Willie: Programación funcional reactiva para robots con bajas capacidades de cómputo

Proyecto de Grado - Facultad de Ingeniería - Universidad de la República

Guillermo Pacheco

Tutores: Marcos Viera, Jorge Visca, Andrés Aguirre

Agenda

- Objetivos
- Motivación
- Introducción a la Programación funcional reactiva
- Plataformas de Hardware
- Solución
- Trabajo Futuro

Objetivos

- Permitir el desarrollo de robots autónomos utilizando plataformas con bajas capacidades de cómputo.

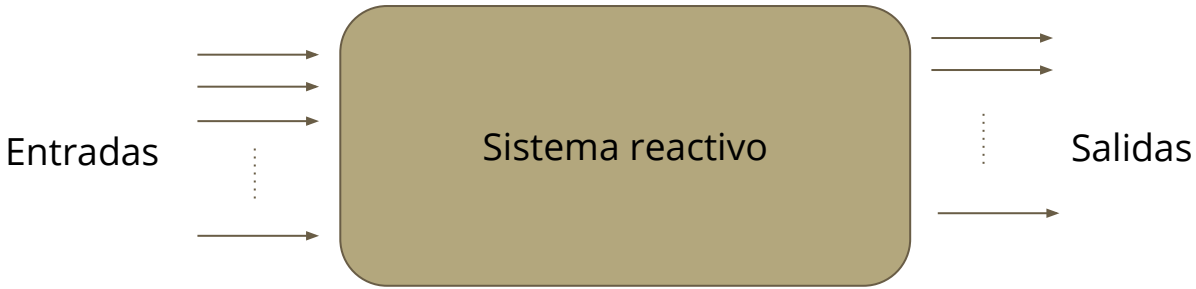
- Desarrollar herramientas para utilizar con fines educativos.

Motivación

- No existen lenguajes de alto nivel para crear dispositivos autónomos con bajas capacidades de cómputo.
- Lenguajes existentes son ***imperativos*** y de bajo nivel.
- Concurrencia en robótica.
- Se necesita mayor abstracción para usuarios inexpertos.

FRP - Sistemas reactivos

- Son aquellos que interactúan con el ambiente, intercalando entradas y salidas dependientes del tiempo.
- En robótica:
  - Entradas: Sensores. Ej: Sensor de distancia, de luz, interruptor.
  - Salidas: Actuadores. Ej: Motores, Luz, Sonido



FRP - Programación funcional

- Utiliza **funciones** en lugar de una secuencia de instrucciones.
- Las funciones siempre retornan el mismo resultado sin efectos secundarios.
- Un programa es el resultado de la composición de funciones.
- Declarativo. Se enfoca en *¿Qué?* en lugar de *¿Cómo?*

Programación funcional reactiva (FRP)

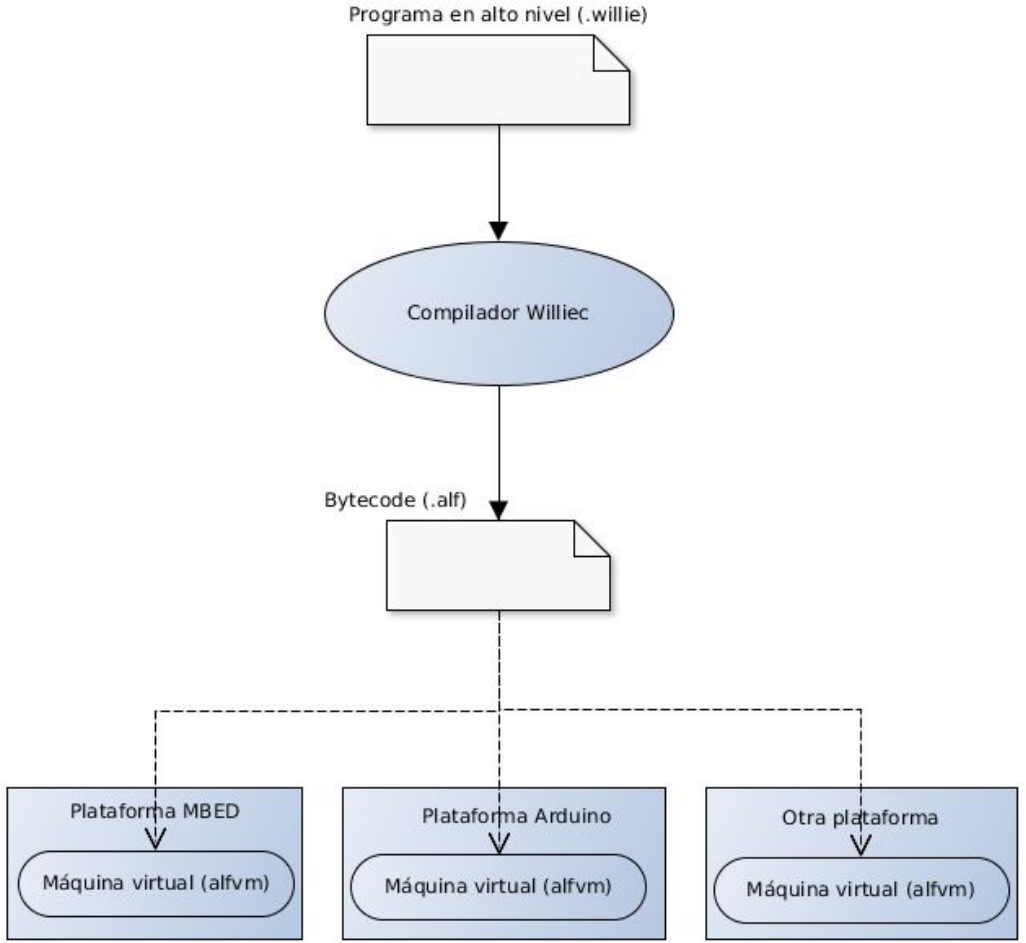
- Es construir ***sistemas reactivos*** utilizando ***programación funcional***.
- Simplifica problemas de concurrencia.
- Todos los ***valores*** dependientes del ***tiempo*** son **señales**
  - type Signal a = Time → a
- Para manipular señales se necesitan funciones sobre señales:
  - type SF a b = Signal a → Signal b
- Se puede crear un conjunto de **Combinadores**

Plataformas de Hardware

- Arduino
  - Se utilizan para prototipar.
  - Flash: Entre 16 KB y 512 KB. RAM: De Entre 1 KB y 96 KB.
  - En general: Flash: 32 KB. RAM: 2 KB.
- Mbed
  - Se utilizan para prototipar.
  - Flash: Entre 16 KB y 512 KB. RAM: De Entre 1 KB y 96 KB.
  - En general: Flash: 512 KB. RAM: >32 KB.
- Robotis
  - Kits robóticos completos. No está diseñado para ser extendido.
- Butiá
  - Kits robóticos desarrollados en FING para utilizar en educación.
  - Placa USB4Butia. Flash: 32 KB. RAM: 2 KB.

