1. **Explicit Initialization** (65 pts.)

   Compound data structures, e.g., arrays, tuples, and records, often need to be initialized step by step, rather than being created atomically.

   For example, in the Java language, when an object is created, before the execution of its constructor, all the non-primitive-typed fields have the default value `null`. The object is then gradually initialized using individual assignments to the fields.

   Now let us try to model step-by-step initialization of tuples in the context of the simply-typed λ-calculus. Here is the language that we want a programmer to use (surface language):

Ground values	b	::=	$\texttt{true} \mid \texttt{false} \mid n$
Values	v	::=	$b \mid \lambda x{:}T.\, e \mid (v_1, \ldots, v_n)$
Terms	e	::=	$v \mid x \mid e_1\ e_2 \mid \texttt{malloc}\ T_1 \times \ldots \times T_n \mid \#i\ e \mid \#i\ e_1 := e_2$
Ground types	B	::=	$\mathsf{Bool} \mid \mathsf{Int}$
Types	T	::=	$B \mid T_1 \to T_2 \mid (T_1 \times \ldots \times T_n)\backslash\{i_1, \ldots, i_k\}$

   In order to create a tuple, the expression $\texttt{malloc}\ T_1 \times \ldots \times T_n$ is used, rather than $(e_1, \ldots, e_n)$ which—as we saw in class—creates a fully initialized tuple at once. The result of $\texttt{malloc}\ T_1 \times \ldots \times T_n$ is a fully *uninitialized* tuple, $(\texttt{null}, \ldots, \texttt{null})$, of type $(T_1 \times \ldots \times T_n)\backslash\{1, \ldots, n\}$.

   The type $(T_1 \times \ldots \times T_n)\backslash\{i_1, \ldots, i_k\}$ is called a *masked type*, which represents a tuple that has not been fully initialized: the elements numbered $i_1, \ldots, i_k$ are *masked*— that is, they are not initialized and have the value `null`. The tuple, after being fully initialized, should have the type $T_1 \times \ldots \times T_n$, which we assume is syntactic sugar for the type $(T_1 \times \ldots \times T_n)\backslash\{\}$.

   Notice that the surface language does not include values of the form `null`—that is, programmers cannot use this value form in the programs they write. However, `null` values do appear in running programs inside unitialized tuples so they must be included in the internal language. Here is the complete grammar for the internal language:

Ground values	b	::=	$\texttt{true} \mid \texttt{false} \mid n$
Uninitialized or Values	u	::=	$\texttt{null} \mid v$
Values	v	::=	$b \mid \lambda x{:}T.\, e \mid (u_1, \ldots, u_n)$
Terms	e	::=	$v \mid x \mid e_1\ e_2 \mid \texttt{malloc}\ T_1 \times \ldots \times T_n \mid \#i\ e \mid \#i\ e_1 := e_2$
Ground types	B	::=	$\mathsf{Bool} \mid \mathsf{Int}$
Types	T	::=	$B \mid T_1 \to T_2 \mid (T_1 \times \ldots \times T_n)\backslash\{i_1, \ldots, i_k\}$

   Tuples are initialized *functionally* with expressions $\#i\ e_1 := e_2$, in which e_1 first evaluates to a tuple with its i-th element masked, and e_2 evaluates to a value that is compatible with the type of the i-th element in the tuple. The expression will generate a new tuple with its i-th element initialized, and otherwise the same as the result of e_1. Note that each element of a tuple should only be initialized once.

   To project the i-th element of a tuple, the expression $\#i\ e$ is used. Note, however, that projection of uninitialized elements is prohibited.

For example, the following expression will evaluate to a tuple $(10, 20)$ of type $\mathsf{Int} \times \mathsf{Int}$.

$$\begin{array}{l}(\lambda x : (\mathsf{Int} \times \mathsf{Int}) \backslash \{2\}.\ \#2\ x := 20) \\ \quad ((\lambda x : (\mathsf{Int} \times \mathsf{Int}) \backslash \{1, 2\}.\ \#1\ x := 10) \\ \quad\quad (\texttt{malloc}\ \mathsf{Int} \times \mathsf{Int}))\end{array}$$

(a) (7 pts) Extend the small-step operational semantics of the simply-typed λ-calculus to include the new expressions: $(u_1, \ldots, u_n)$, $\texttt{malloc}\ T_1 \times \ldots \times T_n$, $\#i\ e$, and $\#i\ e_1 := e_2$. Specifically, assuming left-to-right evaluation, extend the definition of the evaluation contexts and give the additional reduction rules required.

(b) (10 pts) Extend the typing rules of the simply-typed λ-calculus to include the new constructs.

(c) (15 pts) Construct a Redex model of this language. Use of metafunctions should be kept to a minimum. Be sure to include sufficient tests. For a Redex model, about 50% of your file should be tests. (Put this in the file `7.rkt`.)

(d) (4 pts) Create two syntactically correct, surface-language programs that cause a run-time error in the reduction semantics. The two examples should illustrate different issues related to tuples. (Put these examples in `README.txt`.)

(e) (3 pts) Define the function `typed-evaluate`. It consumes grammatically correct programs, type-checks them, and if they check out, runs them on the reduction semantics to produce a result. If a program fails to type check, the function returns "`type error`". Include tests in `7.rkt` showing that `typed-evaluate` returns "`type error`" for the two programs from part (d) that caused run-time errors.

(f) (8 pts) Provide two syntactically correct, surface-language programs that do not cause a run-time error according to the reduction semantics, but fail to type check. (Include tests in `7.rkt` showing this.) The two examples should illustrate two different issues related to tuples. For each of the two issues, explain in 30 words or less what the issue is. (Provide the examples and explanation in `README.txt`.)

(g) (18 pts) Prove the soundness of the type system. For each of the lemmas involved, you only need to show the proofs for cases that involve the new constructs. (Turn in this part on paper. Your paper write-up should include the operational semantics from part (a) and typing rules from part (b) so we do not have to refer to your Redex model when checking your type soundness proof.)