Homework 4

The purpose of this assignment is to get practice with variable binding and scoping, and to build an interpreter for something that begins to feel more like a programming language.

Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expressing computation currently unfamilar to you.

For submission instructions and the due date, please see the `README.md` file.

Problem 1 Variable Binding and Scoping (16 Points)

Consider the following grammar describing the abstract syntax of the simple expression language with constant declarations that we considered in class:

$$
\begin{array}{rr}
n \in \mathit{Num} & \text{numbers} \\
x \in \mathit{Var} & \text{variables} \\
e \in \mathit{Expr} ::= n \mid x \mid e_1\, \mathit{bop}\, e_2 \mid \textbf{const}\ x = e_d; e_b & \text{expressions} \\
\mathit{bop} \in \mathit{Bop} ::= \texttt{+} \mid \texttt{*} & \text{binary operators}
\end{array}
$$

For each of the expressions given below do the following:

(a) Overline the defining variable occurrences and draw an arrow between each bound using occurrence of a variable x and the corresponding defining occurrence of x.

(b) Give the set of free variables of the expression.

(c) Give the AST of the expression in tuple notation. Assume that the different types of expressions are assigned variant numbers in the order in which they appear in the grammar.

(d) Evaluate the expression in the environment $env = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ using the evaluation function *eval* defined in class.

Example: $e = \textbf{const}\ x = 5 + 6; \textbf{const}\ y = x + 0; x * (z + y)$

(a) $\textbf{const}\ \overline{x} = 5 + 6; \textbf{const}\ \overline{y} = x + 0; x * (z + y)$

(b) $fv(e) = \{z\}$

(c) $\langle \underline{4}, x, \langle \underline{3}, \langle \underline{1}, 5\rangle, \langle \underline{1}\rangle, \langle \underline{1}, 6\rangle\rangle,$
$\langle \underline{4}, y, \langle \underline{3}, \langle \underline{2}, x\rangle, \langle \underline{1}\rangle, \langle \underline{1}, 0\rangle\rangle,$
$\langle \underline{3}, \langle \underline{2}, x\rangle, \langle \underline{2}\rangle, \langle \underline{3}, \langle \underline{2}, z\rangle, \langle \underline{1}\rangle, \langle \underline{2}, y\rangle\rangle\rangle\rangle$

(d) $\mathit{eval}(env, e) = 154$

In part (a), instead of arrows, you can also use indices to indicate which using occurrence of a bound variable refers to which defining occurrence of that variable. For example, a valid answer for part (a) and the expression

$$\textbf{const}\ x = (\textbf{const}\ x = x; x); \textbf{const}\ x = x; x$$

would be:

$$\textbf{const}\ \overline{x}_1 = (\textbf{const}\ \overline{x}_2 = x; x_2); \textbf{const}\ \overline{x}_3 = x_1; x_3$$

Before you start working on your answers for each expression, you may want to draw the AST representation of the expression as a tree. Though, you do not need to include the drawn tree in your answer.

Expressions:

(i) $e_1 = x + 4$

(ii) $e_2 = \textbf{const}\ x = 2; y * x$

(iii) $e_3 = \textbf{const}\ z = z; \textbf{const}\ z = z; z$

(iv) $e_4 = \textbf{const}\ x = (\textbf{const}\ x = 3; z + x); z + x$

Problem 2 JakartaScript: Variable Binding and Type Conversions (24 Points)

In this exercise, we will implement an interpreter with support for variable bindings and automatic type conversion between numbers, Booleans, and strings in our JavaScript subset. JavaScript has a distinguished **`undefined`** value that we will also consider. This version of JakartaScript is much like the LET language in Section 3.2 of Friedman and Wand.

The syntax of JakartaScript for this assignment is given in Figure 1. Note that the grammar specifies the abstract syntax using notation borrowed from the concrete syntax.

The concrete syntax accepted by the parser is slightly less flexible than the abstract syntax in order to match the syntactic structure of JavaScript. In particular, all **`const`** bindings must be at the top-level. For example,

```
1 + (const x = 2; x)
```

is not allowed. The reason is that JavaScript layers a language of statements on top of its language of expressions, and the **`const`** binding is considered a statement. A program is a statement *st* as given in Figure 2. A statement is either a **`const`** binding, an expression, an empty statement (i.e., `;`), a grouping of statements (i.e., `{`*st*`}`), or a statement sequence (i.e., $st_1\ st_2$). Expressions are as in Figure 1 except **`const`** binding expressions are removed, and we have a way to parenthesize expressions.

