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

26 de octubre 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 cual 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, no familiarizados con la electrónica ni 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 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 esta forma se implementó 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 *Señales* y relaciones entre ellas, que permiten capturar la naturaleza reactiva del dominio.

Éste lenguaje fue construído para enseñar conceptos de robótica, por lo tanto uno de los objetivos es que sea simple de utilizar e intuitivo.

Otro objetivo planteado es que debe ser posible programar 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 de hardware.

Para cumplir con el objetivo de que la implementación sea portable, se separó la misma en dos fases. Los programas son compilados a código Alf de menor nivel de 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.

Esta forma de programación 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 [2]) 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 o posición. Esta abstracción permite que el desarrollador solo se ocupe de definir cómo 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 [3] [4]) 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 [5], 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* [6], 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* [7].

2.1.3 Arrows

Arrowized FRP (AFRP [8] [9]) 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).*

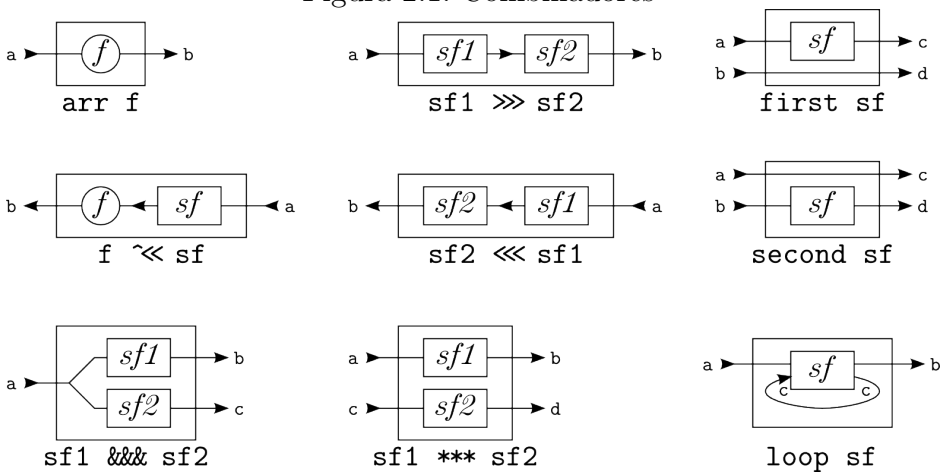
$$\textbf{\textit{Signal}}\ \alpha = \textbf{\textit{Time}} \rightarrow \alpha$$

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* [10] 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 [11] 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 [12], 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.

2.2 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.

2.3 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 este capítulo se describe el diseño del lenguaje Willie junto con su semántica. Luego se explica de qué manera es traducido al lenguaje Alf de bajo nivel, mas simple de interpretar, el cual podrá ser interpretado por implementaciones de una máquina virtual en diferentes plataformas de hardware.

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

El diagrama de la Figura 3.1 resume todas las etapas y componentes necesarios:

El desarrollador escribirá su programa en el lenguaje Willie de alto nivel, ejecutará el compilador y obtendrá un archivo Alf binario.

Por otro lado el robot objetivo, tendrá instalada la máquina virtual (alfvm) correspondiente a la plataforma del mismo.

El desarrollador podrá enviar su código Alf al robot, y la máquina virtual se encargará de interpretarlo.

3.1 Lenguaje Willie

En Willie los programas consisten de un conjunto de funciones, y un conjunto de primitivas del paradigma de programación funcional reactiva.

Es un lenguaje funcional, tipado y con inferencia de tipos. En el todos los valores son inmutables, una vez que son declarados no se pueden modificar. Las funciones declaradas son puras, por lo tanto no pueden tener efectos secundarios.

Para controlar un robot se debe declarar un conjunto de señales utilizando las primitivas de FRP.

Figura 3.1: Etapas y componentes

Para simplificar la implementación, solo se permiten valores naturales ($\mathcal{N}$), y funciones de naturales en naturales. ($\mathcal{N} \to \mathcal{N}$).

El lenguaje evalúa las expresiones, tan pronto como es posible. Las funciones tienen un conjunto de variables libres, cuando un valor les es asignado, el resultado es calculado.

En lenguajes como Haskell las expresiones no son evaluadas hasta que es estrictamente necesario (evaluación a demanda, del inglés: *lazy evaluation*). En la programación funcional reactiva esto puede llevar a aumentar el uso de memoria considerablemente, y al no evaluar incrementalmente, al necesitar un valor puede demorar el cálculo, enlenteciendo la evaluación de todo el programa y reduciendo la reactividad.

3.1.1 Funciones

Las funciones se definen con un nombre, una lista de argumentos y una expresión. Las variables libres de la expresión son sustituídas al evaluar la función.

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

Una expresión puede ser un valor primitivo, una expresión aritmética (por ejemplo una suma o multiplicación), la aplicación de una función, o una expresión condicional. Todas las expresiones retornan un valor al evaluarse.

La sintaxis es muy similar a la del lenguaje `Haskell`, aunque no se permiten funciones anónimas.

Para declarar un valor constante simplemente se escribe una función sin argumentos. Por convención se escriben con mayúsculas, pero no es una restricción.

```
NOMBRE_CONSTANTE = valor
```

Una expresión condicional, debe retornar un valor para cada posible resultado de la condición.

