PROYECTO DE LENGUAJES DE PROGRAMACION

SPRINT FINAL

LENGUAJE RUBY

INTEGRANTES:

RIZZO AGUAYO HECTOR IVAN

DEL ROSARIO CARRASCO MARCO STEVEN

PROFESOR:

SARAGURO BRAVO RODRIGO ALEXANDER

Contenido

INTRODUCCION

Ruby es uno de los lenguajes que contiene un balance cuidado. Su creador, Yukihiro “Matz” Matsumoto, mezcló partes de sus lenguajes favoritos (Perl, SmallTalk, Eiffel, Ada y Lisp) para formar un nuevo lenguaje que incorpora tanto la programación funcional como la imperativa.

Desde su liberación publica en 1995, Ruby ha atraído devotos desarrolladores de todo el mundo. En el 2006, Ruby alcanzó reconocimiento masivo, formándose grupos de usuarios activos en las ciudades más importantes del mundo y llenando las capacidades de las conferencias relacionadas a Ruby.

El índice TIOBE que mide el crecimiento de los lenguajes de programación, ubica a Ruby entre los diez mejores del ranking mundial. Gran parte del crecimiento se atribuye a la popularidad del software escrito en Ruby, particularmente el framework Ruby on Rails.

Ruby es totalmente libre. No solo gratis, sino también libre para usarlo, copiarlo, modificarlo y distribuirlo.

Es considerado un lenguaje flexible, ya que permite a sus usuarios alterarlo libremente. Las partes esenciales de Ruby pueden ser quitadas o redefinidas a placer. Se puede agregar funcionalidad a partes ya existentes. Intentando no restringir al desarrollador.

COMPONENTES LEXICOS

Variables

Existen 3 tipos de variables en Ruby: *las globales, las de instancia, las de clase y las locales.*

Las ***globales*** se declaran con un signo de dólar ($) al inicio y pueden ser accedidas desde cualquier parte del script. Por lo general tienen un valor especial llamado nil si es que no se le indica un valor inicial.

```
ruby> $foo

nil

ruby> $foo = 5

5

ruby> $foo

5
```

Este tipo de variables tienen que usarse con parquedad, ya que son peligrosas porque se pueden modificar desde cualquier lugar del código. Siempre que se encuentre con la necesidad de utilizar una variable global, hay que darle un nombre descriptivo para que no sea utilizada inadvertidamente para otra cosa.

Una característica notable de este tipo de variables es que se pueden trazar, es decir. Es decir, se puede definir un procedimiento que se llame cada vez que se modifique el valor de la variable.

```
ruby> trace_var:$x, proc{print "$x es ahora ", $x, "\n"}

nil

ruby> $x = 5

$x es ahora 5

5
```

Cuando una variable global se la atavía para funcionar como disipador que se llame cada vez que modifica, se la conoce como *variable activa*, muy útiles para mantener un GUI actualizado.

Existe un grupo especial de variables cuyos nombres constan del símbolo del dólar seguido de un carácter, las cuales son las principales variables del sistema y su respectivo significado.

$!	Último mensaje de error
$@	Posición del error
$_	Última cadena leída con gets
$.	Último número de línea leído por el interprete
$&	Última cadena que ha coincidido con una expresión regular
$~	Última cadena que ha coincidido con una expresión regular como array de subexpresiones

$n	La n-ésima subexpresión regular de la última coincidencia (igual que $~[n])
$=	flag para tratar igual las mayúsculas y minúsculas
$/	Separador de registros de entrada
$\	Separador de registros de salida
$0	El nombre del fichero del guión Ruby
$*	El comando de la línea de argumentos
$$	El número de identificación del proceso del intérprete Ruby
$?	Estado de retorno del último proceso hijo ejecutado

Las variables $_ y $~ tienen un ámbito local, sus nombres sugieren ámbito local pero son más útiles de esta forma.

Por otro lado, las variables ***de instancia***, son mayormente utilizadas en el desarrollo de aplicaciones web con Ruby on Rails. Están son declaradas con un signo de arroba al principio (@) y su característica es que pueden ser accedidas desde cualquier parte de la clase en la que fueron declaradas y a su vez, limitado al objeto al que referencia a sí mismo.

Desde el exterior del objeto, no se pueden alterar e incluso, no se pueden observar, es decir que nunca son públicas. Salvo el caso de los métodos proporcionados explícitamente por el programador.

Tal como el caso de las globales, las de instancia tienen el valor nil antes de que se inicialicen.

Estas no necesitan ser declaradas, dando como resultado una flexibilidad en la estructura de los objetos. De hecho, cada variable se añade dinámicamente al objeto la primera vez que se lo referencia.

```
class Ejemplo
  @global = ""
  def initialize()
    @global = "Uriel"
  end
  def ejecutar_global
    @global
  end
end
e = Ejemplo.new()
print e.ejecutar_global()
```

Tal como se indicó anteriormente, en este ejemplo se puede apreciar como la variable global es inicializada al comienzo de la clase y es llamada por diferentes métodos.

