Assignment 6: Compiler for $\mathcal{L}_{\mathsf{Fun}}$

(Deadline: 07.02.2022 09:00)

Exercise 0: General Note

This language adds function definitions. After the shrink pass, module bodies should consist only of function definitions. Hence, you must adjust passes to handle function definitions instead of list of statements or blocks.

Exercise 1: Shrink Pass

Update the shrink pass by handling the new `exp` and `stmt` cases as well as introducing an explicit `main` function that gobbles up all the top-level statement:

`Module(`*def* ... *stmt* ...`)`
$\Rightarrow$ `Module(`*def* ... *mainDef*`)`

Where *mainDef* is:

`FunctionDef('main', [], int, None,` *stmt* ...`Return(Constant(0)), None)`

Exercise 2: Reveal Functions Pass

Implement the new reveal functions pass, translating variables that refer to functions `Name`(f) to the new non-terminal `FunRef`(f, n), where n is the arity of f. Function arities have to be gathered somehow beforehand.

Exercise 3: Limit Functions Pass

Implement the new limit functions pass, limiting functions to 6 parameters. The translation of function definitions is as follows:

`FunctionDef(`f`, [(`x_1`,`T_1`),`$\ldots$`,(`x_n`,`T_n`)],` T_r`, None,` *body*`, None)`
$\Rightarrow$
`FunctionDef(`f`, [(`x_1`,`T_1`),`$\ldots$`,(`x_5`,`T_5`),(tup,TupleType([`$T_6,\ldots,T_n$`]))],`
T_r`, None,` *body*$'$`, None)`

where the *body* is transformed into *body*$'$ by replacing the occurrences of each parameter x_i where $i > 5$ with the kth element of the tuple, where $k = i - 6$:

`Name(`x_i`)` $\Rightarrow$ `Subscript(tup, Constant(`k`), Load())`

For function calls with too many arguments, the `limit_functions` pass transforms them in the following way.

`Call(`e_0`, [`$e_1,\ldots,e_n$`])`
$\Rightarrow$
`Call(`e_0`, [`$e_1,\ldots,e_5$`,Tuple([`$e_6,\ldots,e_n$`])])`

Exercise 4: Expose Allocation Pass

Update the expose allocation pass by handling the new `exp` and `stmt` cases.

Hint: There shouldn't be many.

Exercise 5: Remove Comples Operands Pass

Update the remove complex operands pass. `FunRef` and `Call` are to be treated as complex expressions. Arguments to `Call` must be atomic. Note that we must inspect the expression occuring in `Return` statements in the next pass, so it must not be atomic.

Exercise 6: Explicate Control Pass

Update the explicate control pass by treating the new cases where necessary.

We also need a new auxiliary function: `explicate_tail`. This function handles expressions occuring in `Return` statements (in tail context). A skeleton for this function is provided in your stub. The default case should produce a `Return` statement. The case for `Call` should change it into `TailCall`. The other cases should recursively process their subexpressions and statements, choosing the appropriate explicate functions for the various contexts.

Exercise 7: Select Instructions Pass

Update the select instructions pass. This pass requires some amount of work but should be straight-forward to implement. Refer to the book chapter 7.8 for translations.

Note: A preliminary prologue and conclusion for functions (including the main function) have been added to make the x86 interpreter work.

Exercise 8: Register Allocation Pass

The following changes have to be made:

- Add cases for `IndirectCallq` and `TailJump` in `get_read_write_locations`: The `IndirectCallq` instruction should be treated like `Callq` regarding its written locations W, in that they should include all the caller-saved registers. Recall that the reason for that is to force variables that are live across a function call to be assigned to callee-saved registers or to be spilled to the stack.

  Regarding the set of read locations R, the arity field of `TailJump` and `IndirectCallq` determines how many of the argument-passing registers should be considered as read by those instructions. Also, the target field of `TailJump` and `IndirectCallq` should be included in the set of read locations R.

