FACULTAD DE INGENIERÍA

INGENIERÍA EN COMPUTACIÓN

DE LA ARITMÉTICA AL ÁLGEBRA: FUNCIONES

TESINA

QUE PARA OBTENER EL TÍTULO DE

INGENIERO EN COMPUTACIÓN

PRESENTA:

GUILLERMO RIOS ORTEGA

DIRECTOR:

MTRO. JUAN JOSÉ CARREÓN GRANADOS

CIUDAD UNIVERSITARIA, MÉXICO CDMX 2016

De la Aritmética al Álgebra: funciones

Agradecimientos

A mi madre Martha Lilia Ortega Ortega por las horas de dedicación para que yo sea una persona con grandes valores morales y dedicación, que me permiten estar hoy aquí gracias.

A mi padre Guillermo Ríos Galván por soportar todas mis inconsistencias y aun así estar dispuesto a dar todo para apoyarme.

A mis hermanos Juan Carlos y Alejandro Ríos Ortega , ya que sin su apoyo y ánimos no hubiera sido posible el desarrollo de este proyecto, por que nunca dejaron de creer en mi.

A mi asesor el M. I. Juan José Carreón Granados, por dedicarme su tiempo y sus conocimientos para iniciar, dar forma y término a este proyecto.

A la M. I. Norma Elva Chávez sin su entusiasmo no se hubiera iniciado este proyecto.

A mis hermanos y compañeros de andanzas, que aplaudían mis victorias y me animaban a seguir en las derrotas.

De la aritmética al álgebra: funciones[1]

Resumen

Análisis y evaluación de recursos de programación como ayudas al aprendizaje de asignaturas de Matemáticas del primer semestre y de sus antecedentes de las carreras de ingeniería en la Facultad de Ingeniería, FI, UNAM, principalmente los que pueden emplearse para pasar de los conceptos aritméticos a los algebraicos, en particular los relacionados con el concepto de función.

[1] Los cursos a que se consultaron y analizaron para la elaboración de este proyecto se encuentran en ingles, las imágenes que se emplean se tomaron del documento original por eso se encuentran en éste idioma.

1 Introducción

Este proyecto se desarrolló con la finalidad de que los profesores de la Facultad de Ingeniería consideren a las recetas de diseño[2] para el desarrollo de programas, como una herramienta de apoyo para el aprendizaje y un buen desempeño académico de los estudiantes de primer ingreso, que presentan deficiencias en la materia de álgebra[3], así como en los antecedente del paso de la aritmética al álgebra en específico de variables a funciones[4].

---

[2] Krishnamurthi, M. F. (s.f.). Understanding the Program's Purpose:. Recuperado el 11 de 12 de 2015, de HTDP: http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-5.html#node_sec_2.5

[3] Se localiza en la sección B de anexos un documento que muestra las calificaciones del bloque 106A de la carrera de ingeniería en computación, de la Facultad de ingeniería.

[4] Krishnamurthi, M. F. (s.f.). HTDP second edition. Recuperado el 12 de 12 de 2015, de Inputs and Output: http://www.ccs.neu.edu/home/matthias/HtDP2e/part_prologue.html#%28part._some-i%2Fo%29

Para lo cual se revisaron y analizaron algunos cursos en línea[5]que abordan el tema de manera clara y dinámica, empleando ejemplos, sencillos y concisos para poder explicar el concepto paso a paso logrando una mejor comprensión del tema y un deseable dominio de éste, el empleo de este método de enseñanza también podrá ayudar a generar una eficiente técnica de aprendizaje y de análisis para conseguir un alto desempeño de los alumnos en su primer semestre, que les sirva de plataforma para tener un mejor desarrollo académico y profesional.

---

[5] Kiczales, G. (s.f.). Systematic Program Design. Recuperado el 21 de 12 de 2015, de edx: https://courses.edx.org/courses/course-v1:UBCx+SPD1x+1T2016/77860a93562d40bda45e452ea064998b/#FuncComp

2 Lenguaje estudiante principiante (Beginning Student Language[6], BSL)

De niños, los padres probablemente nos enseñaron a contar y más tarde para realizar cálculos simples a emplear los dedos: "1 + 1 es 2"; "1 + 2 es 3"; y así. Luego preguntaban "¿cuánto es 3 + 2"? y contamos los dedos de la mano. Nos programaban, y calculábamos. Y de alguna manera, eso es realmente todo lo que hace la programación y la computación.

[6] Krishnamurthi, M. F. (s.f.). Intermezzo: BSL. Recuperado el 27 de 12 de 2015, de HTDP second edition : http://www.ccs.neu.edu/home/matthias/HtDP2e/i1-2.html

Es hora de cambiar los roles. Ahora nosotros programamos, y la computadora es el niño. Por supuesto, que comenzamos con el más simple de todos los cálculos.

(+ 1 1)[7]

[7] Comience DrRacket y seleccione "Idioma" en el menú "Idioma". Esto nos lleva a un cuadro de diálogo "Lenguajes de enseñanza" de "Cómo diseñar programas" (y posiblemente otros libros). Seleccione la opción "Lenguaje estudiante principiante" (BSL) y "OK" para configurar DrRacket para este capítulo.

2.1 BSL[8]

Es el lenguaje principiante de DrRacket. Introduce las "palabras básicas" del lenguaje, sugiere cómo componer las "palabras" en "frases", y hace un evocación a nuestros conocimientos de álgebra para una comprensión intuitiva de estas "frases".

2.2El vocabulario de DrRacket[9]

El vocabulario DrRacket básico consiste en cinco categorías de palabras. Las cinco líneas en la Figura 1 muestran cómo los científicos de la computación discuten el vocabulario de un lenguaje[10]. Todas las líneas emplean la misma notación. Enumeran algunos ejemplos simples separándolos por una barra *("|").* Los puntos indican que hay más cosas de la misma clase en alguna categoría.

---

[8]Krishnamurthi, M. F. (s.f.). *Intermezzo: BSL*. Recuperado el 27 de 12 de 2015, de HTDP second edition : http://www.ccs.neu.edu/home/matthias/HtDP2e/i1-2.html

[9]Benson, B. (s.f.). *Racket*. Recuperado el 19 de 01 de 2016, de Sección 8 Intermezzo 1: Sintaxis y semántica: http://docs.racket-lang.org/htdp-langs/beginner.html

[10]wikipedia Notación de Backus-Naur. Recuperado el 22 de 01 de 2016, de Notación de Backus-Naur : https://es.wikipedia.org/wiki/Notaci%C3%B3n_de_Backus-Naur

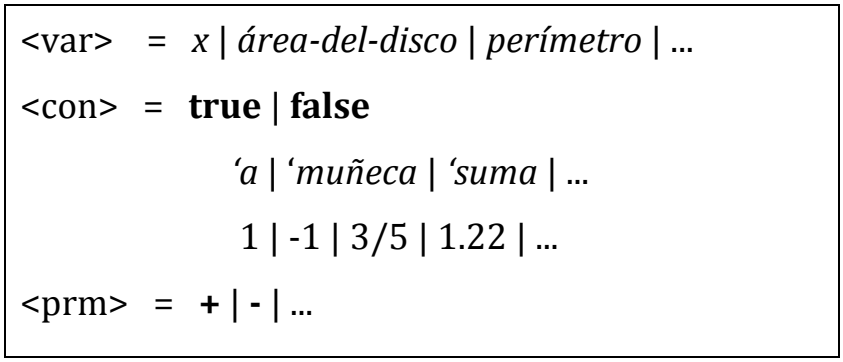


Figura 1. DrRacket Estudiante Principiante: El vocabulario esencial

La primera categoría es la de variables, las cuales son nombres de funciones y valores. La segunda introduce constantes: booleanas, simbólicas y numéricas. DrRacket tiene un poderoso sistema numérico, al cual se introduce mejor gradualmente mediante ejemplos. La categoría final es la de operaciones primitivas, las cuales son aquellas funciones básicas que DrRacket proporciona inicialmente. Si bien es posible especificar esta colección en su totalidad, es mejor presentarla en partes, conforme se requiera.

Para la clasificación de oraciones en DrRacket, se requieren también tres palabras reservadas: **define**, **cond**, y **else**[11]. Estas palabras reservadas no tienen significado. Su papel es parecido al de los signos de puntuación, especialmente el de comas y punto y coma, ayudan a los

[11] Benson, B. (s.f.). Racket. Recuperado el 19 de 01 de 2016, de Sección 8 Intermezzo 1: Sintaxis y semántica: http://docs.racket-lang.org/htdp-langs/beginner.html

programadores a distinguir una oración de otra. Ninguna palabra reservada puede emplearse como variable.

2.3 La gramática en DrRacket

En contraste con muchos otros lenguajes de programación, DrRacket tiene una gramática simple. La cual se muestra en su totalidad en la Figura 2. La gramática define dos categorías de oraciones: definiciones de DrRacket, <def>, y expresiones, <exp>. Si bien la gramática no dicta el empleo de espacios en blanco entre ítems de oraciones, se sigue la convención de colocar cuando menos un espacio en blanco detrás de cada ítem a menos de que el ítem sea seguido por un paréntesis derecho ")". DrRacket es flexible en cuanto a espacios en blanco, y se puede reemplazar un solo espacio en blanco por muchos espacios, cambios de línea y cambios de página.

<def> = (**define** (<var><var> ... <var>) <exp>)

<exp> = <var>
| <con>
| (<prm><exp> ... <exp>)
| (<var><exp> ... <exp>)
| (**cond** (<exp><exp>) ... (<exp><exp>))
| (**cond** (<exp><exp> ... (**else** <exp>))

Figura 2. DrRacket Estudiante Principiante: la gramática esencial

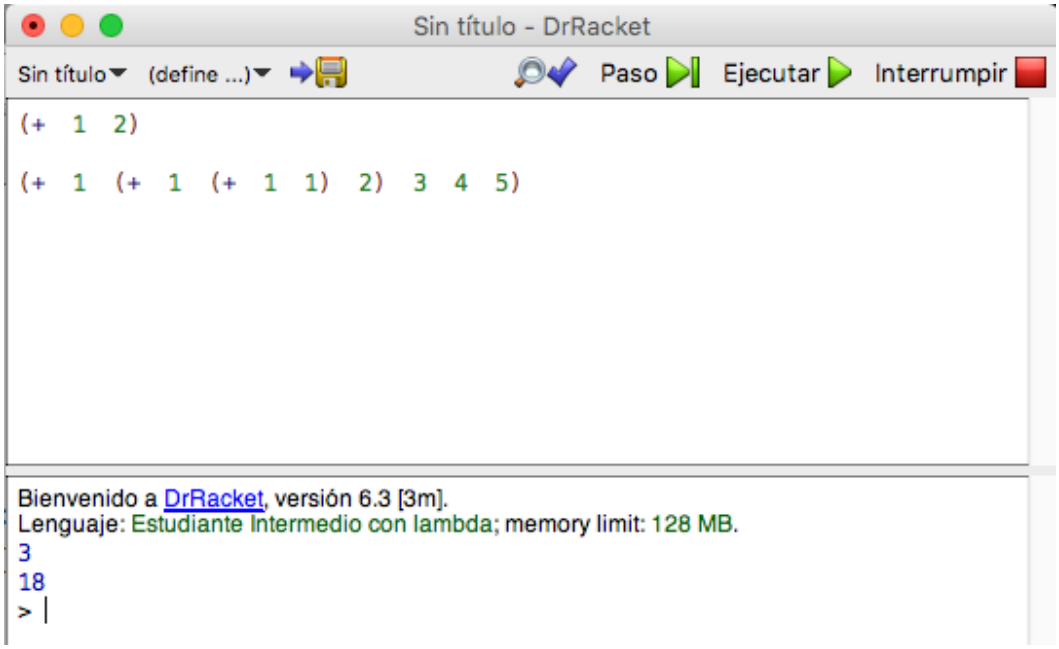
Las dos definiciones de gramática describen cómo formar oraciones atómicas y compuestas, construidas a partir de otras oraciones. Por ejemplo, una definición de función se forma mediante *"("*, seguido por la palabra clave define, seguida por otro *"("*, seguido por una secuencia no vacía de variables, seguidas de *")"*, seguida por una expresión, y cerrada por un paréntesis derecho *")"* que coincide con el inicial. La palabra reservada **define** distingue definiciones de expresiones.

La categoría de expresiones consiste de seis opciones: variables, constantes, aplicaciones de primitivas, aplicaciones (de funciones) y dos variedades de condicionales. Las últimas cuatro se componen a su vez de otras expresiones. La palabra reservada **cond** distingue expresiones condicionales de aplicaciones de primitivas y de funciones.

3 Aritméticas[12]

La evaluación de expresiones es sencilla. En primer lugar se evalúan todos los argumentos de una operación primitiva. En segundo lugar, se analizan los datos de entrada para ser ejecutados y se produce un resultado. Por lo tanto,

[12]Krishnamurthi, M. F. (s.f.). *Fixed-Size Data*. Recuperado el 23 de 12 de 2015, de HTDP second edition: http://www.ccs.neu.edu/home/matthias/HtDP2e/part_one.html



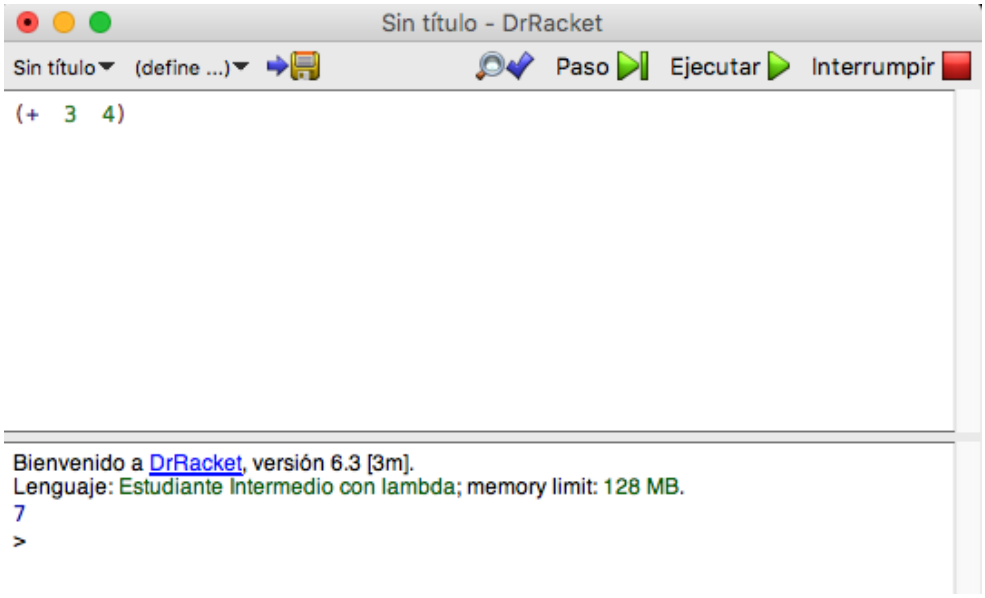
El segundo ejemplo explota dos puntos. En primer lugar, las operaciones primitivas pueden consumir más de dos argumentos. En segundo lugar, los argumentos pueden también ser expresiones.

Estos cálculos se ven familiares, ya que son del mismo tipo de cálculos que se realizan en las clases de matemáticas. Es posible que se conozcan los pasos de una manera diferente; sin embargo DrRacket, los ejecuta de manera eficiente. Garantiza que se entienda lo que hacen las operaciones primitivas con datos primitivos, por lo que se puede predecir los resultados calculados.

A continuación se presenta cuatro formas de datos atómicos de BSL: números, cadenas, imágenes y valores booleanos.