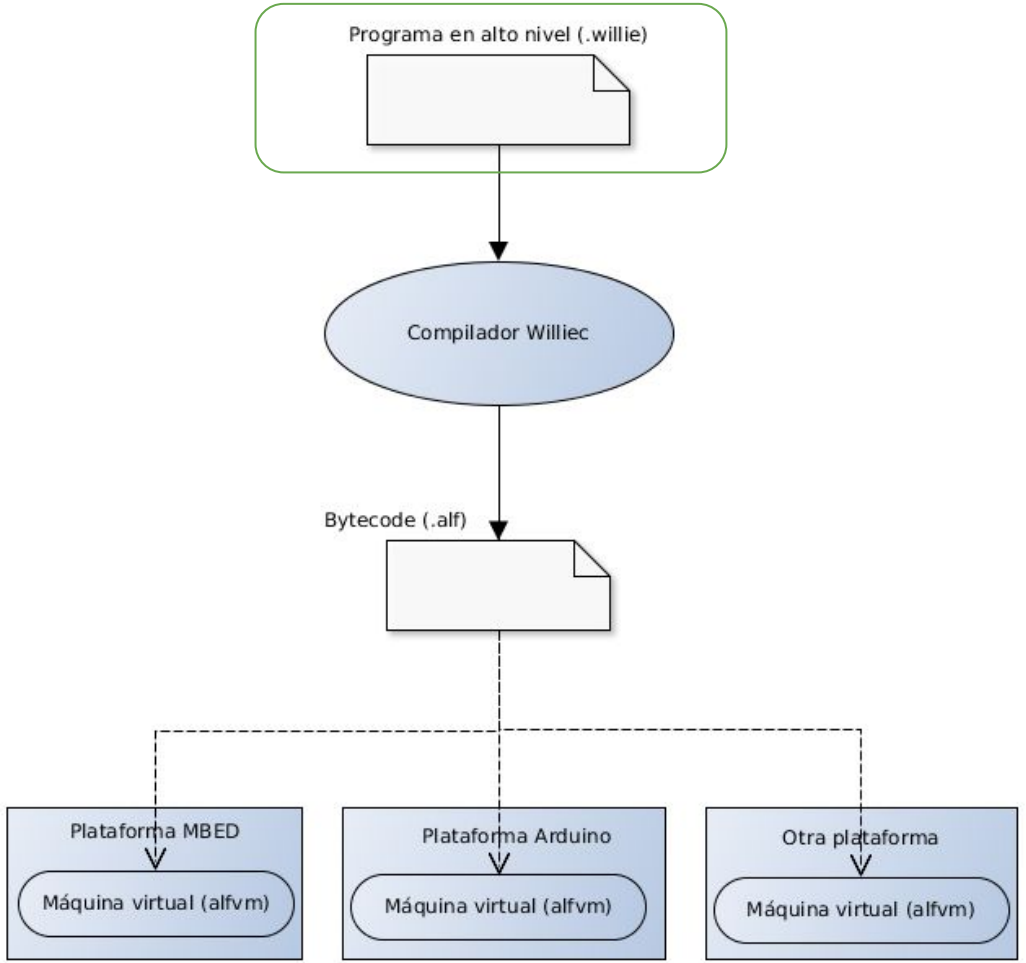
Solución

- Multiplataforma.
- Lenguaje ***Willie***
- Lenguaje ***Alf***
- Compilador ***Williec***
- Máquina virtual (***Alfvm***)



Solución

- Multiplataforma.
- Lenguaje ***Willie***
- Lenguaje ***Alf***
- Compilador ***Williec***
- Máquina virtual (***Alfvm***)



Willie

- Lenguaje de alto nivel, funcional reactivo.
- Funciones:
  - <nombre> <arg1> ... <argN> = <expresión>
  - Ej: sumar a b = a + b
- Constantes:
  - Es una función sin argumentos.
  - Ej: VELOCIDAD = 100
- Un programa se construye con un conjunto de funciones, más un bloque que indica cómo se procesan las entradas, se aplican combinadores de FRP y se envían señales a las salidas.

Willie - Combinadores de FRP

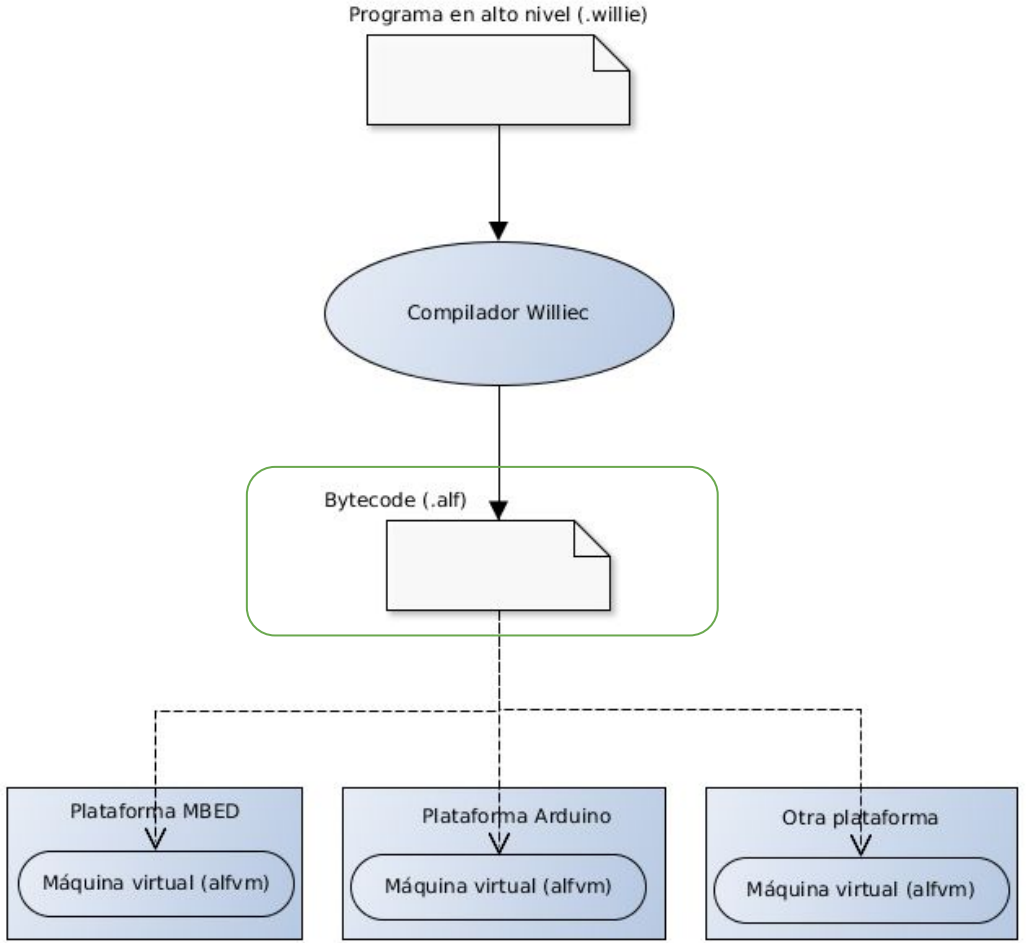
- Combinadores FRP:
  - lift: Aplica una función a una señal y construye una nueva señal.
    - lift :: (a→ b)→ Signal a→ Signal b
  - lift2: Aplica una función binaria a dos señales, y construye una nueva señal.
    - lift2 :: (a→ b→ c)→ Signal a→ Signal b→ Signal c
  - folds: Aplica una función binaria a una señal, y un valor acumulado.
    - folds :: (a→ b→ b) → b→ Signal a → Signal b
- Entrada/Salida
  - read: Crea una señal a partir de una entrada.
  - output: Envía los valores de una señal a una salida.

