This exam has 12 questions, for a total of 100 points.

1. **5 points** What is the observable behavior of the following \(L_{\text{Fun}}\) program? (e.g. does it produce an error at compile time or runtime? does it produce an integer, which one? does it diverge?)

```python
def f(x : Integer) -> tuple[int]
    v = (x, x)
    return v

v1 = f(3)
v2 = f(3)
v3 = v1
if v1 is v2:
    if v1 is v3:
        print(0)
    else:
        print(1)
else:
    if v1 is v3:
        print(2)
    else:
        print(3)
```

**Solution:** This program outputs the number 2 because \(v1 \text{ is } v2\) is false (they are tuples of different identity) and \(v1 \text{ is } v3\) is true (they are aliases for the same tuple).

2. **5 points** What is the observable behavior of the following \(L_{\text{Tup}}\) program? (e.g. does it produce an error at compile time or runtime? does it produce an integer, which one? does it diverge?)

```python
v1 = (3,3)
v2 = (3,3)
if v1[0] == v2[2]:
    print(0)
else:
    print(1)
```

**Solution:** The program is not well-typed because the type of \(v2\) is \(tuple[int, int]\) but \(v2[2]\) tries to access its third element. So the type checker outputs an error.
3. **8 points** Given the following $\mathcal{L}_\text{While}$ program, apply the Explicate Control pass to translate it to $\mathcal{C}_\text{If}$.

```python
sum = 0
i = input_int()
while i > 0:
    sum = sum + i
    i = i - 1
tmp = 27 + sum
print(27 + sum)
```

**Solution: (2 points per basic block)**

```plaintext
start:
    sum = 0
    i = input_int()
    goto loop.35

loop.35:
    if i > 0:
        goto block.37
    else:
        goto block.36

block.37:
    sum = (sum + i)
    i = (i - 1)
    goto loop.35

block.36:
    tmp = 27 + sum
    print(tmp)
    return 0
```
4. 13 points Apply liveness analysis to the following pseudo-x86 program to determine the set of live locations before and after every instruction. (The callee and caller saved registers are listed in the Appendix of this exam.)

```
start:
callq read_int
movq %rax, x57
movq $1, y58
callq read_int
movq %rax, i59
jmp loop65

loop65:
movq i59, tmp60
cmpq $0, tmp60
jg block67
jmp block66

block66:

movq y58, tmp64
movq x57, %rax
addq tmp64, %rax
jmp conclusion

block67:

movq y58, tmp61
movq y58, tmp62
movq tmp61, y58
addq tmp62, y58
movq i59, tmp63
movq tmp63, i59
subq $1, i59
jmp loop65
```

Solution: (1/2 point per liveness set)
start:
  (set)
callq read_int
  (set)
movq %rax, x57
  { x57}
movq $1, y58
  { x57 y58}
callq read_int
  { x57 y58}
movq %rax, i59
  { x57 y58 i59}
jmp loop65
  { x57 y58 i59}

loop65:
  { x57 y58 i59}
movq i59, tmp60
  { x57 y58 i59 tmp60}
cmpq $0, tmp60
  { x57 y58 i59}
jg block67
  { x57 y58 i59}
jmp block66
  { x57 y58}

block66:
  { x57 y58}
movq y58, tmp64
  { tmp64 x57}
movq x57, %rax
  { tmp64}
addq tmp64, %rax
  {}}
jmp conclusion
  {}

