



**UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO**

---

**FACULTAD DE INGENIERÍA**

**Recetas de diseño una forma  
de aprender y enseñar a  
programar**

**TESINA**

Que para obtener el título de  
**Ingeniería en Computación**

**P R E S E N T A**

Omar Sibaja Bautista

**DIRECTOR DE TESINA**

Mtro. Juan José Carreón Granados



Ciudad Universitaria, Cd. Mx., 2016

**Recetas de diseño una forma de  
aprender y enseñar a programar.**

## Índice

Resumen.....	3
1. Introducción.....	4
2. Relevancia del servicio social.....	7
2.1 Beneficios a la comunidad.....	9
2.2 Beneficios a los estudiantes de ingeniería.....	9
2.3 Beneficios en el aprendizaje de las matemáticas.....	10
3. Conceptos Básicos.....	12
3.1. Importancia de las expresiones en la programación.....	23
3.2. Herramientas auxiliares para codificar.....	23
3.3. Evaluación de expresiones.....	25
4. WeScheme.....	27
4.1 WeScheme para el aprendizaje.....	28
4.2 Ventajas de usar WeScheme.....	29
4.3 Manejo de WeScheme.....	29
4.4 Funcionalidad de WeScheme.....	32
5. Recetas de diseño.....	41
5.1 Cómo diseñar funciones.....	42
5.2 Expresiones dentro del diseño de funciones.....	45
5.3 Condicionales en el diseño de funciones.....	49
5.4 Booleanos.....	49
5.5 Expresiones condicionales de tipo if.....	51
5.6 Expresiones condicionales de tipo <i>cond</i> .....	58
6. Cómo diseñar mundos.....	63
6.1. Función big-bang.....	64
6.2. Receta de diseño de mundos.....	65
7. Conclusiones.....	78
7.1 Conclusiones personales.....	79
8. Bibliografía.....	80

## **RESUMEN**

La metodología de diseño de programas con base en recetas de diseño que muestran paso a paso cómo programar y cómo construir programas de calidad que puedan ser comunicables; que además utilizan imágenes, problemas aritméticos y expresiones matemáticas, al mismo tiempo que facilitan el aprendizaje de esta metodología, desarrollan y refuerzan el conocimiento matemático de los estudiantes, innovan su aprendizaje al generar en el estudiante motivación y pasión por la programación y el álgebra.

## 1. INTRODUCCIÓN

Cuando se menciona la palabra receta se piensa en una lista con una serie de pasos o instrucciones para cumplir con un objetivo determinado, en esta metodología de desarrollo mediante **Recetas de Diseño**, no se refiere solamente a una serie de pasos, sino a construir las bases de un programa el cual pueda crecer hasta donde se desee, haciéndolo comunicable; en muchas ocasiones se crean programas que solo su desarrollador entiende al construirlos, y que con el paso del tiempo ni el que los construyó los entiende.

La metodología con base en **Recetas de diseño** permite crear programas que sean entendibles por individuos diferentes a su desarrollador inicial, lo que permite corregirlos y extenderlos en el tiempo.

Este reporte de servicio social se basa en conocer, experimentar y mostrar la metodología acerca del diseño de programas de una forma didáctica, permitiendo a los estudiantes entender la programación y profundizar o aprender en áreas de las matemáticas como son el Álgebra y la Geometría.

Para ello se utilizó principalmente un Ambiente de Desarrollo Integrado, ***Integrated Development Environment (IDE)*** fácil de usar denominado ***WeScheme***, el cual solo requiere disponer de una cuenta de Gmail para poderse emplear. Ese tipo de IDE permite que el primer contacto con la programación sea dinámico, ya que cuenta con herramientas que permiten codificar y ejecutar desde un navegador, pudiéndose acceder desde cualquier dispositivo que lo contenga, y compartir y desarrollar los programas, ya sea mediante correo electrónico o redes sociales.

Este ***IDE*** utiliza el lenguaje ***Racket***, el cual es muy práctico para enseñar esta metodología, a la vez que puede almacenar los programas que el estudiante ha realizado, además de documentación de ayuda con ejemplos prácticos que permiten el aprendizaje, mejorarlo y ampliarlo.

Esta metodología que se presenta para el diseño de programas está basada en décadas de investigación por parte de un consorcio educativo y de investigación en el que participan universidades, grupos y empresas de diferentes lugares del mundo.

Esa metodología se enfoca en crear ***recetas de diseño*** las cuales permiten desde un inicio pensar en el proceso de solución de problemas mediante programación.

Partiendo del análisis de problemas que inicialmente son vagos, difusos e incompletos y pueden describirse mediante información que puede incluir imágenes, expresiones aritméticas y expresiones algebraicas, traduciéndose en datos procesables mediante ***Racket***, generando nuevos datos que pueden traducirse en nueva información que solucione de una forma comunicable el problema inicial.

Aprendiendo la metodología mediante el diseño de animaciones, pequeños videojuegos, que revolucionan la forma tradicional de aprender a programar, lo que lo convierte en una metodología motivante y dinámica.

Este aprendizaje se ha puesto en práctica realizando ponencias, cursos, talleres en universidades de México, y en proyectos como el ***PAPIIT IN102210: Diseño de aplicaciones distribuidas, interactivas y gráficas, en Android.***

*“No podemos resolver problemas pensando de la misma manera que cuando los creamos.”*

**--Albert Einstein--**

## **2. RELEVANCIA DEL SERVICIO SOCIAL**

El objetivo de esta investigación que se ha llevado a cabo por varios años en la Facultad de Ingeniería de la UNAM a cargo del J. J. Carreón Granados es ayudar a los estudiantes de cualquier semestre a manejar una metodología, la cual los enseñe a diseñar programas y simultáneamente fomentar su aprendizaje de Álgebra así como el buen manejo de la Aritmética.

Con base en estos resultados se podrán generar workshops que maneja Bootstrap<sup>1</sup> diseñados en habla hispana, de esta manera, se podrá llevar la programación y el aprendizaje de las matemáticas a comunidades marginadas.

El principal problema que se presenta cuando una persona ya sea estudiante de algún área de computación o cualquiera que desee aprender a programar es no saber por dónde o cómo comenzar.

---

<sup>1</sup> <http://www.bootstrapworld.org/>



¿No saber por dónde o cómo comenzar? Esta pregunta puede sonar absurda para muchas personas, pero es la realidad que se vive en la actualidad. Alguien que quiera iniciar en el área del software por lo regular inicia con lenguajes que necesitan descargar **IDE'S** y compiladores complejos.

Lo anterior provoca que el primer contacto con la programación sea difícil o aburrido por lo cual se abandona, pues no se define una metodología clara de desarrollo que permita no solo enfocarse en un lenguaje en específico sino una metodología que sirva para resolver problemas de programación desde cualquier perspectiva y con cualquier lenguaje.

En este reporte de servicio social se detalla una metodología, la cual fue trabajada teórica y prácticamente, atacando el problema desde una perspectiva fácil, didáctica y dinámica enfocada 100% al aprendizaje.

J. J. Carreón Granados<sup>2</sup> (2016, p.2) menciona que una de las herramientas para este propósito son **IDEs** en la nube tal como **WeScheme**, desarrollado por Bootstrap, (Bootstrap, 2016) que no requieren bajarse ni instalarse, sino solo una cuenta de correo en Gmail, permitiendo recuperar y almacenar archivos en la nube y editarlos desde un navegador, además de poder compartirlos mediante enviar una liga, o postearlos en sitios como Facebook o Twitter.

Si se requieren efectuar todo localmente, Bootstrap se apoya en **DrRacket**, un ambiente gráfico multiplataforma (que corre en Windows, OS X, Unix/Linux), de modo que los programas escritos en una plataforma corren en otras, apoyando una variedad de escenarios de computación tanto escolares como en el hogar. También, enfatizando características amigables a novatos, como la de apoyarse en imágenes y videos. (Racket, 2016)

---

<sup>2</sup> Profesor TC, Sistemas Inteligentes, Dpto Ing. en Comp, DIE, FI, UNAM

Si bien las herramientas mencionadas anteriormente están inicialmente orientadas a estudiantes de secundaria con limitaciones, empleando estrategias didácticas apoyadas fuertemente en imágenes y animaciones, pueden escalarse a temas más avanzados de matemáticas y programación. No sólo educativos de nivel medio superior y superior, también de posgrado.

## **2.1 Beneficios a la comunidad**

Trabajar con el material del PLT<sup>3</sup>, WeScheme y DrRacket, permite generar material (workshops) en habla hispana para que estudiantes o personas con interés tengan acceso a esta metodología, facilitando el entendimiento de conceptos básicos y avanzados de la programación.

Con esta metodología se pretende motivar a los estudiantes para que aprendan a programar y así evitar que abandonen el estudio de la programación, al mismo tiempo que aprenden y repasan conceptos fundamentales de la aritmética y el álgebra, como el concepto de función; con un material pensado para estudiantes en desventaja.

## **2.2 Beneficios a los estudiantes de ingeniería**

En la comunidad ingenieril existen casos de estudiantes que no saben programar pero están interesados en aprenderla, con esta metodología se les puede brindar una inducción motivante a lo que es la programación y si lo desean profundizar en este tema.

Otro caso es de los estudiantes que saben algo de programación, esta metodología les sirve como una herramienta más a su formación, la cual les

---

<sup>3</sup> <http://plt-scheme.org/>

brinda un concepto de diseño de programas, de esta forma mejoran su nivel de abstracción y programación.

### 2.3 Beneficios en el aprendizaje de las matemáticas

Dentro de este modelo se vincula el razonamiento algebraico en la resolución de problemas de programación por ejemplo:

Se tiene la función:

$$1 + 1 = 2$$

$$1 + 2 = 3$$

...

$$1 + x = f(x)$$

Siendo  $f$  la relación, la función que suma 1 a cualquier número

$$f(x) = x + 1$$

lo que en *WeScheme* o *Racket* se representa como

```
(define (f x)
```

```
(+ x 1))
```

J. J. Carreón Granados<sup>4</sup> (2016, p.5,p.6) menciona que al abstraer operaciones aritméticas en operaciones algebraicas parece trivial, pero no lo es. Sin embargo, se puede aprender, sin demasiada dificultad, a manejar definiciones de funciones o sea programas (en los lenguajes funcionales empleados en Bootstrap y DrRacket).

Las mismas reglas de abstracción permiten generalizar la aritmética y manejar aritméticas de cadenas, booleanos, e imágenes, entre otras aritméticas; las que a su vez se pueden abstraer en álgebras de cadenas, booleanos, imágenes ... e

---

<sup>4</sup> Profesor TC, Sistemas Inteligentes, Dpto Ing. en Comp, DIE, FI, UNAM

incluso de funciones. (Racket, 2016) Lo que permite manejar teóricamente de forma eventual cualquier programa de computadora exclusivamente con base en funciones matemáticas, (edX, 2015) entre otros aspectos.

## **Resultados**

Con profesores y estudiantes del nivel de licenciatura y de bachillerato de la UNAM, además de profesores y estudiantes de otras instituciones de educación superior y media nacionales, se han organizado cursos a profesores, impartido conferencias, y realizado seminarios de investigación con profesores, estudiantes de servicio social y tesistas.

En general, se ha obtenido más éxito con estudiantes y profesores fuera de la UNAM, que en la propia UNAM. Más con estudiantes que con profesores. Más con profesores que no tienen que ver nada con carreras como Ingeniería en Computación que con áreas afines a ésta.

En el caso de la Facultad de Ingeniería de la UNAM un obstáculo ha sido la idea en profesores de matemáticas que la programación, más allá de ser una herramienta auxiliar útil, poco tiene que ver con las matemáticas, en particular con la aritmética y el álgebra. Algo parecido sucede con los profesores afines a la computación que consideran que las matemáticas son útiles para la computación, pero que no existe una relación íntima entre ellas.

Sin embargo, a lo largo de los años los profesores de matemáticas han comenzado a intuir que de alguna manera u otra requieren avanzar en su manejo de conocimientos y herramientas de cómputo para ser más eficaces y eficientes.

*“Cualquier tonto puede escribir código que un ordenador entiende. Los buenos programadores escriben código que los humanos pueden entender”  
-Martin Fowler-*

### 3. CONCEPTOS BÁSICOS

Un programa es motivante y emocionante cuando es interactivo entre el usuario y la computadora. Por ejemplo, un programa es interactivo cuando existe interacción con el teclado, el mouse, la pantalla y/o dispositivos externos de la computadora, todo integrado a través de un videojuego.

Para que el aprendizaje de la programación genere motivación en el estudiante, las **recetas de diseño** muestran cómo programar con imágenes, combinando a la vez ejercicios de matemáticas y creando videojuegos sencillos.

El siguiente ejemplo muestra una receta de diseño sencilla, más adelante se explica una **receta de diseño de mundos** más elaborada la cual se enfoca a

crear **programas mundo** que interactúan con el estudiante en forma de videojuegos.

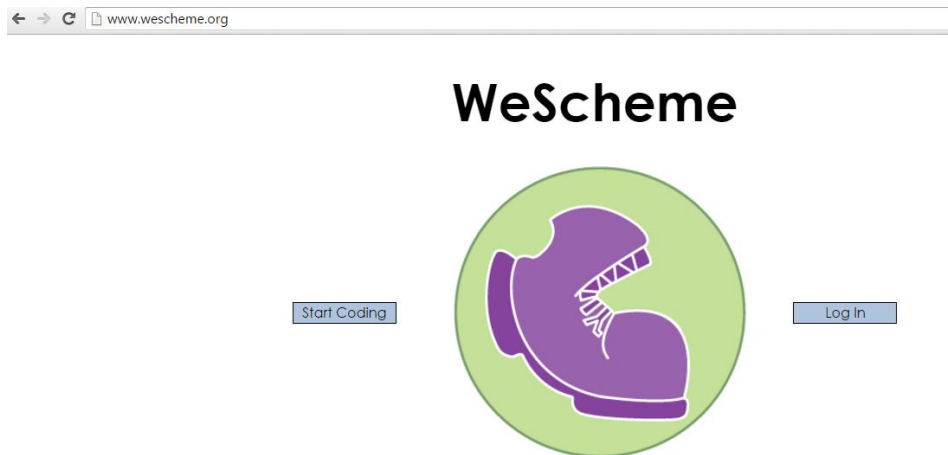
**Ejemplo:**

Crear una animación a partir de un círculo, el cual va incrementando su tamaño con base en el paso del tiempo medido en segundos.

**Solución:**

Para la solución de este problema se utilizó el ambiente de desarrollo de nombre **WeScheme**. Para acceder a este se ingresa la siguiente dirección web:

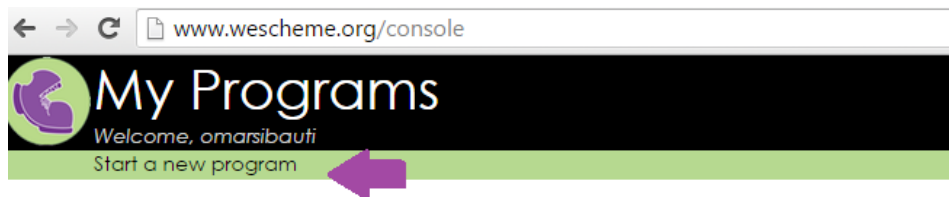
***<http://www.wescheme.org/>***



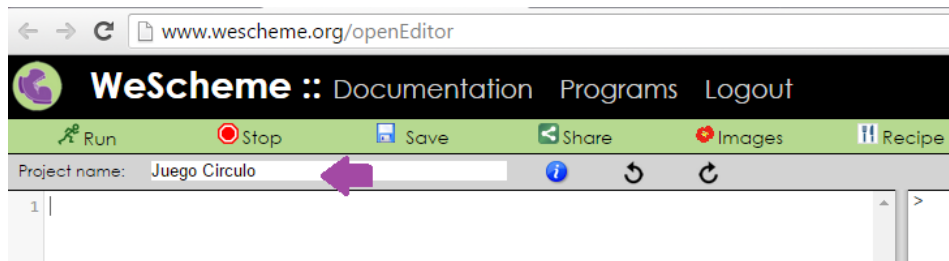
En esta página web se encuentran dos botones: uno **Start Coding** y otro **Log in**. Ambos son la entrada para el ambiente de desarrollo, la única diferencia es que en uno se registra una cuenta de Gmail, para este caso se utilizara el botón **Log in**.

Al ingresar con el botón **Log in** pide una cuenta Gmail, si no se cuenta con una, se puede crear se muestra más detalle en el capítulo 4 de este texto.

En **WeScheme** para crear un programa se presiona el botón **Start a new program**.

