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

Guillermo Pacheco

14 de agosto de 2015

Resumen

El proyecto consiste en la creación de un lenguaje de programación para robots, cuyas capacidades de cómputo son limitadas. Para esto se escogió el paradigma de Programación Funcional Reactiva (*FRP*) el cuál permite expresar naturalmente reacciones a valores que varían en función del tiempo.

El objetivo es utilizarlo con fines educativos, por lo tanto debe ser simple y fácil de usar por usuarios inexpertos y no familiarizados con la electrónica y la informática.

Para resolver el problema, se dividió en dos etapas.

La primera consiste en la implementación de un compilador que traduce el lenguaje cuyo nivel es alto a un lenguaje de bajo nivel (*Bytecode*) más simple e interpretable.

La segunda etapa consiste en implementar una máquina virtual, que sea capaz de interpretar el lenguaje de bajo nivel. Por cada plataforma objetivo, es posible realizar una implementación de la máquina, lo que permite ejecutar los mismos programas en alto nivel, en diferentes plataformas.

El lenguaje se implementará como un DSL embebido en Haskell.

El diseño de la máquina consiste de un núcleo común capaz de interpretar instrucciones, y módulos bien definidos de entrada/salida los cuáles varían de una plataforma a otra. Ésto permite mayor portabilidad y extensibilidad.

Debe ser posible ejecutar programas en dicho lenguaje dentro de plataformas de hardware reducido. Considerando ésto el lenguaje de programación elegido para la implementación de la máquina virtual es C/C++.

De ésta forma se implementación de un lenguaje reactivo con las características deseadas, y una implementación modelo de una máquina virtual que permite su ejecución en una arquitectura objetivo deseada. La misma es fácil de mantener, portable y cuenta con suficiente flexibilidad para ser extendida.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Este documento desarrolla el proceso de construcción de herramientas que permitan programar robots utilizando el paradigma de programación funcional reactiva.

Para ello se define el lenguaje Willie de alto nivel, que permite expresar los comportamientos de un robot y sus interacciones. El lenguaje permite expresar los mismos en base a *Eventos* y relaciones entre ellos, que permiten capturar la naturaleza reactiva del dominio.

Éste lenguaje fue construído con fines académicos y de investigación, uno de los objetivos es que sea simple de utilizar e intuitivo.

Otro objetivo planteado es que debe ser posible programar con él sistemas robóticos embebidos en plataformas de hardware de baja capacidad de cómputo, y en lo posible bajo costo.

En los siguientes capítulos se introduce el concepto de programación funcional reactiva, junto con sus variantes y algunos ejemplos. Luego de decidir cuál de éstas variantes es más útil en nuestro dominio, se muestra la definición del lenguaje Willie y cuál es el proceso desde que se escribe un programa hasta que es ejecutado dentro de una plataforma.

Para cumplir con el objetivo de que la implementación sea portable, se separó la misma en dos fases. Los programas son compilados a código de menor abstracción, el cuál es ejecutado por una máquina virtual que definiremos. La misma puede ser portada a distintas plataformas.

Para finalizar se trata un caso de estudio completo, en el que se implementa un desafío robótico utilizado en la competencia SumoUY.

Capítulo 2

Programación Funcional Reactiva

2.1. Programación Funcional Reactiva

Tradicionalmente los programas reactivos se escriben como una secuencia de acciones imperativas. Existe un ciclo de control principal, donde en cualquier momento se leen valores de las entradas, se procesan, se mantiene un estado y se escriben valores en las salidas.

Los programas también suelen formarse por eventos y código imperativo que se ejecuta cuando un evento ocurre.

Dicho código imperativo suele hacer referencia y manipular un estado global desde diferentes rutinas.

Ésto tiene como consecuencia que como un valor puede ser manipulado desde diferentes lugares, puedan producirse problemas de concurrencia o llegar a un estado global inconsistente.

En el paradigma FRP no hay un estado compartido explícito, un programa se forma con valores dependientes del tiempo cuya única forma de ser modificados, es a partir de su definición, preservando la consistencia.

Definición 1. *Programa reactivo.*
Es aquel que interactúa con el ambiente, intercalando entradas y salidas dependientes del tiempo. Por ejemplo un reproductor de música, video juegos o controladores robóticos.

Difiere de los programas ***transformacionales*** *los cuáles toman una entrada al inicio de la ejecución y producen una salida completa al final. Por ejemplo un compilador.*

2.1.1. FRP Clásico

El paradigma FRP comenzó a ser utilizado por Paul Hudak y Conal Elliot en Fran (Functional Reactive Animation [1]) para crear animaciones interactivas de forma declarativa.

Su implementación está embebida en el lenguaje Haskell.

Los programas funcionales puros, no permiten modificar valores, sinó que una función siempre retorna el mismo valor dadas las mismas entradas, sin causar efectos secundarios.

Ésta propiedad es deseable para fomentar la reutilización del código pero no ayuda a mantener un estado. En la programación reactiva, es necesario mantenerlo por ejemplo para saber la posición del puntero del mouse en una interfaz, o para saber la ubicación de un robot.

En FRP para representar estado, éste se modela como valores dependientes del paso del tiempo.

Para ésto Fran define dos abstracciones principales, que son Eventos y Comportamientos [1].

Definición 2. *Comportamiento (Behaviour).*

Un comportamiento es una función que dado un instante de tiempo retorna un valor.

$$\boldsymbol{Behaviour}\ \alpha = \boldsymbol{Time} \rightarrow \alpha$$

Los comportamientos son muy útiles al realizar animaciones, para modelar propiedades físicas como velocidad, posición. Ésta abstracción permite que el desarrollador solo se ocupe de definir como se calcula un valor, sin implementar la actualización del mismo y dejando esos detalles al compilador.

Ejemplos de comportamientos aplicados a robótica pueden ser:

- *entrada* sensor de distancia, temperatura, video.
- *salida* velocidad, voltaje.
- *estado* explícito como saber que tarea se está haciendo.

Ejemplos de funciones que se pueden aplicar a los comportamientos incluyen:

[1] La definición de comportamientos en Fran no coincide con la definición de comportamiento normalmente utilizada en robótica. En bibliografía posterior, comportamientos fue cambiado por señales para evitar ésta ambiguedad.

