

Visually programming Go

21 November 2023

Daniel Esteban

VPLs - Visual Programming Languages



*"In computing, a visual programming language (visual programming system, VPL, or, VPS) is any programming language that lets users create programs by manipulating program elements graphically rather than **by specifying them textually** ."*

Excerpt from [Wikipedia](https://en.wikipedia.org/wiki/Visual_programming_language) (https://en.wikipedia.org/wiki/Visual_programming_language)

Flow based

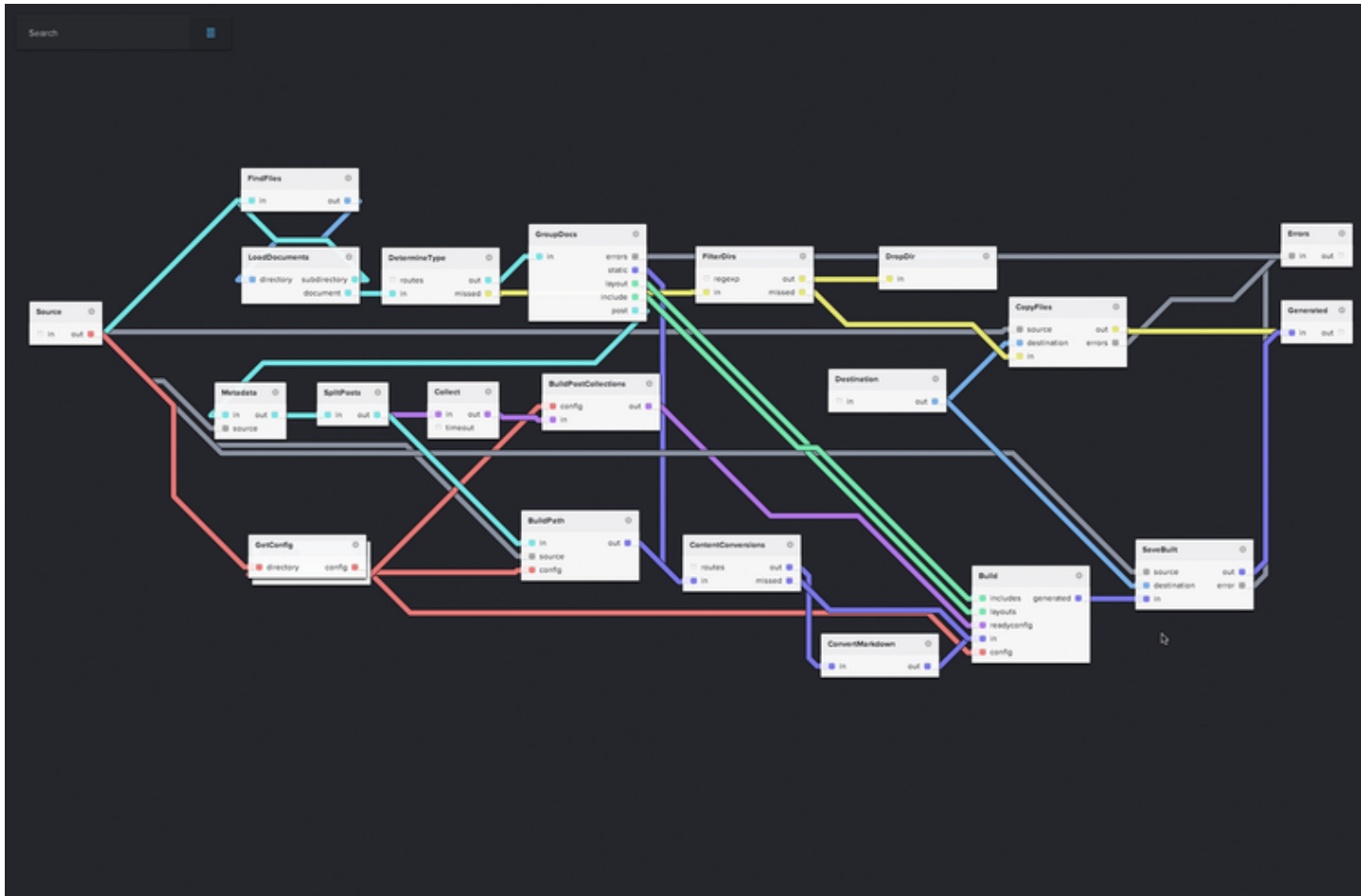


Image from NoFlo - Flow-Based Programming for JavaScript (<https://noflojs.org/>)

Block based

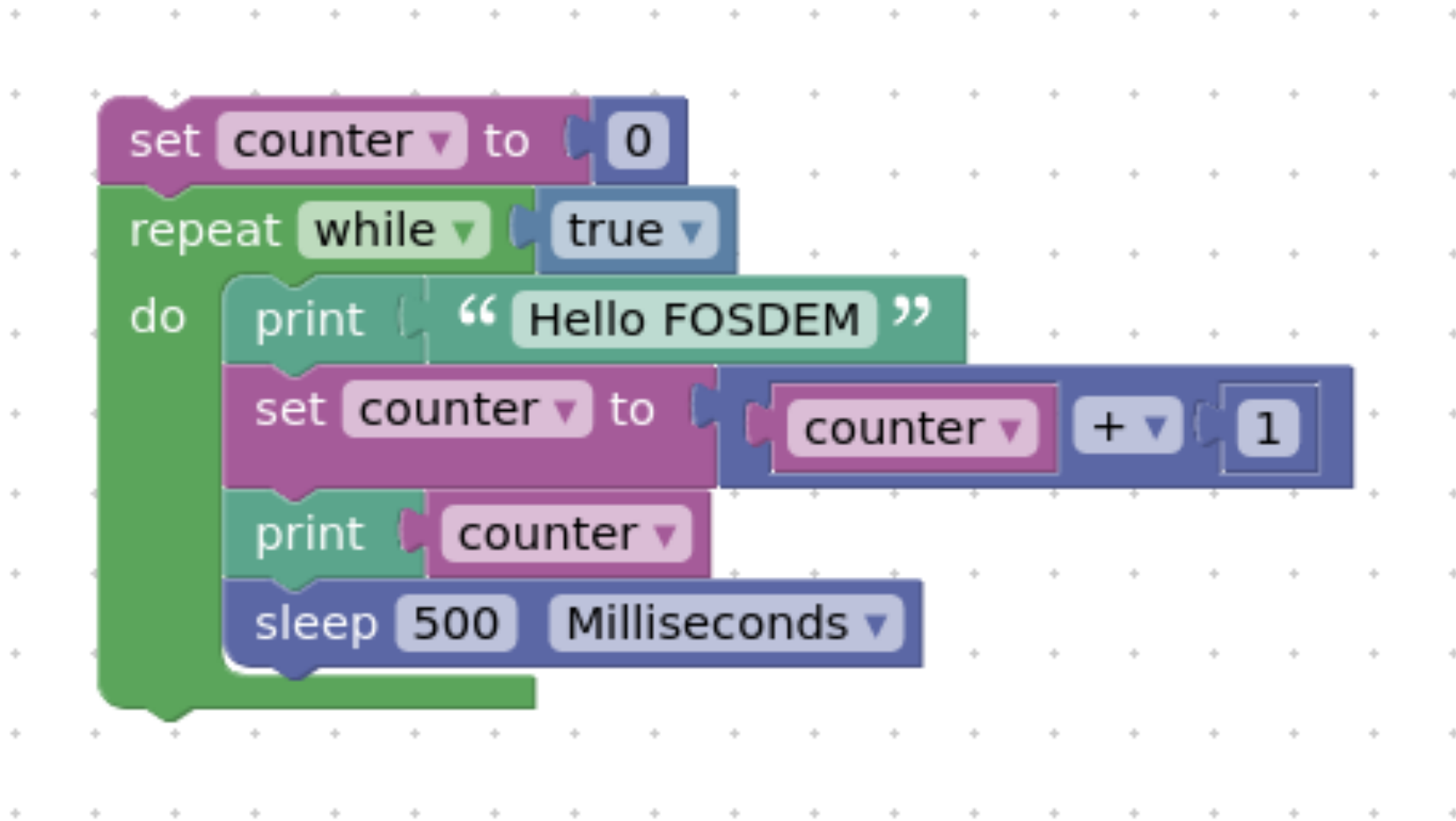


Image from [Blockly TinyGo playground](https://github.com/conejoninja/blockly-tinygo) (<https://github.com/conejoninja/blockly-tinygo>)

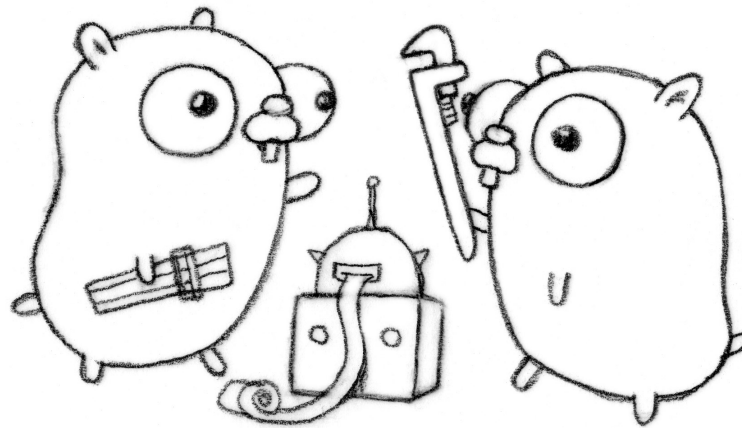
Why??