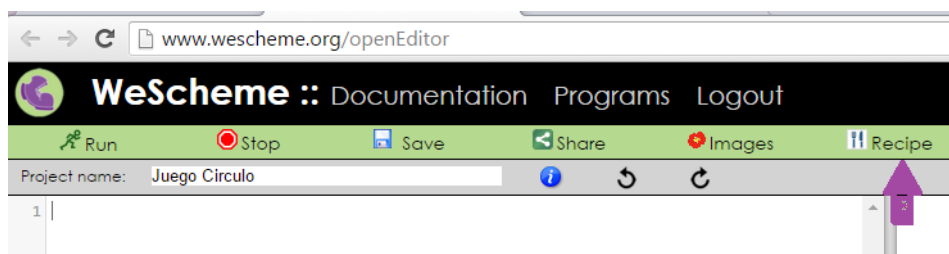


En la etiqueta de nombre **Project name** se ingresa el nombre del programa. Para este caso el nombre es Juego Círculo.



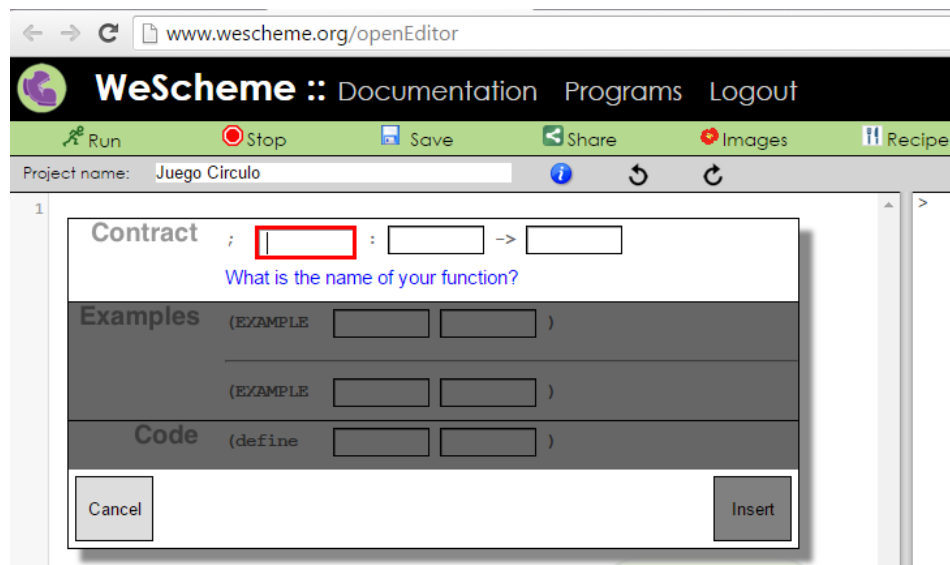
Este ambiente permite utilizar inicialmente formas de **recetas de diseño** sencillas, las cuales proporcionan una inducción a **recetas de diseño** más elaboradas.

Estas recetas se implementan con el botón **Recipe**



Esta sección **Recipe**, contiene 3 bloques que son los siguientes:

- **Contract** (Contrato): Esta sección indica el nombre de la función que se necesita crear y los valores de entrada y salida.
- **Examples** (Ejemplos): Esta sección indica las pruebas que se necesitan para verificar la función.
- **Code** (Código): Esta sección es donde se ingresa el código del programa.



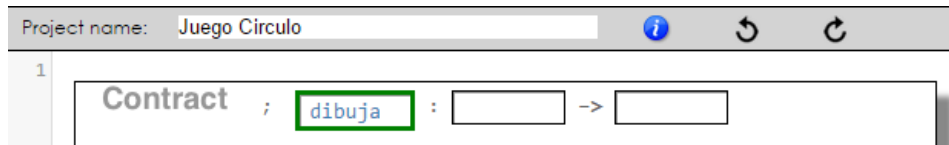
Retomando el problema de programación que se necesita resolver, el cual es crear un círculo que incremente su tamaño con respecto al paso del tiempo.

La **receta de diseño** que se emplea en esta sección permite crear funciones de una manera dinámica; una función es una parte de un programa que devuelve y/o recibe valores, también poseen un nombre específico, y se diseñan de la siguiente forma:



### **Bloque Contract:**

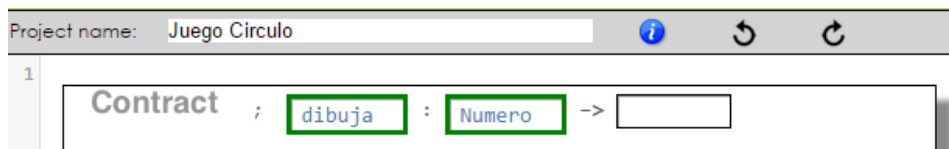
El nombre de la función que se desea generar será **dibuja** esta palabra se ingresa en el primer rectángulo del bloque.



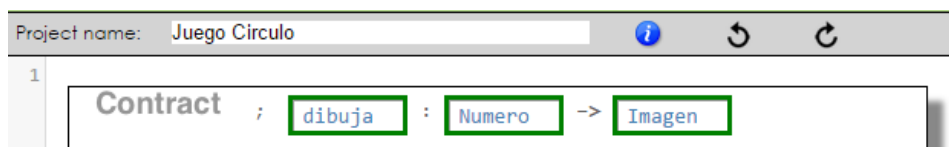
Para dibujar un círculo se tiene la siguiente instrucción:

( **Figura** **Radio** **"Tipo"** **"Color"** )  
    ↓      ↓      ↓      ↓  
( **circle** **5** **"solid"** **"blue"** )

Para que el círculo incremente su tamaño se necesita que el radio modifique su valor con respecto a tiempo, por lo cual, el valor de entrada de la función **dibuja** será un tipo de dato **Numero** se ingresa en el segundo rectángulo del bloque.



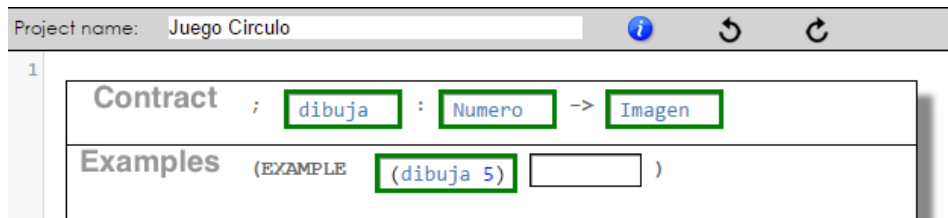
El resultado que se espera de la función **dibuja** es un círculo por lo cual, el tipo de dato es una **Imagen** se coloca en el tercer rectángulo del bloque.



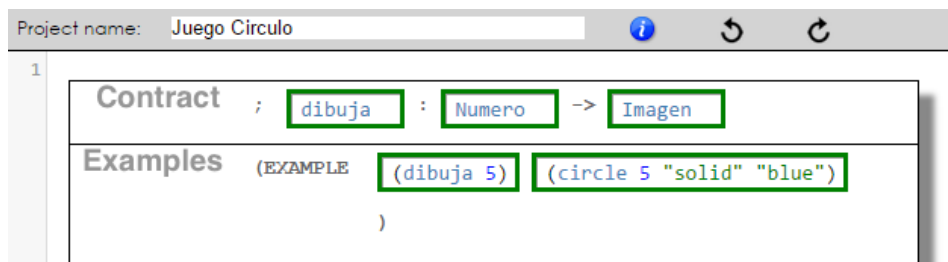
### Bloque *Examples*:

Las pruebas sirven para verificar que la función **dibuja** esté correctamente implementada, para llenar este bloque es necesario apoyarse en el bloque **Contract**, se tiene el nombre de la función y el tipo de valores de entrada y salida.

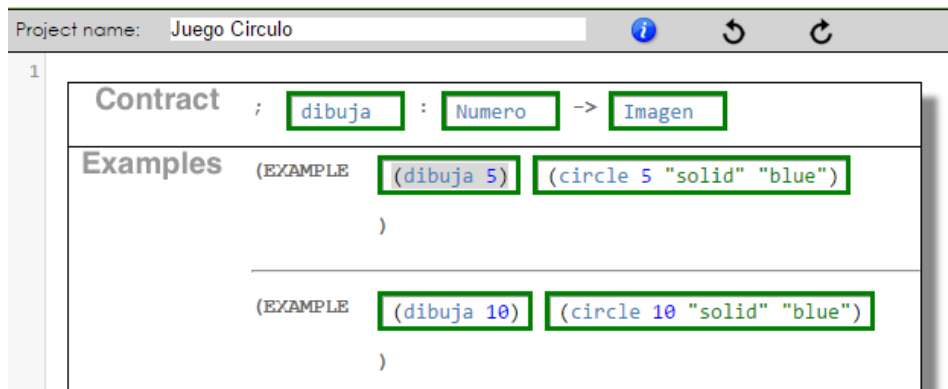
Se sabe que el nombre de la función es **dibuja** y el valor de entrada un número cualquiera que definirá el radio del círculo, se piensa que puede ser un círculo de radio 5, con esto se llena el primer rectángulo del bloque.



El segundo rectángulo se llena pensando qué resultado producirá la función, para este ejemplo, esperamos que la función **dibuja** con radio 5 produzca un círculo con tamaño de radio 5.



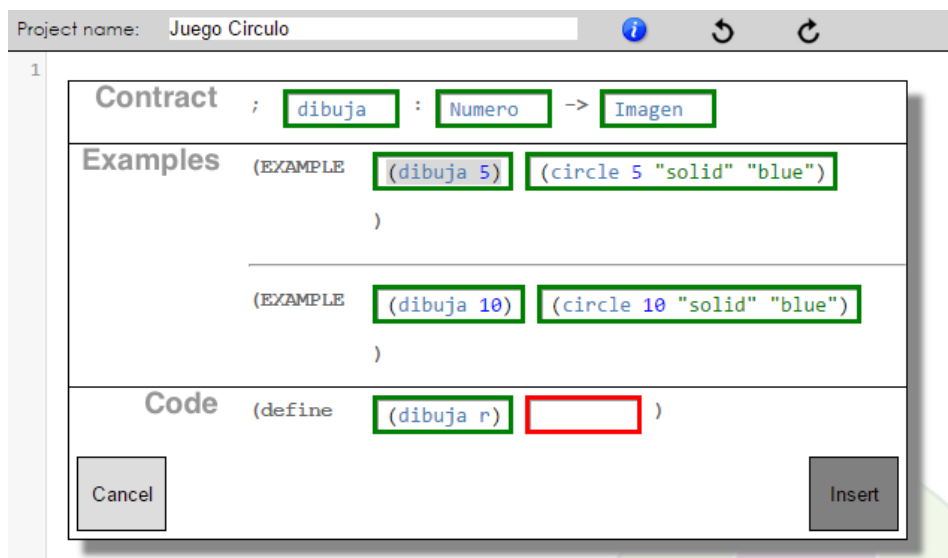
Para completar este bloque, cómo es un ejemplo muy similar al primero solo se cambia el valor del radio. Como se necesita incremento, se pondrá un círculo de radio 10.



### Bloque Code:

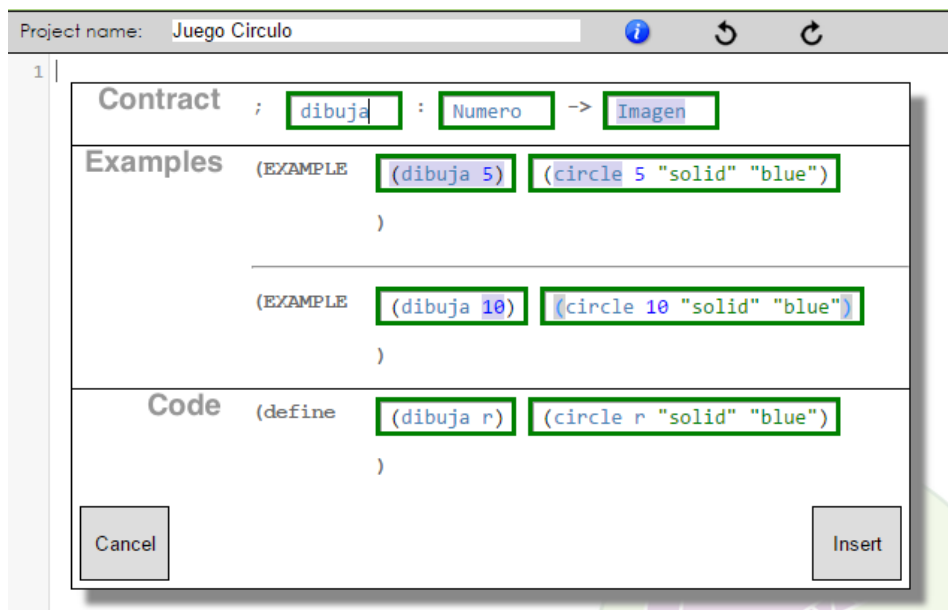
Para llenar esta sección se toman como referencia los bloques **Contract** y **Examples**, observándose que el valor del radio es variable, esto significa que puede tomar cualquier valor.

Utilizando el primer ejemplo realizado en la sección examples (**dibuja 5**), se cambia el valor de 5 por cualquier letra para este caso se utilizara **r (dibuja r)** esto quiere decir que **r** puede tomar cualquier valor que se le proporcione como valor de entrada.

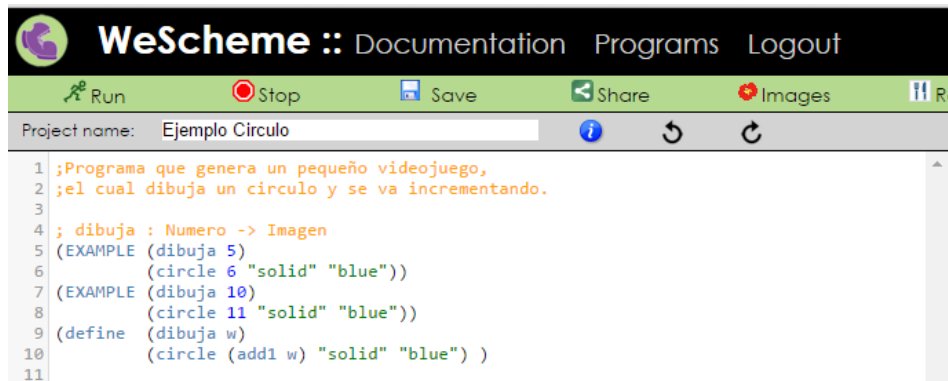


Se toma el siguiente rectángulo del primer ejemplo donde ( *circle 5 "solid" "blue"* ) se sustituye el valor 5 por *r* ( *circle r "solid" "blue"* ) para que el círculo se produzca en función del valor que tome *r*.

Se llena el segundo rectángulo y se termina el diseño de la función *dibuja* la cual produce un círculo.



Al presionar el botón de *Insert* se genera la función *dibuja* con la *receta de diseño*.



El código de la función **dibuja** se divide en las siguientes partes

```
(define (dibuja r)  
  (circle r "solid" "blue"))
```

**dibuja** —————> Nombre de la función.

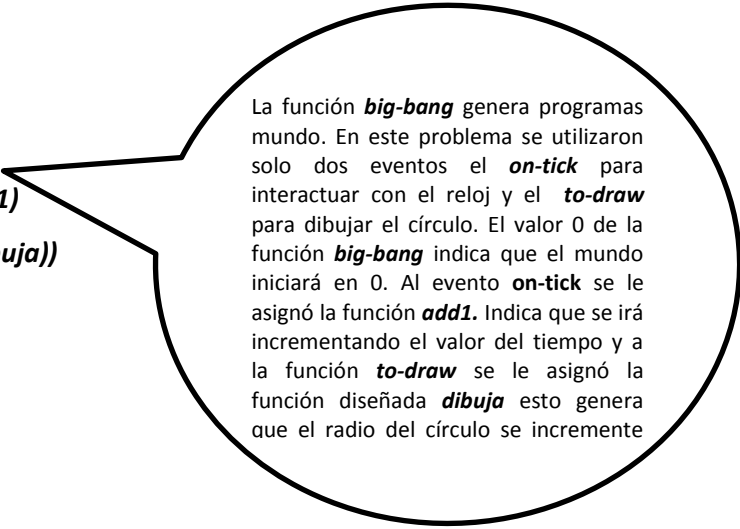
**r** —————> Valor de entrada.

(**circle r "solid" "blue"**) —————> Cuerpo de la función.

