Assignment 5: Compiler for $\mathcal{L}_{\mathsf{Tup}}$

(Deadline: 24.01.2022 09:00)

Exercise 0: Reading

Reread chapters 6.1 and 6.2 in the book to get an idea of what we're trying to implement in the following exercises and to understand the garbage collection functionality of the provided runtime.

Exercise 1: Expose Allocation Pass

Reread chapter 6.3 to get an understanding of what the expose allocation pass does. Implement this pass. See abstract syntax for $\mathcal{L}_{\mathsf{Tup}}^{exposed}$ (end of the assignment sheet) for details.

Note: The implementation of `begin` *is provided in the stub.*

Exercise 2: Remove Complex Operands Pass

Update the remove complex operands pass. The expressions `allocate`, `global_value`, `begin`, and tuple access should be treated as complex operands. The sub-expressions of tuple access must be atomic. See abstract syntax for $\mathcal{L}_{\mathsf{Alloc}}^{mon}$ (end of the assignment sheet) for details.

Exercise 3: Explicate Control Pass

Update the explicate control pass. The output of `explicate_control` is a program in the intermediate language $\mathcal{C}_{\mathsf{Tup}}$. The new expressions of $\mathcal{C}_{\mathsf{Tup}}$ include `allocate`, accessing tuple elements, and `global_value`. $\mathcal{C}_{\mathsf{Tup}}$ also includes the `collect` statement and assignment to a tuple element. The `explicate_control` pass can treat these new forms much like the other forms that we've already encoutered. See abstract syntax for $\mathcal{C}_{\mathsf{Tup}}$ (end of the assignment sheet) for details.

Exercise 4: Select Instructions Pass

Reread chapter 6.6 to get an understanding of what changes have to be made to the select instructions pass. Update the select instructions pass. Note which registers may not be used for register allocation, as this information is needed for the next pass. See abstract syntax for $\text{x86}_{\mathsf{Global}}$ for details (the only difference is the new terminal `Global(`*var*`)`.)

Note: The select instructions pass adds a preliminary prelude as to allow running `interp_x86` *on the pass's intermediate output. This has to be overwritten in the prelude and conclusion pass.*

Exercise 5: Register Allocation Pass

Reread chapter 6.7 to get an understanding of what changes have to be made in the register allocation pass. Update the following functions:

1. `assign_homes`; to spill tuple-typed variables to the root stack
2. `build_interference`; to make sure tuple-typed variables that are live at a call to the collector are spilled
3. `newcolor`; to make sure any registers reserved for garbage collection are not used for other purposes

The type checker for $\mathcal{C}_{\mathsf{Tup}}$ returns a field name `var_types` in the `CProgram` AST node which may be useful. The number of spills to the root stack are needed for the following pass.

Exercise 6: Prelude And Conclusion Pass

Update the prelude and conclusion pass. In the prelude:

1. Initialize the garbage collection: Call the function `initialize` after moving its arguments, root stack size and heap size, into registers `rdi` and `rsi` respectively. The book recommends 16384 for both.
2. Move the global variable `rootstack_begin` into our root stack pointer register (`r15`).
3. Zero-out all (future) locations on the root stack.
4. Allocate required space on the root stack in the prelude by bumping the root stack pointer (`r15`). The root stack grows up instead of down.

In the conclusion, deallocate the required root stack space.

Abstract Syntaxes

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

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

$$
\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{)} \\
\hline
\mathit{exp} & ::= & \texttt{Subscript(}\mathit{exp}\texttt{,Constant(}\mathit{int}\texttt{),Load())} \\
 & \mid & \texttt{Call(Name('len'),[}\mathit{exp}\texttt{])} \mid \texttt{Allocate(}\mathit{int}\texttt{, }\mathit{type}\texttt{)} \\
 & \mid & \texttt{GlobalValue(}\mathit{var}\texttt{)} \mid \texttt{Begin(}\mathit{stmt}^{*}\texttt{, }\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Assign([Subscript(}\mathit{exp}\texttt{,}\mathit{exp}\texttt{,Store())], }\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Collect(}\mathit{int}\texttt{)} \\
\mathcal{L}_{\mathsf{Tup}} & ::= & \texttt{Module(}\mathit{stmt}^{*}\texttt{)}
\end{array}
$$