Because I like to make crazy things with Go



Why VPL?? (more seriously)

- I think programming will be an essential skill in the present/future
- It's a great way to introduce people to programming (specially children)
- Great for simple tasks (home automation, IFTTT,...)
- NoCode / LowCode movements are getting popular
- Go has a nice standard library, easy to read and multiple targets



How?

Meet Blockly (or MakeCode or Scratch or ArduBlock or ...)

The image shows the Blockly Playground interface. On the left is a palette with categories: Logic, Loops, Math, Text, Lists, Colour, Variables, Functions, TinyGo, MaQueen, Sensors, and Gopherbot. The main workspace contains a script with the following blocks: 'set counter to 0', 'repeat while true', 'do' loop containing 'print "Hello FOSDEM"', 'set counter to counter + 1', 'print counter', and 'sleep 500 Milliseconds'. On the right, the 'Blockly Playground' panel includes a 'Console' dropdown, 'Flash', 'Copy to Clipboard', 'Save XML', 'Load', and 'To Go' buttons. Below these are demo buttons: 'Hello World', 'Demo 2', 'Demo 3', and 'Demo Sim'. The code editor shows the following code:

```
"time"
)
var counter int32

func main() {
  counter = 0
  for true {
    println("Hello FOSDEM")
    counter = counter + 1
    println(counter)
  }
}
```

The terminal output shows:

```
TERMINAL
Hello FOSDEM
1
Hello FOSDEM
2
Hello FOSDEM
3
Hello FOSDEM
```

Blockly (<https://developers.google.com/blockly/>)

Blockly is ...

- Pure JavaScript library.
- 100% client side. No server side dependencies.
- Compatible with all major browsers: Chrome, Firefox, Safari, Opera, and Edge.
- Highly customizable and extensible.

Blockly does not officially support Go.

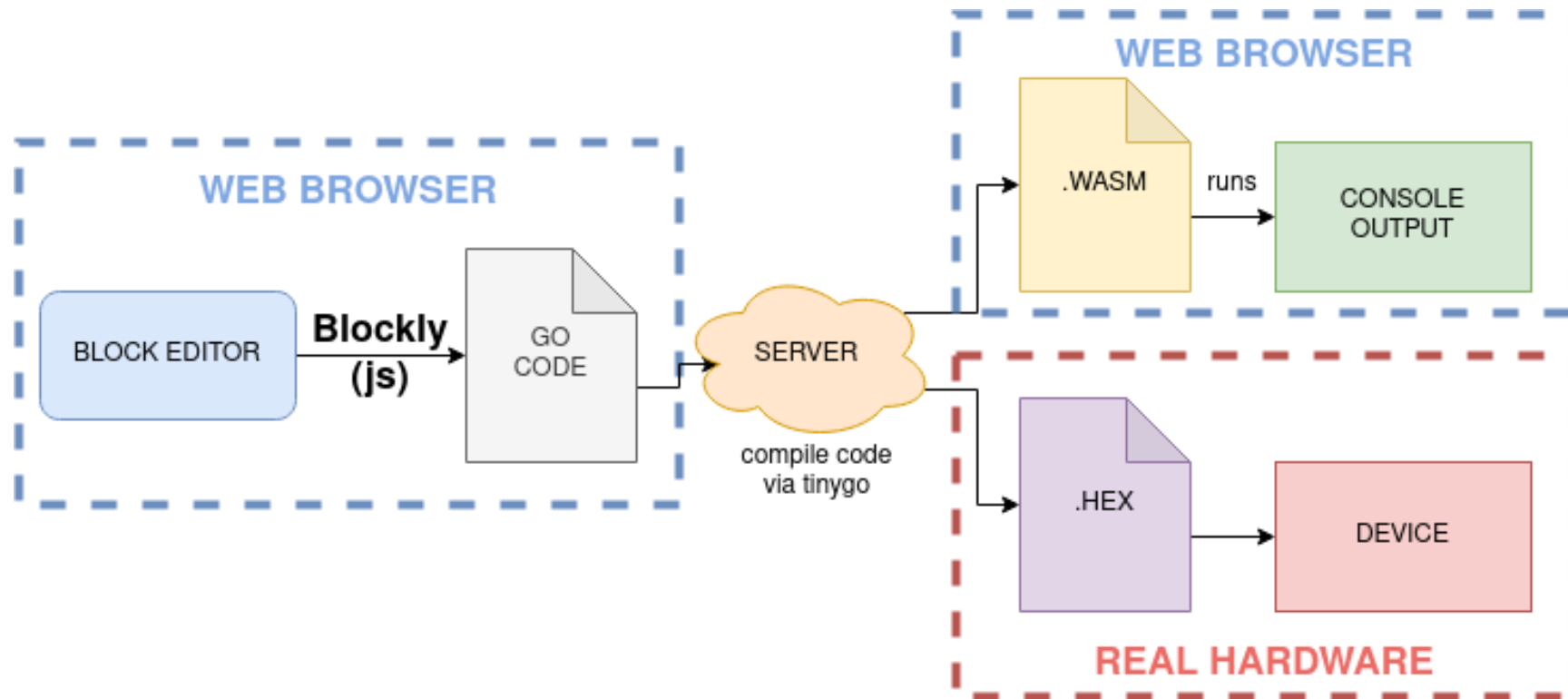
Blockly does not officially support Go

Could it?

Let's take a *quick* Tour of Go

but before the tour...

Note 1: Blockly/TinyGo Playground



Note 2: TinyGo

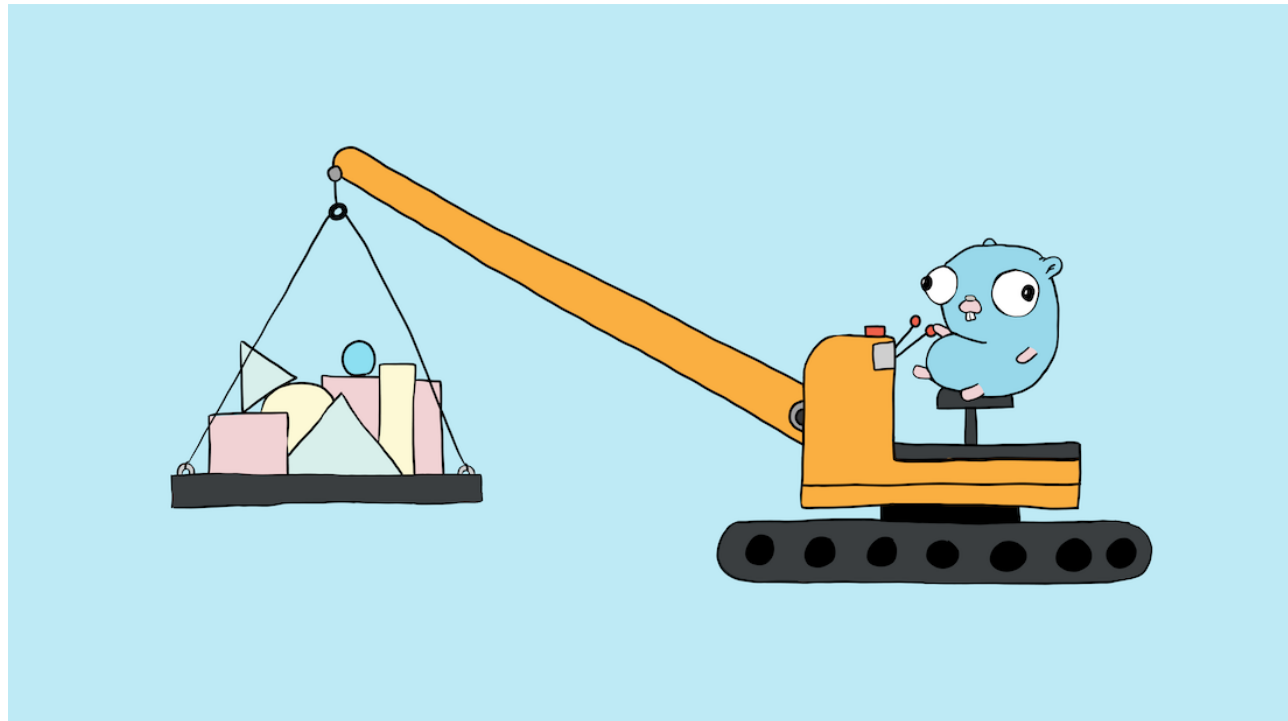


TinyGo is a project to bring the Go programming language to **microcontrollers** and modern web browsers by creating a new compiler based on LLVM.