En la figura 3.2 se ve la implementación de la función que retorna un número en la sucesión de Fibonacci, utilizando una expresión condicional.

Se puede ver que una función puede invocarse a si misma, está permitida la recursión.

Figura 3.2: Función de Fibonacci

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

Un comentario es una línea que comienza con el símbolo `#`.

La gramática completa del lenguaje se puede ver en la Figura 3.3.

3.1.2 Combinadores de FRP

Un robot cuenta con un conjunto de sensores y un conjunto de actuadores, cada uno identificados con un número entero.

Para especificar el comportamiento de un robot en un programa, se crean señales a partir de las entradas (sensores), se les aplican funciones y combinan utilizando los combinadores de FRP, y se mapean señales a las salidas (actuadores).

Figura 3.3: Gramatica de Willie

```
program := definitions "do {" frps "}";

definitions := definition | definition definitions;

definition := ident arg_list "=" expr;

frps = frp | frp frps;

frp := ident "<- read" expr
     | ident "<- lift" ident ident
     | ident "<- lift2" ident ident ident
     | ident "<- folds" ident value ident
     | "output" expr ident

expr := name
      | number
      | expr binop expr
      | "if" expr "then" expr "else" expr;

arg_list := ident | ident arg_list;

ident := [a-z_A-Z]+;

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

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

Los combinadores `lift`, `lift2` y `folds`, y las primitivas de entrada/salida `read`, `output` se encuentran dentro del bloque `do` del programa.

Con las primitivas de entrada/salida se define cómo se conectan las señales con sensores y actuadores, y con los combinadores se define un grafo de señales que especifica el comportamiento del robot.

Para que un programa sea válido, el grafo de señales debe ser acíclico.

El bloque `do` permite de manera declarativa expresar las relaciones entre las señales y que funciones se deben aplicar.

La máquina virtual que interpreta el programa, será la encargada de darle valores a las señales y actualizarlas, así como actualizar las salidas de acuerdo a que señal está conectada a ellas.

A continuación se presenta el conjunto de primitivas y combinadores.

Read

Para crear una señal a partir de una entrada, se utiliza la primitiva `read`.

Asumiendo que un robot tiene un sensor de distancia en la entrada `INPUT_DISTANCE` se puede definir una señal `distance`, que contendrá la distancia en centímetros para cada instante de tiempo.

```
distance <- read INPUT_DISTANCE
```

El tipo de la primitiva `read` es:

$$read :: IO\ a \rightarrow Signal\ a$$

Lift

Usando la primitiva `lift` se puede aplicar una función a la señal, y obtener una nueva señal más compleja resultado de la aplicación.

Se puede definir una función `distanceToSpeed` que de acuerdo a la distancia, calcula la velocidad apropiada a la que se debe mover el robot, para detenerse si hay un objeto muy cercano y evitar una colisión.

```
distanceToSpeed n = if (n < MIN_DIST) then STOP else MAX_SPEED
```

Se puede definir la señal `speed`, resultado de aplicar la función `distanceToSpeed` a la señal `distance`.

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

El tipo de la primitiva lift es

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

Output

La primitiva `output` envía a un actuador, el valor de una señal. Asumiendo que el motor del robot está identificado con el valor entero `OUTPUT_ENGINE`:

```
output OUTPUT_ENGINE speed
```

El tipo de la primitiva output es

$$output :: Signal\ a \rightarrow Int \rightarrow IO\ a$$

En la Figura 3.4 se puede ver el ejemplo completo.

Figura 3.4: Ejemplo completo

```
INPUT_DISTANCE = 1
OUTPUT_ENGINE = 1

MIN_DIST = 30
MAX_SPEED = 100
STOP = 0

distanceToSpeed n = if (n < MIN_DIST) then STOP else MAX_SPEED

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

LiftN

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

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

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

$$lift3 :: (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow Signal\ a \rightarrow Signal\ b \rightarrow Signal\ c \rightarrow Signal\ d$$
$$lift3\ f\ sa\ sb\ sc = lift2\ ((lift2\ f\ sa\ sb)\ sc)$$

Folds

En Willie para mantener un valor que dependa de la historia, se utiliza el combinador `folds`.

El mismo es análogo a la operación `fold` sobre listas, viendo una señal como una lista de valores en el tiempo, el combinador aplica una función al valor actual de una señal y a un valor acumulado. De ésta manera se puede crear una nueva señal para representar estado.

Por ejemplo si se asume definida una señal `button` que tiene el valor 1 cuando se apreta un botón y sinó el valor 0, se puede contar cuántas veces se apretó el botón utilizando el combinador `folds` y una función para sumar el valor acumulado y el nuevo.

```
count <- folds suma 0 button
```

El tipo del combinador `folds` es:

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

3.2 Lenguaje de bajo nivel

Al compilar un programa Willie, se obtiene como salida un código intermedio en lenguaje Alf. El mismo es independiente de la plataforma en la que va a ser ejecutado. Para lograr ésto, se define el lenguaje como un conjunto de instrucciones con su semántica y una máquina virtual abstracta que las ejecuta.

Máquina virtual

La máquina que interpreta el lenguaje Alf es una *máquina de stack*.[1].

En una máquina de stack las instrucciones están en notación postfija.[2] Para evaluar expresiones se colocan sus argumentos en una pila, y luego se ejecuta la operación asociada.

Por ejemplo la expresión "$5 + 19 * 8$" en RPN se escribe "5 19 8 * +".

