Project 7: Interpreter Due Wed, 20 April 2022

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). 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
                    | e1 + e2           addition
                    | e1 - e2           subtraction
                    | e1 * e2           multiplication
                    | e1 / e2           division
                    | e1 && e2          logical and
                    | e1 || e2          logical or
                    | e1 < e2           less than
                    | e1 > e2           greater than
                    | e1 === e2         equal to

Statements   s ::= let x = e;           variable declaration
                    | x = e;            assignment
                    | if (e) b1 else b2 conditional
                    | while (e) b       loop
                    | print(e);         display to console

Blocks       b ::= { s1 … sn }

Programs     p ::= s1 … sn
```

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 defined further below.

// 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 as follows:

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

This notation indicates that a State object has a variable number of properties with values of type number or boolean (representing values of variables that are in scope).

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. For example:

```
> 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 }
```

Error Handling

An interpreter can generally not continue meaningfully after an error (as opposed to compilers). Thus, if you find an error, you should abort the program, with an informative error message. You need to do a number of checks (e.g., correct typing, and missing or duplicate declarations). You may assume that an AST has the right fields and types. Do not assume other checks, even if done by the parser, as your functions can be tested with ASTs that don't come from the parser.

Variable Scoping

A block starts a new inner scope. A variable declared in a block will shadow an outer declaration (any variable use will refer to the inner declaration). On exiting a scope, variables declared there are no longer accessible (since we don't have closures). Thus, they should not be in the global state at the end. The nesting of block scopes corresponds to a stack, which you can implement as a linked list, by adding to your State object a link to an outer scope. Since the link is just another property, this allows all functions to keep their signatures. To ensure the link name does not clash with a program variable, use a property name that is not an identifier (JavaScript allows this). The global state cannot have extra properties, but does not need a link, as the last state on the list.

Suggested 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 variables 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 with some simple programs. For example, you should be able to interpret an iterative factorial or Fibonacci sequence computation.