3.1 Aritmética de números

Operaciones con números, sumar significa tomar dos números para producir un tercero. Una operación aritmética, como **+** se utiliza de esta manera:



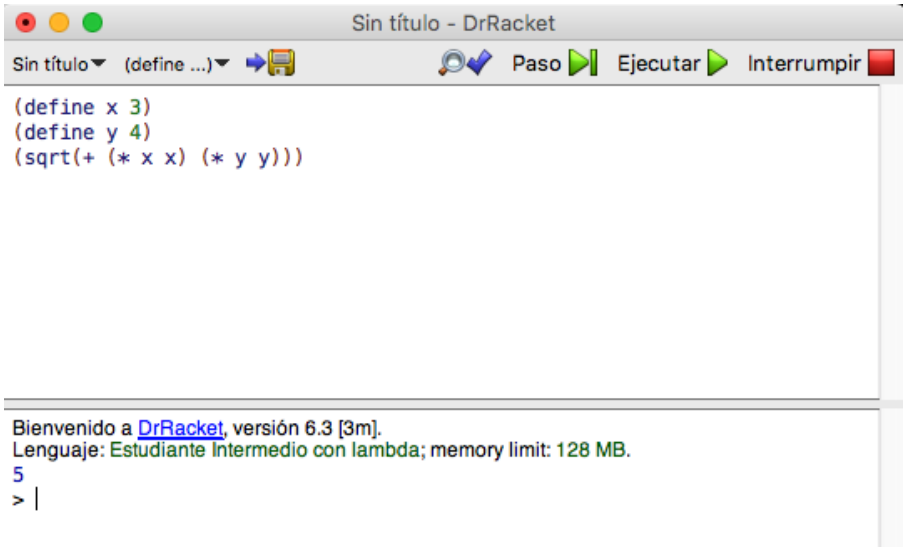
Si se necesita una operación con números es probable que BSL sepa sobre ella y cómo operarla.

Cuando se trata de números, BSL puede usar números naturales, números enteros, números racionales, números reales, y los números complejos.

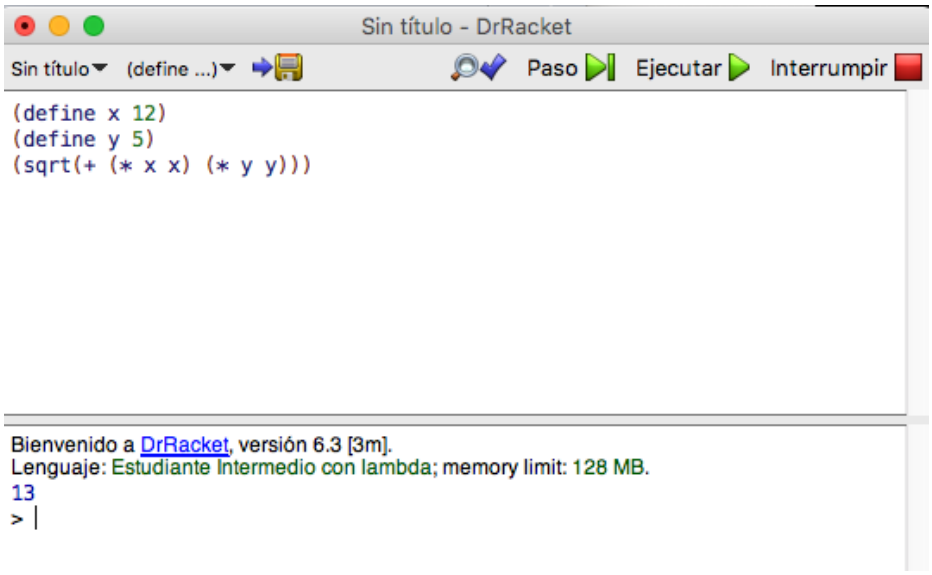
Una distinción verdaderamente importante se refiere a la precisión de los números. Por ahora, es importante entender que BSL distingue cifras exactas y números inexactos. Cuando se calcula con los números exactos, BSL conserva esta precisión siempre que sea posible. Por ejemplo, (/4 6) produce la fracción exacta 2/3 que DrRacket puede mostrar como una fracción propia, una fracción impropia, o como un decimal. Con esta base se pueden plantear ejercicios como los siguientes.

Ejercicio 1. El objetivo directo de este ejercicio es crear una expresión que calcula la distancia de un punto cartesiano específico (x, y) desde el origen (0, 0). El objetivo indirecto es introducir algunos hábitos básicos de cálculo, especialmente el uso de la zona de interacciones para desarrollar expresiones. Cómo asignar valores a identificadores.

El resultado esperado para estos valores es 5, pero su expresión debe producir el resultado correcto, incluso después de cambiar estas definiciones.



Para confirmar que la expresión funciona correctamente, se pueden cambiar las dos definiciones de manera que *x* sea 12 y *y* sea 5. Si hace clic en ejecutar ahora, el resultado debe ser 13.



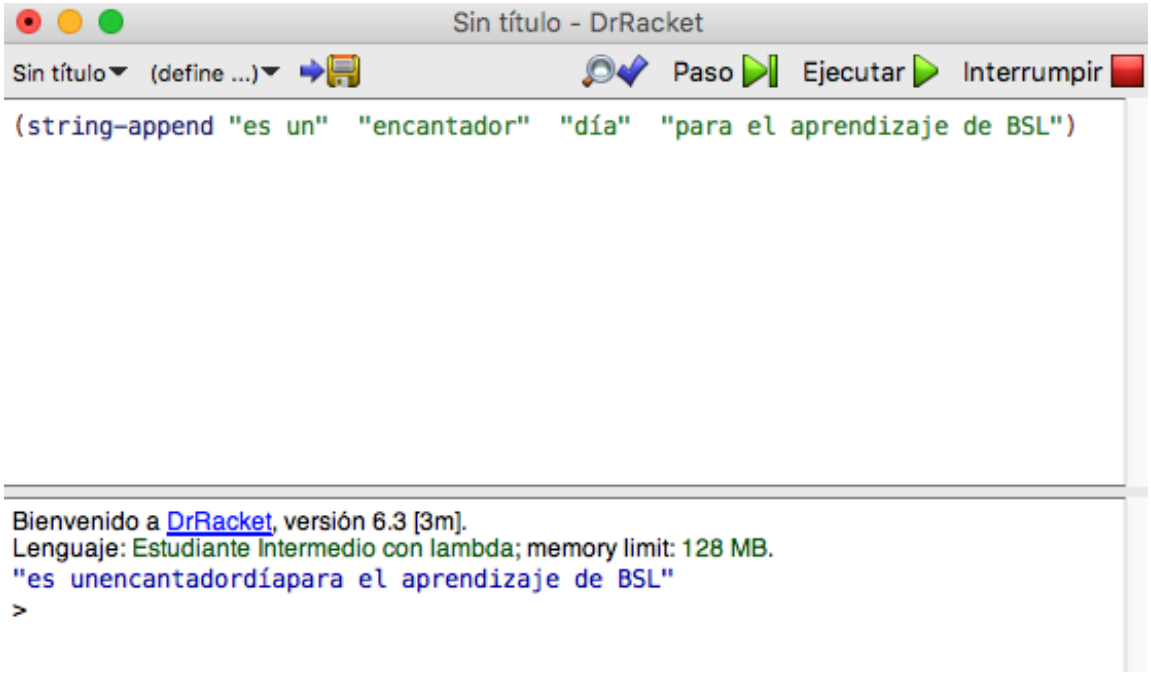
3.2 Aritmética de cadenas

Un prejuicio generalizado acerca de la computación se refiere a sus entrañas. Muchos creen que es todo acerca de los bits y bytes –lo que sean esas cosas- y posiblemente números, porque todo el mundo sabe que las computadoras pueden calcular. Los lenguajes de programación calculan u operan con la información, y la información se presenta de muchas formas. Por ejemplo, un programa puede tratar con colores, nombres, cartas, o las conversaciones entre las personas. A pesar de que podríamos codificar este tipo de información como números, no sería lo más natural. Se tendrían que recordar grandes tablas de códigos, como el 0 significa "rojo" y 1 significa "hola", etc.

La mayoría de los lenguajes de programación proporcionan por lo menos un tipo de datos que se ocupa de la información simbólica los *string* o cadenas.

Generalmente hablando, un *string* es una secuencia de caracteres que se pueden introducir con el teclado, entre comillas dobles: "hola", "mundo", "azul", "rojo", etc. Los dos primeros son palabras que pueden aparecer en una conversación, los otros son nombres de colores que podemos utilizar.

BSL incluye inicialmente una operación que consume y produce *strings*: **string-append**, que, concatena dos o más *strings* en una sola.



La aritmética con los *strings* y con los números no es muy diferente entre sí, basta con analizar algunos ejemplos y comprobar que se aplican las mismas formas de ejecución.

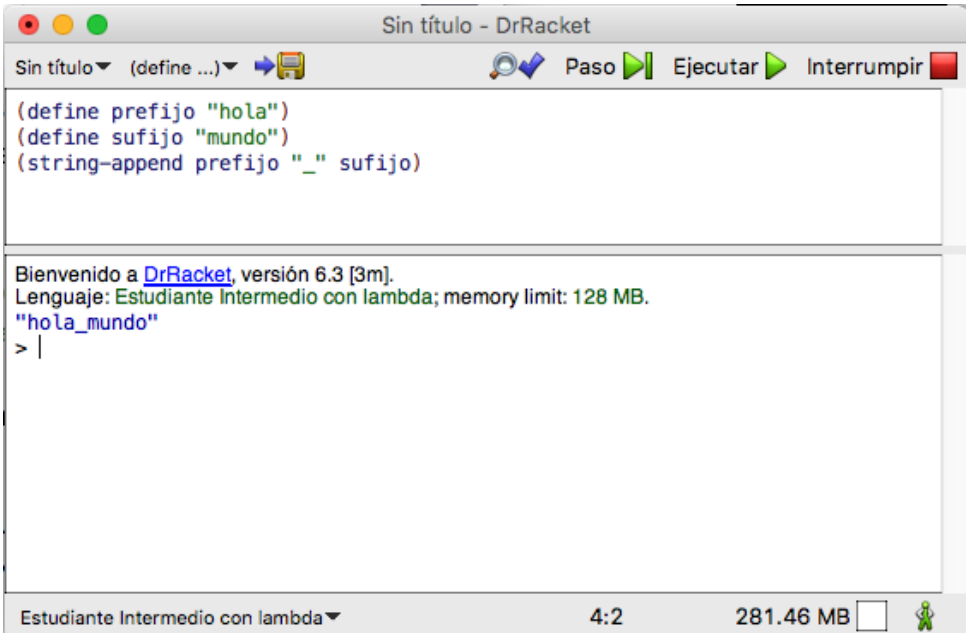
```
(+ 1 1) == 2          (string-append "a" "b") == "ab"
(+ 1 2) == 3          (string-append "ab" "c") == "abc"
(+ 2 2) == 4          (string-append "a" " " "c") == "a c"
```

Ejercicio 2. Agregue las dos líneas siguientes a la zona de las definiciones:

(define prefijo "hola")

(define sufijo "mundo")

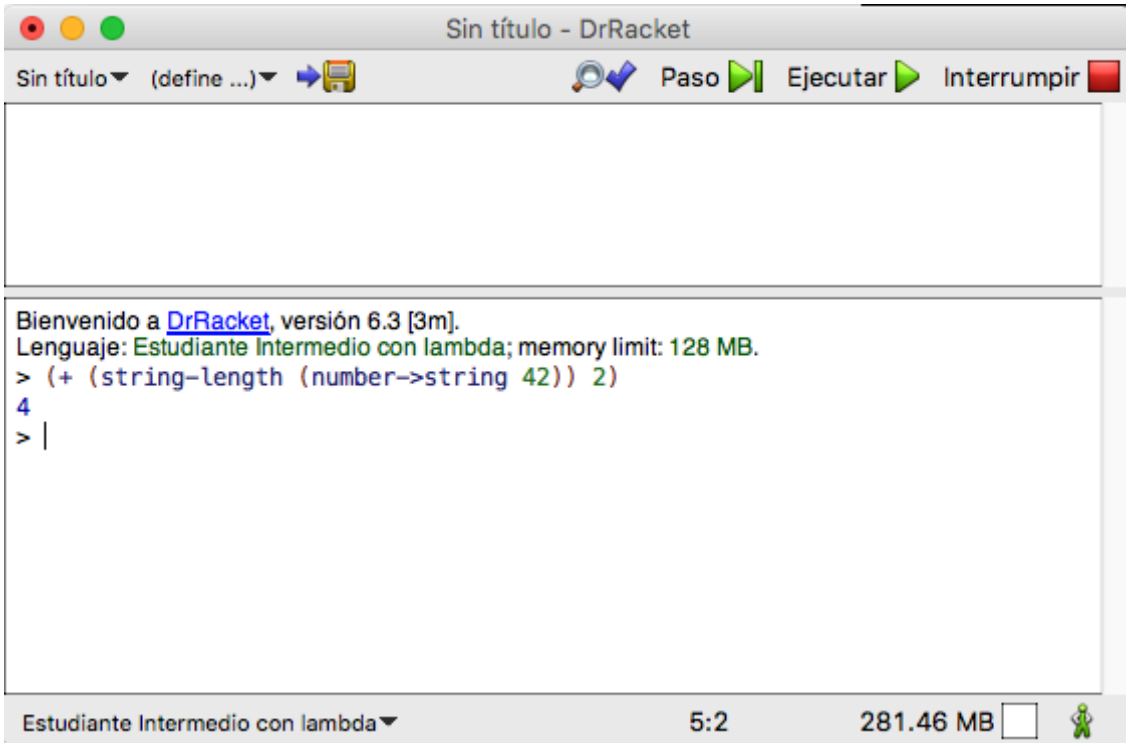
A continuación, utilice primitivas de *strings* para crear una expresión que concatene el prefijo y el sufijo y que añada "_" entre ellos. Cuando se ejecuta este programa, se verá "hola_mundo" en el área de las interacciones.



3.3 Mezclas

El resto de las operaciones relativas a las *strings* consumen y/o producen datos que no son *strings*. Aquí hay unos ejemplos:

- **string-length** consume un *string* y produce un número (natural);
- **string-ith** consume un *string*, junto con un número natural i y luego extrae el dato del *string* ubicado en la iª posición (contando desde 0); y
- **number->string** consume un número y produce un *string*.



Este resultado se produce de la siguiente forma:

```
(+ (string-length (number->string 42)) 2)
==
(+ (string-length "42") 2)
==
(+ 2 2)
==
4
```

3.4 Aritmética de imágenes

Las imágenes representan datos simbólicos se emplean y operan muy similar a los *strings*.

BSL también puede manipular imágenes con operaciones primitivas por ejemplo:

- **circle** produce un circulo a partir un radio, una cadena de modo y una cadena de color;
- **ellipse** produce una elipse a partir de dos radios, una cadena de modo y una cadena de color;
- **line** produce una línea a partir de dos puntos y una cadena de color;
- **rectangle** produce un rectángulo a partir de una anchura, una altura, una cadena modo y una cadena de color;
- **text** produce una imagen de texto a partir de un *string*, un tamaño de fuente y una cadena de color;
- **triangle** produce un triángulo equilátero que apunta hacia arriba a partir de un tamaño, una cadena de modo y una cadena de color.

Las leyes de la aritmética para las imágenes son análogas a las de los números:

```
(+ 1 1) == 2
```

```
(overlay (square 4 "solid" "orange")
         (circle 6 "solid" "yellow"))
==
```

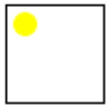


```
(underlay (circle 6 "solid" "yellow")
          (square 4 "solid" "orange"))
==
```



```
(+ 1 2) == 3
```

```
(place-image (circle 6 "solid" "yellow")
             100 100
             (empty-scene 200 200))
==
```



```
(+ 2 2) == 4
```

```
....
```

```
....
```

3.5 Aritmética de booleanos

Necesitamos un último tipo de datos primitivo antes de que podamos diseñar programas: valores booleanos.

Sólo hay dos tipos de valores booleanos: **true** y **false**. Los booleanos se emplean para la representación de decisiones.