Version 0.27.0 released on February 3rd

Fosdem 2019 TinyGo @deadprogram's talk (https://archive.fosdem.org/2019/schedule/event/go_on_microcontrollers/)

Note 3: Still under construction



Credit to Renee French for the Go Gopher

Consider this a *proof of concept* or a *work in progress*. Heavily under construction, use at your own risk.

Hello, 世界

Welcome to a tour of the Go **blocks** programming language.

Logic	
Loops	
Math	
Text	
Lists	
Colour	
Variables	
Functions	
TinyGo	
GopherBadge	
Gopherino	
Sensors	
Net/HTTP	
Go Funcs	

Hello, 世界 (generated code)

```
package main
```

```
import (  
    "fmt"  
)
```

```
func main() {  
    fmt.Println("Hello, 世界")  
}
```

Packages

Every Go program is made up of packages. This program is using the packages with import paths "fmt" and "math/rand".

Packages (generated code)

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println("My favorite number is")
    fmt.Println(rand.Intn(10))
}
```

Functions

A function can take zero or more arguments. In this example, `add` takes two parameters of type *Number**. A return statement without arguments returns the named return values.

Functions (generated code)

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(add(42, 13))
}

func add(x int32, y int32) (z int32) {
    z = x + y
    return
}
```


Variables

The var statement declares a list of variables; as in function argument lists, the type is last.

The image shows a code editor interface. On the left, a sidebar menu lists various categories: Logic, Loops, Math, Text, Lists, Colour, Variables (highlighted in purple), Functions, TinyGo, GopherBadge, Gopherino, Sensors, Net/HTTP, and Go Funcs. The main workspace is a grid of small grey squares. A palette is open over the grid, displaying three options: 'Create string variable...', 'Create number variable...', and 'New list variable'. A mouse cursor is positioned over the 'Create string variable...' option. The text 'pboard' and 'ker' are visible on the left edge of the editor, and 'nyGo")' is visible at the bottom left.

Variables (generated code)

```
package main

import (
    "fmt"
)

var (
    myString string
    myNumber int32
)

func main() {
    myString = "Hello Golab!"
    fmt.Println(myString)
    myNumber = myNumber + 1
    fmt.Println(myNumber)
}
```

For

Go has only one looping construct, the *for* loop.

- Logic
- Loops
- Math
- Text
- Lists
- Colour
- Variables
- Functions
- TinyGo
- GopherBadge
- Gopherino
- Sensors
- Net/HTTP
- Go Funcs

```
set sum to 0
```

For (generated code)

```
package main

import (
    "fmt"
)

var sum int32

func main() {
    sum = 0
    for i := int32(0); i <= 9; i++ {
        sum = sum + i
    }
    fmt.Println(sum)
}
```

If and else

Go's *if* statements are like its *for* loops; the expression need not be surrounded by parentheses () but the braces { } are required.

If and else (generated code)

```
package main

import (
    "fmt"
)

var x int32

func main() {
    if x%2 == 0 {
        fmt.Println("X is even")
    } else {
        fmt.Println("X is odd")
    }
}
```

Defer

A defer statement defers the execution of a function until the surrounding function returns.

Defer (generated code)

```
package main

import (
    "fmt"
)

func main() {
    defer fmt.Println("world")
    fmt.Println("hello")
}
```

Goroutines

A goroutine is a lightweight thread managed by the Go runtime.

- Logic
- Loops
- Math
- Text
- Lists
- Colour
- Variables
- Functions
- TinyGo
- GopherBadge
- Gopherino
- Sensors
- Net/HTTP
- Go Funcs

```
to say with: s string
  count with i from 0 to 4 by 1
  do
    sleep 100 Milliseconds
    print s
end
```

Goroutines (generated code)

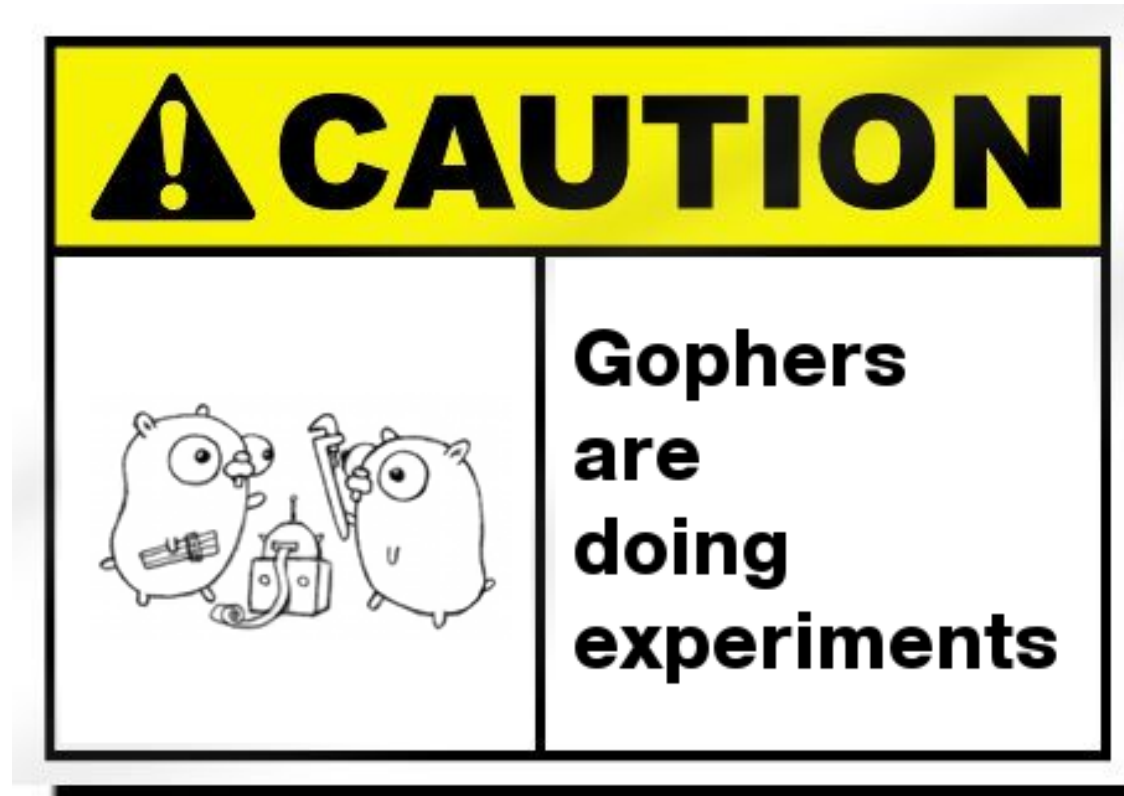
```
package main

import (
    "fmt"
    "time"
)

func main() {
    go say("world")
    say("hello")
}

func say(s string) {
    for i := int32(0); i <= 4; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}
```

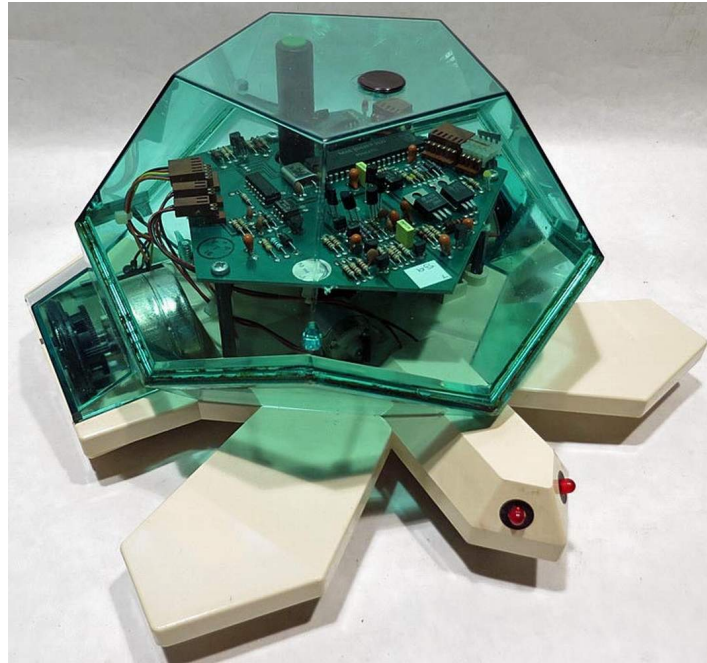
Demo time!



