



PROYECTO DEL CURSO 2018-2 FUNDAMENTOS DE LENGUAJES DE PROGRAMACIÓN

Profesor: Jesús Alexander Aranda

Monitor: Santiago Giraldo

1. Implementando un Ruby básico usando SLLGEN

1.1 Qué es Ruby?

Ruby es un lenguaje de programación simple y poderoso orientado a objetos creado por Yukihiro Matsumoto. Como Perl, Ruby es bueno procesando texto. Como Smalltalk, cada cosa en ruby es un objeto. Ruby tiene bloques, iteradores, meta-clases, y más.

Se puede usar Ruby para escribir servidores, experimentar con prototipos y para tareas de programación de la vida diaria, Como un lenguaje orientado a objetos totalmente integrado, Ruby está bien balanceado. Algunas características:

- Sintaxis simple
- Características OO básicas (clases, métodos, objetos, etc),
- Características OO especiales (Mix-ins, métodos singleton, renaming, ...),
- Sobrecarga de operadores,
- Manejo de excepciones,
- Iteradores y clausuras,
- Garbage collection,
- Cargado dinámico (dependiendo de la arquitectura),
- Alta portabilidad (corre en varios sistemas Unix, Windows, DOS, OSX, OS/2, Amiga, etc)

Ruby FAQ: <https://ruby-doc.org/docs/ruby-doc-bundle/FAQ/FAQ.html>

Ruby Docs: <http://docs.ruby-doc.com/docs/ProgrammingRuby/>

Ruby Tutorial: <https://www.tutorialspoint.com/ruby/index.htm>

1.2 Por qué Ruby?

La sintaxis de ruby es sencilla y muy expresiva, debido a las limitaciones que se tiene definiendo la gramática con SLLGEN no se puede replicar la sintaxis de otro lenguaje como por ejemplo C/C++, e incluso la sintaxis más elaborada de ruby con muchas de sus funcionalidades, además estamos usando un lenguaje declarativo de intermediario como lo es racket/eopl.

1.3 Especificación Léxica y Gramatical

A continuación se especifica una léxica y una gramática que son aproximadas a las expresiones básicas que usa ruby incluyendo algunos detalles extras que son necesarios. Usted la deberá implementar la gramática con la especificación de slgen que se ha venido usando a lo largo del curso y realizar las respectivas funcionalidades para poder evaluarla y obtener las salidas esperadas de su ejecución.

Este semestre la gramática no se deja a definición del estudiante debido al poco tiempo con el que se cuenta.

1.3.1 Léxica

La lexica es la que se ha trabajado hasta ahora en el curso, pero adicionando pequeños cambios.

```
((white-sp (whitespace) skip)
 (comment ("#" (arbno (not #\newline)))) skip)
(identifier ((arbno "@") letter (arbno (or letter digit "_" "?" "=")))
            symbol)
(number (digit (arbno digit)) number)
(number ("-" digit (arbno digit)) number)
(text ("\" (or letter whitespace)
      (arbno (or letter digit whitespace ":" "?" "=" "'")) "\"))
      string)

)
```