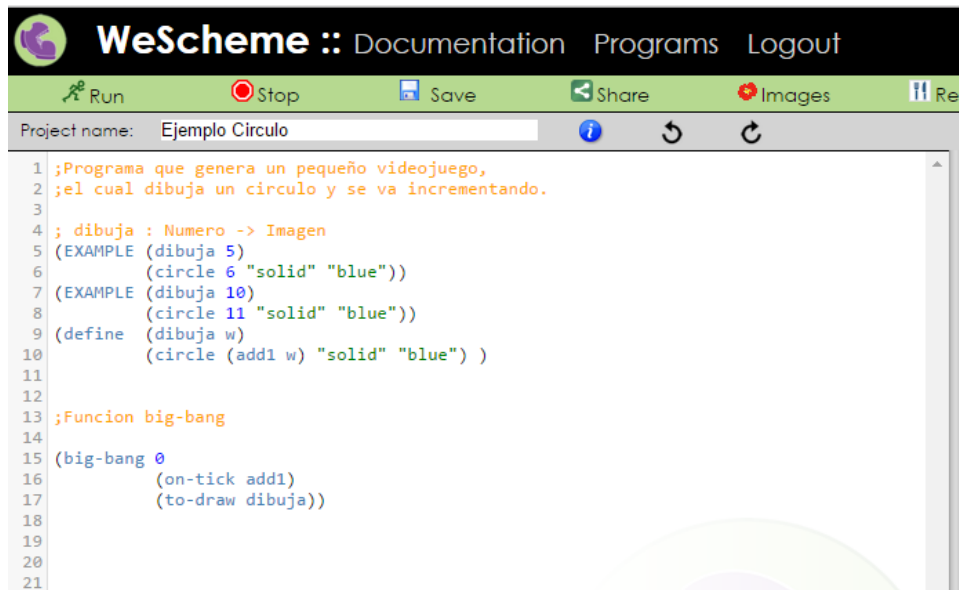
Para generar el primer videojuego, utilizaremos el **framework** conocido como la función **big-bang** que se encarga de manejar **programas mundo**, esto se explica más a detalle en el capítulo 7.

Se maneja la función **big-bang** con solo dos eventos:

```
(big-bang 0  
  (on-tick add1)  
  (to-draw dibuja))
```

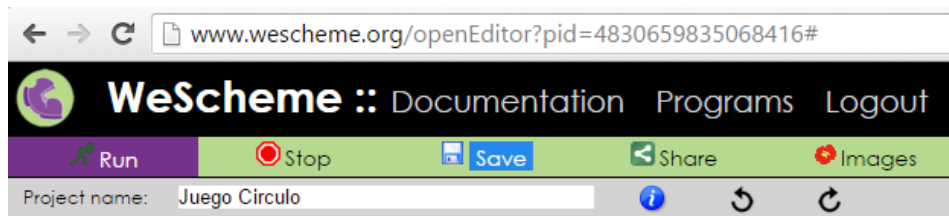


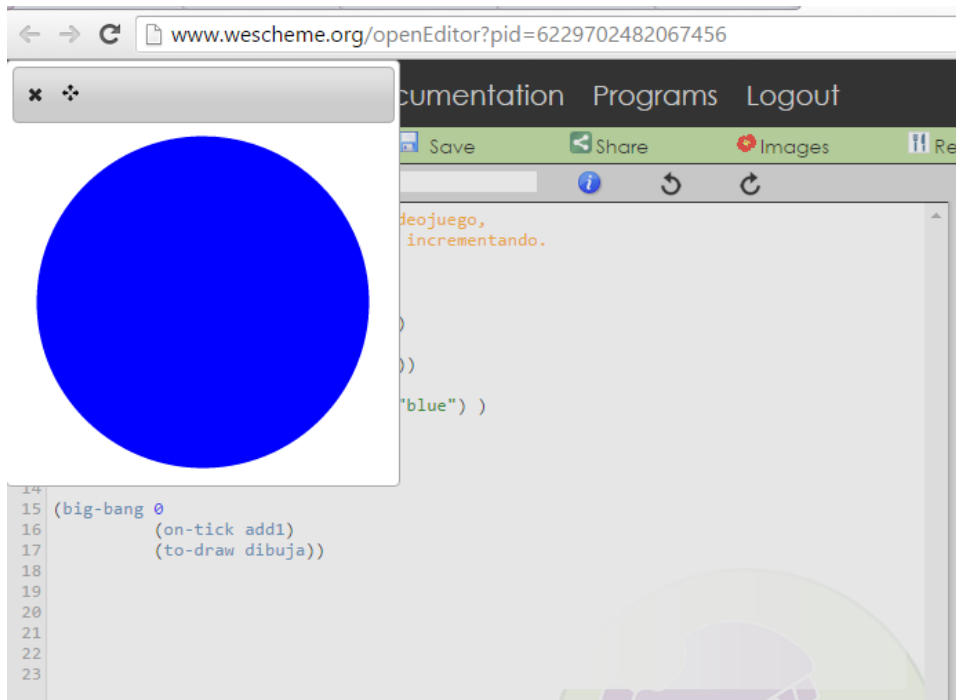
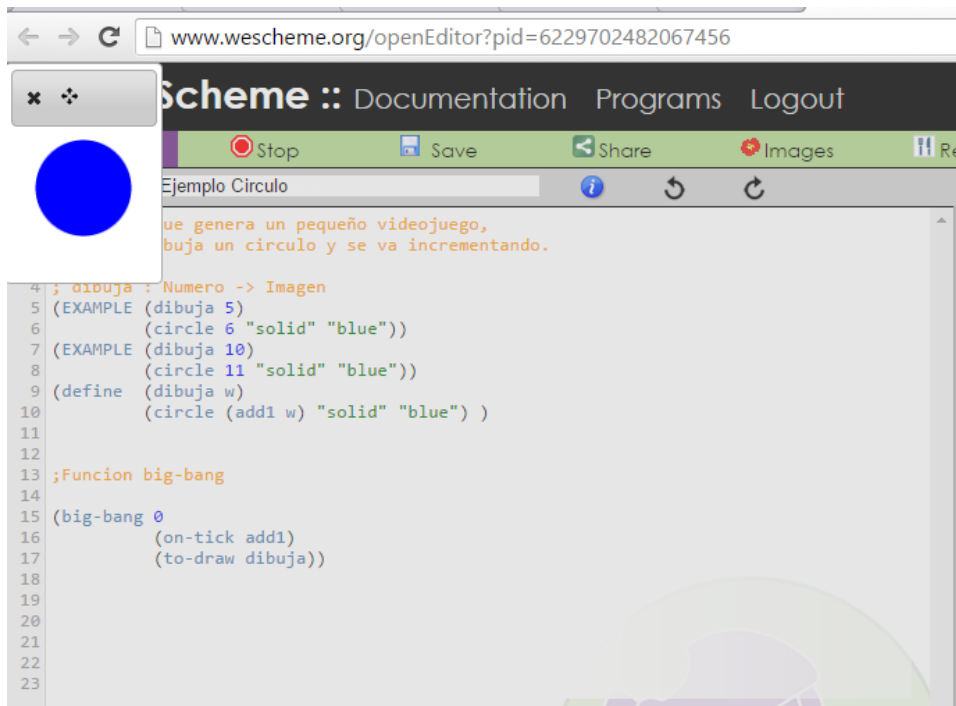
La función **big-bang** genera programas mundo. En este problema se utilizaron solo dos eventos el **on-tick** para interactuar con el reloj y el **to-draw** para dibujar el círculo. El valor 0 de la función **big-bang** indica que el mundo iniciará en 0. Al evento **on-tick** se le asignó la función **add1**. Indica que se irá incrementando el valor del tiempo y a la función **to-draw** se le asignó la función diseñada **dibuja** esto genera que el radio del círculo se incremente



```
1 ;Programa que genera un pequeño videojuego,  
2 ;el cual dibuja un círculo y se va incrementando.  
3  
4 ; dibuja : Numero -> Imagen  
5 (EXAMPLE (dibuja 5)  
6           (circle 6 "solid" "blue"))  
7 (EXAMPLE (dibuja 10)  
8           (circle 11 "solid" "blue"))  
9 (define (dibuja w)  
10         (circle (add1 w) "solid" "blue") )  
11  
12  
13 ;Funcion big-bang  
14  
15 (big-bang 0  
16         (on-tick add1)  
17         (to-draw dibuja))  
18  
19  
20  
21
```

Al presionar el botón **Run**, el videojuego se ejecuta y se observa como el círculo va creciendo de tamaño conforme paso del tiempo.





### 3.1 Importancia de las expresiones en la programación

Los programas de computadora se componen de **expresiones** y **definiciones**.

Las **expresiones** son piezas de código que producen un valor. Al igual que en las clases de matemáticas, las **expresiones** pueden ser simples números como 4 y 9 (4 igual a 4 y 9 igual a 9). Estas **expresiones** pueden ser más complejas usando funciones de sumar y restar números (4 + 9 igual a 13)<sup>5</sup>.

El lenguaje de programación **Racket** funciona como las matemáticas, todo es una función o un valor. Las funciones pueden ser + (suma), - (resta), \* (multiplicación) o / (división), pero también se puede usar funciones más complejas.

Los valores son números como -54, 2/3, 0.66, o 42, y cualquier valor entre comillas es un **String** o **cadena** incluso, "42" porque está entre comillas.

### 3.2 Herramientas auxiliares para codificar

Podemos dibujar un diagrama de expresiones matemáticas y de programación usando un círculo de evaluación véase la siguiente figura:



Círculo de evaluación<sup>6</sup>

Este es un ejemplo del círculo de evaluación para la **expresión** matemática

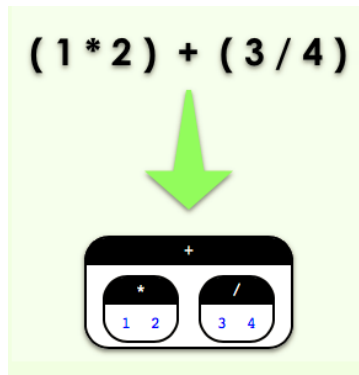
<sup>5</sup> Bootstrap, (Bootstrap, 2015), recuperado de <http://www.bootstrapworld.org/materials/fall2015/tutorial/#sección5>

<sup>6</sup> Bootstrap, (Bootstrap, 2015), recuperado de <http://www.bootstrapworld.org/materials/fall2015/tutorial/#sección5>



9 x 4. El círculo de evaluación muestra la estructura de una expresión, todos los círculos tienen dos reglas.

- Cada círculo debe tener una función que se encuentra en la parte superior.
- Las entradas se encuentran en la parte de abajo del círculo y se escriben de izquierda a derecha.



Ejemplo de la expresión  $(1 \times 2) + (3/4)$ <sup>7</sup>

Los círculos de evaluación son una forma de ver y leer computación, el orden de las operaciones para construirlos es:

Convertir los círculos de evaluación en código:

1. Se abren paréntesis.

( )

2. Se toma la función que se encuentra en la parte superior de círculo y se coloca en el inicio del paréntesis.

(<Función> )

---

<sup>7</sup> Bootstrap, (Bootstrap, 2015), recuperado de <http://www.bootstrapworld.org/materials/fall2015/tutorial/#sección5>

3. Se toman las entradas del círculo y se colocan de izquierda a derecha.

**( <Función> <Entrada 1>... <Entrada n> )**

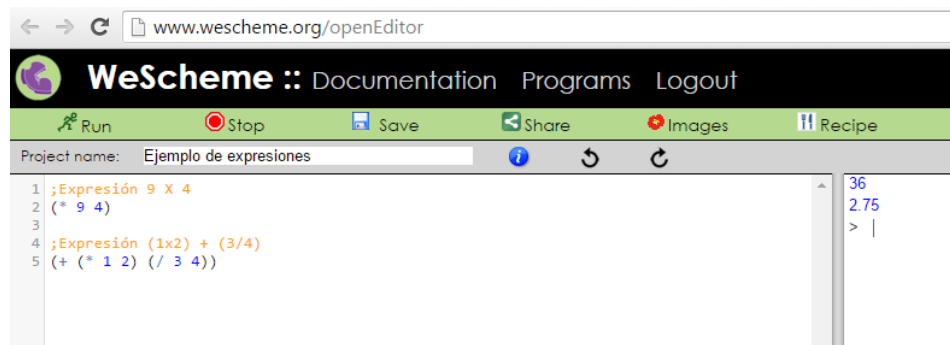
Convirtiendo en código los ejemplos de arriba se tiene:

**; Expresión 9 X 4**

**( \* 9 4 )**

**; Expresión (1x2) + (3/4)**

**( + ( \* 1 2 ) ( / 3 4 ) )**



### 3.3 Evaluación de expresiones

La forma de evaluar las expresiones para obtener el resultado es la siguiente:

Se tiene la expresión.

**( + ( \* 2 3 ) ( / 8 2 ) )**

Se inicia evaluando con los paréntesis internos de izquierda a derecha, en este caso sería con **( \* 2 3 )**.

**( + 6 ( / 8 2 ) )**

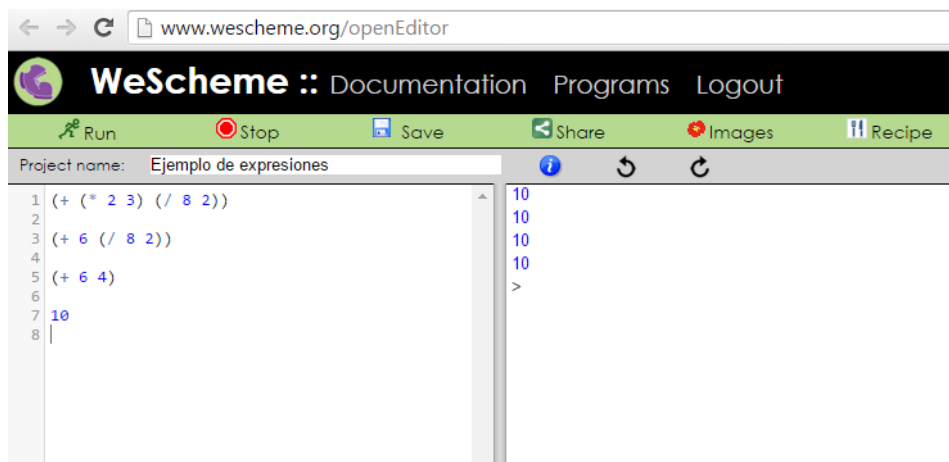
Se continúa con el siguiente paréntesis interno ( / 8 2 ).

$$( + 6 4 )$$

Y para finalizar, se evalúan los paréntesis externos ( + 6 4 ).

10

Codificando esta **expresión** en el ambiente **WeScheme** y evaluando paso por paso se observa que el resultado siempre será el mismo.



The screenshot shows the WeScheme web editor interface. The browser address bar displays `www.wescheme.org/openEditor`. The page header includes the WeScheme logo and navigation links for [Documentation](#), [Programs](#), and [Logout](#). Below the header is a toolbar with icons for [Run](#), [Stop](#), [Save](#), [Share](#), [Images](#), and [Recipe](#). The main area is divided into two sections: a code editor on the left and an output window on the right. The code editor shows the following code:

```
1 (+ (* 2 3) (/ 8 2))
2
3 (+ 6 (/ 8 2))
4
5 (+ 6 4)
6
7 10
8 |
```

The output window on the right displays the result of the evaluation, which is `10`, repeated four times, followed by a greater-than sign (`>`).

*"Hazlo simple, tan simple como sea posible, pero no más"*  
*-Albert Einstein-*

#### **4. WESCHEME**

**WeScheme** es un ambiente de programación que se ejecuta en cualquier explorador, está diseñado para soportar en tiempo real aplicaciones interactivas de fácil acceso para cualquier persona. Esta herramienta permite un manejo fácil y distribuido de aplicaciones entre los estudiantes de programación, esto es muy similar a los manejadores de versiones que se tienen hoy en día en el área del software, ya que permite compartir el programa por correo electrónico. Así mismo, otra persona puede modificarla y actualizarla sin ningún problema. Este software no necesita instalar algún intérprete o algún manejador de versiones, todos sus componentes corren en la web.

Las ventajas de **WeScheme** usado en la enseñanza de la programación es que facilita el desarrollo, compilación y ejecución desde cualquier dispositivo que cuente con un navegador, esto beneficia a los estudiantes y profesores, ya que

pueden almacenar sus aplicaciones en la nube y utilizarlas en cualquier lugar que les permita el acceso a estos archivos.

#### 4.1. WeScheme para el aprendizaje

Existen ambientes muy similares a **WeScheme**, editores, compiladores e intérpretes que permiten al estudiante acercarse a la programación sin necesidad de instalar algún componente que dificulte su aprendizaje.

