UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Manual de prácticas para la asignatura de Lenguajes de Programación

REPORTE DE ACTIVIDAD DOCENTE

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Ciencias de la Computación

PRESENTA:

Rodrigo Ruiz Murguía

TUTORA

M. en A. Karla Ramírez Pulido

2016

1. Datos del alumno
Ruiz
Murguía
Rodrigo
55 43 57 27 25

2. Datos del tutor
M en A
Karla
Ramírez
Pulido

3. Datos del sinodal 1
Dr
Flavio Ezequiel
Miranda
Perea

4. Datos del sinodal 2
M en C
Araceli Liliana
Reyes
Cabello

5. Datos del sinodal 3
Lic en CC
Pilar Selene
Linares
Arévalo

6. Datos del sinodal 4
Dr
María de Luz
Gasta
Soto

7. Datos del trabajo escrito
Manual de prácticas para la asignatura de Lenguajes de Programación
Diseñando intérpretes
117 p
2016

Capítulo 1
Introducción 1

1.1 Objetivos de la materia 2

1.2 Plan de trabajo 3

1.3 Objetivos del trabajo 3

1.4 Estructura del trabajo 3

Capítulo 2
Primeros pasos en DrRacket y Racket 5

2.1 Objetivos 5

2.2 Características de Racket y DrRacket 5

2.3 Contenido de la práctica 6

2.4 Desarrollando un primer programa en Racket 8

2.5 Procedimientos importantes para esta práctica 9

2.6 Ejercicios 13

2.7 Sitios importantes relacionados con el lenguaje 17

Capítulo 3
Tipos de datos abstractos 19

3.1 Objetivos 19

3.2 Predicados 19

3.3 Dialecto PLAI 21

3.4 Ejercicios 25

Capítulo 4
Análisis sintáctico e interpretación 31

4.1 Objetivo 31

4.2 Análisis sintáctico 31

4.3 Construyendo un primer intérprete 32

4.4 Ejercicios 39

Capítulo 5
Evaluación perezosa 43

5.1 Objetivo 43

5.2 Definición 43

5.3 Consideraciones para implementar evaluación perezosa 43

5.4 Ejercicios 50

Capítulo 6
Estado 55

6.1 Objetivos 55

6.2 Mutación y cajas 55

6.3 Técnica de paso de repositorio de valores (store passing-style) 56

6.4 Ejemplos 57

6.5 Ejercicios 61

Capítulo 7
Recursión 63

7.1 Objetivo 63

7.2 Tipos de recursión 63

7.3 Funciones recursivas 65

7.4 Ejercicios 69

Capítulo 8
Sistema verificador de tipos 71

8.1 Objetivo 71

8.2 Definición de tipo y sus características 71

8.3 Sistema verificador de tipos 72

8.4 ¿Por qué tipos? 72

8.5 Ejercicios 73

Capítulo 9
Conceptos de Orientación a Objetos 77

9.1 Objetivo 77

9.2 Conceptos de orientación a objetos 77

9.3 Sintaxis concreta 78

9.4 Semántica del intérprete 81

Capítulo 10
Subtipado 85

10.1 Objetivo 85

10.2 Motivación 85

10.3 Relación entre subtipado y herencia 87

10.4 Añadiendo el concepto de subtipado al intérprete de orientación a objetos 89

10.5 Sintaxis concreta 93

10.6 Semántica del intérprete 94

Conclusiones 97

Referencias bibliográficas 99

Referencias en Internet 100

Apéndices 101

Apéndice A
Instalación del intérprete Haskell 101

Apéndice B
Analizador sintáctico para las gramáticas de orientación a objetos 102

Apéndice C
Notación alterna para sección 2.5 111

Índice de códigos

Código 2.1 Primer programa en Racket 8
Código 3.1 Ejemplo de uso del procedimiento and 21
Código 3.2 Ejemplo de tipo de datos lista 21
Código 3.3 Ejemplos de procedimientos definidos por PLAI 23
Código 3.4. Ejemplo de tipos de datos 24
Código 3.5 Definición tipo conjunto 26
Código 3.6 Definición tipo arbolb 28
Código 4.1 Gramática AE en EBNF 32
Código 4.2 Definición PLAI de gramática AE 33
Código 4.3 Procedimiento parse para gramática AE 34
Código 4.4 Procedimiento interp para gramática AE 35
Código 4.5 diferentes ejemplos with 36
Código 4.6 Ejemplo de cerradura 39
Código 4.7 Gramática extendida de CFWAE 40
Código 5.1 Ejemplo de evaluación perezosa y cerraduras de expresiones 44
Código 5.2 Ejemplo de Racket perezoso 52
Código 5.3 Gramática CFWAE perezosa 54
Código 6.1 Ejemplo de las primitivas box, unbox y set-box! 56
Código 6.2 Gramática extendida BCFWAE 57
Código 6.3 Ejemplo BCFWAE 58
Código 6.4 Conversión del Código 6.3 a sintaxis de Racket 61
Código 7.1 Ejemplo de referencia a sí mismo 64
Código 7.2 Ejemplo de estrategia de tres pasos 64

Código 7.3 Ejemplo de la función factorial sin la estrategia de tres pasos. 65

Código 7.4 Ejemplo de factorial usando la gramática BCFWAE 66

Código 7.5 Código de la aplicación de función BCFWAE (Código 6.2) 66

Código 7.6 Ejemplo de factorial implementado con la estrategia de tres pasos 69

Código 8.1 Sintaxis de la gramática CFWAEL 74

Código 9.1 Gramática BL 79

Código 9.2 Gramática ICL 79

Código 9.3 Gramática CL 79

Código 9.4 Gramática CUL 80

Código 9.5 Gramática EL 80

Código 9.6 Ejemplos gramática CUL 81

Código 9.7 Ejemplos de funciones hash 82

Código 9.8 Ejemplo de importación de funciones de otros archivos 83

Código 10.1 Ejemplo de subtipado de la clase persona% 85

Código 10.2 Ejemplo de subtipado de la clase mascota% 86

Código 10.3 Ejemplo de la función get-edad-en-dias 86

Código 10.4 Estructura árbol binario para subtipado 91

Código 10.5 Ejemplo para estructura de subtipado 93

Código 10.6 Gramática LL 93

Código 10.7 Gramática EL 94

Código 10.8 Ejemplos de la gramática LL 95

Código A. Analizador sintáctico para las gramáticas de orientación a objetos 110

Índice de figuras

Figura 1. Ventana principal del intérprete 7

Figura 2. Ejemplo simple de jerarquía de clases 89

Figura 3. Ejemplo de jerarquía de clases 92

Capítulo 1 Introducción

En este trabajo se plantean una serie de prácticas diseñadas para la asignatura de Lenguajes de Programación de la licenciatura en Ciencias de la Computación, en la Facultad de Ciencias de la UNAM, las cuales permiten a los estudiantes analizar y reforzar, desde un punto de vista teórico y práctico, los temas vistos en dicho curso.

Para comprender los temas de una manera práctica, se hace uso del lenguaje de programación *Racket,* ya que es un lenguaje de espectro amplio, descendiente de `Scheme`, que permite programar a un nivel alto. Las excelentes herramientas de abstracción que ofrece, junto a la sintaxis, semántica asociada, estructuras de datos y herramientas de representación, permitirán al alumno desarrollar las prácticas de manera natural, lo cual hará que el proceso de creación e implementación de un intérprete sea directo, conciso y comprensible. Con ello, el estudiante será capaz de escribir sistemas robustos para el proceso de aprendizaje de lenguajes, los cuales son lo suficientemente compactos para que los estudiantes puedan entenderlos y manipularlos con un esfuerzo razonable.

La finalidad de las primeras prácticas es introducir al estudiante en el lenguaje, tanto en su sintaxis como en las funcionalidades del mismo, para la representación de gramáticas y su interpretación. Al ser una asignatura obligatoria de la licenciatura de Ciencias de la Computación de la UNAM, es indispensable exponer a los estudiantes todos los detalles de los lenguajes de programación a profundidad. Llevar a cabo el resto de las prácticas permitirá al estudiante analizar los lenguajes de programación desde el enfoque de quien diseña e implementa los mismos, enfrentando así al alumno a problemas similares o iguales a los que enfrentan las personas dedicadas a estos temas de estudio. El conjunto de prácticas planteadas serán una guía sobre la cual los estudiantes podrán basarse para solucionar dichos problemas.

Los temas que cubre este trabajo son:

- Modelado de lenguajes.
- Substitución.
- Creación de funciones.

- Substitución diferida.
- Funciones de primera clase.
- Tipos de evaluación.
- Recursión.
- Estado.
- Tipos.
- Orientación a objetos.
- Manejo de memoria.

1.1 Objetivos de la materia

El plan de estudios de la asignatura define los siguientes objetivos, [19].

- Retomar y profundizar la programación recursiva.
- Identificar los puntos más relevantes para representar gramáticas en el lenguaje.
- Conocer el funcionamiento de los analizadores sintácticos para que el estudiante sea capaz de programarlos de forma robusta y eficiente.
- Estudiar el funcionamiento de los intérpretes para que el alumno comprenda el proceso de evaluación de expresiones.
- Analizar los distintos tipos de paso de parámetros.
- Comprender las diferencias entre los diferentes tipos de alcance existentes (estático y dinámico).
- Analizar las semánticas de evaluación que existen y estudiar cómo repercuten en el lenguaje.
- Analizar las implicaciones de que un lenguaje maneje o no estado.
- Entender cómo es que un lenguaje implementa la recursión.
- Conocer las técnicas que se usan en el manejo de memoria.
- Analizar los tipos básicos de un lenguaje, así como el sistema de verificador de tipos.

1.2 Plan de trabajo

Durante el curso los estudiantes formarán equipos de dos a tres personas para resolver las prácticas. Éstas se encuentran diseñadas para que puedan ser resueltas en un periodo de dos semanas aproximadamente cada una. Las primeras prácticas tienen como objetivo introducir el lenguaje de programación que se usará en la asignatura y por ende se necesita que en las primeras sesiones se le enseñe al estudiante la sintaxis y semántica del lenguaje, así como la forma en que se deben enfocar la resolución problemas haciendo uso del mismo.

Una vez pasada esta fase introductoria, en las siguientes prácticas los estudiantes reforzarán de una manera práctica los temas que el profesor haya visto de manera teórica durante las clases. Este reporte contiene algunos de los fundamentos teóricos básicos que se necesitan en el curso, así como las prácticas propuestas, para cubrir los objetivos de cada tema.

1.3 Objetivos del trabajo

Este trabajo tiene los siguientes objetivos:

- Introducir y hacer que el estudiante se sienta cómodo programando tanto en el lenguaje Racket como en su ambiente integrado de desarrollo DrRacket.
- Enseñar al estudiante cómo construir los tipos de datos que representarán a la gramática que se quiere implementar.
- Entender la forma en que se debe construir un intérprete, desde analizar la sintaxis de entrada hasta regresar el resultado de la evaluación de cada expresión pasada al intérprete.
- Analizar las diferencias entre distintos tipos de intérpretes, por ejemplo con la misma gramática pero con semánticas de evaluación diferentes.

1.4 Estructura del trabajo

El reporte está compuesto por una serie de prácticas, las cuales presentarán objetivos particulares y ejercicios correspondientes a cada tema, con el objetivo de que el estudiante sea capaz de implementarlos una vez que se haya visto el tema en clase. De esta forma se presentará una pequeña parte teórica que contará con ejemplos relacionados con el tema, los cuales ayudarán a comprender los ejercicios propuestos y así dar posibles soluciones a éstos. Por otro lado, contendrá una parte práctica la cual permitirá al alumno reforzar los conceptos teóricos mediante la implementación de una gramática que haga uso de los mismos.

Capítulo 2 Primeros pasos en DrRacket y Racket

2.1 Objetivos

- Familiarizar al estudiante tanto con el lenguaje de programación, Racket, como con el intérprete, llamado DrRacket, los cuales se usarán a lo largo del curso.
- Instalar el ambiente de desarrollo integrado DrRacket.
- Escribir y ejecutar programas sencillos en el lenguaje Racket, haciendo uso de los procedimientos base, los cuales que serán necesarios para las siguientes prácticas.

2.2 Características de Racket y DrRacket

Racket (anteriormente llamado PLT Scheme) es un lenguaje de programación multiparadigma perteneciente a la familia de Lisp/Scheme, que también sirve como una plataforma para la creación, diseño e implementación de un lenguaje de programación. La forma en que se usará este lenguaje es a través del ambiente integrado de desarrollo DrRacket, el cual es a su vez un intérprete.

"Un intérprete es un programa capaz de analizar y ejecutar otros programas, escritos en un lenguaje de alto nivel[1]. Los intérpretes se diferencian de los compiladores en que mientras éstos traducen un programa desde su descripción en un lenguaje de programación al código de máquina del sistema, los intérpretes sólo realizan la traducción a medida que sea necesaria, típicamente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción"[6].

[1] *"Un lenguaje de programación de alto nivel se caracteriza por expresar los algoritmos de una manera adecuada a la capacidad cognitiva humana, en lugar de a la capacidad ejecutora de las máquinas"*[6].

Al ser un lenguaje interpretado, se obtienen las siguientes ventajas:

- Un solo archivo fuente puede producir resultados iguales en sistemas sumamente diferentes.
- Los programas son más flexibles como entornos de programación y depuración.
- Ofrece a los programas un entorno no dependiente de la máquina donde se ejecuta el intérprete, sino del propio intérprete.

2.3 Contenido de la práctica

Instalando el intérprete

DrRacket[2] se encuentra disponible para los tres sistemas operativos más comunes del mundo, Mac OS®, Windows® y Linux, [20].

Para instalar el intérprete y empezar a utilizarlo es necesario seguir los siguientes pasos:

Mac OS®: Descargar el archivo con extensión .dmg, abrir y arrastrar la carpeta "Racket v6.0" a la carpeta de aplicaciones. Para ejecutar el intérprete, abrir DrRacket ubicado dentro de dicha carpeta.

Windows®: Descargar el archivo con extensión .exe y ejecutarlo. Para los propósitos de este trabajo, bastan las opciones preestablecidas. Para ejecutar el intérprete, será necesario buscar el archivo ejecutable en el menú de inicio

Linux: Descargar el archivo que corresponda a la distribución utilizada en la computadora en la que se desea instalar[3]. Abrir una terminal y moverse hasta la ubicación del archivo y teclear lo siguiente:

```
$ sudo sh <nombre_del_archivo_descargado>
```

A manera de ejemplo, si se descargó la versión para Ubuntu:

```
$ sudo sh racket-6.3-x86_64-linux-ubuntu-precise.sh
```

El programa que instala ofrecerá después algunas opciones de configuración, para los propósitos de este trabajo basta con seleccionar las opciones preestablecidas.

[2] Al momento de escribir este trabajo, la versión más actual es la número 6.3

[3] Si no se encuentra la distribución en específico, seleccionar la "más cercana", por ejemplo, si se desea instalar en el sistema operativo Mint, se puede descargar la versión para la distribución Ubuntu.

Una vez instalado el intérprete, ejecutar el siguiente comando para abrirlo:

```
$ drracket
```

Ejecutando *DrRacket* por primera vez

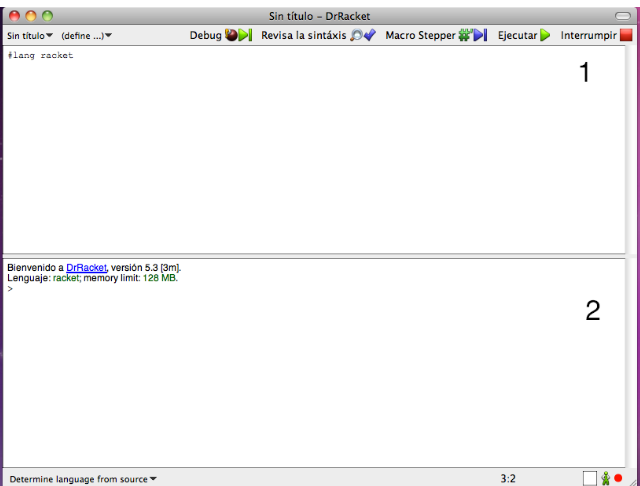


Figura 1. Ventana principal del intérprete

En la Figura 1 se muestra la interfaz gráfica de DrRacket, la cual se divide en dos paneles el superior(1) y el inferior(2). Al panel superior se le conoce como "Definiciones" o "Zona de definiciones" mientras que al inferior se le conoce como "Interacciones" o "Zona de interacción" las cuales se explican a continuación:

La **zona de definiciones** contendrá el código del programa en sí, los procedimientos, variables y constantes que serán necesarias para el funcionamiento del programa. El código que se escribe en esta zona quedará al alcance de aquél usado en la zona de interacciones. Por lo general, se coloca el código que será necesario para el programa y no así el punto de entrada al mismo.

La **zona de interacciones** es donde se prueba o ejecuta todo el código que se encuentre en la zona de definiciones. Es de especial utilidad, ya que nos permite ejecutar el mismo código bajo condiciones o parámetros diferentes a manera de línea de comandos pero con código Racket.

2.4 Desarrollando un primer programa en Racket

Para ejemplificar la diferencia entre los dos paneles, se explica el siguiente programa.

En la zona de definiciones:

```
1.(define x 1)
2.(map +
3.      '(1 2 3 4 5)
4.      '(1 2 3 4 5)
5.      '(1 2 3 4 5))
```

Código 2.1 Primer programa en Racket

Una vez que se tenga el código se debe de oprimir el botón de Ejecutar (o F5 o bien usar la combinación de teclas Ctrl + R) y automáticamente aparecerá lo siguiente en la zona de interacción:

```
1.'(3 6 9 12 15)
```

Lo cual es el resultado de la evaluación del código de las líneas 2 a 5; es decir, una lista que contenga, en su posición i, la suma de cada elemento i-ésimo de las listas involucradas.

Si escribimos el siguiente código (también en la zona de interacciones):

```
2.> (+ x 4)
```

al presionar la tecla de retorno, veremos que el intérprete nos responde con:

```
3.5
```

Es decir que sumará 4 a la variable x que se definió previamente (en la línea 1 de la zona de definiciones).

2.5 Procedimientos importantes para esta práctica

En esta sección se expondrán los procedimientos esenciales de Racket haciendo uso de una notación basada en el estándar del lenguaje, [24]. Sin embargo, en las siguientes prácticas se usará otra notación más adecuada para el desarrollo de dichas prácticas.

El primer procedimiento que se expondrá es define. Este procedimiento permite introducir variables con su respectivo valor. Dichas variables estarán disponibles en todo lo que resta del programa. Su sintaxis es:

```
(define id expr)
```

Donde <id> es el nombre que tendrá la variable y <expr> es el valor de cualquier tipo asociado a ese identificador, a manera de ejemplo:

```
1.(define x 1)
```

Otro procedimiento importante es lambda[4] ya que nos permite generar una función, su sintaxis es la siguiente:

```
(lambda(args …) body…+)
```

Donde `args` son los nombres de los parámetros del procedimiento (cualquier cantidad de parámetros, separados por espacio) y `body` es el cuerpo de lo que se quiere que haga el procedimiento. Para el ejemplo se usará también el procedimiento define

```
1.(define cuadrado (lambda(x) (* x x)))
2.(define suma (lambda(x y) (+ x y)))
```

Para poder aplicar un procedimiento, la sintaxis es:

```
(fun args …*)
```

Donde `fun` puede ser un identificador cuyo valor sea un procedimiento o bien un nuevo procedimiento declarado ahí mismo, mientras `args` representan a los argumentos que recibirá (cualquier cantidad de argumentos, separados por espacio), siguiendo con el

[4] Racket nombra a esta función lambda ya que históricamente el símbolo “λ” es usado para una de las dos construcciones básicas del sistema introducido por Alonzo Church, específicamente la abstracción. Church originalmente usó “∧x” para diferenciarla de otra construcción introducida por Whitehead y Russell representada como “x^”. Para facilitar la impresión, se remplazó a “λx”[1].

ejemplo tenemos:

```
3.(cuadrado 3)
```

A lo que el intérprete responderá con:

```
> 9
```

Aplicando el otro procedimiento:

```
4.(suma 3 7)
> 10
```

Aplicando un procedimiento sin nombrarlo:

```
4.((lambda(x) x) 5)
> 5
```

Racket también facilita el manejo de listas[5], para poder ahondar en el tema, a continuación se verá cómo construirlas usando el procedimiento `list` o el procedimiento `quote`.

```
(list args)
```

o

```
(quote (args))
```

Donde `args` serán los valores que contendrá la lista, separados por espacio.

Siguiendo con los ejemplos:

```
1. (list 1 2 3)
2. (quote (1 2 3))
```

En ambos casos, el resultado que obtendremos será:

```
>  '(1 2 3)
```

El resultado es la manera en que Racket representa internamente las listas, sin embargo, es posible construir listas con esta sintaxis, es decir, usando el símbolo `'`.
Un ejemplo para construir listas de esta nueva forma:

```
1. '(4 5 6)
```

Esta sintaxis es mucho más cómoda que las anteriores debido a la brevedad con la que permite declarar una lista.

5 Éstas son una estructura de datos que almacenan objetos con una organización lineal, [2].

Para operar con los elementos de las listas, veremos los procedimientos car[6] y cdr. El procedimiento car regresa el primer elemento de la lista, mientras que el procedimiento cdr regresa la cola de la lista. La sintaxis de cada un de ellas es:

```
(car lst)
(cdr lst)
```

Dado que `car` y `cdr` operan sobre listas, se espera que el argumento `lst` sea una lista, ya sea que se pase un identificador cuyo valor sea una lista, una lista nueva o la aplicación de un procedimiento que regrese una lista.

