For the language described in Question **4.**, which **one** of the following describes the small-step rule(s) for substr(e, n_1, n_2)? Assume that a string s is formed using alphabets. That is, $s = c_1c_2 \ldots c_n$. (*Hint:* Your small-step rules should agree with the large-step rules. That is, multi-step of small-step rules should yield the same value that a large-step relation returns. Take a closer note of the substring indices.)

a)

$$\textsc{Substr}\ \frac{e \longrightarrow e'}{\text{substr}(e, n_1, n_2) \longrightarrow \text{substr}(e', n_1, n_2)}$$

b)

$$\textsc{Substr1}\ \frac{e \longrightarrow e'}{\text{substr}(e, n_1, n_2) \longrightarrow \text{substr}(e', n_1, n_2)}$$

$$\textsc{Substr2}\ \frac{}{\text{substr}(c_0c_1 \ldots c_{n-1}, n_1, n_2) \longrightarrow (c_{n_1} \ldots c_{(n_1+n_2-1)})}\ 0 \leq n_1 < n, 0 < n_2 \leq n$$

c)

$$\textsc{Substr}\ \frac{}{\text{substr}(c_1c_2 \ldots c_n, n_1, n_2) \longrightarrow (c_{n_1} \ldots c_{(n_1+n_2-1)})}\ n_1, n_2 > 0$$

d)

$$\textsc{Substr}\ \frac{e \longrightarrow c_1c_2 \ldots c_n}{\text{substr}(c_1c_2 \ldots c_n, n_1, n_2) \longrightarrow (c_{n_1} \ldots c_{(n_1+n_2-1)})}\ 0 \leq n_1 < n, 0 < n_2 \leq n$$

6. Suppose we extend the arithmetic language with division operation. That is, extend the grammar shown in Appendix A with division:

$$e ::= \ldots | e_1/e_2$$

The large-step rule for division is:

$$\textsc{Div}\ \frac{\langle e_1, \sigma\rangle \Downarrow \langle n_1, \sigma_1\rangle \quad \langle e_2, \sigma_1\rangle \Downarrow \langle n_2, \sigma_2\rangle}{\langle e_1/e_2, \sigma\rangle \Downarrow \langle n, \sigma_2\rangle}\ \text{where } n = n_1/n_2 \text{ and } n_2 \neq 0$$

Which **one** of the following is true?

a) 0/0 is syntactically not correct.

b) 0/0 is syntactically correct but the evaluation gets stuck for all stores.

c) 0/0 is syntactically correct and yields an answer when evaluated under some stores.

d) 0/0 is syntactically correct and yields NaN (not a number) when evaluated under any store.

Consider the small-step operational semantics of the language of arithmetic expressions (Appendix A). Assuming an initial store that maps all variables to zero, how many steps does the following program take to reach a final configuration?

$$x := (\text{bar} \times (\text{foo} + 3)); y + 2 \times x$$

a) 3

b) 9

c) 6

d) 10

2. Recall the store update and lookup definition (Appendix A). Let the store σ be $[x \mapsto 1, y \mapsto 2]$. What is the result of the store update followed by the lookup of $\sigma[x \mapsto 3](x)$?

a) 3

b) 1

c) Lookup fails as the intermediate update fails due to duplicate keys.

d) 1 or 3 depending on the order in which the keys are stored.

3. Suppose I remove the rules LMUL and RMUL from the small-step rules of the arithmetic expression language (Appendix A). Which **one** of the following definitely gets stuck during the evaluation with small-step rules? Note that you must pick the most precise answer.

a) $\langle n_1 \times n_2, \sigma \rangle$ gets stuck.

b) $\langle n_1 \times n_2 \times n_3, \sigma \rangle$ gets stuck.

c) $\langle x + n_1 \times n_2, \sigma \rangle$ gets stuck.

d) $\langle n_1 \times n_2 + n_3, \sigma \rangle$ gets stuck.