Willie - Ejemplo

```
INPUT_DISTANCE = 1
OUTPUT_ENGINE = 1

distanceToSpeed n = if (n < 30) then 0 else 100

do {
  distance <- read INPUT_DISTANCE,
  speed <- lift distanceToSpeed distance,
  output OUTPUT_ENGINE speed
}
```

Solución

- Multiplataforma.
- Lenguaje ***Willie***
- Lenguaje ***Alf***
- Compilador ***Williec***
- Máquina virtual (***Alfvm***)

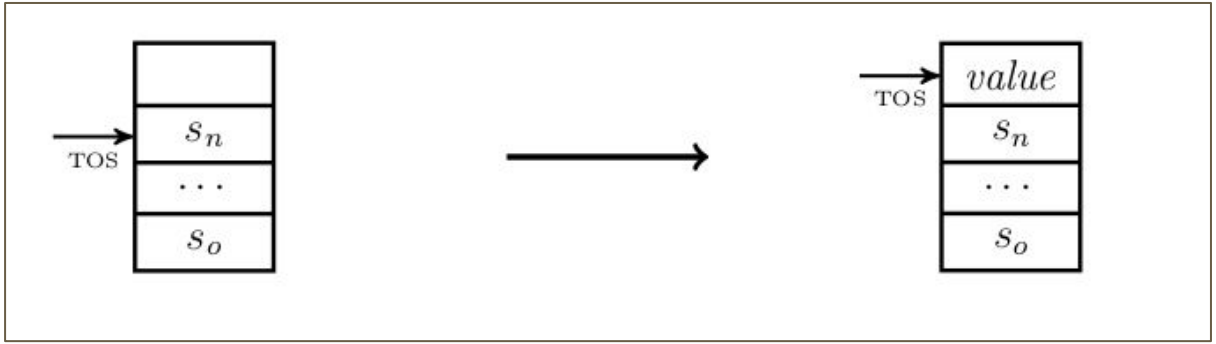


Alf

- Lenguaje de bajo nivel (Bytecode).
- Cuenta con instrucciones específicas para simplificar las operaciones de FRP.
- Funciones:
  - call f
  - ret
  - load_param i
- Combinadores:
  - lift id, src f
  - lift2 id, src1 src2 f
  - folds id, src f
- Entrada/Salida
  - read id
  - write id
- Comparación/control del flujo
  - jump
  - jump_false
  - cmp_eq
  - cmp_neq
  - cmp_gt
  - cmp_lt

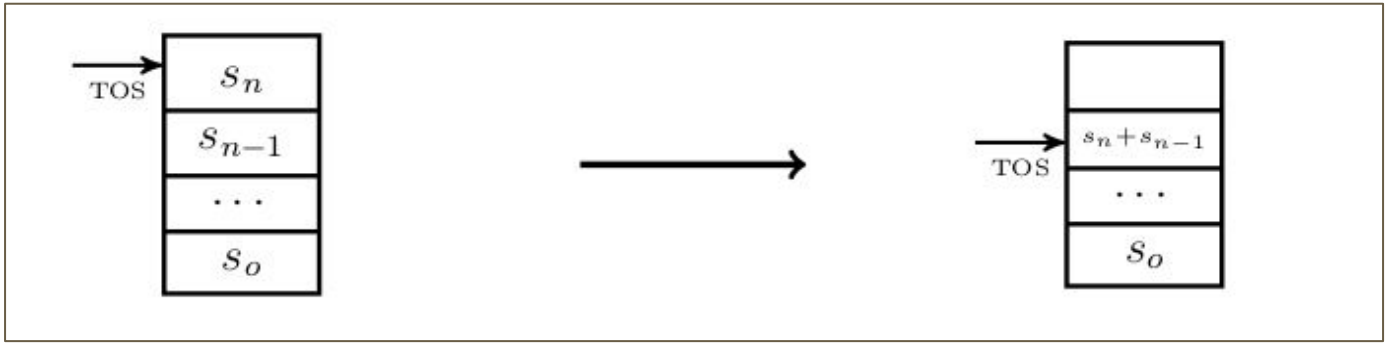
Alf

- Operaciones aritméticas
  - add
  - sub
  - div
  - mul
  - op_and
  - op_or
  - op_not
- Operaciones del stack
  - push value
- Control de VM
  - halt

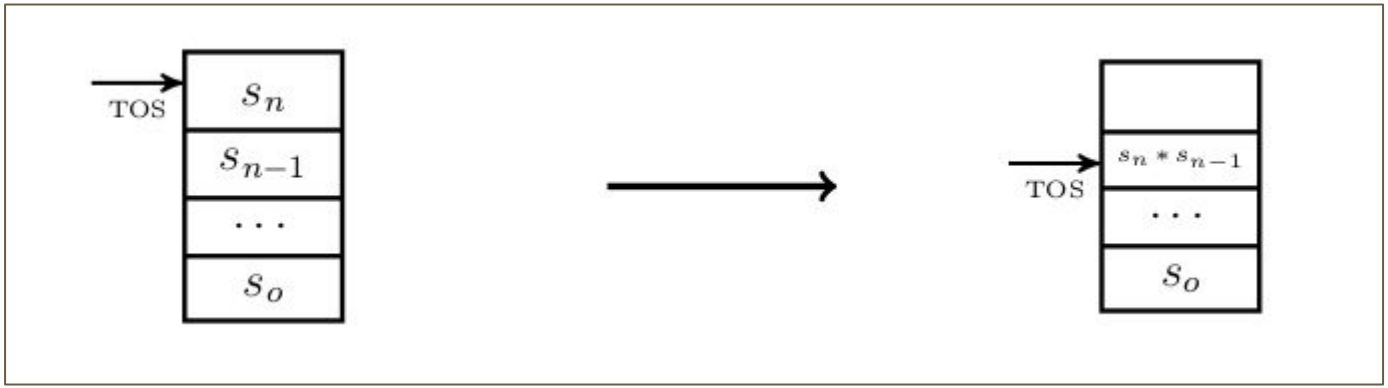


Alf - Operaciones aritméticas

- add:

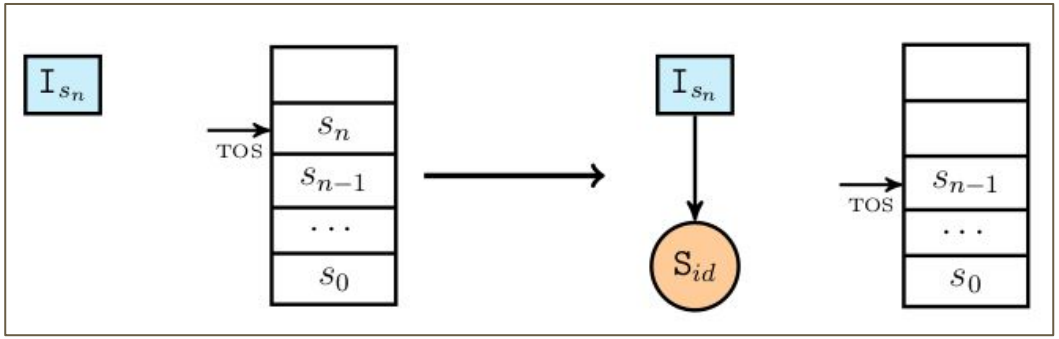


- mul:

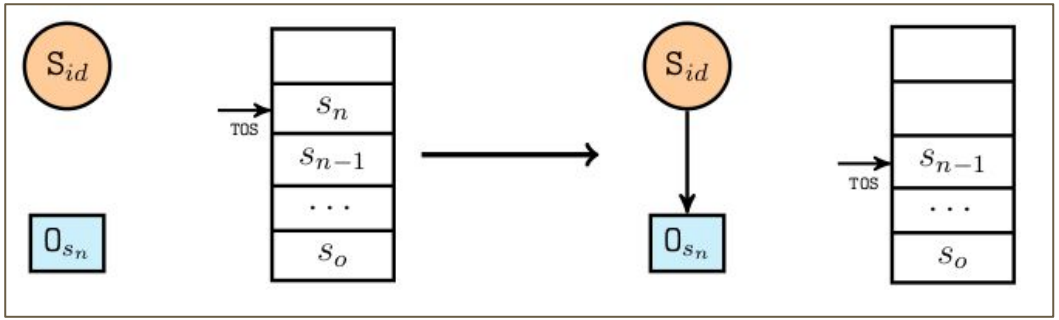


Alf - Entrada/Salida

- read id:

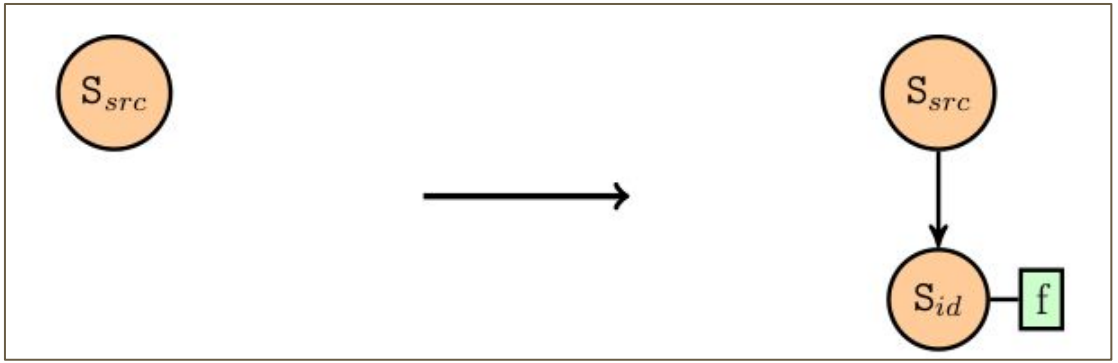


- output id:

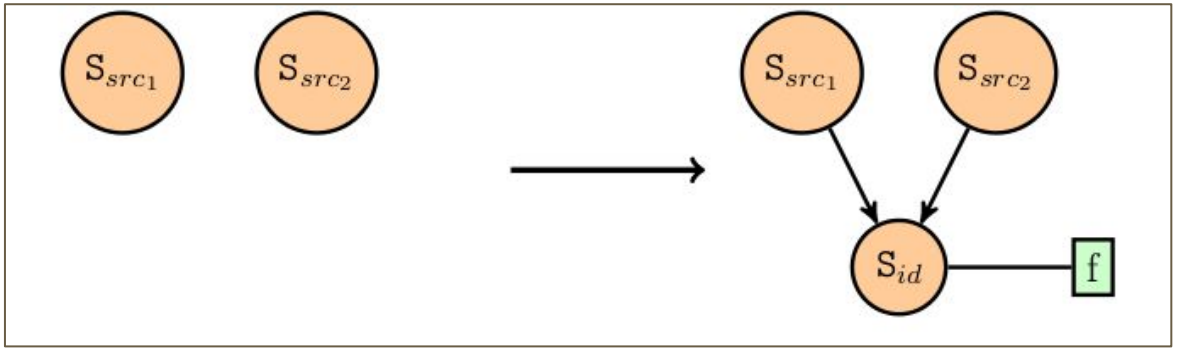


Alf - Combinadores FRP

- lift id, src f:

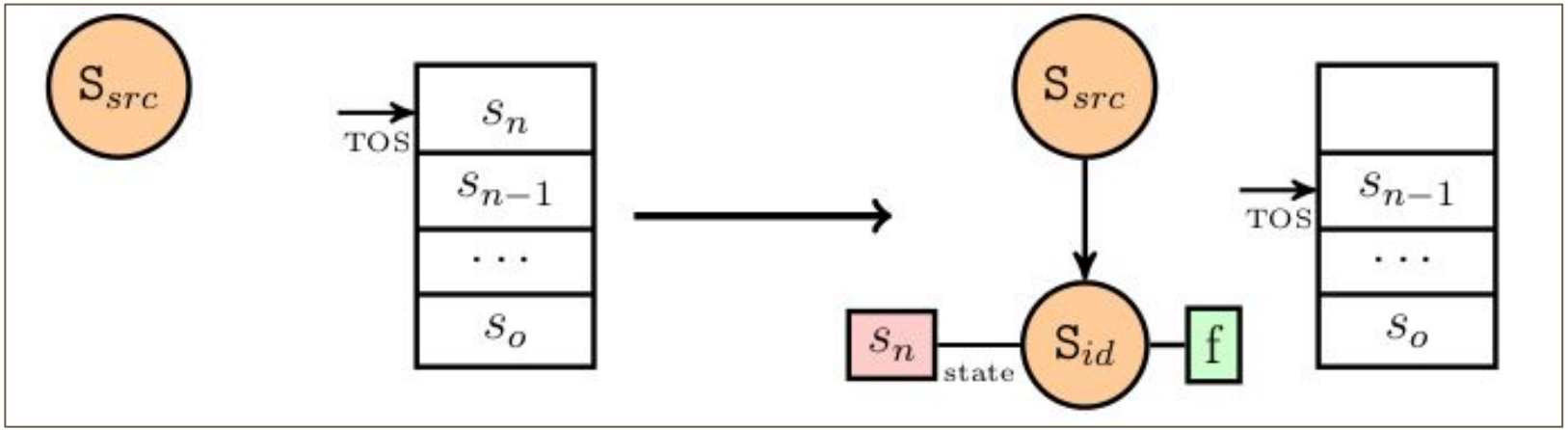


- lift2 id, src1 src2 f:



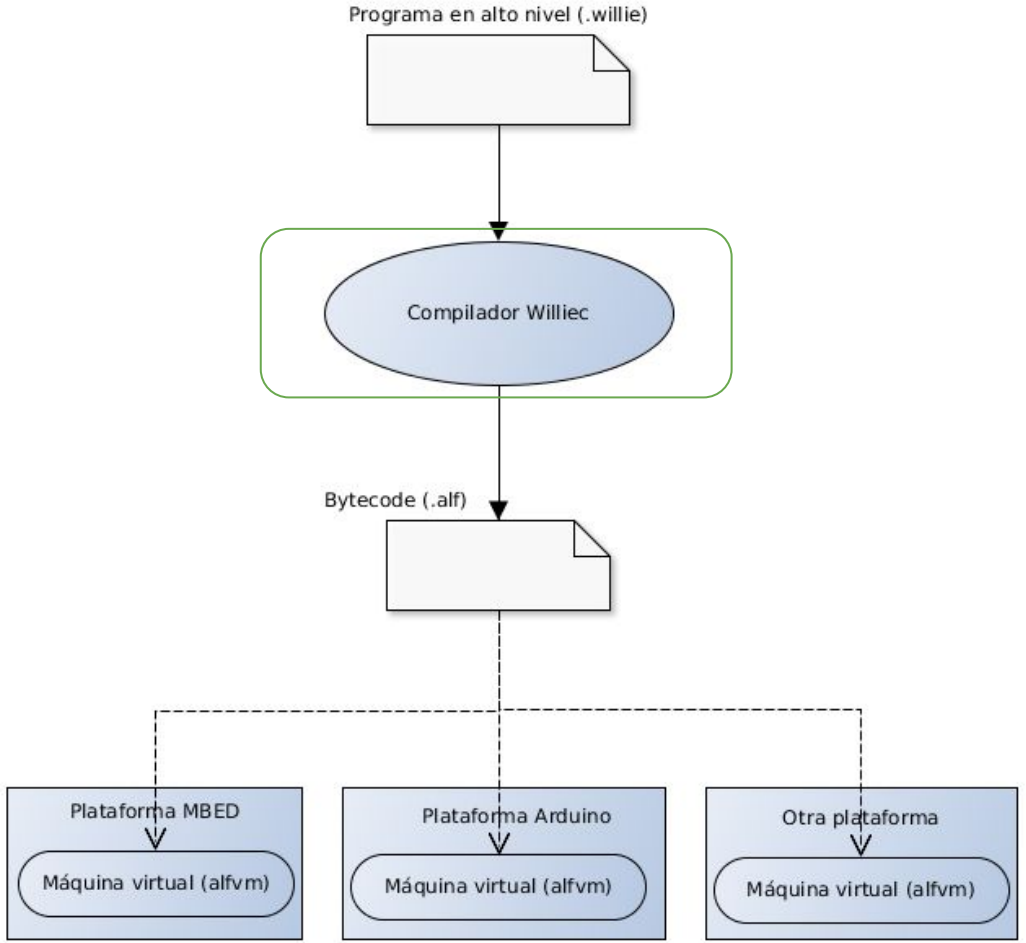
Alf - Combinadores de FRP

- folds id, src f



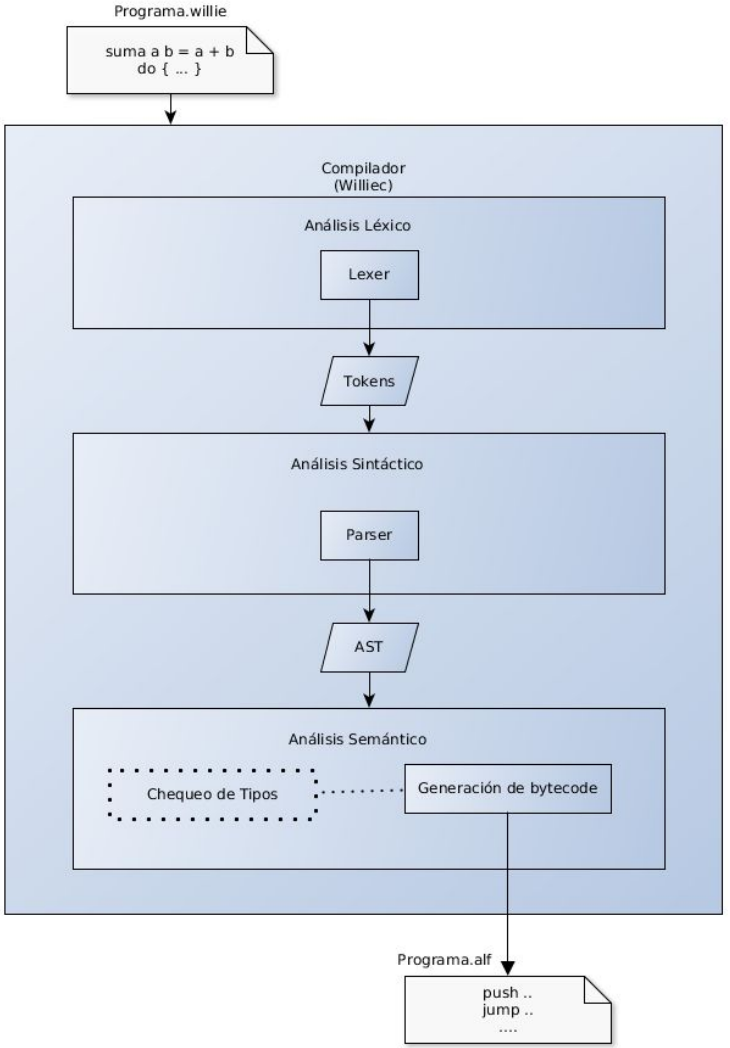
Solución

- Multiplataforma.
- Lenguaje ***Willie***
- Lenguaje ***Alf***
- Compilador ***Williec***
- Máquina virtual (***Alfvm***)