Figure 2: Abstract Syntax of $\mathcal{L}_{\mathsf{Tup}}^{exposed}$

$$
\begin{array}{rcl}
atm & ::= & \texttt{Constant}(int) \mid \texttt{Name}(var) \\
exp & ::= & atm \mid \texttt{Call}(\texttt{Name}(\texttt{'input\_int'}), []) \\
 & \mid & \texttt{UnaryOp}(unaryop, atm) \mid \texttt{BinOp}(atm, binaryop, atm) \\
stmt & ::= & \texttt{Expr}(\texttt{Call}(\texttt{Name}(\texttt{'print'}), [atm])) \mid \texttt{Expr}(exp) \\
 & \mid & \texttt{Assign}([\texttt{Name}(var)],\ exp) \\
\hline
atm & ::= & \texttt{Constant}(bool) \\
exp & ::= & \texttt{Compare}(atm, [cmp], [atm]) \mid \texttt{IfExp}(exp, exp, exp) \\
 & \mid & \texttt{Let}(var, exp, exp) \\
stmt & ::= & \texttt{If}(exp,\ stmt^*,\ stmt^*) \\
\hline
exp & ::= & \texttt{Subscript}(atm, atm, \texttt{Load}()) \\
 & \mid & \texttt{Call}(\texttt{Name}(\texttt{'len'}), [atm]) \mid \texttt{Allocate}(int, type) \\
 & \mid & \texttt{GlobalValue}(var) \mid \texttt{Begin}(stmt^*,\ exp) \\
stmt & ::= & \texttt{Assign}([\texttt{Subscript}(atm, atm, \texttt{Store}())],\ atm) \\
 & \mid & \texttt{Collect}(int) \\
\mathcal{L}^{mon}_{\mathsf{Alloc}} & ::= & \texttt{Module}(stmt^*)
\end{array}
$$

Figure 3: Abstract Syntax of $\mathcal{L}^{mon}_{\mathsf{Alloc}}$

$$
\begin{array}{rcl}
atm & ::= & \texttt{Constant}(int) \mid \texttt{Name}(var) \mid \texttt{Constant}(bool) \\
exp & ::= & atm \mid \texttt{Call}(\texttt{Name}(\texttt{'input\_int'}), []) \\
 & \mid & \texttt{BinOp}(atm, binaryop, atm) \mid \texttt{UnaryOp}(unaryop, atm) \\
 & \mid & \texttt{Compare}(atm, [cmp], [atm]) \\
stmt & ::= & \texttt{Expr}(\texttt{Call}(\texttt{Name}(\texttt{'print'}), [atm])) \mid \texttt{Expr}(exp) \\
 & \mid & \texttt{Assign}([\texttt{Name}(var)],\ exp) \mid \texttt{Return}(exp) \mid \texttt{Goto}(label) \\
 & \mid & \texttt{If}(\texttt{Compare}(atm, [cmp], [atm]),\ [\texttt{Goto}(label)],\ [\texttt{Goto}(label)]) \\
\hline
exp & ::= & \texttt{Subscript}(atm, atm, \texttt{Load}()) \mid \texttt{Allocate}(int, type) \\
 & \mid & \texttt{GlobalValue}(var) \mid \texttt{Call}(\texttt{Name}(\texttt{'len'}), [atm]) \\
stmt & ::= & \texttt{Collect}(int) \\
 & \mid & \texttt{Assign}([\texttt{Subscript}(atm, atm, \texttt{Store}())],\ atm) \\
\mathcal{C}_{\mathsf{Tup}} & ::= & \texttt{CProgram}(\{label\colon stmt^*, \ldots\})
\end{array}
$$

Figure 4: Abstract Syntax of $\mathcal{C}_{\mathsf{Tup}}$

$$
\begin{array}{rcl}
arg & ::= & \texttt{Constant}(int) \mid \texttt{Reg}(reg) \mid \texttt{Deref}(reg, int) \mid \texttt{ByteReg}(reg) \\
 & \mid & \texttt{Global}(var) \\
\text{x86}_{\mathsf{Global}} & ::= & \texttt{X86Program}((\ldots(label\ .\ block)\ldots))
\end{array}
$$

Figure 5: Abstract Syntax of $\text{x86}_{\mathsf{Global}}$ (in part)