4. Consider a simple string language below. It defines concatenation and substring operati... described using the large-step relation.

$$\begin{array}{lll} c & \in & \text{alphabet} \\ s & \in & \text{String} \\ n & \in & \text{Nat} \\ e & ::= & s \mid e_1 \oplus e_2 \mid \mathsf{substr}(e, n, n) \end{array}$$

The large-step semantics are defined as the following.

$$\mathit{Concat}_{lrg} \frac{e_1 \Downarrow s_1 \quad e_2 \Downarrow s_2}{e_1 \oplus e_2 \Downarrow s_1 s_2}$$

$$\mathit{Substr}_{lrg} \frac{e \Downarrow s}{\mathsf{substr}(e, n_1, n_2) \Downarrow c_{n_1} \cdots c_{(n_1+n_2-1)}} \; 0 \le n_1 < n, 0 < n_2 \le n \text{ and } s = c_1 c_2 \cdots c_n$$

Intuitively, $\mathsf{substr}(e, n_1, n_2)$ evaluates e to a string s and returns the substring of length n_2 starting at index n_1 (0-indexed).

Recall that a multi-step relation is the reflexive, transitive closure of the small-step relation. Suppose we need to define a small-step relation for the language such that the definition of multi-step relation is equivalent to the large-step relation. Which **one** of the following describes the small-step rules for $e_1 \oplus e_2$, most precisely and completely?

a)

$$\textsc{Concat} \frac{e_1 \longrightarrow s_1 \quad e_2 \longrightarrow s_2}{e_1 \oplus e_2 \longrightarrow s_1 s_2}$$

b)

$$\textsc{Concat-1} \frac{e_1 \longrightarrow e_1'}{e_1 \oplus e_2 \longrightarrow e_1' \oplus e_2} \quad \textsc{Concat-2} \frac{e_2 \longrightarrow e_2'}{s_1 \oplus e_2 \longrightarrow s_1 \oplus e_2'} \quad \textsc{Concat} \frac{}{s_1 \oplus s_2 \longrightarrow s_1 s_2}$$

c)

$$\textsc{Concat} \frac{e_1 \longrightarrow e_1' \quad e_2 \longrightarrow e_2'}{e_1 \oplus e_2 \longrightarrow e_1' \oplus e_2'}$$

d) None of the above.

Consider the following OCaml program that uses references. You may use **utop** to check the answer.

```
let y = ref 1 in
let x = !y in
let y = 2 in
!x;;
```

What is the output?

a) 3

b) 1

c) 2

d) Error.

12. Consider the standard arithmetic language shown in Appendix A. Suppose we have only unsigned integers that can be represented using 5 bits. We introduce a type $\textbf{int}_5$ that is meant to represent the type of 5-bit width unsigned integers. The typing rules T-INT$_5$ and T-MUL are defined as follows:

$$\text{T-INT}_5 \frac{}{\Gamma \vdash n : \textbf{int}_5} \; 0 \leq n \leq 31$$

$$\text{T-MUL} \frac{\Gamma \vdash n_1 : \textbf{int}_5 \quad \Gamma \vdash n_2 : \textbf{int}_5}{\Gamma \vdash n_1 \times n_2 : \textbf{int}_5}$$

Which **one** of the following is well-typed?

a) 1×31.

b) 2×16.

c) Both (a) and (b).

d) None.

13. Suppose in the previous question, I modify the typing rule T-MUL as follows:

$$\text{T-MUL} \frac{\Gamma \vdash n_1 : \textbf{int}_5 \quad \Gamma \vdash n_2 : \textbf{int}_5}{\Gamma \vdash n_1 \times n_2 : \textbf{int}_5} \; 0 \leq n_1 \times n_2 \leq 31$$

Which **one** of the following is well-typed?

a) 1×31.

b) 2×16.

c) Both (a) and (b).

d) None.

