Compilers (Python)
CSCI P423/523, Fall 2022

Final

Name: ____________________

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

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

```
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)
```

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

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

3. $\boxed{\text{8 points}}$ Given the following $\mathcal{L}_{\mathsf{While}}$ program, apply the Explicate Control pass to translate it to $\mathcal{C}_{\mathsf{If}}$.

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

4. $\boxed{\text{13 points}}$ 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
```

5. $\boxed{\text{10 points}}$ Fill in the blanks for the following `expose_alloc_tuple` auxilliary function of the Expose Allocation pass that translates from $\mathcal{L}_{\mathsf{Tup}}$ to $\mathcal{L}_{\mathsf{Alloc}}$. (The grammar for $\mathcal{L}_{\mathsf{Alloc}}$ is in the Appendix.)

```
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))

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

6. $\boxed{\text{4 points}}$ In the `expose_alloc_tuple` function of the previous question, why are the intializing expressions `es` assigned to the temporary variables `xs` instead of using them directly in the tuple initialization?

7. $\boxed{\text{6 points}}$ Describe the layout of the 64-bit tag at the beginning of every tuple.

8. $\boxed{\text{10 points}}$ Fill in the blanks for the following `explicate_control` that translates $\mathcal{L}^{mon}_{\mathsf{FunRef}}$ programs into $\mathcal{C}_{\mathsf{Fun}}$ programs.

```
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)
        ...
```

9. $\boxed{\text{6 points}}$ 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?

10. $\boxed{\text{8 points}}$ Describe the general layout of the procedure call frame that your compiler uses.

11. $\boxed{\text{13 points}}$ Apply Instruction Selection to the following two functions, translating them from $\mathcal{C}_{\mathsf{Fun}}$ to $\text{x86}^{\mathsf{Def}}_{\mathsf{callq*}}$. (The definitions of $\mathcal{C}_{\mathsf{Fun}}$ and $\text{x86}^{\mathsf{Def}}_{\mathsf{callq*}}$ 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.)

```
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
```

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.)

```
def main() -> int:
  var_types:
    tmp73 : int, tmp72 : int, tmp75 : 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
    {tmp71}
movq tmp71, tmp72
    {tmp72}
addq $24, tmp72
    {tmp72}
movq fromspace_end(%rip), tmp73
    {tmp72 tmp73}
cmpq tmp73, tmp72
    {}
jl block77
    {}
jmp block78
    {}

block77:
     {}
movq $0, _66
    {}
jmp block76
    {}

block78:
    {}
movq %r15, %rdi
    {rdi}
movq $24, %rsi
    {rdi rsi}
callq collect
    {}
jmp block76
    {}

block76:
    {}
movq free_ptr(%rip), %r11
    {}
addq $24, free_ptr(%rip)
    {}
movq $5, 0(%r11)
    {r11}
movq %r11, alloc63
    {alloc63}
movq alloc63, %r11
    {alloc63}
movq $0, 8(%r11)
    {alloc63}
movq $0, _65
    {alloc63}
movq alloc63, %r11
    {alloc63}
movq $41, 16(%r11)
    {alloc63}
movq $0, _64
    {alloc63}
movq alloc63, vec62
    {vec62}
leaq map_vec_57(%rip), tmp74
    {vec62 tmp74}
leaq add158(%rip), tmp75
    {vec62 tmp74 tmp75}
movq tmp75, %rdi
    {tmp74 rdi vec62}
movq vec62, %rsi
    {rsi tmp74 rdi vec62}
callq *tmp74
    {vec62}
movq vec62, %r11
    {r11}
movq 16(%r11), %rax
    {rax}
jmp mainconclusion
    {rax}
)
```

Appendix

The caller-saved registers are:

```
rax rcx rdx rsi rdi r8 r9 r10 r11
```

and the callee-saved registers are:

```
rsp rbp rbx r12 r13 r14 r15
```

The argument-passing registers are:

```
rdi rsi rdx rcx r8 r9
```

Grammar for $\mathcal{L}_{\mathsf{While}}$

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

Grammar for $\mathcal{C}_{\mathsf{If}}$

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

Grammar for $\mathcal{L}_{\mathsf{Tup}}$

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

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

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

$$
\begin{array}{rcl}
\mathit{exp} & ::= & \texttt{Collect}(\mathit{int}) \mid \texttt{Allocate}(\mathit{int}, \mathit{type}) \mid \texttt{GlobalValue}(\mathit{name}) \\
\mathit{stmt} & ::= & \texttt{Assign([Subscript(}\mathit{exp}\texttt{,}\mathit{int}\texttt{,Store())], }\mathit{exp}\texttt{)}
\end{array}
$$