Operar con booleanos es simple, BSL cuenta con tres operaciones: **or**, **and**, y **not**. Estas operaciones son una especie de suma, multiplicación, y la negación de los números. Por supuesto, como sólo hay dos valores booleanos, en realidad es posible demostrar cómo trabajan estas funciones en todas las situaciones posibles:

or comprueba si alguno de los valores booleanos ingresado es **true**:

```
> (or #true #true)
#true
> (or #true #false)
#true
> (or #false #true)
#true
> (or #false #false)
#false
```

and comprueba si todos los valores booleanos ingresados son **true**:

```
> (and #true #true)
#true
> (and #true #false)
#false
> (and #false #true)
#false
> (and #false #false)
#false
```

Y **not** siempre muestra el booleano inverso al ingresado:

```
> (not #true)
#false
> (not #false)
#true
```

3.6 Mezclas con booleanos

Un uso importante de los booleanos es evaluar cálculos con diferentes tipos de datos. Por ejemplo, podríamos empezar un programa como este:

```
(define x 2)
```

y luego calcular su inversa:

```
(define inversa-de-x (/ 1 x))
```

Esto funciona bien, siempre y cuando no editemos el programa y cambiamos *x* a *0.*

Aquí es donde entra el uso de los booleanos, en particular, calcular condicionales. En primer lugar, la función primitiva **=** determina si dos (o más) números son iguales. Si es así, se produce **true**, de lo contrario un **false**. En segundo lugar, hay una especie de expresión en BSL que no hemos mencionado hasta ahora: la expresión **if**. Se utiliza la palabra reservada "**if**", como si se tratara de una función primitiva; aquí hay un ejemplo:

```
(if (= x 0) 0 (/ 1 x))
```

La expresión **if** contiene las sub-expresiones *(= x 0), 0,* y *(/ 1 x).* La evaluación de esta expresión procede en dos etapas:

1. La primera expresión siempre se evalúa. Su resultado debe ser un booleano.
2. Si el resultado de la primera expresión es **true**, se evalúa la segunda expresión; de lo contrario la tercera. Cualesquiera que sean sus resultados, también son el resultado de la totalidad de la expresión **if**.

Y, utilizando las leyes de la aritmética, se puede averiguar el resultado de esta interacción:

```
(if (= x 0) 0 (/ 1 x))
== ; because x stands for 2
(if (= 2 0) 0 (/ 1 2))
== ; 2 is not equal to 0, (= 2 0) is #false
(if #false 0 (/ 1 x))
(/ 1 2)
== ; normalize this to its decimal representation
0.5
```

Además del **=**, BSL ofrece una serie de otras primitivas de comparación (**<, <=, >, > =**).

3.7 Predicados: conozca sus datos

DrRacket le permite identificar los errores a través del texto de color rojo en el área de las interacciones.

Una forma de evitar este tipo de accidentes es utilizar un predicado, que es una función que consume un valor y determina si pertenece o no, a alguna clase de datos. Por ejemplo, el predicado number? determina si el valor dado es un número o no:

```
> (number? 4)
#true
> (number? pi)
#true
> (number? #true)
#false
> (number? "fortytwo")
#false
```

Como se ve, los predicados producen valores booleanos. Por lo tanto, cuando los predicados se combinan con expresiones condicionales, los programas pueden proteger las expresiones de un mal uso.

4 Funciones y programas

Ahora que hemos visto que la aritmética funciona igual tanto para números como para cualquier tipo de datos podemos avanzar y enfocarnos en nuestro objetivo.

En lo que se refiere a la programación, la aritmética es la mitad del juego; la otra mitad es el álgebra. Cuando hablamos de álgebra se relaciona con la idea de la escuela, pero en concreto, las nociones de álgebra que se necesitan son: variables, definición de funciones, la

aplicación de funciones y composición de funciones. Y podemos desarrollarla de una manera divertida y accesible.

4.1 Funciones

Los programas son funciones en BSL. Al igual que las funciones, los programas consumen datos de entrada y producen datos de salida. Con la diferencia de que los programas trabajan con una gran variedad de datos: números, cadenas, imágenes, mezclas de todos ellos, y un largo etc. Además, los programas son provocados por los acontecimientos en el mundo real, y las salidas de los programas afectan al mundo real. Por ejemplo, el programa de calendario en un equipo, puede poner en marcha un programa de nómina mensual, en el último día de cada mes.

Definiciones: Si bien muchos lenguajes de programación oscurecen la relación entre los programas y las funciones, BSL nos permite ver esta relación. Cada programa de BSL se compone de varias definiciones, por lo general, seguido de una expresión que involucra a esas definiciones. Hay dos tipos de definiciones:

- definiciones de constantes. de la forma:
  (**define** variable expresión), y
- las definiciones de funciones, que puede ser de n números de formas

Al igual que las expresiones, las definiciones de funciones en BSL vienen en una forma uniforme:

(**define** (nombre de función variable ... variable) expresión)

Es decir, escribimos

- "(**define** (",
- el nombre de la función,
- seguido de tantas varias variables como se requieran, con espacios separándolas y terminando en ")",
- y una expresión seguida de ")".

Y eso es todo lo que hay que hacer para crear una definición de funciones. Estos son algunos ejemplos:

```
(define (f x) 1)

(define (g x y) (+ 1 1))

(define (h x y z) (+ (* 2 2) 3))
```

En términos generales, una definición de función introduce una nueva operación a los datos; dicho de otra manera, se añade una operación a nuestro vocabulario si pensamos en las operaciones primitivas como las que están siempre disponibles. Como una función primitiva, una función

definida consume entradas. El número de variables determina cuántos argumentos o parámetros consume la función.

Los ejemplos anteriores no son funcionales porque las expresiones dentro de las funciones no implican a las variables. Dado que las variables son acerca de entradas, no mencionarlas en las expresiones significa que la salida de la función es independiente de su entrada y, por tanto, siempre la misma. No necesitamos escribir funciones o programas si la salida siempre es la misma.

Considere el siguiente fragmento de una definición:

(**define** (*f a*) ...)

Su cabecera es la función *(f a)*, lo que significa *f* consume un solo dato de entrada, y la variable *a* es un marcador para esta entrada. Por supuesto, en el momento en que definimos una función, no sabemos cuál va a ser su entrada. De hecho, el objetivo de definir una función es que podemos utilizar la función muchas veces con muchas entradas diferentes.

El cuerpo de la función se refiere a los parámetros de la función. Si completamos la definición de *f* a esto;

*(**define** (f a)*

(10 a))*

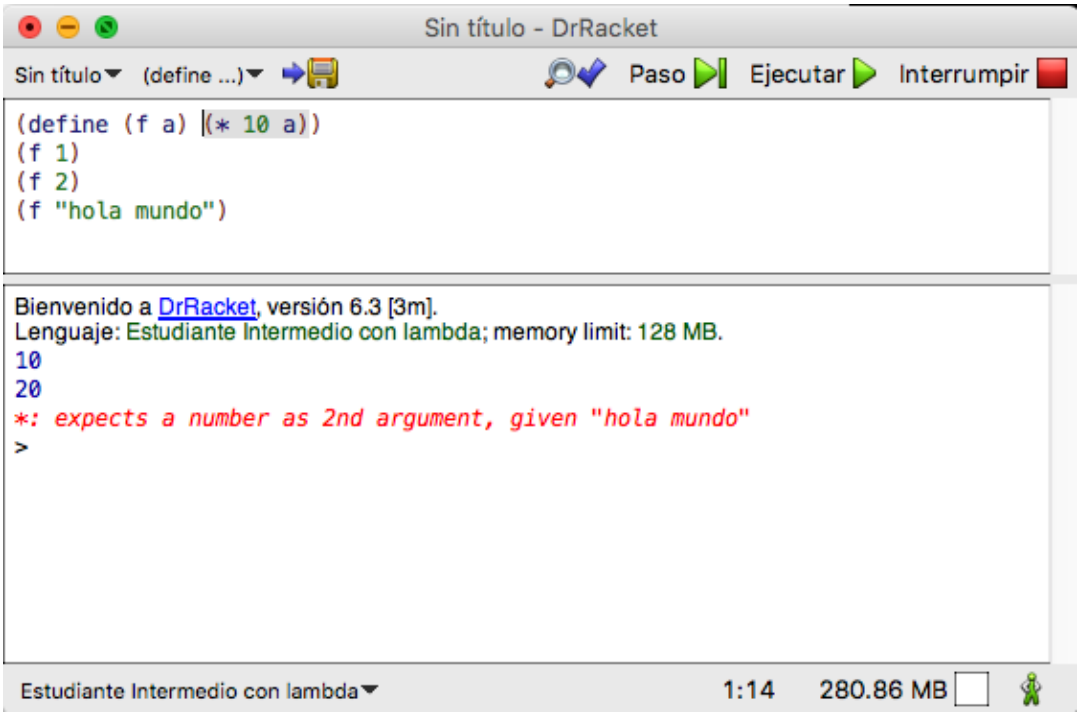
Estamos diciendo que la salida de una función es diez veces su entrada.

Por ahora, la única cuestión pendiente, es cómo una función obtiene sus entradas. Y para ello, nos dirigimos a la idea de aplicar una función.

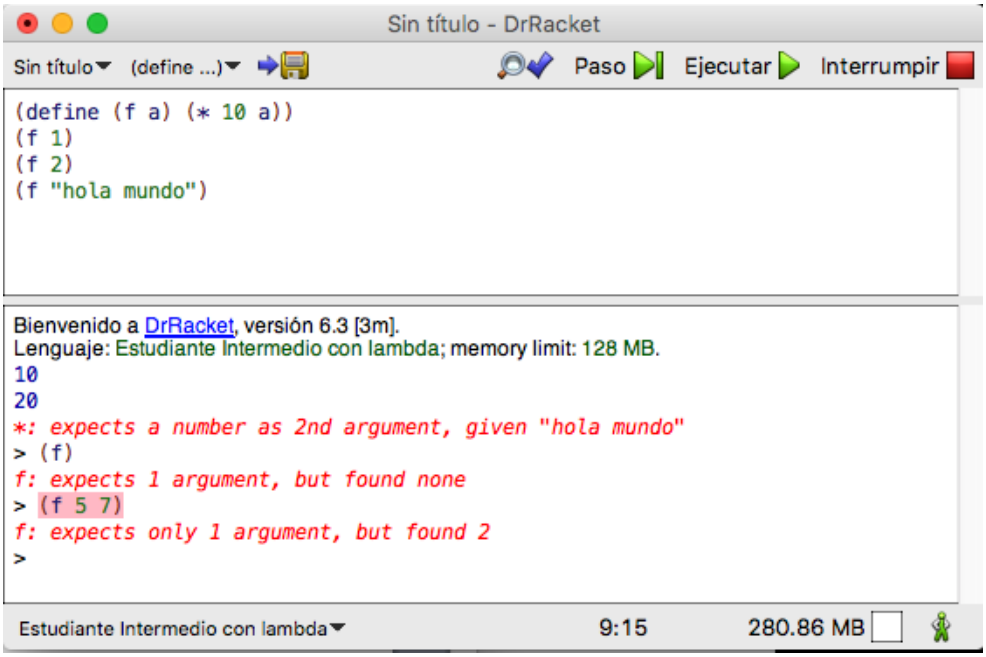
Aplicaciones: una aplicación de función **define** funciones con las que ponemos operar y se parece a las aplicaciones de una operación predefinida:

- se escribe "*(*",
- después el nombre de una función definida por ejemplo *f*,
- y se anotan tantos argumentos como *f* consume, separadas por un espacio, y
- escribe "*)*".

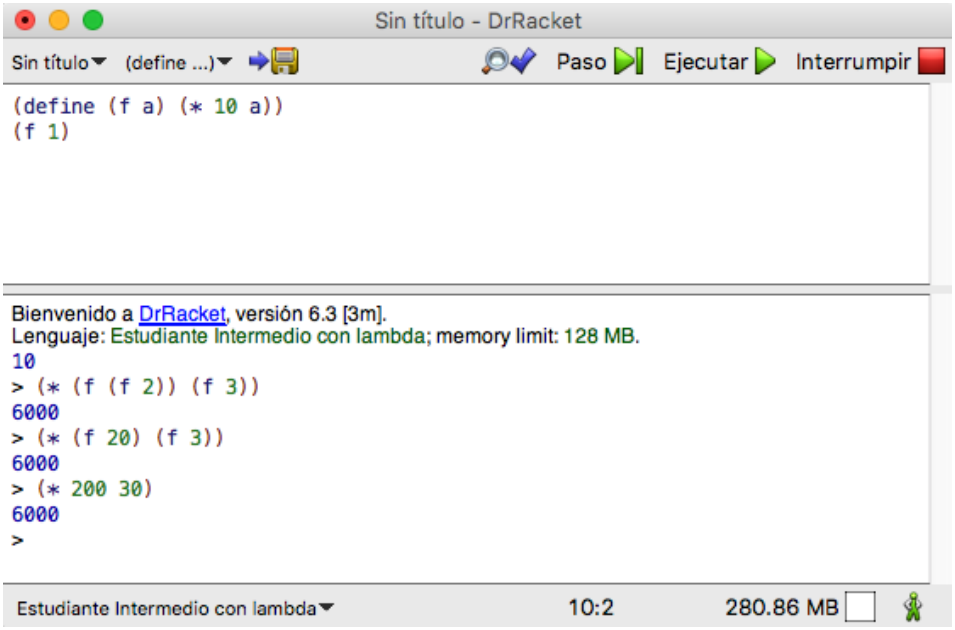
Con este ejemplo, podemos experimentar con funciones y también con las distintas primitivas para averiguar lo que ocurre.



Los resultados que obtenemos nos muestran que las entradas no afectan el comportamiento de la función, así mismo DrRacket nos muestra el porqué del fallo de la ejecución del tercer ejemplo, de igual forma si faltan o sobran argumentos nos mostrará los mensajes correspondientes.



De igual forma podemos experimentar anidando funciones, en otras palabras DrRacket nos permite llamar la función y emplearla como entrada de otra o de la misma función:



Con este ejemplo demostramos que es perfectamente aceptable el uso de aplicaciones de funciones anidadas dentro de otras aplicaciones de función.

4.2 Computación

Las definiciones y aplicaciones de funciones trabajan en conjunto. Si se desea diseñar programas, se debe entender esta colaboración, porque hay que imaginar cómo DrRacket ejecuta los programas y porqué se tiene que averiguar, lo que va mal cuando las cosas no salen como se esperan.

Si bien es posible que haya visto esta idea en un curso de álgebra, preferimos explicar a nuestra manera.

La evaluación de una aplicación de la función en tres pasos: DrRacket determina los valores de los argumentos en las expresiones; comprueba que el número de argumentos y el número de parámetros de la función sean el mismo; si es así, DrRacket calcula el valor del cuerpo de la función, con todos los parámetros reemplazados por los valores de los argumentos correspondientes.