1.3.1 Gramática

```
ruby-program := "ruby" exp-batch "end"      (a-program)

exp-batch:= expression (expression)* (a-batch)

expression ::= simple-exp                    (a-simple-exp)
           ::= "declare" identifier ("," identifier)* ";" (declare-exp)
```

```

    ::= "puts" comp-value ("," comp-value)* ";"      (puts-exp)
    ::= "if" comp-value ("then")* exp-batch
      ("elsif" comp-value ("then")* exp-batch)*
      ("else" exp-batch)* "end"                      (if-exp)
    ::= "unless" comp-value ("then")* exp-batch
      ("else" exp-batch)* "end"                      (unless-exp)
    ::= "while" comp-value ("do")* exp-batch "end"    (while-exp)
    ::= "until" comp-value ("do")* exp-batch "end"    (until-exp)
    ::= "for" identifier "in" comp-value ("do")*
      exp-batch "end"                                (for-exp)
    ::= "def" identifier "(" (identifier)*(",") ")"
      exp-batch
      "end"                                           (function-exp)
    ::= "super" identifier arguments ";"              (super-exp)

class-decl ::= "class" identifier ("<" identifier)*
            "attr" (":" identifier)*(",") ";"
            (method-decl)* "end"                      (class-exp)

method-decl ::= "def" identifier "(" (identifier)*(",") ")"
              exp-batch "end"                        (a-method-decl)

simple-exp  ::= simple-value complement ";"            (val-exp)

complement ::= "=" comp-value calls                  (assign)
            ::= assign-op comp-value calls            (assign-and)
            ::= calls                                  (comp-calls)

calls      ::= (call)*                                (some-calls)

call       ::= "." identifier arguments                (method-call)
            ::= arguments                             (arguments-call)

arguments  ::= "(" (comp-value)*(",") ")"              (m-arguments)
            ::= "[" (comp-value)+(",") "]"              (arr-arguments)

comp-value ::= value                                  (op-value)
            ::= unop comp-value                       (unop-value)

value      ::= simple-value                           (simple-val)
            ::= "(" value val-compl ")"                (call-val)

val-compl  ::= calls                                  (val-call)
            ::= bin-op value                           (binop-val)

```

```

simple-value ::= identifier      (id-val)
               ::= integer       (num-val)
               ::= float         (float-val)
               ::= text          (str-val)
               ::= "true"        (true-val)
               ::= "false"       (false-val)
               ::= "nil"         (nil-val)
               ::= "[" (comp-value)*(,) "]" (arr-val)

```

```

bin-op ::= "+"      (add)
           ::= "-"      (diff)
           ::= "*"      (mult)
           ::= "/"      (div)
           ::= "%"      (mod)
           ::= "**"      (pow)
           ::= ">"      (great)
           ::= ">="     (great-eq)
           ::= "<"      (less)
           ::= "<="     (less-eq)
           ::= "=="     (is-equal)
           ::= "!="     (not-equal)
           ::= "and"    (and-op)
           ::= "&&"     (and-op)
           ::= "or"     (or-op)
           ::= "||"     (or-op)
           ::= ".."     (in-range)
           ::= "..."  (ex-range)
           ::= "step"   (st-range)

```

```

assign-op ::= "+="    (add-eq)
              ::= "-="    (diff-eq)
              ::= "*="    (mult-eq)
              ::= "/="    (div-eq)
              ::= "**="    (pow-eq)

```

```

un-op ::= "not"      (not-op)
          ::= "!"       (not-op)

```

2. [70 pts] Primera Parte: Ruby sin Objetos

A continuación se presentarán las pautas a desarrollar durante el proyecto. Debido al alcance que tiene el curso y cómo está ligado a la implementación de conceptos dadas en el libro *Essentials of Programming Languages* no se harán cambios significativos que no se puedan solucionar por medio de una interfaz, es decir, no se modificarán los tipos de datos que se han visto como por ejemplo: **closure** o **environment**, para cumplir con el comportamiento de ruby.

En esta sección se explicarán los diferentes tipos de datos de la gramática usando la notación extendida de Backus-Naur (EBNF), y se darán diferentes ejemplos. También se mostrarán ejemplos de la ejecución en el interpretador de racket.

2.1 Detalles de la Gramática

2.1.1 ruby-program

Un **ruby-program** es la representación que se usará en el curso para un programa que estará escrito bajo la sintaxis de ruby especificada anteriormente en la gramática.

Un **ruby-program** está definido como:

$\langle \text{ruby-program} \rangle ::= \text{ruby } \langle \text{exp-batch} \rangle \text{ end}$

a-program(body)

donde **exp-batch** está descrito a continuación. La salida de un **ruby-program** es la ejecución de las instrucciones en **exp-batch**.

2.1.2 exp-batch

Un **exp-batch** representa un lote (batch en inglés) de expresiones del lenguaje, donde hay al menos una expresión.

