SCoLang : Small Contract Language

vv2282 - Varun Varahabhotla
jc4697 - Jackson Chen
sa3433 - Sambhav Anand
sr3355 - Sushanth Raman
kv2295 - Kanishk Vashisht

Introduction

A language made with the premise that everything is a contract waiting to execute. Each contract has defined listeners, including listeners which listen for other contracts recursively. When all listeners are satisfied, a set of contract conditions get executed.

Purpose

We live in a world full of routine tasks. Every day, millions of people take repeated actions based on guaranteed stimuli - all the way from turning on the lights when they wake up to making git commits after a function is declared. The scope of these actions is broad, and we envision a general purpose language that can make a dent in this scope. Which is why we're writing a language that works by listening and reacting inspired by blockchain smart contracts. This is the language that will be usable for anything from simple load balancing to complex IOT applications.

Use Cases

1) Notifying users about certain events (e.g. weather drops below 40 degrees)
2) Distributed system management
3) Recurring tasks (e.g every time it hits 7am turn off Nightshift).
4) Cycle numeric operations (eg. GCD)
5) Triggering events based on physical sensors (e.g. if multiple burglar alarms go off, we can trigger multiple events accordingly)

Code Syntax

Data Types

Primitives

`long, float, char`	Regular primitive types (char: ‘’)
`Array`	An aggregate data type; [] Python style (slicing, referencing, etc)
`bool`	Ternary data type: true, false, null
`null`	A data type that represents nothing
`String`	A text value. Can be a single char or an arbitrary number of chars concatenated together.

Operators

`== !=`	Equality and inequality operators. Does a deep comparison to ensure objects contain the same value and are from the same memory location.
`()`	Parentheses helps indicate operator precedence.
`> < >= <=`	Greater than and less than operators.
`&& ||`	Represents logical AND and logical OR respectively.
`.`	Lets you access elements underneath composite data types.
`->`	Binds listeners to actions to create a contract. Can bind multiple listeners and multiple actions at once.
`savage`	Combine different operators using boolean arithmetic (x AND b)
`exit`	An operator to kill a listener; indicates to memory to conduct garbage collection
`resolve`	An operator within a listener that completes the listener logic and sets its value to be 1 in the event loop
`%`	Modulus operator

`print`	Outputs to standard out
`input`	Reads and stores from standard in
`webhook`	Initiates a connection to a specified endpoint (the port is automatically opened on the program's end). Returns a boolean (default 0, set to 1 for a tick if the endpoint is requested).

Reserved Keywords

`if`	if statement as in other standard languages
`Contract`	Binds listeners to actions and then triggers actions if conditions for listeners are satisfied.
`Listener`	Checks to see if certain conditions are met. (e.g. memory usage exceeds a certain amount)
`Action`	A specified event that is meant to triggered by a contract if the conditions for the listener are meant. (E.g. turn lights on)
`/* */`	Indicates a multi or single line comment
`print`	Prints out String or native object types

Code Example

GCD:

```
long a = 10; /*or input*/
long b =  4;
long t;

Listener keep_going = { if(b != 0){ resolve(); } };
Listener found = {if(a==b){ resolve(); }};

Action subtract = {
```

```
      if(a > b) {
            t = b;
            b = a % b;
            a = b;
       }
};

Action display = {
      print(a);
};

keep_going -> subtract;
found->display;
```

Hello World:

```
bool helloHook = webhook('/hello');

Listener hello = {
      if (helloHook) {
            resolve();
      }
}

Action print_hello = {
      print("Hello World"!);
}

hello -> print_hello;
```