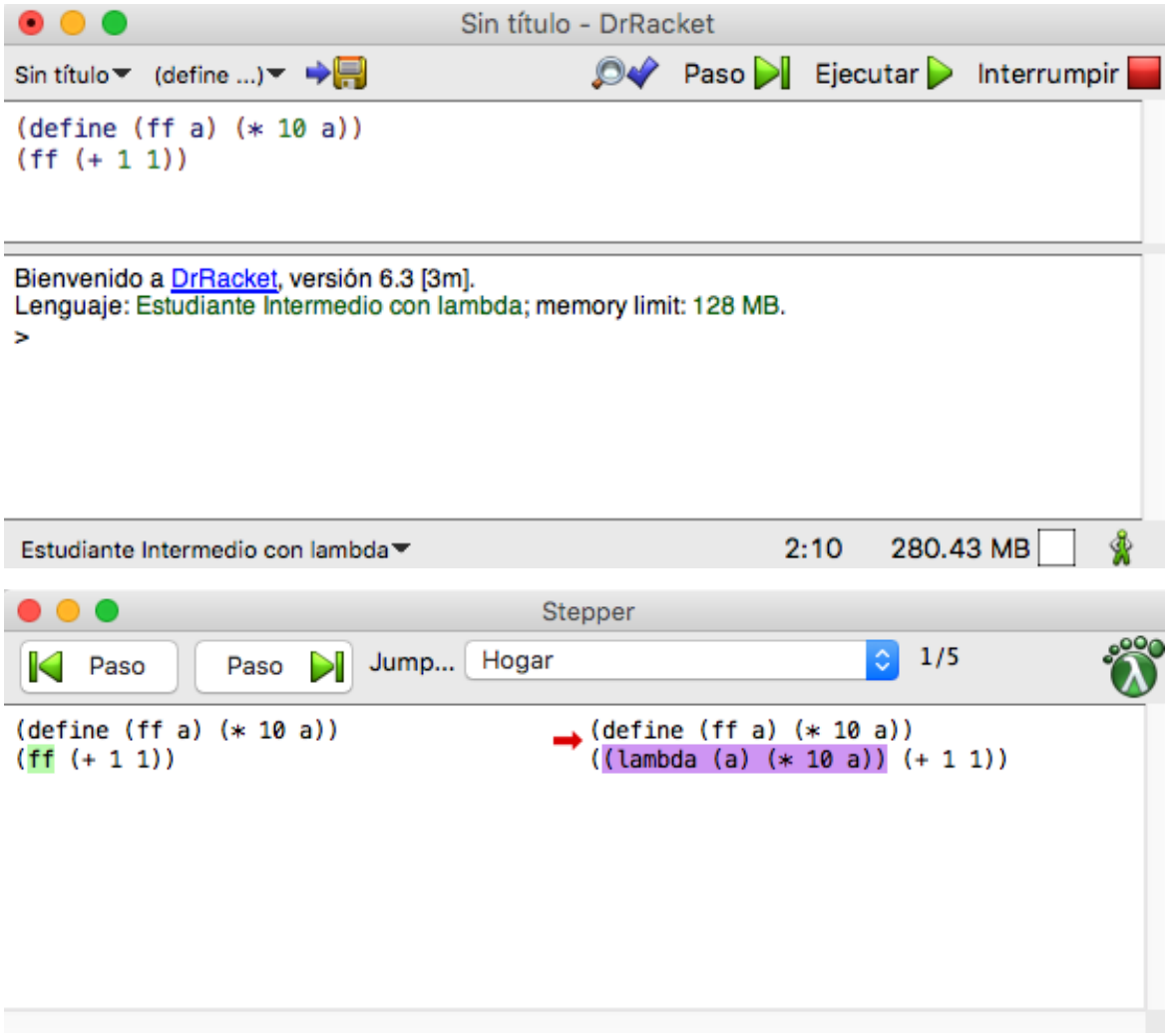
Lo mejor es que cuando se combinan las leyes del cómputo con las de la aritmética, más o menos se puede predecir el resultado de cualquier programa en BSL:

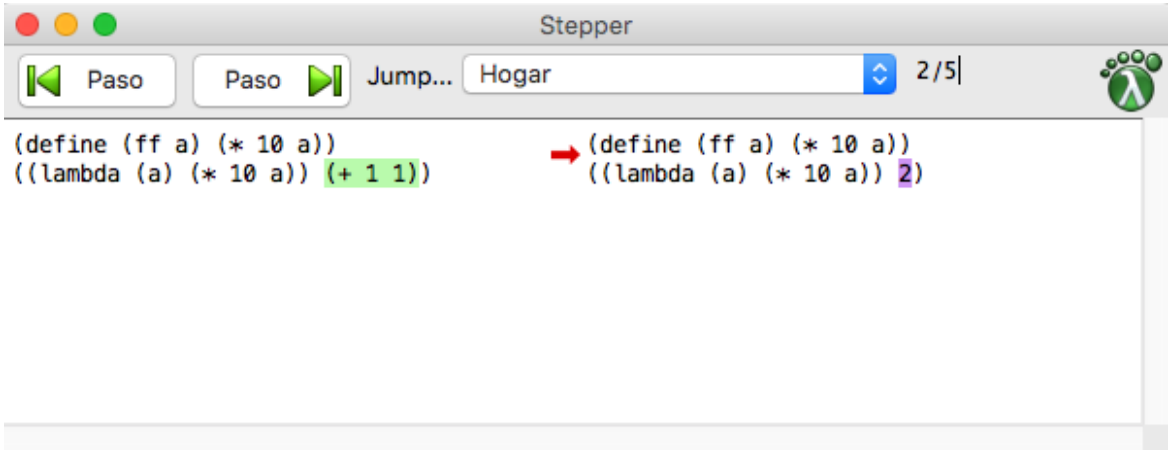
```
(+ (ff (+ 1 2)) 2)
== ; DrRacket knows that (+ 1 2) == 3
(+ (ff 3) 2)
== ; DrRacket replaces a with 3 in (* 10 a), ff's body
(+ (* 10 3) 2)
== ; now DrRacket uses the laws of arithmetic
(+ 30 2)
==
32
```

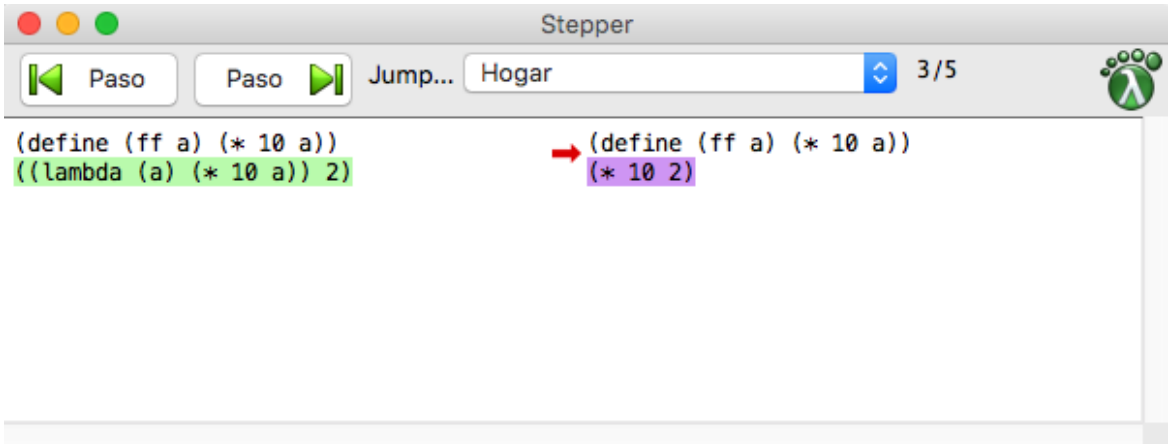
En resumen, DrRacket es básicamente un estudiante de álgebra increíblemente rápido y preciso; que conoce todas las leyes de la aritmética y es muy bueno para la sustitución. Aún mejor, DrRacket no sólo puede determinar el valor de una expresión; sino que también le puede mostrar cómo lo hace. Es decir, se puede mostrar paso a paso

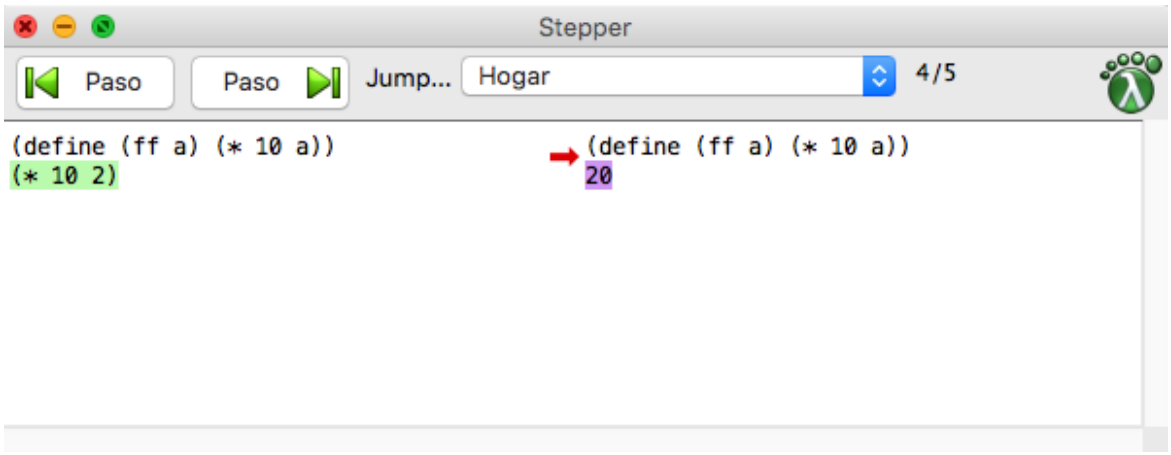
cómo resolver estos problemas de álgebra que le piden, para determinar el valor de una expresión.

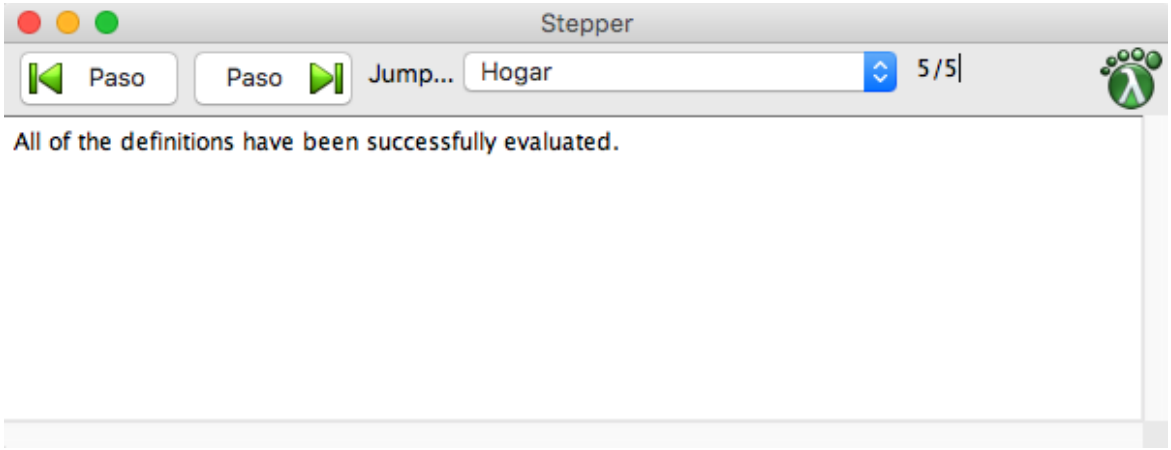
Si se hace clic en el botón de paso a paso, aparece una ventana emergente en la que se puede recorrer la evaluación del programa en el área de las definiciones.




Stepper
Paso
Paso
Jump...
Hogar
2/5
(define (ff a) (* 10 a))
((lambda (a) (* 10 a)) (+ 1 1))
(define (ff a) (* 10 a))
((lambda (a) (* 10 a)) 2)


Stepper
Paso
Paso
Jump...
Hogar
3/5
(define (ff a) (* 10 a))
((lambda (a) (* 10 a)) 2)
(define (ff a) (* 10 a))
(* 10 2)

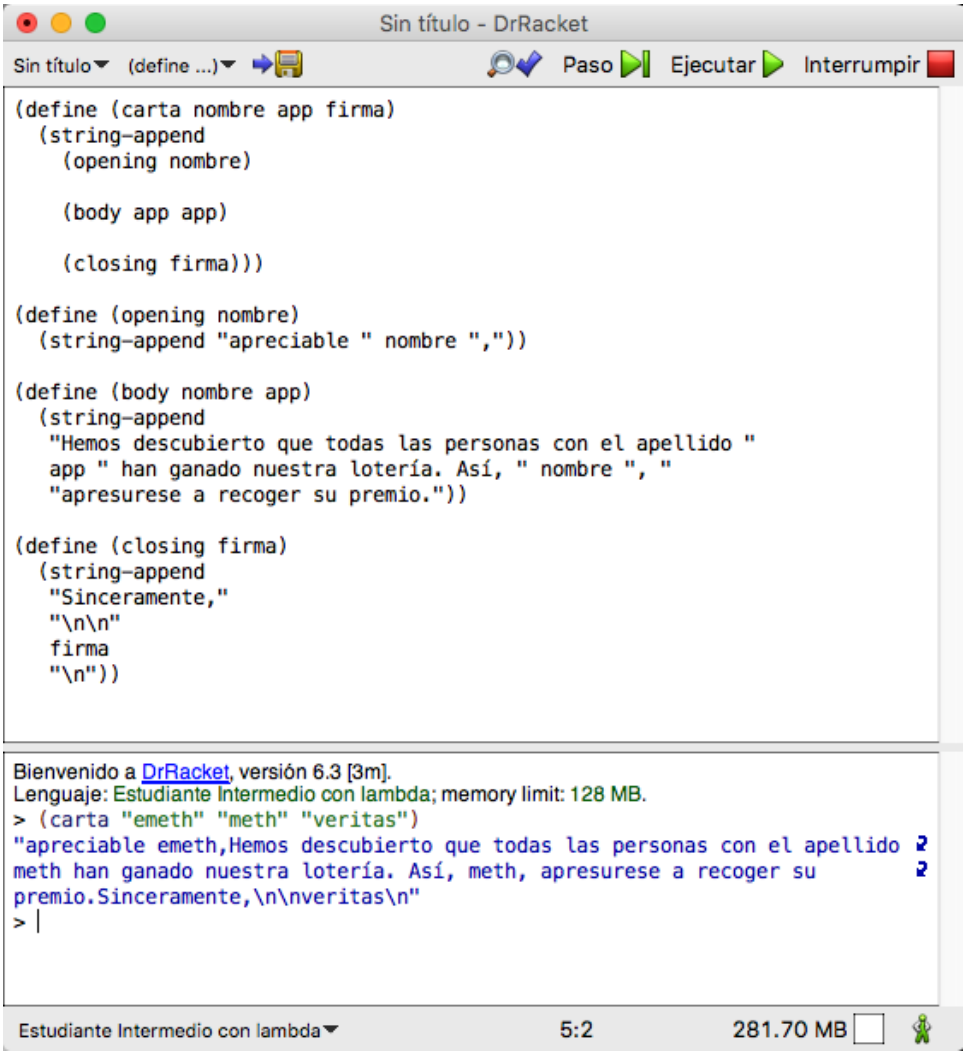

Stepper
Paso
Paso
Jump...
Hogar
4/5
(define (ff a) (* 10 a))
(* 10 2)
(define (ff a) (* 10 a))
20



4.3 Composición de funciones

Un programa rara vez consta de una sola definición de función y una aplicación de esa función. En cambio, un programa típico consiste en una función "principal", que utiliza otras funciones y convierte el resultado de una aplicación de función en la entrada para otra.

Un ejemplo que consta de 4 funciones:



La primera de ellas es una función que utiliza tres funciones auxiliares para producir las tres piezas de la carta, la apertura, el cuerpo y la firma y luego compone los tres resultados en el orden correcto con el **string-append**.

En general, cuando un problema se refiere a tareas distintas, un programa debe consistir en una función por tarea y una función principal que lo una todo. Formulamos esta idea como un simple lema:

Definir una función por tarea.

La ventaja de seguir esta consigna es que se llega razonablemente a pequeñas funciones, cada una de las cuales es fácil de comprender, y cuya composición es fácil de entender. Una vez que se aprende a diseñar funciones, se reconocerá que conseguir funciones pequeñas para que opere correctamente es mucho más fácil, que hacerlo con grandes funciones. Mejor aún, si alguna vez se tiene que cambiar una parte del programa debido a algún cambio en el planteamiento del problema, es mucho más fácil encontrar las partes pertinentes cuando se organiza como un conjunto de funciones pequeñas, en lugar de una sola gran función.

4.4 Constantes globales

Cada lenguaje de programación permite a los programadores definir constantes. En BSL, esta definición tiene la siguiente forma:

- escribir "(**define**",
- anote el nombre,
- seguido de un espacio y una expresión, y
- escribe ")".

```
; the current price of a movie ticket
(define CURRENT-PRICE 5)

; useful to compute the area of a disk:
(define ALMOST-PI 3.14)

; a blank line:
(define NL "\n")

; an empty scene:
(define MT (empty-scene 100 100))
```

Las dos primeras son constantes numéricas, las dos últimas son una cadena y una imagen. Por convención, usamos letras mayúsculas para las constantes globales, ya que asegura que no importa qué tan grande es el programa, los lectores de nuestros programas pueden distinguir fácilmente dichas variables de los demás.

Todas las funciones en un programa pueden referirse a estas variables globales. Una referencia a una variable es como usar las constantes. La ventaja de utilizar los nombres de variables en lugar de constantes es que solo se debe editar la definición y se modificará en todas las funciones que la estén empleando.

4.5 Programas

Con lo anterior ya estamos listos para crear programas sencillos. Desde una perspectiva de codificación, un programa es sólo un montón de funciones y definiciones de constantes. Por lo general, una de las funciones se destaca como la función "principal", y esta función principal tiende a unir las demás. Desde la perspectiva del lanzamiento de un programa, sin embargo, hay dos clases distintas:

- un programa por lotes consume todas sus entradas a la vez y calcula su resultado. Su función principal alimenta a las funciones auxiliares, que puede referirse a funciones auxiliares adicionales, y así sucesivamente. Cuando lanzamos un programa por lotes, el sistema operativo llama a la función principal y espera a la salida del programa.