7. Let us also give the typing rule to the division operation defined in the previous problem.

$$\text{T-DIV}\ \frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 / e_2 : \textbf{int}}$$

Is the typing judgment $[x \mapsto \textbf{int}] \vdash x/0 : \textbf{int}$ valid?

a) Yes, it holds for all values of x.

b) Yes, but only for non-zero values of x.

c) No, it does not hold for any value of x as the rule DIV requires the denominator to be a non-zero integer.

8. In the proof for progress (the first part of type soundness theorem), what is the set that we induct on?

a) e, the set of lambda calculus expressions.

b) $\vdash e : \tau$, the set of tuples (of expressions and types) related by typing judgments.

c) $e \to e'$, the set of tuples (of expressions) related by small-step rules.

d) Both (*b*) and (*c*).

9. Which evaluation strategy does the mainstream programming language "C" follows?

a) Call-by-Value

b) Call-by-Name

10. Consider the following OCaml program:

```
let y = 1 in
let x = y in
let y = 3 in
x;;
```

What is the output? You may use **utop** to check the answer.

a) 3

b) 1

14. Is $x : \textsf{bool} \vdash \lambda x : \textsf{int}.\, x + 10 : \textsf{int} \to \textsf{int}$ well-typed?

a) Yes.

b) No.

15. Consider the small-step semantics of the call-by-value lambda calculus with references and let expressions, expressed using evaluation contexts (see Appendix D). Consider the following derivation of a small step.

$$\textsc{Context}\ \frac{\textsc{Alloc}\ \dfrac{}{\langle \textsf{ref}\ 36, \sigma\rangle \longrightarrow \langle \ell, \sigma[\ell \mapsto 36]\rangle}}{\langle \textsf{let}\ x = !((\lambda y.\, y)\ \textsf{ref}\ 36)\ \textsf{in}\ x + 6, \sigma\rangle \longrightarrow \langle \textsf{let}\ x = !((\lambda y.\, y)\ \ell)\ \textsf{in}\ x + 6, \sigma[\ell \mapsto 36]\rangle}$$

What is the evaluation context E used in the instantiation of the rule CONTEXT above?

a) $E = [\cdot]$

b) $E = \textsf{let}\ x = [\cdot]\ \textsf{in}\ x + 6$

c) $E = \textsf{let}\ x = !((\lambda y.\, y)\ [\cdot])\ \textsf{in}\ x + 6$

d) $E = \langle \textsf{let}\ x = !((\lambda y.\, y)\ [\cdot])\ \textsf{in}\ x + 6, \sigma\rangle$

16. Consider the following lambda-calculus program with references (see Appendix D). Assume CBV semantics.

$$\begin{array}{l}\textsf{let}\ y = \textsf{ref}\ \lambda a.\, a\ \textsf{in}\\ \textsf{let}\ z = (y := \lambda a.\, !y\ 32)\ \textsf{in}\\ !y\ 6\end{array}$$

Which **one** of the following statements is true? (Hint: You may have you seen similar—but not the same—question before.)

a) This program will terminate with value $\lambda a.\, a$.

b) This program will terminate with value $\lambda a.\, !\ell\ 32$ such that the final store is $[\ell \mapsto \lambda a.\, !\ell\ 32]$.

c) This program will diverge (i.e, not terminate).

d) This program will get stuck.

et us modify the second line of the previous program as follows:

$$\begin{array}{l}\text{let } y = \text{ref } \lambda x.\, x \text{ in} \\ \text{let } z = (y := !y) \text{ in} \\ !y\ 6\end{array}$$

Which *one* of the following statements is true?

a) This program will terminate with value $\lambda a.\, a$.

b) This program will terminate with value 6.

c) This program will diverge (not-terminate).

d) This program will get stuck.