A manera de ejemplo:

```
1.(car '(1 2 3 4 5 6))
2.(cdr '(1 2 3 4 5 6))
```

El intérprete responderá en cada caso lo siguiente:

```
> 1
> '(2 3 4 5 6)
```

Un procedimiento útil para el manejo de listas es map, cuya sintaxis es:

```
(map proc lsts …+)
```

Siendo `lsts` una o más listas y `proc` un procedimiento cuya aridad[7] deberá ser igual al número de listas que se hayan proporcionado en la función. Este procedimiento aplicará la función proc tomando como parámetros la cabeza de cada lista, para regresar una nueva lista que contiene el resultado de proc en orden. Se ejemplifica a continuación:

```
1.(map + '(1 2 3) '(4 5 6))
```

El resultado al dejar que el intérprete lo evalúe es:

```
> '(5 7 9)
```

Este resultado se obtuvo porque el primer elemento fue el resultado de sumar `1 + 4`, el

[6] Estos procedimientos son acrónimos del inglés "**C**ontents of the **A**ddress part of **R**egister" y "**C**ontents of the **D**ecrement part of **R**egister", [21]

[7] Nos referimos a aridad de una función como el número de parámetros que dicha función recibe.

segundo elemento de evaluar `2 + 7` y por último `3 + 6`.

Los últimos procedimientos que se expondrán en esta sección son foldr y foldl.
La sintaxis es:

```
(foldl proc init lsts …+) o (foldr proc init lsts …+)
```

Donde `lsts` representan una o más listas, `init` el valor inicial con el que se comenzará a operar y `proc` un procedimiento que deberá tener una aridad de `n + 1` siendo `n` el número de listas pasadas al procedimiento.

El procedimiento `proc` es inicialmente llamado con el primer elemento de cada lista, y el elemento final será `init` (el elemento `n+1`). En las siguientes llamadas, el elemento `init` será el valor resultante de la llamada recursiva anterior de `proc`.

Las listas de entrada son recorridas de izquierda a derecha en `foldl` y de derecha izquierda en `foldr` siendo el resultado de toda la aplicación de `fold` el resultado de la última aplicación de `proc`. Si las listas están vacías, el resultado es `init`.

A continuación se presenta en ejemplo para calcular la suma de todos los enteros en una lista haciendo uso de `foldl`:

```
(foldl (lambda(x init) (+ init x)) 0 '(1 2 3 4))
```

A lo que el intérprete responderá con:

```
> 10
```

Analicemos cómo llegó el intérprete a este resultado:
El procedimiento pasado a foldl fue `(lambda(x init) (+ init x))`. El primer parámetro formal es llamado `x` y éste será el elemento de la lista según el nivel de recursión, mientras que el segundo será el acumulado de las llamadas recursivas anteriores. El valor de regreso del procedimiento es la suma de estos valores.

El primer paso es tomar la cabeza de las listas, debido a que en este caso únicamente se pasó como argumento una lista, sólo se toma la cabeza de dicha lista, es decir el valor `1`. A continuación, se toma el valor inicial (`init`) que es `0` y se pasan estos valores como argumentos al procedimiento descrito en el párrafo anterior. El procedimiento realizará la suma de los valores, es decir, `(+ 0 1)` que se evalúa a `1`.

El siguiente paso es tomar el resto de la lista y pasar como nuevo valor inicial (`init`) la suma previamente calculada, lo cual se podría escribir así:

```
(foldl (lambda(x init) (+ init x)) 1 ‘(2 3 4))
```

Esta llamada se resuelve de la misma manera, es decir, se toma la cabeza de la lista, (la cual tiene valor de 2) junto con el valor inicial (que es la suma de la llamada anterior, o sea 1) y se suman, quedando la expresión `(+ 2 1)` que se evalúa a 3.

De la misma manera, se toma la cola de la lista y se pasa este nuevo resultado como valor inicial (`init`), lo que se escribiría así:

```
(foldl (lambda(x init) (+ init x)) 3 ‘(3 4))
```

Se continúa de la misma manera hasta que la lista se queda vacía y el resultado es el valor que almacene `init`, lo que se vería así:

```
(foldl (lambda(x init) (+ init x)) 10 ‘())
```

El procedimiento `foldl` recorrió las listas de izquierda a derecha, por lo que la suma que se hizo fue 0+1+2+3+4, si en cambio se utilizara el procedimiento `foldr`, el recorrido de las listas se haría en el sentido inverso, por lo que la suma que se realizara sería 0+4+3+2+1.

2.6 Ejercicios

1. **(ackermann m n)**

   Calcular el valor de la función de ackermann(m,n)
   El procedimiento está definido para **m** y **n** naturales de la siguiente manera:
   ackermann(m,n) = n + 1 si m = 0
   ackermann(m-1,1) si m > 0 y n = 0
   ackermann(m-1, ackermann(m,n-1)) si m > 0 y n > 0
   Si **m** o **n** no son naturales, se debe mandar un error.
   Ejemplos:
   ```
   > (ackermann 3 3)
   61
   > (ackermann -3 3)
   "error"
   ```

2. **(filtro lista predicado)**
   Filtrar el contenido de una lista. El procedimiento usa dos parámetros. El primero será la lista a filtrar y el segundo parámetro será un procedimiento que recibe un elemento y regresa **true** si debe ser conservado o **false** si debe ser eliminado. Al aplicar dicho procedimiento a cada elemento de la lista, se sabe si se debe conservar o no.

En caso de recibir la lista vacía se debe regresar también la lista vacía
Ejemplos:

```
>(filtro (list 1 "hola" 2 (list 1 2) 3 'a 'b) number?)
'(1 2 3)
>(filtro (list 1 "hola" 2 (list 1 2) 3 'a 'b) symbol?)
'(a b)
>(filtro (list 1 "hola" 2 (list 1 2) 3 'a 'b) list?)
‘((1 2))
>(filtro (list 1 "hola" 2 (list 1 2) 3 'a 'b) string?)
'("hola")
```

3. **(deriva-suma pol1 pol2 x)**

Representaremos polinomios de la siguiente manera:
La lista `'(4 3 2 1 0)` representa al polinomio $4x^4+3x^3+2x^2+1x^1+0$ mientras que la lista `'(16 9 4 1)` representa al polinomio $16x^3+9x^2+4x^1+1$

En otras palabras, la cabeza de la lista representa al coeficiente, mientras que la **longitud - 1** representa el exponente.
Hacer el procedimiento **(deriva-suma pol1 pol2 x)**, donde **pol1** y **pol2** representan dos listas de polinomios como se describe anteriormente y **x** es un número que representa la variable del polinomio. El procedimiento deberá derivar cada polinomio, después sumar las derivadas y evaluar el polinomio resultante con el parámetro **x**.

Ejemplo:

```
>(deriva-suma '(4 3 2 1 0) '(2 2 2 2) 3)
594
```

Es decir:

$pol1 = f(x) = 4x^4+3x^3+2x^2+1x^1+0$
$pol2 = g(x) = 2x^3+2x^2+2x^1+2$
$f'(x) = 16x^3+9x^2+4x^1+1$
$g'(x) = 6x^2+4x^1+2$
$f'(x)+ g'(x) = 16x^3+15x^2+8x^1+3$
$f'(3)+ g'(3) = 16(3)^3+15(3)^2+8(3)^1+3$
$f'(3)+ g'(3) = 594$

4. (primero-y-ultimo lista funcion)

Dada una lista **l** de longitud par y una función **f** de aridad dos, construir una lista con el resultado de aplicar **f** al primer y último elemento de **l**, quitar dichos elementos y aplicar el mismo procedimiento hasta que la lista quede vacía.

Ejemplos:

```
>(primero-y-ultimo '(1 2 3 4 5 6) +)
'(7 7 7)

>(primero-y-ultimo '(1 2 3 4 5 6) *)
'(6 10 12)

>(primero-y-ultimo '(1 2 3 4 5) /)
“error: la longitud de la lista no es par”
```

5. (compara-longitud-listas l1 l2)

Comparar la longitud de las listas **l1** y **l2**, sin usar el procedimiento length. Los valores de regreso deberán ser únicamente los símbolos **'lista1-mas-grande**, **'lista2-mas-grande** o **'listas-iguales**.

Ejemplo:

```
>(compara-longitud-listas (list 1 2 3)
                          (list 5 6 7 8 9 10))
'lista2-mas-grande
```

6. (entierra nombre n)

Definir el procedimiento que *enterrará* al símbolo **nombre**, **n** número de veces.
Es decir, se deberá *anidar* **n** - 1 listas hasta que se llegue a la lista que tiene al símbolo **nombre**.

Ejemplo:

```
>(entierra ‘Raszagal 5)
'(((((Raszagal)))))
```

7. (uno-si-uno-no lista)

Construye una lista con los elementos que se encuentran en las posiciones nones de la lista.

Ejemplo:

```
>(uno-si-uno-no '(pumas chivas borussia-dortmund bayern-munich))
'(pumas borussia-dortmund)
```

8. (merge lista1 lista2)

Suponer que las listas **lista1** y **lista2** se encuentran ordenadas ascendentemente. Hacer el procedimiento **merge** el cual construirá una lista ordenada de manera ascendente con los elementos de ambas listas. Queda prohibido usar el procedimiento **sort**.

Ejemplo:

```
>(merge '(1 2 6 8 10 12) '(2 3 5 9 13))
'(1 2 2 3 5 6 8 9 10 12 13)
```

9. (ordena-por-longitud lista)

El procedimiento recibirá una lista que contiene varias listas. El propósito es regresar una lista que contenga las mismas listas ordenadas por longitud de menor a mayor.

Ejemplo:

```
>(ordena-por-longitud '((5 4 3 2 1) (5 4 3 2) (5 4 3)
                        (5 4) (10 11) (6 5 4 3 2 1)))
'((5 4) (10 11) (5 4 3) (5 4 3 2) (5 4 3 2 1) (6 5 4 3 2 1))
```

2.7 Sitios importantes relacionados con el lenguaje

Los siguientes son sitios en Internet donde se puede localizar la documentación y especificación oficial del lenguaje Racket:

http://racket-lang.org/
Página principal del lenguaje. Aquí se puede desde bajar el intérprete hasta ver ejemplos de un servidor web implementado con continuaciones, [18].

http://docs.racket-lang.org/
Se recomienda consultar la documentación, en ella se encuentran, entre otras cosas, las definiciones de todas los procedimientos así como ejemplos de cómo usar cada uno de ellos, [17].

Capítulo 3
Tipos de datos abstractos

3.1 Objetivos

- Familiarizar al estudiante con el concepto de tipo de datos abstracto, ya que éstos serán la base de la construcción y manipulación de gramáticas para las siguientes prácticas.
- Construir tipos de datos basados en un modelo abstracto.
- Operar recursivamente sobre tipos de datos.

Tipos de dato

"Un tipo de dato es una clasificación que identifica uno de varios tipos de datos, como valores reales, enteros o booleanos, que determinan los valores posibles para ese tipo de dato, las operaciones que se pueden aplicar a valores de ese tipo, el significado de los datos y la forma en que dicho tipo puede ser almacenado"[11].

3.2 Predicados

Antes de analizar la sintaxis para definir tipos de datos, se expondrá el comportamiento de algunos procedimientos especiales llamados **predicados**.

Un **predicado** es un procedimiento que recibe un único argumento y su valor de regreso siempre es un booleano (La representación de valores lógicos en Racket es true o #t para verdadero y false o #f para falso). La idea detrás de éstos es verificar si cumplen o no cierta propiedad.

Para ejemplificar, podemos analizar los predicados integer? y list?, los cuales nos confirman si el argumento es un entero o una lista, respectivamente.

Al ejecutar en el intérprete la siguiente instrucción:

```
>(list? '(1 2 3 4 5 6))
```

El resultado será:

```
> #t
```

Mientras que:

```
>(integer? "hola")
> #f
```

Lo cual quiere decir que, efectivamente, en el primer ejemplo el argumento pasado al predicado fue una lista, mientras que en el segundo ejemplo no se pasó un número sino una cadena.

Para operar con valores lógicos, podemos hacer uso de los siguientes procedimientos, not, and y or, que regresan la negación, conjunción y disyunción, respectivamente.

El procedimiento not tiene la siguiente sintaxis:

```
(not arg)
```

Mientras que procedimiento and se escribe como:

```
(and expr ...)
```

Igualmente la del procedimiento or:

```
(or expr ...)
```

Es necesario resaltar que los procedimientos and y or pueden recibir cero, uno o varios argumentos. Esta propiedad es muy flexible y útil ya que en caso de que sea necesario calcular el valor de una conjunción o disyunción de múltiples variables, se puede hacer mediante una sola llamada al procedimiento correspondiente.

Para ejemplificar el procedimiento `and`, supongamos que queremos hacer una función f que decida si el único parámetro que recibe es una lista, y además que el primer elemento de dicha lista es un entero.

Dicho procedimiento se escribiría así:

```
1.(define f
2.    (lambda (param)
3.      (and (list? param) (integer? (car? param)))))
```

Código 3.1 Ejemplo de uso del procedimiento and

En el código 3.1, en el tercer renglón se puede observar como en un mismo and verificamos que se tiene una lista y que el primer elemento de la misma sea un entero, definiendo así el predicado f.

3.3 Dialecto PLAI

DrRacket contiene un conjunto de herramientas llamado PLAI (Programming Languages: Application and Interpretation) especialmente diseñadas para el desarrollo de intérpretes.

Dentro de este conjunto de herramientas, estudiaremos dos procedimientos básicos, uno para definir tipos de datos y otro para manipularlos. Empezaremos con el procedimiento define-type[8], cuya sintaxis es:

```
(define-type type-id variant ...)
```

Donde `variant`:

```
variant = (variant-id (field-id contract-expr) ...)
```

Para explicar la sintaxis, veremos un ejemplo de cómo definir nuestro tipo lista el cual únicamente contendrá elementos de tipo entero.

```
1.#lang plai
2.(define-type lista
3.    [vacia]
4.    [mi-cons (cabeza integer?)
5.          (cola lista?)])
```

Código 3.2 Ejemplo de tipo de datos lista

[8] Para poder usar este procedimiento, es necesario cambiar la primer línea del intérprete de “#lang Racket” a “#lang plai” (sin comillas)

La línea `2` define cómo se llamará nuestro tipo de dato, es decir, lo nombra. En la línea `3` definimos la primer variante de nuestro tipo lista de enteros, la lista vacía. En la línea `4` definimos la variante `mi-cons` la cual está compuesta de dos campos o atributos. El primer atributo es el elemento entero que la lista almacenará, el cual llam aremos `cabeza` y después pasaremos como argumento un predicado para que el intérprete valide que el atributo `cabeza` cumpla al momento de construir un elemento de la variante mi-cons. Dado que queremos construir listas de enteros, utilizaremos el predicado `integer?`. El segundo atributo será el resto de la lista, el llamado `cola` el cual queremos que satisfaga el predicado `lista?`, es decir, que sea una expresión que se evalúe al tipo `lista` que recién fue definido en la línea `1`.
Es necesario explicar por qué existe el predicado `lista?` ya que en ningún momento fue definido por nosotros. Al momento de que definimos un tipo mediante el procedimiento `define-type`, PLAI genera cuatro procedimientos adicionales, los cuales son:

- Un predicado `type-id?`, el cual recibe un parámetro cualquiera y nos regresa `true` cuando dicho parámetro se evalúa a algo de ese tipo y `false` en cualquier otro caso.
  En el ejemplo, se genera el predicado `lista?` que usamos en la línea `4`.
- Para cada variante, se genera un predicado `variant-id?`, que recibe un parámetro y nos regresa `true` cuando dicho parámetro se evalúa a algo de esa variante y `false` en cualquier otro caso.
  En el ejemplo no hicimos uso de ninguno de estos predicados, sin embargo se podrían usar los procedimientos `vacia?` y `mi-cons?`
- Para cada atributo de cada variante, un procedimiento para acceder al valor de dicho atributo que tendrá el nombre de `variant-id-field-id`. Cabe mencionar que, en el ejemplo, tampoco hicimos uso de estos procedimientos, los cuales son `mi-cons-cabeza` y `mi-cons-cola`.
- Para cada atributo de cada variante, se crea un procedimiento para modificar el valor de dicho atributo que tendrá el nombre de `set-variant-id-field-id!`. Los procedimientos en el ejemplo son `set-mi-cons-cabeza!` y `set-mi-cons-cola!`.

Veamos ahora cómo usar estos procedimientos:

```
1.(define ejemplo  (mi-cons 1 (mi-cons 2 (mi-cons 3 (vacia)))))
2.(lista? ejemplo)
3.(mi-cons? ejemplo)
4.(mi-cons-cabeza ejemplo)
5.(mi-cons-cola ejemplo)
```

Código 3.3 Ejemplos de procedimientos definidos por PLAI

A lo cual, el intérprete nos responderá con:

```
> #t
> #t
> 1
> (mi-cons 2 (mi-cons 3 (vacia)))
```

Lo anterior debido a que en el renglón `1`, definimos la variable `ejemplo` como una lista de números del 1 al `3`. El procedimiento `define` no regresa nada ya que únicamente asigna un valor a una variable. En el renglón `2` se pregunta si la variable `ejemplo` es de tipo `lista`; la respuesta es el primer `#t`. En la línea `3` preguntamos si la variable `ejemplo` es una variante del tipo `mi-cons`; la respuesta es el segundo `#t`. En la línea `4` se está extrayendo el valor del atributo `cabeza` de una variante de tipo mi-cons; la respuesta es `1`. Por último, en la línea `5` se está extrayendo el valor del atributo `cola` de una variante de tipo `nodo`; la respuesta es
`(mi-cons 2 (mi-cons 3 (vacia)))`.

El otro procedimiento que veremos es type-case. Este procedimiento permite manipular los tipos de datos que hayamos definido con una sintaxis amigable en vez de usar los procedimientos `variant-id-field-id`. Su sintaxis es de esta forma:

```
(type-case type-id expr branch ...)
```

Donde `branch`:

```
branch = (variant-id (field-id ...) result-expr ...)
       | (else result-expr ...)
```

Analicemos la sintaxis siguiendo con el ejemplo anterior. Supongamos ahora que queremos un procedimiento que sume una unidad a los valores numéricos de la lista, dicho procedimiento se vería así:

```
1.(define suma-1
2. (lambda (lista-parametro)
3.  (type-case lista lista-parametro
4.    [vacia () (vacia)]
5.    [mi-cons (cabeza cola)
6.          (mi-cons (add1 cabeza) (suma-1 cola))])))
```

Código 3.4. Ejemplo de tipos de datos
En la línea 1 definimos la variable `suma-1`. En la línea 2 llamamos a `lambda` con el parámetro `lista-parametro` para crear el procedimiento. En la línea 3 empezamos el `type-case`, el primer argumento es el tipo de dato al que queremos acceder, es decir, el tipo dato `lista`, como segundo parámetro, recibe una expresión que se debe evaluar a algo del tipo lista, en este caso usamos la variable que recibirá el procedimiento que estamos construyendo. En la línea 4, comenzamos con la manipulación de las variantes:

La primer variante del tipo `lista` es `vacia`, se usa este identificador, dado que la variante `vacia` no tiene atributos, estamos forzados a escribir un paréntesis que abre e inmediatamente después uno que cierra; después sigue la expresión que queremos que regrese al encontrarnos una lista vacía, en este caso, al ser la lista vacía, ya terminamos y únicamente se regresa la lista `vacia` (es el caso base). La variante interesante empieza en la línea 5 donde empezamos declarándola junto a los dos atributos que tiene, `cabeza` y `cola` . En la línea 6, empieza la expresión que queremos que regrese al encontrarnos con una expresión de tipo `mi-cons`. Lo primero que hacemos es construir un nuevo tipo `mi-cons`, con un nuevo valor `cabeza` que será el valor de la `cabeza` actual más una unidad. Esto lo lleva a cabo el procedimiento `add1` y el valor actual se encuentra en la variable `cabeza`. Después definimos quien será la nueva `cola` de esta nueva lista, la cual será la llamada recursiva a `suma-1` recibiendo como argumento la `cola` actual.

Para probar el nuevo procedimiento `suma-1`, hacemos:

```
>(suma-1 ejemplo)
```

Para obtener como respuesta:

```
(mi-cons 2 (mi-cons 3 (mi-cons 4 (vacia))))
```

3.4 Ejercicios

1. **(construye-lista n f)**

Dado un número **n** , y una función **f**, construir una lista de longitud **n**, donde el elemento **i**-ésimo de la lista (con **i** entre 1 y **n**), sea la aplicación de **f** con parámetro **i**, es decir, **f**(i) Únicamente se deberá aceptar **n** entera y positiva.[9]

Ejemplos:

```
> (construye-lista 10 (lambda (x) x))
'(0 1 2 3 4 5 6 7 8 9)

> (construye-lista 5 (lambda (x) (* x x)))
'(0 1 4 9 16)

> (construye-lista -5 (lambda (x) (sqrt x)))
"error, no se acepta parámetro negativo"
```

2. **(elimina lista)**

Eliminar los elementos repetidos de **lista** dejando solo uno de ellos.

Ejemplos:

```
> (elimina '(1 1 1 1 1))
'(1)

> (elimina '(a b c a b c d e f a b c))
'(a b c d e f)
```

[9] Para esta práctica, no se permite hacer uso del procedimiento `build-list` de Racket

3. Conjuntos

Usando el siguiente tipo de datos (data-type) para conjuntos:

```
1.(define-type conjunto
2.     [conj (lista list?)]
3.     [complemento  (c conjunto?)]
5.     [interseccion (c1 conjunto?) (c2 conjunto?)]
6.     [union (c1 conjunto?) (c2 conjunto?)]
7.     [diferencia (c1 conjunto?) (c2 conjunto?)]
8.     [productocar (c1 conjunto?) (c2 conjunto?)])
```

Código 3.5 Definición tipo conjunto

Define los procedimientos **convertir** y **calcular** tomando en cuenta lo siguiente:

- El universo de discurso U será (0,1,2, ... , 99)
- Si $x \in$ a $c1 \cap c2$ entonces $x \in c1$ y $x \in c2$
- Si $x \in$ a $c1 \cup c2$ entonces $x \in c1$ ó $x \in c2$
- Si $x \in$ a $c1 \setminus c2$ entonces $x \in c1$ y $x \notin c2$
- El complemento de un conjunto c está definido como $U \setminus c$
- El producto cartesiano de c1 y c2 es el conjunto de todos los pares ordenados (**x**,**y**) tal que **x** pertenece a c1 y **y** a c2, es decir:
  $c1 \times c2 = \{(x,y) \mid x \in c1 \wedge y \in c2\}$
- Los conjuntos serán representados por listas, para no aceptar elementos duplicados, en el procedimiento **convertir**, se deberá hacer uso del ejercicio 2.
- El procedimiento **convertir** deberá tomar la lista y regresar una construcción del tipo de datos `conjunto`.
- El procedimiento **calcular** debe recibir el tipo de dato que regrese **convertir** y evaluar la expresión. El valor de retorno debe ser un conjunto con el resultado de hacer todas las operaciones.