- un programa interactivo consume algunas de sus entradas, calcula, produce alguna salida, consume más entradas, y así sucesivamente. La función principal de un programa orientado a eventos utiliza una expresión para describir las funciones que llamar y para qué tipo de eventos. Estas funciones se llaman controladoras de eventos.

4.6 Cómo diseñar programas

Aprender a programar requiere cierto dominio de muchos conceptos. Por un lado, la programación necesita, una notación para comunicar lo que queremos calcular. En resumen tenemos que aprender el vocabulario del lenguaje de programación, para averiguar su gramática, y para entender lo que sus "frases" significan.

Por otro lado, es fundamental aprender cómo llegar del enunciado del problema a un programa. Tenemos que determinar lo que es relevante en el planteamiento del problema y lo que no. Necesitamos desentrañar lo que consume el programa, lo que produce, y cómo se relaciona las entradas a las salidas. Tenemos que saber, o averiguar, si el lenguaje elegido y sus bibliotecas proporcionan ciertas operaciones básicas para los datos que nuestro programa debe procesar. Si no, tendremos que desarrollar funciones auxiliares que implementen estas operaciones. Por último, una vez que tenemos un programa, hay que comprobar si se realiza efectivamente el cómputo previsto. Y esto podría revelar todo tipo de errores, que tenemos que ser capaces de entender y corregir.

4.7 Diseño de funciones

El propósito de un programa es describir un proceso, que trabaje con la información de entrada y que produzca nueva información. En este sentido, un programa es como las instrucciones que un profesor de

matemáticas les da a los estudiantes de escuela primaria. Con la diferencia de que un programa trabaja con más que números; se calcula con la información de navegación, busca la dirección de una persona, se activa interruptores, o inspecciona el estado de un videojuego. Toda esta información viene de una parte del mundo real, a menudo llamada dominio del programa y los resultados del cálculo de un programa representa más información en este dominio.

Para que un programa pueda procesar la información, debe convertir la información del mundo real en algún tipo de datos en el lenguaje de programación; entonces procesa los datos; y una vez que esté terminado, convierte los datos resultantes en información de nuevo. Un programa interactivo puede incluso entremezclan estos pasos.

Usamos BSL y DrRacket de modo que no tenemos que preocuparnos por la traducción de la información en datos. El BSL de DrRacket puede aplicar una función directamente a los datos y observar lo que produce.

El Proceso Una vez que entienda cómo representar información de entrada como datos e interpretar los datos de salida como nueva información, el diseño de una función individual es un proceso sencillo:

1. Expresar para lo que se desea representar con la información como datos. Un comentario de una sola línea, basta, por ejemplo,

; Utilizamos números simples para representar temperaturas.

2. Escriba una firma, una declaración de propósito y una cabecera de función.

   Una *firma de función* es un comentario en BSL que les dice a los lectores del diseño cuántas entradas consume su función, y qué tipo de datos de salida produce. He aquí dos ejemplos:

   - para una función que consume una cadena y produce un número:

     ; Cadena -> Número

   - para una función que consume una temperatura y que produce una cadena:

     ; Número -> Cadena

3. Ilustrar la firma y la declaración de propósito con algunos ejemplos funcionales. Para construir un ejemplo, se elige un dato de cada clase de entrada de la firma y se determina lo que espera obtener.

   Supongamos que se está diseñando una función que calcula el área de un cuadrado. Es evidente que esta función consume la longitud del lado y que está mejor representado con un número (positivo).

```
; Number -> Number
; computes the area of a square whose side is len
(define (area-of-square len) 0)
```

Añadir los ejemplos entre la declaración de propósito y la cabecera o encabezamiento de la función:

```
; Number -> Number
; computes the area of a square whose side is len
; given: 2, expect: 4
; given: 7, expect: 49
(define (area-of-square len) 0)
```

4. El siguiente paso es hacer un inventario, para entender lo que recibimos y lo que tenemos que calcular.

```
; Number -> Number
; computes the area of a square whose side is len
; given: 2, expect: 4
; given: 7, expect: 49
(define (area-of-square len)
   (... len ...))
```

5. Ahora es tiempo de codificar. En general, la codificación significa reemplazar el cuerpo de la función con una expresión que intenta calcular a partir de los tipos de datos del formato lo que promete la declaración del propósito.

```
; Number String Image -> Image
; adds s to img, y pixels from top, 10 pixels to the left
; given:
; 5 for y,
; "hello" for s, and
; (empty-scene 100 100) for img
; expected:
; (place-image (text "hello" 10 "red") 10 5 (empty-scene 100 100))
(define (add-image y s img)
  (place-image (text s 10 "red") 10 y img))
```

6. El último paso de un diseño adecuado es poner a prueba la función con los ejemplos que se trabajaron antes.

```
> (area-of-square 2)
4
> (area-of-square 7)
49
```

4.8 Conocimiento del dominio

Es natural preguntarse qué conocimiento se necesita para codificar el cuerpo de una función. Un poco de reflexión nos dice que este paso exige una comprensión apropiada del dominio del programa. De hecho, hay dos formas de tal conocimiento del dominio:

1. Conocimiento de dominios externos como las matemáticas, la música, la biología, la ingeniería civil, el arte, etc. Debido a que los programadores no pueden conocer todos los ámbitos de aplicación de la computación, debemos estar preparados para entender el

lenguaje de una variedad de áreas de aplicación para que puedan discutir problemas con expertos en el dominio.

2. Y el conocimiento sobre las funciones de la biblioteca en el lenguaje de programación elegido. Cuando su tarea es traducir una fórmula matemática que implica la función tangente, lo que se necesita saber es, si el idioma elegido viene con una función que calcula la tangente.

Dado que nunca se puede predecir el área en que va a trabajar, o lenguaje de programación que tendrá que utilizar, es imprescindible que se tenga una comprensión sólida de todas las posibilidades de cualquier lenguaje de programación.

4.9 De funciones a programas

No todos los programas consisten en una sola definición de la función. Algunos requieren varias funciones, muchos también usan definiciones constantes.

Cuando haya definido constantes globales, sus funciones pueden utilizar esas constantes globales para calcular los resultados. Para acordarse de su existencia, es posible que desee agregar estas constantes a sus formatos; después de todo, pertenecen al inventario de las cosas que pueden contribuir a la definición de función.

4.10 Pruebas

Las pruebas se convierten en una tarea laboriosa. Si bien son fáciles de ejecutar en pequeños programas en el área de interacciones, hacerlo requiere una gran cantidad de mano de obra mecánica. Como los programas tienden a crecer y sus sistemas requieren llevar a cabo muchas pruebas. Pronto este trabajo se vuelve abrumador, y los programadores empiezan a descuidarlo.

Por lo tanto, es crítico para mecanizar pruebas en lugar de realizar manualmente. Como muchos lenguajes de programación, BSL incluye un centro de pruebas, tomemos como ejemplo a la función que convierte las temperaturas de grados Fahrenheit a Celsius.

```
; Number -> Number
; converts Fahrenheit temperatures to Celsius temperatures
; given 32, expect 0
; given 212, expect 100
; given -40, expect -40
(define (f2c f)
  (* 5/9 (- f 32)))
```

Para la prueba de este ejemplo, la función requiere tres cómputos y tres comparaciones entre dos números cada uno:

```
(check-expect (f2c -40) -40)
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)
```

5 Identificación del problema:

Consultando la Bitácora FI[13] (recurso nuevo de la facultad) la mayoría de los alumnos de primer ingreso se dan cuenta de que lo que se les dificulta en la carrera son los antecedentes de álgebra, siendo los temas más mencionados exponentes, despejes (pasar de variables a funciones) y trigonometría.

Semana 2	Los temas que más se me dificultaron esta semana fueron... Debido a...
Pues yo diria que la materia en general como Algebra, calculo, quimica y fisica ya que no tengo muchos antecedentes de estas materias y se me esta dificultando bastante.	

[13]Ingeniería, F. d. (09 de 12 de 2015). *UNAM Facultad de Ingeniería*. Recuperado 09 de 12 de 2015 de Bitacora FI: http://www.ingenieria.unam.mx/~bitacoraFI/bitacora/

Semana 2	Algo que considero que me faltó para comprender mejor los temas vistos en mis clases fue...
Las funciones Trigonométricas, ya que siempre me han fallado ese tipo de funviones, siempre me confundó.	

Semana 3	Algo que considero que me faltó para comprender mejor los temas vistos en mis clases fue...
fue los despejes del modelo de J.J. Thompson, leyes de los senos y cosenos	

Semana 2	Los temas que más se me dificultaron esta semana fueron... Debido a...
En Álgebra se me dificulto la trigonométrica y el circulo unitario. La trigonométrica es debido a que no la he repasado y se me olvido que hacer y el circulo unitario nunca antes lo había visto y la verdad no entendí como funciona. En Química lo que mas se me dificulta es igualar ecuaciones, debido a que no lo he visto en más de dos años y no lo recuerdo.	

Semana 4	Los temas que más se me dificultaron esta semana fueron... Debido a ...
Los tipos de funciones. Creo que en particular las definiciones. Se me hacen un tanto formales, pero al investigar por mi cuenta también vienen así, de la misma manera que en clase.	

El álgebra es la rama de la matemática que estudia la combinación de elementos de estructuras abstractas acorde a ciertas reglas. Estos elementos pueden ser interpretados como números o cantidades, por lo que el álgebra en cierto modo es una generalización y extensión de la aritmética.

Estos elementos que se pueden interpretar como números son el problema raíz, ya que ese paso de pasar de un número a ser una variable y de ahí a desarrollar funciones, es en donde los alumnos se pierden, lo que conlleva a acarrear deficiencias y ya que el estudio de las matemáticas es de forma creciente y gradual, el tener deficiencias en los

primeros pasos, hace que continuar con el estudio de la materia sea una experiencia tortuosa y difícil. Por tanto podemos apoyarnos en el desarrollo de programas, para ayudar a los alumnos a superar esta problemática con el fin de que su estudio de las matemáticas no represente un gran reto y así evitar la deserción de los alumnos.

5.1 Del pensamiento matemático a computacional.

Cuando cambiamos del cálculo matemático al computacional, el mapeo de información a datos es esencial. Hoy en día los programas computacionales procesan representaciones de música, videos, moléculas, compuestos químicos, negocios, diagramas eléctricos y planos. La computación generaliza la aritmética y el álgebra para que los programas puedan operar con cadenas, booleanos, caracteres, estructuras y funciones.

Al igual que los números, los datos vienen con operaciones básicas. Operar se refiere a aplicar esas funciones a los datos. La computación obedece leyes al igual que las leyes de las ecuaciones, estas explican cómo se procesan los datos. También significa utilizar funciones, al igual que en las matemáticas, combina operaciones básicas con otras funciones. Donde tenemos dos mecanismos de combinación importantes: la función composición y las expresiones condicionales. La primera se refiere a que el resultado de una función llega como argumento de otra función. La segunda representa un cambio entre

muchas posibilidades. Cada uno de esos mecanismos de combinación viene con sus respectivas leyes que los gobiernan cuando se encuentran combinados.

Los programas consisten en muchas funciones, a veces miles y millones, y al correr un programa significa aplicar todas esas funciones. Las personas son muy lentas para realizar estas tareas cuando involucran volúmenes de datos enormes y un gran número de funciones y operaciones. En lugar de eso se utilizan computadoras para manejar las demandas actuales, porque son extremadamente buenas usando las leyes de la aritmética y el álgebra.

Entonces por qué no aprovechar la facilidad que nos ofrecen las computadoras, para hacer que los alumnos aprendan los conceptos de álgebra en los que presentan dificultades, de una forma visual y dinámica, que fomente su interés en el aprendizaje.

5.2 Habilidades para el diseño

El desarrollo de programas requiere de una serie de habilidades que son importantes para todas las profesiones: leer críticamente, pensar analíticamente, sintetizar de forma creativa y atender cuidadosamente los detalles.

5.2.1 Leer críticamente[14]

La lectura crítica es una lectura activa. Implica más que solamente comprender lo que un escritor está diciendo. La lectura crítica implica dudar y evaluar lo que el escritor está diciendo, y formar sus propias opiniones sobre lo que el escritor está diciendo.

- Tomar en consideración el contexto de lo escrito.
- Cuestionar las aseveraciones hechas por el escritor.
- Comparar lo escrito con otro trabajo escrito sobre el tema.
- Analizar los supuestos hechos por el escritor.
- Evalar las fuentes que el escritor usa.

5.2.2 Pensar analíticamente

Realmente no analizamos las circunstancias que dan forma y causa, a los distintos problemas y situaciones a las que nos enfrentamos y para desarrollar una posible solución de la manera más óptima, es necesario pensar analíticamente, evaluar las causas y consecuencias de las distintas acciones que realizamos para llegar a la solución o a la obtención de los datos y/o situaciones.

[14]Resources, A. M.-S. (2001). *métodos de estudio*. Recuperado el 09 de 12 de 2015, de how to study.com: http://www.how-to-study.com/metodos-de-estudio/lectura-critica.asp

5.2.3 Sintetizar de forma creativa

Una vez que ya analizamos y comprendemos lo que se nos solicita y entendemos las causas y consecuencias, estamos listos para nuestra siguiente etapa, el desarrollo de una solución, sintetizando los posibles métodos, de tal manera que podamos implementar nuevas ideas que propongan un reto y un nuevo desafío para la sociedad, y con esto consigamos nuevas y mejores formas de obtener resultados a los problemas que se presentan día a día.

5.2.4 Atender cuidadosamente los detalles

Mientras nos encontramos desarrollando la o las ideas que darán solución a nuestros problemas, es importante regresar a la planeación o la presentación del problema para asegurarnos que estamos obteniendo lo esperado.

5.2.5 Recetas de diseño[15]

El diseño de programas requiere las habilidades analíticas de la lectura cuidadosa y la escritura de la lengua natural. Sin estas habilidades críticas de lectura, un estudiante no puede diseñar programas que cumplan las especificaciones, los buenos métodos de programación

[15]Krishnamurthi, M. F. (s.f.). *Understanding the Program's Purpose:*. Recuperado el 11 de 12 de 2015, de HTDP: http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-5.html#node_sec_2.5

propician, que el estudiante formule lógica y lingüísticamente su pensamiento y desarrollar buenos métodos de programación, ayuda a organizar y expresar el pensamiento de forma apropiada en lenguaje natural.

LA RECETA DE DISEÑO PARA FUNCIONES
análisis del problema y definición de datos
contrato, propósito y declaración de efectos, encabezamientos
ejemplos
formato de función
definición de función
pruebas

Tabla 1 pasos básicos de una receta de diseño de programas

La Tabla 1 muestra los seis pasos básicos de una lista de verificación de una receta de diseño. Cada uno de ellos genera los correspondientes productos intermedios.

1. la descripción de clase de datos del problema.
2. la especificación informal del comportamiento del programa.
3. la ilustración con ejemplos del comportamiento del programa.
4. el desarrollo del formato u organización del programa.
5. la transformación del formato en una definición completa.
6. el descubrimiento de errores mediante pruebas.

6 Método

Con base a estos puntos y a los antecedentes que se requieren se debe considerar que el enseñarles a desarrollar de forma formal (empleando recetas) a programar a los alumnos de primer ingreso les permitirá tener un mejor análisis y comprensión de sus dudas no solo de álgebra, si no que les ayudará a mejorar su desempeño a lo largo de toda la carrera.

También nos sirve para quitar la idea de que los programas son aburridos y que al ejecutarlos nos arrojarán los mismos datos cada vez, DrRacket es una herramienta muy útil porque permite trabajar con distintos tipos de datos no solo con números.

La programación[16]básicamente son cálculos matemáticos en un orden lógico, tomando como referencia los datos de entrada para poder aplicar las operaciones correspondientes a fin de conseguir datos de salida congruentes con lo esperado para poder dar una solución a las problemáticas planteadas, en esencia la programación es netamente matemática aplicada, pero la aritmética es solo la mitad del camino ya que para programar funciones se necesita del álgebra, es por eso que, si nos apoyamos en desarrollos prácticos y dinámicos podemos hacer que el paso de la aritmética al álgebra sea muy natural y fácil para los estudiantes.