Let's see a few examples where this could be useful

Logo Turtle

Introducing people to programming

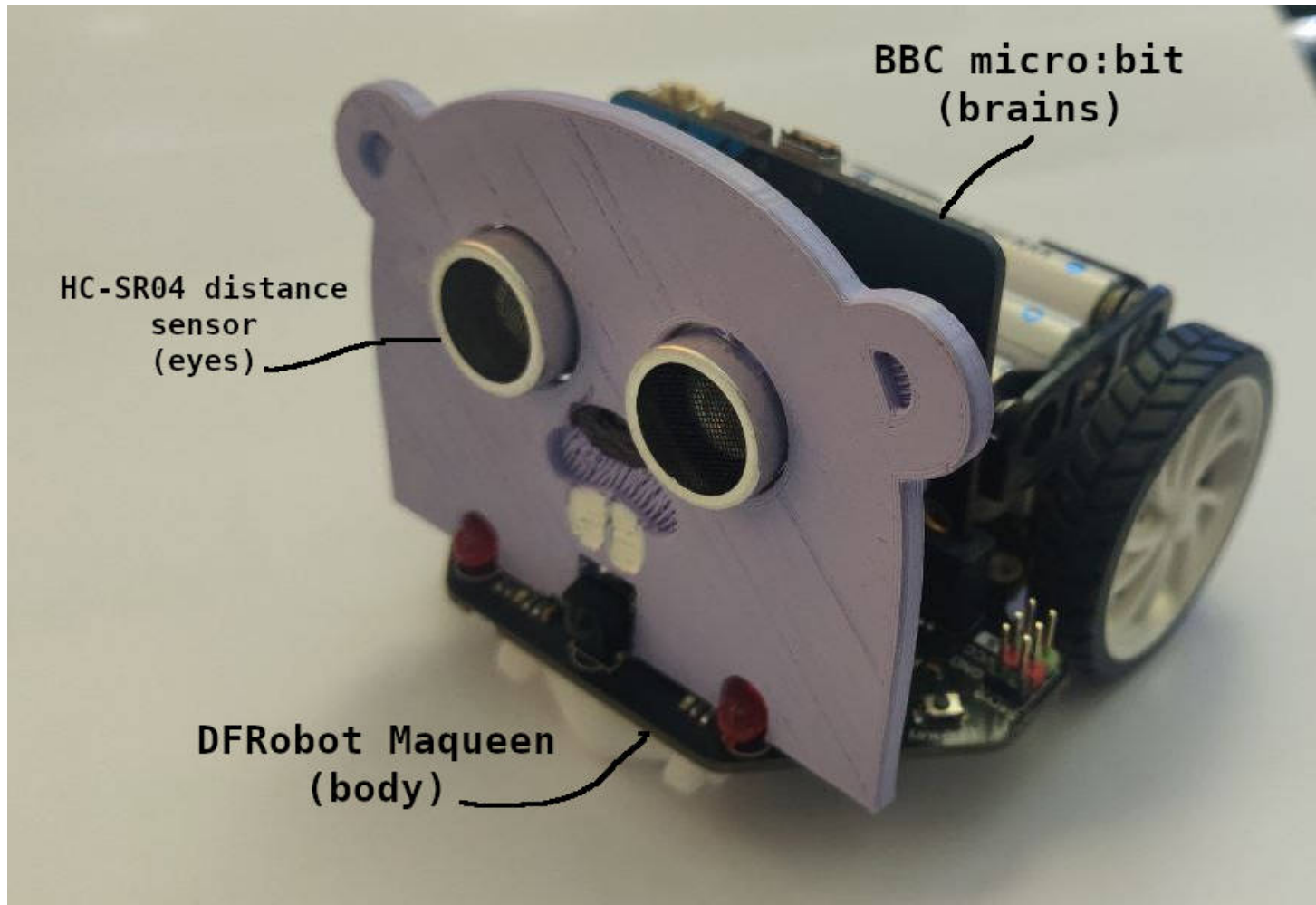


Valiant Turtle <http://www.theoldrobots.com/turtle5.html> (<http://www.theoldrobots.com/turtle5.html>)

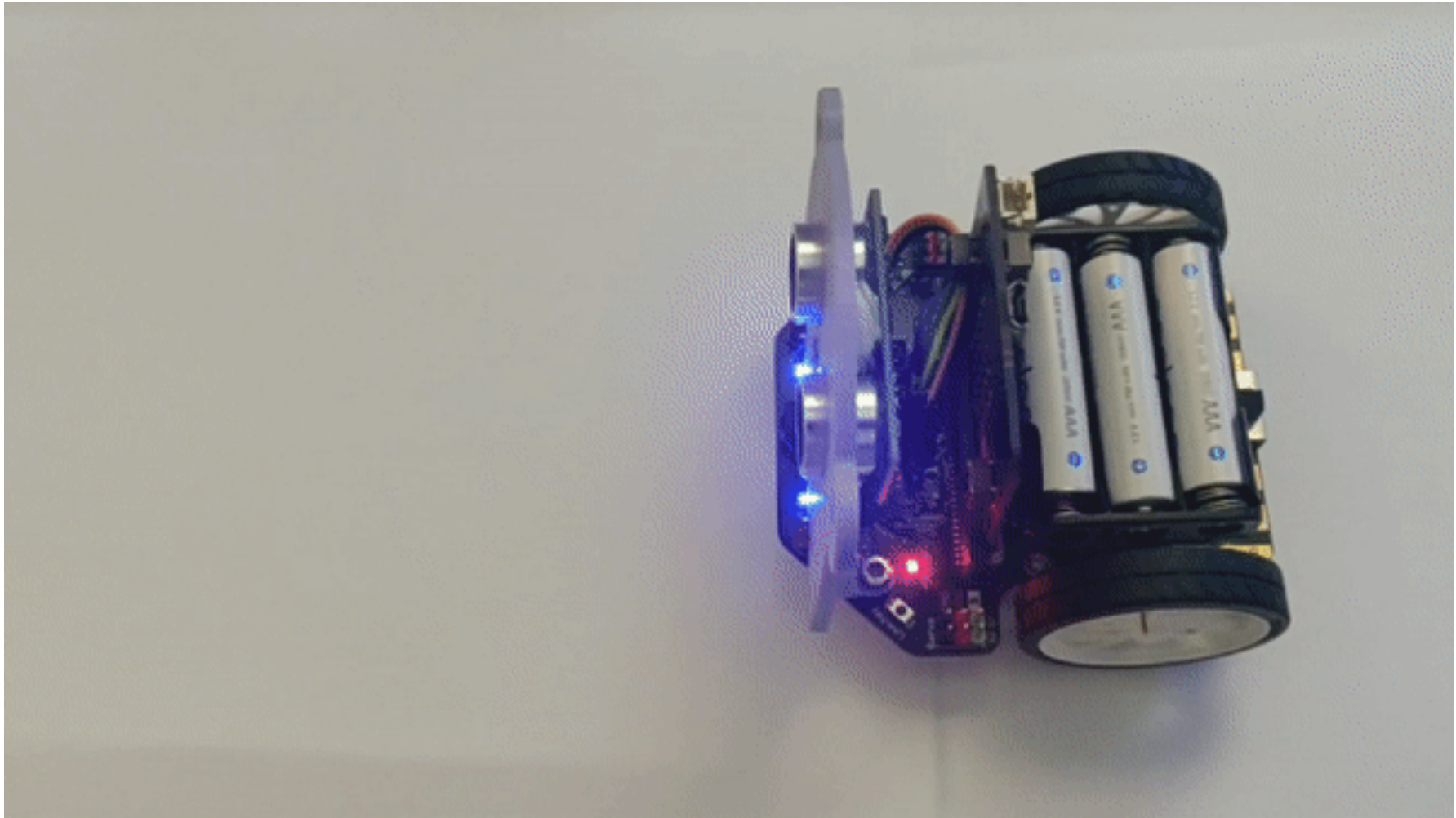
Turtles are educational robots used in computer science and mechanical engineering training, a great way to introduce programming to people.

