Project: Interpreter

Background

For this project, you will write an *interpreter* for a small fragment of JavaScript. To write an interpreter, you need a *parser* to turn JavaScript's *concrete syntax* into an *abstract syntax tree* (as explained in class). However, you don't need to write the parser yourself. We have provided it for you.

Concrete Syntax

The following grammar describes the concrete syntax of the fragment of JavaScript that you will be working with.

Numbers	n ::= …	
Variables	x ::= …	
Expressions	e ::= n	numeric constant
	| true	boolean value true
	| false	boolean value false
	| x	variable reference
	| e_1 + e_2	addition
	| e_1 - e_2	subtraction
	| e_1 * e_2	multiplication
	| e_1 / e_2	division
	| e_1 && e_2	logical and
	| e_1 || e_2	logical or
	| e_1 < e_2	less than
	| e_1 > e_2	greater than
	| e_1 === e_2	equal to
Statements	s ::= let x = e;	variable declaration
	| x = e;	assignment
	| if (e) b_1 else b_2	conditional
	| while (e) b	loop
	| print(e);	display to console
Blocks	b ::= { s_1 … s_n }	
Programs	p ::= s_1 … s_n	

Parser

We have provided two parsing functions. The function `parser.parseExpression` parses an expression (*e*) and the function `parser.parseProgram` parses a program (*p*). The following are their function signatures:

```
parseExpression(str: string): Result<Expr>
parseProgram(str: string): Result<Stmt[]>
```

The type definitions are as follows:

```
type Result<T> = { kind: 'ok', value: T } | { kind: 'error', message: string };

type Binop = '+' | '-' | '*' | '/' | '&&' | '||' | '>' | '<' | '===';

type Expr = { kind: 'boolean', value: boolean }
          | { kind: 'number', value: number }
          | { kind: 'variable', name: string }
          | { kind: 'operator', op: Binop, e1: Expr, e2: Expr };

type Stmt = { kind: 'let', name: string, expression: Expr }
          | { kind: 'assignment', name: string, expression: Expr }
          | { kind: 'if', test: Expr, truePart: Stmt[], falsePart: Stmt[] }
          | { kind: 'while', test: Expr, body: Stmt[] }
          | { kind: 'print', expression: Expr };
```

Here is an example of what `parseExpression` returns given **`"1"`** as the argument:

```
> parser.parseExpression("1")
{
  value: {
    kind: "number",
    value: 1
  },
  kind: "ok"
}
```

Programming Task

Your task is to implement the following functions:

```
// Given a state object and an AST of an expression as arguments,
// interpExpression returns the result of the expression (number or boolean)
interpExpression(state: State, e: Expr): number | boolean
// The State type is explained further down the document.

// Given a state object and an AST of a statement,
```

```
// interpStatement updates the state object and returns nothing
interpStatement(state: State, p: Stmt): void

// Given the AST of a program,
// interpProgram returns the final state of the program
interpProgram(p: Stmt[]): State
```

The `State` type is defined below:

```
type State = { [key: string]: number | boolean }
```

As explained on the previous project specifications, this notation indicates that a State object would have variable number of properties with values of type number or boolean.

Note that the inputs of these functions are abstract syntax trees, *not concrete syntax*. Therefore, you can run your code by using the parser, or by directly constructing ASTs by hand. E.g.:

```
> interpProgram(parser.parseProgram("let x = 10; x = x * 2;").value)
{ x: 20 }
> interpProgram([
    { kind: "let", name: "x", expression: { kind: "number", value: 10 } },
    { kind: "assignment", name: "x",
      expression: {
        kind: "operator", op: "*",  e1: { kind: "variable", name: "x" },
        e2: { kind: "number", value: 2 } } } ]);
{ x: 20 }
```

Suggested Approach

We suggest taking the following approach:

1. Implement `interpExpression`, following the template shown in class. You can use an empty object (`{ }`) for the state if you do not have any variables, or you can set the values of variable by hand. For example

   ```
   test("multiplication with a variable", function() {
     let r = interpExpression({ x: 10 }, parser.parseExpression("x * 2").value);
     assert(r === 20);
   });
   ```

2. Implement `interpStatement` and `interpProgram`, following the template shown in class. You should be able to test that assignment updates variables. For example:

   ```
   test("assignment", function() {
     let st = interpProgram(parser.parseProgram("let x = 10; x = 20;").value);
     assert(st.x === 20);
   });
   ```

3. Finally, test your interpreter, you should try writing some simple programs. For example, you should be able to write an iterative factorial or fibonacci with your interpreter.