gjsy (Go JS Yourself!)

The goal of this project is to run a JavaScript interpreter in the Go programming language. However, why would we want to do this in the first place? JavaScript is an excellent language with much adoption in the front and back-end of web development. One of the root principles of the language is the event-based approach to solving concurrency and parallelism in run-time execution. However, another approach would be to use multi-threading. Because of its nature, JavaScript lacks multi-thread capabilities. So, in this project, we will add a new directive to JavaScript, akin to Go's "goroutine" `go` directive. We will implement it and allow us to fork threads in our interpreter, leveraging the thread management to Go.

With this goal in mind, we decided to delimiter the number of instructions the interpreter will support. Doing a full implementation for JavaScript would blow the intentions of this exercise and probably would be impossible with limited time constrain. We have support for the following:

- Literals:
  - Integer
  - Boolean
  - String
  - Null
  - Undefined
- Operations:
  - Prefix (`-` and `!`)
  - Binary (`+`, `-`, `/`, `*`, `%`)
  - Comparison (`&&`, `||`, `>`, `>=`, `<`, `<=`, `==`, `!=`)
- Statements:
  - New variable definition
    - Support for both `let` and `var` syntaxes (`var name = "value"`), also supporting uninitialized variables (`let name;` which have an `undefined` value)
  - Return
    - With the support of returning values, and without a value, it will return an `undefined`
  - Closures
- Expressions:
  - Function
    - With support for named (`function name() {}`), and anonymous functions (`function() {}`)
  - Function application (`function() {...}()`)
  - If/Else
  - While loops
  - Variable reassignment (`name = "new value"`)
  - Dot (`a.b`)
    - Even though we have support for the dot expression, we have a

    minimal object system, which is used just for built-in exposed functions, namely `console.log`, and a minimal barrier for the threads by using Go's `sync.WaitGroup`, exposing a single instance of a wait group and providing support for `wg.add(<integer>)`, `wg.done()` and `wg.wait()`
- Go function application
    * This will start a new go routine (`go function() {...}()`)

As we preferred to emphasize the project's original goal (to bring multi-threading to JavaScript), we decided not to implement some basic operations like string concatenation and grouped expressions.

Lexing and Tokenization

We decided to write our interpreter from scratch. So, we started with the lexical analysis component, which reads the source code from the REPL or the file parser, then goes through the tokenization, which the parser uses later to generate the Abstract Syntax Tree.

We defined a package with the token constants with the symbols we will support from JavaScript, from defining the end of file (EOF), identifiers, integers, and strings. Operations, binary, logical, and comparison. Delimiters, like the comma, semicolon, parentheses, and braces. And our supported keywords including boolean, null, and undefined literals, variable assignment through let and var, function declarations, return, if, and while statements, and the go operand. The Token structure holds the actual literal token as a string, and the type, which is the constant described before.

The lexer component reads character by character, holding the current and a lookahead positions, along with the existing character of the byte type being analyzed. Given that JavaScript is not dependent on white spacing or tabbing, as other languages are, we discard these characters, including line breaks.

The parser consumes the lexer, exposing a public method to get the next token. If it finds that the current character is a simple token (e.g., `+`, `-`, `*`, `!`, `=`, etc.), it will immediately return the given token. However, if the token may be composed of a second character (e.g., `!=`, `==`, `>=`, etc.), then it looks ahead to the next character to decide if it should return a single character or the composition (i.e., `!`, or `!=`). We also have pairs that need to have a second character, like the and, and or logical operations (`&&`, `||`); these will only return the token if both characters are contiguous. If the lexer finds a number, it will treat is an integer literal and will read until the next character is other than a digit. We also have support for string literals with either single or double quotes, and if the lexer finds one of these characters, it will read until it finds a matching quote ahead. Finally, if the character is a letter, it will read the bytes until the next white space. First, it will check the keyword map (Go's maps are a key-value store, called Hash Maps, Hash Tables, Dictionaries in other languages). If it finds it, it will return the token for that keyword; else, it will treat it as an identifier (i.e., a variable, function name, etc.). The lexer will stop once it finds an EOF. And finally, if it doesn't match the previous rules, it will return an illegal token.

Parsing

The parser is responsible for reading the passed input and using the lexer to build an abstract syntax tree. It keeps track of the current token, and the next token used when there are optional tokens right after (i.e., both `return 0;` and `return;` are valid expressions in JavaScript).