Para ser una variable ***de clase***, debe usar doble arroba (@@) antes de la variable y tienen que ser inicializadas, caso contrario, se generará un error. Pueden ser compartidos entre módulos o submódulos. Con este tipo de variables se debe tener mucho cuidado ya que la sobrecarga de las mismas producirá una advertencia.

Como ultimo punto se tiene las variables ***locales***, que comienzan con una letra minúscula o un subrayado (_). Cuando se llama a una variable local sin inicializar, se interpreta como llamar a un método sin parámetros. Existirá hasta el final del dominio actual hasta el momento.

En el caso de declarar una constante tienen que ser con letra mayúscula. Se las tiene que definir dentro de la clase o modulo y solo puede acceder desde dentro del mismo.

Palabras Reservadas

Como todo lenguaje de programación, Ruby tiene una lista de palabras reservadas que no pueden ser usadas como nombre de una variable y son de uso exclusivo. La siguiente tabla muestra las palabras con una breve explicación de cada una:

Palabra Reservada	Función
alias	Crea un alias para un operador, método o variable global que ya exista.
and	Operador lógico, igual a && pero con menor precedencia.

break	Finaliza un *while* o un *until loop*, o un método dentro de un bloque
case	Compara una expresión con una cláusula *when* correspondiente
class	Define una clase; se cierra con *end*.
def	Inicia la definición de un método; se cierra con *end*.
defined?	Determina si un método, una variable o un bloque existe.
do	Comienza un bloque; se cierra con *end*.
else	Ejecuta el código que continua si la condición previa no es *true*. Funciona con *if*, *elsif*, *unless* o case.
elsif	Ejecuta el código que continua si la condicional previa no es *true*. Funciona con *if* o *elsif*.
end	Finaliza un bloque de código.
ensure	Ejecuta la terminación de un bloque. Se usa detrás del ultimo *rescue*.
false	Lógico o Booleano *false*.
true	Lógico o Booleano *true*.
for	Comienza un loop *for*. Se usa con *in*.
if	Ejecuta un bloque de código si la declaración condicional es *true*. Se cierra con *end*.
in	Usado con el loop *for*.
module	Define un modulo. Se cierra con *end*.
next	Salta al punto inmediatamente después de la evaluación del loop condicional
nil	Vacio, no inicializado, invalido. No es igual a cero.
not	Operador lógico, igual como !.
or	Operador lógico, igual a // pero con menor precedencia.
redo	Salta después de un loop condicional.
rescue	Evalua una expresión después de una excepción es alzada. Usada después de *ensure*.
retry	Cuando es llamada fuera de *rescue*, repite una llamada a método. Dentro de *rescue* salta a un bloque superior.
return	Regresa un valor de un método o un bloque.
self	Objeto contemporáneo. Alude al objeto mismo.

super	Llamada a método del mismo nombre en la superclase.
then	Separador usado con *if*, *unless*, *when*, *case*, y *rescue*.
undef	Crea un método indefinido en la clase contemporánea.
unless	Ejecuta un bloque de código si la declaración condicional es *false*.
until	Ejecuta un bloque de código mientras la declaración condicional es *false*.
when	Inicia una clausula debajo de *under*.
while	Ejecuta un bloque de código mientras la declaración condicional es *true*.
yield	Ejecuta un bloque pasado a un método.
FILE	Nombre del archivo de origen contemporáneo.
LINE	Numero de la linea contemporánea en el archivo de origen contemporáneo.

Tipo de Booleanos

- True
- False

Caracteres especiales en Ruby

Operadores:
Como en toda operación entre dos o mas componentes, existen los operadores de:

```
t_PLUS = r'\+'
t_MINUS = r'\-'
t_TIMES = r'\*'
t_DIVIDE = r'\/'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_MOD = r'\%'
t_POW = r'\*\*'
t_PLUS_EQUAL = r'\+='
t_MINUS_EQUAL = r'-='
t_TIMES_EQUAL = r'\*='
t_DIVIDE_EQUAL = r'/='
t_MOD_EQUAL = r'\%='
t_POW_EQUAL = r'\*\*='
```

Por otra parte, existen operadores relacionales, los cuales nos retornan un valor booleano como resultado. Estos operadores son los siguientes:

```
t_EQUAL = r'\=\='
t_NOTEQUAL = r'\!\='
t_GREATERTHAN = r'\>'
t_GREATERTHANEQUAL = r'\>\='
t_LESSERTHAN = r'\<'
t_LESSERTHANEQUAL = r'\<\='
```

Existen también los operadores lógicos, que se evalúan de izquierda a derecha y la evaluación se detiene tan pronto como el resultado true o false es conocido.