A modo de ejemplo en Alf se representa con las siguientes 5 instrucciones.

```
push 5
push 19
```

[1] Stack machine en inglés

[2] RPN (*Reverse polish notation*) del inglés

```
push 8
mul
add
```

El conjunto *Inputs* representa las entradas de la máquina. Dadas m entradas fijas, cada una se identifica con un entero único entre 1 y $m = |Inputs|$.

Cada $\mathtt{I}_i, i \in (1 \cdots m)$ se corresponderá con un sensor definido en el robot.

Definición 8. *Entradas de la máquina*

$$Inputs \equiv \{\mathtt{I}_1 \cdots \mathtt{I}_m\}.$$

Graficamente las representaré con la notación:

También se cuenta con un conjunto *Outputs* de salidas, identificadas de 1 a $k = |Outputs|$.

Cada $\mathtt{O}_i, i \in (1 \cdots k)$ se corresponderá con un actuador del robot.

Definición 9. *Salidas de la máquina*

$$Outputs \equiv \{\mathtt{O}_1 \cdots \mathtt{I}_k\}.$$

Graficamente las representaré con la notación:

Las señales que se definan se denotarán $\mathtt{S}_i$, siendo i un índice único que las identifica. El conjunto de las señales se llama *Signals*.

Definición 10. *Señales*

$$Signals \equiv \{\mathtt{S}_1 \cdots \mathtt{S}_s\}.$$

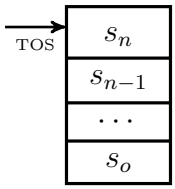
Graficamente las representaré con la notación:

La máquina tendrá una pila global, denotada *Stack*. El mismo se representa con una secuencia de valores.

Definición 11. *Pila global*

$$Stack \equiv s_1, \cdots, s_n.$$

El *Stack* lo representaré graficamente con la notación:



Donde `TOS`[3] indica el índice del tope del mismo. Se cumple que $Stack_{TOS} = s_n$.

Las instrucciones están formadas por un código, un argumento inmediato opcional y una lista de argumentos extra opcionales dependiendo del código.

Utilizaré la siguiente notación para describir las instrucciones:

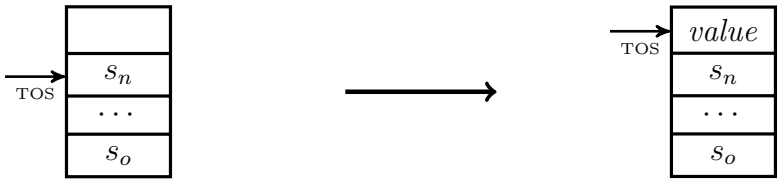
$$\texttt{codigo}[inmediato][, arg_1, \cdots, arg_n]$$

3.2.1 Conjunto de Instrucciones

Instrucciones básicas

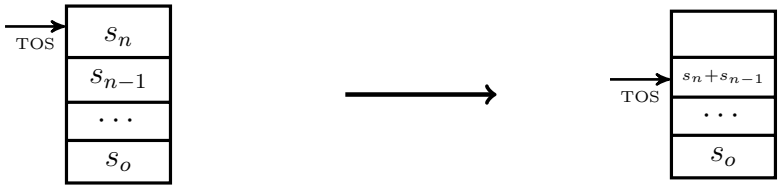
- `push` ,*value*

  La instrucción `push` coloca el valor *value* como tope del stack. En el diagrama a la izquierda se muestra el estado del stack antes de la operación y a la derecha el estado luego de su ejecución.



- `add`

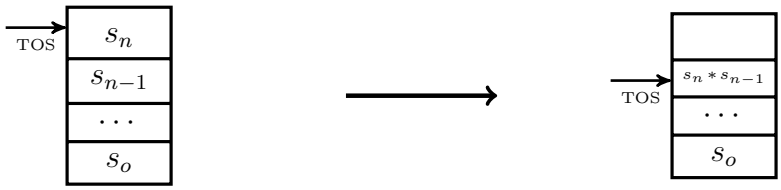
  Remueve dos valores del stack, los suma y coloca el resultado en el tope.



[3]Del inglés: Top of stack

- `mul`

  Remueve dos valores del stack, los multiplica y coloca el resultado en el tope.

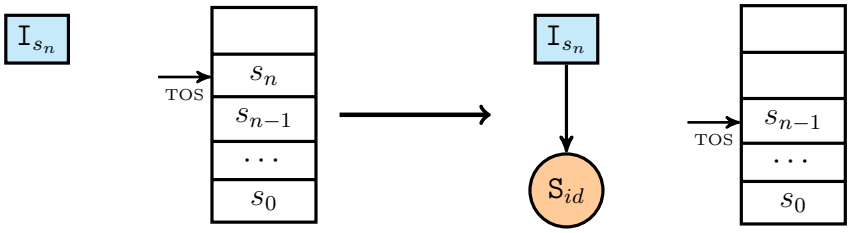


3.2.2 Instrucciones para manipular señales

A continuación se presentan las instrucciones utilizadas para manipular señales.