block67:
  { x57 y58 i59}
movq y58, tmp61
  { tmp61 x57 y58 i59}
movq y58, tmp62
  { tmp61 tmp62 x57 i59}
movq tmp61, y58
  { tmp62 x57 y58 i59}
addq tmp62, y58
  { x57 y58 i59}
movq i59, tmp63
  { tmp63 x57 y58}
movq tmp63, i59
  { x57 y58 i59}
subq $1, i59
  { x57 y58 i59}
jmp loop65
  { x57 y58 i59}
5. **[10 points]** Fill in the blanks for the following `expose_alloc_tuple` auxiliary function of the Expose Allocation pass that translates from $\mathcal{L}_{\text{Tup}}$ to $\mathcal{L}_{\text{Alloc}}$. (The grammar for $\mathcal{L}_{\text{Alloc}}$ is in the Appendix.)

```python
def expose_alloc_tuple(es, tupleType, allocExp):
    n = len(es)
    num_bytes = (a)
    vec = generate_name('alloc')
    space_left = Compare((b), [Lt()],
                           [GlobalValue('fromspace_end')])
    xs = [Name(generate_name('init')) for e in es]
    inits = [Assign([x], e) for (x, e) in zip(xs, es)]
    initVec = []
    i = 0
    for x in xs:
        initVec += [(c)]
        i += 1
    return Begin(inits \
                 + [If(space_left, [], [(d)])] \
                 + [Assign([Name(vec)], allocExp)] \
                 + initVec,
                 Name(vec))
```

```python
def expose_alloc_exp(e: expr) -> expr:
    match e:
        case Tuple(es, Load()):
            alloc = Allocate(len(es), e.has_type)
            return expose_alloc_tuple(es, tupleType, alloc)
        ...
```

**Solution: (2 points each)**

(a) $(n + 1) \times 8$
(b) `BinOp(GlobalValue('free_ptr'), Add(), Constant(num_bytes))`
(c) `Assign([Subscript(Name(vec), Constant(i), Store())], x)`
(d) `Collect(num_bytes)`
(e) `[expose_alloc_exp(e) for e in es]`
6. **4 points** In the `expose_alloc_tuple` function of the previous question, why are the initializing expressions `es` assigned to the temporary variables `xs` instead of using them directly in the tuple initialization?

**Solution:** The reason is that between the allocation of the tuple and the initialization of its elements, we cannot allow a call to `collect` because then the garbage collector would try to traverse a partially-initialized tuple, causing it to potentially jump to random locations in memory. The initializing expressions may contain tuple-creation expressions and hence calls to `collect`. The `let` binding of the initializing expressions causes them to be executed first, prior to the allocation and initialization of the tuple.

7. **6 points** Describe the layout of the 64-bit tag at the beginning of every tuple.

**Solution:**

- Bit position 0 is set to 0 when the tuple has been copied into the TO space (in the process of doing a copy collection) and it is set to 1 otherwise. If it is set to 0, then the 64 bits are the address of the new location in the TO space. **(2 points)**

- Bit position 1 through 6 stores the length of the tuple. **(2 points)**

- Bit position 7 through 57 is the pointer mask. It says, for each element of the tuple, whether the element is a pointer (that is, a tuple) or something else (like an integer or Boolean). **(2 points)**
8. [10 points] Fill in the blanks for the following explicate_control that translates $L^\text{mon}_{\text{FunRef}}$ programs into $C_{\text{Fun}}$ programs.

```python
def explicate_tail(e : expr, blocks: Dict[str, List[stmt]]) -> List[stmt]:
    match e:
        case Call(Name(f), args) if f in builtin_functions:
            return [__(a)__]
        case Call(func, args):
            return [__(b)__]
        ...

def explicate_pred(cnd: expr, thn: List[stmt], els: List[stmt],
                   basic_blocks: Dict[str, List[stmt]]) -> List[stmt]:
    match cnd:
        case Call(func, args):
            tmp = generate_name('call')
            return [__(c__),
                    If(Compare(Name(tmp), [Eq()], [Constant(False)]),
                    create_block(els, basic_blocks),
                    create_block(thn, basic_blocks))]
        ...

def explicate_stmt(s: stmt, cont: List[stmt],
                   blocks: Dict[str, List[stmt]]) -> List[stmt]:
    match s:
        case Return(value):
            return [__(d)__]
        ...

def explicate_def(d) -> stmt:
    match d:
        case FunctionDef(name, params, body, _, returns, _):
            new_body = []
            blocks = {}
            if isinstance(returns, VoidType):
                body = body + [Return(Constant(None))]
            for s in reversed(body):
                new_body = [__(e)__]
            blocks[label_name(name + '_start')] = new_body
            return FunctionDef(name, params, blocks, None, returns, None)
        ...
```