The parsing is split into three main components. First is the parsing of statements, which are two; the variable declaration (`var` and `let`) and the `return` statement. The next component, parses literals (e.g., integer, booleans), prefix expressions (i.e., not [`!`], and minus [`-`]), and expression (e.g., `if`, `while`, `function` definition, etc.). Finally, the last component parses operation expressions, the comparison and binary operations, the assign and dot operation, and the function application.

The parser also has an errors list in case of an error while parsing or lexing the input. This list is exposed through a public method, so other components that use the parser can display the error to the end user.

Environment

The environment represents the evaluation context for any given expression. It is defined in environment.go and allows the persistance of state during the lifetime of the program.

Evaluation

The evaluator is responsible for the execution of the program. It does this by recursively evaluating the program, its statements, and expressions. In the above file, there is a single switch statement which selects on the token type to evaluate. Further evaluation is conducted in subordonate files. For example the identifier.go file evaluates an identifier given the environment. Ultimately the evaluator recursively calls statements and expressions in an abstract syntax tree given a particular environment. The result of the evaluation is the return value of the program, and the environment is also mutated as a side-effect.

Program Evaluation

A program is simply a list of statements which are executed in order. This is considered the top-most node in our AST.

Statement Evaluation

There are 3 basic statements which are executed by our program,

- Expression Statements , allowing expressions to be executed

- Let Statements , allowing variables to be defined in the scope of the current environment
- Closure Statements , allowing closures to be defined (`function funcName() {}` syntax in javascript)

Expression Evaluation

There are many expressions which we can evaluate in `gjsy`. While we don't have exact feature parity with javascript, we are quite close to what someone might learn in their first month. Certainly close enough to write a program to get through most interview questions. Rather than introducing every expression here, we encourage the reader to review evaluator.go at line 32 for a more complete understanding of each expression.

Parallelism and Go

Perhaps the only unique facet of this project is the application of go routines to evaluate function expressions in a go routine. This can be found in function.go in the `applyGoFunction` function. This function spawns a new go routine to run the `Eval` function given the current environment. The routines generated by using the `go` application syntax can then be waited on by using the built in `wg` object builtins.go. There is an example of this in multi_thread.js which can be run by using the following command:

```
make run FILE=examples/multi_thread.js
```

Conclusion

We set the initial goal of having a JavaScript interpreter and adding multi-threading capabilities to it. As we started implementing this project, we realized the plan was ambitious. So we narrowed the support of the instructions to what would allow us for a minimal implementation. We decided to leave out string concatenation, grouped operations for expressions, the user-defined object system, and other number formats (i.e., doubles, floats), and focused on adding the multi-thread capabilities. Initially, we had neither thread barriers nor a way to print out to the standard output. So, we decided to introduce the minimal internal object system to expose `console.log`, and a singleton of Go's `WaitGroup`.

We are satisfied with the project results, considering the given limitations due to the time constraints. We want to continue the work on this implementation, with the possible future improvements of what we couldn't implement, already described before. And possibly even adding a compiler would allow compiling JavaScript source code.

Working with the Source Code

The source code is available at GitHub: https://github.com/go-js-yourself/gjsy.

Pre-requisites

This project uses Go 1.18, but the `Makefile` commands will run on Docker to help with the project set up, prerequisites, etc. So, the only two pre-requisites are:

- make
- Docker

Testing

If using the provided `Makefile`, run:

```
make test
# or with Go
go test ./...
```

Building

After building, a single binary `gjsy` can be found in the `build/bin` directory, if running with go, it is usually installed in `$GOPATH/bin`

```
make build
# or
go install ./...
```

REPL

This project includes a REPL, you can run it with:

```
make repl
# or
go run cmd/gjsy repl
```

File Interpreter

There are several provided examples in the `examples` directory, to interpret a single file, run

```
make run FILE=examples/hello_world.js
# or
go run cmd/gjsy examples/hello_world.js
```

Running the Provided Examples It is possible to run all of the provided examples with a single `make` target, just run the `examples` target.

```
make examples
```

Credits

- Miles Possing (mpossi2)
- Ivan Valdes Castillo (iv4)

References

This effort was heavily influenced by Thorsten Ball's book "Writing an Interpreter in Go"