Un **exp-batch** está definido como:

$\langle \text{exp-batch} \rangle ::= \langle \text{expression} \rangle \{ \langle \text{expression} \rangle \}^*$

a-batch(exp exps)

la salida de un **exp-batch** es el valor de la evaluación de la última expresión. si el valor de la última expresión es vacío se debe retornar el símbolo '=>nil

2.1.3 expression

Un **expression** representa, como su nombre lo indica, una expresión del lenguaje ruby definido en la especificación gramatical propuesta.

`<expression> ::= <simple-exp>`

`a-simple-exp(exp)`

`::= declare {<identifier>}*(,) ;`

`declare-exp(idss)`

`::= puts {<comp-value>}*(,) ;`

`puts-exp(vals)`

`::= if <comp-value> {then}* <exp-batch>
 {elsif <comp-value> {then}* <exp-batch>}*
 {else <exp-batch>}* end`

`if-exp(a-comp-value then-exp
 else-ifs elsif-then else)`

`::= unless <comp-value> {then}* <exp-batch>
 {else <exp-batch>}* end`

`unless-exp(a-comp-value then-exp else)`

`::= while <comp-value> {do}* <exp-batch> end`

`while-exp(a-comp-value body)`

`::= until <comp-value> {do}* <exp-batch> end`

`until-exp(a-comp-value body)`

`::= for <identifier> in <comp-value> {do}*
 <exp-batch> end`

`for-exp(id a-range body)`

`::= def <identifier> ({<identifier>}*(,))
 <exp-batch> end`

`function-exp(proc-name idss body)`

`::= return <comp-value>;`

`return-exp(a-comp-value)`

2.1.4 simple-exp

Un **simple-exp** representa una expresión que se encarga de las asignaciones y los llamados a métodos mediante el uso de un complemento. También se le puede aplicar una operación unitaria a lo que se evalúe. **simple-exp** está definido como:

`<simple-exp> ::= <simple-value> <complement> ;`

`val-exp(a-simple-value a-complement)`

2.1.5 complement

Un **complement** puede representar la asignación, la operación y asignación, o un llamado a función mediante unos argumentos

`<complement> ::= = <comp-value> <calls>`

`assign(a-comp-value call-list)`

`::= <assign-op> <comp-value> <calls>`

`assign-and(an-op a-comp-value calls)`

`::= <calls>`

`comp-calls(some-calls)`

2.1.6 comp-value

Un **comp-value** es un valor compuesto, este representa lo que sería una operación, o la negación de una operación. Está definido como:

`<comp-value> ::= <value>`

`op-value(val)`

`::= <un-op> <comp-value>`

`unop-value(un-op val)`

2.1.8 value

Un **value** es un valor que representa un valor simple o la aplicación de un complemento sobre una valor, encerrado en paréntesis.

`<value> ::= <simple-value>`

`simple-val (a-simple-value)`

`::= (<value> <val-compl>)`

`call-val(val a-val-compl)`

2.1.9 val-compl

Un **val-compl** es el complemento que se le puede hacer a un valor cuando se está operando, representa llamadas o la aplicación de **bin-op** a otro **value**

`<val-compl> ::= <calls>`

`val-call(some-calls)`

`::= <bin-op> <operation>`

`binop-val(binop val)`

2.1.10 simple-value

Un **simple-value** o valor simple, es la mínima representación de un dato, puede ser según la gramática:

`<simple-value> ::= <identifier> | <number> | <text> | true
| false | nil`

`::= '[' {comp-value}+(',') ']`

`arr-val(vals)`

*nota: identifier, number, text están dados por la léxica,
cada uno seria: id-val(id), num-val(val), str-val(val)*

2.1.11 calls

Es la representación de una lista de **call**, está representado como:

`<calls> ::= {<call>}*`

`calls(some-calls)`

2.1.12 call

Es la representación de 2 tipos de llamados:

- 1) hacer un llamado con argumentos, en este caso de acuerdo a la estructura de **<simple-exp>** **<complement>** se tomaría un identificador y se tendría algo como por ejemplo:

operación(1,2...,n)	llamado a método con id: operación
matriz[1,2,...,n]	acceder a un arreglo con id: matriz