Solution: (2 points each)
(a) Return(e)
(b) TailCall(func, args)
(c) Assign([Name(tmp)], cnd)
(d) explicate_tail(value, blocks)
(e) explicate_stmt(s, new_body, blocks)
9. **6 points** What is the purpose of the Reveal Functions pass? How does the output of Reveal Functions facilitate decisions made in later passes of the compiler?

**Solution:** The Reveal Functions pass separates the references to global functions from references to local variables, changing the former to `FunRef`. (2 points) The later passes can then treat them differently, in particular,

- In Remove Complex Operands, `FunRef` is treated as a complex operand to make sure it only appears on the righthand side of an assignment statement. (2 points)
- In Instruction Selection, each assignment with `FunRef` on the right-hand side is translated to a `lea`q instruction that obtains the function’s address from the function’s label. (2 points)

10. **8 points** Describe the general layout of the procedure call frame that your compiler uses.

**Solution:** The procedure call frame stores the following information:

- The return address, i.e., the address of the caller. (2 points)
- The caller’s value for `rbp`. (2 points)
- The caller’s values of the callee-saved registers that are going to be used in this function. (2 points)
- The spilled local variables. (2 points)
11. **13 points** Apply Instruction Selection to the following two functions, translating them from $C_{Fun}$ to $x86_{Def}^{callq}$. (The definitions of $C_{Fun}$ and $x86_{callq}^{Def}$ are in the Appendix, as is the list of argument-passing registers.) (The function `even` calls `odd`, but you do not need to translate the `odd` function for this exam question, so its definition is omitted.) (The below functions are presented in concrete syntax except for the `FunRef` nodes.)

```python
def even(x : int) -> bool:
    even_start:
        if x == 0:
            goto block.146
        else:
            goto block.147
    block.146:
        return True
    block.147:
        fun.138 = FunRef(odd, 1)
        tmp.139 = (x - 1)
        tail fun.138(tmp.139)

def main() -> int:
    main_start:
        fun.142 = FunRef(even, 1)
        tmp.143 = input_int()
        call.152 = fun.142(tmp.143)
        if call.152 == False:
            goto block.153
        else:
            goto block.154
    block.153:
        tmp.144 = 0
        goto block.151
    block.154:
        tmp.144 = 42
        goto block.151
    block.151:
        print(tmp.144)
        return 0
```