Gopherino

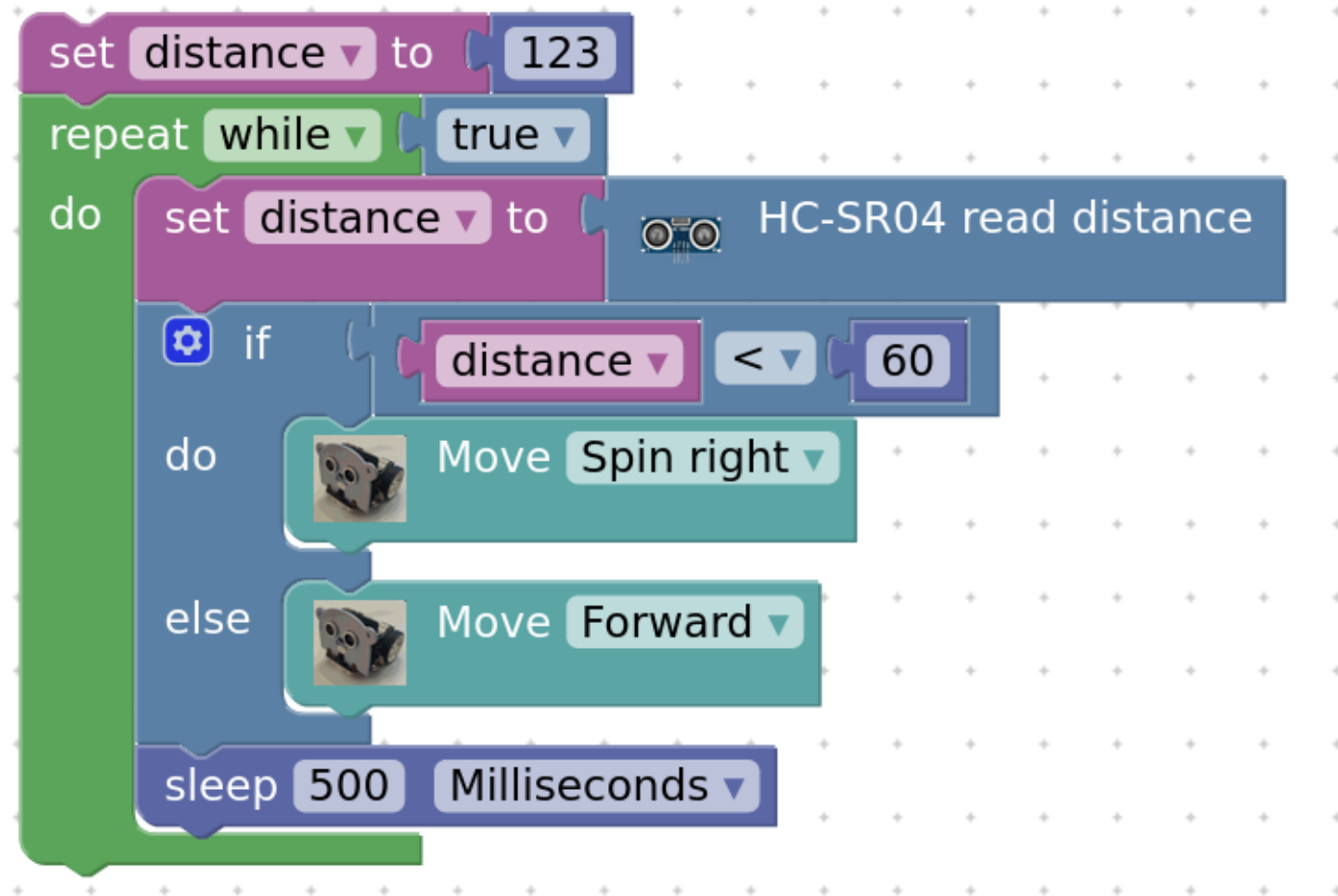
Meet Gopherino, based on DFRobot's MaQueen, powered by BBC micro:bit



Gopherino (avoiding obstacles)



Gopherino (example)



Gopherino (generated code)

```
package main
```

```
import (
```

```
    "machine"
```

```
    "time"
```

```
    "tinygo.org/x/drivers/hcsr04"
```

```
    "github.com/conejoninja/gopherino/motor"
```

```
)
```

```
var (
```

```
    distance int32
```

```
    gopherino_hcsr04 hcsr04.Device
```

```
    i2c = machine.I2C0
```

```
    gopherino_motor *motor.Device
```

```
)
```

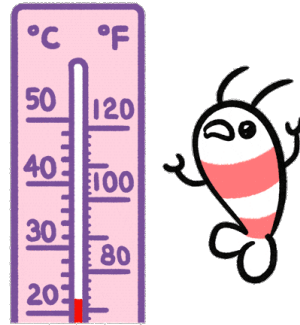

The shrimp-tank problem

Simple tasks / home automation

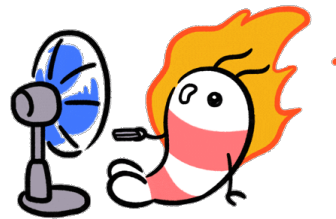


The shrimp-tank problem

Water heaters are cheap, water coolers not so much

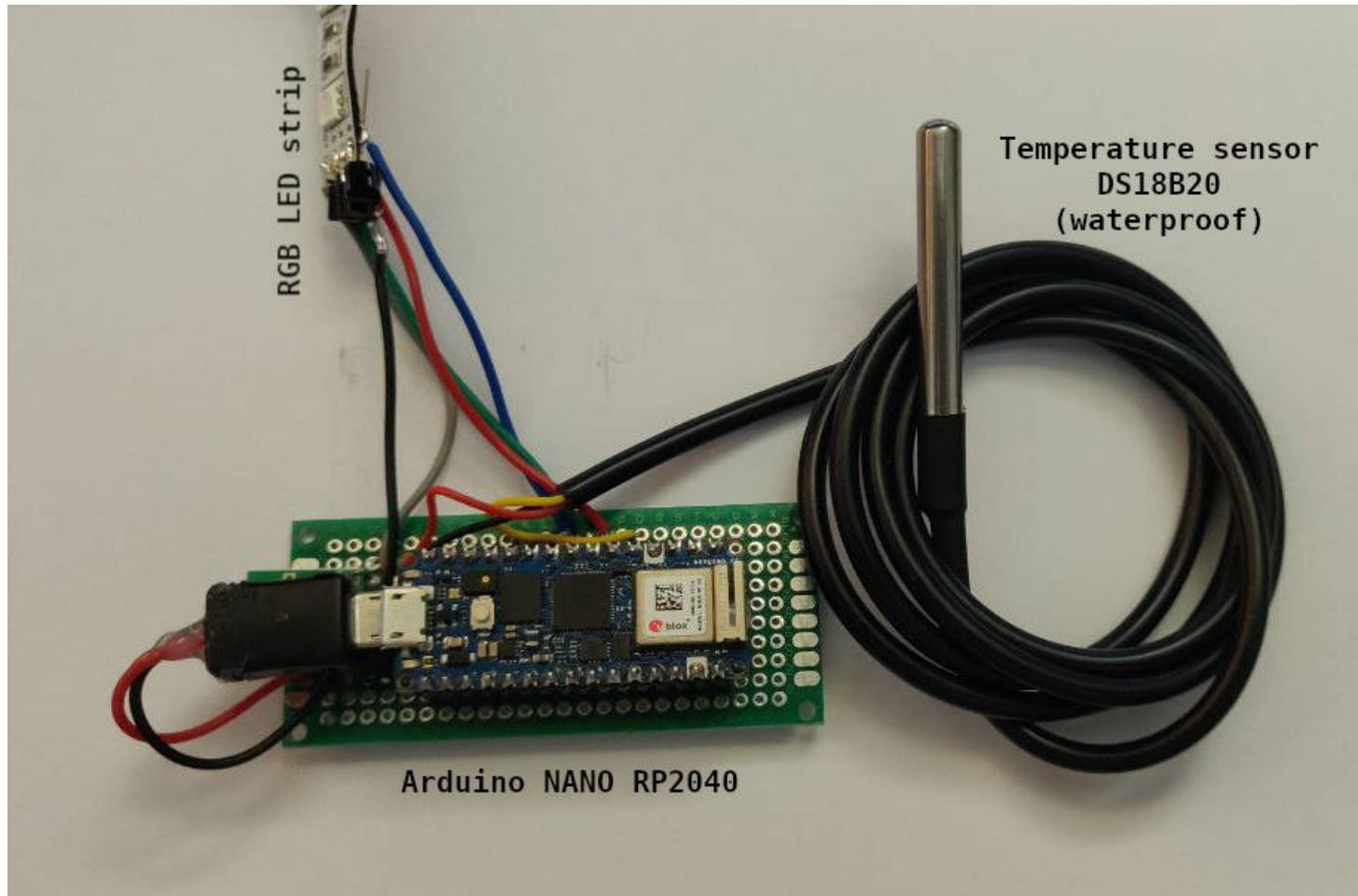


instead you can blow a fan to cool water (it's cheaper)