- Change `build_interference`: Recall that for $\mathcal{L}_{\mathsf{Tup}}$ we discussed the need to spill vector-typed variables that are live during a call to `collect`, the garbage collector. With the addition of functions to our language, we need to revisit this issue. Functions that perform allocation contain calls to the collector. Thus, we should not only spill a vector-typed variable when it is live during a call to `collect`, but we should spill the variable if it is live during call to a user-defined function. Thus, in the `build_interference` pass, add interference edges between call-live vector-typed variables and the callee-saved registers (in addition to the usual addition of edges between call-live variables and the caller-saved registers).

- Change `assign_homes`: Registers should be allocated for each function separately. Some cases in `assign_homes_instr` have to be added.

Exercise 9: Patch Instructions Pass

Update the `patch_instructions` pass. The destination argument of `leaq` must be a register. Additionally, you should ensure that the argument of `TailJump` is *rax*, our reserved register—mostly to make code generation more convenient, because we trample many registers before the tail call.

Exercise 10: Prelude & Conclusion Pass

- Generate a prelude & conclusion for each function definition. For the prelude:
  1. Push `rbp` to the stack and set `rbp` to current stack pointer.
  2. Push to the stack all of the callee-saved registers that were used for register allocation.
  3. Move the stack pointer `rsp` down by the size of the stack frame for this function, which depends on the number of regular spills. (Aligned to 16 bytes.)
  4. Move the root stack pointer `r15` up by the size of the root-stack frame for this function, which depends on the number of spilled vectors.
  5. Initialize to zero all new entries in the root-stack frame.
  6. Jump to the start block.

  The prelude of the `main` function has one additional task: call the `initialize` function to set up the garbage collector and move the value of the global `rootstack_begin` in `r15`. This initialization should happen before step 4 above, which depends on `r15`.

  The conclusion of every function should do the following.
  1. Move the stack pointer back up by the size of the stack frame for this function.
  2. Restore the callee-saved registers by popping them from the stack.
  3. Move the root stack pointer back down by the size of the root-stack frame for this function.
  4. Restore `rbp` by popping it from the stack.
  5. Return to the caller with the `retq` instruction.
- Translate `TailJump`: A straightforward translation of `TailJump` would simply be `jmp *`*arg*. However, before the jump we need to pop the current frame. This sequence of instructions is the same as the code for the conclusion of a function, except the `retq` is replaced with `jmp *`*arg*.

Abstract Syntaxes

binaryop	::=	`Add()` | `Sub()`
unaryop	::=	`USub()`
exp	::=	`Constant(`*int*`)` | `Call(Name('input_int'),[])`
	|	`UnaryOp(`*unaryop*`,`*exp*`)` | `BinOp(`*exp*`,`*binaryop*`,`*exp*`)`
stmt	::=	`Expr(Call(Name('print'),[`*exp*`]))` | `Expr(`*exp*`)`
exp	::=	`Name(`*var*`)`
stmt	::=	`Assign([Name(`*var*`)], `*exp*`)`
boolop	::=	`And()` | `Or()`
unaryop	::=	`Not()`
cmp	::=	`Eq()` | `NotEq()` | `Lt()` | `LtE()` | `Gt()` | `GtE()`
bool	::=	`True` | `False`
exp	::=	`Constant(`*bool*`)` | `BoolOp(`*boolop*`,[`*exp*`,`*exp*`])`
	|	`Compare(`*exp*`,[`*cmp*`],[`*exp*`])` | `IfExp(`*exp*`,`*exp*`,`*exp*`)`
stmt	::=	`If(`*exp*`, `$stmt^+$`, `$stmt^+$`)`
exp	::=	`Tuple(`exp^+`,Load())` | `Subscript(`*exp*`,Constant(`*int*`),Load())`
	|	`Call(Name('len'),[`*exp*`])` | `Begin(`$stmt^*$`, `*exp*`)`
type	::=	`IntType()` | `BoolType()` | `VoidType()` | `TupleType(`$type^+$`)`
	|	`FunctionType(`$type^*$`, `*type*`)`
exp	::=	`Call(`*exp*`, `exp^*`)`
stmt	::=	`Return(`*exp*`)`
params	::=	$($*var*`,`*type*$)^*$
def	::=	`FunctionDef(`*var*`, `*params*`, `*type*`, `$stmt^+$`)`
body	::=	*def* | *stmt*
$\mathcal{L}_{\mathsf{Fun}}$	::=	`Module(`$body^*$`)`