**Solution:** (approx. 1/2 point per instruction)

```assembly
def even() -> bool:
    even_start:
        movq %rdi, x
        cmpq $0, x
        je block.146
        jmp block.147
    block.146:
        movq $1, %rax
        jmp even_conclusion
    block.147:
        leaq odd(%rip), fun.138
        movq x, tmp.139
```

```assembly
leaq odd(%rip), fun.138
movq x, tmp.139
```

```assembly
def main() -> int:
    main_start:
        fun.142 = FunRef(even, 1)
        tmp.143 = input_int()
        call.152 = fun.142(tmp.143)
        if call.152 == False:
            goto block.153
        else:
            goto block.154
    block.153:
        tmp.144 = 0
        goto block.151
    block.154:
        tmp.144 = 42
        goto block.151
    block.151:
        print(tmp.144)
        return 0
```
```python
subq $1, tmp.139
movq tmp.139, %rdi
tailjmp fun.138

def main() -> int:
    main_start:
        leaq even(%rip), fun.142
callq read_int
movq %rax, tmp.143
movq tmp.143, %rdi
callq* fun.142
movq %rax, call.152
cmpq $0, call.152
je block.153
jmp block.154
block.151:
    movq tmp.144, %rdi
callq print_int
movq $0, %rax
jmp main_conclusion
block.153:
    movq $0, tmp.144
jmp block.151
block.154:
    movq $42, tmp.144
jmp block.151
```
12. **12 points** Draw the interference graph for the following program fragment by adding edges between the nodes below. You do not need to include edges between two registers. The live-after set for each instruction is given to the right of each instruction and the types of each variable is listed below. (The callee and caller saved registers are listed in the Appendix of this exam.)

```python
def main() -> int:
    var_types:
    tmp73 : int, tmp72 : int, tmp76 : Callable[[int], int],
    tmp74 : Callable[[Callable[[int], int], tuple[int, int]], Void], tmp71 : int,
    _65 : int, _64 : int, _66 : int, alloc63 : tuple[int, int],
    vec62 : tuple[int,int]

    mainstart:
    {}  
    movq free_ptr(%rip), tmp71
    {}  
    movq tmp71, tmp72
    {}  
    addq $24, tmp72
    {}  
    movq fromspace_end(%rip), tmp73
    {}  
    cmpq tmp73, tmp72
    {}  
    jl block77
    {}  
    jmp block78
    {}  
    block76:
    {}  
    movq free_ptr(%rip), %r11
    movq free_ptr(%rip), tmp71
    movq tmp71, tmp72
    addq $24, tmp72
    movq fromspace_end(%rip), tmp73
    cmpq tmp73, tmp72
    jl block77
    jmp block78
    block77:
    movq $0, _66
    jmp block76
    block78:
    movq %r15, %rdi
    movq $24, %rsi
    callq *tmp74
    jmp block76
```

**Solution:**

![Interference Graph](image)
Appendix

The caller-saved registers are:

\texttt{rax rcx rdx rsi rdi r8 r9 r10 r11}

and the callee-saved registers are:

\texttt{rsp rbp rbx r12 r13 r14 r15}

The argument-passing registers are:

\texttt{rdi rsi rdx rcx r8 r9}

Grammar for \( L \text{While} \)

\[
\begin{align*}
\text{exp} & : = \text{Constant(int)} \mid \text{Name}(\text{'input.int' }), [] \mid \text{UnaryOp(USub(), exp)} \\
& \quad \mid \text{BinOp(exp, Add(), exp)} \mid \text{BinOp(exp, Sub(), exp)} \\
\text{stmt} & : = \text{Expr(Call(Name(\text{'print'}, [exp])))} \mid \text{Expr(exp)}
\end{align*}
\]

\[
\begin{align*}
\text{exp} & : = \text{Name(var)} \\
\text{stmt} & : = \text{Assign(Name(var), exp)}
\end{align*}
\]

\[
\begin{align*}
\text{boolop} & : = \text{And()} \mid \text{Or()} \\
\text{unaryop} & : = \text{Not()} \\
\text{cmp} & : = \text{Eq()} \mid \text{NotEq()} \mid \text{Lt()} \mid \text{LtE()} \mid \text{Gt()} \mid \text{GtE()} \\
\text{bool} & : = \text{True} \mid \text{False} \\
\text{exp} & : = \text{Constant(bool)} \mid \text{BoolOp(boolop, [exp, exp])} \\
& \quad \mid \text{Compare(exp, [cmp], [exp])} \mid \text{IfExp(exp, exp, exp)} \\
\text{stmt} & : = \text{If(Compare(exp, [cmp], [exp]), [Goto(label)], [Goto(label)])}
\end{align*}
\]

\[
\begin{align*}
\text{stmt} & : = \text{While(exp, stmt, [])} \\
L\text{While} & : = \text{Module(stmt)}
\end{align*}
\]