- *Operaciones genéricas* Aritmética, integración, diferenciación
- *Operaciones específicas de un dominio* como escalar video, aplicar filtros, detección de patrones.

Definición 3. *Eventos. (Events)*

Los eventos representan una colección discreta finita o infinita de valores junto al instante de tiempo en el que cada uno ocurre.

$$\boldsymbol{Events}\ \alpha = [(\boldsymbol{Time}, \alpha)]$$

Los eventos se utilizan para representar entradas discretas como por ejemplo cuando una tecla es oprimida, cuando se recibe un mensaje o una interrupción.

También pueden ser generados a partir de valores de un comportamiento, como ser *Temperatura alta*, *Batería baja*, etc.

- *map* Obtiene un nuevo evento aplicando una función a un evento existente.
- *filter* Selecciona valores que son relevantes.

Frob Utilizando como base el trabajo realizado en Fran, se construyó Frob (Functional Robotics [2] [3]) un lenguaje funcional reactivo embebido en haskell, aplicado al dominio de la robótica.

En éste trabajo se introdujo el concepto de reactividad con el cuál utilizando los conceptos de comportamientos y eventos, éstos se combinan para realizar las tareas que un robot debe hacer. La estrategia que presenta, es de formalizar las tareas por medio de comportamientos, y conseguir que los comportamientos se modifiquen utilizando eventos y un conjunto de combinadores específicos.

Un ejemplo es, dado un robot, éste se tiene que mover a una velocidad constante hasta que se supere un tiempo máximo o se detecte un objeto. En Frob ésto se expresaría de ésta manera:

```
goAhead r t =
  (forward 30 'untilB'
    (predicate (time > t) .|.  predicate (frontSonar r < 20))
     -=> stop)
```

Lo que se leería cómo: "Para el robot r, moverse hacia adelante a velocidad 30, hasta que se exceda el tiempo t, o se detecte un objeto a menos de distancia 20. En ese momento detenerse."

La función `predicate` se utiliza para generar eventos a partir de comportamientos en base a una condición. El combinador `untilB` recibe dos comportamientos y un predicado, combina los dos comportamientos, retornando el primero mientras el predicado no se cumpla, y luego pasando al siguiente.

Otro punto importante de Frob, es que los periféricos del robot se asumen implementados, permitiendo que el desarrollador se concentre en la lógica específica de su problema, y no en resolver problemas del hardware. Además la lógica de leer las entradas, procesarlas y escribir las salidas es realizada por el flujo de control de Frob, y no de cada programa. En la implementación utilizaron un esquema simple, donde se leen todos los valores de todas las entradas y se procesan lo más rápido posible. Está claro que no es la mejor estrategia, porque puede causar demoras en el procesamiento y los datos leídos son válidos por un período corto de tiempo.

2.1.2. Real-Time FRP y Event-Driven FRP

Si bien el paradigma clásico de FRP permite expresar naturalmente programas reactivos, ésta expresividad no es gratuita, sinó que puede llevar a errores muy difíciles de encontrar en los programas. Un ejemplo de ésto es el llamado *time leak*, al implementarse en un lenguaje como Haskell, que cuenta con evaluación a demanda, los cálculos a demanda sobre los *Behaviours* puede que se retrasen y se acumulen, y al momento de necesitarse, el cálculo es tan largo que deja al programa sin memoria o afecta demasiado el tiempo de respuesta.

También pueden ocurrir *space leaks* donde un cálculo se retrasa indefinidamente, y la acumulación de los mismos consume el total de la memoria.

Como solución a ésto se propuso Real-Time FRP [4], una simplificación que garantiza mayor eficiencia. Utilizando el tipo de datos paramétrico Maybe [2], se realiza un isomorfismo entre Events y Behaviour.

Definición 4. *Isomorfismo en RT-FRP entre Events y Behaviour.*

$$\textbf{\textit{Events}}\ \alpha \approx \textbf{\textit{Behaviour}}\ (\textbf{\textit{Maybe}}\ \alpha)$$

Utilizando ésta simplificación, se agrupan las dos definiciones en un nuevo tipo llamado `Signal`.

[2] El tipo `Maybe` α en Haskell tiene dos valores posibles, `Just` α y `Nothing`.

Definición 5. *Señal (Signal).*

$$\textbf{\textit{Signal}}\ \alpha = \textbf{\textit{Time}} \rightarrow \alpha$$

Para garantizar las restricciones de que el tiempo y el espacio requerido por los programas es acotado, RT-FRP define un lenguaje base (lambda cálculo) de alto órden, y luego sobre esa base define un lenguaje reactivo que obliga a declarar las señales y sus conexiones. Sobre éste lenguaje restringido, se proveen demostraciones de que se cumplen las restricciones.

Event-Driven FRP Poco después de ésto, se propuso *Event-Driven FRP* [5], otra simplificación que añade como restricción que las señales sólo puedan ser modificadas mediante un evento. Aunque parezca muy restrictiva, la justificación de la misma es que los sistemas reactivos que se desean implementar están fuertemente guiados por eventos.

La propuesta de Event-Driven FRP era de llevar la programación reactiva a microcontroladores, para lo cuál define un lenguaje imperativo llamado *Simple C*, de tal forma que a partir del mismo sea muy simple compilarlo al lenguaje *C* o a las variantes que existen para microcontroladores.

En particular se implementó un prototipo que era capaz de generar código para el microcontrolador *PIC16C66* [6].

2.1.3. Arrows

Arrowized FRP (AFRP [7] [8]) intenta resolver los problemas de la FRP Clásica cambiando la forma en la que se crean los programas. En éste paradigma, tampoco se toman en cuenta por separado los eventos, se utiliza la misma definición de señal que en RT-FRP.

