\[
\grammar{\text{Program}} ::= \langle \text{Block} \rangle \\
\quad | \langle \text{Block} \rangle \ ';$
\]
\[
\grammar{\text{Decls}} ::= \langle \text{DeclList} \rangle | \epsilon
\]
\[
\grammar{\text{DeclList}} ::= \langle \text{Decl} \rangle \\
\quad | \langle \text{Decl} \rangle \ ';$ \langle \text{DeclList} \rangle
\]
\[
\grammar{\text{Decl}} ::= '\text{VAR}' \ id \langle \text{OptType} \rangle \\
\quad | '\text{TYPE}' \ id \ ::= \langle \text{Type} \rangle
\]
\[
\grammar{\text{OptType}} ::= \epsilon \ | \ ::= \langle \text{Type} \rangle
\]
\[
\grammar{\text{ProcDecl}} ::= '\text{PROCEDURE}' \ id \ (C \langle \text{Formals} \rangle \ ') \langle \text{OptType} \rangle \ ::= \langle \text{Block} \rangle \\
\quad | '\text{PROCEDURE}' \ id \ (C \langle \text{Formals} \rangle \ ') \ ::= \langle \text{Block} \rangle
\]
\[
\grammar{\text{Formals}} ::= \langle \text{FormalList} \rangle | \epsilon
\]
\[
\grammar{\text{FormalList}} ::= \langle \text{Formal} \rangle \\
\quad | \langle \text{FormalList} \rangle \ ',' \langle \text{Formal} \rangle
\]
\[
\grammar{\text{Formal}} ::= \text{id} \ ::= \langle \text{Type} \rangle
\]
\[
\grammar{\text{Type}} ::= '\text{INTEGER}' \\
\quad | '\text{UNIT}' \\
\quad | \langle \text{SubrTy} \rangle \\
\quad | \langle \text{ArrayTy} \rangle \\
\quad | \langle \text{ProcTy} \rangle
\]
\[
\grammar{\text{SubrTy}} ::= [' \text{number} \ '\text{TO} \ '\text{number}' ]
\]
\[
\grammar{\text{ArrayTy}} ::= '\text{ARRAY}' \langle \text{SubrTy} \rangle \ '\text{OF}' \langle \text{Type} \rangle
\]
\[
\grammar{\text{ProcTy}} ::= '\text{PROCEDURE}' \ (C \langle \text{Formals} \rangle \ ') \langle \text{OptType} \rangle
\]
\[
\grammar{\text{Block}} ::= \langle \text{Decls} \rangle \ '\text{BEGIN}' \langle \text{Stmts} \rangle \ 'END'
\]
\[
\grammar{\text{Stmts}} ::= \langle \text{StmtList} \rangle \ | \epsilon
\]
\[
\grammar{\text{StmtList}} ::= \langle \text{Stmt} \rangle \\
\quad | \langle \text{StmtList} \rangle \ ';$ \langle \text{Stmt} \rangle
\]
\[
\grammar{\text{Stmt}} ::= \langle \text{Assignment} \rangle \\
\quad | \langle \text{Return} \rangle \\
\quad | \langle \text{Conditional} \rangle \\
\quad | \langle \text{Iteration} \rangle \\
\quad | \langle \text{Output} \rangle \\
\quad | \langle \text{Expr} \rangle
\]
\[
\grammar{\text{Assignment}} ::= \langle \text{Expr} \rangle \ ::= \langle \text{Expr} \rangle
\]
\[
\grammar{\text{Return}} ::= '\text{RETURN}' \langle \text{Expr} \rangle
\]
\[
\grammar{\text{Conditional}} ::= '\text{IF}' \langle \text{Expr} \rangle \ 'THEN' \langle \text{StmtList} \rangle \ 'ELSE' \langle \text{StmtList} \rangle \ 'END'
\]
\[
\grammar{\text{Iteration}} ::= '\text{WHILE}' \langle \text{Expr} \rangle \ 'DO' \langle \text{StmtList} \rangle \ 'END'
\]
\[
\grammar{\text{Output}} ::= '\text{PRINT}' \langle \text{Expr} \rangle
\]
\[
\grammar{\text{Expr}} ::= \langle \text{Operand} \rangle \\
\quad | \langle \text{Expr} \rangle \langle \text{Operator} \rangle \langle \text{Operand} \rangle
\]
\[
\grammar{\text{Operand}} ::= \text{number} \\
\quad | \text{id} \\
\quad | \langle \text{Operand} \rangle \ [' \langle \text{Expr} \rangle \ ']' \\
\quad | \langle \text{Operand} \rangle \ 'C' \langle \text{Actuals} \rangle \ ')'
\quad | 'C' \langle \text{Expr} \rangle \ ')
\]
\[
\grammar{\text{Operator}} ::= '+' | '>' | '==' | 'AND'
\]
\[
\grammar{\text{Actuals}} ::= \langle \text{ActualList} \rangle \ | \epsilon
\]
\[
\grammar{\text{ActualList}} ::= \langle \text{Expr} \rangle \\
\quad | \langle \text{Actuals} \rangle \ ',', \langle \text{Expr} \rangle
\]
Question 1  (7 Points)

