Programming Challenge

Implement an interpreter for the programming language called **SCREAM**.

SCREAM is a loosely typed and interpreted language. It's named that because in scream everything is uppercase. The language is literally shouting! The program is interpreted and executed one line after another. Just like python has a *Read Evaluate Print Loop*, SCREAM too is a *REPL*'ed language.

Time Limit: 2 days.
Submission: Zip the source code and the executable artifact with build and run steps.
Choice of programming language: Any.

Expectations

- Implement an interpreter which takes a SCREAM program as a file and executes it.
- Your program should be able to parse the lines of the program correctly.
- Use appropriate data structures to store the state of the program.
- Your program should print appropriate error and halt in case of an error.
- You should use the test SCREAM programs provided and use it to verify the correctness of the interpreter.
- The interpreter should be able to run as many tests as possible. The more the coverage the higher the score.
- You shall be awarded points for code cleanliness, code structure, descriptive and self explanatory variable and function names, and appropriate comments.
- We shall run tests on our end to verify the correctness.
- Preferred way would be to implement one feature at a time. In case some features are not implemented, it's acceptable till the implemented features work correctly.
- Evaluation of the code will be done in the following order of priority:
    - Successful compilation of the interpreter in your language of choice.
    - Number of language features implemented **correctly**.
    - Number of tests passing.
    - Code hygiene.
- Your interpreter program should take only one input, a file path name of a SCREAM program. It should start executing the SCREAM program immediately.
- **You are allowed to make appropriate assumptions when something is not explicitly stated**.

Language specification

The following sections describe the features of SCREAM and how they are suppose to behave.

Expressions

Expressions evaluate to a value of a particular type. Since SCREAM is a loosely typed language, variables don't have fixed types. Also there are only three fundamental types available:

1. Numbers: These can be integers or floating point numbers and can be of any size. You can assume them to be 64-bit floating point numbers.
2. String: These are sequence of ASCII characters enclosed in single quotes. For example: 'John', 'Raj', '23', etc.
3. Array: Arrays are just a collection of values of any type which can be indexed using a position in the array. Array can be used as a powerful building block for composite values too.
4. `NIL` expression. A value used to denote a null or empty value.

Types of expressions

1. Constant Expression: This represent a constant value. Example, *5* is a constant expression which evaluates to itself. There are only three types of constant expressions: Numbers, Strings and Booleans (`TRUE` or `FALSE`).
2. Array Expression: This declares and/or initializes an array. For example: `ARRAY:100` is an expression which creates an array of size 100. An empty array can also be created by specifying size as zero. The items in the array are initialized to `NIL`. Arrays can also be created using array literal expression, like `[1, 2, 3]`.
3. Variable expression: This returns the value of the variable. Example: `A` will return the value stored at variable `A`.
4. Composite expression: These are expressions which are a composition of other expressions. For example: `2 * 5` is an expression which evaluates to multiplied value of two constant expressions. Similarly, `A + 5` evaluates to sum of value at `A` and constant `5`.
5. Function expression: When the return value of a function is assigned to a variable, it's called a function expression. For example, in `LET C SUM(A, B);`.

Statements:

Statements always end with a `;` and are executed in the order specified.

- `VAR <variable name> [expression];` : Declare a variable and assign a value optionally if provided. Else assign a value `NIL`;
- `LET <variable name> <expression>;` : Assign the evaluated value of the expression to the variable.
- `BEGIN` : Start of an execution block.
  `END` : End of an execution block.
- `FUNC <function name> BEGIN ... END` : Define a function
- `IF (<boolean expression>) BEGIN ... END ELSE BEGIN ... END` : Define a conditional block.
- `WHILE (<boolean expression>)` BEGIN ... END : Define a loop

- `RETURN <expression>` : Returns the value of the expression from a function.
- `HALT` : Terminates the program immediately.

Builtins

- `PRINT <expression>` : Prints an expression on the console on a new line.
- `LEN(<array> | <string>)` : Returns the length of an array or string.
- `APPEND(<array>, <expression>)` : Appends the value of the expression at the end of the array. If the expression evaluates to an array, then it is added as the last item in supplied array. For example:

```
VAR ARR ARRAY:0;
PRINT ARR; => [];
APPEND(ARR, 1);
PRINT ARR; => [1]
APPEND(ARR, 2);
PRINT ARR; => [1, 2]
APPEND(ARR, [])
PRINT ARR; => [1, 2, []]
APPEND(ARR, [1])
PRINT ARR; => [1, 2, [], [1]]
```

- `SUBARRAY(<array>, <start index inclusive>, <end index exclusive>)` : Returns a subarray from the supplied array containing items from the start index to the end index.
- `<expression left> + <expression right>` : Sums the value of two expressions. If both expressions evaluate to array, the arrays are joined and a new array is returned.
- `<expression left> - <expression right>` : Substraction of the value of two expressions.
- `<expression left> * <expression right>` : Product of the value of two expressions.
- `<expression left> / <expression right>` : Quotient of the value of two expressions after division.
- `<expression left> % <expression right>` : Remainder of the value of two expressions after division.
- `<expression left> < <expression right>` : Returns `TRUE` if left expressions is less than right. Else `FALSE`
- `<expression left> > <expression right>` : Returns `TRUE` if left expressions is greater than right. Else `FALSE`
- `<expression left> <= <expression right>` : Returns `TRUE` if left expressions is less than or equal to right. Else `FALSE`
- `<expression left> >= <expression right>` : Returns `TRUE` if left expressions is greater than or equals to right. Else `FALSE`

- `<expression left> == <expression right>` : Returns `TRUE` if the value of left expression matches value of right expression. Else `FALSE`. If the expressions evaluate to an array, then each item is compared in the array.
- `<boolean expression left> && <boolean expression right>` : and of the boolean value of two expressions.
- `<boolean expression left> || <boolean expression right>` : or of the boolean value of two expressions.
- `NOT <boolean expression>` : returns not of the boolean express.

