Right after the last lecture, your professor realized, to his horror, that we finished 131 without actually implementing a real language, which must, of course, have pointers, mutation and loops! In the final, we will implement these features so that, for example, we can compile imperative programs like:

```python
def fib(n):
    i = (0, false)
    vals = (0, 1)
    in while (i[0] < n):
        ( let next = vals[0] + vals[1] in
          i[0] := i[0] + 1;
          vals[0] := vals[1];
          vals[1] := next
        );
    in fib(10)
```
Part I. Updating Tuples [35pts]

First, let's add support for updating tuples. Concretely, that means that the program

\[
\text{let } t = (10, 20), \ a = (t[0] := t[0] + 2) \text{ in } (t[0], t[1])
\]

should evaluate to \((12, 20)\), as the first line creates a tuple, and the second line executes an tuple-update expression to increments the 0-th field by 2.

Similarly,

\[
\text{let } t = (10, 20), \ a = t[1] := t[1] + 2 \text{ in } (t[0], t[1])
\]

updates the 1-th field and hence, should evaluate to \((10, 22)\)

Q1: Represent [3 pts]

Let's represent updates by extending the \texttt{Expr} type with a \texttt{SetItem} constructor:

\[
\text{data Expr a} \\
\quad = \ldots \\
\quad \mid \text{SetItem (Expr a) Field (Expr a) a}
\]

The first \texttt{Expr} is the tuple being updated, the \texttt{Field} describes which part of the tuple is changed, and the second \texttt{Expr} is the value that it is changed to. As before, a \texttt{Field} is defined as:

\[
\text{data Field} = \text{Zero} \mid \text{One}
\]

Intuitively, the update expression \texttt{e1[fld] := e2} will be represented by \texttt{SetItem e1 fld e2 l} (where \(l\) is the tag meta-data as in your assignment.)

Fill in the blank below to get a Haskell representation of \texttt{t[0] := t[0] + 2}

\[
\text{ans1 :: Expr ()} \\
\text{ans1 = } _____________________________
\]
Q2: ANF Example [5pts]

What is the ANF form of

\[ t[0] := t[0] + 2 \]

fill in the blanks below to get the ANF version of the above

\[
\text{let } \underline{\text{_______}} = \underline{\text{______________________________}}, \underline{\text{_______}} = \underline{\text{______________________________}} \\
\text{in} \underline{\text{_____________________________________________}}
\]

Q3: ANF [5pts]

Assume that in \( e1[f] := e2 \) we want to first evaluate the tuple \( e1 \) and then evaluate \( e2 \).

Next, fill in the blanks below to extend \( \text{anf} \) to handle the case for \( \text{SetItem} \)

\[
\text{anf} :: \text{Int} \rightarrow \text{Expr } a \rightarrow (\text{Int}, \text{AnfExpr } a) \\
\text{anf } i \ (\text{SetItem } e1 \ f1d \ e2 \ l) = \underline{\text{______________________________}} \\
\text{where} \underline{\text{______________________________}} = \underline{\text{______________________________}}
\]

\textit{HINT:} Use \( \text{imms} \) and \( \text{stitch} \) described in the Appendix.

Q4: Type Inference [7pts]

Next, let's extend the type inference function to handle tuple updates.

Just like \( \text{Prim1}, \text{Prim2}, \text{If}, \text{Tuple} \) and \( \text{GetItem} \) we can simply treat \( \text{SetItem} \) as a special kind of function call that takes two parameters, the source tuple, and the updated value.

\[
\text{ti} :: (\text{Located } a) \Rightarrow \text{TypeEnv} \rightarrow \text{Subst} \rightarrow \text{Expr } a \rightarrow (\text{Subst}, \text{Type}) \\
\text{ti } \text{env} \ \text{su} \ (\text{GetItem } e \ f \ l) = \text{instApp } (\text{sourceSpan } l) \ \text{env} \ \text{su} \ (\text{fieldPoly } f) \ [e] \\
\text{ti } \text{env} \ \text{su} \ (\text{SetItem } e \ f \ e' \ l) = \text{instApp } (\text{sourceSpan } l) \ \text{env} \ \text{su} \ (\text{updatePoly } f) \ [e1, e2]
\]