- 2) hacer un llamado a un método de un objeto (esto se vería en la segunda parte del proyecto)

ejemplo.metodo(1,2...,n)	llamado a método del objeto ejemplo
--------------------------	-------------------------------------

por lo que **call** está definido de la siguiente manera:

<call> ::= <arguments>

arguments-call(args)

::= . <identifier> <arguments>

method-call(method-name args)

2.1.13 arguments

Es la representación de los tipos de argumentos de los cuales dispone call, los argumentos pueden venir entre paréntesis () , para llamados a procedimientos, o corchetes, para acceder un elemento de un arreglo

<arguments> ::= ({<comp-value>}*^(,))

m-arguments(args)

::= [{<comp-value>}^{+^(,)}]

arr-arguments(args)

2.1.14 bin-op

Son algunas de las operaciones que permite ruby oficialmente

<bin-op> ::= + | - | * | / | % | ** | > | >= | < | <=
| == | != | and | && | or | || | .. | ...
| step

2.1.15 assign-op

Es azúcar sintáctico, son las operaciones que permiten operar sobre el mismo identificador, ejemplo: “a = a+1” se puede escribir como “a+=1”

<assign-op> ::= += | -= | *= | /= | **=

2.1.16 un-op

Son operaciones unitarias sobre los valores de ruby:

<un-op> ::= not | !

2.3 Para Implementar, Detalles Adicionales y Ejemplos

En esta sección se mostrarán diferentes entradas con sus respectivas salidas, incluyendo una ejecución en un interpretador de ruby

2.3.1 [10 pts] Operaciones

Normalmente en ruby tendríamos la flexibilidad de hacer una operación inorden sin tantas restricciones y teniendo en cuenta la prioridad de las operaciones, ejemplo:

```
ruby 2.5.0p0 (2017-12-25 revision 61468) [x86_64-linux]
>
> 1+2+3+4
>
=> 10
> 2*6+4/2
>
=> 14
> 
```

Aunque es más fácil expresar una operación en notación preorden en este caso se operará de forma inorden pero con ciertas restricciones las cuales establecen los tipos de dato **operation**, **value** y **val-compl**. Aquí algunos ejemplos de operaciones válidas:

Entrada	Salida
puts (1+(2+(3+4)));	10
puts ((1+2)+(3+4));	10
puts ((1+(2+3))+4);	10
puts ("hola"+" mundo");	hola mundo

<code>puts (["hola"] + ["mundo"]);</code>	<code>("hola" "mundo")</code>
<code>puts ("hola"+2);</code>	 <i>no implicit conversion of Integer into String</i>
<code>puts (2+"hola");</code>	 <i>String can't be coerced into Integer</i>
<code>puts ("hola"*2);</code>	<code>holahola</code>
<code>ruby declare a; a = 3; puts (a / " mundo"); end</code>	 <i>String can't be coerced into Integer</i>
<code>puts ([1,2]*2)</code>	<code>(1 2 1 2)</code>

2.3.1.1 Otros tipos de errores

Estos mensajes de error no son exactos a los que usa ruby pero tienen más o menos el mismo estilo



Operación	Error
boolean <bin-op> number	<i>Not defined method <bin-op> for boolean</i>
number <bin-op> boolean	<i>Boolean can't be coerced into Integer</i>
number <bin-op> array	<i>Array can't be coerced into Integer</i>
array <bin-op> number excepto con la operación: *	<i>No implicit conversion of Integer into Array</i>
number boolean string <calls>	<i>Can't apply args to ~a</i>

2.3.2 [10 pts] Expresión: declare-exp

En ruby no tenemos una expresión llamada **declare**, pero para facilitar un poco las cosas es necesaria incluirla en la especificación gramatical para ir acorde al contenido del curso.

Debido al manejo de ambientes que se tiene en el curso para poder evaluar una expresión es necesario extender un ambiente de uno previo, y hacer que este nuevo ambiente contenga las declaraciones de las variables a utilizar con sus respectivos valores, normalmente se hace esta operación evaluando un **let-exp**. Sin embargo ésta expresión no existe explícitamente en el lenguaje ruby, aun así se puede lograr el mismo efecto con **declare-exp** cuando esta se está evaluando en un **exp-batch**. Al encontrarnos con un **declare-exp(ids)** lo que se debe hacer es evaluar las expresiones siguientes del **exp-batch** con un ambiente extendido el cual tenga los **ids** con valor inicializado en 'nil', y después cambiar este valor usando la asignación.

