1. Consider the following unambiguous context-free grammar over the symbols $a$ and $b$ that accepts palindromes—strings that read the same forward and backward. For example, $a$, $bab$, and $aabaa$ are all accepted by this grammar, whereas $ab$, $aab$, and $baaaba$ are not.

\[
P ::= \\
| \epsilon \\
| a \\
| b \\
| aPa \\
| bPb
\]

(a) (Yes or No) Is this grammar deterministically parsable by an LL(1) parser? Briefly explain.

(b) (Yes or No) Is this grammar deterministically parsable by an LR(1) parser? Briefly explain.

(c) (Yes or No) Is it possible to write a regular expression that matches the same strings accepted by this grammar? Briefly explain.
2. Consider the following unambiguous context-free grammar over the symbols a and b that accepts balanced strings in which every a is followed by a matching b. For example, ab, aabb, and aababb are all accepted, but ba, a, and aab are not.

\[
Q ::= \\
\quad \epsilon \\
\quad QaQb
\]

(a) (Yes or No) Is this grammar deterministically parsable by an LL(1) parser? Briefly explain.

(b) (Yes or No) Is this grammar deterministically parsable by an LR(1) parser? Briefly explain.

(c) (Yes or No) Is it possible to write a regular expression that matches the same strings accepted by this grammar? Briefly explain.
3. The following ambiguous grammar of mathematical expressions includes infix binary multiplication \( \ast \) and exponentiation operators \( \hat{\ast} \) as well as tokens representing variables (ranged over by \( x \)).

\[
E ::= \\
  | \ x \\
  | E \ast E \\
  | E \hat{\ast} E 
\]

Write down a disambiguated grammar that accepts the same set of strings, but makes multiplication left associative, exponentiation right associative, and gives exponentiation higher precedence than multiplication. For example, the token sequence \( x \ast y \hat{\ast} z \hat{\ast} w \ast v \ast u \) would be parsed as though it had parentheses inserted like: \((x \ast (y \hat{\ast} (z \hat{\ast} w))) \ast v) \ast u\).
4. **Typechecking**

In this problem we will consider the typechecking rules for a simple Haskell-like programming language that has built-in support for *sum types*. To aid your intuition, we could implement sum types in Haskell using the following type declaration:

```haskell
data Either a b
  = Left a
  | Right b
```

Our new language will use the notation $\tau_1 + \tau_2$ as syntax for a built-in type that acts like `Either $\tau_1 \tau_2$` does in Haskell. We create values of type $\tau_1 + \tau_2$ using the `Left` and `Right` constructors, and we inspect such values using `case`, whose semantics is the same as in Haskell.

The grammar and (most of the) typechecking rules for this language is given in Appendix A.

(a) Which of the following terms are well-typed according to the rules shown in the Appendix? (Circle all correct answers.)

i. let x = 3 in (let x = 4 in x)

ii. let x = (let x = 4 in x) in x

iii. let x = (let y = 4 in x) in y

iv. let x = 3 in Left x

(b) Recall that in our course compiler, we implemented the typechecker by viewing the judgment $\Gamma \vdash e : \tau$ as a Haskell function that takes in (Haskell representations of) $\Gamma$ and $e$ as inputs and produces (the Haskell representation of) $\tau$ as the output. That same strategy can no longer be used for this toy language, even with just the rules shown in the appendix. Briefly explain why not.

(c) Write the correct inference rule for the `case` expression. Assume that the operational behavior of `case` is the same as in Haskell. It is OK if variables bound in the case branches shadow occurrences from an outer scope.
5. **Optimization**

Consider the following function written in Java.

```java
int foo(int n, int[] a, int[] b) {
    int sum = 0;
    for(int i = 0; i < n; i = i + 1) {
        a[i] = a[i] + b[2];
        sum = sum + a[i];
    }
    return sum;
}
```

Now consider this *incorrectly* “optimized” version, which tries to hoist the expression `b[2]` outside of the loop:

```java
int foo_opt(int n, int[] a, int[] b) {
    int sum = 0;
    int tmp = b[2];
    for(int i = 0; i < n; i = i + 1) {
        a[i] = a[i] + tmp;
        sum = sum + a[i];
    }
    return sum;
}
```

There are at least *two* different reasons why the proposed “optimization” is incorrect, one having to do with aliasing, and one that does not involve aliasing.