En algunos casos instalar herramientas de programación hace que los estudiantes lo vean complicado y pierdan el interés aprender a programar.

Estos son algunos de los ambientes similares a **WeScheme**<sup>8</sup>:

- **Try Ruby** : Es un compilador en la Web el cual ofrece un curso de inducción al lenguaje **Ruby** permitiéndonos aprender éste y compilar en el explorador.
- **Try Haskell**: Es un compilador en la web inspirador en **Try Ruby** que ofrece un curso didáctico para el aprendizaje del lenguaje de programación **Haskell**.
- **W3schools**: Es una web que ofrece cursos para aprender: **Html, Css, Javascript, Sql, Php, Bootstrap** los cuales pueden correr los ejercicios y modificarlos en tiempo real desde la web.
- **Ideone**: Es un compilador en la Web para aprender el lenguaje **JAVA**, nos ofrece un entorno fácil de usar en el cual podemos correr

---

<sup>8</sup> Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, Kathi Fisler. (2011). WeScheme: The Browser is Your Programming Environment. 2011, de Conference on Innovation and Technology in Computer Science Education Sitio web: <http://cs.brown.edu/~sk/Publications/Papers/Published/yskf-wescheme/>

programas en tiempo real con tan solo tener un explorador. La condición es que estos programas estén hechos en lenguaje **JAVA**.

## 4.2 Ventajas de usar WeScheme

Al momento de buscar herramientas que facilitaran el aprendizaje de la programación encontramos este poderoso ambiente de programación que se apoya en el lenguaje de programación **Racket**; este lenguaje es sencillo y está especializado en el aprendizaje de la programación, enfocándose en el diseño de programas no en aprender un lenguaje de programación en particular.

Al enfocarse en aprender un lenguaje de programación, en ocasiones se complica aprender otros lenguajes de programación porque el enfoque solo está en la sintaxis del lenguaje, si se aprende una metodología de programación enfocada al diseño de programas no pensando en el lenguaje, se facilita transformar los programas en cualquier otro lenguaje.

Más adelante se explica la metodología fundamentada en años de investigación que ofrece **Racket** en el aprendizaje y enseñanza de la programación.

## 4.3 Manejo de WeScheme

### Primer paso:

Cumplir con los siguientes requisitos:

- Tener algún dispositivo que tenga un explorador Web instalado.
- Ganas de Aprender a diseñar programas.

### Segundo paso:

Se ingresa a la siguiente liga:

- **Url : <http://www.wescheme.org/>**

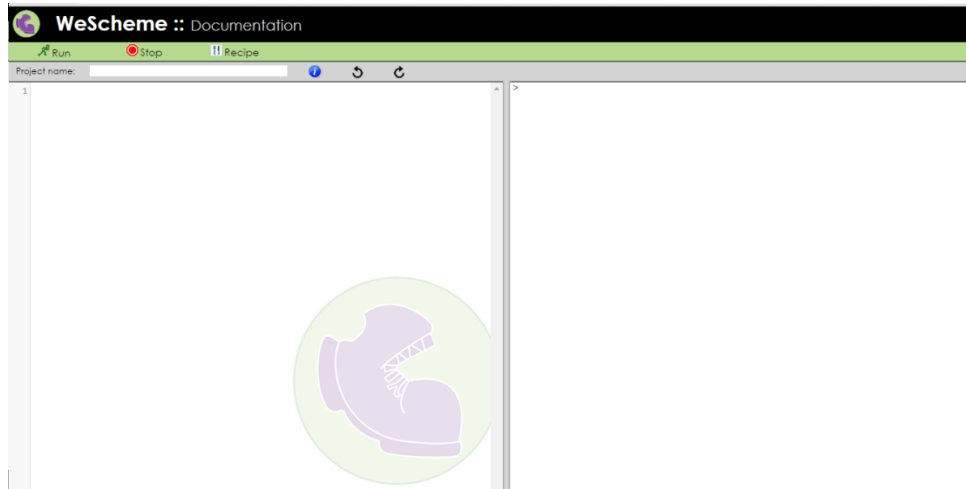
**Tercer paso:**

En la pantalla de inicio existen dos botones **Start Coding** y **Log in** como muestra la figura:



**Start Coding:**

Esta opción permite iniciar la creación del programa inmediatamente, nos direcciona al editor del ambiente de programación:

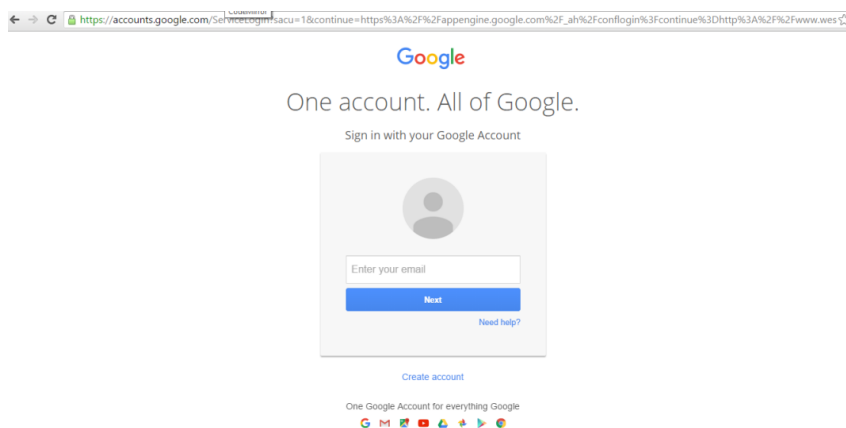


### **Log In:**

Esta opción permite tener todas las ventajas de la herramienta: guardar programas en la nube, iniciar en el desarrollo y uso de aplicaciones distribuidas, compartir y modificar código entre personas.

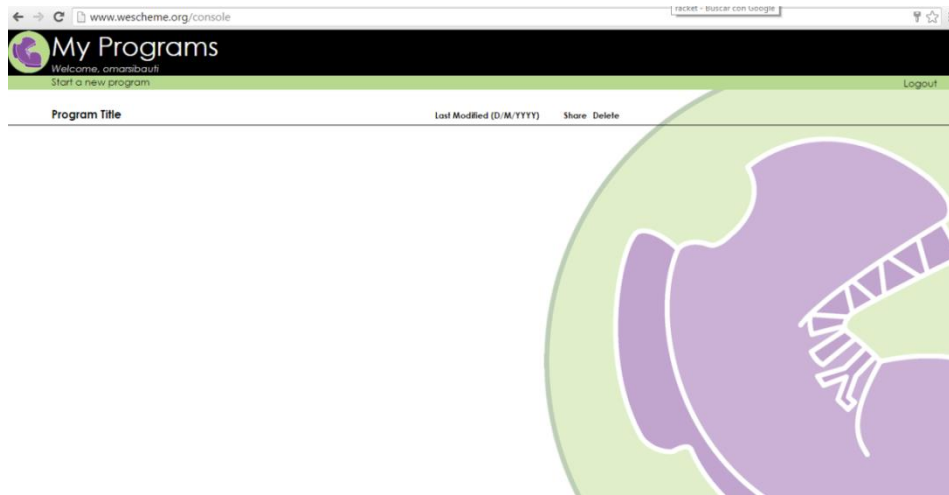
Esto nos induce al uso de Manejadores de Versiones de software.

Se pide una cuenta de Google para ingresar si se cuenta con una, solo se llenan los datos, y si no se tiene se puede crear en ese momento.



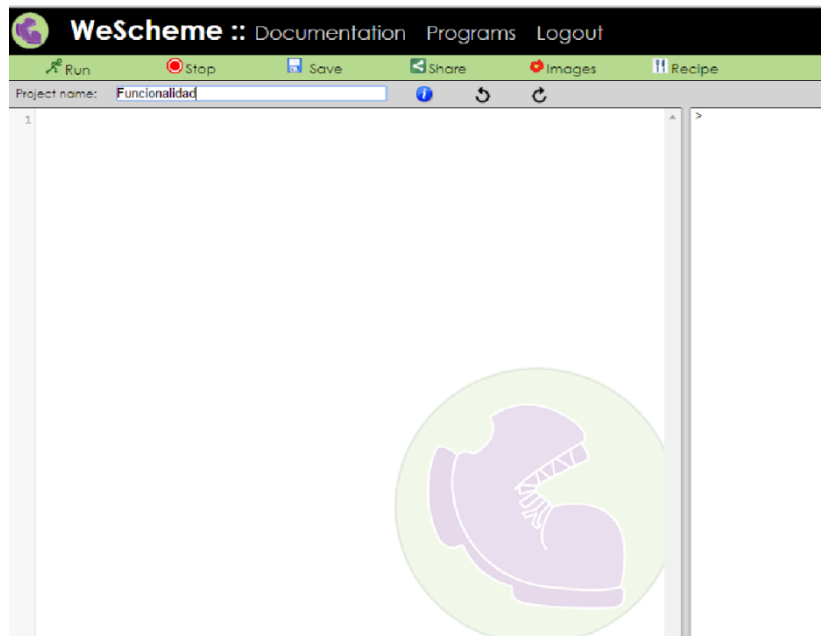


Cuando se ingresa al ambiente de desarrollo se tiene una lista con los programas guardados, y se pueden crear programas nuevos más todas las ventajas que este ambiente ofrece.



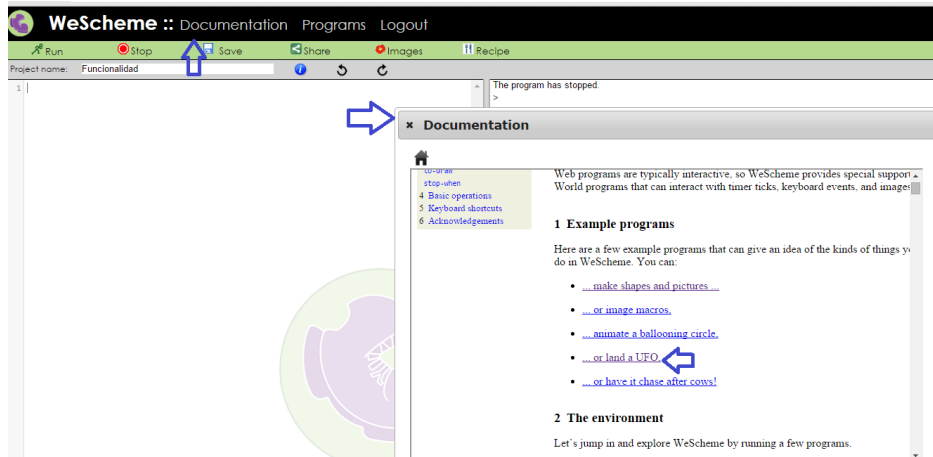
#### 4.4 Funcionalidad de WeScheme

El siguiente ejemplo muestra todas las ventajas que brinda el ambiente de desarrollo **WeScheme**. Se Ingresa en la opción **Start a new program** y en la etiqueta **Project name** se introduce el nombre del programa, este se llamará **Funcionalidad**.

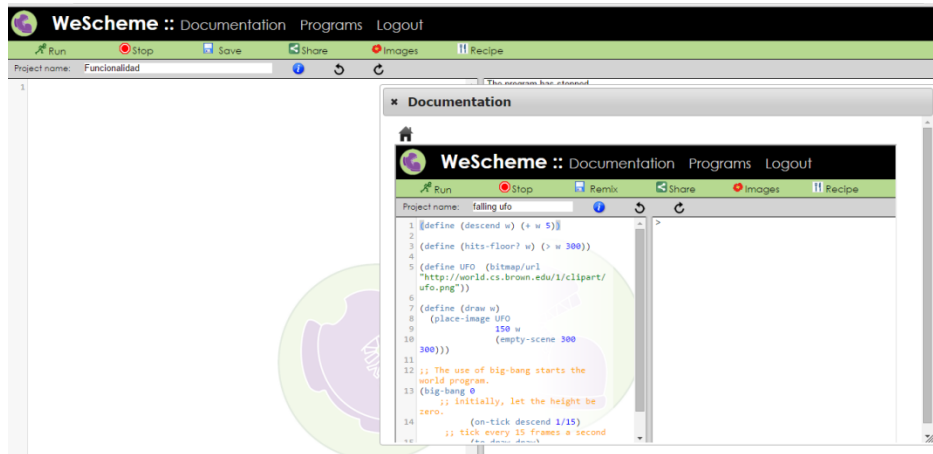


Se utilizó un ejemplo el cual no provee la documentación del ambiente de desarrollo. En el capítulo 6 se explica a detalle la construcción de estos programas.

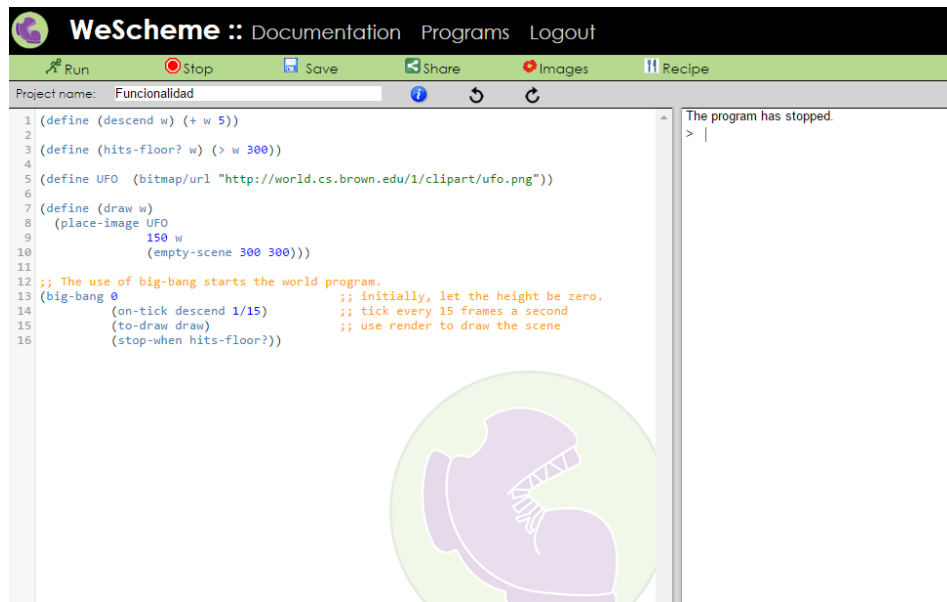
Posteriormente se ingresa en el botón **Documentation**, inmediatamente después, se muestra una ventana, la cual contiene la documentación que proporciona el ambiente. En la sección de **Example programs** se ingresa a la opción...**or land a UFO**.



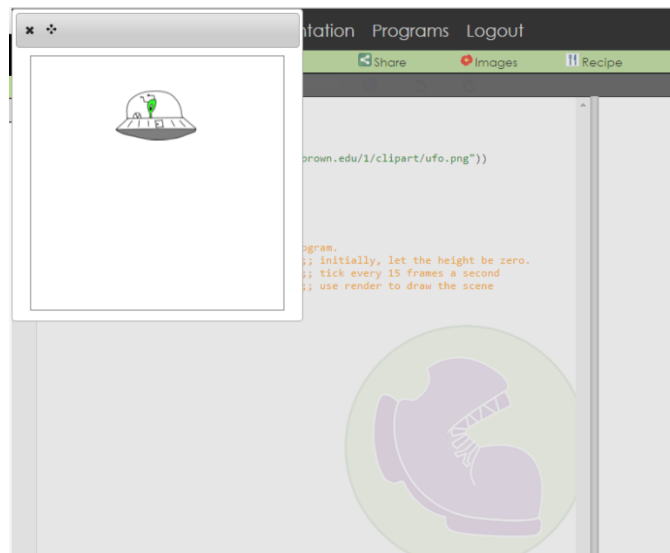
En esta sección se abre un ambiente de desarrollo pequeño que contiene la misma funcionalidad, esto es de ayuda al programar, para no interferir con el código, se pueden probar funciones y ejemplos de aplicaciones de funciones individualmente.



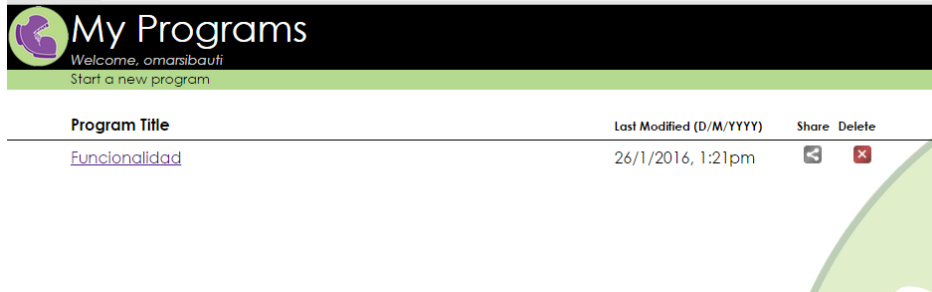
Se copia el ejemplo que proporciona la herramienta, y se pasa al editor del ambiente de desarrollo.