El propósito de la programación es hacer frente a una gran cantidad de datos y obtener un montón de diferentes resultados, con más o menos los mismos cálculos. (También debe calcular estos resultados de forma rápida, por lo menos lo más rápido que podamos.) Es decir, se necesita aprender a analizar las problemáticas, antes de saber cómo programar.

[16]Krishnamurthi, M. F. (s.f.). *Fixed-Size Data*. Recuperado el 23 de 12 de 2015, de HTDP second edition: http://www.ccs.neu.edu/home/matthias/HtDP2e/part_one.html

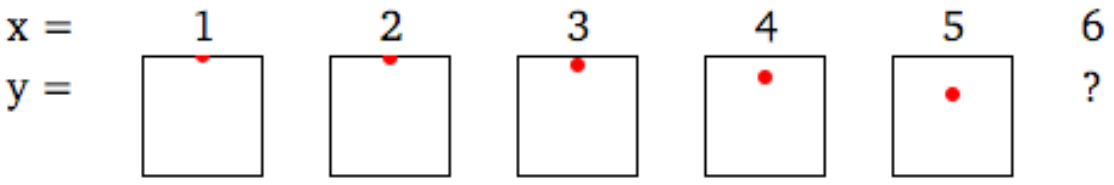
Tomemos el siguiente ejemplo de la segunda edición del libro HtDP[17], para demostrar este concepto antes de profundizar en el método.

Recordemos esas tablas que los profesores muestran cuando empezamos a trabajar con variables:

x =	1	2	3	4	5	6	7	8	9	10
y =	1	4	9	16	25	36	49	64	81	?

El profesor normalmente pediría encontrar el valor desconocido, del número faltante.

Resulta que hacer una película no es más complicado que completar una tabla de números.



También después de dominar estas tablas los profesores no sólo pedían el quinto, sexto o séptimo número de alguna secuencia, sino que ahora

[17] Krishnamurthi, M. F. (s.f.). *HTDP second edition*. Recuperado el 12 de 12 de 2015, de Inputs and Output: http://www.ccs.neu.edu/home/matthias/HtDP2e/part_prologue.html#%28part._some-i%2Fo%29

también una expresión que determina cualquier elemento de la secuencia de un determinado número *x*.

$$y = x * x$$

Y si analizamos más afondo esta secuencia para cuando *x* toma los valores de 1, 2, 3, 4 los valores que obtiene *y* serían 1, 4, 9, 16 y podríamos reducir la expresión:

$$y = x^2$$

Este tipo expresiones es la forma en la que empezamos a analizar las funciones, de tal forma que el alumno puede identificar más fácilmente las entradas y las salidas de datos y poder interpretar mejor los resultados.

En DrRacket para implementar funciones se tienen que definir, la forma de definirlas tiene una estructura muy intuitiva por lo que esta herramienta tampoco es difícil de comprender:

```
(define (y x) (* x x))
```

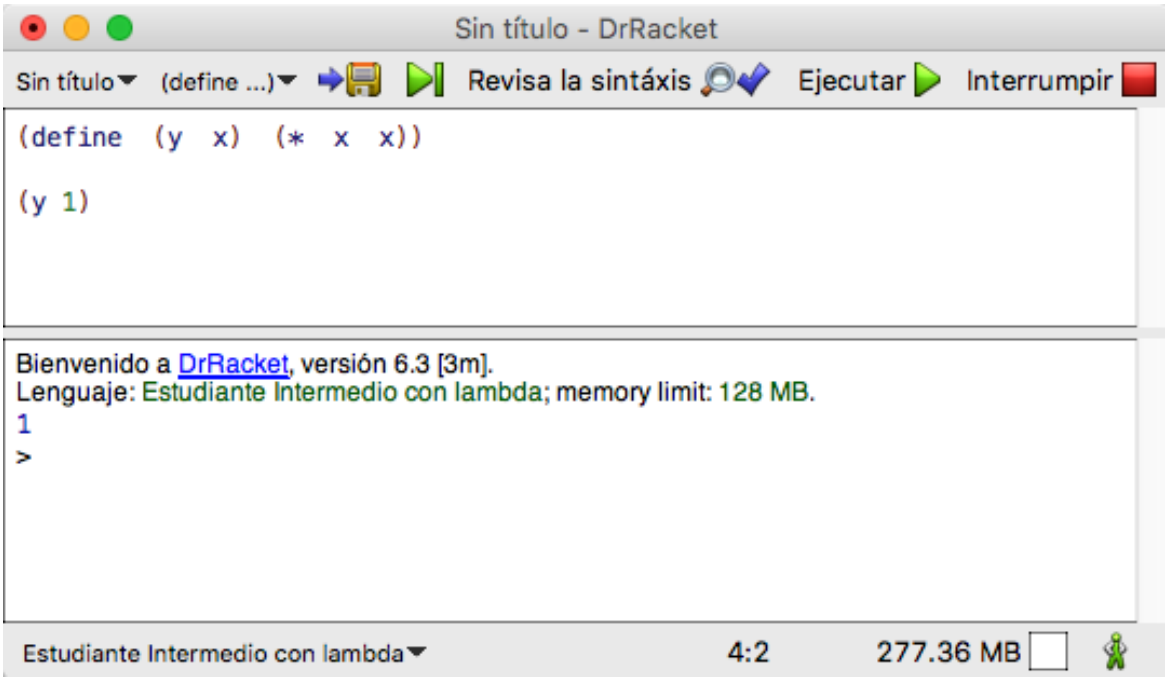
La sentencia **define** dice "considerar (variables entrada y salida) y una función", que al igual que una expresión, calcula un valor. El valor de una

función, sin embargo, depende del valor de algo llamado la entrada, lo que expresamos con *(y x)*. Dado que no sabemos qué es esto de entrada, se utiliza un nombre para representar la entrada. Siguiendo la tradición matemática, utilizamos *x* aquí para sustituir a la entrada desconocida.

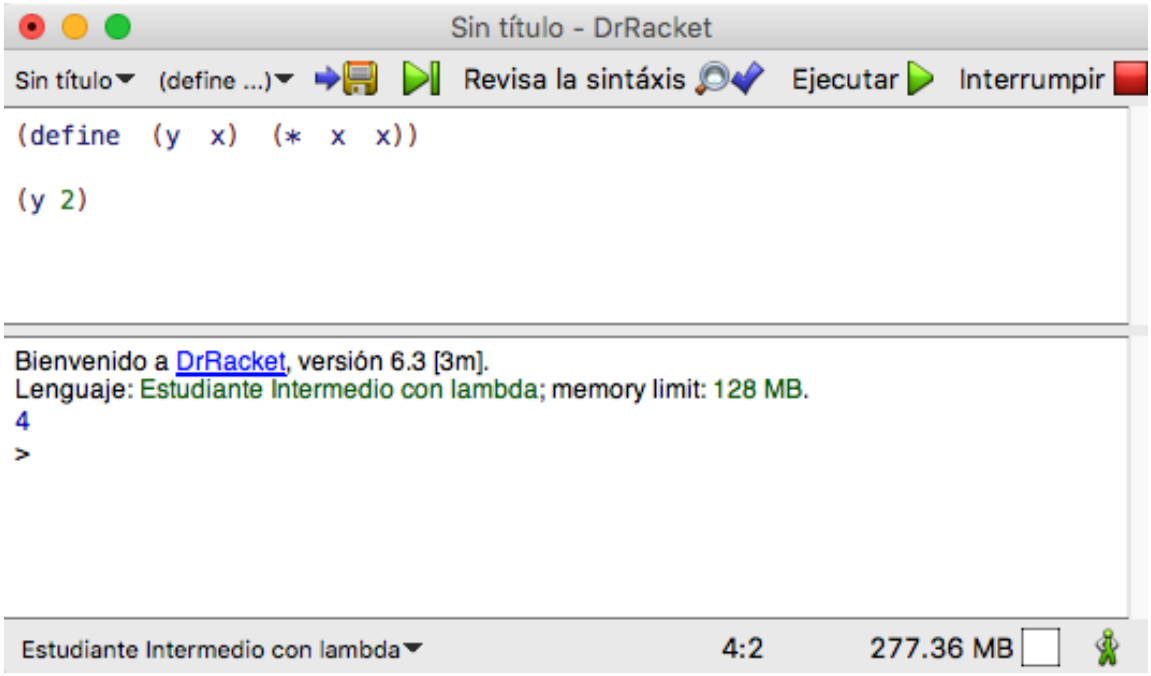
La segunda parte significa que debe suministrar un solo valor de un número de *x* para determinar un valor específico para *y*. DrRacket enchufa en el valor de x en la expresión asociada a la función. Aquí la expresión es *(* x x).* Una vez que x se sustituye con un valor, por ejemplo 1, DrRacket puede calcular el resultado de las expresiones, que también se llaman salida de la función.

Al hacer clic en ejecutar, se definió la función y se informó a DrRacket de su existencia y ya está listo para hacer uso de la función.

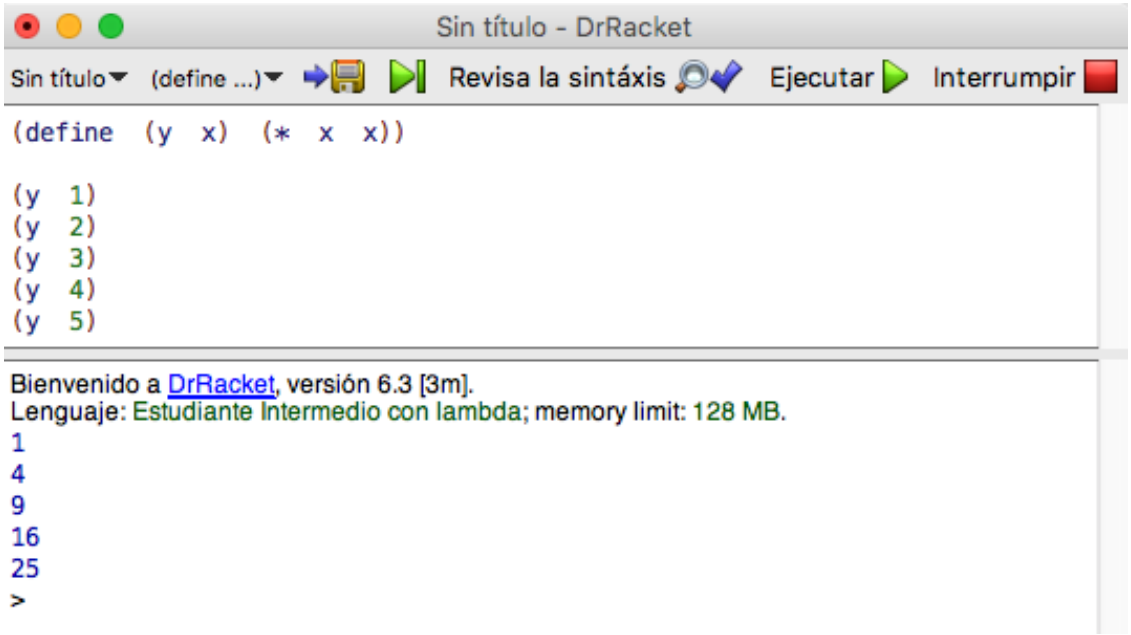
Si escribimos *(y 1)* en el indicador de la zona interacciones veremos un 1 como respuesta. El *(y 1)* es una aplicación de función en DrRacket.



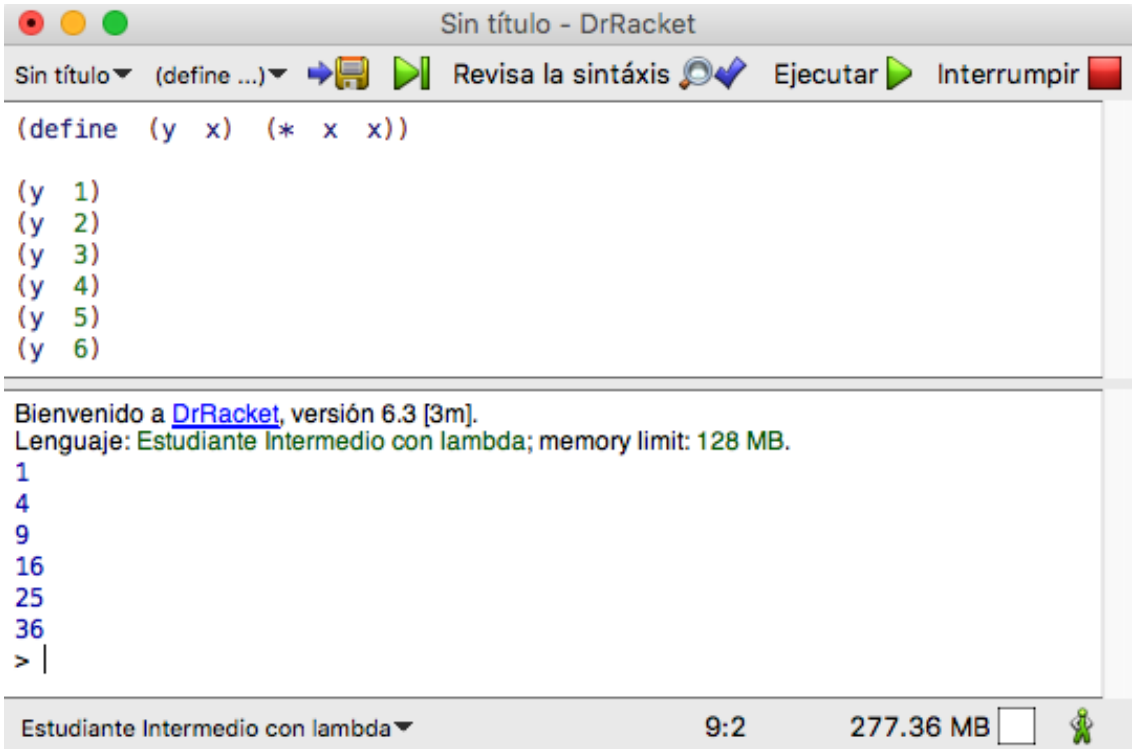
Pruebe *(y 2)* y verá un 4 en la zona de interacciones.



Por supuesto, también puede introducir todas estas expresiones en el área de las definiciones y haga clic en ejecutar:



En respuesta, DrRacket muestra: 1 4 9 16 25, que son los números de la tabla. Ahora determine la entrada que falta.



Agregamos *(y 6)* para poder obtener el resultados que nos hace falta y DrRacket nos muestra 36 como resultado.

Lo importante de todo esto es que las funciones proporcionan una forma más económica de calcular un montón de valores con una sola expresión. De hecho, los programas son funciones, y una vez que se entienden bien las funciones, se puede saber casi todo lo que hay que saber para tener un buen desarrollo de programación.

Dada su importancia, vamos a recapitular lo que sabemos acerca de las funciones hasta el momento:

En primer lugar

*(**define** (nombre de función variable) cuerpo de la expresión)*

Es una definición de la función. Se reconoce como tal, ya que se inicia con el "**define** palabra reservada". Una definición (de función) consta de tres partes: dos nombres y una expresión. El primer nombre es el nombre de la función. El segundo nombre representa la entrada de la función, que se conoce hasta que se aplique la función.

La expresión, denominada cuerpo calcula la salida de la función para una entrada específica. Como se ve, la expresión implica el parámetro, y también puede consistir en muchas otras expresiones.

Segundo,

(nombrefunción argumentoexpresión)

Es una aplicación de función. La primera parte le dice a DrRacket cuál es la función que se desea utilizar. La segunda parte es la acción que se desea realizar para obtener resultados de la función.