(a) First complete the code below to give a set of inputs that *use* aliasing to demonstrate that the calls `foo(n,a,b)` and `foo_opt(n,a,b)` have different behaviors:

```java
int[] a = new int[] {0, 1, 2, 3};
int n =
int[] b =
```

(b) Now complete the code snippet to give a set of inputs that does not use aliasing, yet still demonstrates that the calls `foo(n,a,b)` and `foo_opt(n,a,b)` have different behaviors:

```java
int[] a = new int[] {0, 1, 2, 3};
int n =
int[] b =
```
6. We have seen that a source C variable declaration like the one shown on the left below will typically first be translated to the LLVM code shown on the right, which uses alloca:

```c
int x = 0;
_%var_x = alloca i64
store i64 0, i64 *_%var_x
/* here */
```

Which of the following LLVM instructions, if it appeared at the point marked /* here */ in the LLVM code, would prevent alloca-promotion of_%var_x (i.e. replacing its uses with uids instead of pointer lookups)? There may be zero or more than one answer.

(a) %y = load i64*_%var_x
(b) store i64*_%var_x, i64**_%var_y
(c) %y = call i64*foo(i64*_%var_x)
(d) store i64_%var_y, i64*_%var_x
7. Register Allocation

(a) Consider the following straight-line LLVM code.

```llvm
%c = call i64 init()
%a = add i64 %c, %c
%b = add i64 %c, %c
%e = add i64 %a, %b
%d = add i64 %c, %e
ret i64 %d
```

Label the vertices of the following interference graph (where the solid lines denote interference edges) so that it corresponds to the code above. Each label %a–%e should be used once. (There may be more than one correct solution.)

(b) What is the minimal number of colors needed to color the graph?

(c) Briefly explain the purpose of coalescing nodes of the interference graph.
APPENDIX A: Typechecking Sum Types

Grammar for a simple expression-based language with sum types:

\[
\begin{align*}
\tau &::= \text{Types} \\
&\mid \text{int} \\
&\mid \tau_1 + \tau_2 \\

\ e &::= \text{Expressions} \\
&\mid x \quad \text{variables} \\
&\mid n \quad \text{integer constants} \\
&\mid \text{let } x = e_1 \text{ in } e_2 \quad \text{local lets} \\
&\mid \text{Left } e \quad \text{sum constructors} \\
&\mid \text{Right } e \\
&\mid \text{case } e \text{ of Left } x_1 \to e_1; \text{Right } x_2 \to e_2 \quad \text{case analysis} \\
&\mid (e)
\end{align*}
\]

\[
\begin{align*}
\Gamma &::= \text{Typechecking Contexts} \\
&\mid \cdot \\
&\mid x : \tau, \Gamma
\end{align*}
\]

Typechecking expressions is defined, in part, by the following inference rules. Recall that the notation 
\(x : \tau \in \Gamma\) means that \(x\) occurs in the context \(\Gamma\) at type \(\tau\) and \(x \not\in \Gamma\) means that \(x\) is not bound in \(\Gamma\).

\[
\begin{align*}
\Gamma &\vdash e : \tau \\
\hline
\hline
x : \tau &\in \Gamma \quad \text{VAR} \quad \Gamma \vdash n : \text{int} \\
\hline
x \not\in \Gamma \quad \Gamma \vdash e_1 : \tau_1 \quad x : \tau_1, \Gamma \vdash e_2 : \tau_2 \quad \text{LET} \\
\hline
\hline
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \\
\hline
\hline
\Gamma \vdash \text{Left } e : \tau_1 + \tau_2 \\
\hline
\hline
\Gamma \vdash \text{Right } e : \tau_1 + \tau_2
\end{align*}
\]