Complete the code for \( \text{ti} \) by completing the definition for \( \text{updatePoly} \).
updatePoly :: Field -> Poly

updatePoly Zero = Forall [____] ([_______, ________] :=> _________ )

updatePoly One = Forall [____] ([_______, ________] :=> _________ )

The output value will be the same as that being assigned. That is,

• (t[0] := false) should evaluate to false, and
• (t[0] := 12) + 1 should evaluate to 13.

HINT: See how ti was implemented for Prim1 in the Appendix.

Q5: Assemble [7pts]

Lets assume that as in the assignments:

1. Variables at stack position i have their value at address [ebp - 4 * i],
2. The variable t lives at position 10 on the stack,
3. Tuple pointers are 8-byte aligned, and end with 001 (in binary.)
4. Tuples are layed out in the heap as in egg-eater so, a tuple of values v0,\ldots,v_n is layed out in the heap as below and user-constructed tuples (i.e. not closures) tuples have n = 1 (i.e. just v0, v1).

\[
\begin{array}{cccc}
\mid n + 1 \mid v0 \mid v1 \mid \ldots \mid v_n \mid \\
\end{array}
\]

Consider the expression

t[0] := 12

Assume that t lives at position 10 on the stack. Fill in the blanks below to get the assembly generated by the above expression. Recall that (t[0] := 12) + 1 should evaluate to 13, so at the end, eax should hold (the representation of) 12.

```
# __________________________
# __________________________
# __________________________
# __________________________
```

4
Q6: Compile [8pts]

Inspired by the above, fill in the implementation that compiles an assignment expression.

\[
\text{compileEnv :: Env -> AExp -> [Instruction]}
\]

\[
\text{compileEnv env (SetItem v1 fld v2)}
\]

\[
= [ \text{---------------------------------------------}

, \text{---------------------------------------------}

, \text{---------------------------------------------}

, \text{---------------------------------------------}

]
\]

\text{where}

\[
\text{fOff :: Field -> Int}
\]

\[
fOff \text{ Zero } = 4
\]

\[
fOff \text{ One } = 8
\]

\text{HINT: You may want to use the helper immArg in the Appendix}
Part II. Sequencing [30pts]

Next, let's implement sequencing, i.e. evaluating one expression after another, so, for example:

```haskell
let t = (0, 0)
  in
  t[0] := 2;
  t[1] := 6;
  t[0] + t[1]
```

should evaluate to 8. That is e1; e2 should evaluate e1 and then e2 and then evaluate the value that e2 produced.

Q7: Represent [5pts]

Let's represent sequenced expressions as:

```haskell
data Expr a
  = ...
  | Seq (Expr a) (Expr a) a
```

The first Expr is executed first, and then the second Expr should be executed.

Fill in the blanks below to show how the expression

```haskell
t[0] := 2;
  t[1] := 6;
  t[0] + t[1]
```

can be represented as an Expr

```haskell
ans7 :: Expr ()

ans7 = Seq ( ___________________________________________ )
  (Seq ( ___________________________________________ )
    ( ___________________________________________ )()) ()
```
Q8: ANF Example [5pts]

Fill in the blanks below to get an A-Normal Form of:

\[
\begin{align*}
    t[0] & := t[0] + 2; \\
    t[0] & := t[0] + 6
\end{align*}
\]

**HINT:** Be careful about that ; !

Q9: ANF [5pts]

Fill in the blanks below to implement \textbf{anf} for sequences.