Ejemplos:

```
>(calcular (convertir '{interseccion {1 2 3} {4 5 6}}))
(conj '())
```

```
> (calcular (convertir
               '{productocar {union {1} {2 3}}
                             {interseccion {10 20 30}
                                           {10 20 30 1 2 3}}}))
(conj '((1 10) (1 20) (1 30)
        (2 10) (2 20) (2 30)
        (3 10) (3 20) (3 30)))
```

4. (contenido? c1 c2) e **(iguales? c1 c2)**

Hacer dos procedimientos, basándose en el tipo de datos del ejercicio 4, que decidan si un conjunto está contenido en otro o si son iguales. Es decir:
(contenido? c1 c2), nos dice si **c1** está contenido en **c2**
(iguales? c1 c2), nos dice si **c1** = **c2**

Ejemplos:

```
> (contenido? '{interseccion {1 2 3 4 5} {3 4 5 6 7}}
              '{1 2 3 4 5 6 7})
true

> (iguales? '{union {10 20 30} {1 2 3}} '{10 1 20 2 30 3})
true


> (contenido? '{0 1 2} ‘{10 20 30})
false
```

5. *Árboles: (altura arb), (inorden arb), (postorden arb), (preorden arb) y (opera-en-hojas arb fun neutro)*

Usando el siguiente datatype para árboles binarios:

```
1.(define-type arbolb
2.     [nodo (elem number?) (sai arbolb?) (sad arbolb?)]
3.     [hoja  (elem number?)]
4.     [vacio])
```

Código 3.6 Definición tipo `arbolb`

Define un procedimiento que reciba un *arbolb* y regrese su **altura**. Recuerda que la altura es la longitud del camino más largo entre la raíz y el nodo más profundo.
La profundidad de un nodo es la longitud del camino entre el nodo y la raíz.
Ejemplo:

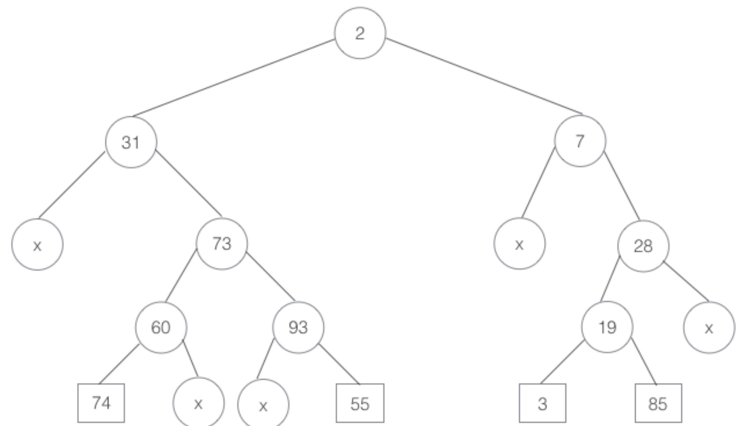


Figura 2: Ejemplo de árbol binario

Pasemos a código el ejemplo mostrado en la Figura 2, los círculos con número representan un nodo, los círculos con “x” representan un árbol vacío y los cuadros con número representan una hoja:

```
> (define ejemplo
   (nodo 2
        (nodo 31
              (vacio)
              (nodo 73
                    (nodo 60
                          (hoja 74) (vacio))
                    (nodo 93
                          (vacio) (hoja 55))))
        (nodo 7
              (vacio)
              (nodo 28
                    (nodo 19
                          (hoja 3)
                          (hoja 85))
                    (vacio)))))

>(altura ejemplo)
4
```

Definir los procedimientos **preorden**, **inorden** y **postorden** que regresarán una lista con los valores numéricos que haya en los nodos haciendo el recorrido correspondiente.
Ejemplos:

```
>(preorden ejemplo)
'(2 31 73 60 74 93 55 7 28 19 3 85)

>(inorden ejemplo)
'(31 74 60 73 93 55 2 7 3 19 85 28)

>(postorden ejemplo)
'(74 60 55 93 73 31 3 85 19 28 7 2)
```

Definir el procedimiento **opera-en-hojas** que recibirá un árbol binario **arb**, una función llamada **fun** cuya característica es que tiene aridad 2, y un valor **neutro** asociado al procedimiento.

El procedimiento **opera-en-hojas** deberá aplicar la función **fun** únicamente a los valores numéricos alojados en las hojas.

Ejemplo:

```
> (opera-en-hojas ejemplo + 0)
217

>(opera-en-hojas ejemplo * 1)
1037850
```

6. *Pilas: agregar (push), quitar (pop) y ver (peek)*

Una pila es una estructura de datos que cumple la característica que el último elemento insertado es el primero que debe ser eliminado. Lo primero que se debe implementar es el tipo de dato que representa a esta estructura, la cual únicamente contendrá valores numéricos.
A continuación, se deberán implementar los procedimientos ***agregar, quitar*** y ***ver***. El procedimiento ***agregar*** recibe la pila y un nuevo número e inserta ese nuevo número al tope de la pila. El procedimiento ***quitar*** recibe una pila y remueve el último elemento que se haya agregado para regresar una nueva pila sin dicho elemento. Por último, el procedimiento ***ver*** recibe una pila y regresa el último número que haya sido agregado.

Ejemplos:

```
(define ejemplo (nodo 1 (nodo 2 (nodo 3 (nodo 4 (vacia))))))
;;En esta pila, el primer elemento en agregarse fue el 1
;;mientras que el último elemento fue el 4

> (ver ejemplo)
  4

> (quitar ejemplo)
  (nodo 1 (nodo 2 (nodo 3 (vacia))))

> (agregar ejemplo 0)
  (nodo 1 (nodo 2 (nodo 3 (nodo 4 (nodo 0 (vacia))))))
```

Capítulo 4 Análisis sintáctico e interpretación

4.1 Objetivo

- Estudiar los conceptos de análisis sintáctico e interpretación, y elaborar tanto un analizador sintáctico (parser), como un intérprete que contemple las primeras características de éste
- Construir árboles de sintaxis abstracta recibiendo como entrada la sintaxis concreta de un programa, verificando que cumpla las especificaciones de la gramática
- Evaluar el resultado de un programa, usando como entrada el árbol de sintaxis abstracta, lo cual será la base para la interpretación de las prácticas siguientes.

4.2 Análisis sintáctico

"*El análisis sintáctico es el proceso de organizar la secuencia de lexemas en estructuras sintácticas jerárquicas tales como expresiones, enunciados y bloques. Esto es como organizar o diagramar un enunciado en cláusulas. La estructura sintáctica de un lenguaje es típicamente especificado usando una definición EBNF*[10]*, también llamada gramática libre de contexto*"[6].

---

[10] Por sus siglas en inglés Extended Backus-Naür Form, las cuales fueron tomadas de los apellidos de los científicos que propusieron esta notación.
John Warner Backus científico estadounidense, en 1950 inventó Fortran, el primer lenguaje de programación de alto nivel mientras que en 1959 creó una notación para describir la sintaxis de un lenguaje de programación, [16].
Peter Naür científico danés que fue reconocido por el diseño del leguaje Algol 60, [14].

"*El analizador sintáctico es un programa que convierte sintaxis concreta (lo que el usuario pueda escribir) a sintaxis abstracta. La palabra abstracta significa que el resultado del programa es idealizado, de este modo está separado de la representación física o sintáctica*" [11].

4.3 Construyendo un primer intérprete

Un intérprete es: *"Un programa que va leyendo "sentencia por sentencia", ejecutándolas en dicho orden"*[6].

Para construir el primer intérprete, se empezará con la gramática más simple, con la intención de que quede claro el proceso de construcción del mismo. La gramática AE, en su forma EBNF es la siguiente:

```
<AE> ::= <num>
       | {+ <AE> <AE>}
       | {- <AE> <AE>}
```

Código 4.1 Gramática `AE` en `EBNF`

El siguiente es un ejemplo[11] de una operación con la gramática anterior `‘{+ 2 3}`, donde se indica una suma de dos números; otro ejemplo, un poco más elaborado sería:
`‘{+ {- 1 1} {+ 2 1}}`.

Lo primero que se debe hacer es escribir el analizador sintáctico, mismo que recibirá las expresiones de la gramática AE como se acaba de describir. Es necesario resaltar el símbolo ‘ que se encuentra justo antes de la primer llave "`{`". Esto se hace con la intención de que la expresión de la gramática AE sea una lista de Racket para que resulte más fácil hacer el análisis sintáctico usando el constructor de listas `quote`.

Definiendo la sintaxis abstracta

El trabajo del procedimiento `parse`, entre otras cosas, es verificar que no haya errores de sintaxis y traducir la sintaxis concreta (lo que escribe el usuario) a sintaxis abstracta (la representación que el intérprete hace de la gramática).

11 A los ejemplos de las gramáticas se les antepone el símbolo ‘ para que al pasarlo a Racket lo interprete como una lista de símbolos y números.

Es necesario describir la gramática, para lo cual haremos uso de tipos de datos.

```
1.#lang plai
2.(define-type AE
3.   [num (n number?)]
4.   [suma (izq AE?) (der AE?)]
5.   [resta (izq AE?) (der AE?)])
```

Código 4.2 Definición PLAI de gramática AE

La línea 1 es para indicar que usaremos el dialecto plai necesario para poder hacer uso de los procedimientos que éste provee. En la línea 2 definimos el tipo de datos AE. En la línea 3 se define la variante `num` y se revisa que sea de tipo numérico. En la línea 4 definimos la variante `suma` que consta de dos partes, el operando izquierdo y el derecho. Éstas se piden de tipo AE para poder formar expresiones compuestas y no únicamente sumas o restas. En la línea 5 se procede de la misma manera pero con la variante `resta`.

Una vez definida la sintaxis abstracta, se puede empezar a implementar el analizador sintáctico. Para tener en mente el funcionamiento que se desea del analizador sintáctico, se presentarán tres ejemplos:

Ejemplo 1. `‘3`

Ejemplo 2. `‘{+ 2 1}`

Ejemplo 3. `‘{+ {- 2 5} {+ 5 9}}`

Estos tres ejemplos son expresiones válidas y el analizador sintáctico deberá ser capaz de transformarlas a sintaxis abstracta. Hablando en términos del lenguaje de programación Racket, la gramática AE únicamente puede ser expresada mediante el tipo de dato numérico (para casos como el Ejemplo 1) o bien una lista de símbolos y números (Ejemplos 2 y 3).

Definiremos al analizador sintáctico como un procedimiento llamado `parse` cuyo objetivo será verificar que las expresiones cumplan esta condición de tipos. Además, comprobar que los símbolos sean exclusivamente los que la gramática define, es decir, + y -.

Una vez que se haya verificado que la expresión cumple con estas condiciones, el analizador sintáctico deberá regresar una construcción del tipo de dato AE , que representa a la sintaxis abstracta (definida en el Código 4.2).

Analicemos el código del procedimiento `parse`:

```
 1. (define parse
 2.  (lambda (exp)
 3.    (cond
 4.        [(number? exp) (num exp)]
 5.        [(list? exp)
 6.          (case (car exp)
 7.             [(+) (suma (parse (cadr exp)) (parse (caddr exp)))]
 8.             [(-) (resta (parse (cadr exp)) (parse (caddr exp)))]
 9.             [else (error 'parse "Operación no definida")])]
10.   [else (error 'parse "Expresión inválida")])))
```

Código 4.3 Procedimiento `parse` para gramática `AE`

En la línea 1 se define el procedimiento parse, en la línea 2 se nombra al parámetro que recibirá el procedimiento como `exp`. En la 3 vemos una condicional requerida para analizar cada caso, debido a que esta gramática es muy simple, sólo hay dos casos, que se reciba un número o una lista.

A partir de la línea 4 se examina el caso de que la expresión recibida haya sido un número, como en el Ejemplo 1 ya que cualquier número es una expresión válida. En la línea 5, verificamos si se recibe una lista, y en la línea 6 preguntamos por el primer elemento de lista que se recibió como parámetro.

En la 7, si se tiene el símbolo "+" entonces construimos un tipo `suma` cuyo lado izquierdo será la llamada recursiva a la función parse, pasándole como parámetro el segundo elemento de la lista (el procedimiento cadr regresa el segundo elemento de una lista), y cuyo lado derecho será la llamada recursiva a parse, pasándole como parámetro el tercer elemento de la lista (el procedimiento caddr regresa el tercer elemento de una lista), esto es para resolver casos como el de los Ejemplos 2 y 3. En la línea 8 hacemos lo mismo que en la línea anterior, con la diferencia de que es una `resta`; mientras que en la 9 se envía un error si es que no recibimos alguno de los dos operadores definidos.

El procedimiento error recibe dos parámetros, el primero es el nombre del procedimiento que manda el error con el tipo de dato símbolo y el segundo es una cadena que describa el error. Finalmente en la línea `10` se manda un error si no se recibió una expresión de tipo lista o de tipo numérico.

Una vez construido el analizador sintáctico, analicemos la ejecución del procedimiento con los ejemplos:

```
(parse '3)
> (num 3)
```

```
(parse '{+ 2 1})
> (suma (num 2) (num 1))
```

```
(parse '{+ {- 2 5} {+ 5 9}})
> (suma (resta (num 2) (num 5)) (suma (num 5) (num 9)))
```

Una vez que se tiene la sintaxis abstracta, podemos hacer la parte del intérprete que se encargará de evaluarla, a este procedimiento le llamaremos `interp`. El código se ve así:

```
1. (define interp
2.  (lambda (exp)
3.    (type-case AE exp
4.      [num (n) n]
5.      [suma (izq der) (+ (interp izq) (interp der))]
6.      [resta (izq der) (- (interp izq) (interp der))])))
```

Código 4.4 Procedimiento `interp` para gramática AE

En la línea 1 se define el procedimiento interp, en la 2 se nombra al parámetro que recibirá el procedimiento, en este caso será `exp`. En la 3 se verifica el tipo de dato para saber qué variante del tipo AE se recibirá, esto ya que interp "asume" que la expresión pasada como parámetro ha sido procesada por el procedimiento parse. En la línea 4 se comienza el análisis de los casos, el primero de ellos es recibir un número, a lo que simplemente regresamos ese mismo número, para casos como en el Ejemplo 1.

En la línea 5 se nombran los atributos de la variante suma como `izq` y `der`. El propósito es sumar ambos lados, sin embargo se debe aplicar recursivamente el procedimiento interp para casos como en los Ejemplos 2 y 3. El funcionamiento del código en la línea 6 es igual al descrito en la línea 5 pero con la variante resta.

Una vez construido el intérprete, analicemos la ejecución con los ejemplos (es necesario recordar que el procedimiento interp recibe la sintaxis abstracta del procedimiento parse)

```
(interp (parse '3))
> 3

(interp (parse '{+ 2 1}))
> 3

(interp (parse '{+ {- 2 5} {+ 5 9}}))
> 11
```

Substitución, *with* y alcance

"*El proceso de substitución de un identificador i por un valor v, en una expresión e, consiste en remplazar todas las presencias libres del identificador i en la expresión e por el valor v*"[11].

Para poder entender mejor la definición, añadiremos la primitiva with[12] que brinda la posibilidad de asignar variables con un valor. La sintaxis de dicha primitiva es:

```
{with {<var> <val>} <cuerpo> }
```

Donde `<var>` es la nueva variable que se está introduciendo, `<val>` es el valor que tendrá y `<cuerpo>` es la expresión donde se utilizará esa variable. Ejemplos:

Ejemplo 1. `{with {a 3} {+ a b}}`

Ejemplo 2.
```
{with {x {+ 1 1}}
      {with {y {+ 2 2}}
            {+ x y}}}
```

Código 4.5 diferentes ejemplos `with`

Una variable de ligado es "*aquella variable donde se introduce por primera vez su valor*"[11]. En el caso del with, las variables de ligado son las que se encuentran en la posición `<var>`. En el ejemplo 1, la primer presencia de la variable a es una variable de ligado.

El alcance de una variable de ligado "*es la región del programa en el cual las presencias del identificador refieren al valor acotado por la variable de ligado*"[11]. En el ejemplo 1, el alcance de variable a es la expresión `{+ a b}`.

"*En un lenguaje con alcance estático, el alcance de una variable de ligado es una región sintácticamente delimitada*"[11].

"*En un lenguaje con alcance dinámico, el cuerpo del procedimiento es evaluado en un ambiente obtenido mediante la extensión del ambiente previo al punto en que se llamó*"[7].

---

12 La primitiva with funciona de la misma manera que el procedimiento let de Racket

Una variable ligada es "*aquella que está contenida en el alcance de una variable de ligado*"[11].

En el Ejemplo 1, la segunda aparición de la variable a es una variable de ligado.

Una variable libre es "*una variable que no está en el alcance de una variable de ligado*"[11].

En el Ejemplo 1, la aparición de la variable b aparece como una variable libre.

Existen dos maneras de llevar a cabo la sustitución. La primera es hacer sustitución textual, es decir, remplazar todas las variables ligadas por el valor indicado en la variable de ligado. Esta forma presenta el inconveniente que su implementación tiene un orden de $O(n^2)$ donde n es la cantidad de variables a sustituir (es decir, la cantidad de variables de ligado).

La segunda consiste en hacer uso de una estructura llamada **ambiente**, la cual es un repositorio de sustituciones diferidas, es decir, se almacenará en esta estructura de datos cada una de las variables con su valor correspondiente, [11].

En estas prácticas implementaremos sustitución diferida con alcance estático. Para lograrlo, se implementarán los ambientes usando una estructura de lista con comportamiento de pila. A continuación se detalla la ejecución del Ejemplo 2 (Código 4.5) y el modo en que se usan los ambientes.

#	Análisis	Código a evaluar	Figura representativa del ambiente

1	Se empieza la ejecución, inicialmente el ambiente está vacío	`{with {x {+ 1 1}}`
		`{with {y {+ 2 2}}`
		`{+ x y}}}`
			(tabla vacía)
			Ambiente

2	Se agrega **x** al ambiente, cabe mencionar que primero evaluamos la expresión (+ 1 1) que le da valor a la variable x	`{with {y {+ 2 2}}` `{+ x y}}`	x | 2 Ambiente

#	Análisis	Código a evaluar	Figura representativa del ambiente
3	Se agrega **y** al ambiente, debido a la estructura de pila, y es ubicada en el tope de ésta	{+ x y}	y 4 x 2 Ambiente
4	Se busca **x** en el ambiente empezando por el tope. Se regresa la primer ocurrencia y se substituye.	{+ 2 y}	y 4 x 2 Ambiente
5	Se busca **y** en el ambiente empezando por el tope. Se regresa la primer presencia y se substituye. Al evaluar esta expresión, se llega al resultado de 6.	{+ 2 4}	y 4 x 2 Ambiente

Funciones y su uso

Primer orden: “*Las funciones no son valores en el lenguaje. Éstas sólo pueden ser definidas en una porción designada del programa, donde tienen que recibir un nombre para ser usadas en el resto del programa*”[11].

Orden superior: “*Las funciones pueden regresar otras funciones como valores*”[11].

Primera clase: “*Las funciones son valores con todos los derechos de otros valores en el lenguaje. En particular, pueden ser pasadas como argumentos a otras funciones, ser regresadas por funciones como respuestas y ser almacenadas en estructuras de datos*”[11].

Debido a que se busca que los intérpretes de estas prácticas se apeguen lo más posible al paradigma funcional y con ello se alcancen los objetivos de cada tema de la materia, implementaremos funciones de primera clase en éstos, lo que permitirá hacer construcciones análogas a las de Racket mismo (por ejemplo `map` y `fold`).

Para llevar a cabo la implementación de funciones, haremos uso de **cerraduras (closures)**, las cuales son *"una estructura de datos que contiene todo lo que el procedimiento necesita para ser aplicado. Decimos que el procedimiento está* cerrado sobre *o* cerrado en *su ambiente de creación"*[7].

Para los propósitos de estas prácticas, las cerraduras deberán almacenar:

1. Los parámetros formales[13].
2. El cuerpo de la función.
3. El ambiente donde fue definida la función.

El siguiente es un ejemplo de una función para obtener su cerradura:

```
{with {x 1}
  {with {y 2}
    {with {fun {lambda {param} {+ param x}}}
      f}}}
```

Código 4.6 Ejemplo de cerradura

La cerradura de la función `fun` almacenaría lo siguiente:

1. Parámetros formales: `param`
2. Cuerpo de la función: `{+ param x}`
3. Ambiente donde fue definida la función: `((x 1) (y 3))`

4.4 Ejercicios

Usaremos el lenguaje objetivo CFWAE (Condicionales, Funciones, With y Expresiones Aritméticas) con funciones de ***primera clase*** y la siguiente sintaxis descrita con EBNF:

```
<CFWAE> ::= <num>
          | {binop <CFWAE> <CFWAE>}
          | <id>
```

13 Al nombre de los argumentos de la función se le conoce como parámetro formal, mientras que a la expresión que representa el argumento suministrado a la función, se le llama parámetro real, [11].

```
          | {if0 <CFWAE>  <CFWAE>  <CFWAE>}
          | {with {{<id> <CFWAE>}*} <CFWAE>}
          | {fun {<id>*} <CFWAE>}
          | {<CFWAE> <CFWAE>*}

<binop> ::= +
          | -
          | *
          | /
```

Código 4.7 Gramática extendida de CFWAE

Lo que se debe implementar es:

1. Condicionales:
   Añadir if0 al lenguaje usando la sintaxis abstracta descrita en clase. Así se evitan añadir valores lógicos (verdadero o falso) y operadores sobre éstos. Es necesario resaltar que el if0 tiene tres ramas: una expresión de prueba, una expresión "then" la cual se evalúa si la expresión de prueba se evalúa a 0, y la expresión "else" que se evalúa en otro caso. Se debe mandar un error en caso de que la prueba no sea numérica.
   Por ejemplo:

   ```
   {if0 0 {+ 1 1} {+ 2 2}}
   ```

   se evaluará a 2, mientras que:

   ```
   {if0 {+ 2 2} 0 1}
   ```

   se debe evaluar a 1.