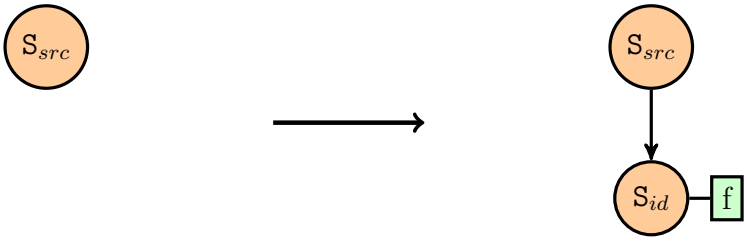
- `read` *id*

  Toma el tope del stack como identificador de una entrada. Crea una señal *id* que contendrá el valor de la entrada en el tiempo. Como precondición, la señal *id* no debe existir.



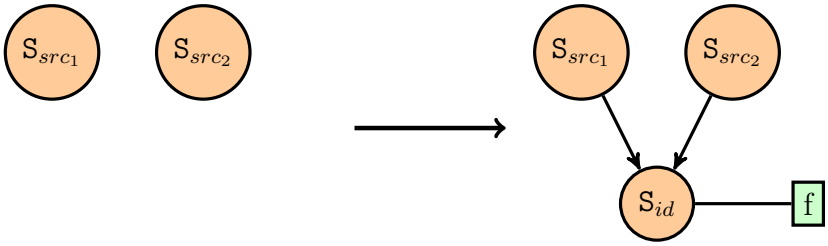
- `lift` *id*, *src f*

  Crea una señal *id* aplicando la función *f* a la señal *src*. Cada vez que la señal *src* cambie de valor, se le aplica la función *f* y la señal *id* cambia de valor.



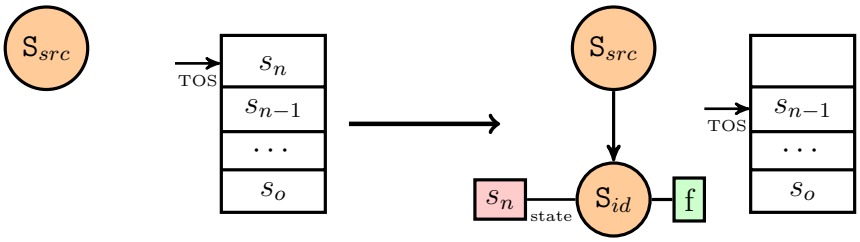
- `lift2` *id*, src_1 src_2 *f*

Crea una señal *id* aplicando el combinador `lift2` usando la función *f*, y las señales src_1 y src_2. Cuando ambas señales cambien de valor, se aplica la función y la señal *id* cambia de valor.



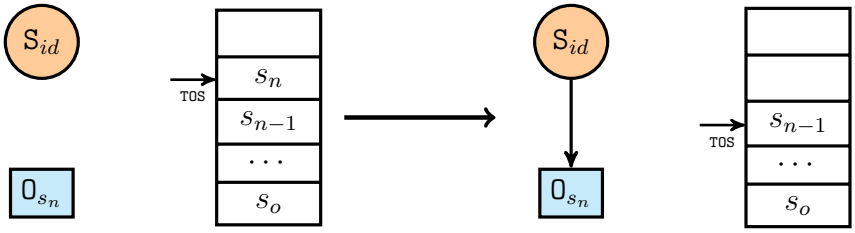
- `folds` *id*, *src f*

  Crea una señal *id* aplicando el combinador `folds`. El valor inicial de la señal está dado por el tope del stack, luego el mismo se actualiza aplicando la función *f* al valor actual y a los valores recibidos de la señal *src*.



- `write` *id*

  Envía los valores de la señal *id* a la salida identificada con el valor s_n (O$_{s_n}$) que se encuentra en el tope del stack (`TOS`).



En el apéndice A.2 se encuentra el listado completo de operaciones y su descripción.

3.2.3 Ejemplo de traducción

Dado un robot con un sensor de distancia y un led, el siguiente programa Willie enciende el led cuando el robot detecta una casa.

```
#Inputs
INPUT_DISTANCE = 1
#Outputs
OUTPUT_LED = 1

isHouse distance = if (distance < 100) then 1 else 0

do {
  signal_distance <- read INPUT_DISTANCE
  signal_house <- lift isHouse signal_distance
  output OUTPUT_LED signal_house
}
```

El mismo se traduce a Alf de la siguiente forma:

```
0: t_call
1: 10
2: t_read 1
3: t_lift 0
4: 1
5: 16
6: t_call
7: 13
8: t_write 0
9: t_halt
10: t_push
11: 1
12: t_ret
13: t_push
14: 1
15: t_ret
16: t_load_param 0
17: t_push
18: 100
19: t_cmp_lt
20: t_jump_false
21: 26
22: t_push
23: 1
24: t_jump
25: 28
```

```
26: t_push
27: 0
28: t_ret
```

Para entender el programa Alf, primero se divide en dos secciones. Entre la línea 0 y la línea 9 está el código correspondiente a la sección `do`.

A partir de la línea 10 están las declaraciones de funciones.

La declaración:

```
10: t_push
11: 1
12: t_ret
```

se corresponde con la definición de la constante `INPUT_DISTANCE`, y la declaración

```
13: t_push
14: 1
15: t_ret
```

es la definición de la constance `OUTPUT_LED`.