Grammar for \( C\text{If} \)

\[
\begin{align*}
\text{atm} & : = \text{Constant(int)} \mid \text{Name(var)} \mid \text{Constant(bool)} \\
\text{exp} & : = \text{atm} \mid \text{Name('input.int'), []} \mid \text{UnaryOp(USub(), atm)} \\
& \quad \mid \text{BinOp(atm, Sub(), atm)} \mid \text{BinOp(atm, Add(), atm)} \\
& \quad \mid \text{Compare(atm, [cmp], [atm])} \\
\text{stmt} & : = \text{Expr(Call(Name('print'), [atm]))} \mid \text{Expr(exp)} \\
& \quad \mid \text{Assign([Name(var)], exp)} \\
\text{tail} & : = \text{Return(exp)} \mid \text{Goto(label)} \\
& \quad \mid \text{If(Compare(atm, [cmp], [atm]), [Goto(label)], [Goto(label)])}
\end{align*}
\]

\[
\begin{align*}
\text{C} \text{If} & : = \text{CProgram({label: [stmt, ... , tail], ... })}
\end{align*}
\]
Grammar for $\mathcal{L}_{\text{Tup}}$

\[
\begin{align*}
exp & ::= \quad \text{Constant}(\text{int}) \mid \text{Call}(\text{Name}(\text{`input\_int'},[])) \\
& \quad \mid \text{UnaryOp}(\text{USub()}, exp) \mid \text{BinOp}(exp, \text{Add()}, exp) \\
& \quad \mid \text{BinOp}(exp, \text{Sub()}, exp) \\
stmt & ::= \quad \text{Expr}(\text{Call}(\text{Name}(\text{`print'},[exp]))) \mid \text{Expr}(exp) \\
exp & ::= \quad \text{Name}(\text{var}) \\
stmt & ::= \quad \text{Assign}([\text{Name}(\text{var}]), exp) \\
boolop & ::= \quad \text{And()} \mid \text{Or()} \\
unaryop & ::= \quad \text{Not()} \\
cmp & ::= \quad \text{Eq()} \mid \text{NotEq()} \mid \text{Lt()} \mid \text{LtE()} \mid \text{Gt()} \mid \text{GtE()} \\
bool & ::= \quad \text{True} \mid \text{False} \\
exp & ::= \quad \text{Constant(\text{bool})} \mid \text{BoolOp}(\text{boolop}, [exp, exp]) \\
& \quad \mid \text{Compare}(exp, [cmp], [exp]) \mid \text{IfExp}(exp, exp, exp) \\
stmt & ::= \quad \text{If}(exp, stmt^+, stmt^+) \\
stmt & ::= \quad \text{While}(exp, stmt^+, []) \\
cmp & ::= \quad \text{Is()} \\
exp & ::= \quad \text{Tuple}(exp^+, \text{Load()}) \mid \text{Subscript}(exp, \text{Constant(\text{int})}, \text{Load()}) \\
& \quad \mid \text{Call}(\text{Name}(\text{`len'}, [exp])) \\
L_{\text{Tup}} & ::= \quad \text{Module}(stmt^*)
\end{align*}
\]

Grammar for $\mathcal{L}_{\text{Alloc}}$

The $\mathcal{L}_{\text{Alloc}}$ language extends $\mathcal{L}_{\text{Tup}}$ with the following grammar rules:

\[
\begin{align*}
exp & ::= \quad \text{Collect}(\text{int}) \mid \text{Allocate}(\text{int}, \text{type}) \mid \text{GlobalValue}(\text{name}) \\
stmt & ::= \quad \text{Assign}([\text{Subscript}(exp, \text{int}, \text{Store()}]), exp)
\end{align*}
\]
Grammar for $\mathcal{L}_{\text{Fun}}$