\[
\text{anf} :: \text{Int} \to \text{Expr} \ a \to (\text{Int}, \text{AnfExpr} \ a)
\]

\[
\text{anf} \ i \ (\text{Seq} \ e1 \ e2 \ l) = \quad \text{------------------------}
\]

\[
\text{where}
\]

\[
\quad \quad \quad \quad \text{------------------------}
\]

\[
\quad \quad \quad \quad \text{------------------------}
\]

\[
\quad \quad \quad \quad \text{------------------------}
\]

\[
\quad \quad \quad \quad \text{------------------------}
\]

**HINT:** You definitely don’t need all the space given above...
Q10: Type Inference [5pts]

Next, let’s extend the type inference function to handle sequences.
Again, we can do so by treating `Seq` as a special kind of function call that “takes” two parameters, the first and second expressions, and “returns” the second expression’s value as the result.

\[
ti :: (Located a) => TypeEnv -> Subst -> Expr a -> (Subst, Type)
\]

\[
ti \ env\ su\ (Seq\ e1\ e2\ l) =\ instApp\ (sourceSpan\ l)\ env\ su\ seqPoly\ [e1,\ e2]
\]

Complete the above implementation by filling in the definition of `SeqPoly`

\[
seqPoly :: Poly
\]

\[
seqPoly =\ Forall\ [_______]\ ([_______,_______] :=> _________)
\]

Q11: Assemble [5pts]

Fill in the blanks below to show the assembly that should be generated for

\[
t[0] := 12;\ t[1] := add1(t[0])
\]

Again, assume that `t` lives at position 10 on the stack.

```
------------------------------------------------------------- #  -------------------------
------------------------------------------------------------- #  -------------------------
------------------------------------------------------------- #  -------------------------
------------------------------------------------------------- #  -------------------------
------------------------------------------------------------- #  -------------------------
------------------------------------------------------------- #  -------------------------
------------------------------------------------------------- #  -------------------------
------------------------------------------------------------- #  -------------------------
------------------------------------------------------------- #  -------------------------
------------------------------------------------------------- #  -------------------------
```

8
Q12: Compile [5pts]

Next, fill in the implementation of `compileEnv` for sequences.

```haskell
compileEnv :: Env -> AExp -> [Instruction]
compileEnv env (Seq e1 e2 _)
```

HINT: You definitely don’t need all the space given above...
Part III. While [40pts]

Finally, let's add support for while loops.
Concretely that means that the expression:

```
let t = (0, 1) in
while (t[0] < 10):
    ( t[0] := t[0] + 1
    );
  t[1]
```

should evaluate to 1024.

Q13: Represent [4pts]

Let's represent while loops by extending Expr as:

```haskell
data Expr a
  = ...
  | While (Expr a) (Expr a) a
```

The first Expr is the loop “condition” and the latter is the loop “body”.
Fill in the blanks below to show how the expression

```
while (t[0] < 10):
    t[0] := add1(t[0])
```

can be represented as an Expr

```haskell
ans13 :: Expr ()
ans13 = While ( ________________________________ )
          ( ________________________________ ) ()
```

Q14: ANF Example [6pts]

Consider the expression:

```
while (t[0] < 10):
    t[0] := add1(t[0])
```
Fill in the blanks below to get an A-Normal Form representation of the above.

---
---
---
---

Q15: ANF [5pts]
Drawing inspiration from the above, fill in the blanks below to implement `anf` for `while`

\[
anf :: \text{Int} \to \text{Expr} \ a \to (\text{Int}, \text{AnfExpr} \ a)
\]

\[
anf \ i \ (\text{While} \ e1 \ e2 \ l) = \ldots
\]

where

---
---
---
---

HINT Just because I gave you four lines doesn’t mean you need to use them.

