysis

Update your `uncover_live` function: Iterate through the blocks in the order calculated in 5.1. Save the live-before set of a block in the `live_before_block` dictionary.

Regarding `Jump` and `JumpIf` instructions: The locations that are live before a `Jump` should be the locations in $L_{\texttt{before}}$ at the target of the jump. Liveness analysis for `JumpIf` is particularly interesting because, during compilation, we do not know which way a conditional jump will go. So we do not know whether to use the live-before set for the following instruction or the live-before set for the block associated with the *label*. However, there is no harm to the correctness of the generated code if we classify more locations as live than the ones that are truly live during one particular execution of the instruction. Thus, we can take the union of the live-before sets from the following instruction and from the mapping for *label* in `live_before_block`.

5.4: Interference Graph

Adapt `build_interference` to the new language.

Exercise 6: Patch Instructions & Prelude and Conclusion

6.1: Patch Instructions

The new instructions `cmpq` and `movzbq` have some special restrictions that need to be handled in the `patch_instructions` pass. The second argument of the `cmpq` instruction must not be an immediate value (such as an integer). So if you are comparing two immediates, we recommend inserting a `movq` instruction to put the second argument in `rax`. As usual, `cmpq` may have at most one memory reference. The second argument of the `movzbq` must be a register.

6.2: Prelude and Conclusion

The generation of the `main` function with its prelude and conclusion must change to accomodate how the program now consists of one or more basic blocks. After the prelude in `main`, jump to the `start` block. Place the conclusion in a basic block labelled with `conclusion`.

Concrete and Abstract Syntax of $\mathcal{L}_{\mathsf{If}}$, $\mathcal{C}_{\mathsf{If}}$, x86$_{\mathsf{If}}$

$$
\begin{array}{rcl}
\mathit{exp} &::=& \mathit{int} \mid \texttt{input\_int()} \mid \texttt{-}\,\mathit{exp} \mid \mathit{exp}\,\texttt{+}\,\mathit{exp} \mid \mathit{exp}\,\texttt{-}\,\mathit{exp} \mid \texttt{(}\mathit{exp}\texttt{)} \\
\mathit{stmt} &::=& \texttt{print(}\mathit{exp}\texttt{)} \mid \mathit{exp} \\ \hline
\mathit{exp} &::=& \mathit{var} \\
\mathit{stmt} &::=& \mathit{var}\,\texttt{=}\,\mathit{exp} \\ \hline
\mathit{cmp} &::=& \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{exp} &::=& \texttt{True} \mid \texttt{False} \mid \mathit{exp}\;\texttt{and}\;\mathit{exp} \mid \mathit{exp}\;\texttt{or}\;\mathit{exp} \mid \texttt{not}\;\mathit{exp} \\
&\mid& \mathit{exp}\;\mathit{cmp}\;\mathit{exp} \mid \mathit{exp}\;\texttt{if}\;\mathit{exp}\;\texttt{else}\;\mathit{exp} \\
\mathit{stmt} &::=& \texttt{if}\;\mathit{exp}\texttt{:}\;\mathit{stmt}^{+}\;\texttt{else:}\;\mathit{stmt}^{+} \\
\mathcal{L}_{\mathsf{If}} &::=& \mathit{stmt}^{*}
\end{array}
$$

Figure 1: The concrete syntax of $\mathcal{L}_{\mathsf{If}}$.