Animations by [pikaole](https://giphy.com/pikaole/) (https://giphy.com/pikaole/)

Circuit NANO RP2040 + DS18B20



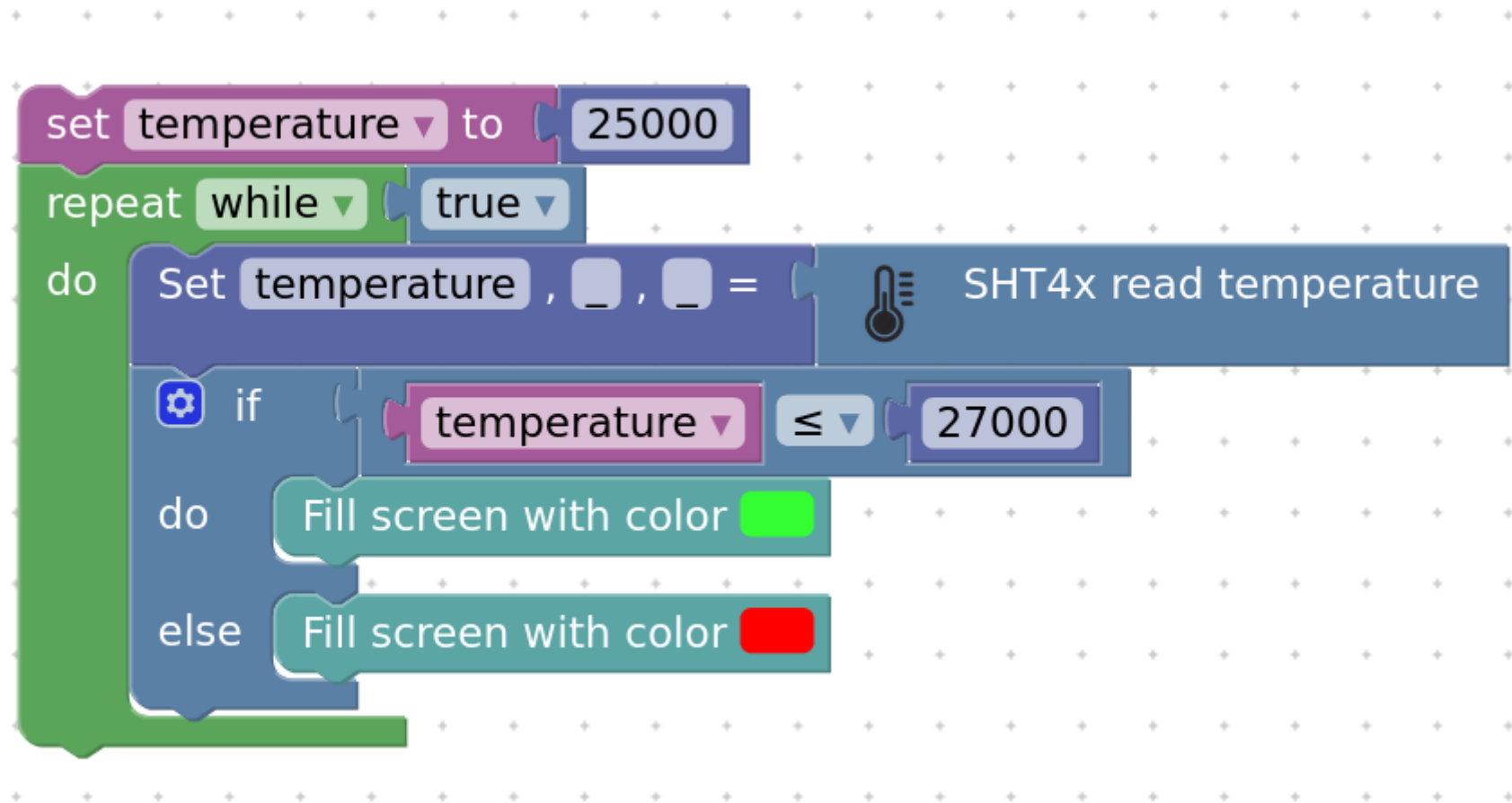
Instead of a fan, I'll use a RGB LED strip for demo purposes

Plan B: GopherBadge + SHT4x



Instead of a fan, I'll color the screen

Code (blocks)



```
set temperature to 25000
repeat while true
do
  Set temperature, _, _ = SHT4x read temperature
  if temperature ≤ 27000
  do
    Fill screen with color green
  else
    Fill screen with color red
```

The image shows a Scratch code block structure on a grid background. It starts with a purple 'set temperature to 25000' block. This is followed by a green 'repeat while true' loop block. Inside the loop, there is a blue 'do' block containing three sub-blocks: a blue 'Set temperature, _, _ = SHT4x read temperature' block with a thermometer icon, a blue 'if temperature ≤ 27000' block, and two teal 'Fill screen with color' blocks. The first 'Fill screen with color' block has a green color swatch, and the second has a red color swatch.

Code (generated)

```
temperature = 25000
for true {
    temperature, _, _ = sensors_sht4x.ReadTemperatureHumidity()

    if temperature <= 27000 {
        display.FillScreen(color.RGBA{51, 255, 51, 255})
    } else {
        display.FillScreen(color.RGBA{255, 0, 0, 255})
    }
}
```

No code / Low code + WASM

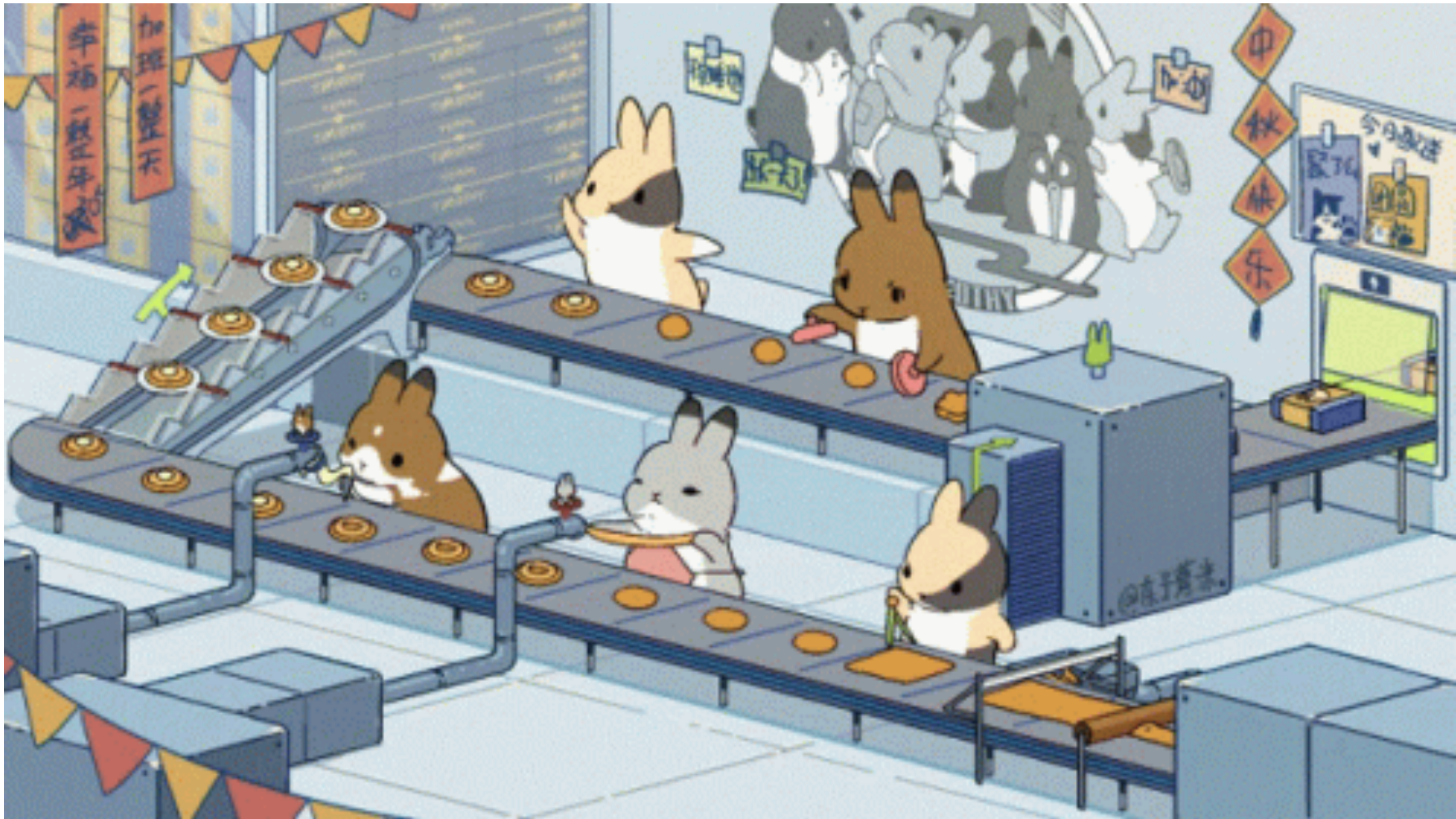
WebAssembly is getting supported by more and more entities.