Q16: Type Inference [7pts]
Stop me if you’ve heard this before: we can do type inference for `while` using a special function call that takes two \textit{input} parameters: the “condition” and “body” expressions, and returns an \textit{output} that is ... ? I don’t know! Can you help me complete `ti` for `While` by filling in the definition of `whilePoly`?

\[
ti :: (\text{Located} \ a) \to \text{TypeEnv} \to \text{Subst} \to \text{Expr} \ a \to (\text{Subst}, \text{Type})
\]

\[
ti \ \text{env} \ \text{su} \ (\text{While} \ e1 \ e2 \ l) = \text{instApp} \ (\text{sourceSpan} \ l) \ \text{env} \ \text{su} \ \text{whilePoly} \ [e1, e2]
\]

\[
\text{whilePoly} :: \text{Poly}
\]

\[
\text{whilePoly} = \text{Forall} \ [\ldots] \ ([\ldots], \ldots] :\Rightarrow \ldots
\]
Q17: Assemble [8pts]

Suppose that

- `instrs1` is a list of instructions evaluating $t[0] < 10$, at the end of which `eax` holds the representation for `true` if $t[0]$ was indeed less than 10 (and `false` otherwise)
- `instrs2` is a list of instructions evaluating $t[0] := \text{add1}(t[0])$,

Use `instrs1` and `instrs2`, to obtain the assembly corresponding to:

```python
while (t[0] < 10):
    t[0] := \text{add1}(t[0])
```

**HINT** You may write `repr True` and `repr False` if you need to use the (representations) of `true` and `false` in the assembly below.

```

```
Q18: Compile [10pts]

Drawing inspiration from the above, complete the implementation of `compileEnv` for `while` loops. You can assume that `tagLabel` generates assembly control flow labels:

```haskell
tagLabel :: Int -> Tag -> Label
```

and that `l` has type `Tag`, a unique value for each sub-expression.

```haskell
compileEnv :: Env -> AExp -> [Instruction]
compileEnv env (While e1 e2 l)
    = ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________
    ______________________________________________________________

where

```haskell
    labelBegin :: Label
    labelBegin = tagLabel 0 l

    labelEnd :: Label
    labelEnd = tagLabel 1 l
```

**HINT:** See the appendix for a list of assembly instructions.
Part IV. Recursion via Mutation [75pts]

Lets rewind and assume we do not have while-loops in the language. It turns out that with tuple assignment (i.e. mutation), we can get rid of recursive functions, i.e. we can implement recursive def functions by combining mutable tuples and plain old lambda functions.

Q19: Factorial without Recursion [35pts]

Consider the following lambda-expression that is a wrapper around a recursive definition of factorial

\[
\text{lambda}(m): \\
\text{def factorial(n):} \\
\quad \text{if (n < 1): 1 else: n * factorial(n - 1) in} \\
\quad \text{factorial(m)}
\]

Write an equivalent lambda-expression that

- does not use def i.e. does not use recursive functions, but
- does use tuple assignment (and lambda)

to have the the same behavior as the original, i.e. computes the factorial function.

\[
\text{lambda}(m): \\
\text{let _________ = ________________________________ in} \\
\quad ( _________ := ________________________________ ) ; \\
\quad ______ (m)
\]

HINT: Assume that the above (non-recursive) program need not be type-checked.

Q20: Translating Recursion to Mutation [40pts]

Recall from FDL that internally, def was represented as a named function Fun, so the (recursive version) of the above code is internally represented as an Expr that looks something like:
(Lam ["m"]
  (Let "factorial"
    (Fun "factorial" ["n"] ( If (n < 1) 1 (n * App "factorial" [n - 1]))
      (App "factorial" ["m"])))
)

Complete the implementation of noFun that systematically performs the above translation; i.e. which would convert Expr that contain named (recursive) functions Fun f xs e into equivalent programs that contain no occurrences of Fun and hence, no explicitly recursive functions.

noFun :: Expr a -> Expr a
noFun e = go e
  where
    go (Number n l) = __________________________________________
    go (Id x l) = __________________________________________
    go (Prim2 o e1 e2 l) = __________________________________________
    go (If b e1 e2 l) = __________________________________________
    go (Tuple e1 e2 l) = __________________________________________
    go (GetItem e1 f l) = __________________________________________
    go (App e es l) = __________________________________________
    go (Lam xs e l) = __________________________________________
    go (Seq e1 e2 l) = __________________________________________
    go (SetItem e1 f e2 l) = __________________________________________
    go (Fun f xs e l) = __________________________________________

    -------------------------------------------------------------------

HINT: The interesting stuff happens in the case for Fun. You may assume that you have at your disposal, a library function such that substitute e (x, e') replaces all "free" occurrences of x inside e with e'.

substitute :: Expr a -> (Id a, Expr a) -> Expr a

15
Appendix

Type Definitions

-- | ANF Expressions labeled with a unique Tag
type AExp = Expr Tag

-- | Representing Expressions
data Expr a
  = ...
    | Number Integer a
    | Id Id a
    | Prim2 Prim2 (Expr a) (Expr a) a
    | Tuple (Expr a) (Expr a) a
    | GetItem (Expr a) Field a
    | SetItem (Expr a) Field (Expr a) a -- NEW

-- | Fields
data Field = Zero | One

-- | Primitive Operations
data Prim2 = ... | Plus

-- | Polymorphic Types
data Poly = Forall [TVar] Type -- forall a. a -> a -> Bool

data Type = TVar TVar -- a
  | TInt -- Int
  | TBool -- Bool
  | [Type] => Type -- (t1,...,tn) => t2
  | TPair Type Type -- (t0, t1)

-- | Machine (x86) Instructions
data Instruction
  = IMov Arg Arg
  | IAdd Arg Arg
  | ISub Arg Arg
  | IMul Arg Arg
  | IShr Arg Arg
  | ISar Arg Arg
  | ISlh Arg Arg
  | IAnd Arg Arg
  | IOr Arg Arg
  | IXor Arg Arg
  | ILabel Label
  | IPush Arg
functions for ANF conversion