Si se presiona el botón **Run**, automáticamente se ejecuta el programa de ejemplo en tiempo real desde el explorador.



Si se presiona el botón **Save**, se guarda el programa y lo agrega a una lista al inicio del ambiente en la sección de **My Programs**.

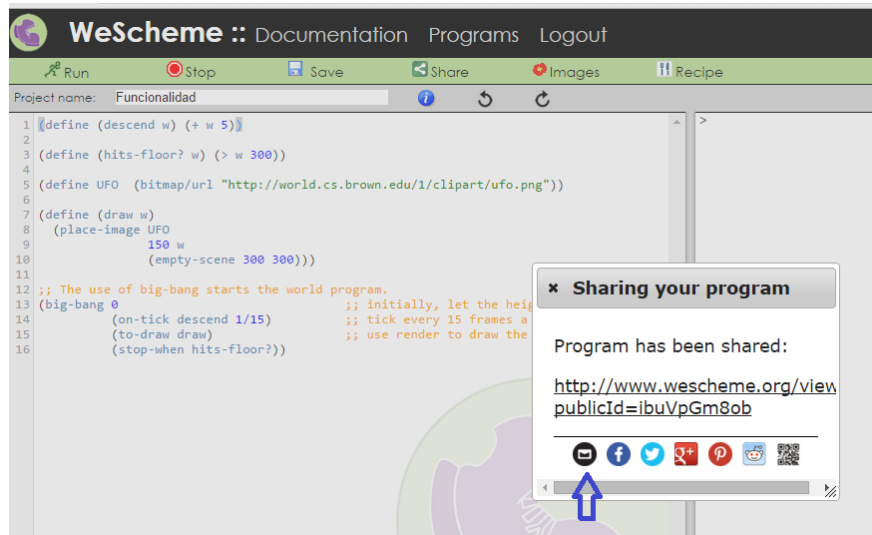


Las opción que permite compartir con otras personas y en redes sociales, brinda una clara y fácil inducción al mundo de las aplicaciones distribuidas, que es muy importante para el mundo del software, esto es que lo podemos realizar desde cualquier dispositivo. El siguiente ejemplo muestra un caso de cómo se comparte un programa vía correo electrónico.

#### **Compartir vía correo electrónico:**

En un proyecto entre dos estudiantes (un estudiante A y un estudiante B), al estudiante B se le presenta un inconveniente, por lo cual tiene que salir fuera del país. El estudiante A realiza alguna parte del proyecto pero es urgente que el estudiante B lo finalice, pues la fecha de entrega es en tres días. Este problema se resuelve muy fácilmente de la siguiente forma:

El estudiante A comparte lo que tiene vía correo electrónico con el estudiante B.



Al estudiante B le llega un correo electrónico en el cual notifica al estudiante A que ha finalizado su parte y él puede terminar.

Hola Estudiante B:

Ya termine mi parte te comparto el avance para finalices con el proyecto.

Saludos

<http://www.wescheme.org/view?publicId=ibuVpGm8ob>

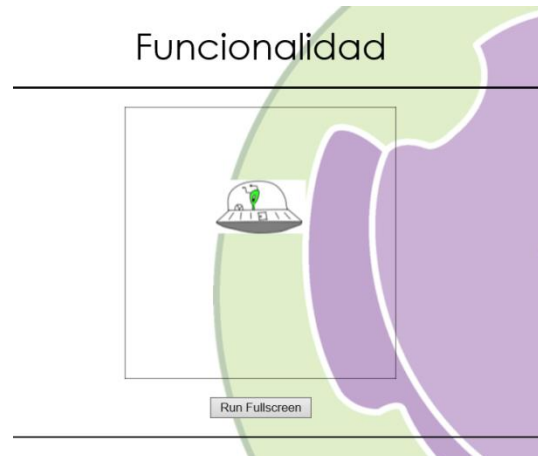
El estudiante B presiona la liga que lo direcciona al ambiente de desarrollo dentro del proyecto, éste ambiente le proporciona dos opciones Ejecutar o editar.

# Funcionalidad

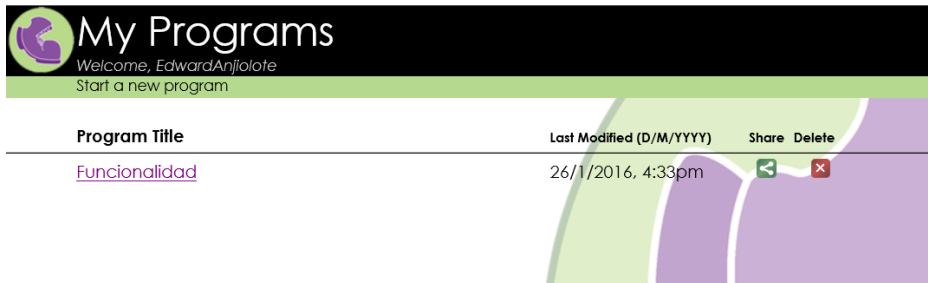


Sometimes YouTube. Perhaps iPhone. Together, WeScheme!

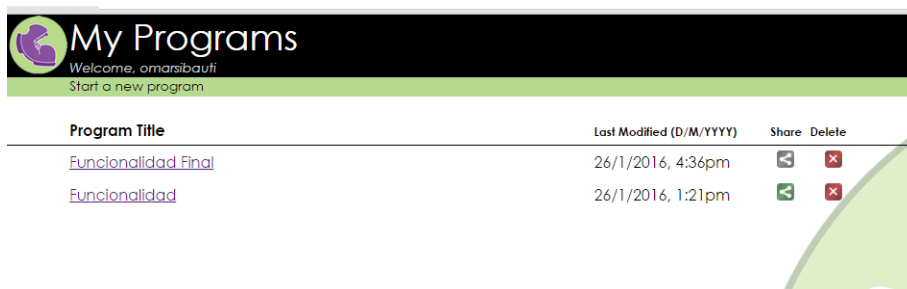
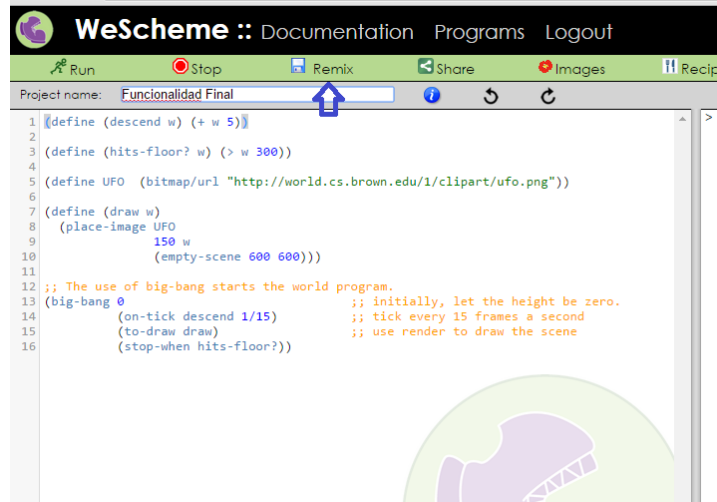
Si se selecciona **Play**, el programa se ejecuta en la pantalla del explorador.



Si se selecciona **Edit**, el programa se abre automáticamente y se encuentra listo para modificarse. Al presionar el botón **Remix**. El programa pasa automáticamente a su lista de programas. Ahora el Estudiante B realiza unas modificaciones y lo vuelve a compartir con el Estudiante A.

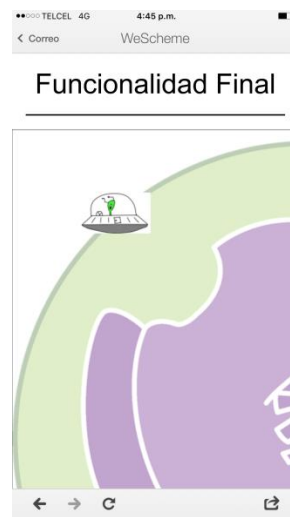
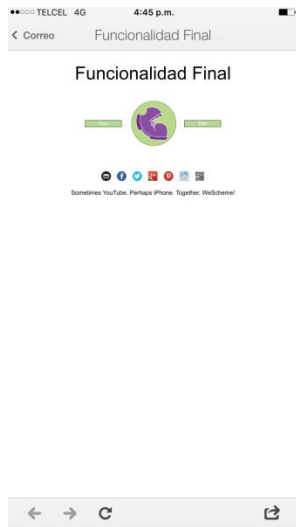


El estudiante A recibe la notificación, abre el programa, revisa los cambios y le pone un nuevo nombre al proyecto: **Funcionalidad Final**, presiona el botón **Remix** y automáticamente se genera una nueva versión de programa



El estudiante A manda esta versión final al profesor vía mail para fines prácticos el profesor revisa en el celular, presiona el botón **Play** y se da cuenta que el proyecto funciona correctamente.





Este ejemplo mostró una breve introducción a la funcionalidad del ambiente de desarrollo **WeScheme**, se creó y se ejecutó un programa en el explorador Web, se compartió, se modificó y se generó una nueva versión, también fue ejecutado en un dispositivo móvil que contaba con un explorador Web.

Contar con portabilidad de este tipo en el software facilita mucho el trabajo en equipo porque en la actualidad los sistemas son enormes, esto requiere trabajar con equipos grandes, en ocasiones vía remota, de ciudad a ciudad o país a país, etc. Esto facilita a los desarrolladores tener un mejor control del código que se genera al día y un buen manejo de las versiones del sistema, además de facilitar la detección de errores en el código para una pronta solución.

*"Todo el mundo debería aprender a programar,  
porque eso te enseña a pensar"*  
-Steve Jobs-

## 5. RECETAS DE DISEÑO

La metodología que se muestra para el diseño de programas está basada en **recetas de diseño**, estas recetas de diseño ayudan a resolver problemas desde una lógica de programación enfocada al diseño.

Una **receta de diseño** de programas ayuda al planteamiento de la solución de algún problema de programación, esto permite pensar primero en el problema antes de empezar con la codificación, ya que este es un error muy común que cometen los programadores y estudiantes de programación.

## 5.1 Cómo diseñar funciones

La **receta de diseño** es una metodología que permite crear **funciones**. Una **función** se define como una parte de un programa (subrutina) que devuelve o recibe valores, poseen un nombre específico y se hace uso de ella desde otras partes del programa las veces que se necesite.

La **Receta de diseño** o cómo **diseñar funciones** consiste en los siguientes pasos:

1	Firma, propósito, borrador (stub)
2	Pruebas (check-expect)
3	Formato
4	Cuerpo de la función
5	Pruebas y Ejecución

El siguiente ejemplo muestra paso a paso cómo se emplea la **receta de diseño**.

Ejemplo<sup>9</sup>:

Diseñar una **función** que reciba un carácter **string** "Hola" y le agregue el carácter "!" y devuelva la **cadena** o **String** "Hola!"

Solución:

**Paso 1: Firma, propósito, stub:**

La **firma** contiene los tipos de datos que recibe y regresa la función:

***String - > String***

---

<sup>9</sup> Gregor Kiczales Erika Thompson. (2015). Systematic Program Design Part 1. 2015, de EDX Sitio web: <https://courses.edx.org/courses/course-v1:UBCx+SPD1x+1T2016/info>

El **propósito** es un breve resumen de lo que genera la función:

***Agrega "!" al final de un String***

El **stub** es la definición de la función que produce un valor del tipo correcto.

***(define (agrega s) "a")***

### **Paso 2: Pruebas (*check-expect*)**

En este paso se crean las **pruebas** imaginando el resultado correcto que regresa la **función**:

***(check-expect (agrega "Hola") "Hola!")***

***(check-expect (agrega "Prueba") "Prueba!")***

### **Paso 3: Formato**

En este paso antes de codificar se piensa en el cuerpo de la **función**, esto ayuda cuando se necesita agregarle funcionalidad. Es un planteamiento basado en los pasos anteriores. El valor que recibe y el valor que regresa la función, se llenaran con tres puntos (**...**) esto indicara que en esta parte se necesita agregar lógica.

***(define (agrega s) (... s))***

### **Paso 4: Cuerpo de la función**

En este paso se completa la funcionalidad apoyándose en el **formato** y las **pruebas** realizadas.

```
(define (agrega s)
  (string-append s "!") )
```

## Paso 5: Pruebas y Ejecución

En este paso se garantiza que todas las pruebas sean correctas al momento de ejecutar el programa. En caso de error se deberá regresar nuevamente al paso 2.

El código queda de la siguiente forma.

```
;; Firma
;; String -> String

;; Propósito

;; Agrega "!" al final de un String

;; Pruebas

(check-expect (agrega "Hola") "Hola!")

(check-expect (agrega "Prueba") "Prueba!")

;; Stub

;(define (agrega s) "a")

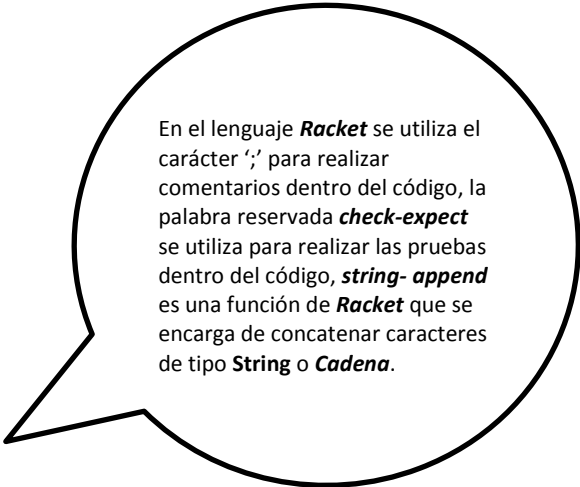
;; Formato

;(define (agrega s) (... s))

;; Cuerpo de la función

(define (agrega s)

  (string-append s "!") ).
```



En el lenguaje *Racket* se utiliza el carácter ‘;’ para realizar comentarios dentro del código, la palabra reservada *check-expect* se utiliza para realizar las pruebas dentro del código, *string-append* es una función de *Racket* que se encarga de concatenar caracteres de tipo *String* o *Cadena*.

Pasando el programa al ambiente **WeScheme**:



```
1 ;;Firma
2 ;; String -> String
3
4 ;;Propósito
5 ;; Agrega "!" al final de un String
6
7 ;;Pruebas
8 (check-expect (agrega "Hola") "Hola!")
9 (check-expect (agrega "Prueba") (string-append "Prueba" "!"))
10
11 ;;Stub
12 (define (agrega s) "a")
13
14 ;;Formato
15 (define (agrega s) (... s))
16
17 ;;Cuerpo de la función
18 (define (agrega s)
19   (string-append s "!"))
20
21
```

Al ejecutarlo solo y saber que la **función** es correcta, **WeScheme** no muestra ningún mensaje solo muestra cuando alguna prueba no es correcta, más adelante se explica cómo producir una salida del programa.

## 5.2 Expresiones dentro del diseño de funciones

Una **expresión** se define como un elemento dentro de un programa que se evalúa y produce un valor, las **expresiones** se pueden construir de la siguiente manera:

**( Operación Argumento ... )**

Con esta regla en las expresiones aritméticas podemos generar las siguientes:

**Expresiones simples:**

**( + 5 6 )** Expresión que suma 5 a 6. El resultado es 11

**( - 7 4 )** Expresión que resta 7 a 4. El resultado es 3

**Expresiones compuestas:**

Expresión que suma 3 al resultado de la multiplicación 4 por 7. El resultado es 31:

$$( + 3 ( * 4 7 ) )$$

Expresión que divide entre 8 el resultado de la multiplicación 5 por 3. El resultado es 0.53:

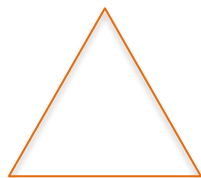
$$( / 8 ( * 5 3 ) )$$

Ejemplo:

Con las expresiones aritméticas se pueden crear **funciones** complejas para resolver problemas, el siguiente ejemplo muestra cómo crear una **función** que contenga una expresión aritmética que calcule el área de un triángulo.

Problema:

Diseñar una **función** que reciba como parámetros la base (b) y la altura (h) del triángulo para que produzca como resultado el área del triángulo.