La función `isHouse` se traduce a:

```
16: t_load_param 0
17: t_push
18: 100
19: t_cmp_lt
20: t_jump_false
21: 26
22: t_push
23: 1
24: t_jump
25: 28
26: t_push
27: 0
28: t_ret
```

En el bloque `do` la señal `signal_distance` se crea cargando el valor de `INPUT_DISTANCE` en la pila, y luego usando la instrucción `read`. El argumento 1 de la instrucción `read` será el identificador de la señal.

```
0: t_call
1: 10
2: t_read 1
```

Capítulo 4

Implementación

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

4.1 Compilador

El compilador será el encargado de leer el programa Willie y traducirlo a Alf.

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

Es usual realizar tareas de compilación en *Haskell* por lo que existen herramientas estándar para cada etapa.

El compilador constará de una secuencia de etapas: Análisis Léxico, Análisis Sintáctico, Análisis Semántico y Generación de Código.

En la Figura 4.1 se puede ver la estructura más detallada del compilador y a continuación se describe cada etapa representada en la Figura.

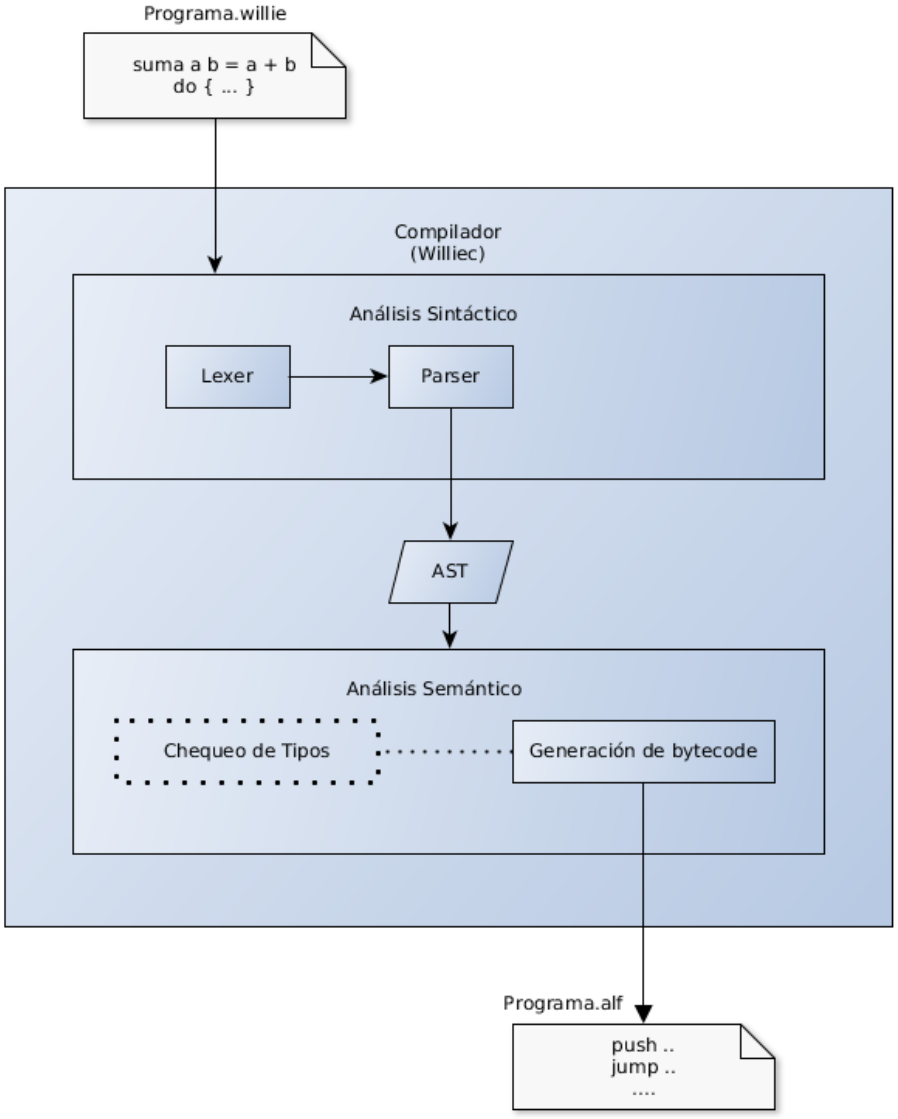
4.1.1 Análisis Léxico

La primera etapa se llama análisis léxico, en esta se lee el código fuente en lenguaje Willie (.willie) y lo transforma en una lista de lexemas.

Un lexema puede ser una palabra reservada (ej: `do`), un valor (ej: 19), un identificador (eg: `distance`) o un símbolo reservado (eg: `+`).

Para representar los lexemas, se utiliza la herramienta *UU.Scanner* [13] que estandariza los mismos en el tipo de datos `Token`.

Figura 4.1: Diagrama del compilador



Usando *Alex*[14] se procesa el código fuente, se reconocen los lexemas y se retorna una lista de tipo `[Token]`.

La etapa se puede resumir en la implementación de la función `tokenize`.

```
tokenize :: String -> String -> [Token]
```

4.1.2 Análisis Sintáctico

La segunda fase del compilador, recibe la lista de lexemas (`[Token]`) y reconoce el lenguaje, generando un árbol de sintaxis abstracta (AST[1]).

Para reconocer la gramática se implementó un parser recursivo descendente. Utilizando la herramienta *UU.Parser* [13], se definió un tipo de datos `TokenParser a` que representa un parser que recibe una secuencia de lexemas de tipo `Token` y retorna un AST de tipo `a`.

```
type TokenParser a = Parser Token a
```

UU.Parser define un conjunto de combinadores de parsers y utilizándolos se construyen parsers complejos a partir de parsers simples.

