PoPL Spring 2026, Final Review 2

EN.601.426/626 Principles of Programming Languages

Name: _____

1. **Short Answers.**

1.1. **Abstract Syntax Trees.** Consider the following expression language:

e	::=	x	`type Expr = Var of string`
	|	i	`| Int of int`
	|	$e + e$	`| Add of Expr * Expr`
	|	$e(e)$	`| App of Expr * Expr`
	|	`If` e `Then` e `Else` e	`| If of Expr * Expr * Expr`

Please draw the abstract syntax tree of the following expression:

```
If ok Then f(x + 1) Else g(0)
```

1.2. **Record Subtyping.** Let there be two record types, `Person` and `Named`:

$$\begin{aligned} \texttt{Person} &= \{\texttt{name} : \texttt{String}, \texttt{age} : \texttt{Int}\} \\ \texttt{Named} &= \{\texttt{name} : \texttt{String}\} \end{aligned}$$

What is the relation between them in terms of subtype, supertype, or neither?

$$(\texttt{Named} \to \texttt{Int}) \to \texttt{Bool} \qquad (\texttt{Person} \to \texttt{Int}) \to \texttt{Bool}$$

1.3. **Lambda Calculus and Church Integers.** Recall that Church numerals encode natural numbers as functions:

$$\begin{array}{rcl} 0 & \triangleq & \lambda f.\lambda x.x \\ 1 & \triangleq & \lambda f.\lambda x.f\ x \\ 2 & \triangleq & \lambda f.\lambda x.f\ (f\ x) \\ \text{add} & \triangleq & \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x) \end{array}$$

Please desugar the following expression and reduce it to normal form:

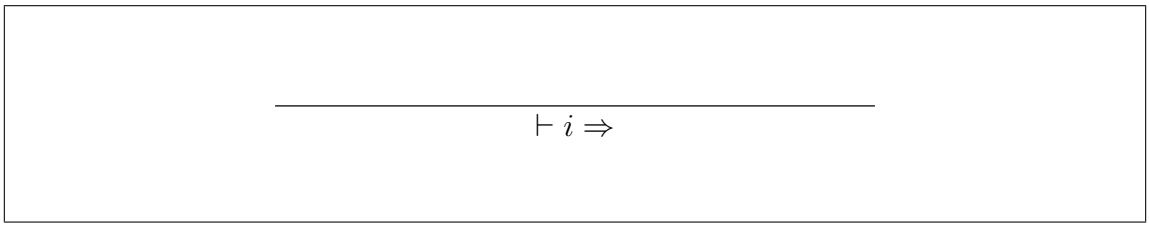
$$\begin{array}{rl} & \texttt{add}\ 1\ 2 \\ \triangleq & \\ \rightarrow_\beta & \end{array}$$

2. **Closure F$^\flat$.** In this problem, we are going to consider a variant of the F$^\flat$ language, which we will call CF$^\flat$. In this language, we will never use the substitution operation to replace variables with their values. Instead, we will use closures to capture the environment.

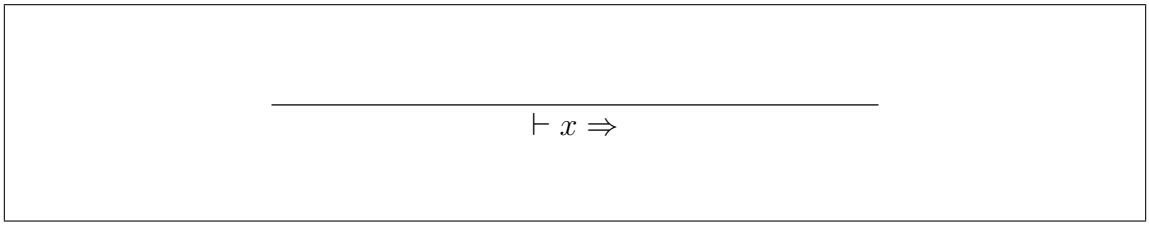
$$\begin{array}{lcl} e & ::= & x \mid i \mid \texttt{Fun}\ x\ \texttt{->}\ e \mid e\ e \mid \texttt{Let}\ x = e\ \texttt{In}\ e \\ v & ::= & i \mid \langle \texttt{Fun}\ x\ \texttt{->}\ e, \rho \rangle \\ \rho & = & \{x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots\} \end{array}$$

The operational semantics of CF$^\flat$ is centered around the judgement denoted as $\rho \vdash e \Rightarrow v$, which means that under the environment ρ, the expression e evaluates to the value v. Below, let's work through the operational semantics of CF$^\flat$ by writing the inference rules one by one.

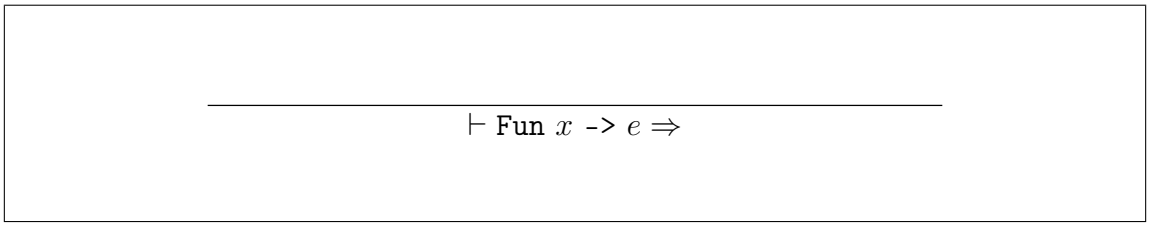
2.1. When evaluating an integer i, we get i itself as the value.