Altura h = 3 [cm]

Base b = 5 [cm]

Solución:

Paso 1:

La función recibe dos parámetros de tipo numérico y regresa un valor tipo numérico

```
;; Number Number -> Number
```

La función calcula el área de un triángulo con los valores de la base y la altura.

```
;; Calcula el área con la base y la altura
```

La función recibe los valores de la base y la altura y regresa un valor numérico.

```
( define ( área b h ) 7.5 )
```

Paso 2:

El resultado del área de un triángulo es **(base x altura) dividido entre 2**. Con esta información se generan las pruebas siguientes:

```
( check-expect ( área 3 5 ) 7.5 )
```

```
( check-expect ( área 1 15 ) 7.5 )
```

Paso 3:

El formato se construye pensando en que es lo que recibe, que es lo que regresa la función y tomando como base el **stub**.

```
;; ( define ( área b h ) (... b h ) )
```



Paso 4:

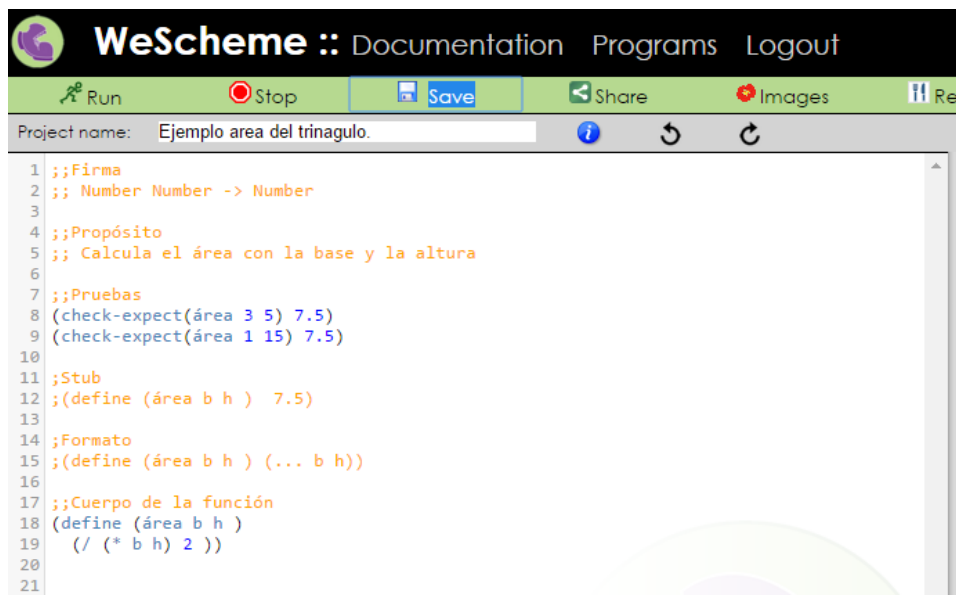
Con el paso anterior y las *pruebas*, el cuerpo de la *función* se genera de la siguiente forma:

```
( define ( área b h )  
      ( / ( * b h ) 2 ) )
```

Paso 5:

En este paso se verifica que todas las pruebas sean correctas. En caso contrario, regresar al paso 2 para modificar lo necesario.

El código de este programa en *WeScheme* se muestra de la siguiente forma:



```
1 ;;Firma  
2 ;; Number Number -> Number  
3  
4 ;;Propósito  
5 ;; Calcula el área con la base y la altura  
6  
7 ;;Pruebas  
8 (check-expect(área 3 5) 7.5)  
9 (check-expect(área 1 15) 7.5)  
10  
11 ;Stub  
12 ;(define (área b h ) 7.5)  
13  
14 ;Formato  
15 ;(define (área b h ) (... b h))  
16  
17 ;;Cuerpo de la función  
18 (define (área b h )  
19   ( / ( * b h ) 2 ) )  
20  
21
```

### 5.3 Condicionales en el diseño de funciones

Los condicionales son **expresiones**, las cuales permiten evaluar valores en función de una condición, esto ayuda a generar funciones más complejas y con un objetivo concreto.

Para comprender sobre el manejo de los condicionales en la programación el lenguaje **Racket**, permite utilizar tipos de datos **booleanos**, **expresiones** condicionales de tipo **if** entre otras.

### 5.4 Booleanos

Los **booleanos** son un tipo de datos que solo contienen dos valores **true**(verdadero) y **false**(falso), en **Racket** una de las formas más sencillas de utilizarlos es con **operaciones primitivas** o **funciones** que producen valores **booleanos** como **not**, **=**, **<**, **>**, **string=?**<sup>10</sup>.

Ejemplo:

```
( define WIDTH 100 )  
( define HEIGHT 200 )  
( > WIDTH HEIGHT )
```

Las **constantes** son valores con un nombre asignado que se pueden usar en diferente partes del programa, en **Racket** se definen de la siguiente manera:

**(define <name> <expression>)**

Ejemplo: **(define altura 150)**, el valor de la constante altura será 150,

**Racket** nos permite asignar los valores de tipo **Number**(numero), **String**(cadena) e **image**(imagen).

<sup>10</sup> Gregor Kiczales Erika Thompson. (2015). Systematic Program Design Part 1. 2015, de EDX Sitio web: <https://courses.edx.org/courses/course-v1:UBCx+SPD1x+1T2016/info>

El resultado produce un valor **false** porque el valor de la **constante WIDTH** no es mayor que el valor de la **constante HEIGHT**.

Estas **funciones** que producen valores **booleanos** se usan con cualquier tipo de dato por ejemplo:

Con datos Tipo **Number** (numérico):

- **( = 1 1 )** el valor producido es **true**.
- **( > 3 9 )** el valor producido es **false**.

Con datos tipo **String (cadena)**:

Todos los datos de tipo **String** se declaran entre " ".

- **(string=? "foo" "bar")** :  
El valor producido es **false** por que el **string "foo"** no es igual al **string "bar"**.

Con imágenes:

**Racket** permite integrar imágenes (propias o las predefinidas de **Racket**.) a los programas para facilitar el aprendizaje.

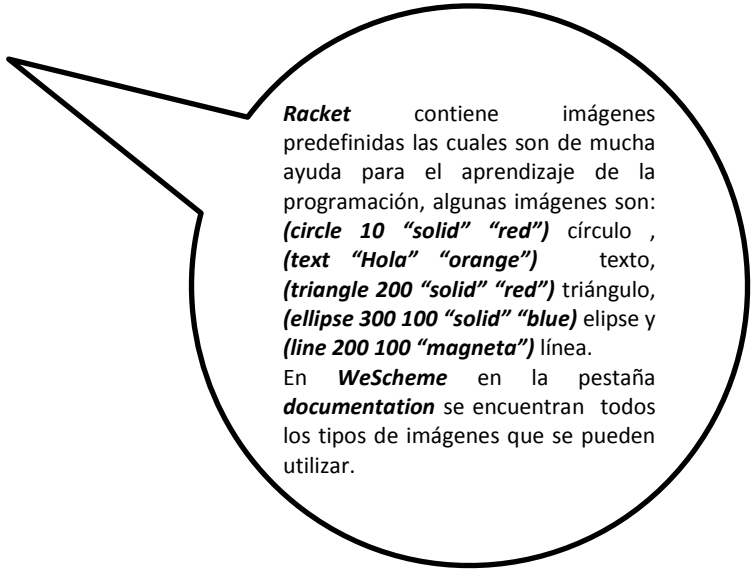
Ejemplo<sup>11</sup>:

```
(define imagen1 ( rectangle 10 20 "solid" "red"))
```

```
(define imagen2 ( rectangle 20 30 "solid" "blue"))
```

```
(< (image-width imagen1)
```

```
(image-width imagen2))
```



**Racket** contiene imágenes predefinidas las cuales son de mucha ayuda para el aprendizaje de la programación, algunas imágenes son: **(circle 10 "solid" "red")** círculo , **(text "Hola" "orange")** texto, **(triangle 200 "solid" "red")** triángulo, **(ellipse 300 100 "solid" "blue")** elipse y **(line 200 100 "magenta")** línea. En **WeScheme** en la pestaña **documentation** se encuentran todos los tipos de imágenes que se pueden utilizar.

En el ejemplo mencionado arriba se utilizan dos **constantes**, **imagen1** e **imagen2** se les asigna un valor tipo imagen. La primera es un rectángulo color rojo que mide de largo 10 y de alto 20 en la segunda se tiene un rectángulo color azul que mide de largo 20 y de alto 30, el resultado **booleano** es **true**. Se utilizó la **función image-width** que obtiene el valor del largo de la imagen, en este caso el valor es **true** porque el largo de la **imagen1** es menor que el largo de la **imagen2**.

## 5.5 Expresiones condicionales de tipo *if*

Gregor Kiczales<sup>12</sup> (Gregor Kiczales, 2015) comenta que las **expresiones condicionales tipo if** sirven para poder crear funciones que permitan condicionar datos. Se definen de la forma en que se muestra:

---

<sup>11</sup> Gregor Kiczales Erika Thompson. (2015). Systematic Program Design Part 1. 2015, de EDX Sitio web: <https://courses.edx.org/courses/course-v1:UBCx+SPD1x+1T2016/info>

```
( if < pregunta >  
  < respuesta verdadera >  
  < respuesta falsa > )
```



**Pregunta:**

En esta parte se utiliza alguna operación primitiva que genere un valor de tipo *booleano*.

**Respuesta verdadera y respuesta falsa:**

En esta parte se utilizan expresiones.

**Ejemplo:**

```
( define ovni  )  
( define vaca  )
```

*Racket* nos permite utilizar nuestras propias imágenes con el fin de suavizar el aprendizaje, dentro de *WeScheme* en la pestaña de *images* podemos insertarlas y se definen: **(define <nombre> <Imagen>)**

```
( if ( = ( image-width ovni )  
        ( image-width vaca ) )  
      "Verdadero"  
      "Falso" )
```

En este ejemplo se utilizaron dos imágenes: ovni y vaca, las cuales tienen las mismas medidas de largo, por lo cual al evaluar la expresión tipo *if*, el resultado

<sup>12</sup> Profesor de ciencias de la computación de la UBC(University of British Columbia) en Canadá.

produce el **String o Cadena** "**Verdadero**", si el largo de las imágenes no fueran iguales el resultado es el **String o Cadena** "**Falso**" por ejemplo:

```
( if ( > ( image-width ovni )  
      ( image-width vaca ) )  
      "Verdadero"  
      "Falso" )
```

El resultado es "**Falso**".

Un ejemplo más básico con datos tipo **Number** (numérico) es:

```
( if ( = 5 4 )  
      "Es igual"  
      "No es igual" )
```

En este caso el número 5 no es igual al número 4, lo cual produce el resultado de la respuesta falsa.

En el uso de estas **expresiones condicionales** se utilizaron los operadores **AND** y **OR** que ayudan a condicionar más de una expresión.

El operador **AND** se utiliza cuando dos o más preguntas son verdaderas, y se define de la siguiente manera:

```
( and ( <pregunta1> <pregunta2> ... )
```

El operador **OR** se utiliza cuando necesitamos que alguna pregunta sea verdadera y se define de la siguiente manera.

```
( or ( <pregunta1> <pregunta2> ... )
```

Por ejemplo:

Se comparan dos números enteros, por ejemplo el 4 y el 5:

```
( and ( = 4 4 )  
      ( = 5 5 ) )
```

El resultado de esta **expresión** es **true** porque las dos condiciones se cumplen. Se observa que el número 4 es igual al número 4 de la primera condición y el número 5 es igual al número 5 de la segunda condición.

Para el caso del operador **OR** solo se necesita que una de las condiciones se cumpla se muestra en el siguiente ejemplo:

```
( or ( = 9 4 )  
     ( = 5 5 ) )
```

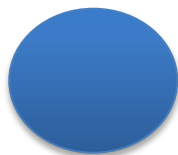
El resultado de esta expresión es **true** porque la segunda condición se cumple pues el número 5 es igual al número 5.

Ejemplo:

Diseñar una **función** que genere un triángulo o un círculo azul con los valores que recibe, recibirá como parámetros de entrada el nombre de la figura y el tamaño.

Solución:

Se necesita crear una **función** que genere las siguientes figuras:



Paso 1:

```
(circle 48 "solid" "blue") (triangle 48 "solid"
                             "blue")
```

La **función** recibe como parámetros de entrada el nombre de la figura y el tamaño que se desea.

```
;; String Number -> Image
```

El **propósito** de la función es producir un triángulo o un círculo según sea el caso.

```
;; Produce un triángulo o un círculo con los parámetros que recibe.
```

La función recibe dos valores y produce una imagen.

```
;(define (figura s n) (triangle 48 "solid" "blue"))
```

Paso 2:

Se espera que nuestra función produzca una figura, sea círculo o triángulo.

```
(check-expect (figura "solid" 48)
               (triangle 48 "solid" "blue"))
```

```
(check-expect (figura "solid" 48)
               (triangle 50 "solid" "red"))
```



Paso 3:

Se necesita que se produzcan dos imágenes, por lo que el formato quedaría de la siguiente forma:

```
;(define ( figura s n )  
  ( ... s n )  
  ( ... s n ) )
```

Paso 4:

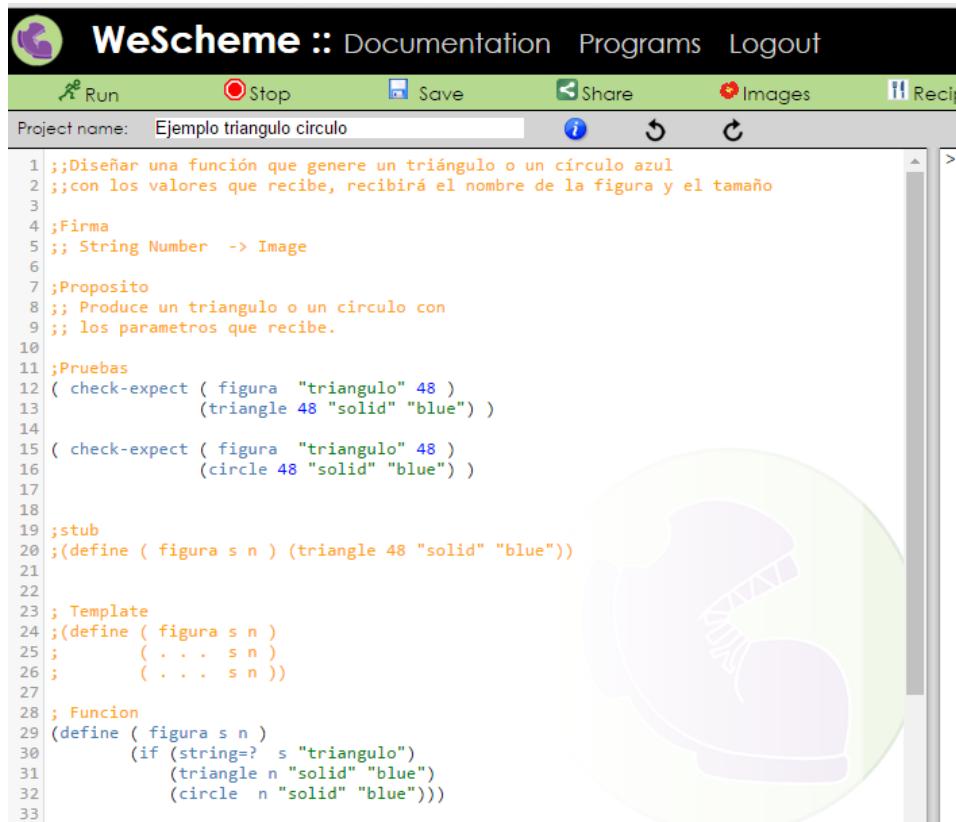
Con el paso anterior se puede generar el cuerpo de la función:

```
(define ( figura s n )  
  (if (string=? s "triangulo")  
      (triangle n "solid" "blue")  
      (circle n "solid" "blue")))
```

Paso 5:

Revisamos que todas las pruebas sean correctas.