Keywords

`LET`, `VAR`, `FUNC`, `BEGIN`, `END`, `ARRAY`, `IF`, `ELSE`, `WHILE`, `TRUE`, `FALSE`, `NIL`, `RETURN`, `HALT`

Variables

Variables in SCREAM have a name which cannot be one of the above keywords. A variable's name can contain only symbols from the set: `[A-Z, _, 0-9]`. A name can start with an upper case character or an underscore but never a digit. It can have an underscore and digit at any position in between or at the end. For example:

```
PERSON => valid
PERSON1 => valid
PERSON99 => valid
_VAR => valid
9AB => Invalid, name cannot start with a number.
VAR => Invalid, it's a keyword.
A2B => Valid
A_2B => Valid
_BAT_MAN_ => Valid
```

Arrays

Arrays are a collection of items indexed using an integer. Arrays are dynamically increasing by appending items at the end. They can contain items of any type and mixed types are also allowed. Arrays are always stored as reference and passed around as reference. Item in an array can be set by specifying index and similarly, it can be queried using the index. Arrays are always zero indexed, i.e. index always starts with 0 and is always non negative. Examples:

```
VAR ARR ARRAY:5;
PRINT ARR; => [NIL, NIL, NIL, NIL, NIL];

LET IDX 0;

WHILE(IDX < LEN(5))
BEGIN
```

```
  LET ARR[IDX] IDX;
  LET IDX (IDX + 1);
END

PRINT ARR; => [1, 2, 3, 4, 5]

LET ARR[1] 8;

PRINT ARR; => [1, 8, 3, 4, 5]

VAR ITEM ARR[3];

PRINT ITEM; => 4

APPEND(ARR, 6);

PRINT ARR; => [1, 8, 3, 4, 5, 6]

VAR ARR_LENGTH LEN(ARR);

PRINT ARR_LENGTH; => 6

VAR ARR2 SUBARRAY(ARR, 2, 6);

PRINT ARR2; => [3, 4, 5, 6]

PRINT ARR; => [1, 8, 3, 4, 5, 6]
```

Functions

Functions are stored procedures defined using the `FUNC` keyword. The body of a function is always enclosed in a `BEGIN` and an `END` statement. A function can call another function but it has to be defined before calling it. A function can take zero or more arguments. In SCREAM, arguments are always supplied by value. Since array in SCREAM is a reference value, it's reference is passed to a function. Functions return value using the `RETURN` statement. If no return statement is present, it returns `NIL`.
SCREAM doesn't support recursion for now. Also it doesn't support passing functions as an argument to another function or assignment to a variable.

Example:

```
FUNC SUM(A, B)
BEGIN
  RETURN A + B;
END

VAR B = SUM(3, 6);
PRINT B; => 9

LET B = SUM(6, B);
PRINT B; => 15
```

Scope

Scope defines the lifecycle of variables and other entities in a program. There are few scoping rules which SCREAM follows.

- **GLOBAL SCOPE:** This is the scope at the SCREAM program's file level. Functions are always defined in the global scope. This scope is inherited by all other execution scope of `BEGIN` and `END`. The entities declared or defined in this scope live through the entirety of the program.
- **BEGIN ... END:** The variables defined in this scope have a lifecycle of within the `BEGIN` and `END` block. The entities are not visible outside that scope. These scopes inherit the global scope as well as any parent `BEGIN ... END` scope.

Conditionals

These are the `IF (<condition>) BEGIN ... END` and the *optional* `ELSE BEGIN ... END` block.

Error

Whenever the program encounters an error like `DivideByZero`, `ArrayIndexOutOfBounds`, etc, the interpreter should panic and halt printing the error, it's description and line number.

Comments

SCREAM supports only single line comments. Comments start with the identifier `//` and ends at a new line, unless it's specified inside a string, in which case it's considered a part of the string. For example:

```
// Hello, I am a comment!
'This is not // a comment'
```

Test SCREAM Program

```
VAR A 5;
VAR B 10;

PRINT A+B; // prints '15'
PRINT A*B; // prints '50'
PRINT B/A; // prints '2'
PRINT B-A; // prints '5'
PRINT B%A; // prints '0'

LET A 'Hello, ';
LET B 'World!';

PRINT A+B; // prints: 'Hello, World!'
```

```
PRINT B+A; // prints: 'World!Hello, '

LET A = 3;
LET B = 5;

IF (A > B)
BEGIN
  PRINT '3 is greater than 5';
END
ELSE
BEGIN
  PRINT '5 is greater than 3';
END

VAR FACTORIAL 1;
VAR NUM 1;
WHILE(NUM <= 5)
BEGIN
  LET FACTORIAL (FACTORIAL * NUM);
  LET NUM (NUM+1);
END

PRINT FACTORIAL; // prints 120

VAR ITEMS ARRAY:5;
PRINT ITEMS; // prints [NIL, NIL, NIL, NIL, NIL]

LET NUM 0;

WHILE (NUM < 5)
BEGIN
  LET ITEMS[NUM] (NUM * 2);
  LET NUM (NUM + 1);
END

PRINT ITEMS; // prints [0, 2, 4, 6, 8]

FUNC SQUARE_SUM(NUM1, NUM2)
BEGIN
  RETURN (NUM1 * NUM1) + (NUM2 * NUM2);
END

LET A = 3;
LET B = 5;

VAR C = SQUARE_SUM(A, B);

PRINT C; // prints '34'
```

GOOD LUCK!