$$
\begin{array}{rcl}
\mathit{binaryop} &::=& \texttt{Add()} \mid \texttt{Sub()} \\
\mathit{unaryop} &::=& \texttt{USub()} \\
\mathit{exp} &::=& \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Call(Name('input\_int'),[])} \\
&\mid& \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{exp}\texttt{)} \mid \texttt{BinOp(}\mathit{exp}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{exp}\texttt{)} \\
\mathit{stmt} &::=& \texttt{Expr(Call(Name('print'),[}\mathit{exp}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\ \hline
\mathit{exp} &::=& \texttt{Name(}\mathit{var}\texttt{)} \\
\mathit{stmt} &::=& \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\ \hline
\mathit{boolop} &::=& \texttt{And()} \mid \texttt{Or()} \\
\mathit{unaryop} &::=& \texttt{Not()} \\
\mathit{cmp} &::=& \texttt{Eq()} \mid \texttt{NotEq()} \mid \texttt{Lt()} \mid \texttt{LtE()} \mid \texttt{Gt()} \mid \texttt{GtE()} \\
\mathit{bool} &::=& \texttt{True} \mid \texttt{False} \\
\mathit{exp} &::=& \texttt{Constant(}\mathit{bool}\texttt{)} \mid \texttt{BoolOp(}\mathit{boolop}\texttt{,[}\mathit{exp}\texttt{,}\mathit{exp}\texttt{])} \\
&\mid& \texttt{Compare(}\mathit{exp}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{exp}\texttt{])} \mid \texttt{IfExp(}\mathit{exp}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
\mathit{stmt} &::=& \texttt{If(}\mathit{exp}\texttt{, }\mathit{stmt}^{+}\texttt{, }\mathit{stmt}^{+}\texttt{)} \\
\mathcal{L}_{\mathsf{If}} &::=& \texttt{Module(}\mathit{stmt}^{*}\texttt{)}
\end{array}
$$

Figure 2: The abstract syntax of $\mathcal{L}_{\mathsf{If}}$.

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \mathit{int} \mid \mathit{var} \mid \mathit{bool} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{input\_int()} \mid \mathit{atm}\ \mathit{binaryop}\ \mathit{atm} \mid \mathit{unaryop}\ \mathit{atm} \\
 & \mid & \mathit{atm}\ \mathit{cmp}\ \mathit{atm} \\
\mathit{stmt} & ::= & \texttt{print(}\mathit{exp}\texttt{)} \mid \mathit{exp} \\
 & \mid & \mathit{var}\ \texttt{=}\ \mathit{exp} \mid \texttt{return}\ \mathit{exp} \mid \texttt{goto}\ \mathit{label} \\
 & \mid & \texttt{if}\ \mathit{atm}\ \mathit{cmp}\ \mathit{atm}\texttt{:}\ \texttt{goto}\ \mathit{label}\ \texttt{else:}\ \texttt{goto}\ \mathit{label} \\
\mathcal{C}_{\mathsf{If}} & ::= & (\mathit{label}\texttt{:}\ \mathit{stmt}^*)\ldots
\end{array}
$$

Figure 3: The concrete syntax of $\mathcal{C}_{\mathsf{If}}$.

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \mid \texttt{Constant(}\mathit{bool}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{BinOp(}\mathit{atm}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{atm}\texttt{)} \mid \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{atm}\texttt{)} \\
 & \mid & \texttt{Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{])} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{exp}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \mid \texttt{Return(}\mathit{exp}\texttt{)} \mid \texttt{Goto(}\mathit{label}\texttt{)} \\
 & \mid & \texttt{If(Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{]), [Goto(}\mathit{label}\texttt{)], [Goto(}\mathit{label}\texttt{)])} \\
\mathcal{C}_{\mathsf{If}} & ::= & \texttt{CProgram(}\{\mathit{label}\texttt{:}\ \mathit{stmt}^*, \ldots\}\texttt{)}
\end{array}
$$

Figure 4: The abstract syntax of $\mathcal{C}_{\mathsf{If}}$.

