Clove-lang

Clove is a Java-based, interpreted, dynamically typed, general-purpose programming language. It is primarily a procedural language, but has functional programming features, such as lambda expressions, and the programs can be composed in ways that utilise the declarative paradigm. The language allows for defining immutable primitive values, but compound structures, such as lists, can still be changed, but not reassigned; otherwise, all values are mutable.

Common statements and expressions, such as loops, if-statements, variable and function declarations/calls/invocations are conventional and similar to other languages, namely JavaScript. Variable names cannot start with a number and must be preceded by either *let*/*const* keywords or their aliases.

Many keywords included in the language can be expressed in a bunch of ways – Clove-lang contains Polish translations of commonly used keywords. The translations can easily be expanded by slightly altering the grammar. These translations enable people not knowing English to start programming and familiarise themselves with programming concepts more easily.

```
if (true)
  write("Yes")

jesli (prawda)     // if := jesli
  wyswietl("Tak") // write := wyswietl
```

Data types

The following data types are supported:

- integers,
- rational numbers,
- strings,
- booleans,
- anonymous objects,
- lists,
- arrays, and
- anonymous functions.

The integer and rational types are in Java's *long* and *double* ranges respectively. The list type is similar in functionality to Python's *List* and JavaScript's *Array*. Array includes all the list's features, but has a fixed size. Anonymous function is unique in that it is a container for a function definition instead of being a standalone implementation of a type.

All the types evaluate to values that are treated similarly – they can be passed around, nested, evaluated, and returned from functions. It applies to both literals and dereferenced values. Some operations, such as addition, can be performed on mixed types, but there are reasonable restrictions, e.g. it is impossible to multiply a string by a number, but it is fine to add a number to a string.

Another type supported by the language that is not listed above is *ValueReflection* – it is a container for any Java class that is referenced by the *reflect()* built-in function. The container holds the chosen class-name, a constructor found in the class based on the given arguments, and an instance of the class created with the constructor. This feature of the language allows for using virtually any data type supported by Java and leveraging their own built-in methods. Please note that this feature is very early in the development and it does not allow for more advanced operations.

Declarations and definitions

Declarations and definitions in Clove are similar to those in JavaScript. To define an empty list or anonymous object, their empty literals must be used; empty array can only be declared, not defined. Functions can be defined in two ways: standalone, with the *function* keyword, or as an anonymous *value-function*. Examples of some common declarations and definitions can be seen below[1]. The variables defined at the top are accessible from every subsequent block, therefore they have a 'global' scope in this example. Variables in function invocations or any other block are defined in their corresponding scope (are inaccessible from the outside) and removed after the block is finished.

```
let foo                                    // Variable declaration
let num = 77                               // Variable definition
var str = "bar"                            // 'let' and 'var' are equivalent
const pi = 3.14                            // Constant definition
let flag = false                           // Boolean definition

function sumAndDouble(param1, param2) { // Function definition
  return (param1 + param2) * 2
}
const sq1 = fn(val) {return val * val}  // Value-function definition
const sq2 = (val) => {return val * val} // Arrow function

let emptyList = []                         // List declaration
let list = ["foo", "bar", "baz", 2000]     // List definition
let emptyObject = {}                       // Anonymous object declaration
let object = {                             // Anonymous object definition
  key: "value"
}
let emptyArray[]                           // Array declaration -- 0 capacity
let emptyArrayWithCap[10]                  // Array declaration -- explicit cap
let array[] = {str, list, object, flag} // Array definition -- implicit cap
let arr2[15] = {"str", ["list"], false} // Array definition -- explicit cap
```

Syntax

Semi-colons are truly optional; this way anyone used to typing them at the end of statements will not be punished by the parser. Furthermore, it allows some basic JavaScript code snippets to be copy-pasted into a Clove program without special accommodations and rewriting. There are a few shorthand operators for arithmetic operations which can be used in complex expressions. Basic examples shown below.

```
let ten = 10;
let twenty = 20;

ten++;          // 10 (11)
++ten;          // 12 (12)
twenty--;       // 20 (19)
--twenty;       // 18 (18)

ten += twenty; // 30
ten -= twenty; // 12
ten *= twenty; // 216
ten /= twenty; // 12
```

Note: all arithmetic operations try to result in an integer. If the result cannot be parsed to an integer, a rational number will be returned.

```
3.25 / 0.25 // 13 (not 13.0)
```

[1]More code snippets and example programs are available in directories with corresponding names.

Prototype functions

The string, list, array and anonymous object value-types have built-in prototype functions that can be accessed using the arrow-notation. Prototype functions are invoked just like normal functions, with the parentheses after the function's name. Examples shown below.

```
const list = [13, "foo", ["bar", "baz"]]
const nestedBar = list[2]->shift()        // bar

const obj = { ten: 10, twenty: 20 }
obj->remove("ten")                        // { twenty: 20 }

const array[] = {77, {key: "val"}, [88]} // Array with capacity of 3
array->resize(array->cap() + 7)           // Increases the array's cap by 7

const str = "stringstring"
str->length()                             // 12
```

Data structures

The supported data types allow for creating compound data structures. The code below demonstrates an example usage of anonymous objects and functions to imitate a class-like structure.

```
const Car = {
  cars: [],
  get: (carInstance, index) => { return carInstance.cars[index] },
  set: (carInstance, brand, year, colour) => {
    carInstance.cars->append({
      brand: brand,
      year: year,
      colour: colour
    })
  }
}
const vehicle = Car
vehicle.set(vehicle, "Toyota", "2002", "red")
log(vehicle.get(vehicle, 0)) // {colour: red, year: 2002, brand: Toyota}
```

Built-in functions

Clove-lang has some useful built-in features, some of which interact with the outside world.

get_args() Returns a list of command-line arguments passed into the program.

random() Returns a random integer or a rational number in a given range. `random(0, 3.14)` would return a random rational number in the 0 <= x < 3.14 range – it recognises a floating-point number in the second argument, therefore the result will also be a float.

```
const args = get_args()
const r = random(0, args->length())
log(args[r]) // Random argument
```

file() It takes three arguments: writing option, path, content. The function's behaviour depends on the first argument which specifies whether the new content should be appended to the file, or replace the old one. If the path does not exist, it will be created, although it is limited to one subdirectory.

http() This function supports four main HTTP requests: GET, POST, PUT, and DELETE, one of which has to be specified in the first argument. The second argument is the resource's URL, and the third, optional argument is the request body (only applicable to POST and PUT requests) which is a string of url-encoded pairs, or an anonymous object.

```
const url = "https://reqres.in/api/users/2"
const body = { // Equivalent to: "name=John Doe&job=director"
  name: "John Doe",
  job: "director"
}
const response = http("POST", url, body)
file("overwrite", "./subdir/file-writing-demo.txt", response.body)
```

reflect() Java's Reflection API is accessible via Clove's reflect() function. It takes one or two arguments, first being a string with a full package name (e.g. "java.lang.String"), second is a list with arguments which are used to instantiate the class. If the second argument is not supplied, the class is not instantiated. Current implementation only allows for simple tasks, such as string comparison or concatenation.

```
const math = reflect('java.lang.Math')
const rational = 13.77777
math.round(rational) // 14

const str1 = reflect('java.lang.String', ['imastring'])
const str2 = 'otherstring'
str1.compareTo(str2) // -6
```