Para representar el AST se utiliza una gramática de atributos. Una gramática de atributos es como una gramática libre de contexto, pero agrega semántica a la misma. Para el análisis sintáctico, la semántica no es utilizada, pero será usada en la próxima etapa.

El sistema de gramáticas de atributos $UUAG$[15] fue usado para la implementación.

Se define un tipo de datos `Root` que representa la raíz del árbol. El mismo tiene un único constructor `Root_Root` que recibe un árbol de tipo `Decls` que representa las declaraciones, y un árbol de tipo `Dodecls` que representa el bloque `do`.

Para crear el AST usando *UU.Parser* se define el parser `pRoot`:

```
pRoot :: TokenParser Root
pRoot
  = (\x y -> Root_Root x y) <$> pDecls <*> pDodecls
```

El cuál asume definido un parser de declaraciones `pDecls` y un parser del bloque `do` (`pDodecls`).

```
pDecls :: TokenParser Decls

pDodecls :: TokenParser Dodecls
```

Se va refinando sucesivamente en parsers mas específicos, hasta construir completamente el AST.

[1] Del inglés Abstract Syntax Tree

4.1.3 Análisis Semántico

Para la última etapa se utiliza la gramática de atributos para definir semántica sobre el *AST*.

Las gramáticas de atributos [**?**](*Attribute Grammars*) simplifican la tarea de escribir catamorfismos. Un catamorfismo es una función análoga a la función de alto orden `foldr` pero aplicada sobre cualquier tipo de datos recursivo.

De ésta manera se pueden definir atributos sintéticos, heredados o mixtos en el *AST*.

Uno de dichos atributos será el código en bajo nivel, que será la salida de esta etapa.

Se implementó una gramática de atributos usando *UUAG*, y con el compilador de gramáticas *UUAGC*[16] se compiló a haskell.

El compilador *UUAGC* toma la gramática y construye los catamorfismos necesarios para procesar todos los atributos.

Construir el compilador, se reduce a obtener una secuencia de atributos sobre el *AST* que sirven para generar el código Alf.

Por ejemplo para construir el código de un programa, la raiz del *AST* está dada por el tipo de datos `Root`.

```
data Root
  | Root
      decls :: Decls
      dodecls :: Dodecls
```

Se puede definir el código como la concatenación del código de las declaraciones del bloque `do`, una instrucción `halt` y el código de las declaraciones de funciones (`Decls`).

Para ésto se define un atributo sintetizado (`syn`) llamado `code`.

```
set All = Root Decls Decl Dodecls Dodecl Expr
attr All syn code use {++} {[]} :: BC

sem Root
  | Root
    lhs.code = @dodecls.code ++ [Thalt] ++ @decls.code
```

En la definición del atributo, se especifica que en caso de no haber una regla específica, se calcula usando la concatenación `++` y como atributo por defecto toma `[]`.

```
attr All syn code use {++} {[]} :: BC
```

Por ejemplo para la definición de la lista de declaraciones, no es necesario especificar que el código se obtiene concatenando sus partes, se infiere automáticamente usando la regla anterior.

```
type Decls = [Decl]
```

Para poder generar el código de todo el programa, es necesario calcular otros atributos previos. Se necesita saber la posición en la que quedarán las funciones para poder tener una referencia a ellas. Para saber la posición, es necesario calcular el largo del código antes de tener el código.

Para ésto se definió un atributo sintetizado `len` que contiene el largo que tendrá cada bloque luego de traducido a código, pero sin llegar a traducirlo.

También se definió un atributo `pos` que indica en que posición estará ubicado el código que se genere para cada producción de la gramática. El atributo `pos` es un atributo heredado (`inh`) en el *AST*.

Por ejemplo en `Root`, se utiliza el atributo `len` de las declaraciones del bloque `do` para saber a partir de que posición `pos` estarán ubicadas las declaraciones de funciones.

```
attr All syn code use {++} {[]} :: BC
         syn len use {+} {0} :: Int
         inh pos :: Int

sem Root
  | Root
      lhs.code = @dodecls.code ++ [Thalt] ++ @decls.code
      dodecls.pos = 0
      decls.pos = @dodecls.len + 1
```

Utilizando el atributo `pos`, se puede saber en que posición estará cada función en el código generado. Para tener la posición de todas las funciones se utiliza un atributo encadenado (`chn`) llamado `labels`, es heredado pero también es sintetizado. Por ejemplo al declarar una función, se agrega la posición `pos` asociada al nombre de la misma.

```
sem Decl
  | Function
      lhs.code = @body.code ++ [Tret]
      lhs.len = @body.len + 1
      lhs.labels = addLabel @name @lhs.pos @lhs.labels
```

El atributo contiene un mapa que dado un nombre de una función retorna la posición de la misma, `labels` recolecta la posición de todas las funciones. Luego otro atributo `labelMap` se declara como heredado `inh` y se le asigna en `Root` el valor de `labels`, `labelMap` se usa para distribuir el mapa completo en todo el *AST*.

```
sem Root
  | Root
      decls.labels = emptyLabelMap
      decls.labelMap = @decls.labels
      dodecls.labelMap = @decls.labels
```

Por último un atributo `env` encadenado recolecta las declaraciones de identificadores, a cada identificador de señal le asigna un número entero único y mantiene la lista de las variables en el alcance (scope) dentro de una función. Luego que el atributo `env` recolecta todas las declaraciones, el resultado es distribuido con el atributo heredado `envInh`.

```
sem Root
  | Root
      decls.env = emptyEnv
      dodecls.env = emptyEnv
      dodecls.envInh = @dodecls.env
```

Usando todos éstos atributos se genera el código para cada producción de la gramática, y el atributo `code` se puede calcular.