Definición 6. *Señal (Signal).*

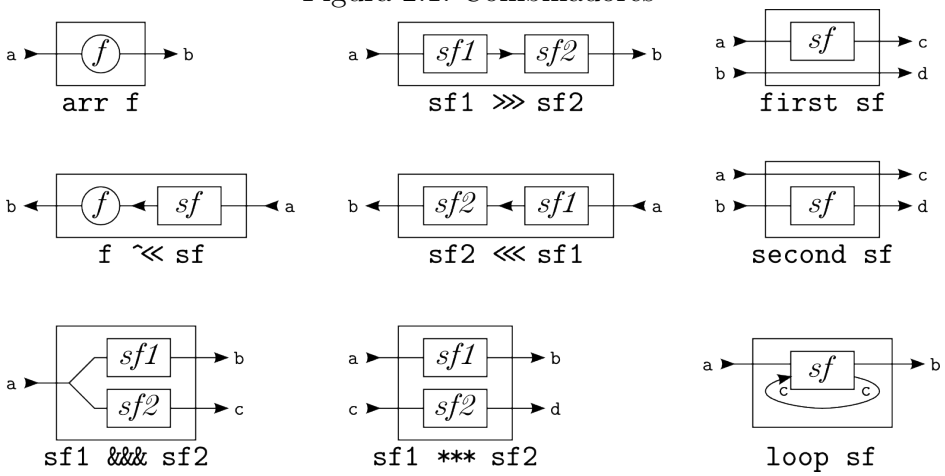
$$\textbf{\textit{Signal}}\ \alpha = \textbf{\textit{Time}} \rightarrow \alpha$$

En lugar de permitir que se manipulen las señales como valores de primera clase, ésto se prohibe y se define el concepto de *Signal Functions* (funciones sobre señales). El programador tendrá acceso sólo a las señales utilizándolas.

Definición 7. *Funciones sobre señales (Signal Functions).*

$$\textbf{\textit{SF}}\ \alpha\beta = \textbf{\textit{Signal}}\ \alpha \rightarrow \textbf{\textit{Signal}}\ \beta$$

Figura 2.1: Combinadores



Sin embargo, la representación de `SF` no es accesible al desarrollador, en lugar de eso se define un conjunto de funciones sobre señales primitivas y un conjunto de combinadores para componerlas.

AFRP está basado en *Arrows* [9] una generalización del concepto de *Monads*. En particular `SF` es una instancia de la clase *Arrow*.

Un programa es una `SF` global, compuesta por un conjunto de otras *Signal functions*, y un intérprete corre ésta instancia global.

En la figura 2.1 se puede ver un conjunto extenso de combinadores, sin embargo se puede definir un conjunto minimal utilizando sólo los combinadores `arr`, $\gg$ y `first`, aunque no es el único.

Yampa El framework Yampa [10] embebido en Haskell, define operadores simples para combinarlas y funciones que procesan las señales.

2.1.4. Elm: Programación funcional reactiva concurrente

Por último existe un lenguaje llamado Elm [11], creado para poder escribir aplicaciones web reactivas, de una forma funcional declarativa. En Elm, las entradas se asumen conocidas y son dadas, por ejemplo un click o el movimiento del mouse, o una tecla presionada.
Todos los valores son señales, el combinador $lift$ toma una señal y una función, y define otra señal resultado de la aplicación de la función sobre cada valor de la señal.
Para combinar más de una señal, se usa el combinador $lift_2$ o $lift_n$ con

$n \in N$.
En los casos que se desea tener memoria, por ejemplo llevar una cuenta de ocurrencias de una señal, o mantener un estado explícito, se utiliza el combinador $foldp$.

El combinador $foldp$ opera sobre una señal como el operador $fold$ sobre una lista. Dado un valor inicial y una función, aplica la función sobre cada valor, utilizando el último valor conocido como primer argumento.

2.1.5. Simplificación del paradigma

Al intentar implementar en un computador programas utilizando éste paradigma, nos encontramos con varias limitaciones. Una de ellas es que no es posible tener valores que se modifiquen de forma contínua, la capacidad de cómputo es finita, y no es posible ejecutar tareas al mismo tiempo, incluso en un entorno con paralelismo, el mismo está acotado a la cantidad de procesadores con los que se cuente.

Aunque el paradigma distinga Eventos de Comportamientos, se puede hacer una simplificación y asumir que todos los valores son Eventos. Los comportamientos en realidad serán una secuencia de cambios de valor, los eventos que dependan de un comportamiento serán notificados sólo en el momento que éste cambie.
Valores como el tiempo (reloj) y otros sensores como ser un sensor de distancia, entregarán valores periódicamente. Sería imposible que un programa reaccione a un valor contínuo, sin embargo, es muy fácil asumir que lo que nos importa del tiempo en realidad es dada una variación de tiempo cómo se debe comportar nuestro programa.
A su vez un sensor de distancia no nos puede entregar infinitos valores, y generalmente sólo importa poder recibir un valor nuevo en un período relativamente corto de tiempo.

Otra limitación en nuestro caso, al tratarse de robots con bajas capacidades de cómputo, es que la implementación del lenguaje no siempre tendrá más de un procesador disponible.
Para simular éste paralelismo se asumirá que el tiempo que se necesita para hacer cálculos es de una magnitud mucho menor al tiempo que lleva recibir un dato de una entrada, o enviar un dato a una salida. Se implementará una máquina capaz de correr dichos programas, la cuál esperará por valores en las entradas, y en base a los mismos, actualizará todos los eventos que dependan de ellas, y a su vez actualizará luego las salidas que dependan de éstos.
Cada actualización se hará secuencialmente hasta terminar, y en caso de con-

tar con más de un procesador, se asignarán en paralelo las actualizaciones utilizando varios hilos de ejecución.

2.1.6. Ventajas

2.2. Ventajas

La motivación para utilizar el paradigma presentado, es que un programa reactivo, si es escrito de forma iterativa, es susceptible a cometer errores de concurrencia al modificar valores en diferentes rutinas. A su vez, es difícil estructurar un programa iterativo para que reaccione rápidamente a los cambios.

Un patrón utilizado comunmente para estructurar un programa reactivo es el patrón `Observer`. En dicho patrón, un `Sujeto` puede ser observado por un `Observador`, éste último se subscribe al sujeto, y el sujeto notifica a todos sus observadores cuando su valor cambia.

La desventaja de hacer un programa reactivo siguiendo ese esquema, es que es difícil ver en que momento ocurren las actualizaciones de los observer. Si hay muchos observer suscritos a cambios de varios sujetos, se vuelve complejo mantener el código al tener tantas interacciones implícitas.