18. Recall the syntax of abstraction in lambda calculus. Consider the term $\lambda x.x\ y\ \lambda y.y$. Select all that apply:

a) The leftmost occurrence of y is bound to λx.

b) The rightmost occurrence of y in the abstraction is bound to λy.

c) The leftmost occurrence of y is free.

d) The leftmost occurrence of x is bound to λx.

19. What is the result of the substitution $(\lambda x.\, a\ x)\ \{x/a\}$ equivalent to?

a) Substitution fails.

b) $(\lambda x.\, a\ x)\ \{y/a\}$.

c) $(\lambda x.\, x\ x)\ \{x/a\}$.

d) $(\lambda b.\, a\ b)\ \{x/a\}$.

20. Consider the term $(\lambda x.x)((\lambda y.y)\ 5)$. Recall that a redex is a sub-term that will be evaluated immediately. What is the redex in the above term? Select all that apply.

a) $((\lambda y.y)\ 5)$ under call-by-value evaluation.

b) $(\lambda x.x)((\lambda y.y)\ 5)$ under call-by-name evaluation.

c) $((\lambda y.y)\ 5)$ under both call-by-name and call-by-value evaluation.

d) There is no redex.

21. An evaluation strategy is more optimizing than the other if the former leads to a normal form in a fewer number of steps.

Assume that a lambda calculus is extended with integers and a '+' operator. The small-step semantics of '+' use LADD, RADD AND ADD RULES FROM THE ARITHMETIC EXPRESSION LANGUAGE (APPENDIX A). COMPARE CBN AND CBV EVALUATION STRATEGIES FOR EVALUATING THE TERM:

$$(\lambda x.x + x)((\lambda y.y)\ 5)$$

A) CBN IS MORE OPTIMIZING THAN CBV.

B) CBV IS MORE OPTIMIZING THAN CBN.

C) BOTH CBN AND CBV RESULTS IN SAME NUMBER OF STEPS.

D) THE NUMBER OF STEPS IN CBN AND CBV ARE INCOMPARABLE.

22. An evaluation strategy is more optimizing than the other if the former leads to a normal form in a fewer number of steps. Compare CBN and CBV evaluation strategies for evaluating the term: $(\lambda x.x)\ \Omega$ where is $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$. Which of the following is true:

a) CBN is more optimizing than CBV.

b) CBV is more optimizing than CBN.

c) Both CBN and CBV are non-terminating and thus result in the same number of steps.

d) The number of steps in CBN and CBV are incomparable.

23. What should be the type of a if $[a \mapsto ?] \vdash (\lambda x{:}\textsf{unit} \to \textsf{int}.\ x\ ())\ a{:}\textsf{int}$ holds?

a) $a \mapsto \textsf{int}$

b) $a \mapsto \textsf{int} \to \textsf{unit}$

c) $a \mapsto \textsf{unit} \to \textsf{int}$

d) $a \mapsto \textsf{int} \to \textsf{int}$

24. Which one of the following expressions generates the following constraint set? (See Appendix G)

$$\{X \to Z = (\textsf{int} \times \textsf{int}) \to Y, X = \textsf{int} \to Z\}$$

a) $(\lambda x{:}X.\ (\#1\ x))\ (3,2)$

b) $(\lambda x{:}X.\ (x\ 10))\ (3,2)$

c) $(\lambda x{:}X.\ x)\ (\#1\ (3,2))$

d) $(\lambda x{:}X.\ x)\ (3,2)$

Name:

Consider the following Rust code.

```
1  fn test(s:String)->String{
2    return s;
3  }
4
5  fn main(){
6    let s_main = String::from("hello");
7    test(s_main);
8    println!("{}", s_main);
9  }
10
```

Which **one** of the following is true?

a) The above code does not compile due to a syntax error in line #7.

b) The above code compiles but throws a runtime error that the lifetime of the pointer to `s_main` has ended.

c) The above code does not compile as `s_main` is not of the type `str`.

d) The above code does not compile as the value "hello" moved to the argument `s` in line #7 is not moved back to `s_main`.