\[
\begin{align*}
\text{exp} &::= \text{Constant}(\text{int}) \mid \text{Call}(\text{Name}'\text{input\_int)'),[]) \\
&\quad\mid \text{UnaryOp}(\text{USub}(),\text{exp}) \mid \text{BinOp}(\text{exp},\text{Add}(),\text{exp}) \\
&\quad\mid \text{BinOp}(\text{exp},\text{Sub}(),\text{exp}) \\
\text{stmt} &::= \text{Expr}(\text{Call}(\text{Name}'\text{print'}',[\text{exp}])) \mid \text{Expr}(\text{exp}) \\
\text{exp} &::= \text{Name}(\text{var}) \\
\text{stmt} &::= \text{Assign}([\text{Name}(\text{var})],\text{exp}) \\
\text{boolop} &::= \text{And}() \mid \text{Or}() \\
\text{unaryop} &::= \text{Not}() \\
\text{cmp} &::= \text{Eq}() \mid \text{NotEq}() \mid \text{Lt}() \mid \text{LtE}() \mid \text{Gt}() \mid \text{GtE}() \\
\text{bool} &::= \text{True} \mid \text{False} \\
\text{exp} &::= \text{Constant}(\text{bool}) \mid \text{BoolOp}(\text{boolop},[\text{exp},\text{exp}]) \\
&\quad\mid \text{Compare}(\text{exp},[\text{cmp}],[\text{exp}]) \mid \text{IfExp}(\text{exp},\text{exp},\text{exp}) \\
\text{stmt} &::= \text{If}(\text{exp},\text{stmt}^+,\text{stmt}^+) \\
\text{stmt} &::= \text{While}(\text{exp},\text{stmt}^+,[])
\end{align*}
\]

\[
\begin{align*}
\text{exp} &::= \text{Tuple}(\text{exp}^+,\text{Load}()) \mid \text{Subscript}(\text{exp},\text{Constant}(\text{int}),\text{Load}()) \\
&\quad\mid \text{Call}(\text{Name}'\text{len'},[\text{exp}]) \\
\text{type} &::= \text{IntType}() \mid \text{BoolType}() \mid \text{VoidType}() \mid \text{TupleType}[\text{type}^+] \\
&\quad\mid \text{FunctionType}(\text{type}^*,\text{type}) \\
\text{exp} &::= \text{Call}(\text{exp},\text{exp}^*) \\
\text{stmt} &::= \text{Return}(\text{exp}) \\
\text{params} &::= (\text{var},\text{type})^* \\
\text{def} &::= \text{FunctionDef}(\text{var},\text{params},\text{stmt}^+,\text{None},\text{type},\text{None}) \\
\mathcal{L}_{\text{Fun}} &::= \text{Module}([\text{def}...\text{stmt}...])
\end{align*}
\]
Grammar for $\mathcal{L}^\text{mon}_{\text{FunRef}}$

\[
\begin{align*}
\text{atm} &::= \text{Constant}(\text{int}) \mid \text{Name}(\text{var}) \\
\text{exp} &::= \text{atm} \mid \text{Call}(\text{Name}(\text{input}_\text{int}),[]) \mid \text{UnaryOp}(\text{USub}(), \text{atm}) \mid \text{BinOp}(\text{atm}, \text{Add}(), \text{atm}) \mid \text{BinOp}(\text{atm}, \text{Sub}(), \text{atm}) \\
\text{stmt} &::= \text{Expr}(\text{Call}(\text{Name}(\text{print}),[\text{atm}])) \mid \text{Expr}(\text{exp}) \mid \text{Assign}([\text{Name}(\text{var})], \text{exp}) \\
\text{type} &::= \text{IntType()} \mid \text{BoolType()} \mid \text{VoidType()} \mid \text{TupleType}[\text{type}^+] \\
\text{exp} &::= \text{FunRef}(\text{label}, \text{int}) \mid \text{Call}(\text{atm}, \text{atm}^*) \\
\text{stmt} &::= \text{Return}(\text{exp}) \mid \text{Goto}(\text{label}) \mid \text{If}(\text{Compare}(\text{atm}, [\text{cmp}], [\text{atm}]), [\text{Goto}(\text{label})], [\text{Goto}(\text{label})]) \\
\text{def} &::= \text{FunctionDef}(\text{var}, \text{params}, \text{stmt}^+, \text{None}, \text{type}, \text{None}) \\
\mathcal{L}^\text{mon}_{\text{FunRef}} &::= \text{Module}([\text{def} \ldots \text{stmt} \ldots])
\end{align*}
\]