2. Funciones de múltiples argumentos
   Modificar el lenguaje para que una función pueda recibir una lista de cero o más argumentos, no sólo uno. Todos los argumentos de la función se evalúan en el mismo ambiente. Asume que el número de argumentos en una invocación a función es el mismo que el número en la definición del procedimiento. El intérprete debe respetar la sintaxis que se especifica arriba con EBNF para el lenguaje. Por ejemplo:
   {{fun {x y} {+ x y}} 10 3}

se debe evaluar a 13.

3. Eliminación del **with** de la sintaxis abstracta
La primitiva `with` ya no deberá ser una variante del **tipo de dato CFWAE**. Esto implica que el analizador sintáctico deberá construir la sintaxis abstracta para esta variante como la aplicación de una función anónima[14]. Por ejemplo:

```
{with {{x 1} {y 2} {z 3}} {+ x {+ y z}}}
se usará así:
(app
     (fun (list ‘x ‘y ‘z)
          (binop ‘+
                 (id ‘x)
                 (binop ‘+ (id ‘y) (id ‘z))))
      (list 1 2 3))
```

4. Analizador sintáctico
Implementar el procedimiento parse que consume una expresión en la sintaxis concreta del lenguaje y regresa la representación en sintaxis abstracta de la misma. Los tipos de datos que se usan para en **Racket** para representar expresiones **CFWAE** son que usamos listas, números y símbolos.

5. Intérprete
Implementar un intérprete para **CFWAE** con sustitución diferida (defrdSub) y alcance estático. El procedimiento que hará esto se deberá llamar interp, el cual:

- Recibe un árbol de sintaxis abstracta y un ambiente
- Regresa un **CFWAE-Value** (es decir, el tipo de datos que se debe definir para los valores del lenguaje)
- Se deberá implementar haciendo uso de ambientes procedimentales.

14 Una función anónima es aquella función que no está ligada a un identificador, [22].

Capítulo 5 Evaluación perezosa

5.1 Objetivo

- Analizar las diferencias entre la semántica de la evaluación glotona y perezosa de las expresiones en un lenguaje.
- Escribir programas sencillos extendiendo lenguaje propuesto con semántica de evaluación perezosa.
- Implementar un lenguaje con la semántica usada por lenguajes como Haskell y la línea de comandos (entre otros), lo anterior resultará útil para la resolución de problemas que la evaluación glotona no pueda tratar, como las estructuras de datos infinitas.

5.2 Definición

La evaluación perezosa se define como: "*Una estrategia de evaluación donde los parámetros en un llamada de un procedimiento no son evaluados hasta que sean usados en el cuerpo del procedimiento*"[6].

5.3 Consideraciones para implementar evaluación perezosa

Cerraduras de expresiones

En esta semántica de evaluación, no se almacenarán pares variable-valor en los ambientes, ahora se deberá guardar pares variables-expresiones. Sin embargo, estas expresiones pueden tener identificadores que ya habían sido substituidos. Este problema es similar al que se tuvo al cambiar la substitución por ambientes. La solución es similar, se usaron cerraduras para almacenar a la función con su ambiente. Ahora se deberá almacenar a todas las expresiones que no son inmediatamente reducidas a valores con el ambiente en el que fueron definidas para evitar hacer substituciones donde no se deben hacer. Nos referiremos a estos nuevas cerraduras como cerraduras de expresiones. Analicemos un ejemplo que muestre el funcionamiento de las cerraduras de expresiones usando la gramática CFWAE (ver Código 4.6).

```
1. {with {[a {+ 4 5}]}
2.   {with {[b {+ a a}]}
3.     {with {[c b]}
4.       {with {[a 4]}
5.         c}}}}
```

Código 5.1 Ejemplo de evaluación perezosa y cerraduras de expresiones

En la líneas 1 a 4, se hace uso de la primitiva `with` introducimos nuevas variables al ambiente, estas nuevas variables son a, b, c y por último otra variable a. En la línea 5 se regresa el resultado, es decir, aquel valor que tendrá la variable c cuando eventualmente sea evaluada.

Analicemos la ejecución de este código sin hacer uso de cerraduras de expresiones.

#	Análisis	Código a evaluar	Figura representativa del ambiente
1	En la línea 1 se introduce la variable **a** al ambiente.	1. `{with {[a {+ 4 5}]}`	a: `{+ 4 5}` Ambiente
2	En la línea 2 se introduce la variable **b** al ambiente.	2. `{with {[b {+ a a}]}`	b: `{+ a a}` a: `{+ 4 5}` Ambiente

#	Análisis	Código a evaluar	Figura representativa del ambiente
3	En la línea 3 se introduce la variable **c** al ambiente.	`3.{with {[c b]}`	c b b {+ a a} a {+ 4 5} Ambiente
4	En la línea 4 se introduce la segunda variable **a** al ambiente.	`4.` `{with {[a 4]}`	a 4 c b b {+ a a} a {+ 4 5} Ambiente
5	En la línea 5 se evalúa la variable **c** .	`c`	a 4 c b b {+ a a} a {+ 4 5} Ambiente
6	Se busca el valor de la variable **c** en el ambiente, es decir, el valor de la variable **b** .	b	a 4 c b b {+ a a} a {+ 4 5} Ambiente

#	Análisis	Código a evaluar	Figura representativa del ambiente
7	Se busca el valor de la variable **b**, lo que resulta en la suma de la variable **a** consigo misma	`{+ a a}`	a: 4 c: b b: {+ a a} a: {+ 4 5} Ambiente
8	Al reducir la expresión haciendo las substituciones, llegamos al resultado erróneo de 8.	`{+ 4 4}`	a: 4 c: b b: {+ a a} a: {+ 4 5} Ambiente

El problema se presenta en la fila de la tabla número 8, ya que al buscar el valor de la variable a, se tomó el valor incorrecto. Veamos el mismo ejemplo pero haciendo uso de cerraduras de expresiones a los que llamaremos `exprV`, recordemos que éstos guardan expresiones y ambientes.

#	Análisis	Código a evaluar	Figura representativa del ambiente
1	En la línea 1 se introduce la variable **a** al ambiente como una cerradura de expresión, `env0` hace referencia a un ambiente vacío, a este nuevo ambiente que contiene a a lo llamaremos `env1`.	1. `{with {[a {+ 4 5}]}`	env1: a | {+ 4 5} | env0 — Ambiente
2	En la línea 2 se introduce la variable **b** ambiente con la cerradura de expresión correspondiente, a este nuevo ambiente que contiene a a lo llamaremos `env2`. Cabe resaltar que cada nuevo ambiente extiende al anterior.	2. `{with {[b {+ a a}]}`	env2: b | {+ a a} | env1; a | {+ 4 5} | env0 — Ambiente

<table>
<tr><th>#</th><th>Análisis</th><th>Código a evaluar</th><th>Figura representativa del ambiente</th></tr>
<tr><td>3</td><td>En la línea 3 se introduce la variable c ambiente con la cerradura de expresión correspondiente, a este nuevo ambiente que contiene a <code>a</code> lo llamaremos <code>env3</code>.</td><td><code>3.{with {[c b]}</code></td><td>
<table>
<tr><td></td><td></td><td></td></tr>
<tr><td>c</td><td>b</td><td>env2</td></tr>
<tr><td>b</td><td>{+ a a}</td><td>env1</td></tr>
<tr><td>a</td><td>{+ 4 5}</td><td>env0</td></tr>
</table>
env3
Ambiente</td></tr>
<tr><td>4</td><td>En la línea 4 se introduce la segunda variable a al ambiente con la cerradura de expresión correspondiente, a este nuevo ambiente que contiene a <code>a</code> lo llamaremos <code>env4</code>.</td><td><code>4.{with {[a 4]}</code></td><td>
<table>
<tr><td>a</td><td>4</td><td>env3</td></tr>
<tr><td>c</td><td>b</td><td>env2</td></tr>
<tr><td>b</td><td>{+ a a}</td><td>env1</td></tr>
<tr><td>a</td><td>{+ 4 5}</td><td>env0</td></tr>
</table>
env4
Ambiente</td></tr>
<tr><td>5</td><td>En la línea 5 se evalúa la variable c en el ambiente <code>env4</code>.</td><td><code>c</code></td><td>
<table>
<tr><td>a</td><td>4</td><td>env3</td></tr>
<tr><td>c</td><td>b</td><td>env2</td></tr>
<tr><td>b</td><td>{+ a a}</td><td>env1</td></tr>
<tr><td>a</td><td>{+ 4 5}</td><td>env0</td></tr>
</table>
env4
Ambiente</td></tr>
</table>

<table>
<tr><th>#</th><th>Análisis</th><th>Código a evaluar</th><th>Figura representativa del ambiente</th></tr>
<tr><td>6</td><td>Al buscar la variable c en el ambiente env4, vemos que se evalúa a la variable b, que se definió en el ambiente env2, pues está contenido dentro del ambiente 4.</td><td><code>b</code></td><td><table><tr><td>a</td><td>4</td><td>env3</td></tr><tr><td>c</td><td>b</td><td>env2</td></tr><tr><td>b</td><td>{+ a a}</td><td>env1</td></tr><tr><td>a</td><td>{+ 4 5}</td><td>env0</td></tr></table>env4
Ambiente</td></tr>
<tr><td>7</td><td>Se evalúa la variable b en el ambiente env2.</td><td><code>{+ a a}</code></td><td><table><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td>b</td><td>{+ a a}</td><td>env1</td></tr><tr><td>a</td><td>{+ 4 5}</td><td>env0</td></tr></table>env2
Ambiente</td></tr>
<tr><td></td><td>A su vez, se busca la variable a en el ambiente que fue definido (env1), después de desarrollar la expresión, se llega al resultado correcto que es 18.</td><td><code>{+ {4 5} {4 5}}</code></td><td><table><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td>a</td><td>{+ 4 5}</td><td>env0</td></tr></table>env1
Ambiente</td></tr>
</table>

Puntos estrictos

"*A los puntos donde la implementación de un lenguaje perezoso fuerza la reducción de una expresión a un valor (si existe) se les llama puntos estrictos del lenguaje*"[11].

En algunos casos, se debe forzar la evaluación para seguir la interpretación. Por ejemplo, las operaciones aritméticas operan sobre valores numéricos y no sobre cerraduras de expresiones. Para poder evaluar estas operaciones aritméticas es necesario forzar la evaluación de los parámetros de las operaciones aritméticas (que son cerraduras de expresiones). Estos casos se les llama puntos estrictos. Los puntos estrictos en esta práctica sólo serán las operaciones binarias, la condicional del if0, y la primer expresión de la aplicación de función.

5.4 Ejercicios

Para la primera parte de la práctica, se usará el lenguaje de programación **Haskell**[15].

Función de Haskell	Análogo de Racket
==	eq?
<, >, <=, >=	<, >, <=, >=
mod	modulo
div	quotient
not	not
!!	list-ref
filter	filter
all	andmap
any	ormap
take	(regresa un prefijo de una lista)
takeWhile	(regresa el prefijo de una lista que satisfaga un predicado)

Tabla 1. Procedimientos importantes para esta práctica.

En la Tabla 1 se muestran procedimientos útiles para la realización de esta práctica. Todo procedimiento cuyo nombre comienza con un carácter simbólico, es un operador infijo (por ejemplo: `[1, 2, 3] !! 2`). Los otros procedimientos binarios se pueden utilizar como operadores infijos si se encierran entre apóstrofes inversos (por ejemplo. `x 'mod' y`). También, los operadores infijos se pueden utilizar como procedimientos en notación prefija, si se encierran entre paréntesis (p. ej. `(!!) [1, 2, 3] 2`).

[15] En el Apéndice A se encuentra información detallada sobre cómo instalar un intérprete de Haskell.

Haskell permite la definición de nuevos tipos mediante la declaración `data`, la cual tiene un comportamiento similar al procedimiento `define-type` de Racket. A continuación se ejemplifica con el tipo predefinido `Bool` de Haskell:

```
data Bool = False | True
```

El tipo que se define es llamado `Bool` y tiene exactamente dos valores, `False` y `True`. Es necesario resaltar que Haskell requiere que los nombres tanto de los tipos como de los constructores empiecen con letra mayúscula.

También es posible definir tipos recursivos, para ejemplificarlo, definamos árboles binarios que almacenen números enteros y llamemos a este nuevo tipo Arbol:

```
data Arbol = Hoja Int
                  | Nodo Int Arbol Arbol
                  deriving (Show, Eq)
```

El tipo `Arbol` esta constituido de dos variantes, `Hoja` y `Nodo`. La variante `Hoja` únicamente almacena un entero mientras que la variante `Nodo` además de almacenar el entero, tiene dos referencias al tipo `Arbol` que estamos definiendo. El renglón `deriving (Show, Eq)` nos permite convertir el tipo `Arbol` a tipo cadena y probar la igualdad de los árboles.

Se deberán implementar los siguientes ejercicios:

1. Haciendo uso del tipo `Arbol` definido anteriormente, definir el procedimiento
   `mapArb :: Arbol -> (Int -> Int) -> Arbol`
   Debe regresar el árbol resultante de aplicar el procedimiento recibido como segundo parámetro a cada elemento del árbol.

   Ejemplo:

   ```
   Main> mapArb (Nodo 1 (Hoja 2) (Hoja 3)) (\x -> x + 1)
   (Nodo 2 (Hoja 3) (Hoja 4))
   ```

   En este ejemplo, la función pasada al procedimiento `mapArb`, `(\x -> x + 1)`, es una función sin nombre, es decir, sólo puede ser usada por `mapArb`. También es posible definir una función de tipo `(Int -> Int)` con nombre y pasar dicho identificador a `mapArb`.

2. Definir el procedimiento `conservaDivisores :: int -> [int] -> [int]`
   Debe regresar una lista que conserve todos aquellos números que sean divisibles por el primer parámetro.

Ejemplo:

```
Main> conservaDivisores 2 [1..10]
[2 4 6 8 10]
```

3. Definir el procedimiento `listaRaicesExactas:: Int -> Int -> [Int]`
Debe regresar la lista de números enteros cuya raíz cuadrada sea exacta, en el intervalo cuyo mínimo será el primer entero y el máximo sea el segundo entero.
Ejemplo:

```
Main> listaRaicesExactas 1 10
[1 4 9]
```

4. Definir la función `Lucas :: [Integer]`
Regresa la lista infinita de la sucesión de `Lucas`, la sucesión se define para `n >= 0`
Lucas(0) = 2
Lucas(1) = 1
Lucas(n) = lucas(n-1) + lucas(n-2)

Para la segunda parte de la práctica, usaremos la versión perezosa del intérprete de Racket.

Un ejemplo de un programa en Racket perezoso es el siguiente:

```
1. #lang lazy
2. (define (factorial n)
3.   (if (eqv? 0 n) 1 (* n (factorial (sub1 n)))))
4. (let ([n 5]
5.       [nunca (factorial 1000000000)])
6.   (! (* n n)))
```

Código 5.2 Ejemplo de `Racket` perezoso

Para hacer uso de Racket perezoso, es necesario cambiar la primer línea del programa a `#lang lazy`. En las líneas `2` y `3` se define la función factorial mientras que en las líneas `4` y `5` se definen dos variables `n` y `nunca`. El valor de `n` es la expresión 5 mientras que el valor de `nunca` es la expresión factorial de `1,000,000,000`. En la línea `6` se usa el procedimiento `!` para forzar la evaluación de la multiplicación de `n` por `n` y es hasta este momento que el intérprete calcula el valor de `n`.

Al ejecutar este código, el intérprete regresará el valor `25`. Cabe resaltar que la variable `nunca` no fue usada en el cuerpo del `let` (línea 6), por lo que el intérprete no calculó su valor.

Ahora bien, evaluemos este mismo código en Racket glotón, para ello, es necesario cambiar la primera línea del código de `#lang lazy` a `#lang racket`, lo cual tendría el siguiente comportamiento:

Análogamente a la versión perezosa, las líneas `2` y `3` definen la función factorial. Sin embargo, en la línea 4 le asigna a `n` el valor de 5 y a `nunca` el valor de factorial de `1,000,000,000`. Al ejecutar el código veríamos que al intérprete le llevaría mucho tiempo regresar el valor 25 debido a que debe primero calcular el valor de la variable `nunca`. En la versión perezosa esto no sucede, ya que, al no usar la variable `nunca`, el valor no es calculado por el intérprete y simplemente realiza la multiplicación de `n` por `n`.

5. Escribir un programa en Racket perezoso que entregue un resultado correcto o esperado y que ese mismo programa no pueda ser evaluado correctamente en Racket con evaluación glotona. Explicar brevemente en un comentario en el código fuente del programa el motivo por el que no es posible que el programa funcione, como se hizo en la explicación del Código `5.2`.

6. Para la tercera parte, se deberá modificar el intérprete de la práctica anterior para que use semántica de aplicación perezosa en lugar de glotona. Deberías poner especial atención en las siguientes reglas y aplicar las cerraduras de expresiones como corresponda:

   - Condicional del if.
   - Operaciones binarias.
   - Aplicación de función.

   Incluir procedimientos parse y test como en la Práctica 3.

   La gramática que debe aceptar el analizador sintáctico, en notación EBNF, es:

```
<CFWAE> :: = <num>
           | {binop <CFWAE> <CFWAE>}
           | <id>
           | {if0 <CFWAE>  <CFWAE>  <CFWAE>}
           | {with {{<id> <CFWAE>}*} <CFWAE>}
           | {fun {<id>*} <CFWAE>}
```

```
                | {<CFWAE> <CFWAE>*}

<binop> :: = +
             | -
             | *
             | /
```

Código 5.3 Gramática `CFWAE` perezosa

Esta tercera parte se podrá implementar en el lenguaje Haskell o Racket.

Capítulo 6 Estado

6.1 Objetivos

- Analizar el concepto de estado e implementar un intérprete que ofrezca construcciones mutables haciendo uso de cajas.
- Añadir construcciones mutables a un intérprete mediante cajas para implementar el concepto de estado.

6.2 Mutación y cajas

La mutación es "*el cambio de los valores asociados mediante los nombres*"[11], ésta se encuentra como una característica en la mayoría de los lenguajes de programación. Si se hiciera un programa para modelar el mundo real, entonces dichos programas necesitarán ajustar el hecho de que los eventos del mundo real lo cambian y alteran. Por ejemplo, un programa que modelase el consumo de gasolina de un carro, podría utilizar el concepto de estado para recordar los cambios en el tanque de gasolina; es decir, cuando el carro haya recorrido una cantidad suficiente de kilómetros, la cantidad de gasolina en el tanque debe disminuir. De manera contraria, si se lleva el carro a la gasolinera y se llena el tanque, la cantidad debe aumentar. Con este ejemplo podemos observar que se está alterando el valor del objeto mismo y además no se están construyendo "nuevos tanques" por cada cambio que se desee.

Cajas

Las cajas son una estructura de datos capaces de almacenar un único elemento en ellas. Los procedimientos para operarlas son `box`, `unbox` y `set-box!`
Veamos un ejemplo donde se usen tales procedimientos:

```
1.(let ([caja (box 3)])
2.  (begin
3.    (set-box! caja 9)
4.  (unbox caja)))
```

Código 6.1 Ejemplo de las primitivas `box`, `unbox` y `set-box!`

En la línea `1` usamos `box` (quien es el constructor de cajas) para crear una caja con el valor `3` (vale la pena recalcar que podemos almacenar cualquier tipo de dato). En la línea 3 usamos `set-box!` para mutar el valor que tiene la caja, es decir, cambiamos su valor, este procedimiento recibe una caja (o la variable de la `caja` en el ejemplo) y cambia el valor de `3` a `9`. Por último en la línea `4`, abrimos la caja para recuperar el valor que se haya almacenado en ésta.

En la línea `2` se usa el procedimiento `begin` que evalúa las expresiones pasadas en ese orden y todos los resultados son ignorados salvo el último. Este último resultado es el valor de retorno. Debido a este comportamiento, la primitiva `begin` regresa la llamada a `unbox`.

6.3 Técnica de paso de repositorio de valores (store passing-style)

El store passing-style es una técnica para implementar el concepto de estado y será la que utilicemos en nuestro intérprete. Se usará esta técnica y no las cajas de Racket ya que el store passing-style nos permitirá analizar a detalle como se implementa el concepto de estado desde cero.

En las prácticas pasadas hemos usado un único repositorio para almacenar variables, los cuales hemos llamado ambientes. En esta práctica haremos uso de un segundo repositorio, que por razones históricas llamaremos store, [6].

Se deben agregar algunas modificaciones al intérprete, ahora en vez de almacenar en los ambientes pares variable-valor, guardaremos los pares variable-ubicación, donde ubicación hace referencia a la posición en el store en el cual estará almacenada dicha variable. Por otra parte, el store almacenará los pares ubicación-valor. Si se desea hacer una operación que mute el estado de una caja, en vez de borrar la entrada del store, se deberá agregar una

nueva, usando el mismo número de localidad. La forma de operar el store es igual a la de los ambientes, donde ambos presentan comportamiento de una pila.

Por último, se debe modificar el valor de regreso del intérprete para que no sólo regrese el valor de cada expresión sino también un store actualizado con los cambios que se pueden hacer debido a las mutaciones en el proceso para calcular el valor. A este nuevo tipo de regreso lo llamaremos ValueXStore, pues es un valor pasado por el store.