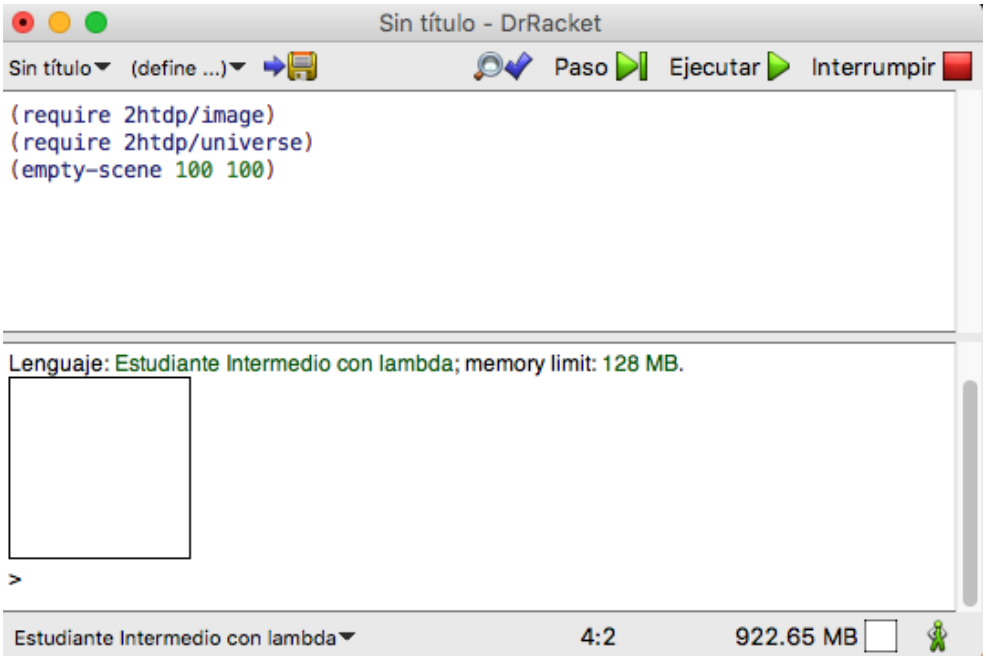
A las funciones se les puede ingresar más de un número, y puede dar salida a todo tipo de datos. Nuestra próxima tarea es crear una función que simula la segunda tabla la que tiene imágenes de un punto rojo; Al igual que la primera función simula la tabla de números que ya observamos.

Empecemos con la sentencia

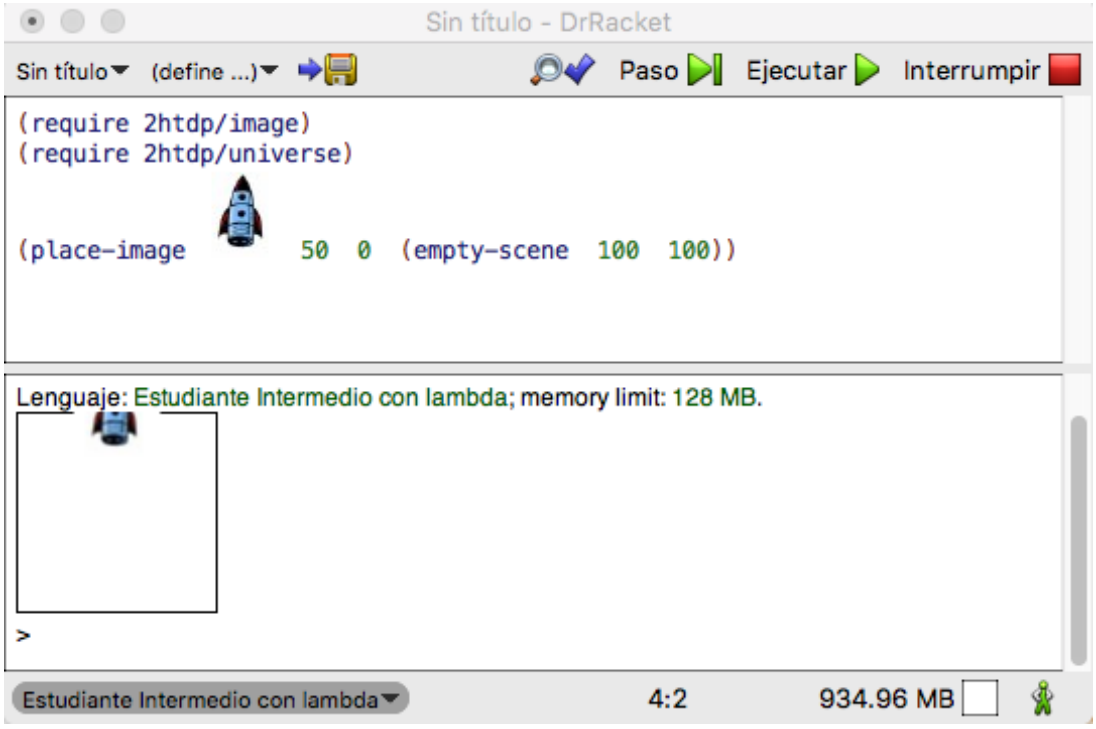
(empty-scene 100 100)[18]

Ejecutamos y observamos que en el área de interacción aparece un cuadro vacío.

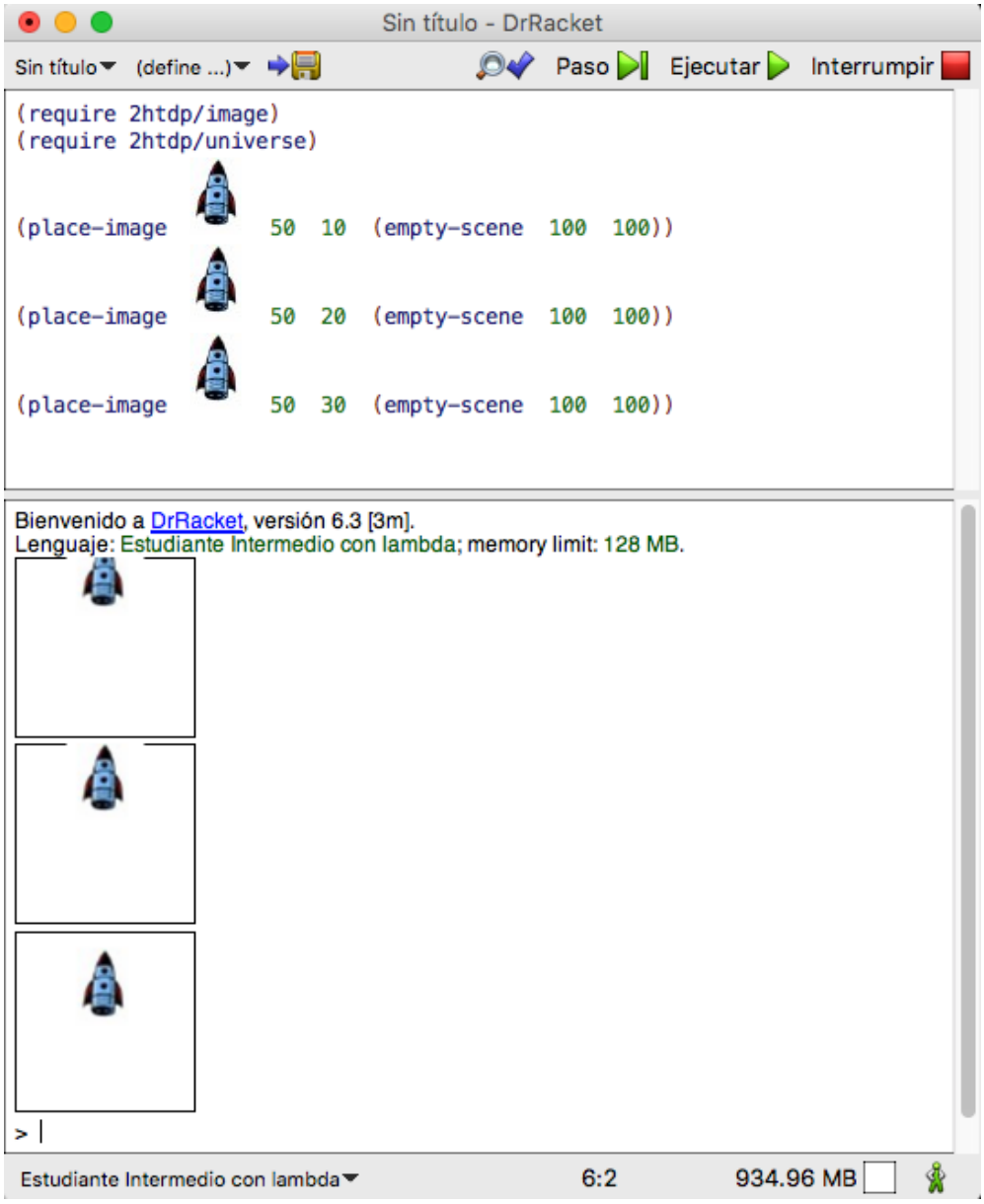
[18]para que esta sentencia se ejecute sin problemas se requiere agregar un par de sentencias para el manejo de imágenes y universos



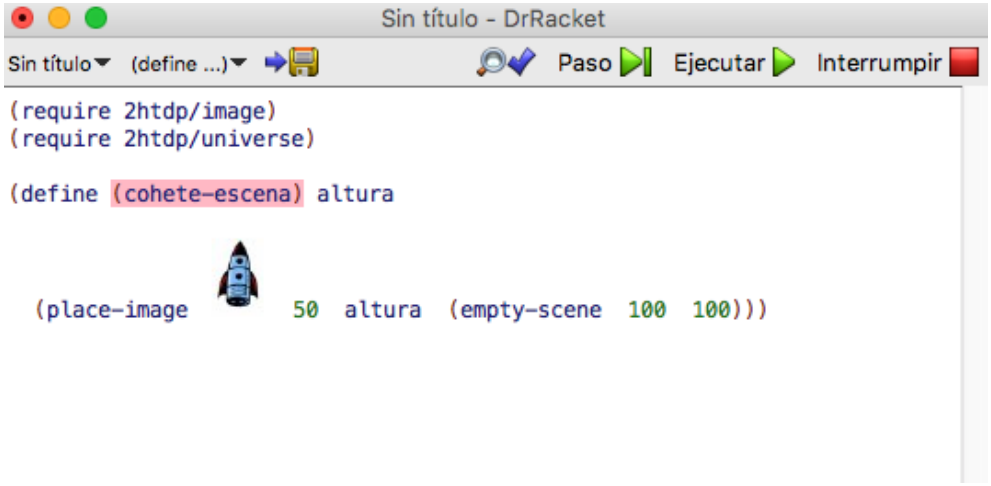
Con la sentencia **place-image** podemos agregar alguna imagen a la escena de esta manera;



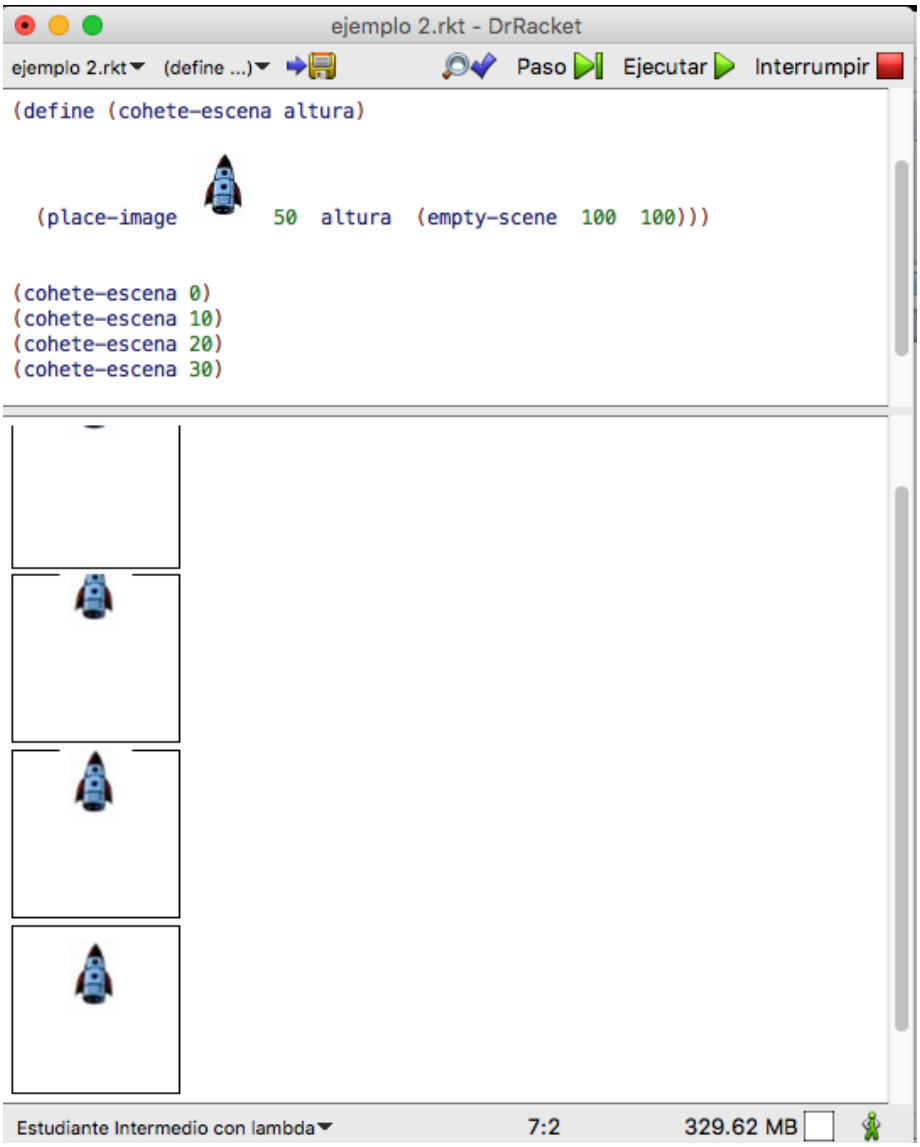
Ya que tenemos dominado esto, ahora debemos simular el descenso del cohete emplearemos la misma sentencia anterior solo variaremos la posición de la altura en la que se muestra la imagen.



Ahora que sabemos cómo funcionan las sentencias y las posiciones de las imágenes podremos construir una función para poder mostrar más de unas cuantas escenas sin tener que escribirlas de una por una.

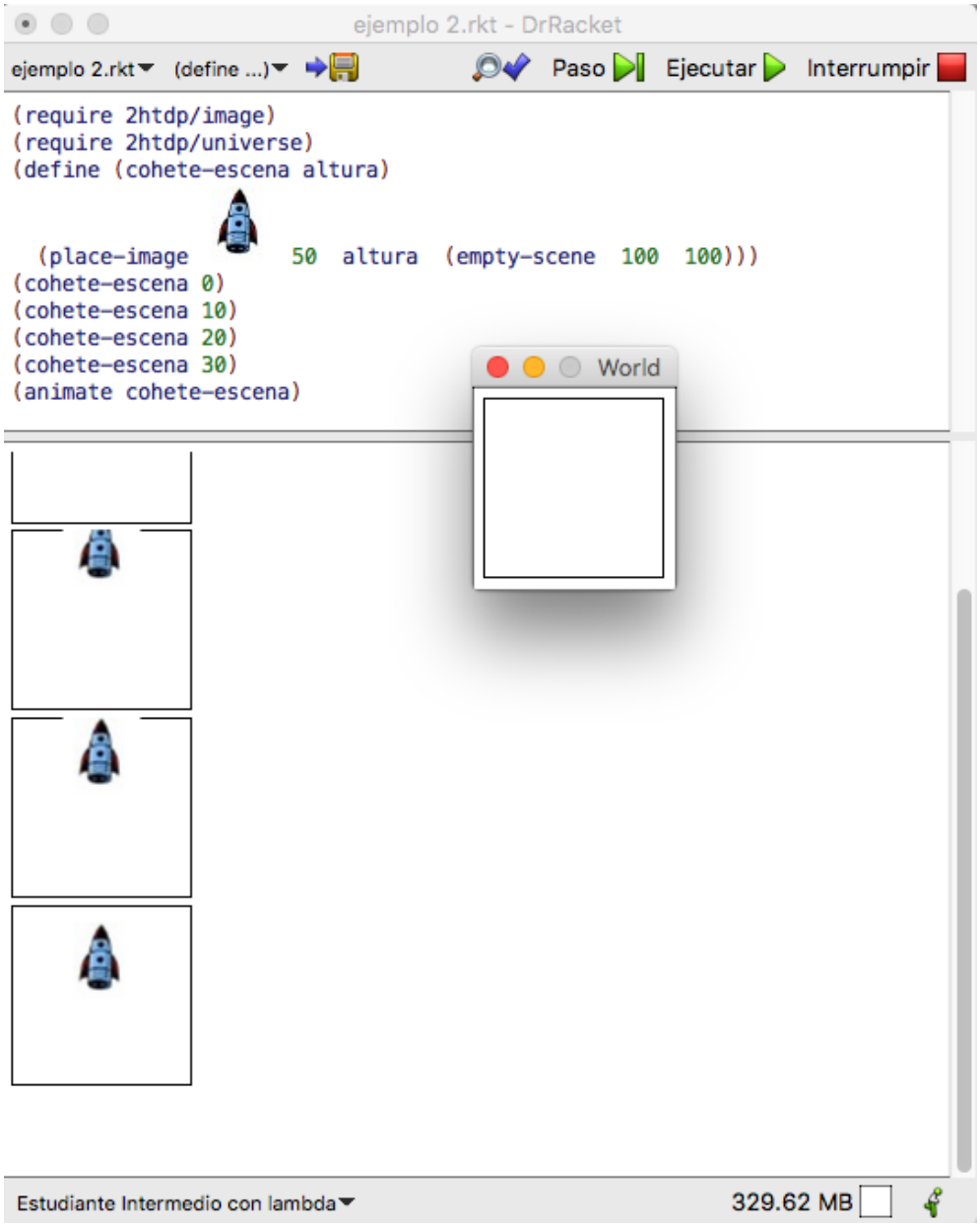


En esta definición de función sustituimos a *y* por cohete-escena un nombre que nos ayuda a identificar el objetivo de la función, en lugar de *x* empleamos altura porque es la variable que modificaremos por lo tanto es nuestra entrada, y el cuerpo de la función opera como la anterior solo que ahora se exhibe una escena como resultado.