Grammar for $\mathcal{L}^\text{Fun}$

\[
\begin{align*}
\text{atm} &::= \text{Constant}(\text{int}) \mid \text{Name}(\text{var}) \mid \text{Constant}(\text{bool}) \\
\text{exp} &::= \text{atm} \mid \text{Call}(\text{Name}(\text{input}_\text{int}),[]) \mid \text{UnaryOp}(\text{USub}(), \text{atm}) \mid \text{BinOp}(\text{atm}, \text{Add}(), \text{atm}) \mid \text{BinOp}(\text{atm}, \text{Sub}(), \text{atm}) \\
\text{stmt} &::= \text{Expr}(\text{Call}(\text{Name}(\text{print}),[\text{atm}])) \mid \text{Expr}(\text{exp}) \mid \text{Assign}([\text{Name}(\text{var})], \text{exp}) \\
\text{tail} &::= \text{Return}(\text{exp}) \mid \text{Goto}(\text{label}) \mid \text{If}(\text{Compare}(\text{atm}, [\text{cmp}], [\text{atm}]), [\text{Goto}(\text{label})], [\text{Goto}(\text{label})]) \\
\text{exp} &::= \text{Subscript}(\text{atm}, \text{atm}, \text{Load}()) \mid \text{Call}(\text{Name}(\text{len}), [\text{atm}]) \\
\text{stmt} &::= \text{Collect}(\text{int}) \mid \text{Assign}([\text{Subscript}(\text{atm}, \text{atm}, \text{Store}())], \text{atm}) \\
\text{type} &::= \text{IntType()} \mid \text{BoolType()} \mid \text{VoidType()} \mid \text{TupleType}[\text{type}^+] \\
\text{exp} &::= \text{FunRef}(\text{label}, \text{int}) \mid \text{Call}(\text{atm}, \text{atm}^*) \\
\text{stmt} &::= \text{TailCall}(\text{atm}, \text{atm}^*) \\
\text{params} &::= [(\text{var}, \text{type}), \ldots] \\
\text{block} &::= \text{label} : \text{stmt}^* \text{ tail} \\
\text{blocks} &::= \{ \text{block}, \ldots \} \\
\text{def} &::= \text{FunctionDef}(\text{label}, \text{params}, \text{blocks}, \text{None}, \text{type}, \text{None}) \\
\mathcal{L}^\text{Fun} &::= \text{CProgramDefs}([\text{def} \ldots])
\end{align*}
\]
Grammar for $x86_{\text{callq}}^\text{Def}$

$$arg ::= \text{Constant}(\text{int}) \mid \text{Reg}(\text{reg}) \mid \text{Deref}(\text{reg}, \text{int}) \mid \text{ByteReg}(\text{reg})$$

$$\mid \text{Global}(\text{label}) \mid \text{FunRef}(\text{label}, \text{int})$$

$$instr ::= \ldots \mid \text{IndirectCallq}(arg, \text{int}) \mid \text{TailJmp}(arg, \text{int})$$

$$\mid \text{Instr}('\text{leaq}', [arg, \text{Reg}(\text{reg})])$$

$$block ::= label: instr^*$$

$$blocks ::= \{block, \ldots\}$$

$$def ::= \text{FunctionDef}(\text{label}, [], blocks, _, \text{type}, _)$$

$$x86_{\text{callq}}^\text{Def} ::= \text{X86ProgramDefs([def, \ldots])}$$