Add the *easiness* of nocode/lowcode and the possibilities are limitless, from serverless code to program extensions.

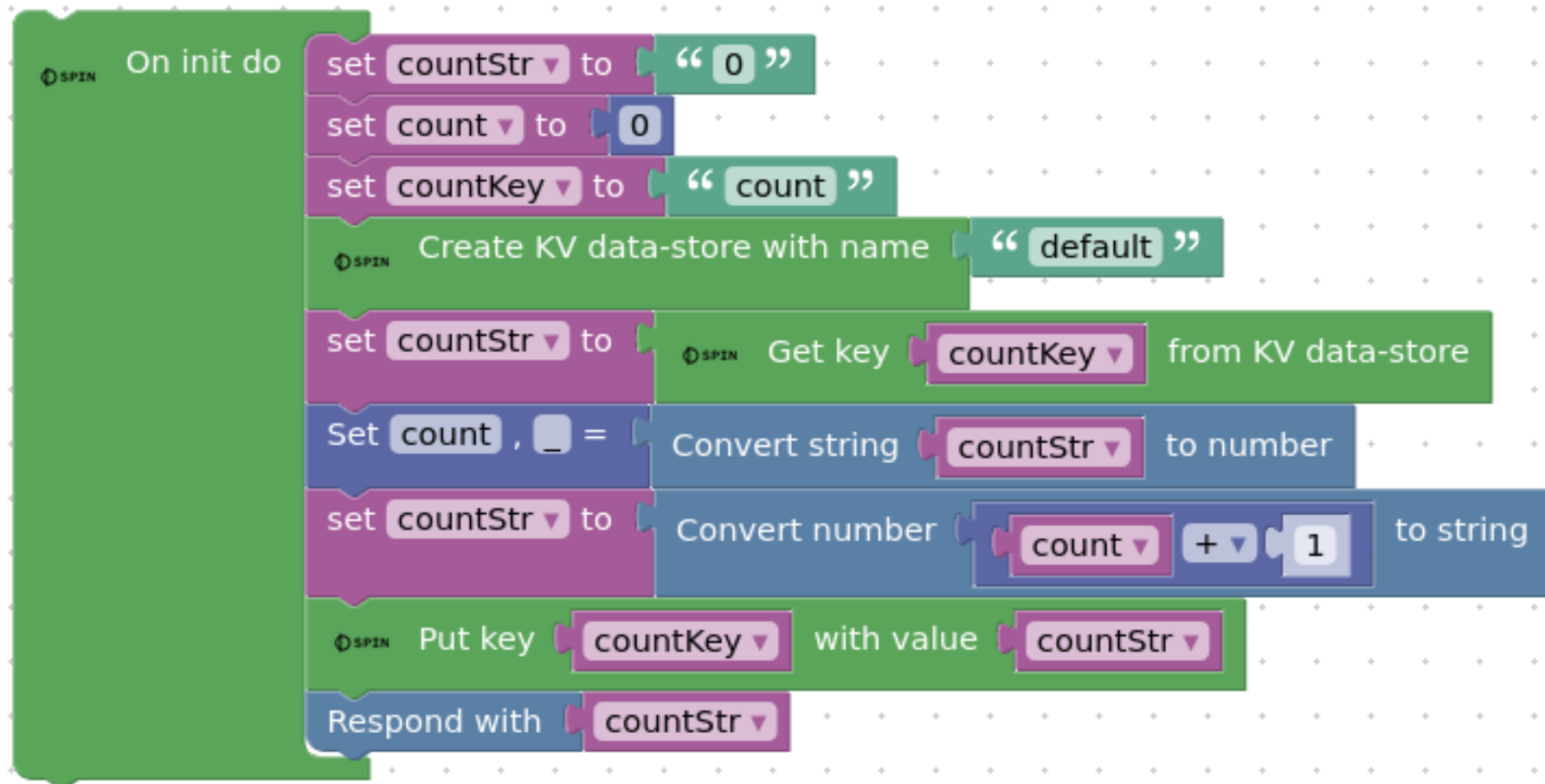
To name a few:

- [Fermyon Spin](https://www.fermyon.com/spin) - build & run event-driven applications (<https://www.fermyon.com/spin>)
- [Cloudflare workers](https://workers.cloudflare.com/) - deploy serverless code instantly across the globe (<https://workers.cloudflare.com/>)
- [Capsule](https://bots-garden.github.io/capsule/) - WASM Function Runner (<https://bots-garden.github.io/capsule/>)
- [Extism](https://extism.org/) - plug-in system for everyone (<https://extism.org/>)

Fermyon SPIN WASM worker (visit counter)



Code (blocks)



```
On init do
  set countStr to " 0 "
  set count to 0
  set countKey to " count "
  Create KV data-store with name " default "
  set countStr to Get key countKey from KV data-store
  Set count, _ = Convert string countStr to number
  set countStr to Convert number count + 1 to string
  Put key countKey with value countStr
  Respond with countStr
```

The image shows a sequence of code blocks for an 'On init do' event. The blocks are: 1. 'set countStr to " 0 "' (purple), 2. 'set count to 0' (blue), 3. 'set countKey to " count "' (purple), 4. 'Create KV data-store with name " default "' (green), 5. 'set countStr to Get key countKey from KV data-store' (green), 6. 'Set count, _ = Convert string countStr to number' (blue), 7. 'set countStr to Convert number count + 1 to string' (blue), 8. 'Put key countKey with value countStr' (green), and 9. 'Respond with countStr' (blue).

Code (generated)


```

func init() {
    spinhttp.Handle(func(w http.ResponseWriter, req *http.Request) {
        store, err := kv.OpenStore("default")
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        defer store.Close()
        countStr = func() string {
            v, err := store.Get(countKey)
            if err != nil {
                return "0"
            }
            return string(v)
        }()

        count, _ = func() (int32, error) {
            i, err := strconv.Atoi(countStr)
            return int32(i), err
        }()
        countStr = strconv.Itoa(int(count + 1))
        store.Set(countKey, []byte(countStr))

        w.Write([]byte(countStr))
    })
}

```


The blocks

The blocks (definition)



My Custom Block

```
{  
  "type": "block_type",  
  "message0": "My Custom Block",  
  "previousStatement": null,  
  "nextStatement": null,  
  "colour": 230,  
  "tooltip": "",  
  "helpUrl": ""  
}
```

The blocks (code)

```
Blockly.Go['block_type'] = function(block) {  
  // TODO: Assemble Go into code variable.  
  var code = 'myCustomBlockFunction();\n';  
  return code;  
};
```

The block generator

The screenshot displays the Block Factory interface with three tabs: Block Factory, Block Exporter, and Workspace Factory. The Block Factory tab is active, showing a Block Library on the left and a Preview area on the right. The Block Library lists various properties for a block: name (block_type), inputs (dummy input), fields (left), text (My Custom Block), automatic inputs, top+bottom connections, tooltip, help url, top type, bottom type, and colour (hue: 230°). The Preview area shows a blue block labeled 'My Custom Block'. Below the Preview area, the Block Definition is shown in JSON format, and the Generator stub is shown in JavaScript format.