Al usar un lenguaje funcional, las interacciones son especificadas declarativamente, y se puede entender como un valor es formado a partir de otros. Para ver que es lo que está sucediendo en un programa se pueden obtener los valores en un instante de tiempo, como una fotografía y evaluar los errores o corroborar si el mismo es correcto.
De ésta manera si un programa tiene un error, se puede tomar una secuencia de instantes, como si fuera una grabación y entender donde está el problema, sin necesidad de seguir varios hilos de ejecución ni razonar sobre la concurrencia.

A su vez, no es necesario tener toda

Capítulo 3

Diseño

En esta sección se describe el diseño del lenguaje junto con su semántica. Luego se explica de que manera es traducido a un lenguaje de bajo nivel, más simple de interpretar, el cuál podrá ser interpretado por implementaciones de una máquina virtual en diferentes plataformas de hardware.

También se describirá las etapas de compilación, desde que se escribe un programa en alto nivel, hasta que el mismo es ejecutado en una plataforma objetivo.

El siguiente diagrama resume todas las etapas y componentes necesarios:

Figura 3.1: Etapas y componentes

3.1. Lenguaje de alto nivel

Un programa constará de dos secciones principales, una sección de declaraciones, donde se declararán funciones y una sección donde se aplicarán combinadores de programación funcional reactiva, especificando cómo son transformadas las señales para especificar el comportamiento de los robots.

3.1.1. Declaraciones

Las funciones se declaran de la siguiente manera.

```
nombre argumento_1 .. argumento_n = expresion
```

Donde una expresión puede ser un valor, una expresión aritmética (por ejemplo una suma o multiplicación), una aplicación de una función, o una

expresión condicional.

Para declarar un valor constante simplemente se escribe:

```
NOMBRE_CONSTANTE = valor
```

Ejemplo de declaración:

```
# fibonacci
fibo n = if (n < 2) then (1) else (fibo(n-1) + fibo(n-2))
```

3.1.2. Combinadores de FRP

Las entradas en el robot se puden manipular utilizando la función `read`. La misma crea una señal que cambia de acuerdo a los valores recibidos en la entrada.

Para aplicar una función a una señal, se aplica el combinador `lift` el cuál toma una función y una señal y produce una nueva señal resultado de la aplicación.

Luego se usa una lista de aplicaciones de `lift`, `lift2` y `folds` para combinar las señales hasta obtener los valores que conformarán la salida del robot y su estado.
La función `output` se utiliza para mapear esas señales a las salidas correspondientes.

La función `lift` combina una función y un valor dependiente del tiempo, y crea un nuevo valor dependiente del tiempo, es decir, aplica una función que transforma una señal en otra:

$lift :: (a \to b) \to Signal\ a \to Signal\ b$

Para combinar señales, se utiliza la función `lift2` que recibe dos señales y produce una nueva aplicando una función.

$lift2 :: (a \to b \to c) \to Signal\ a \to Signal\ b \to Signal\ c$

Utilizando `lift2` se pueden definir funciones `liftN` combinándola suscesivas veces, por ejemplo:

$lift3 :: (a \to b \to c \to d) \to Signal\ a \to Signal\ b \to Signal\ c \to Signal\ d$

$lift3\ f\ sa\ sb\ sc = lift2\ ((lift2\ f\ sa\ sb)\ sc)$

Para crear señales que no solo dependen de otra señal, sinó que dependen de ella y además de su historia (los valores anteriores), se define otro combinador `folds` que cuando recibe los valores de la señal, los combina y acumula. Éste combinador es análogo a la operación `fold` sobre listas.

$folds :: (a \to b \to b) \to b \to Signal\ a \to Signal\ b$

3.1.3. Ejemplo

En el siguiente ejemplo se muestra como haría para detener un robot cuando su sensor de distancia muestra un valor menor a un mínimo.

```
INPUT_DISTANCE = 1
OUTPUT_MOTOR = 1

MIN_DISTANCE = 30
FULL_SPEED = 100
STOP = 0

distanceToSpeed n = if (n < MIN_DISTANCE) then STOP else FULL_SPEED

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

Primero se definió una fuente de eventos llamada *distance*, de tipo *Event Number*.

```
  distance <- read INPUT_DISTANCE
```

La misma emitirá un evento con la distancia en *cm* leída por un sensor en el robot.

Luego, se quiere que cuando la distancia sea menor al valor *MINIMO* dado, el robot se detenga completamente, ésto quiere decir, que la velocidad sea 0, en otro caso el mismo se mueve a velocidad 100, la cuál asumimos

es una velocidad apropiada. Para ésto se crea una nueva fuente de eventos, definida a partir de *distance*, aplicandole la función `distanceToSpeed` a cada valor para obtener la velocidad.