6.4 Ejemplos

En esta práctica usaremos el lenguaje objetivo BCFWAE (Boxes - cajas, Condicionales, Funciones, With y Expresiones Aritméticas) con funciones de primera clase. La siguiente sintaxis descrita con EBNF lo expresa.

```
<BCFWAE> :: = <num>
            | {binop <BCFWAE> <BCFWAE>}
            | <id>
            | {if0 <BCFWAE>  <BCFWAE>  <BCFWAE>}
            | {with {{<id> <BCFWAE>}*} <BCFWAE>}
            | {fun {<id>*} <BCFWAE>}
            | {<BCFWAE> <BCFWAE>*}
            | {box <BCFWAE>}
            | {unbox <BCFWAE> <BCFWAE>}
            | {set-box! <BCFWAE> <BCFWAE>}
            | {seqn <BCFWAE> <BCFWAE>}

<binop> :: = +
           | -
           | *
           | /
```

Código 6.2 Gramática extendida BCFWAE

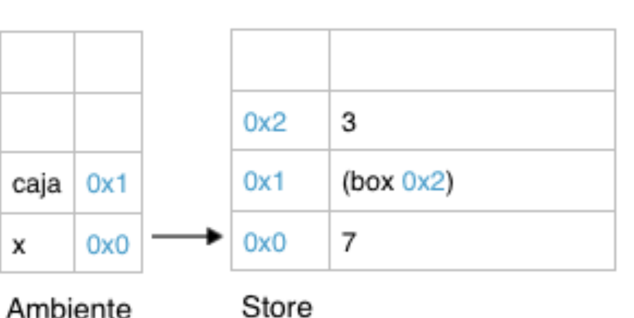
El siguiente código es un ejemplo de un programa escrito con la gramática BCFWAE que mutará el valor de una variable haciendo uso de una caja:

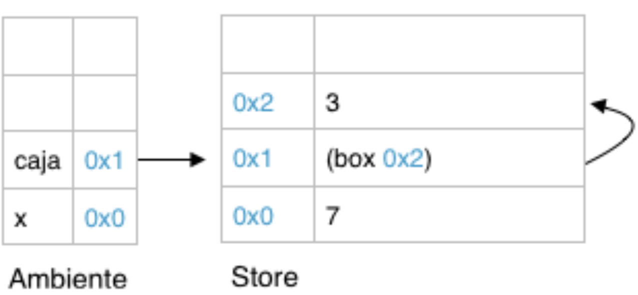
```
1.{with {[x 7]
2.        [caja {box 3}]}
3.  {seqn
4.    {set-box! caja
5.        {+ x
6.        {unbox caja}}}
7.  caja}}
```

Código 6.3 Ejemplo BCFWAE

A continuación se explica la ejecución del ejemplo.

Descripción	Figura representativa del ambiente y la memoria
En las líneas 1 y 2, el `with` introduce dos nuevas variables tanto al ambiente como en el store. La variable **x** es guardada en el ambiente junto con la referencia a la localidad 0x0 del store. La variable `caja` tiene la referencia a la localidad 0x1 del store. En dicha localidad, la caja almacena una referencia a otra localidad, la número 0x2, donde se encuentra alojado el contenido de la caja, el número 3.	Ambiente: caja → 0x1; x → 0x0 Store: 0x2 → 3; 0x1 → (box 0x2); 0x0 → 7

<table>
<tr><th>Descripción</th><th>Figura representativa del ambiente y la memoria</th></tr>
<tr><td>En la línea 3 se hace uso de la primitiva <code>seqn</code>, su comportamiento es similar al de <code>begin</code> de Racket, es decir, evalúa ambas instrucciones regresando únicamente el resultado de la segunda. Cabe destacar que si en la primer instrucción se altera el estado de alguna caja, dicho cambio debe verse reflejado al momento de evaluar la segunda instrucción.</td><td>
</td></tr>
<tr><td>En la línea 4, se usa el procedimiento <code>set-box!</code>, que se encarga de mutar el contenido de una caja (la cual se pasó como el primer parámetro) con un nuevo valor, (el cual fue pasado como segundo parámetro). Este nuevo valor será almacenado por la caja.

La parte importante de esta operación se presenta cuando se evalúa el identificador caja. Primero se busca en el ambiente el identificador para obtener su posición en el store, que en este ejemplo es la localidad <code>0x1</code>. Una vez que se obtiene la posición, se debe buscar en el <code>store</code> su contenido, que en este caso es una caja. A su vez, la caja contiene una referencia a otra posición del mismo store, en este ejemplo es la localidad <code>0x2</code>. Finalmente, en la localidad <code>0x2</code> se encuentra el valor que almacena la caja, es decir, el valor de 3.</td><td>
</td></tr>
</table>

Descripción	Figura representativa del ambiente y la memoria
En la línea `5`, se usa la operación binaria `+`. Por convención, para ésta (y todas las operaciones binarias) se evaluará primero el lado izquierdo, y con el store resultante se evaluará el lado derecho. En este ejemplo no se puede observar este comportamiento porque la rama izquierda de la suma no realiza operaciones de estado y simplemente se evalúa al identificado **x**.	
En la línea `6`, se hace uso de la primitiva `unbox`, cuya finalidad es regresar el valor que contiene una caja. Como ya se ha visto anteriormente, el valor que contiene la caja está en la posición `0x2` del store la cual tiene el valor de `3`.	
Para lograr el comportamiento de mutación, el procedimiento `set-box!` añadirá una nueva entrada en el store con el nuevo valor, usando como ubicación "la misma referencia" (`0x2`) para que sea este nuevo registro el que sea tomado en cuenta, debido al comportamiento a manera de pila que tiene store.	Ambiente: caja 0x1; x 0x0 Store: 0x2 10; 0x2 3; 0x1 (box 0x2); 0x0 7
Por último, en la línea `7` se regresa la variable **caja**, es decir, una caja cuyo valor apunta a la referencia `0x2` del store que tiene el valor de `10`.	

Con el fin de analizar a fondo el comportamiento de las cajas del intérprete que se implementará en esta práctica, comparémoslo con el las cajas de DrRacket. Adaptando el código de BCFWAE a la sintaxis de Racket, tenemos:

```
1.(let ([x 7]
2.      [caja (box 3)])
3.  (begin
4.    (set-box! caja
5.        (+ x
6.        (unbox caja)))
7.   caja))
```

Código 6.4 Conversión del Código 6.3 a sintaxis de `Racket`

A lo que DrRacket responde con `'#&10` donde `'#&` significa que es una caja y `10` es el valor que la caja está almacenando en memoria. En nuestro intérprete bastará con regresar una representación de las cajas haciendo uso de un nuevo tipo creado usando los procedimientos de PLAI.

6.5 Ejercicios

Usando el lenguaje objetivo BCFWAE (ver el Código 6.2), se debe implementar:

1. ValueXStore:
   Para poder implementar estado, será necesario hacer las modificaciones necesarias a los ambientes e implementar los stores descritos anteriormente. Aunado a eso, este intérprete tiene la singularidad que los valores de regreso son de tipo de ValueXStore, es decir, al evaluar una expresión se debe regresar una estructura que contenga tanto el valor de dicha expresión como el store que posiblemente se modificó al reducir dicha expresión.

2. Cajas:
Añadir las siguientes primitivas al tipo de dato usando la sintaxis abstracta descrita en la gramática del Código 6.2:

- box recibe lo que se desea guardar y crea una nueva localidad en store `(box 3)`.
- unbox recibe una caja y regresa el valor que almacena `(unbox caja)`.
- set-box! recibe una caja y un nuevo valor a almacenar. En Racket, set-box no regresa nada pues únicamente realiza la asignación en el ambiente y en el store, sin embargo en esta práctica regresaremos el nuevo valor almacenado en la caja para evitar regresar valores Void[16]. `(set-box! caja 5)`.

3. seqn:
Esta primitiva recibirá dos expresiones, ejecuta ambas y regresa el resultado de evaluar la segunda. Cualquier cambio en el estado de las variables hecho en la primera expresión se debe ver reflejado al momento de empezar a ejecutar la segunda expresión.

[16] Void “es el tipo del resultado de una llamada a función pero no provee un valor de retorno a quién la invocó”[13].

Capítulo 7 Recursión

7.1 Objetivo

- Construir un intérprete que ofrezca construcciones recursivas haciendo uso de contenedores mutables.

7.2 Tipos de recursión

Al hablar de recursión, podemos referirnos a ella bajo dos contextos, recursión sobre algún tipo de dato o recursión como control, analicemos a qué se refiere cada una.

Datos recursivos y cíclicos

Los datos recursivos los podemos clasificar en dos, datos que hacen referencia a otros elementos del mismo tipo o datos que hacen referencia a sí mismos.

A la recursión que se emplea sobre datos que hacen referencia a otros elementos del mismo tipo es lo que conocemos como recursión sobre algún tipo. Por ejemplo, un árbol es una estructura de datos recursiva, cada vértice puede tener múltiples hijos donde cada uno de ellos es por sí mismo un árbol. Si escribiéramos un procedimiento que hiciera un recorrido sobre el árbol, esperaríamos que terminara sin la necesidad de recordar que vértice hemos ya visitado, [11].

En contraste, las gráficas son estructuras de datos cíclicas, donde un nodo refiere a otro, que a su vez podría referir de regreso al primero (o para tal caso, referirse a sí mismo). Si escribiéramos un procedimiento para recorrer una gráfica, sin algún método explícito para verificar los nodos que ya hayamos visitado, esperaríamos que el procedimiento divergiera (no terminara). Los algoritmos para recorrer gráficas necesitan memoria de los nodos que ya hayan sido visitados para evitar así recorridos repetidos, [11].

En los pequeños lenguajes que hemos implementado, añadir tipos de datos recursivos es directo, ya que simplemente necesitamos dos cosas:

1. Una primitiva para crear estructuras compuestas (por ejemplo, nodos que hacen referencia a nodos hijo en un árbol).
2. La habilidad de llegar al caso base de la recursión (por ejemplo, hojas en un árbol).

En cambio, el añadir tipos de datos cíclicos no resulta tan directo ya que no se puede definir un caso base. Tomemos como ejemplo un tipo de dato que haga referencia a sí mismo, el código que nos gustaría hacer sería:

```
1.(let ([b b])
2.  b)
```

Código 7.1 Ejemplo de referencia a sí mismo

Si ejecutáramos el código nos regresará un error que especificará que b no ha sido definido. Para poder lograrlo, utilizaremos la siguiente estrategia de tres pasos:

1. Nombrar un contenedor vacío. (En Racket usaremos cajas)
2. Mutar el contenedor para que su contenido sea él mismo.
3. Para obtenerlo, usar el nombre definido previamente.

Para poder llevar a cabo esta estrategia, usaremos cajas de Racket vistas anteriormente, sin embargo, no es posible construir una caja vacía. La estrategia a seguir será la de construir una caja con un elemento cualquiera, para después reemplazarlo.

```
1.(let ([b (box ‘dummy)])
2.  (begin
3.    (set-box! b b)
4.    b))
```

Código 7.2 Ejemplo de estrategia de tres pasos

Al ejecutar este código, Racket regresará `#0='#�#`, que es la representación que buscamos. Los caracteres `&#` son la forma en que Racket imprime las cajas. A su vez, `#0` (e igualmente con otros números) es la manera en que Racket maneja los datos cíclicos. Lo que nos dice Racket es que `#0` está ligada a una caja cuyo contenido es `#0`, es decir `#0` mismo.

7.3 Funciones recursivas

Una función recursiva hace referencia a la misma función en sí, [11]. Es necesario tener condicionales para poder escribir programas que terminen (es decir, que no diverjan), en el caso de estas prácticas usaremos `if0`.

Implementemos ahora la función recursiva factorial para un entero no negativo n que está definida como la multiplicación de todos los números enteros positivos menores o iguales a n.

Observemos el siguiente código en Racket:

```
1. (let ([ fact (lambda (n)
2.                  (if (= n 0)
3.                       1
4.                       (* n (fact (- n 1))))))])
5.         (fact 5))
```

Código 7.3 Ejemplo de la función factorial sin la estrategia de tres pasos.

Sin embargo, este código aunque es sintácticamente correcto, no lo es semánticamente, en otras palabras, no funciona en lo absoluto, ya que el identificador `fact` interior queda libre de la misma manera que el tipo de dato cíclico del ejemplo.

Analicemos una ejecución del mismo código escrita con la sintaxis de la gramática de la sintaxis BCFWAE del Código 6.2:

```
1. (with ([ fact (fun (n)
2.                  (if0 n
3.                       1
4.                       (* n (fact (- n 1))))))])
5.     (fact 5))
```

Código 7.4 Ejemplo de factorial usando la gramática BCFWAE

Para hacer el análisis de esta llamada a función, veamos la sección del código que se encarga de interpretar una aplicación de función:

```
1. [app (fun-expr arg-expr)
2.      (let ([fun-val (interp fun-expr ds)]
3.            [arg-val (interp arg-expr ds)])
4.        (interp (closureV-body fun-val)
5.                (aSub (closureV-param fun-val)
6.                      arg-val
7.                      (closureV-env fun-val))))]
```

Código 7.5 Código de la aplicación de función BCFWAE (Código 6.2)

Analicemos la ejecución de la función `fact` (ver Código 7.4) haciendo uso del código de aplicación de función (Código 7.5) del lenguaje BCFWAE (Código 6.2)

<table>
<tr><th>#</th><th>Descripción</th><th>Código a analizar</th><th>Figura representativa del ambiente</th></tr>
<tr><td>1</td><td>Se empieza la ejecución.
El ambiente se encuentra vacío.</td><td><pre>(with(
   [fact
    (fun (n)
     (if0 n 1
       (* n
          (fact
            (- n 1))
              )))
    ])
   (fact 5))</pre></td><td>Ambiente</td></tr>
<tr><td>2</td><td>La primitiva <code>with</code> inserta al ambiente la variable fact, con el resultado de interpretar la primitiva <code>fun</code>, es decir, la cerradura que tiene a n como parámetro formal, al cuerpo de la función y al ambiente donde fue definida la función, en este caso el ambiente vacío (al que llamamos <code>mtSub</code>)</td><td><pre>(fact 5)</pre></td><td>fact | <code>(closureV 'n (if0 n 1 (* n (fact (- n 1))))</code>
Ambiente</td></tr>
</table>

#	Descripción	Código a analizar	Figura representativa del ambiente

3	Al evaluar el identificador **fact**, se busca en el ambiente y se obtiene la cerradura. Para seguir con la ejecución, analicemos el Código 7.5.
	
	A los parámetros formales `fun-expr` y `arg-expr` les hemos pasado los parámetros reales **fact** y **5** respectivamente. La variable `fun-val` tendrá como valor la interpretación de `fact` en el ambiente, es decir, la cerradura, mientras que la variable `arg-val` tendrá la interpretación de **5**. El siguiente paso es interpretar el cuerpo de la cerradura, en un nuevo ambiente. Este nuevo ambiente será aquel que se encontraba en la cerradura, extendiéndolo con los parámetros reales y formales de la función **fact**, es decir, el ambiente vacío se extenderá con **n** y **5**.
		`(if0 n`
		`     1`
		`     * n`
		`       (fact`
		`         (- n 1)))`
			n 5
			Ambiente

4	En este punto se puede ver el problema, al terminar de evaluar la primitiva `if0`, se buscará al identificador **fact** el cual ya no se encuentra en el ambiente, por lo que el intérprete mandará un error al buscar el identificador en el ambiente	`(fact (- n 1))`	n 5 Ambiente

Ahora implementémoslo usando la estrategia de 3 pasos en Racket:

```
1. (let ([fact (box 'dummy)])
2.   (begin
3.     (set-box! fact
4.               (lambda(n) (if (= n 0) 1
5.                              (* n ((unbox fact) (sub1 n))))))
6.     ((unbox fact) 5)))
```

Código 7.6 Ejemplo de factorial implementado con la estrategia de tres pasos

Este nuevo código da como resultado `120` que es el resultado de factorial de `5`. Lo importante de este ejemplo es la forma en que se usa el identificador `fact`, ya que, al ser una caja, no es posible usarlo directamente, es necesario abrir la caja para aplicar la función.

Debido a que `fact` tiene una referencia a él mismo gracias a la estrategia de 3 pasos, las llamadas recursivas se resuelven de manera adecuada, es por ello que este código funciona correctamente.

7.4 Ejercicios

En esta práctica usaremos el lenguaje objetivo RCFWAE (Recursión, Condicionales, Funciones, With y Expresiones Aritméticas) con funciones de primera clase y la siguiente sintaxis descrita con EBNF:

```
<RCFWAE> :: = <num>
          | {binop <RCFWAE> <RCFWAE>}
          | <id>
          | {fun {<id>*} <RCFWAE>}
          | {if0 <RCFWAE>  <RCFWAE>  <RCFWAE>}
          | {rec {<id> <RCFWAE>} <RCFWAE>}
          | {with {{<id> <RCFWAE>}*}
          | {<RCFWAE> <RCFWAE>*}

<binop> :: = +
           | -
           | *
           | /
```

Código 7.8 Sintaxis de RCFWAE

La nueva primitiva que se debe agregar a la gramática que vamos a implementar es `rec`, analicemos un ejemplo:

```
{rec {fac {fun {n}
            {if0 n
                  1
                  {* n {fac {+ n -1}}}}}}
    {fac 5}}
```

Es decir, recibe el identificador (`id`) del procedimiento para definir la construcción recursiva (`fac` en el ejemplo), junto con una expresión donde se definirá (`fun` en el ejemplo).
Por último, el cuerpo donde se evaluará la construcción recursiva (`{fac 5}` continuando con el ejemplo).
Esta primitiva debe tener el comportamiento mostrado en el Código 7.7, sin embargo, la manipulación de la caja (añadir elementos, extraerlos y modificarlos) deberá ejecutarse por el intérprete y no por el usuario.

Capítulo 8 Sistema verificador de tipos

8.1 Objetivo

- Analizar los conceptos de tipo y sistema de tipos, ya que permiten encontrar errores en algún programa sin siquiera ejecutarlo así como llevar a cabo ciertas optimizaciones, por nombrar algunos beneficios.
- Implementar un intérprete que haga uso de un sistema de tipos.
- Extender la gramática previa para hacer uso de tipos de manera explícita
- Construir un sistema verificador de tipos para verificar que se realice un uso correcto de las declaraciones con tipos

8.2 Definición de tipo y sus características

"Un tipo es cualquier propiedad de un programa que podemos establecer sin la necesidad de ejecutar dicho programa. En particular, los tipos capturan la intuición de poder predecir la conducta de un programa sin ejecutarlo. Sin embargo, dado un lenguaje de programación de propósito general, no podemos predecir su conducta en su totalidad sin ejecutarlo (pensemos en la entrada que un usuario podría insertar al mostrarle un diálogo, por ejemplo). Debido a esto, cualquier predicción de la conducta de un programa debe ser, necesariamente, una aproximación de lo que sucede. Convencionalmente se usa el término tipo para referirnos no únicamente a una aproximación, sino a una ***abstracción de un conjunto de valores****.*

Un tipo etiqueta toda expresión en el lenguaje, marcando en qué clase de valor resultará evaluar dicha expresión. Esto es, los tipos describen invariantes que se mantienen para todas las ejecuciones de un programa. Ellos aproximan esta información guardando únicamente la clase del valor que resulte la evaluación de la expresión, no el valor específico"[12].

8.3 Sistema verificador de tipos

"*Un sistema de tipos es un método sintáctico para probar la ausencia de ciertos comportamientos del programa mediante la clasificación de las frases de acuerdo con las clases de valores que computan"*[12].

Un sistema de tipos lo podemos ver como un conjunto de reglas que etiquetan toda expresión del lenguaje en una cierta abstracción de los valores que existen.

"Diseñar un sistema de tipos involucra encontrar un balance adecuado entre dos fuerzas que compiten entre sí:

1. *Tener más información hace posible hacer mejores y más ricas conclusiones acerca del comportamiento de un programa, de este modo se pueden rechazar menos programas válidos o permitir menos programas con errores semánticos.*
2. *Obtener más información es difícil, ya que:*
    a. *Podría añadir restricciones inaceptables al lenguaje de programación*
    b. *Podría incurrir en un mayor costo computacional*
    c. *Se podría obligar al usuario a realizar anotaciones de partes de un programa. Muchos programadores (a veces injustamente) se resisten a escribir todo aquello distinto del código ejecutable, por lo que pueden ver las anotaciones como una actividad pesada.*
    d. *En última instancia, se pueden rozar los límites de la computabilidad, una barrera insuperable. (A menudo, los diseñadores pueden superar esta barrera cambiando ligeramente el problema, aunque esto mueve la tarea en una de las tres categorías anteriores.)"*[11].

8.4 ¿Por qué tipos?

"Los tipos forman una muy valiosa primera línea de defensa contra los errores de los programas. Sin embargo, un sistema de tipos pobremente diseñado puede ser bastante frustrante: programar en Java a veces tiene este sabor. Un sistema de tipos poderoso como el de ML, sin embargo, es un placer usarlo.

Los tipos que no han sido subvertidos (como las conversiones explícitas de tipo que han hecho en Java, por ejemplo) juegan varios roles importantes:"[11].

- Un sistema de tipos ayuda a reducir el tiempo invertido en la depuración de un programa, ya que son capaces de detectar errores que en primera instancia pudieran pasar desapercibidos, [11].
- Los sistemas de tipos detectan errores en un código que no ha sido ejecutado por el programador, es decir, en tiempo de compilación. Esto es importante, ya que el programador tiene las herramientas necesarias para depurar el código sin siquiera llegar a la fase de pruebas, o peor aún, de producción. Sin embargo, satisfacer los requerimientos del sistema de tipos puede hacer que algunos programadores lleven a

cabo una fase de pruebas más corta o incompleta, una consecuencia de lo más indeseada y desafortunada al respecto, [11].