Computarizando este código en **WeScheme** se ve de la siguiente forma:



```
1 ;;Diseñar una función que genere un triángulo o un círculo azul
2 ;;con los valores que recibe, recibirá el nombre de la figura y el tamaño
3
4 ;Firma
5 ;; String Number -> Image
6
7 ;Proposito
8 ;; Produce un triangulo o un círculo con
9 ;; los parametros que recibe.
10
11 ;Pruebas
12 ( check-expect ( figura "triangulo" 48 )
13               (triangle 48 "solid" "blue") )
14
15 ( check-expect ( figura "triangulo" 48 )
16               (circle 48 "solid" "blue") )
17
18
19 ;stub
20 (define ( figura s n ) (triangle 48 "solid" "blue"))
21
22
23 ; Template
24 (define ( figura s n )
25   ( . . . s n )
26   ( . . . s n ) )
27
28 ; Funcion
29 (define ( figura s n )
30   (if (string=? s "triangulo")
31       (triangle n "solid" "blue")
32       (circle n "solid" "blue")))
33
```

En la parte de abajo se agregó una sección que dice "**; Forma de producir salida**" esta es una forma de producir un resultado y se hace escribiendo el nombre de la **función** y escribiendo los valores que recibe:

**(<Nombre de la función> <Valor 1> ...< Valor n>)**

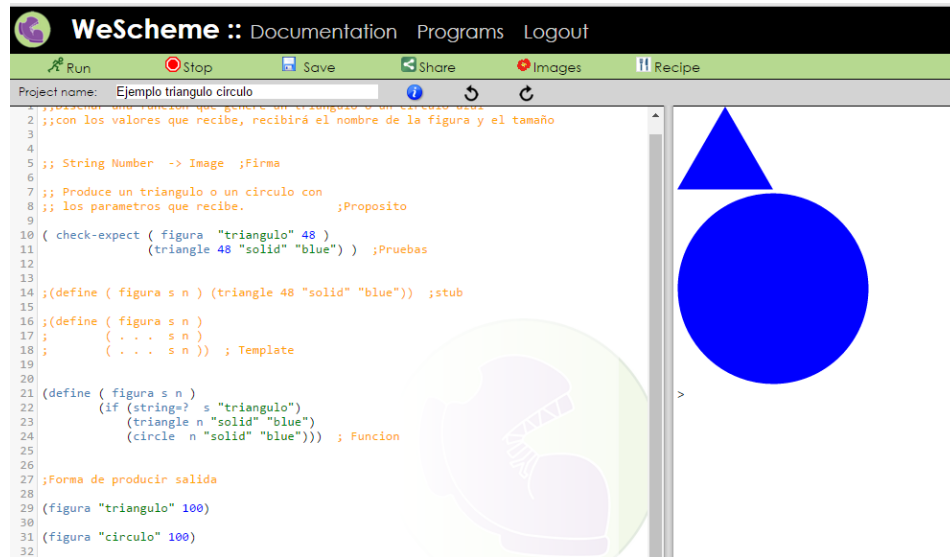
Para producir la salida de la función **figura** aplicando el formato de arriba, se tiene la siguiente forma:

Se tiene una condición de tipo **if**, por lo cual existen dos posibles salidas, un círculo o un triángulo, eso depende del primer valor de entrada que se le ponga:

***;Forma de producir Salida***

***( figura "triangulo" 100)***

***( figura "círculo" 100)***



## 5.6 Expresiones condicionales de tipo *cond*

Este tipo de expresiones ayudan cuando se tienen dos o más ***expresiones condicionales*** para un mismo problema y se definen de la siguiente forma:

***(cond [<Pregunta> <Respuesta>  
...])***

***<Pregunta>:***

En esta parte se utilizan operaciones primitivas que generen un valor de tipo ***booleano***.

**<Respuesta verdadera>:**

En esta parte se utilizan **expresiones**.

Ejemplo:

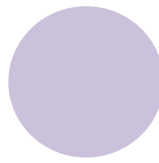
Se deberá diseñar un programa que al recibir el nombre de las figuras mostradas abajo, las produzca.



Cuadrado



Triángulo



Círculo



Estrella

Solución:

Paso 1:

La **función** recibirá una cadena y producirá una imagen:

**;; Firma**

**;; Cadena -> Imagen**

El propósito de la **función** es generar una imagen con un nombre dado:

**;;Propósito**

**;; Produce una imagen de acuerdo a su nombre**

El nombre de la **función** será **figura** recibe una cadena y dibuja una imagen:

***;*Stub**

***(define (figura c) (square 60 "solid" "mediumslateblue") )***

Paso 2:

La función dibujará 4 imágenes diferentes, por la cual se harán 4 pruebas:

***(check-expect (figura "Cuadrado") (square 60 "solid" "mediumslateblue"))***

***(check-expect (figura "Cuadrado") (triangle 80 "solid" "lightsteelblue"))***

***(check-expect (figura "Cuadrado") (circle 40 "solid" "mediumslateblue"))***

***(check-expect (figura "Cuadrado") (star 60 "solid" "lightsteelblue"))***

Paso 3:

Se toma el ***stub***, se observa que en el paso anterior se tienen 4 pruebas con las 4 diferentes figuras que se necesitan producir, se toma como base la definición de las expresiones condicionales de tipo ***cond***. El formato se expresa de la siguiente forma:

***;; Formato***

***;( define ( figura c )***

***; (cond***

***; [ ( ... c ) ... ]***

***; [ ( ... c ) ... ]***

***; [ ( ... c ) ... ]***

***; [ ( ... c ) ... ]))***

Paso 4:

Con el formato de arriba y sustituyendo los ( . . . ) por las ***expresiones condicionales*** que se necesitan, el cuerpo de la ***función*** queda:

**;Cuerpo de la función.**

**(define (figura c)**

**(cond**

**[[string=? "cuadrado" c](square 60 "solid" "mediumslateblue")]]**

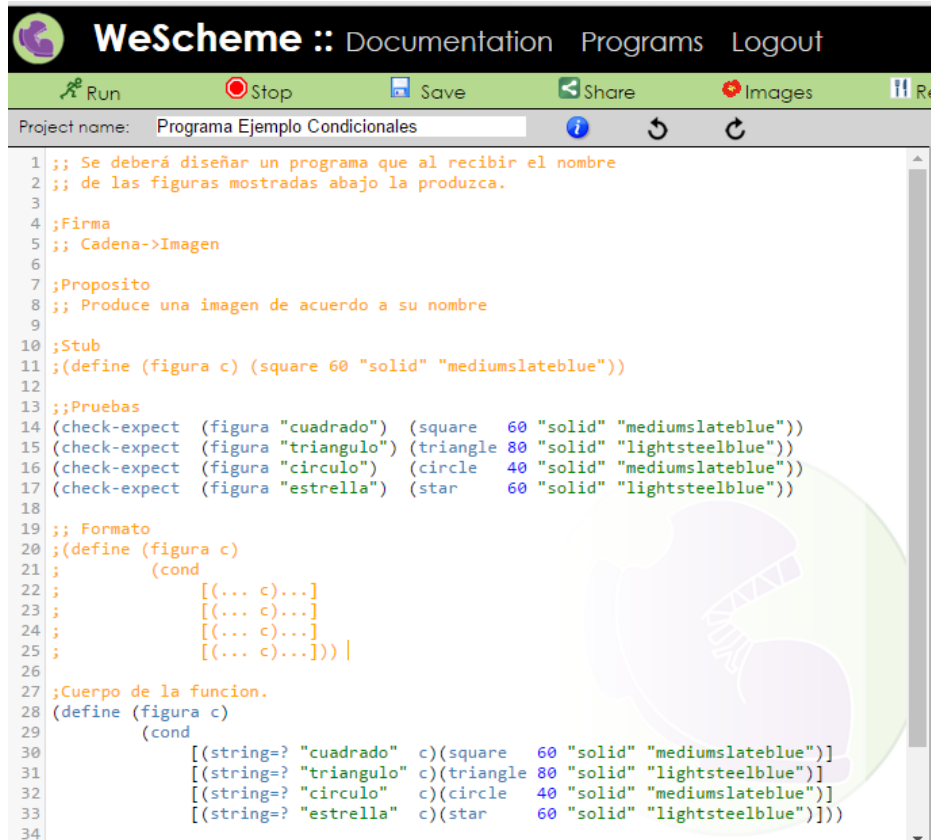
**[[string=? "triángulo" c](triangle 80 "solid" "lightsteelblue")]]**

**[[string=? "círculo" c](circle 40 "solid" "mediumslateblue")]]**

**[[string=? "estrella" c](star 60 "solid" "lightsteelblue")]]))**

Paso 5:

Se presiona el botón **Run**, si no se genera algún error, se garantiza que la **función** cumple su objetivo correctamente, el código computarizado en el ambiente **WeScheme** se muestra de la siguiente forma:



```
1 ;; Se deberá diseñar un programa que al recibir el nombre
2 ;; de las figuras mostradas abajo la produzca.
3
4 ;Firma
5 ;; Cadena->Imagen
6
7 ;Proposito
8 ;; Produce una imagen de acuerdo a su nombre
9
10 ;Stub
11 ;(define (figura c) (square 60 "solid" "mediumslateblue"))
12
13 ;;Pruebas
14 (check-expect (figura "cuadrado") (square 60 "solid" "mediumslateblue"))
15 (check-expect (figura "triangulo") (triangle 80 "solid" "lightsteelblue"))
16 (check-expect (figura "circulo") (circle 40 "solid" "mediumslateblue"))
17 (check-expect (figura "estrella") (star 60 "solid" "lightsteelblue"))
18
19 ;; Formato
20 ;(define (figura c)
21 ;  (cond
22 ;    [(... c)...]
23 ;    [(... c)...]
24 ;    [(... c)...]
25 ;    [(... c)...])) |
26
27 ;Cuerpo de la funcion.
28 (define (figura c)
29   (cond
30     [(string=? "cuadrado" c)(square 60 "solid" "mediumslateblue")]
31     [(string=? "triangulo" c)(triangle 80 "solid" "lightsteelblue")]
32     [(string=? "circulo" c)(circle 40 "solid" "mediumslateblue")]
33     [(string=? "estrella" c)(star 60 "solid" "lightsteelblue"))])
34
```

El resultado se produce escribiendo el nombre de la **función** e ingresando como valor de entrada el nombre de la figura que se desea mostrar:

( <Nombre de la función> <valor de entrada n> ...)

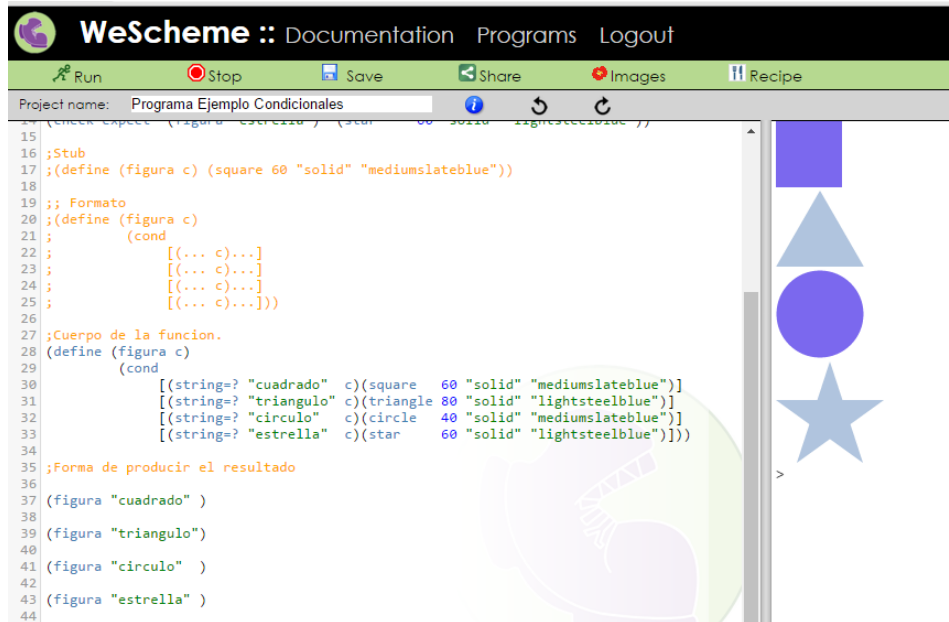
Para las 4 figuras se tiene lo siguiente:

(figura "cuadrado" )

(figura "triángulo" )

(figura "círculo" )

(figura "estrella" )



The screenshot shows the WeScheme web interface. At the top, there is a navigation bar with 'WeScheme :: Documentation Programs Logout'. Below it is a toolbar with 'Run', 'Stop', 'Save', 'Share', 'Images', and 'Recipe' buttons. The 'Project name' field contains 'Programa Ejemplo Condicionales'. The main area displays Scheme code in a text editor, with line numbers 15 to 44. The code defines a function 'figura' that takes a string 'c' and uses a 'cond' statement to call different drawing functions based on the input: 'square' for 'cuadrado', 'triangle' for 'triángulo', 'circle' for 'círculo', and 'star' for 'estrella'. The output area on the right shows four vertically stacked shapes: a blue square, a light blue triangle, a blue circle, and a light blue star.

```
15 (check-expect (figura "estrella") (star 60 "solid" "lightsteelblue"))
16 ;Stub
17 ;(define (figura c) (square 60 "solid" "mediumslateblue"))
18
19 ;; Formato
20 ;(define (figura c)
21 ;  (cond
22 ;    [(string=? "cuadrado" c)] [(... c)...]
23 ;    [(string=? "triángulo" c)] [(... c)...]
24 ;    [(string=? "círculo" c)] [(... c)...]
25 ;    [(string=? "estrella" c)] [(... c)...]))
26
27 ;Cuerpo de la funcion.
28 (define (figura c)
29   (cond
30     [(string=? "cuadrado" c) (square 60 "solid" "mediumslateblue")]
31     [(string=? "triángulo" c) (triangle 80 "solid" "lightsteelblue")]
32     [(string=? "círculo" c) (circle 40 "solid" "mediumslateblue")]
33     [(string=? "estrella" c) (star 60 "solid" "lightsteelblue")]))
34
35 ;Forma de producir el resultado
36
37 (figura "cuadrado" )
38 (figura "triángulo")
39 (figura "círculo" )
40 (figura "estrella" )
41
42
43
44
```

*"Lo que tenemos que aprender lo aprendemos haciendo"*

*-Aristóteles-*

## **6. CÓMO DISEÑAR MUNDOS**

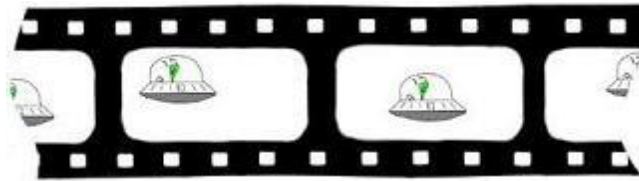
Un **mundo** es una estructura de datos que representa todo lo que el programa puede cambiar bajo un programa control. Ambos programas los seleccionamos nosotros, la manera de indicarlos es escribiendo funciones.

Estas funciones consumen un valor para el **mundo**, representan el mundo viejo y producen un nuevo valor para poder representar el mundo nuevo.

Los llamados **programa mundo** son programas interactivos que tienen que ver con eventos (acciones que el usuario realiza sobre el programa) como impulsos de reloj, clics del mouse, pulsaciones de teclas o cualquier dispositivo externo de la computadora con el fin de interactuar con los humanos donde **mundo** cambia con respecto al tiempo y se muestra a través de una ventana de animación.



Muchos *programas mundo* son exactamente como las películas, una película consiste en una series de escenas y en cada escena los caracteres están en diferentes posiciones, formas, tamaños etc<sup>13</sup>.



La receta de diseño para diseñar programas se basa en 4 pasos y utiliza un formato que es una herramienta que facilita el uso:

### **6.1 Funcion *big-bang***

La función *big-bang* es la encargada de iniciar el *mundo*, toma uno o más parámetros, el primer parámetro es la primera imagen de animación que mostrará, a los parámetro siguientes se les denomina manipuladores de eventos, cuando la animación termina la función *big-bang* regresa la última imagen mostrada.

Un manipulador de evento es una función que controla las diferentes formas de interacción máquina-humano, y se define de la siguiente forma.

```
(big-bang <... .>
  (on-tick    <... .> )
  (to-draw   <... .> )
  (stop-when <... .> )
  (on-mouse <... .> )
  (on-key   <... .> ) )
```

---

<sup>13</sup> Matthias Felleisen, Robby Findler, Kathi Fisler, Matthew Flatt, Shriram Krishnamurthi. (2008). How to Design Worlds: Imaginative Programming in DrScheme. Estados Unidos: Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License.

## 6.2 Receta de diseño de mundos

### Pasos:

1. Análisis del dominio.
  1. Boceto de los escenarios del programa.
  2. Identificar la información que es constante.
  3. Identificar la información que cambia.
  4. identificar las opciones **big-bang**.
2. Construcción actual del programa.
  1. Constantes.
  2. Definición de datos.
  3. Funciones
    1. Función **main**
    2. Iniciar las funciones en la lista de deseos (!!!)
  4. Trabajar sobre las funciones en la lista de deseos marcadas con !!!.