Compilador

- Traduce un programa ***Willie*** al lenguaje ***Alf***
- Escrito en ***Haskell***
- Análisis Léxico:
  - Produce tokens.
- Análisis Sintáctico:
  - Produce AST.
- Análisis Semántico:
  - Chequeo de tipos (No implementado)
  - Generación de código ***Alf***



Ejemplo de compilación

```
INPUT_DISTANCE = 1
OUTPUT_ENGINE = 1

distanceToSpeed n = if (n < 30) then 0 else 100

do {
  distance <- read INPUT_DISTANCE,
  speed <- lift distanceToSpeed distance,
  output OUTPUT_ENGINE speed
}
```

```
0: call
1: 10
2: read 1
3: lift 0
4: 1
5: 16
6: call
7: 13
8: write 0
9: halt
```

```
10: push
11: 1
12: ret
13: push
14: 1
15: ret
16: load_param 0
17: push
18: 30
19: cmp_lt
20: jump_false
21: 26
22: push
23: 0
24: jump
25: 28
26: push
27: 100
28: ret
```

Ejemplo de compilación

```
INPUT_DISTANCE = 1
OUTPUT_ENGINE = 1

distanceToSpeed n = if (n < 30) then 0 else 100
```

```
do {
  distance <- read INPUT_DISTANCE,
  speed <- lift distanceToSpeed distance,
  output OUTPUT_ENGINE speed
}
```

```
0: call
1: 10
2: read 1
3: lift 0
4: 1
5: 16
6: call
7: 13
8: write 0
9: halt
```

```
10: push
11: 1
12: ret
13: push
14: 1
15: ret
16: load_param 0
17: push
18: 30
19: cmp_lt
20: jump_false
21: 26
22: push
23: 0
24: jump
25: 28
26: push
27: 100
28: ret
```

Ejemplo de compilación

```
INPUT_DISTANCE = 1
OUTPUT_ENGINE = 1

distanceToSpeed n = if (n < 30) then 0 else 100

do {
  distance <- read INPUT_DISTANCE,
  speed <- lift distanceToSpeed distance,
  output OUTPUT_ENGINE speed
}
```

```
0: call
1: 10
2: read 1
3: lift 0
4: 1
5: 16
6: call
7: 13
8: write 0
9: halt
```

```
10: push
11: 1
12: ret
13: push
14: 1
15: ret
16: load_param 0
17: push
18: 30
19: cmp_lt
20: jump_false
21: 26
22: push
23: 0
24: jump
25: 28
26: push
27: 100
28: ret
```

Ejemplo de compilación

```
INPUT_DISTANCE = 1
OUTPUT_ENGINE = 1

distanceToSpeed n = if (n < 30) then 0 else 100

do {
  distance <- read INPUT_DISTANCE,
  speed <- lift distanceToSpeed distance,
  output OUTPUT_ENGINE speed
}
```

```
0: call
1: 10
2: read 1
3: lift 0
4: 1
5: 16
6: call
7: 13
8: write 0
9: halt
```

```
10: push
11: 1
12: ret
13: push
14: 1
15: ret
16: load_param 0
17: push
18: 30
19: cmp_lt
20: jump_false
21: 26
22: push
23: 0
24: jump
25: 28
26: push
27: 100
28: ret
```

Ejemplo de compilación

```
INPUT_DISTANCE = 1
OUTPUT_ENGINE = 1

distanceToSpeed n = if (n < 30) then 0 else 100

do {
  distance <- read INPUT_DISTANCE,
  speed <- lift distanceToSpeed distance,
  output OUTPUT_ENGINE speed
}
```

```
0: call
1: 10
2: read 1
3: lift 0
4: 1
5: 16
6: call
7: 13
8: write 0
9: halt
```

```
10: push
11: 1
12: ret
13: push
14: 1
15: ret
16: load_param 0
17: push
18: 30
19: cmp_lt
20: jump_false
21: 26
22: push
23: 0
24: jump
25: 28
26: push
27: 100
28: ret
```

Ejemplo de compilación

```
INPUT_DISTANCE = 1
OUTPUT_ENGINE = 1

distanceToSpeed n = if (n < 30) then 0 else 100

do {
  distance <- read INPUT_DISTANCE,
  speed <- lift distanceToSpeed distance,
  output OUTPUT_ENGINE speed
}
```

```
0: call
1: 10
2: read 1
3: lift 0
4: 1
5: 16
6: call
7: 13
8: write 0
9: halt
```

```
10: push
11: 1
12: ret
13: push
14: 1
15: ret
16: load_param 0
17: push
18: 30
19: cmp_lt
20: jump_false
21: 26
22: push
23: 0
24: jump
25: 28
26: push
27: 100
28: ret
```

Ejemplo de compilación

```
INPUT_DISTANCE = 1
OUTPUT_ENGINE = 1

distanceToSpeed n = if (n < 30) then 0 else 100

do {
  distance <- read INPUT_DISTANCE,
  speed <- lift distanceToSpeed distance,
  output OUTPUT_ENGINE speed
}
```

```
0: call
1: 10
2: read 1
3: lift 0
4: 1
5: 16
6: call
7: 13
8: write 0
9: halt
```

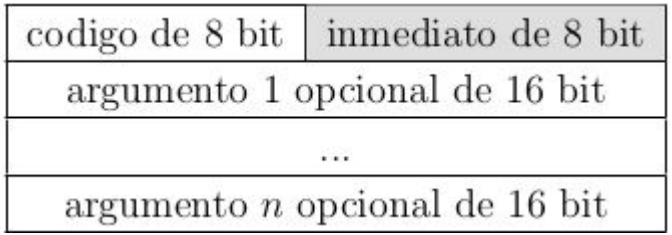
```
10: push
11: 1
12: ret
13: push
14: 1
15: ret
16: load_param 0
17: push
18: 30
19: cmp_lt
20: jump_false
21: 26
22: push
23: 0
24: jump
25: 28
26: push
27: 100
28: ret
```

Solución

- Multiplataforma.
- Lenguaje ***Willie***
- Lenguaje ***Alf***
- Compilador ***Williec***
- Máquina virtual (***Alfvm***)

AlfVM - Máquina Virtual

- C++
- Ejecuta código Alf
- Abstrae Entrada/Salida
  - Se implementan bibliotecas para cada periférico con interfaz estándar.
- Ancho de palabra de 16 bit
- Instrucciones de largo variable
- ~28 KB Flash en MBED