Block Factory | Block Exporter | Workspace Factory

Block Library | Save "block_type" | Delete "block_type" | Preview: LTR

Input
Field
Type
Colour

name: block_type
inputs: dummy input
fields: left | text: My Custom Block
automatic inputs
top+bottom connections
tooltip: ""
help url: ""
top type: any
bottom type: any
colour: hue: 230°

My Custom Block

Block Definition: JSON

```
{
  "type": "block_type",
  "message0": "My Custom Block",
  "previousStatement": null,
  "nextStatement": null,
  "colour": 230,
  "tooltip": "",
  "helpUrl": ""
}
```

Generator stub: JavaScript

```
Blockly.JavaScript['block_type'] = function(block) {
  // TODO: Assemble JavaScript into code variable.
  var code = '...;\n';
  return code;
};
```









Features

Type checking

The image shows a Scratch-style code editor with a grid background. On the left is a vertical sidebar with categories: Logic, Loops, Math, Text, Lists, Colour, Variables, Functions, TinyGo, MaQueen, Sensors, Gopherbot, and Net/HTTP. The main workspace contains three code blocks:

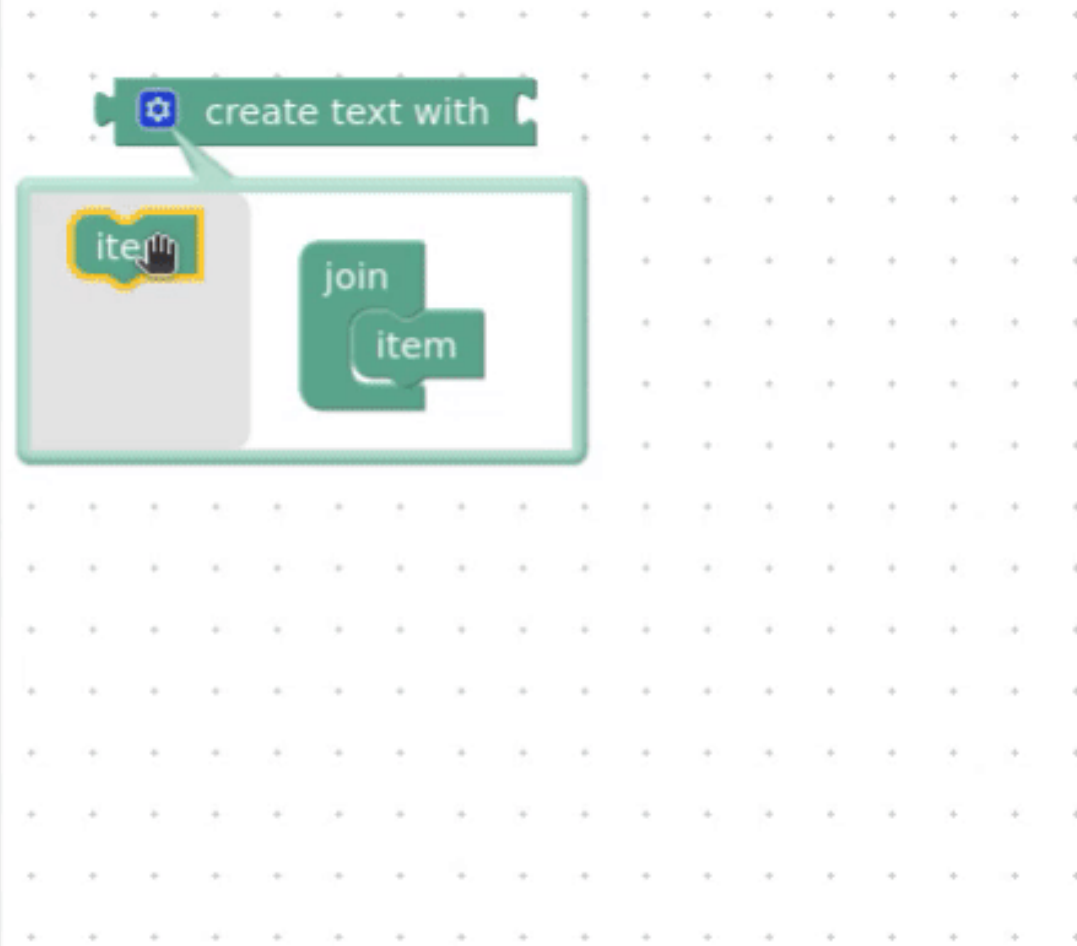
- A purple block labeled "set stringVariable to" connected to a green block labeled "Text value".
- A purple block labeled "set numberVariable to" connected to a blue block labeled "123".

Conditionals

	Logic
	Loops
	Math
	Text
	Lists
	Colour
	Variables
	Functions
	TinyGo
	MaQueen
	Sensors
	Gopherbot
	Net/HTTP

Lists

- Logic
- Loops
- Math
- Text
- Lists
- Colour
- Variables
- Functions
- TinyGo
- MaQueen
- Sensors
- Gopherbot
- Net/HTTP



In-line documentation

The image displays a block palette on the left and a workspace on the right. The palette is organized into several categories, each with a colored header: Logic (blue), Loops (green), Math (dark blue), Text (teal), Lists (purple), and Colour (brown). Below these are 'Variables' (maroon) and 'Functions' (purple). The bottom section of the palette lists hardware and protocols: TinyGo, MaQueen, Sensors, Gopherbot, and Net/HTTP. The workspace contains two blocks: a 'set stringVariable to' block with a dropdown arrow and a value of 'Text value', and a 'set numberVariable to' block with a dropdown arrow and a value of 123.

Logic
Loops
Math
Text
Lists
Colour
Variables
Functions
TinyGo
MaQueen
Sensors
Gopherbot
Net/HTTP

set stringVariable to "Text value"

set numberVariable to 123

Translations

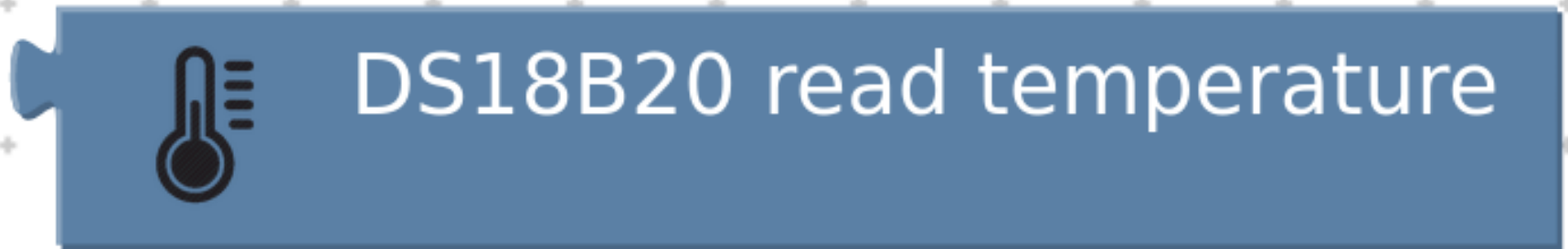
The image shows a Scratch code editor with a sidebar on the left containing the following categories: Logic, Loops, Math, Text, Lists, Colour, Variables, and Functions. The main workspace contains the following code blocks:

- A green **repeat** block with the number **10** and the text **times**.
- A blue **do** block nested inside the repeat block.
- A blue **if** block nested inside the do block, with a dropdown menu set to **true**.
- A green **do** block nested inside the if block.
- A purple **print** block nested inside the do block, with a dropdown menu set to **variable**.

Colors by categories

Logic	+	+	+	+	+	+	+	+	+	+	+	+
Loops	+	+	+	+	+	+	+	+	+	+	+	+
Math	+	+	+	+	+	+	+	+	+	+	+	+
Text	+	+	+	+	+	+	+	+	+	+	+	+
Lists	+	+	+	+	+	+	+	+	+	+	+	+
Colour	+	+	+	+	+	+	+	+	+	+	+	+
Variables	+	+	+	+	+	+	+	+	+	+	+	+
Functions	+	+	+	+	+	+	+	+	+	+	+	+

Images



Easy to copy, fun to share

Your programs looks good on paper, easier to copy by a student than just text (highlighting built-in).



Hide complexity

(as much as you want)

Limitations

- not everything supported (yet), need to create a block for it
- probably worse for vision impaired people or screen reader users
- Go static typing is complicated
- make a lot of decisions on behalf of the user (variable names, all numbers are int32,...)
- ugly code sometimes (because we're hiding complexity from the user)
- not much documentation, a bit hard to debug

Links

- [Blockly TinyGo](https://github.com/conejoninja/blockly-tinygo) (https://github.com/conejoninja/blockly-tinygo)
- [Gopherino](https://github.com/conejoninja/gopherino) (https://github.com/conejoninja/gopherino)
- [Shrimp tank project](https://github.com/conejoninja/shrimp-tank) (https://github.com/conejoninja/shrimp-tank)
- [Fermyon SPIN WASM Worker](https://github.com/conejoninja/spinworker) (https://github.com/conejoninja/spinworker)
- [TinyGo](https://tinygo.org/) (https://tinygo.org/)

Thank you

Daniel Esteban

<https://social.tinygo.org/@conejo/> (<https://social.tinygo.org/@conejo/>)