In the table below you see pairs of types with a box in between. Write an X in the box if neither type is a subtype of the other, or draw a <: or :> (suitably) to indicate that one is a subtype of the other.

Use the same assumptions as in class, i.e., that (1) we are using an imperative language (updates are allowed), that (2) the type system enforces strong typing and (3) the type system permits any type to be a subtype of another if and only if doing so will not require dynamic checks.

(a) (3 Points) Fill in as indicated above:

\[
\begin{array}{c}
[5 \text{ TO } 10] & [0 \text{ TO } 10] \\
[5 \text{ TO } 10] & [3 \text{ TO } 7] \\
[5 \text{ TO } 10] & [5 \text{ TO } 7] \\
\text{ARRAY [0 \text{ TO } 10] OF [5 \text{ TO } 10]} & \text{ARRAY [0 \text{ TO } 10] OF [0 \text{ TO } 10]} \\
\text{ARRAY [0 \text{ TO } 10] OF [0 \text{ TO } 10]} & \text{ARRAY [5 \text{ TO } 10] OF [0 \text{ TO } 10]}
\end{array}
\]

(b) (4 Points) Continue filling in. For the following, assume that A is a supertype of B, and that the type C[X] is covariant in type parameter X.

\[
\begin{array}{c}
A & B \\
\text{C[A]} & \text{C[B]} \\
A \to A & A \to B \\
A \to A & B \to A \\
A \to A & B \to B \\
A \to B & B \to A
\end{array}
\]
Question 2  (9 Points)
Consider the following MYSTERY program. Assume that Mystery is configured so that the pro-
gram can execute without any errors.

1 VAR z : INTEGER;
2 PROCEDURE P(y : INTEGER) : INTEGER =
3 BEGIN
4 y := y + 1;
5 PRINT z;
6 RETURN y + 2
7 END;
8 PROCEDURE Q(x : INTEGER) : INTEGER =
9 BEGIN
10 PRINT x;
11 PRINT x
12 END
13 BEGIN
14 z := 0;
15 Q(P(z))
16 END

(a) (3 Points) What will the program print under by-value-result parameter passing? Explain.

(b) (3 Points) What will the program print under by-reference parameter passing? Explain.

(c) (3 Points) What will the program print under by-name parameter passing? Explain.
Question 3  (12 Points)
Answer the following four questions about programming language concepts.

(a) (3 Points) What is short-circuit evaluation? Explain with an example.

(b) (3 Points) What is widening conversion? Explain with an example.
(c) (3 Points) What is the difference between discriminated (tagged) union types and free (untagged) union types? Explain with an example.

(d) (3 Points) What is the difference between type inference and dynamic typing? Explain with an example.
Question 4  (8 Points)
Consider one of the following class interfaces (the two are equivalent; one is in Java, and one in Scala). The constructor definitions are omitted for brevity, as they play no role in this discussion.

// Java
class C<X> {
    int statusCode() { ... };
    X select(int z) { ... };
    C<X> copy() { ... };
}

// Scala
class C[X] {
    def statusCode(): Int = ...
    def select(x: Int): X = ...
    def copy(): C[X] = ...
}

Assume that we are defining such a class in a language with definition-site variance. We now try to determine the variance of type parameter X.

(a) (4 Points) Can we safely mark type variable X as covariant? Explain.