AlfVM - Pseudocódigo

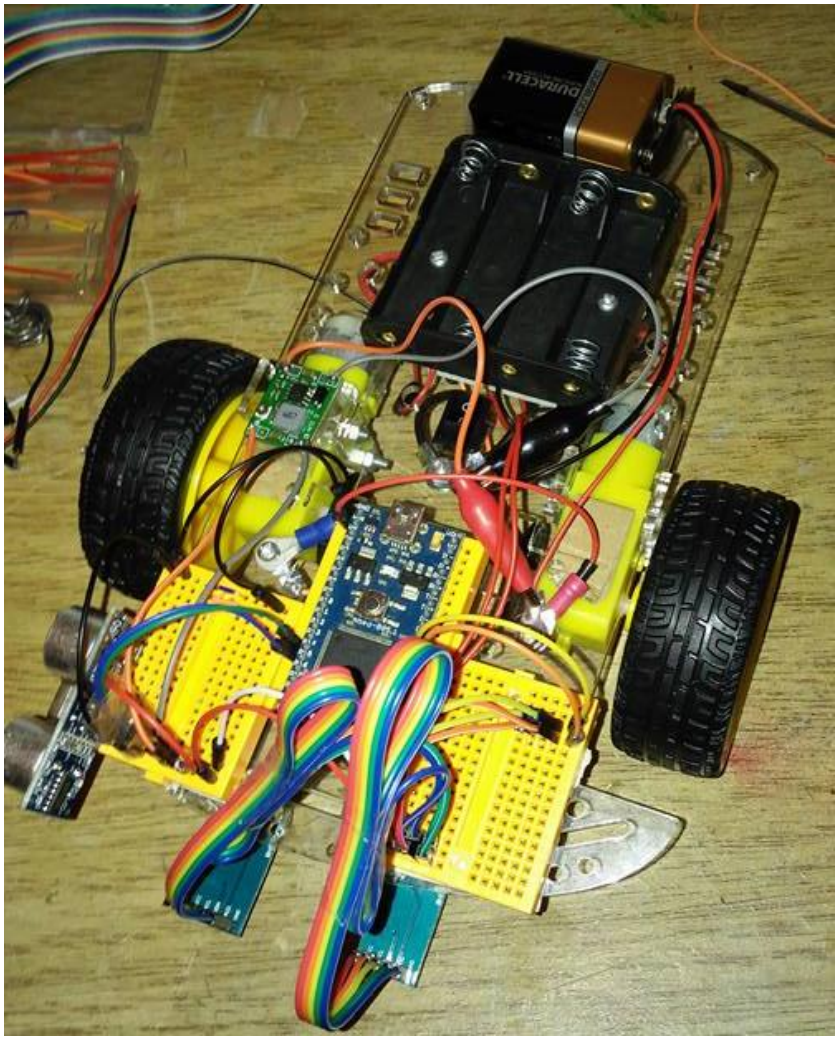
- Crear grafo de señales vacío. Crear pila vacía.
- Apuntar ***ip*** al inicio del código.
- Ejecutar hasta que ***ip*** se haga nulo.
- Por siempre:
  - Leer entradas
  - Actualizar señales conectadas a las mismas, marcar como listas.
  - Mientras hay señales listas para procesar:
    - Cargar valores en la pila.
    - Apuntar ***ip*** a inicio de la función asociada.
    - Ejecutar hasta que ***ip*** se haga nulo.
    - Actualiza señales conectadas a la misma.
  - Escribir salidas.

Caso de estudio

Desafío escolar SumoUY 2013 - Delivery-Bot

Caso de estudio

- Sensores:
  - Sensor de distancia.
  - Sensores de luz.
- Actuadores:
  - Motores. Izquierdo y derecho.