- Los tipos ayudan a documentar el programa. Las declaraciones de tipo explícitas, hacen que el programador obtenga una idea más cercana del comportamiento del código, ya que toda expresión en el código tiene asignada una abstracción de valores.
- Los compiladores pueden explotar los tipos para hacer que los programas se ejecuten más rápido, consuman menos espacio y se gaste menos tiempo en la recolección de basura, [11].
- Mientras que ningún lenguaje puede eliminar arbitrariamente el código que no siga las convenciones del lenguaje, un sistema de tipos impone una cierta línea base a seguir, pudiendo prevenir al menos algunos programas verdaderamente impenetrables - o, al menos, prohibe ciertos tipos de terribles estilos de código, [11].

8.5 Ejercicios

Implementaremos ahora un verificador de tipos para el lenguaje objetivo CFWAEL (Condicionales, Funciones, With, Expresiones Aritméticas y Listas) con funciones de primera clase. El lenguaje tiene la siguiente sintaxis en EBFN:

```
<CFWAEL>  ::= <num>
           | <char>
           | <string>
           | true
           | false
           | {<binop> <CFWAEL> <CFWAEL>}
           | {boolean? <CFWAEL>}
           | {char? <CFWAEL>}
           | {number? <CFWAEL>}
           | {list? <CFWAEL>}
           | {string? <CFWAEL>}
           | <id>
           | {if <CFWAEL> <CFWAEL> <CFWAEL>}
           | {with {[<id> : <type> <CFWAEL>]+} <CFWAEL>}
           | {with* {[<id> : <type> <CFWAEL>]+} <CFWAEL>}
           | {fun {[<id> : <type>]+} <CFWAEL>}
           | {lcons <CFWAEL> <CFWAEL>}
           | lempty
           | {lcar <CFWAEL>}
           | {lcdr <CFWAEL>}
           | {<CFWAEL> <CFWAEL>*}
```

```
<binop> ::= +
          | -
          | *
          | /
          | =
          | <
          | >
          | and
          | or
          | string-append
```

```
<type> ::= number
         | char
         | boolean
         | string
         | (listof <type>)
         | ([<type> ->]+ <type>)[17]
```

Código 8.1 Sintaxis de la gramática `CFWAEL`

Los cambios en cuanto a la sintaxis respecto a la prácticas anteriores se presentan en la regla para **with**, **with*** y para **fun**, ahora se espera que después del identificador se encuentre el caracter "**:**" seguido del tipo que tendrá dicha variable.

Los tipos existentes están descritos en EBNF definidos como <type>. Los tipos **number**, **char**, **boolean** y **string** deben resultar familiares en esta parte del curso y el tipo lista estará dado por la variante **(listof <type>)**. La idea de estas listas es construirlas homogéneas del tipo correspondiente. Se debe prestar especial atención cuando se utilicen "listas de listas". El otro tipo sirve para la representación de funciones, el cual opera de manera similar a la del lenguaje de programación *Haskell*, por ejemplo, **(number -> boolean)** representa una función de un único parámetro de tipo numérico y un valor de regreso de tipo lógico (booleano)[18].

El objetivo de esta práctica es escribir el procedimiento `type-of` que recibirá la sintaxis abstracta de un programa (es decir, lo que regresa el procedimiento `parse`) y regresará el tipo al que se evalúa el programa en caso de no haber ningún error. Si lo hubiera, se debe informar del error de tipo encontrado.

---

[17] En este caso, los corchetes no forman parte de la sintaxis, son meta-caracteres para indicar que se espera al menos uno.

[18] En la descripción se hace abuso de notación para poder construir funciones de n parámetros, es decir, construcciones de la forma:
(<type> -> <type> -> <type> -> ... -> <type>)

Las verificaciones que debe hacer `typeof` son:

1. **binop**
   Dependiendo de la función binaria en cuestión, se debe verificar que los parámetros sean del tipo adecuado. En caso de **+**, **-**, *****, **/** e **=** se esperan parámetros de tipo number, en el caso del **and** y **or** parámetros de tipo lógico (booleanos) y finalmente para la función **string-append** parámetros de tipo cadena.
2. **Predicados** (`boolean?`, `char?`, `number?`, `list?`, `string?`)
   Estos predicados verifican que la expresión pasada como argumento sea del tipo correspondiente y su valor de regreso se evalúe en tipo lógico
3. **if**
   La primitiva if consta de tres componentes, la parte condicional, la rama a ejecutar cuando se cumple la condicional y la rama a ejecutar cuando no se cumple la condicional. Se debe verificar que la condicional se evalúe a un tipo lógico
4. **cons**
   Para construir listas homogéneas, tomaremos la convención de que el primer elemento será quien indique el tipo de los demás, es decir, si la cabeza de la lista es de tipo number, la lista deberá tener solamente números y se debe enviar un error en cualquier otro caso.
5. **car** y **cdr**
   Se deberá verificar que sus argumentos sean de tipo lista.
6. **fun**
   A diferencia de las prácticas anteriores, ahora **fun** recibe el nombre del parámetro formal seguido del separador "**:**" y por último el tipo que tendrá asociado. La información del tipo de cada parámetro deberá ser guardada por si se llegase a aplicar la función.
7. **with** y **with***
   Se debe verificar que cada parámetro se evalué con el mismo tipo con el que fue declarado, en otro caso se debe mandar error.
8. **Aplicación de función**
   Lo primero que se debe verificar es que el primer valor sea de tipo función, si no lo fuera entonces se debe enviar error. Una vez que se sabe que es una función, se debe verificar que el tipo del parámetro real, efectivamente sea el tipo declarado por el parámetro formal en la definición de la función.

Ejemplos:
Dado que los valores de regreso de verificador de tipos son los tipos en sí (o errores) y no los valores de las expresiones, se debe crear un tipo de dato que represente los tipos de esta gramática. En los ejemplos, se usará la convención de anteponer una "t" al tipo de dato en cuestión, es decir, `(tnum)` quiere decir que la expresión se evaluaría correctamente a tipo numérico.

```
>(define (prueba exp) (type-of (parse exp)))

> (prueba ‘{+ 1 3})
(tnum)

> (prueba ‘{+ 1 true})
“Error: En + se esperaban argumentos de tipo number”

> (prueba ‘{cons 1 {cons 2 {cons 3 empty}}})
(tlistof number)

> (prueba ‘{cons {fun {[a : number]} {+ a a}} {cons 2 empty}})
“Error: Se esperaba que todos los elementos fueran de tipo (number ->
number)”
>(prueba ‘{{fun {[a : number]
                 [b : number]}
                      {+ a b}}
                  3
                 true})

“Error: Se esperaba que el parámetro b fuera de tipo number”
```

Capítulo 9 Conceptos de Orientación a Objetos

9.1 Objetivo

- Analizar los conceptos más importantes de la orientación a objetos, implementar un intérprete que los ponga en práctica y examinar los efectos que se presentan al agregar objetos al lenguaje.
- Construir un intérprete que ofrezca orientación a objetos.

9.2 Conceptos de orientación a objetos

Muchas tareas de programación requieren que el programa haga uso del concepto de estado apoyándose en una interfaz[19]. Por ejemplo, un sistema manejador de base de datos, tiene un estado interno, sin embargo, únicamente accedemos y modificamos el estado mediante la interfaz proporcionada para ello.

La programación orientada a objetos es un herramienta que permite alterar el estado asegurando que los múltiples cambios a las variables que lo constituyen sean actualizadas de manera coordinada, con el fin de mantener la consistencia del estado.

Se define como **objeto** a cada pieza que se encarga de manejar el estado, el cual consiste de ciertos valores almacenados llamados **atributos** y de procedimientos asociados llamados **métodos**, los cuales tienen acceso a estos atributos.

[19] Bajo el contexto de este ejemplo, interfaz se utilizará como un conjunto de operaciones, funciones o procedimientos que se ofrecen a manera de biblioteca, por lo que no se debe confundir con el concepto interfaz de algunos lenguajes de programación como Java, el cual hace referencia a un tipo abstracto de dato para definir el comportamiento de las subclases, [23].

La operación de mandar llamar a un método frecuentemente es visto como el envío del método y sus argumentos como un mensaje al objeto, a esto se le conoce como la vista de **envío de mensajes** de la programación orientada a objetos.

Posiblemente, se necesite más de un objeto para realizar una tarea específica, alterando el estado de estos objetos mediante los mismos métodos. Por ejemplo, hacer uso de múltiples sistemas manejadores de bases datos. Los lenguajes orientados a objetos ofrecen estructuras que especifican los atributos y métodos de cada objeto, a las cuales llamamos **clases**. Al crear un objeto, se dice que el objeto es una instancia de clase.

Algunas veces, se desea que las clases compartan propiedades (atributos o métodos) y que, a su vez, permita hacerlas más específicas para realizar una tarea en particular. Los lenguajes orientados a objetos proveen la **herencia**, que permite definir una clase como modificación de una existente, añadiendo o cambiando métodos y atributos.

Al modificar una clase existente para crear una nueva, se dice que esta nueva clase **hereda de** o **extiende a** la clase anterior, ya que el comportamiento fue heredado de la clase original, [6].

9.3 Sintaxis concreta

La sintaxis concreta del lenguaje está descrita usando notación EBNF de la siguiente forma:

BL (Base Language): La gramática `BL` servirá como base para definir a las demás gramáticas, ya que contiene las primitivas comunes, las cuales en este punto del curso deberán resultar familiares a los estudiantes.

```
<BL> ::= <num>
       | <string>
       | true
       | false
       | {<binop> <BL> <BL>}
       | {if <BL> <BL> <BL>}
       | {with* {[<id> <BL>]+} <BL>}
```

```
<binop> ::= +
          | -
          | *
          | /
          | =
          | <
          | >
          | and
          | or
          | string-append
```

Código 9.1 Gramática `BL`

`ICL (In-Class Language)`: La gramática `ICL` únicamente será usada en la definición de una clase.

```
<ICL> ::= <BL>
        | {sqn <ICL> <ICL>}
        | {this.<idMetodo/Atributo> <ICL>*}
        | {set! <idAtributo> <ICL>*}
```

Código 9.2 Gramática `ICL`

Se hace abuso de notación en la primitiva `set!` ya que el primer parámetro deberá ser, forzosamente, una primitiva `this` descrita en esta misma gramática debido a una convención que se explicará a fondo en la semántica del intérprete.

`CL (Class Language)`: La gramática `CL` nos servirá para definir una nueva clase.

```
<CL> ::= {class <idClase> extends <idClase>
            {<idAtributo> *}
            {init {<id>*} <ICL>}
            {method <idMetodo> {<id>*} <ICL>}*
         }
```

Código 9.3 Gramática `CL`

Para definir nueva clase, se espera que tras la palabra reservada `class`, se coloque el identificador que le dará nombre a ésta. En seguida, se espera la palabra reservada `extends` que servirá para indicar que esta nueva clase será una clase que herede de otra, seguido del nombre de la clase padre.

Después del nombre de la clase padre, se espera una lista con los nombres de los atributos de clase.

El constructor se identificará con la palabra reservada `init`, seguida por una lista de los parámetros del mismo. El cuerpo del constructor deberá seguir las reglas definidas en `ICL` lo que implica que el constructor únicamente podrá dar valores a los atributos del objeto. La existencia del constructor es forzosa.

Los métodos serán identificados por la palabra reservada `method`, que funciona de manera similar al constructor, con la diferencia que se le deberá nombrar insertando el identificador después de la palabra reservada.

CUL (Class Usage Language): La gramática `CUL` nos servirá para instanciar una clase o hacer llamadas a los métodos de un objeto.

```
<CUL> ::= {sqn <ICL> <ICL>}
        | {new <idClase> <BL>*}
        | {new <CL> <BL>*}
```

Código 9.4 Gramática `CUL`

Para instanciar una clase sólo se permiten dos construcciones. La primera es mediante una clase declarada ahí mismo, mientras que la segunda se realiza mediante una variable en la que se haya almacenado una clase (esto se explicará a fondo en la semántica del intérprete).

La llamada a los métodos es de la forma “objeto.método” seguido de los parámetros reales.

EL (External Language): La gramática `EL` nos servirá como punto de entrada y permitirá nombrar e instanciar clases para (posiblemente) ser usadas posteriormente.

```
<EL> ::= <BL>
       | {sqn <ICL> <ICL>}
       | {with* {[<id> <EL>]+} <BL>}
       | <CL>
       | <CUL>
```

Código 9.5 Gramática `EL`

9.4 Semántica del intérprete

Clases y herencia

Las clases como tal son un nuevo valor para este intérprete. Será necesario que al interpretar una expresión de la gramática `<CL>` se devuelva la clase como un nuevo valor, es decir, de la misma forma que interpretar el número 3 da como resultado un tipo de dato `(numV 3)`, interpretar una clase deberá regresar un tipo de dato `(classV)`.

Al implementar herencia, haremos dos convenciones. La primer convención es que una clase padre sólo pueda tener dos clases hijas, ya que el alcance de este intérprete es dar un primer enfoque de cómo se debe implementar un lenguaje de orientación a objetos. La segunda convención es que todas las clases deben extender a otra clase. En caso de que no se desee extender una clase propia, se debe extender a la clase `object%` que para los propósitos de este intérprete es una clase sin atributos, ni métodos y cuyo constructor no recibe argumentos.

Cuando una clase extiende a otra, todos los atributos y métodos se heredan automáticamente. En caso de que la clase que extiende defina un método con el mismo nombre que un método de la clase padre, se deberá usar aquél definido en la clase que extiende.

Instanciar clases

La gramática que describe cómo instanciar las clases es `<CUL>`, lo cual se puede realizar de dos formas:

```
1. {new <CL> <BL>*}
2. {new <idClase> <BL>*}
```

Código 9.6 Ejemplos gramática `CUL`

La primera forma permite instanciar un único objeto de una clase declarada en ese momento, es decir, la clase no será almacenada en una variable (y por lo tanto no llegará al ambiente), pues el único propósito fue instanciar dicho objeto.

La segunda forma deberá buscar en el ambiente el identificador `<idClase>`, en caso de que esa variable sea una clase, se deberá instanciar dicha clase. Cabe mencionar que si la clase se encuentra en el ambiente, entonces ésta se debió de haber introducido mediante una instrucción `with`.

Objetos

Para implementar objetos, usaremos la estructura[20] hash de Racket. Esta estructura nos permite almacenar relaciones (**llave**, **valor**), donde la **llave** es el identificador que le daremos a la entidad a almacenar y el **valor** es aquél que se quiere almacenar. Para recuperar el valor de la estructura, únicamente necesitamos la llave. En este intérprete, usaremos esta estructura para almacenar los atributos y métodos de los objetos. Veamos el siguiente ejemplo

```
(define tabla
  (make-hash (list
              (cons 'x 1)
              (cons 'y 2))))

(hash-ref tabla ‘x) ;;regresa 1
(hash-set! tabla 'z 3) ;;añade el par z -> 3
(hash-ref tabla ‘z ‘___noexiste___) ;;regresa 3
```

Código 9.7 Ejemplos de funciones hash

La función `make-hash` inicializa la estructura y de manera opcional recibe una lista de pares, (llave, valor-inicial). En este caso (ver el Código 9,7) hemos inicializado la estructura con `(x,1)` y `(y,2)`.

La función `hash-ref` recibe una estructura `hash` y una llave, para regresarnos el valor que contiene. Cabe mencionar que ésta puede además recibir un tercer valor opcional, el cual sería el valor de regreso si no se encontrara dicha llave.

El procedimiento `hash-set!` introduce un nuevo par (llave , valor) al `hash` que se usará para almacenar los atributos y métodos del objeto.

Al intérprete se le debe agregar un nuevo valor objeto (`objectV`), el cual hará uso de esta estructura, para almacenar valores y métodos, no se debe olvidar ejecutar el constructor para inicializar las variables.

[20] Estructura tipo tabla hash (hash table) que usa a su vez una función hash para asignar valores a posiciones de un vector o arreglo. Una función hash es un algoritmo que asigna cualquier cantidad de valores a una cantidad fija de llaves o posiciones, [2].

Como una convención para este intérprete, cuando nos estemos refiriendo a las variables de clase, será necesario usar la construcción `this` (descrita en la gramática <ICL>). Cuando se haga uso de `this`, se deberá buscar el valor en la estructura hash del objeto, y en caso de que no se haga uso de `this`, se deberá buscar el valor en el ambiente.
Se recomienda importar las funciones del *analizador sintáctico*[21] a un archivo nuevo. Suponiendo que el analizador sintáctico está en un archivo llamado “`ParserObjetos.rkt`” y en el mismo directorio que el intérprete, llamado “`InterpObjetos.rkt`”, podemos entonces importar las funciones de la siguiente manera:

```
;;archivo InterpObjetos.rkt
#lang plai
(require “ParserObjetos.rkt")
(define (interp …))
```

Código 9.8 Ejemplo de importación de funciones de otros archivos

Ejemplo de creación de un objeto:

```
 1.  '{with* {[robert {new {class persona% extends object%
 2.     {nombre apellido edad estatura}
 3.     {init {nombre apellido edad estatura}
 4.           {sqn
 5.             {set! {this.nombre} nombre}
 6.             {set! {this.apellido} apellido}
 7.             {set! {this.edad} edad}
 8.             {set! {this.estatura} estatura}
 9.             #t }}
10.     {method crecer {estatura} {set! {this.estatura} estatura}}
11.     {method cumpleaños {} {set! {this.edad} {+ {this.edad} 1}}}
12.     {method get-edad {} {this.edad}}
13.     {method get-estatura {} {this.estatura}}
14.     } "Robert" "Lewandowski" 25 1.84}]}
15.         {sqn
16.           {robert.crecer 1.86}
17.           {robert.get-estatura}
18.           }}
```

Código 9.9 Ejemplo de creación de un objeto

[21] El analizador sintáctico de esta práctica se encuentra en el Apéndice B.

En la línea `1` se introduce una nueva variable llamada `robert` que tendrá como valor una instancia (objeto) de una clase que no debe de llegar al ambiente llamada `persona%`. En la línea `2` se definen los atributos de la clase los cuales son `nombre, apellido, edad` y `estatura`. El constructor está definido de las líneas `3` a la `9`, su propósito es dar valores de inicio a los atributos de clase inicializándolos con los parámetros recibidos. Es importante recordar que para referirnos a las variables de clase es necesario hacer uso de `this`. En las líneas `10` y `13` definimos distintos métodos (`crecer, cumpleaños, get-edad y get-estatura`) para definir el comportamiento de los objetos; en la línea `14` se pasan los valores que recibirá el constructor al instanciar el objeto. Por último, en la línea `16` se usa el método `crecer` para modificar la estatura del objeto `robert` y finalmente en la línea `17` se pide el valor de la estatura del mismo objeto.
El resultado de interpretar el código es `(numV 1.86)` ya que se modificó el valor que originalmente tenía.
El alumno debe implementar el procedimiento `interp` que lleve a cabo la interpretación de la gramática expuesta, ayudado por el analizador sintáctico proporcionado[16].

Capítulo 10 Subtipado

10.1 Objetivo

- Analizar el concepto de subtipado añadiendo al intérprete con orientación a objetos una implementación de dicho concepto.
- Construir un intérprete que haga uso del concepto de subtipado para verificar la consistencia de listas.

10.2 Motivación

"Sin el concepto de subtipado, las reglas de tipado simple pueden llegar a ser rígidas a tal punto que resulte incómodo. La insistencia del sistema de tipado para que los tipos de los argumentos de los parámetros reales sean exactamente los mismos que los formales, hacen que el verificador de tipos rechace muchos programas que para el programador, sean programas que intuitivamente pudieran funcionar"[12].

Veamos el siguiente ejemplo escrito con la sintaxis de la práctica de orientación a objetos (Códigos 9.1 al 9.5):

```
1. {class persona% extends object%
2.   {nombre apellido edad}
3.   ;;definición de la clase persona% con constructor y
4.   ;;métodos get y set de atributos nombre, apellido y edad
5.   }
```

Código 10.1 Ejemplo de subtipado de la clase `persona%`

```
1. {class mascota% extends object%
2.   {nombre edad dueño}
3.   ;;definición de la clase mascota% con constructor y
4.   ;;métodos get y set de atributos nombre, edad y dueño
5.   }
```

Código 10.2 Ejemplo de subtipado de la clase `mascota%`

Supongamos ahora una función auxiliar, declarada de manera externa tanto de la clase `persona%` como `mascota%`. Esta función tiene como propósito calcular la edad expresada en días[22].

Se muestra a continuación la definición de dicha función, se hace uso de la sintaxis de la práctica de verificador de tipos CFWAEL (ver Código 8.1)

```
1. {with {[get-edad-en-dias
2.          {fun {[a : persona%]} {* 365 {persona.get-edad}}}]}
3.        ;;hacer algo con la función get-edad-en-dias
4.        }
```

Código 10.3 Ejemplo de la función `get-edad-en-dias`

El verificador de tipos acepta que el parámetro de la función `get-edad-en-dias` sea únicamente una instancia de la clase `persona%`. Sin embargo, podría parecer intuitivo que funcionara de igual manera con instancias de la clase `mascota%`, ya que ambas clases definen tanto atributo `edad` como método `get-edad`. La finalidad del subtipado es refinar las reglas de tipado para que términos como los de este ejemplo sean aceptados.

*"Decimos que **S** es un **subtipo** de **T**, denotado como* S <: T*, para expresar que cualquier término del tipo **S** puede ser usado de manera segura en un contexto donde se espera un término del tipo **T**. Esta manera de ver al subtipado a menudo se le llama el principio de la substitución segura"*[12].

[22] La implementación de este procedimiento estaría mejor ubicado dentro de la clase `persona%` o `mascota%`, se construye fuera para que el ejemplo sea más ilustrativo

Suponiendo que se llegara a implementar el ejemplo, se podría hacer una clase padre llamada `ser-vivo%` con el atributo `edad` y método `get-edad`, para que sea extendida por las clases `persona%` y `mascota%`. La función `get-edad-en-dias` tendría como parámetro el tipo `ser-vivo%` y la misma función serviría para instancias de ambas clases (`persona%` y `mascota%`)

10.3 Relación entre subtipado y herencia

Uno de los avances fundamentales en el paradigma de la programación orientada a objetos es la habilidad de reutilizar código. Frecuentemente sucede que un nuevo problema puede ser resuelto con una pequeña variación de un código hecho con anterioridad.