Hasta este punto podemos decir que las funciones en DrRacket no solo las podemos trabajar con números como entrada y como salida, sino que también podemos interactuar con muchos más tipos de datos lo que lo convierte en una herramienta más dinámica y llamativa con la cual los alumnos pueden aprender conceptos nuevos o que se les complican, y aún así, DrRacket nos permite tener una experiencia más dinámica.

Ahora emplearemos la sentencia (**animate** *cohete-escena*) lo que hará que al ejecutar el programa se despliegue una ventana emergente en la que se exhibirá una animación con los parámetros que le dimos a la función.



Esta animación se crea por que al momento en que ejecutamos el programa, comienza un reloj con un conteo de 28 *ticks*[19] por segundo. Cada vez que el reloj avanza, DrRacket aplica cohete-escena con el número de *ticks* que han pasado desde la llamada de función. Los resultados de estas llamadas a función se muestran en el lienzo, y se produce el efecto de una película de animación.

Con este ejemplo queda demostrado que sí se puede emplear DrRacket como una herramienta importante en la cual tanto alumnos como profesores pueden apoyarse para lograr superar las deficiencias que tienen y poder dar el paso de variables a funciones.

Ahora profundizaremos en la estructura de los programas para que los desarrollos sean más eficientes, y un mejor apoyo de aprendizaje, para que no solo sirvan de cimiento en la enseñanza de las matemáticas, sino también para la programación y ayudar a mejorar las técnicas de aprendizaje.

[19] Krishnamurthi, M. F. (s.f.). HTDP second edition. Recuperado el 12 de 12 de 2015, de Inputs and Output: http://www.ccs.neu.edu/home/matthias/HtDP2e/part_prologue.html#%28part._some-i%2Fo%29

7 Diseño de recetas (Gregor Kiczales[20])

En este curso, se enseña un enfoque para el diseño de programas basado en recetas de diseño. Cada receta es aplicable a ciertos problemas, y sistematiza el proceso de diseñar soluciones a esos problemas.

Hay recetas básicas que se utilizan con más frecuencia. Las recetas de formatos se utilizan como parte del diseño de cada definición y función

[20]Kiczales, G. (s.f.). *Systematic Program Design*. Recuperado el 21 de 12 de 2015, de edx: https://courses.edx.org/courses/course-v1:UBCx+SPD1x+1T2016/77860a93562d40bda45e452ea064998b/#FuncComp

de datos. Las recetas de Abstracción se utilizan para reducir la redundancia en el código.

7.1 Cómo diseñar funciones (HtDP)[21]

El Cómo diseñar Funciones (HtDP) es un método de diseño que permite el diseño sistemático de funciones.

La receta HtDP consta de los siguientes pasos:

1. Firma, el propósito y borrador.
2. Definir ejemplos.
3. Formato e inventario.
4. Codificar el cuerpo de la función.
5. Probar y depurar hasta corregir.

7.1.1 Firma, el propósito y borrador.

Una firma tiene el tipo de cada argumento, separados por espacios, seguido por ->, seguido por el tipo de dato que se espera obtener. Así que una función que consume una imagen y produce una serie tendría la firma imagen -> número.

---

[21] Kiczales, G. (s.f.). *Systematic Program Design*. Recuperado el 21 de 12 de 2015, de edx: https://courses.edx.org/courses/course-v1:UBCx+SPD1x+1T2016/77860a93562d40bda45e452ea064998b/#FuncComp

Tenga en cuenta que el borrador es una definición de función sintáctica completa que produce un valor del tipo correcto. Si el tipo es número que es común el uso de 0, si el tipo es *string* es común el uso de "*a*" y así sucesivamente. En el siguiente ejemplo el borrador produce 0, que es un número, pero sólo coincide con el propósito cuando *double*[22] pasa a llamarse con 0.

;; Número -> Número
;; Produce n veces 2
*(**define** (double n) 0); este es el borrador*

El propósito del borrador es servir como una especie de andamio para que sea posible ejecutar los ejemplos incluso antes de que el diseño de la función este completo.

7.1.2 Definir ejemplos.

Escribe por lo menos un ejemplo de una llamada a la función y el resultado esperado que debe producir.

A menudo se necesitan más ejemplos, para ayudar a comprender mejor la función o para probar la función. Si no se está seguro de cómo

[22] Benson, B. (s.f.). Racket. Recuperado el 19 de 01 de 2016, de Sección 8 Intermezzo 1: Sintaxis y semántica: http://docs.racket-lang.org/htdp-langs/beginner.html

empezar los ejemplos, se utilizan la combinación de la firma de la función y de la definición de datos para ayudar a generar ejemplos. Frecuentemente el ejemplo de la definición de datos es útil, pero no cubre necesariamente todos los casos importantes para una función determinada.

El primer papel de un ejemplo es para ayudar a entender lo que la función se supone que debe hacer. Si hay condiciones de contorno asegúrese de incluir un ejemplo de cada uno de ellos. Si hay diferentes comportamientos que la función debe tener, incluya un ejemplo de cada uno. Puesto que son ejemplos, primero se podría escribir en este formulario:

```
;; (double 0) debe producir 0
;; (double 1) debe producir 2
;; (double 2) debe producir 4
```

Al escribir ejemplos a veces es útil escribir no sólo el resultado esperado, sino también la forma en que se calcula. Por ejemplo, se puede escribir lo siguiente en lugar de lo anterior:

```
;; (double 0) debe producir (* 0 2)
;; (double 1) debe producir (* 1 2)
;; (double 2) debe producir (* 2 2)
```

Si bien el formulario de arriba satisface nuestra necesidad de ejemplos, DrRacket nos da una mejor manera de escribirlos, encerrándolos en el registro de entrada a esperar. Esto permitirá a DrRacket comprobar de forma automática cuando la función se ha completado. (En términos técnicos se convertirá los ejemplos en las pruebas unitarias.)

```
;; Número -> Número
;; produce n veces 2
(check-expect (double 0) (0 * 2))
(check-expect (double 1) (* 1 2))
(check-expect (double 3) (* 3 2))
(define (double n) 0); este es el borrador
```

La existencia del borrador le permitirá ejecutar las pruebas.

7.1.3 Formatos e inventario

Antes de codificar el cuerpo de la función es útil para tener una idea clara de lo que la función tiene que trabajar, el formato proporciona esto.

Los formatos se fabrican siguiendo las normas de Data Driven Templates[23]. Solo se debe copiar o partir de la definición de datos, para el diseño de la función, cambiar el nombre del formato, y escribir un comentario en el que se indique que el formato fue copiado. Se tiene que considerar que el formato que se copia se elige de acuerdo al tipo de datos de entrada, no del tipo de dato producido.

Para los datos primitivos como los números, las cadenas y las imágenes el formato es simplemente (... x) donde x es el nombre del parámetro a la función.

```
Número -> Número
produce n veces 2
(check-expect (double 0) (0 * 2))
(check-expect (double 1) (* 1 2))
(check-expect (double 3) (* 3 2))
(define (double n) 0); este es el borrador
(define (double n), esto es el formato  (... n))
```

[23] Kiczales, G. (s.f.). Systematic Program Design. Recuperado el 21 de 12 de 2015, de edx: https://courses.edx.org/courses/course-v1:UBCx+SPD1x+1T2016/77860a93562d40bda45e452ea064998b/#FuncComp

7.1.4 Codificar el cuerpo de la función

Ahora completar el cuerpo de la función mediante el cumplimiento del formato.

Tenga en cuenta que:

- la firma describe el tipo de parámetros y el tipo de los datos que la función debe producir.
- el propósito describe lo que el cuerpo de la función debe producir.
- los ejemplos proporcionan varios ejemplos concretos de lo que la función debe producir
- el formato nos indica el tipo de datos con los que operará la función

Debemos usar todo lo anterior para ayudarnos a codificar la función. En algunos casos aún más la reescritura de ejemplos podría hacerlo más claro cómo se ejecutan ciertos valores, y qué puede hacer para que sea más fácil de codificar la función.

```
;; Número -> Número
;; produce n veces 2
(check-expect (doble 0) (0 * 2))
(check-expect (doble 1) (* 1 2))
(check-expect (doble 3) (* 3 2))
; (define (doble n) 0); este es el borrador
; (define (doble n), lo que es el formato (... n))
(define (doble n)
(2 * n))
```

7.1.5 Probar y depurar hasta corregir

Ejecutar el programa y asegurar de que todas las pruebas pasan, si no depurar hasta que lo hagan.

7.2 Cómo diseñar datos (HtDD)[24]

Las definiciones de datos son un elemento de la conducción en las recetas de diseño.

Una definición de datos establece la relación entre la información y los datos:

---

[24] Kiczales, G. (s.f.). *Systematic Program Design*. Recuperado el 21 de 12 de 2015, de edx: https://courses.edx.org/courses/course-v1:UBCx+SPD1x+1T2016/77860a93562d40bda45e452ea064998b/#FuncComp

- La información contenida en el dominio del programa está representado por los datos en el programa.
- Los datos en el programa se pueden interpretar como la información en el dominio del programa.

Una definición de datos debe describir cómo formar (o hacer) los datos que satisfaga la definición de datos y también la forma de saber si un valor de datos satisface la definición de datos. También debe describir cómo representar la información en el dominio del programa como los datos e interpretar un valor de datos como información.

Así, por ejemplo, una definición de datos podría decir que los números se utilizan para representar la velocidad de una pelota. Otra definición de datos podría decir que los números se utilizan para representar la altura de un avión. Así que da igual si declaramos un número 6, necesitamos una definición de los datos que nos diga cómo interpretarlo: ¿es una velocidad, o una altura o algo completamente distinto? Sin una definición de datos, el 6 podría significar cualquier cosa.

El primer paso de la receta es identificar la estructura inherente de la información.

7.2.1 ¿Cuál es la estructura inherente de la información?

Uno de los puntos más importantes en el curso es que:

- la estructura de la información en el dominio del programa determina el tipo de definición de datos utilizado,
- que a su vez determina la estructura de los formatos y ayuda a determinar los ejemplos de la función *(check-expect s),*
- y por lo tanto la estructura de la gran parte del diseño final del programa.

En esta tabla[25] se resume y proporciona una rápida referencia al tipo de definición de datos a utilizar para diferentes estructuras de información.

Cuando la forma de la información a ser representado ...	**Utilice una definición de datos de este tipo**
es atómica	Datos Atómica simple
es un número dentro de un cierto rango	Intervalo
se compone de un número fijo de elementos distintos	Enumeración
está compuesta por 2 o más subclases, por lo menos uno de los cuales no es un elemento distinto	Desglose
consta de dos o más elementos que, naturalmente, van de la mano	Datos Compuesto
es natural compuesto por diferentes partes	Las referencias a otro tipo definido
es de tamaño arbitrario (desconocido)	autorreferencial o mutuamente referencial

[25] Kiczales, G. (s.f.). Systematic Program Design. Recuperado el 21 de 12 de 2015, de edx: https://courses.edx.org/courses/course-v1:UBCx+SPD1x+1T2016/77860a93562d40bda45e452ea064998b/#FuncComp

8 Conclusiones

Al analizar algunas deficiencias de alumnos de primer ingreso; algunas veces requiere, por ejemplo, que si se desea que manejen exponentes fraccionarios, primero habría que lograr que manejaran fracciones, algo que se esperaría hubieran visto desde la primaria.

De forma similar, pero a un nivel superior se esperaría que si manejan conceptos aritméticos hubieran podido abstraerlos y podrían manejar conceptos algebraicos, en particular los relacionados con el concepto de función.

Abstraer operaciones aritméticas en operaciones algebraicas parece trivial, pero no lo es. De hecho es lo que permite manejar definiciones de funciones o sea programas. Vincular la aritmética al álgebra y ésta a sus procesos de abstracción es importante para el aprendizaje de matemáticas.

Es por eso que el objetivo de este proyecto es presentar DrRacket como una herramienta de apoyo que junto con las recetas de diseño, facilite a los estudiantes la abstracción de conceptos aritméticos para que con ellos consigan un mejor aprendizaje del álgebra.

Se pretende facilitar esto con el empleo de algunas cualidades de DrRacket como son la capacidad de generar videos y manipular imágenes para que los resultados de las funciones y/o los procesos sean más visibles, además de la opción de paso a paso que muestra punto a punto como el programa trabaja con los datos de entrada para obtener los resultados, lo que ayuda a los alumnos a analizar cómo trabajar y manipular las información que reciben para lograr obtener resultados deseables.

Es por eso que se considera que el uso de esa herramienta en conjunto con sus cualidades y las recetas de diseño, podría aumentar el promedio de los estudiantes así como sus capacidades y métodos de estudio.

Bibliografía

Benson, B. (s.f.). *Racket*. Recuperado el 19 de 01 de 2016, de Sección 8 Intermezzo 1: Sintaxis y semántica: http://docs.racket-lang.org/htdp-langs/beginner.html

Ingeniería, F. d. (09 de 12 de 2015). *UNAM Facultad de Ingeniería*. Recuperado el 09 de 12 de 2015, de Bitacora FI: http://www.ingenieria.unam.mx/~bitacoraFI/bitacora/

Kiczales, G. (s.f.). *Systematic Program Design*. Recuperado el 21 de 12 de 2015, de edx: https://courses.edx.org/courses/course-v1:UBCx+SPD1x+1T2016/77860a93562d40bda45e452ea064998b/#FuncComp

Krishnamurthi, M. F. (s.f.). *Fixed-Size Data*. Recuperado el 23 de 12 de 2015, de HTDP second edition: http://www.ccs.neu.edu/home/matthias/HtDP2e/part_one.html

Krishnamurthi, M. F. (s.f.). *HTDP second edition*. Recuperado el 12 de 12 de 2015, de Inputs and Output: http://www.ccs.neu.edu/home/matthias/HtDP2e/part_prologue.html#%28part._some-i%2Fo%29

Krishnamurthi, M. F. (s.f.). *Intermezzo: BSL*. Recuperado el 27 de 12 de 2015, de HTDP second edition : http://www.ccs.neu.edu/home/matthias/HtDP2e/i1-2.html

Krishnamurthi, M. F. (s.f.). *Understanding the Program's Purpose:*. Recuperado el 11 de 12 de 2015, de HTDP: http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-5.html#node_sec_2.5

Resources, A. M.-S. (2001). *metodos de estudio*. Recuperado el 09 de 12 de 2015, de how to study.com: http://www.how-to-study.com/metodos-de-estudio/lectura-critica.asp

wikipedia Notación de Backus-Naur. Recuperado el 22 de 01 de 2016, de Notación de Backus-Naur : https://es.wikipedia.org/wiki/Notaci%C3%B3n_de_Backus-Naur