An abstract syntax tree representation is provided for you in `ast.scala`. We also provide a parser and main driver for testing. The correspondence between the concrete syntax and the abstract syntax representation is shown in Figure 3.

$$
\begin{array}{rll}
n \in \mathit{Num} & & \text{numbers (double)} \\
s \in \mathit{Str} & & \text{strings} \\
x \in \mathit{Var} & & \text{variables} \\
b \in \mathit{Bool} & ::= \texttt{true} \mid \texttt{false} & \text{Booleans} \\
v \in \mathit{Val} & ::= \texttt{undefined} \mid n \mid b \mid s & \text{values} \\
e \in \mathit{Expr} & ::= x \mid v \mid \mathit{uop}\ e \mid e_1\ \mathit{bop}\ e_2 \mid & \text{expressions} \\
 & \quad e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \mid \texttt{const}\ x = e_d; e_b \mid \texttt{console.log}(e) & \\
\mathit{uop} \in \mathit{Uop} & ::= \texttt{-} \mid \texttt{!} & \text{unary operators} \\
\mathit{bop} \in \mathit{Bop} & ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{===} \mid \texttt{!==} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{,} & \text{binary operators}
\end{array}
$$

Figure 1: Abstract syntax

$$
\begin{array}{rll}
e & ::= \cdots \mid \cancel{\texttt{const}\ x = e_1; e_2} \mid (e) & \text{expressions} \\
st & ::= \texttt{const}\ x = e_1\texttt{;} \mid e\texttt{;} \mid \ \texttt{;} \mid \{st\} \mid st_1\ st_2 & \text{statements}
\end{array}
$$

Figure 2: Concrete syntax

To make the project simpler, we also deviate slightly with respect to scope. Whereas JavaScript considers all **const** bindings to be in the same scope, our JakartaScript bindings each introduce their own scope. In particular, for the binding **const** $x = e_d; e_b$, the scope of variable x is the expression e_b.

Statement sequencing and expression sequencing are right-associative. All other binary operator expressions are left-associative. Precedence of the operators follow JavaScript.

The semantics are defined by the corresponding JavaScript program. We also have a system function **console.log** for printing out values to the console. This function always returns the value **undefined**. Its implementation is provided for you.

(a) First, reconsider the JakartaScript expressions that you experimented with in Problem 2 of Homework 3. Write some additional JakartaScript programs that use the new constructs shown in Figure 1 and execute them as JavaScript programs. This step will inform you how you will implement your interpreter and will serve as tests for your interpreter. You do **not** need to submit these programs as part of your solution. However, we strongly suggest to use them (as well as your programs from Homework 3) as additional test cases for your implementation in part (b).

(b) Then, implement

```
def eval(env: Env, e: Expr): Val
```

that evaluates a JakartaScript expression `e` in a value environment `env` to a value. A value is a number n, a Boolean b, a string s, or **undefined**. In Scala, the set of all values is represented by the type `Val`.

```
enum Expr extends Positional:
  // Literals
  case Num(n: Double)   // <~ n
  case Bool(b: Boolean) // <~ b
  case Str(s: String)   // <~ "s"
  case Undefined        // <~ undefined
  // Variables
  case Var(x: String) // <~ x
  // Unary and Binary Operators
  case UnOp(uop: Uop, e1: Expr) // <~ uop e1
  case BinOp(bop: Bop, e1: Expr, e2: Expr) // <~ e1 bop e2
  // Constant declarations
  case ConstDecl(x: String, ed: Expr, eb: Expr) // <~ const x = ed; eb
  // Conditionals
  case If(e1: Expr, e2: Expr, e3: Expr) // <~ e1 ? e2 : e3
  // Printing
  class Print(e: Expr) // <~ console.log(e)

// Values
type Val = Num | Boolean | Str | Undefined.type

// Unary operators
enum Uop:
  case UMinus, Not // <~ -, !

// Binary operators
enum Bop:
  case Plus, Minus, Times, Div // <~ + - * /
  case Eq, Ne, Lt, Le, Gt, Ge  // <~ === !== < <= > >=
  case And, Or // <~ && ||
  case Seq // <~ , ;
```

Figure 3: The abstract syntax of JakartaScript as represented in Scala. After each **case**, we show the corresponding JavaScript expression represented by that case.

Before implementing the function `eval` itself, first implement three helper functions for converting values to numbers, Booleans, and strings.

```
def toNum(v: Val): Double
def toBool(v: Val): Boolean
def toStr(v: Val): String
```