Un sistema de tipos que permite que los procedimientos ganen en generalidad explotando la posibilidad que un valor pueda presentarse en múltiples formas se le conoce como un **sistema de tipos polimórfico**.

El tipo de polimorfismo que se presenta en Java se es llamado polimorfismo de subtipos que está basado en la idea que puede existir una relación de subtipado entre los tipos. (No confundir la noción de subtipado con la de subconjunto).

Si declaramos a la clase B como una extensión de la clase A (es decir, que la clase B hereda de A), tendremos -a parte de todo aquello relacionado con la herencia- que B <: A. En otras palabras, si en algún momento se tiene un método que espere un argumento de tipo A, dicho método podrá aceptar un objeto de tipo B.

Si extendemos la clase B con una nueva clase C, tendremos: C <: B <: A, y si volvemos a extender la clase A con otra nueva clase D, tendremos que D <: A pero no habrá ninguna relación de subtipo entre D ≮: B ni D ≮: C. Un método que espere un objeto de tipo A aceptará objetos de tipo B, C o D. Esto otorga una amplia generalidad al código.

Java soporta polimorfismo de tipo independiente a la herencia. Esto lo hace mediante interfaces. Una interfaz es una declaración de una colección de nombres de métodos - excluyendo la implementación - y tal vez algunas constantes. Las interfaces describen un (sub)conjunto de métodos que una clase tendrá. No existe código en una interfaz, por lo que no hay nada que heredar. Una clase puede ser declarada para implementar una interfaz. Esta relación es realmente una declaración de subtipo. Una interfaz nombra un nuevo tipo (no una clase) y cuando se dice que una clase C implementa una interfaz I, hemos configurado la relación de subtipado C <: I. Debido a que no hay código en la interfaz, C no hereda ningún código, pero debe tener un método con la misma firma que cada método definido en la interfaz I.

Analicemos ahora el siguiente ejemplo:

Imaginemos que somos los encargados de desarrollar código para dibujar en pantalla la gráfica de una función matemática. Quisiéramos que dicho proceso esté separado de la función, es decir, evitar escribir el código para dibujar la función coseno y otro diferente -aunque casi idéntico- para dibujar la función exponencial, es decir, nos gustaría que la función a graficar sea un parámetro.

¿Cómo podemos diseñar, en orientación a objetos, un modelo de función matemática que nos permita abstraer el proceso de graficación?

Sabemos que para graficar una función necesitamos un método, al que llamaremos `evalúa`, que reciba un double y regrese otro double como resultado. Pueden haber otros métodos y atributos asociados a las funciones matemáticas, pero no son necesarias para llevar a cabo la graficación. Basta con saber que una función `f` tiene un método evalúa, digamos, `f.evalúa(3.14159)`, que regresa otro double como resultado. De este modo, en vez de tener que analizar todos los detalles de la clase de funciones matemáticas, únicamente definimos una interfaz, a la cual llamaremos `Plottable`, con la firma del método `evalúa`.

Cuando definamos nuestros objetos de funciones matemáticas, podemos hacerlos tan complejos como queramos y sabemos que, mientras implementen la interfaz `Plottable`, nos aseguraremos que realmente tienen el método evalúa.

Si tuviéramos que otro tipo de objetos, como por ejemplo, datos financieros, esperaríamos definir una clase totalmente diferente que no tenga ninguna relación con las funciones matemáticas. Sin embargo, la clase modele los datos financieros, podría también implementar la interfaz `Plottable`, por lo que tendría método `evalúa`.

Hemos logrado generalidad en nuestro proceso de graficación mediante el uso de polimorfismo de subtipos en una situación mucho más general que aquella que se pudiera haber logrado con herencia, [26].

10.4 Añadiendo el concepto de subtipado al intérprete de orientación a objetos

Recordemos que en el intérprete de orientación a objetos se tomó la convención que una clase padre no puede tener más de dos clases, hijas. Para añadir el concepto de subtipado al intérprete se necesita llevar la jerarquía de las clases, para esto se puede hacer uso de una estructura de tipo árbol binario. Por ejemplo:

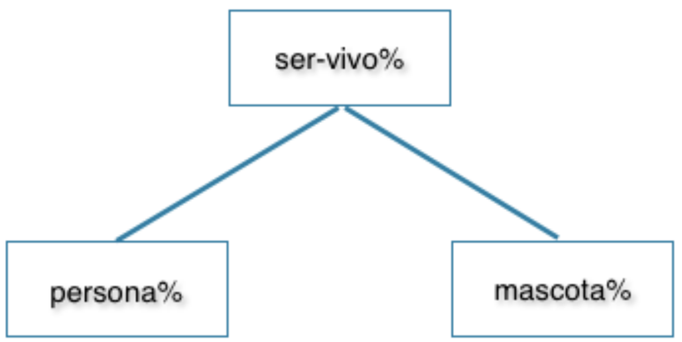


Figura 2. Ejemplo simple de jerarquía de clases

Definamos la estructura de árbol binario para modelar la jerarquía de clases:

```
1. #lang plai
2. ;;Definiendo tipo de datos (data-type) de arbol
3. (define-type arbolB
4.   [vacio]
5.   [nodo (nombre-clase symbol?) (izq arbolB?) (der arbolB?)])
6.
7. ;;Procedimiento para construir un árbol cuya raíz sea ‘object%
8. (define (jerarquia-clases)
9.   (nodo ‘object% (vacio) (vacio)))
10.
11. ;;Agrega una nueva clase a la variable arbolB jerarquía
```

```
12. ;;el procedimiento supone que padre ya se encuentra en el árbol
13. ;;la nueva clase se inserta como su hijo
14. (define (agrega jerarquia padre nueva)
15.   (type-case arbolB jerarquia
16.     [nodo (clase izq der)
17.           (cond
18.             [(and (eqv? clase padre)
19.                   (vacio? izq)
20.                    (vacio? der))
21.                (nodo clase (nodo nueva (vacio) (vacio)) (vacio))]
22.             [(and (eqv? clase padre)
23.                   (vacio? izq)
24.                   (not (vacio? der)))
25.                (nodo clase
26.                      (nodo nueva (vacio) (vacio)) (vacio) der)]
27.             [(and (eqv? clase padre)
28.                   (not (vacio? izq))
29.                   (vacio? der))
30.                (nodo clase izq (nodo nueva (vacio) (vacio)))]
31.             [(not (eqv? clase padre))
32.                (nodo clase
33.                      (agrega izq padre nueva)
34.                      (agrega der padre nueva))]
35.             [(and (eqv? clase padre)
36.                   (not (vacio? izq))
37.                   (not (vacio? der)))
38.              (error 'agrega
39.                "Una clase padre solo puede tener 2 clases hijas")]
```

```
40.             [else (error 'agrega "revisar parámetros")])]
41.     [vacio() (vacio)]))
42.
43.;;Decide si el parámetro hijo es un subtipo
44.;;del parámetro ancestro dado un arbolB jerarquía
45.(define (subtipo? jerarquia hijo ancestro)
46.   (type-case arbolB jerarquia
47.     [vacio() false]
48.     [nodo (clase izq der)
49.           (cond
50.             [(not (eqv? clase ancestro))
51.              (or (subtipo? izq hijo ancestro)
52.                  (subtipo? der hijo ancestro))]
53.             [(eqv? clase ancestro)
54.              (or (encuentra izq hijo)
55.                  (encuentra der hijo))])]))
56.
57.;;Decide si el parámetro hijo está en el arbolB jerarquia
58.(define (encuentra jerarquia hijo)
59.   (type-case arbolB jerarquia
60.     [vacio() false]
61.     [nodo (clase izq der)
62.           (if (eqv? hijo clase)
63.               true
64.               (or (encuentra izq hijo)
65.                   (encuentra der hijo)))]))
```

Código 10.4 Estructura árbol binario para subtipado

Modelemos la siguiente jerarquía usando la estructura recién definida:

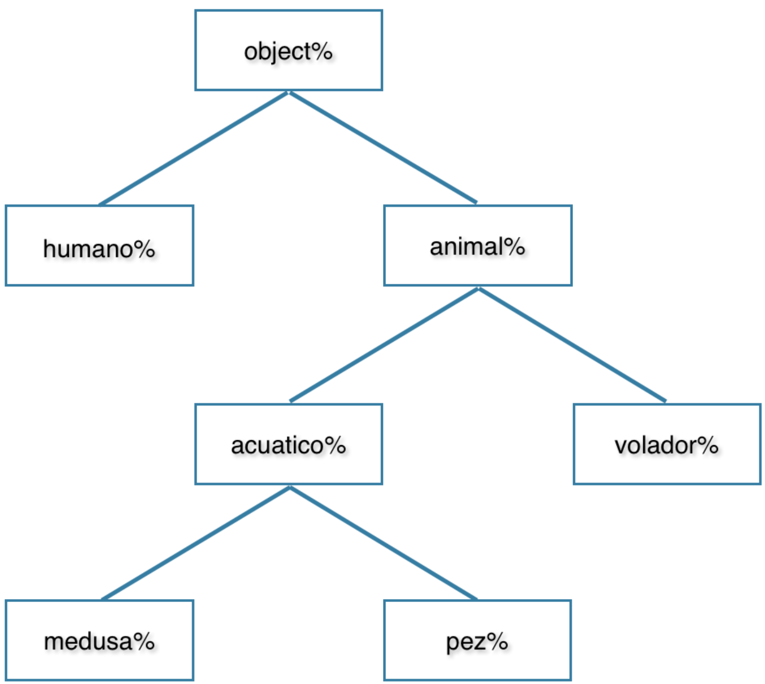


Figura 3. Ejemplo de jerarquía de clases

```
1.  (define raiz (jerarquia-clases))
2.  (begin
3.    (set! raiz (agrega raiz 'object% 'animal%))
4.    (set! raiz (agrega raiz 'object% 'humano%))
5.    (set! raiz (agrega raiz 'animal% 'acuatico%))
6.    (set! raiz (agrega raiz 'animal% ‘volador%))
7.    (set! raiz (agrega raiz 'acuatico% ‘medusa%))
8.    (set! raiz (agrega raiz 'acuatico% ‘pez%)))
9.  (subtipo? raiz 'object% 'animal%);;false
10. (subtipo? raiz 'animal% 'object%);;true
```

```
11. (subtipo? raiz ‘medusa% 'acuatico%);;true
12. (subtipo? raiz ‘pez% ‘object%);;true
```

Código 10.5 Ejemplo para estructura de subtipado

Analizando el Código 10.5 tenemos que en la línea 1 se asigna en la variable raíz el resultado de la función `jerarquia-clases` que inicializa la estructura, colocando al nodo `object%` como raíz. De las líneas 3 a 8, se especifica la relación padre - hijo de las clases. Por ejemplo, en la línea 3 se especifica que la clase `animal%` hereda de la clase `object%` mientras que en la línea 6 se especifica que `volador%` hereda de la clase `animal%`.

En la línea 9 se pregunta si `object%` es hijo de `animal%`, lo cual se evalúa a falso, en la línea 10 se pregunta si `animal%` es hijo de `object%` lo cual es resulta en valor de verdadero.

Esta estructura, basada en un árbol binario, nos permitirá agregar al intérprete las relaciones padre - hijo de las clases conforme se declaren. Una vez que se hayan agregado todas las relaciones, será la herramienta con la que se verificará que se respeten las reglas de subtipado.

10.5 Sintaxis concreta

La sintaxis concreta para esta práctica será la misma que la utilizada en la práctica de orientación a objetos, (Códigos 9.1 al 9.5), haciendo una ligera modificación al Código 9.5 y añadiendo una nueva gramática. Se explican los cambios a continuación.

La nueva gramática que se añade es `LL (List Language)` que nos permitirá construir listas, la notación de ésta en EBNF es:

```
<LL> ::= empty
       | {cons <CUL> <LL>}
       | {car <LL>}
       | {cdr <LL>}
```

Código 10.6 Gramática LL

Haciendo uso de esta gramática ahora se podrán construir listas de objetos. Para poder hacer uso de las listas, se modificará la gramática `EL (External Language)` de la siguiente manera:

```
<EL> ::= <BL>
       | {sqn <ICL> <ICL>}
       | {with* {[<id> <EL>]+} <BL>}
       | <CL>
       | <CUL>
       | <LL>
```

Código 10.7 Gramática `EL`

Este cambio nos permite crear listas con instancias de clases que se hayan definido previamente.

10.6 Semántica del intérprete

Ahora que se pueden añadir instancias de clases a listas, tomaremos la convención que la cabeza de la lista definirá el tipo de la cola de esa misma lista.

Las listas funcionan de la misma manera como las hemos trabajado desde el inicio del intérprete, es decir, de la misma manera que los procedimientos `cons`, `car` y `cdr` de Racket.

A manera de ejemplo, continuaremos con aquellos que hemos visto en el Código 10.5. Si la cabeza de la lista tiene una instancia de la clase `object%`, la lista podrá tener cualquiera de sus subtipos en el resto de ésta. Sin embargo, si la cabeza de la lista fuera de tipo `acuatico%`, la cola de lista solo podría tener instancias de las clases `acuatico%`, `pez%` y `medusa%`.

En caso de que la cola no sea subtipo de la cabeza, se debe mandar un error especificando lo sucedido.

```
1. {with {[animal%
2.         {class animal% extends object%
3.           {edad}
4.           ;;definición de la clase animal% con constructor y
5.           ;;métodos get y set para el atributo edad
6.           }]
7.        [acuatico%
```

```
8.          {class acuatico% extends animal%
9.            {edad}
10.           ;;definición de la clase acuatico% con constructor y
11.           ;;métodos get y set para el atributo edad
12.           ;;y método nadar
13.           }]
14.         [medusa%
15.          {clas medusa% extends acuatico%
16.                {edad tentáculos}
17.                ;;definición de la clase medusa% con constructor y
18.                ;;métodos get y set para atributos edad y tentáculos
19.                }]
20.         [pez%
21.          {class pez% extends acuatico%
22.            {edad aletas}
23.            ;;definición de la clase pez% con constructor y
24.            ;;métodos get y set para atributos edad y aletas
25.            }]}
26.        {begin
27.          {cons {new animal% 5}
28.                {cons {new medusa% 10 20}
29.                      {cons {new pez% 3 2} empty}}};;ok
30.          {cons {new medusa% 5}
31.                {cons {new pez% 3 2} empty}};;error
32.          }
33.        }
```

Código 10.8 Ejemplos de la gramática LL

En las líneas `1` a `25`, se definen las clases `animal%`, `acuatico%`, `medusa%` y `pez%`. En las líneas `27` a `29` se crea una lista cuya cabeza es de tipo `animal%`, mientras que los siguientes elementos son de tipo `medusa%` y `pez%`. Dado que `medusa%` y `pez%` son subtipos de `acuatico%`, no deberá existir un error.

En las líneas `30` y `31`, se define una nueva lista cuya cabeza es de tipo `medusa%` mientras que el siguiente elemento es de tipo `pez%`, el error se da porque `pez%` no es subtipo de `medusa%`.

El objetivo del alumno será construir un intérprete que:

- Haga uso de la estructura expuesta en el Código 10.4 para almacenar las relaciones padre hijo de las clases.
- Permita construir listas cuyos elementos sean subtipos de la cabeza.
- Informe a manera de error cuando el punto anterior no se cumpla.

Conclusiones

Este trabajo realizado para la materia de Lenguajes de Programación, permite a los estudiantes obtener un enfoque práctico del temario del curso. A este respecto hago hincapié en que éste debe ser visto como un complemento al curso y no como un suplemento del mismo.

En los primeros capítulos, se introduce al estudiante tanto al lenguaje de programación, Racket, como al ambiente integrado de desarrollo DrRacket, lo cual permite una adaptación que será vital para poder desarrollar las siguientes prácticas que requieren que el estudiante esté familiarizado con las funciones y bibliotecas que el lenguaje ofrece.

Los siguientes capítulos presentan ejercicios que aumentan gradualmente en complejidad, se empieza con el análisis sintáctico de una pequeña gramática que ofrece operaciones aritméticas y se termina con un intérprete (tanto el analizador sintáctico como el intérprete en sí) que ofrece la construcción de listas que únicamente contenga elementos que son subtipos de la cabeza.

Dichos ejercicios son acompañados por su respectivo fundamento teórico, así como con ejemplos o análisis de una ejecución del intérprete en cuestión, con la finalidad de acercar al estudiante lo más que se pueda al concepto que deberá implementar para ese ejercicio.

El resolver estos ejercicios, permitirá al estudiante enfrentarse a los mismos problemas con los que tratan los diseñadores y desarrolladores de lenguajes de programación que se usan hoy en día, tanto de manera comercial como académica. Esto, le dará al estudiante un enfoque más amplio de los lenguajes de programación con los que a lo largo de la carrera ha trabajado, el cual puede ser aprovechado para varios fines, sin embargo, quisiera resaltar el que considero más importante:

Traducir el conocimiento adquirido en herramientas que le serán de utilidad para construir programas futuros, tanto por deberes académicos como profesionales, que aprovechen al máximo el potencial ofrecido por el lenguaje que se haya decidido usar para llevar a cabo el trabajo o tarea en cuestión. Aunque si bien es cierto que explotar al máximo el potencial de un lenguaje de programación es consecuencia de múltiples factores, el tener una visión del mismo, tanto del lado de los diseñadores como el del usuario, es sin duda un gran paso en esa dirección.

Una vez que el estudiante concluya con el material, deberá haber ampliado su panorama acerca de cómo se implementa un lenguaje de programación. Aunado a esto, el presente trabajo introduce algunos temas que el plan de estudios de la carrera no contempla en asignaturas anteriores y refuerza otros vistos previamente, enfocados desde un punto de vista más práctico.

Por ejemplo, el objetivo del Capítulo 7 es construir un intérprete que ofrezca recursión, el cual es un tema que el alumno ha visto en materias anteriores según el plan de estudios actual. Estas prácticas, presentan la base teórica de la recursión, sin embargo, el objetivo no es que alumno memorice que una función recursiva es una función que hace referencia a sí misma, tampoco es el objetivo que escriba funciones recursivas por el hecho de poder hacerlas. El objetivo primordial es buscar cómo implementar recursión (y los demás temas expuestos en este trabajo) en un intérprete de un lenguaje de programación "pequeño", para que un "eventual usuario de este pequeño intérprete" pueda hacer uso de ella.

Este trabajo, aunque presenta muchos de los temas esenciales del diseño e implementación de los lenguajes de programación, es necesario hacer notar que se ha intentado ajustar el tiempo que se debe emplear para llevar a cabo cada una las prácticas, haciendo uso de las estructuras y primitivas que ofrece Racket. Cabe mencionar que el alcance de las prácticas ha sido también ajustado para poder tener una visión general de todos los temas en el tiempo que dura un semestre en la UNAM.

Los temas que se han expuesto aquí alcanzarían para tener cursos y manuales de prácticas completos para ellos mismos, por lo que se recomienda profundizar en los temas que hayan sido de interés, ya que de estos temas se derivan otros a su vez importantes en las ciencias de la computación (como compiladores) o añadir temas que por diversas razones se han dejado fuera (como construir un intérprete que ofrezca un sistema de hilos, threads).

De esta forma, este material ofrece un compendio de prácticas de dificultad razonable para que los estudiantes de la carrera las realicen y complementen así las clases teóricas que imparte el profesor titular del curso, permitiéndoles de esta forma analizar los temas desde un punto de vista práctico, cumpliendo a su vez con los objetivos que fueron planteados para este manual y el curso mismo.

Referencias bibliográficas

1. Cardone, Felice, and J. Roger Hindley. **History of Lambda Calculus and Combinatory Logic**. Elsevier, 2009.
2. Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. **Introduction to Algorithms**. MIT Press and McGraw-Hill, 2001.
3. Dowek, Gilles. **Principles of Programming Languages**. London: Springer, 2009.
4. Dowek, Gilles, and Jean-Jacques Lévy. **Introduction to the Theory of Programming Languages**. London: Springer, 2011.
5. Felleisen, Matthias. **91n Introduction to Programming and Computing**. Cambridge, MA: MIT, 2001.
6. Felleisen, Matthias, Robert Bruce. Findler, and Matthew Flatt. **Semantics Engineering with PLT Redex**. Cambridge, MA: MIT, 2009.
7. Friedman, Daniel P., Mitchell Wand, and Christopher Thomas. Haynes. **Essentials of Programming Languages**. Cambridge, MA: MIT, 1992.
8. Gabbrielli, Maurizio, and Simone Martini. **Programming Languages: Principles and Paradigms**. London: Springer, 2010.
9. Harper, Robert. **Practical Foundations for Programming Languages**. Cambridge: Cambridge UP, 2013.
10. Krishnamurthi, Shriram, and Martin Odersky. **Compiler Construction**. [New York]: Springer-Verlag Berlin Heidelberg, 2007.
11. Krishnamurthi, Shriram. **Programming Languages: Application and Interpretation**, 2003.
12. Pierce, Benjamin C. **Types and Programming Languages**. Cambridge, MA: MIT, 2002.
13. Stroustrup, Bjarne. **Programming: Principles and Practice Using C**. Upper Saddle River, NJ: Addison-Wesley, 2009.