```
imms :: Int -> [Expr a] -> (Int, Binds a, [ImmExpr a])

stitch :: Binds a -> AnfExpr a -> AnfExpr a
```

-- machine arguments

```hs
data Arg
  = Const Int
  | HexConst Int
  | Reg Reg
  | RegOffset Nat Reg
  | RegIndex Reg Reg
  | Sized Size Arg
  | CodePtr Label
  | GlobVar Text
```

-- registers

```hs
data Reg
  = EAX | EBX | ECX
  | ESP | EBP | ESI
```

```
Functions for ANF Conversion

-- `imms i es` takes as input a "start" counter `i` and expressions `es`, and
-- and returns an output `(i', bs, es')` where
-- * `i'` is the output counter (i.e. `i' = i`) anf-variables were generated
-- * `bs` are the temporary binders needed to convert `es` to immediate vals
-- * `es'` are the immediate values equivalent to `es`

imms :: Int -> [Expr a] -> (Int, Binds a, [ImmExpr a])

-- `stitch bs e` takes a "context" `bs` which is a list of temp-vars and their
-- definitions, and an expression `e` that uses the temp-vars in `bs` and glues
-- them together into a `Let` expression.

stitch :: Binds a -> AnfExpr a -> AnfExpr a
```
Functions for Type Inference

ti :: (Located a) => TypeEnv -> Subst -> Expr a -> (Subst, Type)
ti env su (Prim1 p e l) = instApp (sourceSpan l) env su (prim1Poly p) [e]

prim1Poly :: Prim1 -> Poly
prim1Poly Add1 = Forall [] ([TInt] :=> TInt)
prim1Poly Sub1 = Forall [] ([TInt] :=> TInt)
prim1Poly Print = Forall ["a"] (["a"] :=> "a")

Functions for Compiling

-- | immArg converts an immediate value, i.e. a Number, Boolean or Id
-- (on the stack) into an Arg
immArg :: Env -> ImmExpr a -> Arg