Grammar for $\mathcal{L}_{\mathsf{Fun}}$

exp	::=	`Constant(`*int*`)` | `Call(Name('input_int'),[])`
	|	`UnaryOp(USub(),`*exp*`)` | `BinOp(`*exp*`,Add(),`*exp*`)`
	|	`BinOp(`*exp*`,Sub(),`*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^{+}`)`

stmt	::=	`While(`*exp*`, `stmt^{+}`, [])`

cmp	::=	`Is()`
exp	::=	`Tuple(`exp^{+}`,Load())` | `Subscript(`*exp*`,Constant(`*int*`),Load())`
	|	`Call(Name('len'),[`*exp*`])`

type	::=	`IntType()` | `BoolType()` | `VoidType()` | `TupleType[`type^{+}`]`
	|	`FunctionType(`type^{*}`, `*type*`)`
exp	::=	`Call(`*exp*`, `exp^{*}`)`
stmt	::=	`Return(`*exp*`)`
params	::=	`(`*var*`,`*type*`)`*
def	::=	`FunctionDef(`*var*`, `*params*`, `stmt^{+}`, None, `*type*`, None)`
$\mathcal{L}_{\mathsf{Fun}}$	::=	`Module([`*def* … *stmt* …`])`

Grammar for $\mathcal{L}^{mon}_{\mathsf{FunRef}}$

$$
\begin{array}{rcl}
atm &::=& \texttt{Constant(}int\texttt{)} \mid \texttt{Name(}var\texttt{)}\\
exp &::=& atm \mid \texttt{Call(Name('input\_int'),[])}\\
&\mid& \texttt{UnaryOp(USub(),}atm\texttt{)} \mid \texttt{BinOp(}atm\texttt{,Add(),}atm\texttt{)}\\
&\mid& \texttt{BinOp(}atm\texttt{,Sub(),}atm\texttt{)}\\
stmt &::=& \texttt{Expr(Call(Name('print'),[}atm\texttt{]))} \mid \texttt{Expr(}exp\texttt{)}\\
&\mid& \texttt{Assign([Name(}var\texttt{)], }exp\texttt{)}\\
\hline
atm &::=& \texttt{Constant(}bool\texttt{)}\\
exp &::=& \texttt{Compare(}atm\texttt{,[}cmp\texttt{],[}atm\texttt{])} \mid \texttt{IfExp(}exp\texttt{,}exp\texttt{,}exp\texttt{)}\\
&\mid& \texttt{Begin(}stmt^*\texttt{, }exp\texttt{)}\\
stmt &::=& \texttt{If(}exp\texttt{, }stmt^*\texttt{, }stmt^*\texttt{)}\\
\hline
stmt &::=& \texttt{While(}exp\texttt{, }stmt^+\texttt{, [])}\\
\hline
exp &::=& \texttt{Subscript(}atm\texttt{,}atm\texttt{,Load())}\\
&\mid& \texttt{Call(Name('len'),[}atm\texttt{])}\\
&\mid& \texttt{Allocate(}int, type\texttt{)} \mid \texttt{GlobalValue(}var\texttt{)}\\
stmt &::=& \texttt{Assign([Subscript(}atm\texttt{,}atm\texttt{,Store())], }atm\texttt{)}\\
&\mid& \texttt{Collect(}int\texttt{)}\\
\hline
type &::=& \texttt{IntType()} \mid \texttt{BoolType()} \mid \texttt{VoidType()} \mid \texttt{TupleType[}type^+\texttt{]}\\
&\mid& \texttt{FunctionType(}type^*\texttt{, }type\texttt{)}\\
exp &::=& \texttt{FunRef(}label\texttt{, }int\texttt{)} \mid \texttt{Call(}atm\texttt{, }atm^*\texttt{)}\\
stmt &::=& \texttt{Return(}exp\texttt{)}\\
params &::=& \texttt{(}var\texttt{,}type\texttt{)}^*\\
def &::=& \texttt{FunctionDef(}var\texttt{, }params\texttt{, }stmt^+\texttt{, None, }type\texttt{, None)}\\
\mathcal{L}^{mon}_{\mathsf{FunRef}} &::=& \texttt{Module([}def \ldots stmt \ldots\texttt{])}
\end{array}
$$

Grammar for $\mathcal{C}_{\mathsf{Fun}}$

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

Grammar for $\text{x86}^{\mathsf{Def}}_{\mathsf{callq*}}$

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