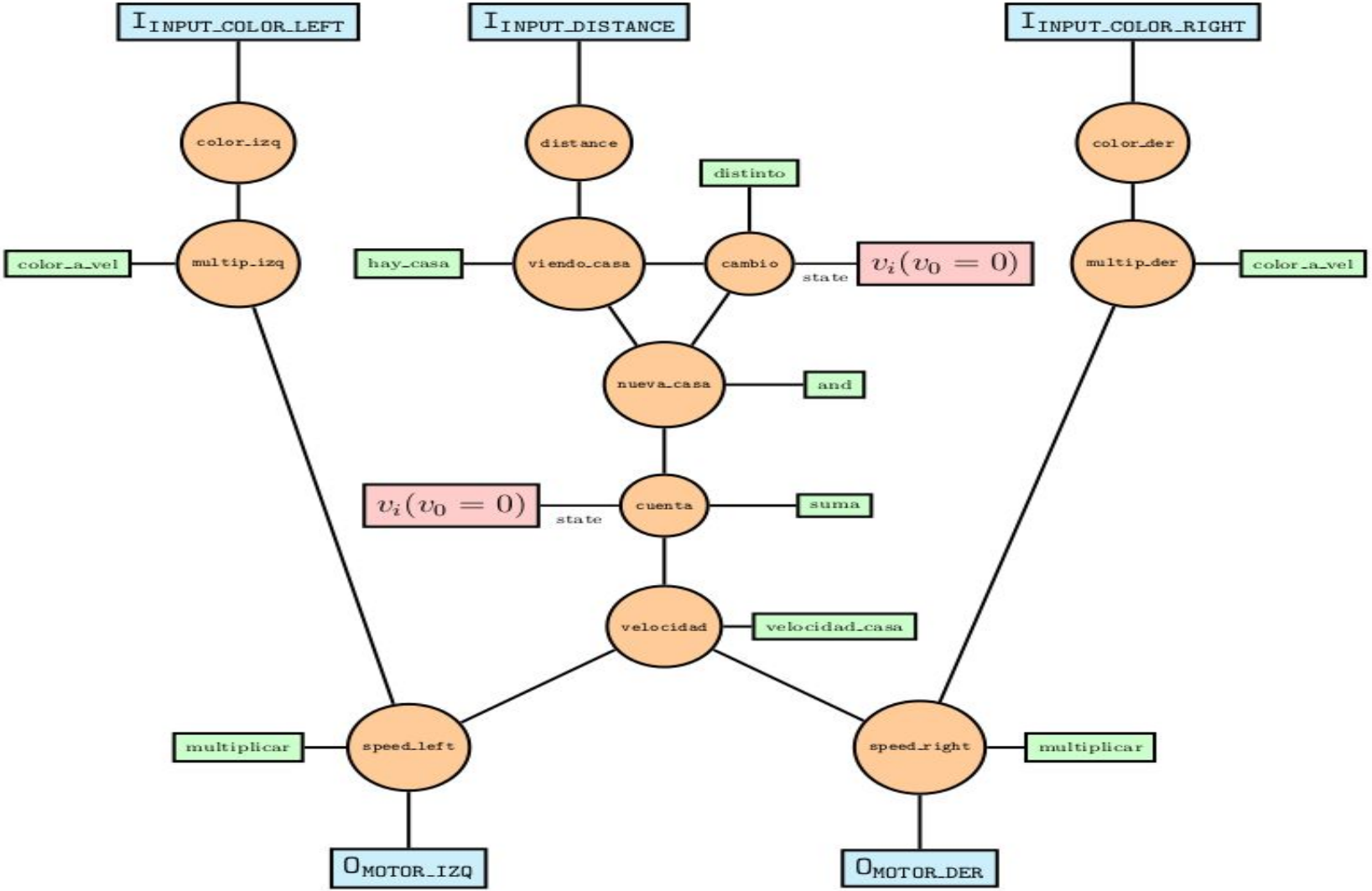


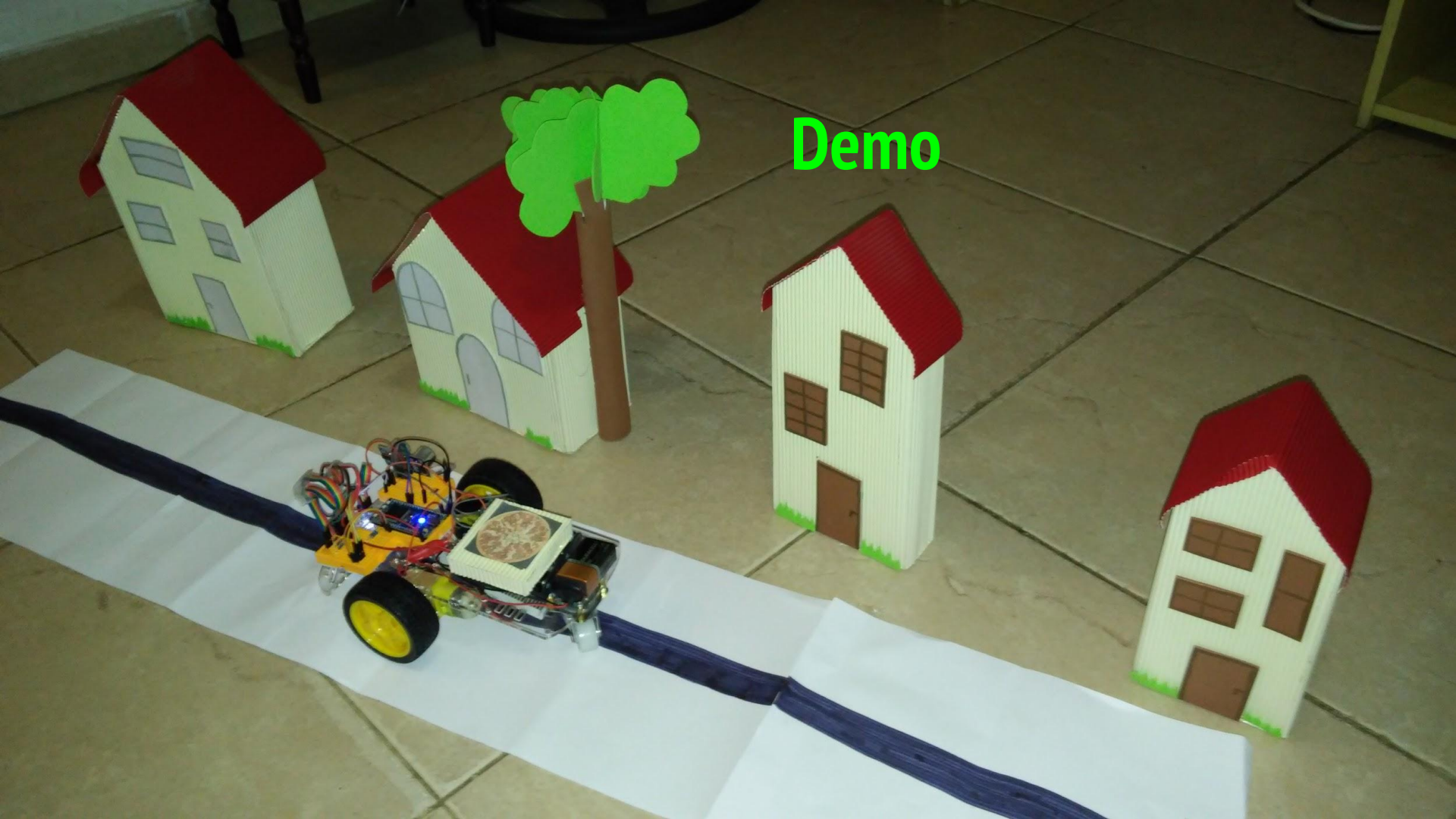
Caso de estudio - Programa Willie

```
INPUT_DISTANCE = 1
INPUT_COLOR_LEFT = 2
INPUT_COLOR_RIGHT = 3
OUTPUT_ENGINE_LEFT = 1
OUTPUT_ENGINE_RIGHT = 2

MIN_DISTANCE = 100
MIN_GREY = 50
HOUSE = 3
```

```
hay_casa d = if (d < MIN_DISTANCE) then 1 else 0
velocidad_casa num = if (num >= HOUSE) then 0 else 100
color_a_vel gris = if (gris > MIN_GREY) 2 else 1

and a b = if (a && b) then 1 else 0
suma a b = (a + b)
multiplicar a b = (a * b)
distinto a b = if (a /= b) then 1 else 0
```

```
do {
     distance <- read INPUT_DISTANCE,
     color_izq <- read INPUT_COLOR_LEFT,
     color_der <- read INPUT_COLOR_RIGHT,

     viendo_casa <- lift hay_casa distance,
     cambio <- folds distinto 0 viendo_casa,
     nueva_casa <- lift2 and viendo_casa cambio,
     cuenta <- folds suma 0 nueva_casa,
     velocidad <- lift velocidad_casa cuenta,

     multip_izq <- lift color_a_vel color_izq,
     multip_der <- lift color_a_vel color_der,

     speed_left <- lift2 multiplicar velocidad multip_izq,
     speed_right <- lift2 multiplicar velocidad multip_der,
     output MOTOR_IZQ speed_left,
     output MOTOR_DER speed_right
}
```


I_INPUT_COLOR_LEFT
I_INPUT_DISTANCE
I_INPUT_COLOR_RIGHT
color_izq
distance
color_der
distinto
color_a_vel
multip_izq
hay_casa
viendo_casa
cambio
state
$v_i(v_0 = 0)$
multip_der
color_a_vel
nueva_casa
and
$v_i(v_0 = 0)$
state
cuenta
suma
velocidad
velocidad_casa
multiplicar
speed_left
speed_right
multiplicar
O_MOTOR_IZQ
O_MOTOR_DER


Demo

Conclusiones

- Se diseñó el lenguaje ***Willie***.
- Se diseñó el lenguaje ***Alf*** y su máquina virtual.
- Se implementó un compilador y la máquina.
- Se logró que funcione con autonomía.

Trabajo Futuro

- Simulador.
- Depuración.
- Chequeo de Tipos.
- Eliminar restricciones de tipos.
- Expresividad de Willie, mejorar cómo estructurar y escalar. (Ej: Módulos.)
- Implementación de VM para otras plataformas.


¿Preguntas?