(b) (4 Points) Can we safely mark type variable X as contravariant? Explain.
Question 5  (7 Points)
Consider the following program in MYSTERY. Assume that the program is well-formed and executable, and uses by-value parameter passing, *static storage binding* for global variables, *stack-dynamic storage binding* for all other variables, and *static scoping*:

```
1  VAR  x :  INTEGER;
2  PROCEDURE  P(z :  INTEGER)  :  INTEGER  =
3      PROCEDURE  Q(w :  INTEGER)  :  INTEGER  =
4          BEGIN
5            RETURN  w + x
6          END;
7  PROCEDURE  R(x :  INTEGER, y :  INTEGER)  :  INTEGER  =
8      BEGIN
9          RETURN  Q(z)
10      END
11      BEGIN
12          RETURN  R(2, z)
13      END
14      BEGIN
15          x := 0;
16      PRINT  P(1);
17      PRINT  0
18      END
```

(a) (4 Points) What is the scope of the variables listed below? List the number of all lines during whose execution the variable is in scope. You can use range notation (e.g., “5–12”).

```
x (line 1)
z (line 2)
y (line 7)
```

(b) (3 Points) What is the difference between the *scope* and the *lifetime* of a variable? Use the code above as an example.
Question 6  (7 Points)
Consider the language $L_0$ whose syntax we define via the nonterminal $\langle expr \rangle$ in the following grammar:

\[
\begin{align*}
\langle expr \rangle & ::= \quad \text{nat} \\
& \quad | \langle ltv \rangle \\
& \quad | \langle expr \rangle \ '@' \langle expr \rangle \\
& \quad | \langle expr \rangle \ '+' \langle expr \rangle \\
& \quad | \ '[' \langle cont \rangle \\
\langle ltv \rangle & ::= \ 'U' \\
& \quad | \ 'D' \\
\langle cont \rangle & ::= \langle ltv \rangle \langle lock \rangle \\
& \quad | \langle expr \rangle \ ',', \langle cont \rangle \\
\langle lock \rangle & ::= \ 'L' \langle lock \rangle \\
& \quad | \ ']' \\
\end{align*}
\]

where $\text{nat}$ describes the natural numbers ($\mathbb{N}$).

(a) (4 Points) For each of the following token sequences, mark whether they are productions of the $L_0$ grammar:

<table>
<thead>
<tr>
<th>[U]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 1 , [ D L ] ]</td>
</tr>
<tr>
<td>[ 1 , 3 , L L ]</td>
</tr>
<tr>
<td>[ 3 , U L ]</td>
</tr>
</tbody>
</table>

(b) (3 Points) Assume that the operators ‘+’ and ‘@’ are left-associative, and that ‘@’ has a higher precedence than ‘+’. Draw the parse tree for the following expression: $1 + 2 \ @ 3 + 4$
Question 7  (12 Points)
  Consider the following program in Java (on the left) or Scala (on the right); both programs are equivalent. Assume that we run this program once.

```
1 class A {
2     void f() { }
3     void g(A a) { a.f(); }
4 }
5
6 class B extends A {
7     @Override
8     void f() { }
9 }
10
11 class C extends A {
12     @Override
13     void f() { h(); }
14     void h() { }
15 }
16
17 class D extends C {
18     @Override
19     void h() { }
20 }
21
22 var v : A = new B();
23 var z : A = new D();
24 v.f();
25 z.g(v);
26 v.g(z);
```

(a) (3 Points) What static type(s) is variable a (line 3) bound to? Explain.

(b) (3 Points) What dynamic type(s) is variable a (line 3) bound to? Explain.

(c) (6 Points) What methods (e.g., A.f, A.g) will this program call, and in which order?
Question 8  (15 Points)
Consider the language defined below:

\[
\langle \text{expr} \rangle ::= \text{num} \\
| \langle \text{pol} \rangle \\
| \langle \text{expr} \rangle ^* \langle \text{expr} \rangle \\
\langle \text{pol} \rangle ::= \text{ID} | \text{ZERO} | \text{NEG}
\]

where \text{num} describes the integers \((\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\})\).

To define the type system and the natural semantics, we use the following metavariables:

- \(n_1, n_2\) Integer values (from \text{num})
- \(p_1, p_2\) Productions of \langle \text{pol} \rangle
- \(e_1, e_2\) Productions of \langle \text{expr} \rangle
- \(\tau_1, \tau_2\) Types; must be either \text{Int} or \text{Pol}.