Ejemplos:

Entrada	Salida
<pre>ruby declare a,b,c; a = 5; b = 6; c = 10; puts a,b,c; end</pre>	<pre>5 6 10 =>nil</pre>
<pre>ruby declare a,b,c; b = 6; c = 10; puts a,b,c; end</pre>	<pre>nil 6 10 =>nil</pre>
<pre>ruby declare a,b,c; b = 6; c = 10; puts a,b,d; end</pre>	 <pre>Error: undefined local variable or method d</pre>
<pre>ruby puts d; declare d; end</pre>	 <pre>Error: undefined local variable or method d</pre>

2.3.3 [18 pts] Expresión simple

2.3.3.1 Expresión simple: val-exp

Un **val-exp** está conformado por un valor simple y un complemento, como se mencionó anteriormente un complemento puede ser: assign, assign-and, y calls. Por lo que se pueden tener las siguientes entradas:

Entrada	Salida
<pre>ruby declare a; #id a, assign 5 (calls '()') a = 5; puts a; #id a, assign-and += 1 (calls '()') a += 1; puts a; end</pre>	<pre>5 6 =>nil</pre>
<pre>ruby def print(val) puts val; end # id print, calls '(...)' print("hola mundo"); end</pre>	<pre>hola mundo =>nil</pre>
<pre>ruby def add1(val) val += 1; return val; end add1(100);# id: add1, calls '(100)' end</pre>	<pre>101</pre>
<pre>ruby declare a; a = [[1,2],3]; puts (a[0][0]); end</pre>	<pre>1 =>nil</pre>
<pre>ruby declare a; a = [1,2,3,4,5,6,7]; puts (a[0,4]); a[0,4];</pre>	<pre>1 2 3 4 (1 2 3 4)</pre>

end	
-----	--

2.3.4 [14 pts] Expresión: function-exp

Los métodos en ruby (procedimientos y funciones) son por defecto recursivos, y nos encontramos en una situación parecida a la anterior, y como tal la solución es parecida, en este caso no entenderíamos un ambiente de forma normal como con un **let-exp**, en este caso es necesario extender un ambiente recursivamente y evaluar el resto de **exp-batch** con ese ambiente. El único inconveniente es que no podríamos tener llamados recursivos cruzados entre 2 funciones o más, pero llegar a hacer eso no será tomado en cuenta.

Ejemplos:

Entrada	Salida
<pre>ruby def fact (n) if (n == 0) then return 1; else puts "el valor de n es ",n; return (n * (fact ((n - 1)))); end end puts "el factorial de 5 es", (fact(5)); end</pre>	<pre>el valor de n es 5 el valor de n es 4 el valor de n es 3 el valor de n es 2 el valor de n es 1 el factorial de 5 es 120 =>nil</pre>
<pre>ruby def fibo(n) if (n <= 2) then return 1; else declare left,right; left = (n - 1); right = (n - 2); return ((fibo (left)) + (fibo (right))); end end puts (fibo(10)); end</pre>	<pre>55 =>nil</pre>

<pre> ruby def to_text(num) declare text; text = ["cero","uno", "dos","tres", "cuatro","cinco", "seis", "siete", "ocho", "nueve"]; return (text[num]); end to_text(5); end </pre>	<pre> "cinco" </pre>
---	----------------------

2.3.5 [4 pts] Expresión: if-exp

Esta expresión está conformada por una condición, una lista de condiciones alternativas (elsif), una lista de cuerpos de qué hacer si alguna de las alternativas se cumple, y un cuerpo de else que aunque la gramática permite poner varios, solo se aceptaría uno. Ejemplos:

Entrada	Salida
<pre> #Si pudiéramos otro else este no #tendría efecto ruby declare x; x = 3; if (x > 5) then puts x; elsif (x < 3) then puts "mundo"; else puts "hola"; else puts "otro else"; end end </pre>	<pre> #si x = 3 hola =>nil #si x = 2 mundo =>nil #si x = 6 6 =>nil </pre>

2.3.6 [4 pts] Expresión: unless-exp

La expresión unless-exp funciona como un if, pero se evalúa la primera condición si esta es falsa (contrario a **if** que evalúa si la condición es verdadera)

Entrada	Salida
<pre>ruby unless false then puts 4; end end</pre>	<pre>4 =>nil</pre>
<pre>ruby unless true then puts 4; else puts "hola mundo"; end end</pre>	<pre>hola mundo =>nil</pre>
<pre>#al igual por la gramática se #pueden poner muchos else, pero #solo uno cuenta ruby unless true then puts 4; else puts "hola mundo"; else puts "otro else"; end end</pre>	<pre>hola mundo =>nil</pre>

2.3.7 [5 pts] Expresión: while-exp y until-exp

El ciclo **while** como todos lo conocemos ejecuta un cuerpo mientras se cumpla cierta condición, es decir, que la condición sea verdadera. El ciclo **until** se ejecuta mientras la condición que evalúa sea falsa, es decir, hasta que la condición sea verdadera.

Entrada	Salida
<pre>ruby declare x; x = 0; while (x < 10) do puts x; x += 1; end end</pre>	<pre>0 1 2 3 4 5 6 7 8 9 #<void></pre>
<pre>ruby declare x; x = 0; until (x >= 10) do puts x; x += 1; end end</pre>	<pre>0 1 2 3 4 5 6 7 8 9 #<void></pre>

2.3.8 [5 pts] Expresión: for-exp

Un ciclo **for** en ruby funciona sobre secuencias o rangos, en el caso de nuestro interpretador hemos definido dos operadores: `..` y `...` los cuales deben generar un tipo de dato range

`<range> ::= <number> <number> <number>`

`a-range(start end step)`

Además tenemos un operador llamado **step** el cual nos ayuda a modificar el incremento/decremento que queramos hacer cuando recorramos el rango. Ejemplos:

Entrada	Salida
<pre>ruby for x in (1..10) do puts x; end end</pre>	<pre>1 2 3 4 5 6 7 8 9 10 #<void></pre>
<pre>ruby for x in (1...10) do puts x; end end</pre>	<pre>1 2 3 4 5 6 7 8 9 #<void></pre>
<pre>ruby for x in ((1..10) step 2) do puts x; end end</pre>	<pre>1 3 5 7 9 #<void></pre>

A) Nombres recomendados de procedimientos a implementar

Esta es una lista de nombres de procedimientos y sus argumentos que pueden usar para evaluar desde un **ruby-program** a un **simple-value**:

<ul style="list-style-type: none">- eval-program: pgm- eval-exp-batch: a-batch env- eval-a-batch: exp exps env- eval-simple-exp: s-exp env- eval-comp-value c-val env- eval-if-exp: if-list then-list else-batch env- eval-value: a-val env- apply-call-list a-value c-list env- apply-call a-value a-call env- (eval-binop binop) -> lambda: val1 val2- eval-range: a-range	<ul style="list-style-type: none">- eval-val-compl: a-val a-v-compl env- eval-simple-value: s-val env- just-simple-value: s-val env- eval-comp-calls: c-list calls env- apply-assing-and: as-op var-val a-val- apply-complement: s-val compl env- times-string: a-string times- apply-arguments: a-val args env- just-eval-arguments: args env- eval-arguments: args env- iota-range: start end step
---	--

3. [30 pts] Segunda Parte: Ruby con Objetos

Modificaremos ahora la gramática que define un programa de ruby, **ruby-program** será ahora:

```
ruby-program := "ruby" (class-decl)* exp-batch "end" (a-program)
```

donde las declaraciones de las clases anteceden al programa principal.