Figure 1: Abstract Syntax of $\mathcal{L}_{\mathsf{Fun}}$

$$
\begin{array}{lcl}
atm & ::= & \texttt{Constant(}int\texttt{)} \mid \texttt{Name(}var\texttt{)} \mid \texttt{Constant(}bool\texttt{)} \\
exp & ::= & atm \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{BinOp(}atm\texttt{,}binaryop\texttt{,}atm\texttt{)} \mid \texttt{UnaryOp(}unaryop\texttt{,}atm\texttt{)} \\
 & \mid & \texttt{Compare(}atm\texttt{,[}cmp\texttt{],[}atm\texttt{])} \\
stmt & ::= & \texttt{Expr(Call(Name('print'),[}atm\texttt{]))} \mid \texttt{Expr(}exp\texttt{)} \\
 & \mid & \texttt{Assign([Name(}var\texttt{)], }exp\texttt{)} \mid \texttt{Return(}exp\texttt{)} \mid \texttt{Goto(}label\texttt{)} \\
 & \mid & \texttt{If(Compare(}atm\texttt{,[}cmp\texttt{],[}atm\texttt{]), [Goto(}label\texttt{)], [Goto(}label\texttt{)])} \\
\hline
exp & ::= & \texttt{Subscript(}atm\texttt{,}atm\texttt{,Load())} \mid \texttt{Allocate(}int, type\texttt{)} \\
 & \mid & \texttt{GlobalValue(}var\texttt{)} \mid \texttt{Call(Name('len'),[}atm\texttt{])} \\
stmt & ::= & \texttt{Collect(}int\texttt{)} \\
 & \mid & \texttt{Assign([Subscript(}atm\texttt{,}atm\texttt{,Store())], }atm\texttt{)} \\
\hline
exp & ::= & \texttt{FunRef(}label\texttt{, }int\texttt{)} \mid \texttt{Call(}atm\texttt{, }atm^*\texttt{)} \\
stmt & ::= & \texttt{TailCall(}atm\texttt{,}atm^*\texttt{)} \\
params & ::= & \texttt{(}var\texttt{,}type\texttt{)}^* \\
block & ::= & label\texttt{:}stmt^* \\
blocks & ::= & \{block, \ldots\} \\
def & ::= & \texttt{FunctionDef(}label\texttt{, }params\texttt{, }blocks\texttt{, None, }type\texttt{, None)} \\
\mathcal{C}_{\mathsf{Fun}} & ::= & \texttt{CProgramDefs(}def^*\texttt{)}
\end{array}
$$

Figure 2: Abstract Syntax of $\mathcal{C}_{\mathsf{Fun}}$

$$
\begin{array}{lcl}
arg & ::= & \texttt{Constant(}int\texttt{)} \mid \texttt{Reg(}reg\texttt{)} \mid \texttt{Deref(}reg\texttt{,}int\texttt{)} \mid \texttt{ByteReg(}reg\texttt{)} \\
 & \mid & \texttt{Global(}var\texttt{)} \mid \texttt{FunRef(}label\texttt{, }int\texttt{)} \\
instr & ::= & \ldots \mid \texttt{IndirectCallq(}arg\texttt{, }int\texttt{)} \mid \texttt{TailJump(}arg\texttt{, }int\texttt{)} \\
 & \mid & \texttt{Instr('leaq',[}arg\texttt{,Reg(}reg\texttt{)])} \\
block & ::= & label\texttt{: }instr^* \\
blocks & ::= & \{block, \ldots\} \\
def & ::= & \texttt{FunctionDef(}label\texttt{, [], }blocks\texttt{, \_, }type\texttt{, \_)} \\
\mathrm{x86}_{\mathsf{callq*}} & ::= & \texttt{X86ProgramDefs(}def^*\texttt{)}
\end{array}
$$

Figure 3: Abstract Syntax of $\mathrm{x86}_{\mathsf{callq*}}$