```
speed <- lift distanceToSpeed distance
```

Ésta expresión, también es una fuente de eventos de tipo *Event Number*.

Finalmente, se aplica la función nativa `output` cuyo primer parámetro es el *id* de la salida, en éste caso es `OUTPUT_MOTOR` y su segundo parámetro es una fuente de eventos, en éste caso son velocidades.

3.2. Lenguaje de bajo nivel

Para presentar el lenguaje, primero defino las estructuras necesarias para explicar la semántica de cada instrucción.

1. *Inputs*

   Los valores leídos en las entradas de hardware se mapean en esta lista. Por ejemplo si el hardware cuenta con un botón, y el identificador del botón es i, su valor se representará con la notación:

   $Inputs_i$

   A cada entrada se le asocia un conjunto de rutinas que deben invocarse cuando se tenga un valor disponible en la entrada. El mismo puede ser vacío si no hay rutinas esperando por su valor. Queda a criterio de quien implementa la máquina si éstos valores deben ser guardados o descartados. A éste conjunto de rutinas lo denotamos cómo:

   $Callbacks_i$

2. *Nodes*

   Es utilizado para almacenar todas las transformaciones de señales. Cada nodo, tiene una lista de otros nodos que dependen de él y la posición del argumento por el que esperan. Además el nodo almacena el último valor calculado, y una lista de argumentos que le serán pasados, si son nuevos o no.

   Cada aplicación de `lift`, `lift2` o `folds` será mapeada en un nodo.

   Cada nodo $Node_i$ tiene una lista de nodos adyacentes que dependen de él.

3. *Outputs*

   Mapea señales en salidas de hardware, los valores son asignados con la operación `output`.

4. *Stack*

   Es una pila global, se utiliza para ejecutar operaciones, realizar cálculos, es único y global, y los hilos de ejecución no pueden guardar valores persistentes en él.

   Usaremos el símbolo TOS para referirnos al elemento en el tope de la pila y $Stack$ para referirnos a la pila.

5. *Ready*

   Es una *cola* que contiene punteros a los nodos listos para ser ejecutados.

6. *Dispatcher*

   El dispatcher, es quien implementa las acciones de la máquina. El mismo se encarga de recibir valores de los sensores, mapearlos a eventos en las entradas $Input_i$.

   Éstos eventos, serán recibidos por los nodos $Nodes$, cada $Node_i$ que espera por eventos, entrará en estado activo cuando todos los nodos por los que espera le envíen un evento. A su vez, el nodo en estado activo calcula un resultado y notifica a todos sus nodos adyacentes.

   El dispatcher, realizará implicitamente un orden topológico de los nodos, como $Nodes$ es un grafo acíclico, éste proceso es posible y termina, y cada salida cuenta con un valor, que será mapeado a los actuadores.

3.2.1. Instrucciones

A continuación se listan las instrucciones de bajo nivel más relevantes junto con su semántica.

- `lift id source_id function_pointer`

  Crea un nodo que recibe valores de la fuente de eventos $source_id$, les aplica la función $function_pointer$ y emite el resultado en la fuente de eventos id.
  El nodo se agrega a $Nodes$ y el dispatcher se encargará de aplicar la función siempre que sea necesario.

- `lift2 id source-id_1 source-id_2 function_pointer`

  Crea un nodo que recibe valores de $source - id_1$ y $source - id_2$, les aplica la función $function_pointer$ y emite el resultado en la fuente de eventos id.
  Cuando reciba un valor nuevo por cada uno de los que espera, el dispatcher aplica la función y emite el resultado.

- `read id input-id`

  Agrega un nodo a $Nodes$ que recibe valores de una entrada, y emite el resultado en la fuente de eventos id. Cuando el dispatcher reciba un cambio en la entrada, el nodo emitirá el resultado.

- `write index id`

  Crea un nodo $Output$ que recibe valores de la señal id y los emite en la salida número $index$. El nodo guarda dicho valor y el dispatcher decide en que momento debe escribirlo en la salida asociada. Se intentará que sea lo más instantáneo posible ya que el valor depende del tiempo y a mayor demora, menos correcto será el comportamiento.

3.3. Compilador

El compilador será el encargado de leer el programa de alto nivel y traducirlo al lenguaje de bajo nivel.

El programa resultado tendrá al principio las traducción de las instrucciones correspondientes al bloque `do`. Las mismas al ser ejecutadas armarán el grafo de señales del programa.
Al final del bloque, habrá una instrucción `halt` que cederá el control al *dispatcher*.
Luego estará el código correspondiente a cada función definida. Cada invocación a función, tendrá la referencia directa hacia la posición en el código de la misma.

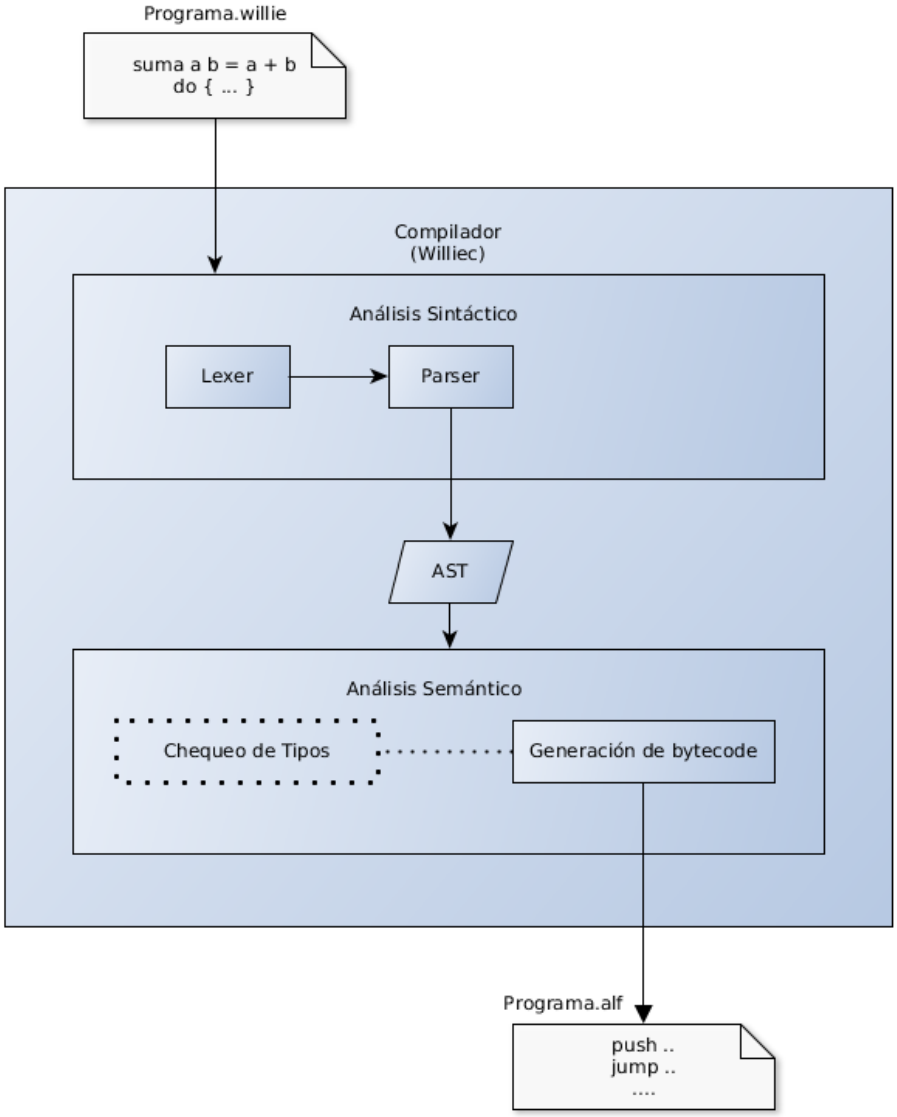
El compilador constará de dos etapas principales.

En la primer etapa lee el programa de alto nivel y genera un árbol de sintaxis abstracto (AST) del mismo. Dicha etapa es llamada análisis sintáctico.

Luego el ast se recorre y la segunda etapa es la generación del código de bajo nivel.

En la siguiente figura se puede ver la estructura más detallada del compilador.

Figura 3.2: Diagrama del compilador



3.3.1. Ejemplo de traducción

Dado el siguiente programa de ejemplo en alto nivel:

```
# Este programa de prueba muestra con un led si se esta frente a una casa.