3.1. [20 pts] Como pudimos observar en la primera parte no se abordó la declaración de clases (**class-decl**) y tampoco se revisó lo que era **super-exp**. En esta parte del proyecto se buscará proveer de objetos nuestro lenguaje ruby. Originalmente en ruby todos los elementos como **simple-val** son objetos, pero no es la finalidad de este proyecto. Usted deberá implementar lo siguiente con base en el interpretador de objetos simples.

En el ruby oficial las clases se definen de la siguiente forma, Ejemplo:

```
class Vector2D
  attr :x, :y; #atributos
```

```

def initialize(x,y) #Se ejecuta al hacer .new(...)
  @x = x
  @y = y
end

def x()
  @x      #Retorna el valor de @x
end

def x=(x)
  @x = x   #Permite cambiar el valor de @x
end

def y()
  @y      #Retorna el valor de @y
end

def y=(y)
  @y = y   #Permite cambiar el valor de @y
end
end

```

También existen otras propiedades pero el ejemplo anterior y el siguiente definen todo lo que necesitamos saber para utilizar en nuestro proyecto. La gramática que se presentó en el inicio de este documento ya contiene la declaración de clases que explicaremos a continuación usando ENBF. Ciertamente hay pequeños detalles tales como ; al finalizar ciertas sentencias, pero esto fue necesario al trabajar con SLLGEN.

Como está la gramática que se ha definido, cuando se crea un objeto en nuestro interpretador (por ejemplo: Vector2D), los campos de la clase quedarían declarados como '(x y)', al momento de crear el objeto y obtener los campos de la clase es necesario adicionar el símbolo '@' a cada uno para que tenga efecto la evaluación de una variable en el ambiente.

3.1.2 class-decl

Es la representación de una clase de ruby, una clase tiene un nombre dado por un identificador, una lista de nombres de clases donde hereda, una lista de atributos, y unas declaraciones de métodos.

```

<class-decl> ::= class <identifier> {< <identifier>}*
               attr {: <identifier>}* {method-decl}* end

```

<pre> a-class-decl(class-name super-names field-ids m-decls) </pre>

3.1.2 method-decl

Es la representación de un método de una clase, gramaticalmente está definido igual que un **function-exp** pero no se evalúa igual.

```
<method-decl> ::= def <identifier> ({<identifier>}*(,))  
                  <exp-batch> end
```

method-decl(method-name idss body)

3.2. [10 pts] Como hemos visto en POO una clase puede heredar de una clase ya definida, en nuestro caso cuando la lista super-names esté vacía, como el caso de Vector2D, la clase va a heredar de **'object'**. El siguiente es un ejemplo de herencia en ruby:

```
class Vector3D < Vector2D  
  attr :z; #atributos  
  
  def initialize(x,y,z)  
    super(x,y)  
    @z = z  
  end  
  
  def z()  
    @z      #Retorna el valor de @z  
  end  
  
  def z=(z)  
    @z = z  #Permite cambiar el valor de @z  
  end  
  
end
```

La clase Vector3D puede acceder a los métodos de la clase padre.

3.2.1 Expresión: super-exp

Esta expresión se encarga de llamar un método que se encuentre en el padre (clase de donde hereda) de la clase, normalmente ruby infiere el nombre del método en la clase padre al cual debe llamar, en el caso de la implementación que estamos haciendo es necesario especificar el nombre del método al que se está llamando.

```
<super-exp> ::= super <identifier> ({<comp-value>}*(,)) end
```

super-exp(method-name args)

Con lo que se ha visto hasta ahora también es necesario revisar el caso **method-call** perteneciente al tipo de dato **call**, recordemos:

3.2.1 call

Está definido de la siguiente manera:

`<call> ::= <arguments>`

`arguments-call(args)`

`::= . <identifier> <arguments>`

`method-call(method-name args)`

cuando se evalúa un **simple-exp** este tiene un complemento, cuando hay asignación hay unos llamados, o hay llamados solamente; en cualquier caso si un identificador se ve acompañado de un **method-call** debe evaluarse si el identificador es un objeto o una clase, si es un objeto simplemente se llama a **find-method-and-apply** con la información requerida; en el caso de que sea el nombre de una clase, el **method-name** llamar debe ser **'new'** donde al hacer **find-method-and-apply** se debe buscar el método **initialize** definido en el código que nosotros ingresamos y regresar una nueva instancia, si el método initialize no existe no debería de pasar nada, solo se retorna un objeto con sus valores por defecto.

