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.

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:

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

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

Expressions:

(i) $e_1 = x + 2$

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

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

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

Example: $e = \textbf{const}\ x = 2 * 3; \textbf{const}\ y = x + 5; x * (z + y)$

(a) $\textbf{const}\ \overline{x} = 2 * 3; \textbf{const}\ \overline{y} = x + 5; x * (z + y)$

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

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

Figure 1: Abstract syntax

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

(d) $\mathit{eval}(\mathit{env}, e) = 72$

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.

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

In this exercise, we will implement an interpreter with automatic type conversion for the JavaScript subset with numbers, booleans, strings, and variable binding. 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,

$$e ::= \cdots \mid \cancel{\textbf{const } x = e_1; e_2} \mid \texttt{(}e\texttt{)} \qquad \text{expressions}$$
$$st ::= \textbf{const } x = e_1\texttt{;} \mid e\texttt{;} \mid \texttt{;} \mid \texttt{\{}st\texttt{\}} \mid st_1\ st_2 \qquad \text{statements}$$

Figure 2: Concrete syntax

```
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.

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.

(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`**. It will be useful to 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
```

```
sealed abstract class Expr extends Positional
sealed abstract class Val extends Expr
case class Num(n: Double) extends Val
Num(n)   n
case class Bool(b: Boolean) extends Val
Bool(b)   b
case class Str(s: String) extends Val
Str(s)  s
case object Undefined extends Val
Undefined   undefined
case class Var(x: String) extends Expr
Var(x)   x
case class ConstDecl(x: String, ed: Expr, eb: Expr) extends Expr
ConstDecl(x,ed,eb)   const x = ed; eb
case class UnOp(uop: Uop, e: Expr) extends Expr
UnOp(uop,e)   uop e
case class BinOp(bop: Bop, e1: Expr, e2: Expr) extends Expr
BinOp(bop,e1,e2)   e1 bop e2
sealed abstract class Uop
case object UMinus extends Uop  -
case object Not extends Uop  !
sealed abstract class Bop
case object Plus extends Bop  +
case object Minus extends Bop  -
case object Times extends Bop  *
case object Div extends Bop  /
case object Eq extends Bop  ===
case object Ne extends Bop  !==
case object Lt extends Bop  <
case object Le extends Bop  <=
case object Gt extends Bop  >
case object Ge extends Bop  >=
case object And extends Bop  &&
case object Or extends Bop  ||
case object Seq extends Bop  , ;
case class If(e1: Expr, e2: Expr, e3: Expr) extends Expr
If(e1,e2,e3)   e1 ? e2 : e3
case class Print(e: Expr) extends Expr
Print(e)   console.log(e)
```

Figure 3: Representing in Scala the abstract syntax of JakartaScript. After each **case class** or **case** object, we show the correspondence between the representation and the concrete syntax.