$$
\begin{array}{lcl}
\mathit{bytereg} & ::= & \texttt{ah} \mid \texttt{al} \mid \texttt{bh} \mid \texttt{bl} \mid \texttt{ch} \mid \texttt{cl} \mid \texttt{dh} \mid \texttt{dl} \\
\mathit{arg} & ::= & \$\mathit{int} \mid \%\mathit{reg} \mid \mathit{int}(\%\mathit{reg}) \mid \%\mathit{bytereg} \\
\mathit{cc} & ::= & \texttt{e} \mid \texttt{ne} \mid \texttt{l} \mid \texttt{le} \mid \texttt{g} \mid \texttt{ge} \\
\mathit{instr} & ::= & \texttt{addq}\ \mathit{arg}\texttt{,}\mathit{arg} \mid \texttt{subq}\ \mathit{arg}\texttt{,}\mathit{arg} \mid \texttt{negq}\ \mathit{arg} \mid \texttt{movq}\ \mathit{arg}\texttt{,}\mathit{arg} \mid \\
 & & \texttt{callq}\ \mathit{label} \mid \texttt{pushq}\ \mathit{arg} \mid \texttt{popq}\ \mathit{arg} \mid \texttt{retq} \mid \texttt{jmp}\ \mathit{label} \mid \\
 & & \mathit{label}\texttt{:}\ \mathit{instr} \mid \texttt{xorq}\ \mathit{arg}\texttt{, }\mathit{arg} \mid \texttt{cmpq}\ \mathit{arg}\texttt{, }\mathit{arg} \mid \\
 & & \texttt{set}\mathit{cc}\ \mathit{arg} \mid \texttt{movzbq}\ \mathit{arg}\texttt{, }\mathit{arg} \mid \texttt{j}\mathit{cc}\ \mathit{label} \\
\mathrm{x86}_{\mathsf{If}} & ::= & \texttt{.globl main} \\
 & & \texttt{main:}\ \mathit{instr}\ldots
\end{array}
$$

Figure 5: The concrete syntax of $\mathrm{x86}_{\mathsf{If}}$.

$$
\begin{array}{lcl}
\mathit{bytereg} & ::= & \texttt{'ah'} \mid \texttt{'al'} \mid \texttt{'bh'} \mid \texttt{'bl'} \mid \texttt{'ch'} \mid \texttt{'cl'} \mid \texttt{'dh'} \mid \texttt{'dl'} \\
\mathit{arg} & ::= & \texttt{Immediate(}\mathit{int}\texttt{)} \mid \texttt{Reg(}\mathit{reg}\texttt{)} \mid \texttt{Deref(}\mathit{reg},\mathit{int}\texttt{)} \mid \texttt{ByteReg(}\mathit{bytereg}\texttt{)} \\
\mathit{cc} & ::= & \texttt{'e'} \mid \texttt{'ne'} \mid \texttt{'l'} \mid \texttt{'le'} \mid \texttt{'g'} \mid \texttt{'ge'} \\
\mathit{instr} & ::= & \texttt{Instr('addq',[}\mathit{arg},\mathit{arg}\texttt{])} \mid \texttt{Instr('subq',[}\mathit{arg},\mathit{arg}\texttt{])} \\
& \mid & \texttt{Instr('movq',[}\mathit{arg},\mathit{arg}\texttt{])} \mid \texttt{Instr('negq',[}\mathit{arg}\texttt{])} \\
& \mid & \texttt{Callq(}\mathit{label},\mathit{int}\texttt{)} \mid \texttt{Retq()} \mid \texttt{Instr('pushq',[}\mathit{arg}\texttt{])} \\
& \mid & \texttt{Instr('popq',[}\mathit{arg}\texttt{])} \mid \texttt{Jump(}\mathit{label}\texttt{)} \\
& \mid & \texttt{Instr('xorq',[}\mathit{arg},\mathit{arg}\texttt{])} \mid \texttt{Instr('cmpq',[}\mathit{arg},\mathit{arg}\texttt{])} \\
& \mid & \texttt{Instr('set'} + \mathit{cc}\texttt{,[}\mathit{arg}\texttt{])} \mid \texttt{Instr('movzbq',[}\mathit{arg},\mathit{arg}\texttt{])} \\
& \mid & \texttt{JumpIf(}\mathit{cc},\mathit{label}\texttt{)} \\
\mathrm{x86}_{\mathsf{If}} & ::= & \texttt{X86Program(}\{\mathit{label} : \mathit{instr}^*, \ldots\}\texttt{)}
\end{array}
$$

Figure 6: The abstract syntax of $\mathrm{x86}_{\mathsf{If}}$.