The Type System below assigns one of the two types \text{Int} or \text{Pol}:

\[
\frac{n_1 : \text{Int}}{n_1} \quad (Tn) \\
\frac{p_1 : \text{Pol}}{p_1} \quad (Tp) \\
\frac{e_1 : \tau_1}{e_1} \quad (T\tau_1) \\
\frac{e_2 : \text{Int}}{e_2} \quad (Tm1) \\
\frac{e_1 : \text{Int}}{e_1} \quad (Tm2)
\]

\[
\frac{e_1 ^* e_2 : \text{Int}}{e_1 ^* e_2} \quad (mul)
\]

The Natural Semantics are:

\[
\frac{\text{num}}{n_1 \Downarrow n_1} \quad (\text{num}) \\
\frac{\text{pol}}{p_1 \Downarrow p_1} \quad (\text{pol}) \\
\frac{\text{int}}{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2} \quad (\text{mul}) \\
\frac{\text{int}}{e_1 \Downarrow \text{ID} \quad e_2 \Downarrow n_2} \quad (\text{mpI}) \\
\frac{\text{int}}{e_1 \Downarrow \text{ZERO} \quad e_2 \Downarrow n_2} \quad (\text{mpZ}) \\
\frac{\text{int}}{e_1 \Downarrow \text{NEG} \quad e_2 \Downarrow n_2} \quad (\text{mpN})
\]

where \(n_1 \cdot n_2\) stands for arithmetic multiplication of \(n_1\) and \(n_2\).

(a) (4 Points) Are any parts of the language’s semantics undefined? Explain.

(b) (3 Points) What does the expression ‘\text{NEG} ^* 2 ^* 3’ evaluate to? Explain which rules you used to arrive at your conclusion.
(c) (6 Points) Extend the natural semantics such that the operator $\ast$ can combine two $Pol$ symbols, e.g., $ID \ast ZERO$. Ignore the type system for now. Your semantics should ensure that evaluating $(p_1 \ast p_2) \ast n_1$ gives the same result as evaluating $p_1 \ast (p_2 \ast n_1)$.

(d) (2 Points) Are any changes to the type system necessary to allow the operator $\ast$ to combine two $Pol$ symbols? If yes, describe the necessary changes on a high level. If no, explain your answer and name the applicable type rules.
Question 9 (12 Points)
Consider the following piece of Standard ML code:

```ml
datatype tree = N
  | B of tree (* left child *)
  | int (* value *)
  | tree (* right child *)
```

This binary tree stores int values in each B node.

In the following, you do not have to get the syntax exactly right, as long as it is unambiguous what you are doing. Add explanations whenever you are in doubt about your syntax.

(a) (4 Points) Write a function `contains : (tree, int) -> bool` that checks whether a tree contains the int parameter that is passed in. Assume that the tree is sorted so that for any B node b, the left child of b only contains values less than the value stored in b and the right child of b only contains values greater than the value stored in b.

(b) (5 Points) Write a function `map : (int -> int) -> tree -> tree` that works analogously to the list `map` function that we discussed in class. In other words, a call to `map` with parameter `f : int -> int` and parameter `tr : tree` should replace all values v in tr by f(v). Your function should otherwise leave the tree structure unchanged.

(c) (3 Points) Assume that you have `t : tree`. Call `map : (int -> int) -> tree -> tree` on t to increment all values in t by 2.
Question 10  (8 Points)
Answer the following question about abstract datatypes.

(a) (4 Points) What is an abstract datatype? Explain with a short code example in a language of your choice. Make sure to state which language you are using.

(b) (4 Points) We discussed two concepts that programming languages use for supporting abstract datatypes in the type system: subtyping and typeclasses. Give one difference between these two concepts that affects expressivity or reliability.
Question 11  (3 Points)
Which of these following three kinds of storage location binding can give rise to the Dangling Pointer Problem in a language with manual (explicit) memory management? Explain your answers.

(a) (1 Point) Variables with static memory binding.

(b) (1 Point) Variables with stack-dynamic memory binding.

(c) (1 Point) Variables with explicit heap-dynamic memory binding.