Referencias en Internet

14. "BibliographyNaur." BibliographyNaur. Web. 26 Marzo. 2014. <http://www.naur.com/bibliography.html>.
15. "The Haskell Platform." Download Haskell. Web. 26 Marzo. 2014. <http://www.haskell.org/platform/>.
16. "John Warner Backus." Backus Biography. Web. 26 Marzo. 2014. <http://www-groups.dcs.st-and.ac.uk/~history/Biographies/Backus.html>.
17. "Racket Documentation." Racket Documentation. Web. 26 Marzo. 2014. <http://docs.racket-lang.org/>.
18. "The Racket Language." The Racket Language. Web. 26 Marzo. 2014. <http://racket-lang.org/>.
19. "Mapa Curricular" Facultad de Ciencias. Web. 26 Marzo. 2014. <http://www.fciencias.unam.mx/licenciatura/mapa/104/1556>.
20. "Desktop PCs operating system market share 2012-2015" statista. Web. 21 Abril. 2015. <http://www.statista.com/statistics/218089/global-market-share-of-windows-7/>.
21. "History of Lisp" John McCarthy. Web. 11 Mayo. 2015. <www-formal.stanford.edu/jmc/history/lisp/lisp.html>.
22. "Haskell: Lambda Expressions" University of Birmingham. Web. 11 Mayo. 2015. <http://www.cs.bham.ac.uk/~vxs/teaching/Haskell/handouts/lambda.pdf>.
23. "Interfaces" The Java™ Tutorials . Web. 11 Mayo. 2015. <https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html>.
24. “Notation for Documentation” Racket Documentation. Web. 21 Mayo. 2015. <http://docs.racket-lang.org/reference/notation.html?q=Notation%20for%20Syntactic%20Form%20Documentation#%28part._.Notation_for_.Syntactic_.Form_.Documentation%29>.
25. “Documentation” Haskell. Web. 19 Agosto 2015. <https://www.haskell.org/documentation>.
26. “Subtyping and Inheritance in Java” McGill School of computer Science. Web. 20 Agosto 2015. <http://www.cs.mcgill.ca/~cs202/2011-01/lectures/maja/subtyping-and-inheritance-in-java.pdf>.

Apéndices

Apéndice A
Instalación del intérprete Haskell

El intérprete recomendado de Haskell lleva por nombre Haskell Platform, al momento de escribir este documento la última versión es 2013.2.0.0, [15].

A continuación se describen los pasos necesarios para instalarlo en los tres sistemas operativos más comunes del mundo:

Mac OS®: Descargar el archivo con extensión .pkg y ejecutarlo. Para los propósitos de este documento, las opciones por defecto son suficientes.

Windows®: Descargar el archivo con extensión .exe y ejecutarlo. Para los propósitos de este documento, las opciones por defecto son suficientes.

Linux: Se recomienda usar los repositorios de la distribución en cuestión. A manera de ejemplo, para instalar desde una distribución basada en Debian (Ubuntu, Mint o el mismo Debian)

```
$sudo apt-get install haskell-platform
```

Apéndice B

Analizador sintáctico para las gramáticas de orientación a objetos

A continuación se muestra el analizador sintáctico para las gramáticas de orientación a objetos:

```
;;Archivo “ParseObjetos.rkt”
;;Decide si la variable buscando se encuentra como subcadena
;;de la variable cadena

(define (contains? buscando cadena)

  (let ([buscandoS (if (string? buscando)
                       buscando
                       (symbol->string buscando))]

        [cadenaS (if (string? cadena)
                     cadena
                     (symbol->string cadena))])

    (regexp-match? (regexp buscandoS) cadenaS)))


;;symbol|string ->boolean
;;Decide si la cadena “.” es subcadena de la variable cadena

(define (contains-punto? cadena)
    (contains? "(\\.)" cadena))


;; Tipo de datos (data-type) que representa una liga entre
;; un id y un valor

(define-type binding
  [bind (name symbol?)
        (value (lambda (x)
                 (or (ICL? x)
                     (EL? x)
                     (CUL? x)
                     (CL? x)
                     (BL? x))))])
```

```
;; Busca repeticiones en la lista l usando como comparador a
;; a la variable comp. De no haber repetidos se regresa falso,
;; de haberlos se regresa el elemento repetido
(define (buscaRepetido l comp)
   (cond
     [(empty? l) #f]
     [(member? (car l) (cdr l) comp) (car l)]
     [else (buscaRepetido (cdr l) comp)]))


;;Construye el árbol de sintaxis abstracta, recibiendo la sintaxis
;;concreta como una lista en la variable lst y una función de
;;retorno dado que este procedimiento será la base
;;para los siguientes
(define (parse-bindings lst returnFunc)
    (let ([bindRep
            (buscaRepetido
                lst
                (lambda (e1 e2) (symbol=? (car e1) (car e2))))])
    (if (boolean? bindRep)
           (map (lambda (b) (bind (car b) (returnFunc (cadr b))))
                 lst)
        (error 'parse-bindings
               (string-append
                  "El id "
                  (symbol->string (car bindRep))
                  " está repetido”)))))
;; Decide si el elemento x está en la lista l usando como
;; comparador al procedimiento del mismo nombre
(define (member? x l comparador)
  (cond
     [(empty? l) #f]
     [(comparador (car l) x) #t]
     [else (member? x (cdr l) comparador)]))


;;Verfica que el simbolo de función esté en la gramática
(define (eligeOp simbolo)
```

```
     (case simbolo
       [(+) '+]
       [(-) '-]
       [(*) '*]
       [(/) '/]
       [(=) '=]
       [(<) '<]
       [(>) '>]
       [(and) 'and]
       [(or) 'or]
       [(string-append) 'string-append]
       [else (error 'eligeOp "Operación binaria no definida”)]))


;;tipo para la gramática new instance language
(define-type BL
  [BLnum (n number?)]
  [BLboolean (b boolean?)]
  [BLstring (s string?)]
  [BLbinop (f symbol?)
           (izq (lambda(x) (or (BL? x) (ICL? x) (or CL?))) )
           (der (lambda(x) (or (BL? x) (ICL? x) (or CL?))) )]
  [BLid (id symbol?)]
  [BLwith* (bindings (listof bind?)) (body BL?)]
  [BLif (test (lambda(x) (or (BL? x) (ICL? x) (or CL?))) )
        (then (lambda(x) (or (BL? x) (ICL? x) (or CL?))) )
        (else (lambda(x) (or (BL? x) (ICL? x) (or CL?))) )])


;;Construye el árbol de sintaxis abstracta para la gramática BL
(define (parseBL exp)
  (cond
    [(number? exp) (BLnum exp)]
    [(boolean? exp) (BLboolean exp)]
    [(string? exp) (BLstring exp)]
    [(symbol? exp) (BLid exp)]
    [(list? exp)
     (case (car exp)
       [(+ - * / = string-append)
        (BLbinop (eligeOp (car exp))
```

```
                  (parseBL (cadr exp))
                  (parseBL (caddr exp)))]
        [(if) (BLif (parseBL (cadr exp))
                    (parseBL (caddr exp))
                    (parseBL (cadddr exp)))]
        [(with*) (BLwith* (parse-bindings (cadr exp) parseBL)
                          (parseBL (caddr exp)))]
        [else (error 'parseBL “BL palabra reservada no reconocida")])]
     [else (error 'parseBL “BL tipo de dato incorrecto")]))

;;tipo para la gramática in-class language
(define-type ICL
  [ICLid (id symbol?)]
  [ICLset! (id ICLthis?)
           (body (lambda(x) (or (ICL? x) (BL? x))))]
  [ICLbegin (bodys
              (lambda(x) (or/c ((listof ICL?) x) ((listof BL?) x))))]
  [ICLthis (id-metodo/atributo symbol?) (parametros (listof ICL?))])

;;Construye el árbol de sintaxis abstracta para la gramática ICL
(define (parseICL exp)
  (cond
    [(number? exp) (BLnum exp)]
    [(boolean? exp) (BLboolean exp)]
    [(string? exp) (BLstring exp)]
    [(symbol? exp) (ICLid exp)]
    [(list? exp)
     (if (contains-punto? (car exp))
         (let* ([separado (string-split
                            (symbol->string (car exp))
                            ".")]
                [identificador-this
                  (string->symbol
                    (car separado))]
                [identificador-metodo/atributo
                  (string->symbol
                    (cadr separado))]
                [parametos (map (lambda(x) (parseICL x)) (cdr exp))])
           (ICLthis identificador-metodo/atributo parametos))
         (case (car exp)
```

```
           [(+ - * / = string-append)
            (BLbinop (eligeOp (car exp))
                     (parseICL (cadr exp))
                     (parseICL (caddr exp)))]
           [(set!)
            (ICLset! (parseICL (cadr exp))
                     (parseICL (caddr exp)))]
           [(if)
            (BLif (parseICL (cadr exp))
                  (parseICL (caddr exp))
                  (parseICL (cadddr exp)))]
           [(begin)
            (ICLbegin (map (lambda(x) (parseICL x)) (cdr exp)))]
           [(with*)
            (BLwith* (parse-bindings (cadr exp) parseICL)
                     (parseICL (caddr exp)))]
           [else (error 'parseICL “parseICL palabra reservada")]))]
    [else (error 'parseICL “parseICL tipo de dato incorrecto")]))

;;tipo para modelar una clase
(define-type claseP
  [constructorP (parametros (listof ICLid?));;parámetros formales
                (body ICL?)]
  [metodoP (id symbol?);;nombre del método
           (parametros (listof ICLid?));;parámetros formales
           (body ICL?)])

;;tipo para el lenguage CL
(define-type CL
  [claseExpr (nombre symbol?)
             (padre symbol?)
             (atributos (listof symbol?))
             (constructor constructorP?)
             (metodos (listof metodoP?))])

;;Construye el árbol de sintaxis abstracta para el lenguaje CL
(define (parseCL exp)
  (if (and (list? exp) (eqv? (car exp) 'class) (symbol? (cadr exp)))
      (parseCL-aux exp)
      "Mala sintaxis al definir clase”))
```

```
;;Construye el árbol de sintaxis abstracta para el lenguaje CL
(define (parseCL-aux exp)
  (let ([nombre-let (cadr exp)]
        [extends-let (caddr exp)]
        [clase-padre-let (cadddr exp)]
        [atributos-let (car (cdr (cdr (cdr (cdr exp)))))]
        [constructor-let
         (parse-constructor (car (cdr (cdr (cdr (cdr (cdr exp)))))))]
        [metodos-let
         (parse-metodos (cdr (cdr (cdr (cdr (cdr (cdr exp)))))))])
    (claseExpr nombre-let
               clase-padre-let
               atributos-let
               constructor-let
               metodos-let)))



;;busca el constructor de la clase y lo crea
(define (parse-constructor exp)
  (if (eqv? 'init (car exp))
      (constructorP (map (lambda(x) (parseICL x))(cadr exp))
(parseICL (caddr exp)))
      "Error en la sintaxis del constructor”))

;;busca los metodos de la clase y los crea
(define (parse-metodos exp)
  (map (lambda(x)
         (if (eqv? (car x) 'method)
             (let ([nombre-metodo-let (cadr x)]
                   [parametos-metodo-let
                     (map (lambda(x) (parseICL x)) (caddr x))]
                   [body-metodo-let (parseICL (cadddr x))])
               (metodoP nombre-metodo-let
                        parametos-metodo-let
                        body-metodo-let))
             "Error de sintaxis en el método"))
        exp))
```

```
;;tipo para modelar el lenguaje  CUL
(define-type CUL
  [llamada-metodo (id-objeto symbol?)
                  (id-metodo symbol?)
                  (parametros (listof BL?))]
  [nueva-instancia-nombrada (id-clase symbol?)
                            (parametros (listof BL?))]
  [nueva-instancia-anonima (id-clase CL?)
                           (parametros (listof BL?))])

;;construye el árbol de sintaxis abstractas para el lenguaje CUL
(define (parseCUL exp)
  (if (list? exp)
      (cond
        [(and (eqv? (car exp) 'new) (symbol? (cadr exp)))
         (nueva-instancia-nombrada
           (cadr exp)
           (map (lambda(x) (parseBL x)) (cddr exp)))]
        [(and (eqv? (car exp) 'new)
              (eqv? (caadr exp) 'class))
         (nueva-instancia-anonima
           (parseCL (cadr exp))
           (map (lambda(x) (parseBL x)) (cddr exp)))]
        [(contains-punto? (car exp))
         (let* ([separado
                  (string-split (symbol->string (car exp)) ".")]
                [identificador-objeto
                  (string->symbol (car separado))]
                [identificador-metodo
                  (string->symbol (cadr separado))]
                [parametos (map (lambda(x) (parseBL x)) (cdr exp))])
           (llamada-metodo identificador-objeto
                           identificador-metodo
                           parametos))]
        [else  (error 'parseCUL
                      "Error de sintaxis en el uso de clase new o
                       llamada a metodo")])
      (error 'parseCUL "Error de sintaxis en el uso de clase new o
                       llamada a metodo”)))
```

```
;;tipo para la gramática external language
(define-type EL
  [ELidA (id symbol?)]
  [ELbegin (bodys (listof CUL?))]
  [ELwith* (bindings (listof bind?))
           (body (lambda(x) (or (EL? x) (BL? x))))])

;;Construye el árbol de sintaxis abstrata para la gramática EL
(define (parseEL exp)
  (cond
    [(number? exp) (BLnum exp)]
    [(boolean? exp) (BLboolean exp)]
    [(string? exp) (BLstring exp)]
    [(symbol? exp) (ELidA exp)]
    [(list? exp)
     (if (contains-punto? (car exp)) (parseCUL exp)
         (case (car exp)
           [(+ - * / = string-append)
            (BLbinop (eligeOp (car exp))
                     (parseEL (cadr exp))
                     (parseEL (caddr exp)))]
           [(if)
            (BLif (parseEL (cadr exp))
                  (parseEL (caddr exp))
                  (parseEL (cadddr exp)))]
           [(begin)
            (ELbegin (map (lambda(x) (parseEL x)) (cdr exp)))]
           [(with*)
            (ELwith* (parse-bindings (cadr exp) parseEL)
                     (parseEL (caddr exp)))]
           [(class) (parseCL exp)]
           [(new) (parseCUL exp)]
           [else (error 'parseEL “parseEL palabra reservada")]))]
    [else (error 'parseICL “parseEL tipo de dato incorrecto")]))
```

```
;;procedimiento global para construir árboles de sintaxis abstracta
;;de cualquier gramática
(define (parse exp)
  (parseEL exp))
```

Código A. Analizador sintáctico para las gramáticas de orientación a objetos

Apéndice C
Notación alterna para sección 2.5

En esta sección se expondrán los procedimientos esenciales de Racket haciendo uso de una notación basada en EBNF.

El primer procedimiento que se expondrá es define. Este procedimiento permite introducir variables con su respectivo valor. Dichas variables estarán disponibles en todo lo que resta del programa. Su sintaxis es:

```
(define var val[23])
var ::= <id>
val ::= <Boolean>
      | <Number>
      | <Char>
      | <String>
      | <Symbol>
      | <id>
```

Donde `<id>` de la variante `var` es el nombre que tendrá la variable y `val` es el valor de cualquier tipo (u otro identificador) que se asociará a ese identificador, a manera de ejemplo:

```
1.(define x 1)
```

Otro procedimiento importante es lambda ya que nos permite generar una función, su sintaxis es la siguiente:

```
(lambda(args) body)
args ::= <id>*
body = <expr>
```

Donde `<id>*` son los nombres de los parámetros del procedimiento (cualquier cantidad de parámetros, separados por espacio) y `<expr>` es el cuerpo de lo que se quiere que haga el procedimiento.

---

23 Aunque el lenguaje Racket tiene más tipos básicos, se han expuesto únicamente los más usados para realizar las prácticas del curso.

Para el ejemplo se usará también el procedimiento define

```
1.(define cuadrado (lambda(x) (* x x)))
2.(define suma (lambda(x y) (+ x y)))
```

Para poder aplicar un procedimiento, la sintaxis es:

```
(proc args*)
proc ::= <id>
       | (lambda(args) body)
args ::= <Boolean>
      | <Number>
      | <Char>
      | <String>
      | <Symbol>
      | <id>
```

Donde `proc` puede ser un identificador cuyo valor sea un procedimiento o bien un nuevo procedimiento declarado ahí mismo, mientras `args` presenta a los argumentos que espera recibir (cualquier cantidad de argumentos, separados por espacio), siguiendo con el ejemplo tenemos:

```
3.(cuadrado 3)
```
A lo que el intérprete responderá con:
```
> 9
```

Aplicando el otro procedimiento:
```
4.(suma 3 7)
> 10
```

Aplicando un procedimiento sin nombrarlo:
```
4.((lambda(x) x) 5)
> 5
```

Racket también facilita el manejo de listas, para poder ahondar en el tema, a continuación se verá cómo construirlas usando el procedimiento `list` o el procedimiento `quote`.

```
(list args*)
args ::= <Boolean>
       | <Number>
       | <Char>
       | <String>
       | <Symbol>
       | <id>
```

o

```
(quote (args*))
args ::= <Boolean>
       | <Number>
       | <Char>
       | <String>
       | <Symbol>
       | <id>
```

Donde `args` serán los valores que contendrá la lista, separados por espacio.

Siguiendo con los ejemplos:

```
1. (list 1 2 3)
2. (quote (1 2 3))
```

En ambos casos, el resultado que obtendremos será:

```
>  '(1 2 3)
```

El resultado es la manera en que Racket representa internamente las listas, sin embargo, es posible construir listas con esta sintaxis, es decir, usando el símbolo `'`.

Un par de ejemplos para construir listas de esta nueva forma son:

1. `'(4 5 6)`
2. `'(4 5 6)`

Esta sintaxis es mucho más cómoda que las anteriores debido a la brevedad con la que permite declarar una lista.

Para operar con los elementos de las listas, veremos los procedimientos car y cdr. El procedimiento car regresa el primer elemento de la lista, mientras que el procedimiento cdr regresa la cola de la lista. La sintaxis de cada un de ellas es:

```
(car lst)
(cdr lst)
lst ::= <id>
      | (list args)
      | (proc args)
```

Dado que `car` y `cdr` operan sobre listas, se espera que el argumento `lst` sea una lista, ya sea que se pase un identificador cuyo valor sea una lista, una lista nueva o la aplicación de un procedimiento que regrese una lista.

A manera de ejemplo:

```
1.(car '(1 2 3 4 5 6))
2.(cdr '(1 2 3 4 5 6))
```

El intérprete responderá en cada caso lo siguiente:

```
> 1
> '(2 3 4 5 6)
```

Un procedimiento útil para el manejo de listas es map, cuya sintaxis es:

```
(map proc lsts*)

proc ::= <id>

       | (lambda(args) body)

lsts ::= <id>

       | (list args)

       | (fun args)
```

Siendo `lsts` una o más listas y `proc` un procedimiento cuya aridad deberá ser igual al número de listas que se hayan proporcionado en la función. Este procedimiento aplicará la función `proc` tomando como parámetros la cabeza de cada lista, para regresar una nueva lista que contiene el resultado de `proc` en orden. Se ejemplifica a continuación:

```
1.(map + '(1 2 3) '(4 5 6))
```

El resultado al dejar que el intérprete lo evalúe es:

```
> '(5 7 9)
```

Este resultado se obtuvo porque el primer elemento fue el resultado de sumar `1 + 4`, el segundo elemento de evaluar `2 + 7` y por último `3 + 6`.

Los últimos procedimientos que se expondrán en esta sección son foldr y foldl.
La sintaxis es:

```
(foldl proc init lsts*) o (foldr proc init lsts*)

proc ::= <id>

       | (lambda(args) body)

init ::= <Boolean>

      | <Number>

      | <Char>

      | <String>

      | <Symbol>

      | <id>
```

```
lsts ::= <id>
       | (list args)
       | (proc args)
```

Donde `lsts` representan una o más listas, `init` el valor inicial con el que se comenzará a operar y `proc` un procedimiento que deberá tener una aridad de `n + 1` siendo `n` el número de listas pasadas al procedimiento.

El procedimiento `proc` es inicialmente llamado con el primer elemento de cada lista, y el elemento final será `init` (el elemento `n+1`). En las siguientes llamadas, el elemento `init` será el valor resultante de la llamada recursiva anterior de `proc`.

Las listas de entrada son recorridas de izquierda a derecha en `foldl` y de derecha izquierda en `foldr` siendo el resultado de toda la aplicación de `fold` el resultado de la última aplicación de `proc`. Si las listas están vacías, el resultado es `init`.

A continuación se presenta en ejemplo para calcular la suma de todos los enteros en una lista haciendo uso de `foldl`:

```
(foldl (lambda(x init) (+ init x)) 0 '(1 2 3 4))
```

A lo que el intérprete responderá con:

```
> 10
```

Analicemos cómo llegó el intérprete a este resultado:
El procedimiento pasado a foldl fue `(lambda(x init) (+ init x))`. El primer parámetro formal es llamado `x` y éste será el elemento de la lista según el nivel de recursión, mientras que el segundo será el acumulado de las llamadas recursivas anteriores. El valor de regreso del procedimiento es la suma de estos valores.

El primer paso es tomar la cabeza de las listas, debido a que en este caso sólo se pasó como argumento una lista, únicamente se toma la cabeza de dicha lista, es decir el valor `1`. A continuación, se toma el valor inicial (`init`) que es `0` y se pasan estos valores como argumentos al procedimiento descrito en el párrafo anterior. El procedimiento realizará la suma de los valores, es decir, `(+ 0 1)` que se evalúa a `1`.

El siguiente paso es tomar el resto de la lista y pasar como nuevo valor inicial (`init`) la suma previamente calculada, lo cual se podría escribir así:

```
(foldl (lambda(x init) (+ init x)) 1 ‘(2 3 4))
```

Esta llamada se resuelve de la misma manera, es decir, se toma la cabeza de la lista, (la cual tiene valor de 2) junto con el valor inicial (que es la suma de la llamada anterior, o sea 1) y se suman, quedando la expresión `(+ 2 1)` que se evalúa a 3.

De la misma manera, se toma la cola de la lista y se pasa este nuevo resultado como valor inicial (`init`), lo que se escribiría así:

```
(foldl (lambda(x init) (+ init x)) 3 ‘(3 4))
```

Se continúa de la misma manera hasta que la lista se queda vacía y el resultado es el valor que almacene `init`, lo que se vería así:

```
(foldl (lambda(x init) (+ init x)) 10 ‘())
```

El procedimiento `foldl` recorrió las listas de izquierda a derecha, por lo que la suma que se hizo fue 0+1+2+3+4, si en cambio se utilizara el procedimiento `foldr`, el recorrido de las listas se haría en el sentido inverso, por lo que la suma que se realizara sería 0+4+3+2+1.