#Inputs
sensor_distancia = 1
#Outputs
led = 1

hay_casa = \distancia -> (distancia < 100)

do {
  signal_distancia = read sensor_distancia,
  signal_viendo = lift hay_casa sensor_distancia,
  write led signal_viendo
}
```

El mismo se representaría en bajo nivel de la siguiente forma:

```
01:  push 1
02:  store 0 //sensor_distancia = 1
03:  push 1
04:  store 1 //led = 1
05:  read 0 0 //signal_distancia = read 0
06:  lift 1 0 hay_casa //signal_viendo = lift hay_casa sensor_distancia
07:  write 1 1 //write led signal_viendo
08:  halt
09:  load_arg 0 //hay_casa:
10:  push 100
11:  cmp_le //distancia < 100
12:  return
```

3.4. Máquina Virtual

La máquina virtual será la encargada de recibir el bytecode creado por el compilador, e interpretarlo en la plataforma que esté ejecutando. A diferencia del compilador, es necesario implementar una máquina virtual para cada arquitectura objetivo.

Por ejemplo, para ejecutar programas en un robot con un procesador *arduino*, debe existir una implementación de la máquina para ese modelo de *arduino*.

Al momento de implementar la máquina, se tomará en cuenta ésto para factorizar partes en común y sólo implementar por arquitectura, las partes que realmente sean diferentes como ser la comunicación con los periféricos de entrada/salida y las llamadas al sistema.

Capítulo 4

Implementación

En este capítulo se detalla la implementación de el compilador y la máquina virtual diseñadas para utilizar el lenguaje Willieen la plataforma elegida. También se explica cuál sería el mecanismo para portar la implementación a otra plataforma.

4.1. Compilador

El lenguaje utilizado para desarrollar el compilador fue *Haskell*. Las razones que llevaron a su elección son la portabilidad y la expresividad del mismo. Por un lado, el compilador es portable, ya que se puede compilar y ejecutar en diversos sistemas operativos utilizando el compilador *ghc*.

Por otro lado, la arquitectura del compilador es de tubos y filtros, algo que es natural expresar en un lenguaje funcional. La gramática se corresponde directamente con un tipo de datos así como el código generado.

Es usual realizar tareas de compilación en *Haskell*, y existen herramientas estándar para cada etapa. Para el análisis léxico, se utiliza *Alex* [12] y para parsear y generar la gramática se utiliza *Happy*. [13]

La primera fase del compilador utiliza las dos herramientas mencionadas, recibe el programa en lenguaje Willie(.willie) y genera un árbol de sintaxis abstracta (*AST*).

La segunda etapa, utiliza gramáticas de atributos (*Attribute Grammars*) definiendo atributos en el *AST*. Uno de dichos atributos será el código en bajo nivel, que será la salida de ésta etapa. Ésta salida se escribe en un archivo

(.alf) terminando el proceso de compilación.

Para utilizar el compilador, dado un archivo *Ejemplo.willie*, se ejecuta:

```
> williec < Ejemplo.willie > Ejemplo.alf
```

4.2. Máquina virtual

La máquina, deberá ejecutar el código de bajo nivel en una plataforma objetivo. El lenguaje de programación elegido para el desarrollo de la máquina virtual es *C++* ya que es posible compilarlo para casi cualquier plataforma objetivo. Además *C++* permite acceder a muy bajo nivel, y manipular a nivel de *bytes* las estructuras.

Existen dos limitaciones importantes a tener en cuenta, la primera es que el espacio de memoria varía en diferentes plataformas, por lo que se desea sea posible compilar la máquina aún con un espacio muy reducido. La segunda es que las plataformas varían en capacidades de *Entrada/Salida*, es importante que quien compila la máquina y arma un entorno tenga conocimiento de cómo disponer las mismas y qué limitaciones existen, por ejemplo: Cantidad de pines digitales o analógicos.

La implementación modelo, se hizo utilizando la plataforma *MBED LPC1768*, se puede encontrar documentación de la misma en [14] y en [15].

MBED es una plataforma pensada para colaborar mediante un entorno de desarrollo web, y compilador online, ese esquema de trabajo no es el más práctico para desarrollar la máquina virtual, por lo que se descargaron de la página de mbed, en [16] las herramientas de desarrollo para compilar offline.

El código de la máquina virtual está en el directorio /src/alfvm, para compilarlo se ejecuta:

```
> cd src/alfvm
> make
```

Ésto genera un archivo *mbed_alfvm.bin*. Para cargar la máquina en la placa *MBED*, alcanza con conectarla a un puerto USB y pegar el archivo en la carpeta /media/MBED.

Capítulo 5

Casos de estudio

En esta sección veremos un caso de estudio usado para verificar la implementación. El problema fue tomado de la competencia SumoUY [17], el mismo fue el desafío planteado a escolares en el año 2013.

5.1. Problema

Se desea implementar un robot autónomo móvil que sea capaz de hacer la entrega de un pedido en una casa determinada. El mismo debe moverse por un escenario e identificar las casas. Para recorrer la ruta de entrega, podrá valerse de una línea negra que representará la calle de la ciudad.

Las casas estarán ubicadas a un lado de la calle. En el recorrido se encuentran varias casas, el robot deberá entregar un pedido en la quinta casa por la que pase.

El robot deberá pasar por alto las casas anteriores y al llegar a la casa objetivo debe detenerse totalmente.

Para probar la solución, se armará un escenario que consiste de un piso blanco con una línea negra que puede tener curvas.

Al lado derecho de la línea se ubicarán cajas a menos de 30 centímetros representando las casas.

5.2. Solución

Se armó un robot móvil (5.1) que cuenta con 3 sensores:

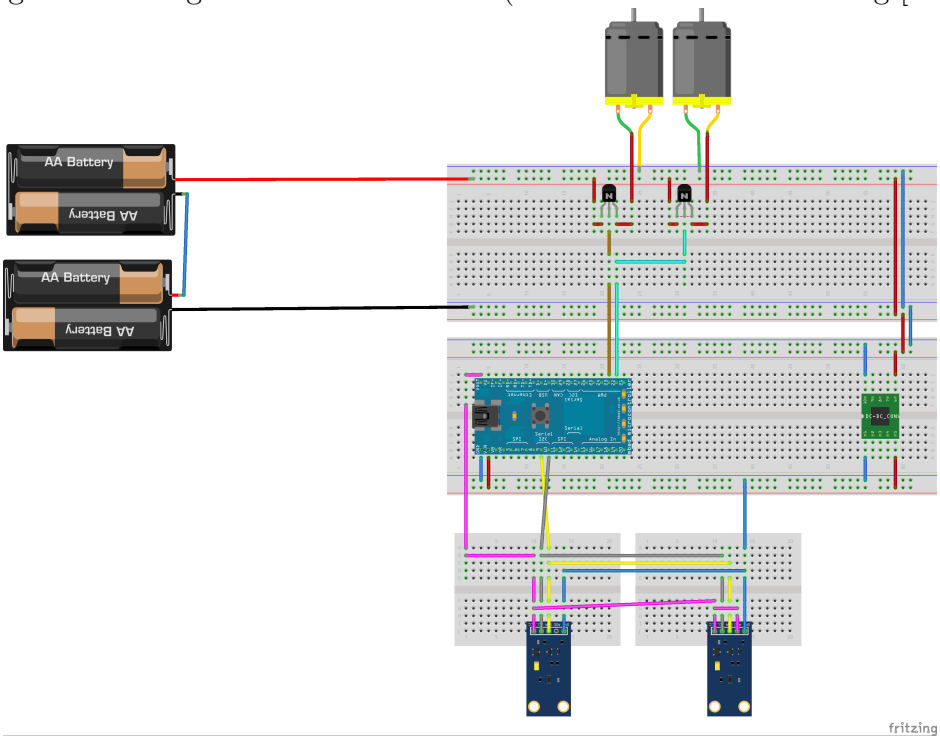
- Sensor de grises izquierdo
- Sensor de grises derecho

- Sensor de distancia apuntando hacia la derecha

Y 2 actuadores:

- Motor izquierdo
- Motor derecho

Figura 5.1: Diagrama del robot móvil (realizado utilizando fritzing [18])



El robot utilizará los sensores de grises para mantenerse sobre la línea y el sensor de distancia para saber cuándo está frente a una casa.

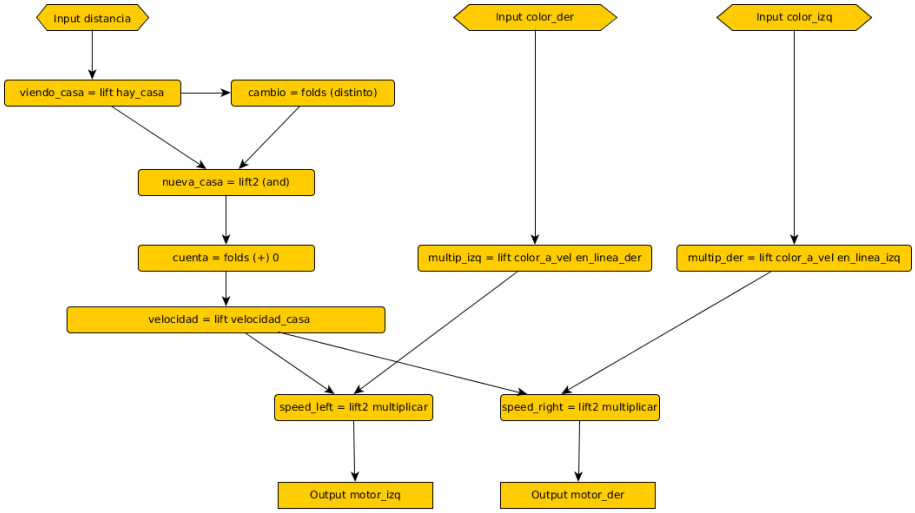
Con los motores se moverá hacia adelante inicialmente, e irá corrigiendo su dirección desacelerando el motor del lado que se salga de la línea.

Durante el trayecto contará las casas, y se detendrá cuando la cuenta llegue al valor 5.

En la figura 5.2 se puede ver gráficamente de que forma se combinan los eventos para lograr el objetivo.

Luego se llega a la implementación en el lenguaje Willie:

Figura 5.2: Diagrama del caso de estudio




```
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

hay_casa d = if (d < MIN_DISTANCE) then 1 else 0
distinto a b = if (a /= b) then 1 else 0
velocidad_casa num = if (num >= 5) then 0 else 100

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

color_a_vel gris = if (gris > MIN_GREY) 1 else 1/2


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
}
```

5.3. Solución sin utilizar 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

hay_casa d = if (d < MIN_DISTANCE) then 1 else 0
distinto a b = if (a /= b) then 1 else 0
velocidad_casa num = if (num >= 5) then 0 else 100
color_a_vel gris = if (gris > MIN_GREY) 1 else 1/2

bool viendo = false
int velocidad = 100
int cuenta = 0

while (cuenta < 5) {
```

```
  //chequear si hay casa
  distance = read(INPUT_DISTANCE)
  if (hay_casa(distance)) {
    if (!viendo) {
      cuenta = cuenta + 1
      viendo = true
    }
  } else {
    viendo = false
  }
  //chequear que no estoy fuera de la linea
  color_izq = read(INPUT_COLOR_LEFT)
  color_der = read(INPUT_COLOR_RIGHT)
  if (color_izq > MIN_GREY) {
    write(MOTOR_DER, velocidad)
  } else {
    write(MOTOR_DER, 1/2 * velocidad)
  }
  if (color_der > MIN_GREY) {
    write(MOTOR_DER, velocidad)
  } else {
    write(MOTOR_DER, 1/2 * velocidad)
  }
}
//Detener robot
write(MOTOR_IZQ, 0)
write(MOTOR_DER, 0)
```

5.4. Conclusiones del caso

Capítulo 6

Conclusiones

Intro .... que se hizo, puntos a favor, etc..