3.3 Ejemplo de un programa con objetos

Adicional a lo que se tiene regularmente en el ruby oficial, en nuestro caso si queremos imprimir mediante **puts** un objeto, es necesario tener definido el método **to_str** el cual lo único que haría es retornar un valor de nuestra preferencia

```
ruby
class Integer
  attr :num;
  def initialize(num)
    @num = num;
  end

  def num=(num)
    @num = num;
  end

  def to_s()
    return @num;
  end
end

class Float < Integer
  attr :decm;
```

```

        def initialize(ent,decm)
            @decm = decm;
            super initialize(ent);
        end
        def to_s()
            return (@num + (@decm/(@decm * 10)));
        end
    end
    #Main program
    declare a;
    a = Float.new(1,1);
    puts a;      #Salida: 11/10 === 1.1
    a.num=(5);
    puts a;      #Salida: 51/10 === 5.1
end

```

3.4 Ejemplo de Vector2D y Vector3D

```

ruby
class Vector2D
    attr :x, :y; #atributos

    def initialize(x,y)
        @x = x;
        @y = y;
    end

    def x()
        return @x;      #Retorna el valor de @x
    end

    def x=(x)
        @x = x;  #Permite cambiar el valor de @x
    end

    def y()
        return @y;      #Retorna el valor de @x
    end

    def y=(y)
        @y = y;  #Permite cambiar el valor de @x
    end
end

```

```

class Vector3D < Vector2D
  attr :z; #atributos

  def initialize(x,y,z)
    @z = z;
    super initialize (x,y);
  end

  def z()
    return @z;      #Retorna el valor de @z
  end

  def z=(z)
    @z = z;  #Permite cambiar el valor de @z
  end
end

declare a,b;
a = Vector2D.new(1,2);
b = Vector3D.new(3,4,5);
puts (a.x()); #Salida: 1
puts (b.z()); #Salida: 5

end

```

4. Condiciones de Entrega

El proyecto se recomienda que sea resuelto en grupos de no más de 4 personas.

Para la entrega es necesario enviar un archivo comprimido (**.zip/tar/rar**) que contenga:

- Un archivo llamado **ruby_sin_objetos.rkt** donde se encuentre el desarrollo de la primera parte.
- Un archivo llamado **ruby_con_objetos.rkt** donde se encuentre el desarrollo de la segunda parte.
- Un archivo llamado **pruebas.txt** donde haya escrito varias entradas para el interpretador

Recuerde poner en un comentario al inicio de los archivos **.rkt** con los integrantes y sus respectivos códigos, además documente las funciones auxiliares que implemente las cuales deben estar en el mismo archivo. El archivo **pruebas.txt** también debe tener los nombres de los integrantes con sus códigos.

Debido al estilo de programación no debe usarse la funcionalidad **set!** de racket aunque eopl lo permita (debido a que no hay que salirse de los conceptos/implementación que ofrece el curso), no usar **set!** no implica no poder hacer asignación en el interpretador ya que la implementación que se ha dado en el curso usa principalmente **vector-set!** al trabajar con los ambientes.

Solamente cuando evalúe un **exp-batch**, ciclos **while**, **until**, **for**, **aplicar complement** o **calls** es necesario usar **begin**.

Hay muchas funciones ya implementadas que permiten un manejo de los tipos de datos como las listas (en eopl están: **append**, **map**, **for-each** y otras más) que están en las librerías del propio racket y se pueden importar si lo considera necesario.

Cuando en la léxica se definen las comillas que encierran un texto es necesario eliminarlas al evaluar un **str-val**, debido a que si por ejemplo se hace: **"a"+"b"** en el interpretador este queda como **"a""b"**, para eliminar las comillas agregue (require racket/string) después de **"#lang eopl"** y use la función **string-trim**.