Se definió un módulo `Bytecode` que abstrae el código de máquina en un tipo `OpCode` y define funciones para exportarlo como texto o en formato binario.

Al compilar la gramática usando *UUAGC* se obtiene un módulo en lenguaje `Haskell` que expone la función `code_Syn_Root` y deja accesible el código resultado como una lista de tipo `[OpCode]`.

Utilizando el módulo `Bytecode`, el código se obtiene y escribe en un archivo (.alf) terminando el proceso de compilación.

4.2 Máquina virtual

La máquina, deberá ejecutar el código de bajo nivel en una plataforma objetivo.

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 [17] y en [18].

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.

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 [19], las herramientas de desarrollo para compilar offline.

La máquina tendrá dos partes principales, una que interpreta el código e implementa el despachador que actualiza las señales. Ésta parte es común y puede ser portada a diferentes plataformas sin necesidad de modificarla.

Las instrucciones en memoria tendrán un ancho de palabra de 16 bit. La primer palabra contiene en los 8 bits más representativos, el código de la operación (`opcode`). Los 8 bits menos representativos, contienen un argumento inmediato opcional. Luego según el `opcode`, algunas instrucciones pueden tener argumentos adicionales, en las siguientes palabras de 16 bit.

codigo de 8 bit	inmediato de 8 bit
argumento 1 opcional de 16 bit	
...	
argumento n opcional de 16 bit	

La máquina mantiene un puntero a la siguiente instrucción a ejecutar llamado `ip` [2]. Cuando `ip` no es nulo, la máquina ejecuta todas las instrucciones hasta que el mismo se haga nulo. El pseudocódigo de la máquina es:

```
1 - Crear grafo de señales vacío, inicializar pila.
2 - Apuntar ip al inicio del código.
3 - Ejecutar código hasta que ip se haga nulo.
4 - Para siempre:
  4.1 - Leer entradas.
  4.2 - Propagar valor de las señales.
  4.3 - Escribir salidas.
```

[2] IP: del inglés, Instruction Pointer significa puntero a instrucción

Al principio se crea un grafo de señales vacío, y se reserva espacio para la pila, todo es memoria estática, tanto para los nodos del grafo como para la pila. En el punto 2, se interpretan las instrucciones del programa, hasta llegar a la instrucción `halt`. Las instrucciones al inicio del código se corresponden con el bloque `do` del programa, por lo tanto al ejecutarlo se obtiene el grafo de las señales completo.

Con el grafo armado, luego se obtienen los valores de las entradas necesarias, y para cada señal se calcula su valor. Las señales que estén conectadas a una salida, se usan para actualizar el valor de las mismas.

Para cada instrucción hay una función definida que interpreta, el despachador tomará una a una las instrucciones e invocará la función que la maneja de acuerdo a que operación es. Las funciones son de tipo `void` y realizan cambios sobre el estado de la máquina. La referencia a las mismas es guardada en un vector `functions`. La posición de cada instrucción en el vector, coincide con el código de operación.

```
void (*functions[])() = {
    f_halt,
    f_call, f_ret, f_load_param,
    f_lift, f_lift2, f_folds,
    f_read, f_write,
    f_jump, f_jump_false,
    f_cmp_eq, f_cmp_neq, f_cmp_gt, f_cmp_lt,
    f_add, f_sub, f_div,
    f_mul, f_op_and, f_op_or,
    f_op_not,
    f_push, f_pop, f_dup,
    f_store, f_load
};
```

Por ejemplo, al despachar el operador `push`, la siguiente palabra contiene el valor a colocar en el stack. El código en lenguaje `C` que maneja la instrucción es:

```
void f_push() {
    *++sp = *ip++;
}
```

Se creó un archivo `Makefile` para construir una imagen binaria de la máquina virtual. 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 [20], 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 (Figura 5.1) que cuenta con 3 sensores:

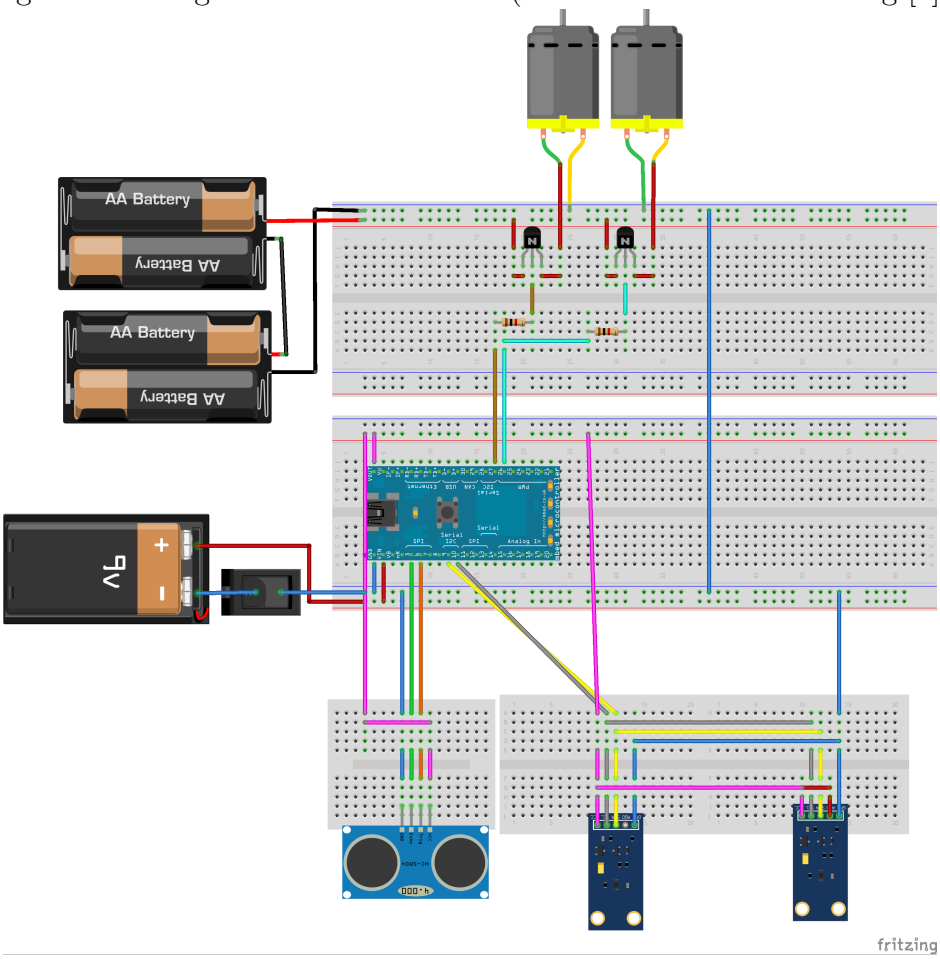
- Sensor de grises izquierdo
- Sensor de grises derecho

- Sensor de distancia apuntando hacia la derecha

Y 2 actuadores:

- Motor izquierdo
- Motor derecho

Figura 5.1: Diagrama del robot móvil (realizado utilizando fritzing [1])



TODO: Explicar diagrama y corregirlo.

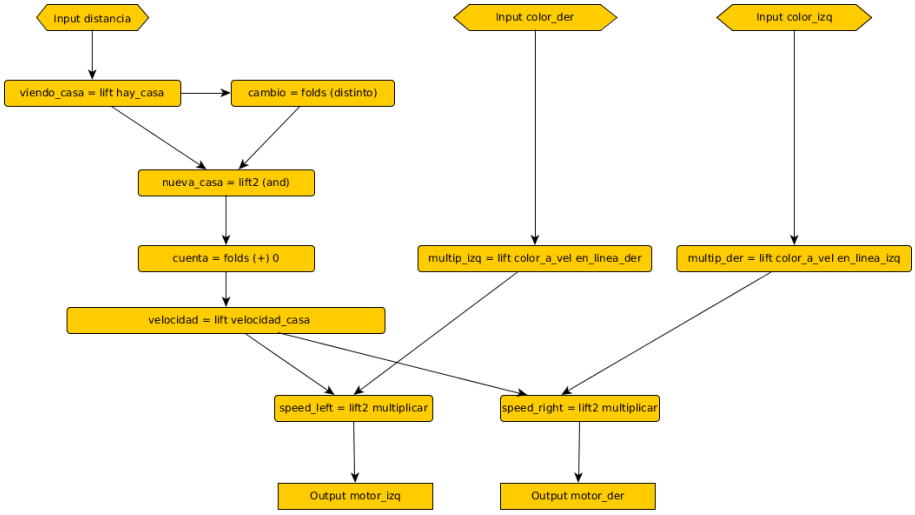
El robot utilizará los sensores de grises para mantenerse sobre la línea y el sensor de distancia para saber cuándo está frente a una casa.

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 qué forma se combinan los eventos para lograr el objetivo.

Figura 5.2: Diagrama del caso de estudio



TODO: Explicar el algoritmo en lenguaje natural.

Luego se llega a la implementación en el lenguaje 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

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

TODO: Corregirlo, elegir un lenguaje, decir en que está implementado, etc.

```
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 [21]). 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 [22])

Bibliografía

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

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

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

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

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

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

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

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

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

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

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

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

[13] Sitio web de Ulrecht University Parser Combinators (uu-parsinglib). `http://foswiki.cs.uu.nl/foswiki/HUT/ParserCombinators`. Último acceso: 25/10/2015.

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

[15] Sitio web de Ulrecht University Attribute Grammar System (UUAG). `http://foswiki.cs.uu.nl/foswiki/HUT/AttributeGrammarSystem`. Último acceso: 25/10/2015.

[16] Sitio web de Ulrecht University Attribute Grammar System Compiler (UUAGC). `http://foswiki.cs.uu.nl/foswiki/HUT/AttributeGrammarSystem`. Último acceso: 25/10/2015.

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

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

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

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

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

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

Apéndices

Apéndice A

Apéndice

A.1 Manual de usuario

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

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

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

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

A.2 Manual de Referencia

A.2.1 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])`