2.2. When evaluating a variable x, we look it up in the environment ρ.



2.3. When evaluating a function definition, we create a closure that captures the current environment.



2.4. When evaluating a function application, we demand that the LHS is evaluated to a closure, then we evaluate the argument expression to get its value, and finally we evaluate the body of the function in the extended environment.

$$\frac{}{\vdash e_1\ e_2 \Rightarrow}$$

2.5. When evaluating a let expression, we first evaluate the bound expression to get its value, then we extend the environment with this new binding and evaluate the body of the let expression in the extended environment.

$$\frac{}{\vdash \texttt{Let}\ x = e_1\ \texttt{In}\ e_2 \Rightarrow}$$

2.6. Based on the rules you have formalized above, please evaluate the following expression and draw the proof tree for it.

$$\frac{}{\vdash \texttt{Let x = 1 In (Fun y -> y + x) 3} \Rightarrow}$$

3. **Floating Points and NaNs.** Consider extending our F♭ programming language with floating point numbers, giving us F♭F. We will use a variant of the IEEE 754 standard for floating point representation, which includes special values such as NaN (Not a Number). We also introduce multiply and division operations.

$$\begin{array}{rcl} f & ::= & \text{floating point literals (e.g., 1.0, 3.14, } -5.21) \mid \texttt{NaN} \\ e & ::= & (\ldots \text{F}^\flat \text{ Expressions} \ldots) \mid f \mid e \texttt{ * } e \mid e \texttt{ / } e \end{array}$$

Syntactically, the floating point literals must include a decimal point to distinguish them from integers. For example, `1` is an integer, while `1.0` is a floating point number.

Floating-point numbers behave rather peculiarly, with the rules laid out below. Please formalize these rules (3.1–3.6) in the form of evaluation inference rules for the F♭F language. Note that for some cases you might need more than one inference rule. Please aim to be as general as possible, that is, when we are handling the operator of =, you should handle all cases of the operands.

3.1. Suppose you have two normal floating point numbers, they can multiply normally and produce a normal floating point number.

3.2. If you have a divide-by-zero, that is, the denominator is evaluated to 0.0, then the result is `NaN`. Other divide cases produce normal floating point numbers.

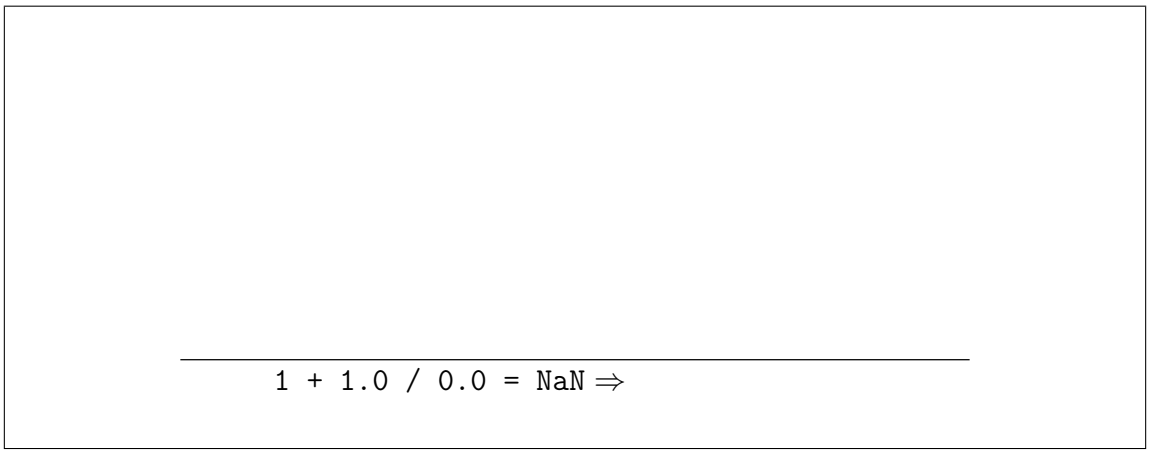
3.3. Any arithmetic operation (please abbreviate it to a general binary operation denoted as $\otimes$) involving `NaN` also results in `NaN`, when the other operand is also evaluated to a floating-point value or `NaN`.

3.4. During equality comparison, `NaN` is not equal to any value, including itself.

3.5. During inequality comparison, `NaN` is not equal to any value, including itself.

3.6. An expression that evaluates to an integer can be implicitly coerced to a floating point number by adding a decimal point.

3.7. Based on the rules you have formalized above, please evaluate the following expression and draw the proof tree for it.



3.8. Let us further extend the language into TF$^{\flat}$F, which includes a type system for floating point numbers. Our type system is as follows:

$$t \ ::= \ \texttt{Int} \mid \texttt{Bool} \mid \texttt{Float} \mid t \to t$$

We are going to go one step further to say `Int <: Float`, similar to the mathematical fact that $\mathbb{Z} \subseteq \mathbb{R}$, integers are a subset of real numbers. With this subtyping relation, we will change the original typing rules for addition and subtraction to accept only operands of `Float`s. Please make the type judgement of the following expression and draw the proof tree for it.