6.1. Trabajo futuro

El principal trabajo futuro sería ...

Sería muy útil contar con una funcionalidad de depuración, la cuál mostrara dependiendo del tiempo los valores de cada fuente de eventos.

Una opción es comunicar mediante el puerto serial el valor de cada señal al cambiar, y mostrarlo en una interfaz web como la que provee RXMarbles (ver [19]). El lenguaje Elm provee de una herramienta que permite viajar en el tiempo, modificar y mostrar la ejecución de un programa, en nuestro caso no sería posible modificar lo que el robot físico realiza, pero si sería útil ver en la línea de tiempo que valores tomaron sus señales. (ver [20])

Bibliografía

[1] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[2] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages*, 1999.

[3] John Peterson, Gregory D. Hager, and Paul Hudak. A language for declarative robotic programming. In *IEEE International Conference on Robotics and Automation*, 1999.

[4] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *International Conference on Functional Programming*, 2001.

[5] Zhangyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. jan 2002.

[6] Sitio web de Microchip. `https://www.microchip.com`. Último acceso: 13/08/2015.

[7] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. oct 2002.

[8] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. aug 2002.

[9] John Hughes. Generalising monads to arrows. In *Science of Computer Programming*, 1998.

[10] Sitio web de Yampa. `https://wiki.haskell.org/Yampa`. Último acceso: 29/04/2015.

[11] Evan Czaplicki. Elm: Concurrent frp for functional guis. Master's thesis, Harvard University, mar 2012.

[12] Sitio web de Alex. http://www.haskell.org/alex. Último acceso: 26/07/2015.

[13] Sitio web de Happy. http://www.haskell.org/happy. Último acceso: 26/07/2015.

[14] Sitio web de MBED-LPC1768. http://developer.mbed.org/platforms/mbed-LPC1768. Último acceso: 30/04/2015.

[15] Sitio web de MBED. https://mbed.org. Último acceso: 30/04/2015.

[16] Sitio web Exporting Mbed to GCC ARM. https://developer.mbed.org/handbook/Exporting-to-GCC-ARM-Embedded. Último acceso: 26/07/2015.

[17] Sitio web de SumoUY. http://sumo.uy/. Último acceso: 03/05/2015.

[18] Sitio web del fritzing. http://fritzing.org/. Último acceso: 13/08/2015.

[19] Sitio web de RXMarbles. http://rxmarbles.com. Último acceso: 18/05/2015.

[20] Sitio web del debugger de Elm. http://debug.elm-lang.org/. Último acceso: 18/05/2015.

Apéndices

Apéndice A

Apéndice

A.1. Gramática del lenguaje de alto nivel

A.1.1. Sintaxis

La gramática del lenguaje es la siguiente:

```
number := [-+]?[0-9]*\.?[0-9]+

name := [a-z_A-Z]+

value := number | name

binop := '+' | '-' | '/' | '*'
       | 'or' | 'and' | '==' | '<='
       | '>' | '<' | '<>' | '>=';

arg_list := name arg_list | name;

expr := value
      | expr binop expr
      | "if" expr "then" expr "else" expr;

assign := name "<- read" name
        | name "<- lift" name name
        | name "<- lift2" name name name
        | name "<- folds" value name
        | "output" name name
```

```
assign_list = assign assign_list | assign;

definitions := name arg_list "=" expr;

main_block := "do {" assign_list "}";

program := definitions main_block;
```

A.1.2. Instrucciones de bajo nivel

A continuación se presenta el resto de las instrucciones de bajo nivel y pseudocódigo indicando su semántica.

- `halt`

  Detiene el hilo de ejecución actual.

  ```
  ip = 0
  ```

- `call function`

  Invoca la funcion *function*. Se asume que los parámetros están en el stack.

- `ret`

  Toma el resultado de una función del tope del stack, limpia el espacio ocupado por la función, y deja el resultado en el nuevo tope del stack.

  ```
  value = stack.pop()
  stack.pop_frame();
  ip = code[stack.pop()]
  fp = stack.pop()
  stack.pop_args()
  stack.push(value)
  ```

- `load_param inm`

  ```
  a = stack.get_arg(inm)
  stack.push(a)
  ```

- `jump`

  ```
  goto position
  ```

- jump_false position

  a = stack.pop()

  if not a: goto position

- cmp_eq

  a = stack.pop()

  b = stack.pop()

  stack.push(a == b)

- cmp_neq

  a = stack.pop()

  b = stack.pop()

  stack.push(a != b)

- cmp_gt

  a = stack.pop()

  b = stack.pop()

  stack.push(a >b)

- cmp_lt

  a = stack.pop()

  b = stack.pop()

  stack.push(a <b)

- add

  a = stack.pop()

  b = stack.pop()

  stack.push(a + b)

- sub

  a = stack.pop()

  b = stack.pop()

  stack.push(a - b)

- `div`

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a / b)
  ```

- `mul`

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a * b)
  ```

- `op_and`

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a and b)
  ```

- `op_or`

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a or b)
  ```

- `op_not`

  Coloca un valor constante en el stack.

  ```
  stack.push(word)
  ```

- `push word`

  Coloca un valor constante en el stack.

  ```
  stack.push(word)
  ```

- `pop`

  Elimina el tope del stack.

  ```
  stack.pop()
  ```

- `dup`

  Duplica el tope del stack.

  ```
  stask.push(stack.tos())
  ```

- `store inm`

  Guarda el tope del stack en la variable *inm*.

  `var[inm] = stack.pop()`

- `load inm`

  Carga la variable $inm \in 0.,255$ en el stack.

  `stack.push(var[inm])`