Iniciar la funciones en de lista de deseos marcadas con !!!.

### Formato:

**;; Mi programa mundo (Especificar el funcionamiento)**

**;; =====**

**;; Constantes:**

**;; =====**

**;; Definición de datos**

**;; WS es ... (dar a WS un nombre)**

**;; =====**

**;; Funciones**

**;; WS -> WS**

**;; El mundo inicia con ...**

**;;**

```

(define (main ws)
  (big-bang ws          ; WS
    (on-tick tock)    ; WS -> WS
    (to-draw render) ; WS -> Imagen
    (stop-when ...)  ; WS -> Booleano
    (on-mouse ...)   ; WS Entero Entero EventoMouse -> WS
    (on-key ...)))  ; WS EventoTecla -> WS

;; WS -> WS
;; Produce el siguiente...
;; !!!
(define (tock ws) ...)
;; WS -> Image
;; Dibuja ...
;; !!!
(define (render ws) ...)

```

Ejemplo:

Se deberá diseñar un programa mundo en el cual una vaca atraviese la pantalla.

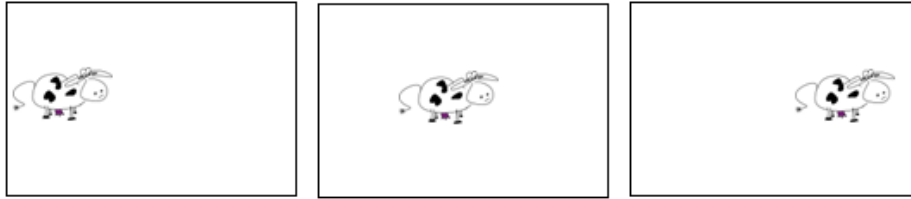
Solución:

### **Paso 1: Análisis del dominio**

Se identifican los escenarios del programa, las constantes, la información que cambia y las opciones **big-bang**.

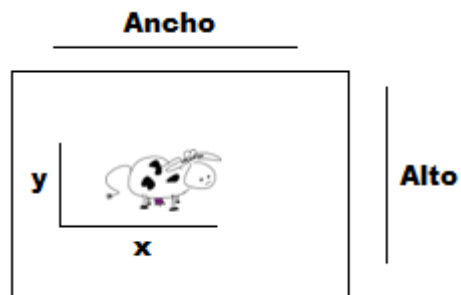
Boceto de los escenarios del programa:

La vaca tiene que cruzar por la pantalla, lo cual simula el movimiento que lleva.



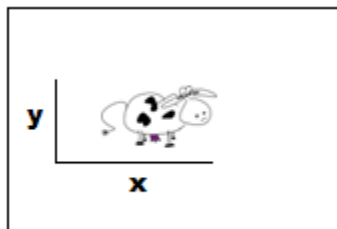
Identificar la información que es constante:

La información que no cambiará durante el control de programa es el **Alto** y el **Ancho** de la escena, así como la coordenada **y** de la imagen dentro de la escena.



Identificar la información que cambia:

La vaca camina hacia el lado derecho, por lo cual la información que cambia es la coordenada **x** de la vaca dentro de la escena.



Identificar las opciones **big-bang**:

Las opciones **big-bang** que se necesitan para reproducir la animación son:

**on-tick** cambia con el tiempo

**to-draw** dibuja algo

## **Paso 2: Construcción actual del programa.}**

En este paso se utiliza el formato para llenarlo con los datos del programa, se analiza la información del paso anterior para iniciar con el llenado.

En el paso anterior se detectaron las constantes, con esto se pueden llenar las partes de la descripción del programa y la sección de **constantes**:

***:: Mi programa mundo***

***:: Una vaca camina de izquierda a derecha en la pantalla.***

***:: =====***

***:: Constantes:***

***(define ANCHO 600)***

***(define ALTO 400)***

***(define CTR-Y (/ ALTO 2))***

***(define MTS (empty-scene ANCHO ALTO))***

***(define VACA (bitmap/url***

***"https://drive.google.com/uc?export=download&id=0B1dfCNKNHWJqVF9LM***

***EFtZDZpUGs"))***

The screenshot shows the WeScheme web interface. At the top, there is a navigation bar with 'WeScheme :: Documentation Programs Logout'. Below that is a toolbar with 'Run', 'Stop', 'Save', 'Share', 'Images', and 'R'. The project name is 'Programa mundo'. The code editor shows the following code:

```

1  ;; Mi programa mundo
2  ;; Una vaca camina de izquierda a derecha en la pantalla.
3
4  ;; =====
5  ;; Constantes:
6
7  (define ANCHO 600)
8  (define ALTO 400)
9  (define CTR-Y (/ ALTO 2))
10 (define MTS (empty-scene ANCHO ALTO))
11 (define VACA (bitmap/url "https://drive.google.com/uc?
export=download&id=0B1dfCNKNHWJqVF9LMEFtZDZpUGs"))
12

```

Definición de datos:

Con la información que cambia dentro del programa, se llena la parte de definición de datos:

Se toman en cuenta las posiciones en que puede estar la vaca dentro de la pantalla.

```

;; =====
;; Definición de datos
;; La vaca es un numero
;; Interpreta la posición en la coordenada X de la pantalla
(define C1 0)           ;Izquierda
(define C2 (/ ANCHO 2)) ;Mitad
(define C3 ANCHO)      ;Derecha

```

The screenshot shows the WeScheme web interface with the updated code in the editor. The code is as follows:

```

12
13 ;; =====
14 ;; Definición de datos
15 ;; La vaca es un numero
16 ;; Interpreta la posición en la coordenada X de la pantalla
17
18 (define C1 0)           ;Izquierda
19 (define C2 (/ ANCHO 2)) ;Mitad
20 (define C3 ANCHO)      ;Derecha
21

```

### Paso 3: Funciones.

#### *Función\_main*

*WS* es el cambio de *mundo* dentro del programa, la **VACA** es el mundo en este programa, en la sección de comentarios sustituimos **WS** por **VACA**, en las funciones **big-bang**, **tock** y **render** se coloca cualquier letra en este caso será **c** esto indicara que es el valor de entrada de las funciones, para la función **big-bang** se dejan las opciones que se identificaron en el paso 1.

```
:: =====  
:: Funciones  
:: VACA -> VACA  
:: El mundo inicia con ...  
::  
(define (main c)  
  (big-bang c ; VACA  
    (on-tick tock) ; VACA -> VACA  
    (to-draw render) ; VACA -> Imagen  
  
:: WS -> WS  
:: Produce el siguiente...  
:: !!!  
(define (tock c) ...)  
  
:: WS -> Image  
:: Dibuja ...  
:: !!!  
(define (render c) ...)
```

```

21
22 ;; =====
23 ;; Funciones
24 ;; VACA -> VACA
25 ;; El mundo inicia con ...]
26 ;;
27 (define (main c)
28   (big-bang c
29     (on-tick tock) ; VACA
30     (to-draw render) ; VACA -> VACA
31                       ; VACA -> Imagen
32 ;; WS -> WS
33 ;; Produce el siguiente...
34 ;; !!!
35 (define (tock c) ...)
36
37 ;; WS -> Image
38 ;; Dibuja ...
39 ;; !!!
40 (define (render c) ...)
41

```



Iniciar las funciones en la lista de deseos (!!!):

Para los dos manejadores de eventos de la función **big-bang** se inicia poniendo el propósito y generando el **stub** como la receta de cómo diseñar funciones.

***;; WS -> WS***

***;; Produce la siguiente vaca avanzando 1pixel hacia la derecha***

***;; !!!***

***(define (tock c) 0)***

***;; WS -> Image***

***;; Dibuja la imagen de la vaca dentro de la escena MTS***

***;; !!!***

***(define (render c) MTS)***



```

22 ;; =====
23 ;; Funciones
24 ;; VACA -> VACA
25 ;; El mundo inicia con ...
26 ;;
27 (define (main c)
28   (big-bang c
29     (on-tick tock) ; VACA
30     (to-draw render)) ; VACA -> Imagen
31
32 ;; WS -> WS
33 ;; Produce la siguiente vaca avanzando 1pixel hacia la derecha
34 ;; !!!
35 (define (tock c) 0)
36
37 ;; WS -> Image
38 ;; Dibuja la imagen de la vaca dentro de la escena MTS
39 ;; !!!
40 (define (render c) MTS )
41

```



Se puede modificar el nombre de las funciones de la lista de deseos. Esto es opcional, pero si se modifica se tiene que modificar también en la función **big-bang** .

```

;; =====
;; Funciones
;; VACA -> VACA
;; El mundo inicia con ...
;;
(define (main c)
  (big-bang c ; VACA
    (on-tick avanza) ; VACA -> VACA
    (to-draw dibuja)) ; VACA -> Imagen

;; WS -> WS
;; Produce la siguiente vaca avanzando 1pixel hacia la derecha
;; !!!
(define (avanza c) 0)

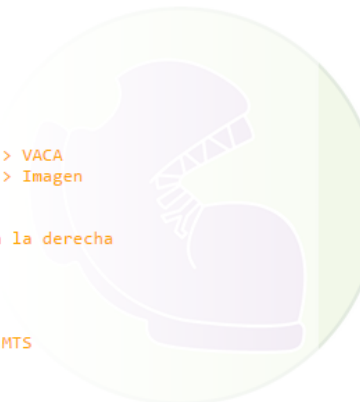
;; WS -> Image
;; Dibuja la imagen de la vaca dentro de la escena MTS
;; !!!
(define (dibuja c) MTS )

```

```

22 ;; =====
23 ;; Funciones
24 ;; VACA -> VACA
25 ;; El mundo inicia con ...|
26 ;;
27 (define (main c)
28   (big-bang c
29     (on-tick  avanza)    ; VACA
30     (to-draw  dibuja))  ; VACA -> Imagen
31
32 ;; WS -> WS
33 ;; Produce la siguiente vaca avanzando 1pixel hacia la derecha
34 ;; !!!
35 (define (avanza c) 0)
36
37 ;; WS -> Image
38 ;; Dibuja la imagen de la vaca dentro de la escena MTS
39 ;; !!!
40 (define (dibuja c) MTS )
41

```



#### Paso 4: Trabajar sobre las funciones en la lista de deseos marcadas con !!!.

Esta parte se desarrolla con la receta de diseño que si vio en el capítulo 5 porque son funciones que se tienen que diseñar.

El siguiente paso es generar las pruebas para cada función:

La función *avanza* su objetivo de incrementar un pixel para que la vaca se mueva de izquierda a derecha:

```
(check-expect (avanza 0) 1)
```

```
(check-expect (avanza 3) 4)
```

La función *avanza* debe de producir una escena con la imagen de la vaca

```
(check-expect (dibuja 5) (place-image VACA 5 CTR-Y MTS))
```

Se comenta el *stub* y se crea el formato para ambas funciones:

```
;(define (avanza c) (... c))
```

```
;(define (dibuja c) (... c))
```

En base al formato y las pruebas, se genera el cuerpo de la función:

```
(define (avanza c)
  (+ c 1))
```

```
(define (dibuja c)
  (place-image VACA c CTR-Y MTS))
```

Para ver el programa en funcionamiento se agrega la siguiente línea:

```
(main 0)
```

Este programa en **WeScheme** se ve de la siguiente forma



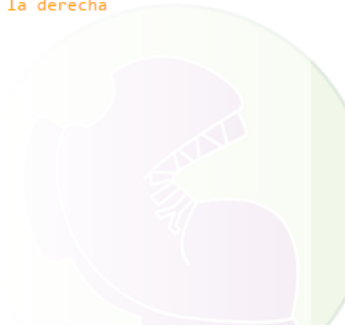
The screenshot shows the WeScheme IDE interface. At the top, there's a navigation bar with 'WeScheme :: Documentation Programs Logout'. Below that is a toolbar with 'Run', 'Stop', 'Save', 'Share', 'Images', and 'Re'. The main area displays a Racket program with the following code:

```
1 ;; Mi programa mundo
2 ;; Una vaca camina de izquierda a derecha en la pantalla.
3
4 ;; =====
5 ;; Constantes:
6
7 (define ANCHO 600)
8 (define ALTO 400)
9 (define CTR-Y (/ ALTO 2))
10 (define MTS (empty-scene ANCHO ALTO))
11 (define VACA (bitmap/url "https://drive.google.com/uc?
export=download&id=0B1dfCNKNHWJqVF9LMEftZDZpUGs"))
12
13 ;; =====
14 ;; Definición de datos
15 ;; La vaca es un numero
16 ;; Interpreta la posición en la coordenada X de la pantalla
17
18 (define C1 0) ;Izquierda
19 (define C2 (/ ANCHO 2)) ;Mitad
20 (define C3 ANCHO) ;Derecha
21
22 ;; =====
23 ;; Funciones
24 ;; VACA -> VACA
25 ;; El mundo inicia con ...
26 ;;
27 (define (main c)
28   (big-bang c
29     (on-tick avanza) ; VACA
30     (to-draw dibuja)) ; VACA -> VACA
31                       ; VACA -> Imagen
```

```

31
32 ;; WS -> WS
33 ;; Produce la siguiente vaca avanzando 1pixel hacia la derecha
34
35 ;Pruebas
36 (check-expect (avanza 0) 1)
37 (check-expect (avanza 3) 4)
38
39 ;stub
40 ;(define (avanza c) 0)
41
42 ;Formato
43 ;(define (avanza c) (...c))
44
45 ;Cuerpo de la funcion
46 (define (avanza c)
47   (+ c 1))
48

```



```

50 ;; WS -> Image
51 ;; Dibuja la imagen de la vaca dentro de la escena MTS
52
53 ;Pruebas
54 (check-expect (dibuja 5) (place-image VACA 5 CTR-Y MTS))
55
56 ;stub
57 ;(define (dibuja c) MTS )
58
59 ;Formato
60 ;(define (dibuja c) (...c))
61
62
63 ;Cuerpo de la funcion
64 (define (dibuja c)
65   (place-image VACA c CTR-Y MTS))
66
67
68 ;Inicia el programa
69 (main 0)
70

```



Al presionar el botón **RUN**, se muestra a la vaca moviéndose de izquierda a derecha.





## 7. CONCLUSIONES

El aprendizaje de la programación es complejo, el utilizar una metodología que permita diseñar programas con herramientas y lenguajes sencillos de usar permite que la enseñanza y el aprendizaje sean dinámicos, fáciles y divertidos.

Las **recetas de diseño** son una metodología que permite diseñar programas que son comunicables entre personas. Estas recetas se complementan con la creación de videojuegos, los cuales son motivantes para los aprendices con el fin de generar interés y facilitar el aprendizaje.

El enfoque que se dio en este reporte de servicio social es principalmente para niños de secundaria pero también para estudiantes y personas de cualquier nivel con el objetivo de generar una inducción de la investigación que se ha generado y puesto en práctica a nivel mundial de la metodología sobre el diseño de programas.

## 7.1 Conclusiones personales

En mi formación profesional esta metodología me ayudo a entender lo que es realmente diseñar software con los siguientes puntos:

- Diseñar software no solo es conocer la sintaxis del lenguaje, esto me ha permitido poder cambiar de lenguaje y aprender nuevos rápidamente.
- Pensar antes de codificar, esta metodología me ha inculcado al habilidad de antes de iniciar con líneas de código primero entender el problema y diseñar una solución.
- Diseñar un código que sea comunicable con cualquier persona, agregando comentarios, nombre de variables y constantes acorde al problema que se está resolviendo, indentación del código, por si se necesita cambiar algo en un futuro, se pueda hacer fácilmente.
- Comprender más a fondo los conceptos de la programación.



## 8. BIBLIOGRAFÍA

- *Bootstrap, (Bootstrap, 2015), recuperado de*  
*<http://www.bootstrapworld.org/materials/fall2015/tutorial/>*
- *Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, Kathi Fisler. (2011). WeScheme: The Browser is Your Programming Environment. 2011, de Conference on Innovation and Technology in Computer Science Education Sitio web:*  
*<http://cs.brown.edu/~sk/Publications/Papers/Published/yskf-wescheme/>*
- *Gregor Kiczales Erika Thompson. (2015). Systematic Program Design Part 1. 2015, de EDX Sitio web: <https://courses.edx.org/courses/course-v1:UBCx+SPD1x+1T2016/info>*
- *Matthias Felleisen, Robby Findler, Kathi Fisler, Matthew Flatt, Shriram Krishnamurthi. (2008). How to Design Worlds: Imaginative*

*Programming in DrScheme. Estados Unidos: Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License.*

- *Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi . (Septiembre 26, 2003). How to Design Programs. 2003, de The MIT Press Sitio web: <http://www.htdp.org/>*