- **AND** : and ó &&
- **OR** : or ó ||

Caracteres Especiales:
En Ruby existen una variedad de caracteres especiales usados. Los cuales se muestran a continuación:

```
t_CASE_EQUALITY = r'==='
t_COMBINED_COMPARISON_OP = r'\<=\>'
t_EXPONENT_AND = r'\*\*='
t_BINARY_XOR_OP = r'\^'
t_BINARY_AND_OP = r'&'
t_MATCHED_STRINGS_OP = r'=\~'
t_OPPOSITE_MATCHED_STRINGS_OP = r'\!\~'
t_OVERLOAD_PLUS = r'\+\@'
t_OVERLOAD_MINUS = r'-\@'
t_HASH_ROCKET = r'\=\>'
t_RANGE_INCLUSIVE = r'\.\.'
t_RANGE_EXCLUSIVE = r'\.\.\.'
t_UNARY_OP = r'::'
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_LKEY = r'\{'
t_RKEY = r'\}'
t_COMMA = r'\,'
t_DOT = r'\.'
t_OR_SYMBOL = r'\|'
t_NOT_SYMBOL = r'\!'
t_EQUAL_SYMBOL = r'='
t_OPTIONAL_SYMBOL = r'\?'
t_BINARY_LEFT_SHIFT_OP = r'\<\<'
t_BINARY_RIGHT_SHIFT_OP = r'\>\>'
t_COMPLEMENT_OP = r'~'
t_DOUBLE_QUOTED =   r'\"'
t_NUMBER_SIGN = r'\#'
t_LPAREN = r'\('
t_RPAREN = r'\)'
```

Símbolos

Un símbolo parece una variable, pero esta precedido de dos puntos (:) como, por ejemplo:

- :action
- :line_tines

Estos dos puntos son interpretados como “la cosa llamada”. No contienen valores como las variables. Mas bien, es considerado una etiqueta, un nombre, nada mas.

Un símbolo es el objeto mas básico que se puede crear en Ruby: es un nombre y una ID interna. Y son útiles porque dado un símbolo, se refiere al mismo objeto en todo el programa. Por lo tanto, son mas eficientes que los strings.

```
puts "hola".object_id   # 21066960
puts "hola".object_id   # 21066730
puts :hola.object_id    # 132178
puts :hola.object_id    # 132178
```

Como se puede apreciar, dos strings con el mismo nombre, son dos objetos distintos. Esto implica un ahorro de tiempo y memoria.

Entonces, *¿Cuándo es recomendable usar un string y cuando un símbolo?*

- Si el contenido es importante, **usar string.**
- Si la identidad es importante, **usar símbolos.**

Ruby usa una tabla de símbolos interna con los nombres de las variables, objetos, métodos, clases… Por ejemplo, si hay un método con el nombre de *control_movie*, automáticamente se crea el símbolo *:control_movie*. Para ver la tabla de símbolos *Symbol.all_symbols*.

COMPONENTE SINTACTICO

El primer método que comienza el programa es **p_programa()** en la cual se invoca una expresión, la cual puede ser todos los tipos de expresiones declaradas en el método **p_expressions()**

```
def p_program(p):
    'program : expression'

def p_expression(p):
    '''expression : string_literals
                  | prints
                  | variable
                  | array
                  | hash
                  | method_invocation
                  | super
                  | assignment
                  | expression_operations
                  | control_structure
                  | class_definitions
                  | singleton_class_definitions
                  | module_definitions
                  | method_definitions
                  | singleton_method_definitions
                  | alias
                  | undef
                  | defined'''
```

la función **p_string_literals()** puede contener una sola palabra o varias como string, sin embargo, tambn se define **p_string_concat()**, en la cual indica la existencia de una cadena de strings con la concatenación de alguna variable.

```
def p_string_literals(p):
    '''string_literals : STRING
                       | string_concat '''

def p_string_concat(p):
    ''' string_concat : DOUBLE_QUOTED IDENTIFIER concat DOUBLE_QUOTED
                      | DOUBLE_QUOTED IDENTIFIER concat IDENTIFIER
DOUBLE_QUOTED '''

def p_concat(p):
    '''concat : NUMBER_SIGN LKEY IDENTIFIER RKEY'''
```

se definen funciones print para las dos palabras reservadas que pueden imprimir cualquier tipo de dato:

```
def p_prints(p):
    '''prints : print
              | puts'''

def p_print(p):
    ''' print : PRINT expression
              | PRINT LPAREN expression RPAREN'''
    global resultado
```

```
    resultado = "expresión print correcta"

def p_puts(p):
    ''' puts : PUTS expression
                | PUTS LPAREN expression RPAREN'''
    global resultado
    resultado = "expresión puts correcta"
```

se define **p_assignment()** en donde se asigna valores a una variable o dato.

```
def p_assignment(p):
    '''assignment : variable EQUAL_SYMBOL data
                  | array_data EQUAL_SYMBOL data
                  | method_invocation EQUAL_SYMBOL data
                  | self_assigment
                  | mult_assigment'''
    global resultado
    resultado = "asignacion de variable correcta"

def p_self_assigment(p):
    '''self_assigment : variable op_assigment data'''

def p_op_assigment(p):
    '''op_assigment : PLUS_EQUAL
                    | MINUS_EQUAL
                    | TIMES_EQUAL
                    | DIVIDE_EQUAL
                    | MOD_EQUAL
                    | POW_EQUAL'''

def p_mult_assigment(p):
    '''mult_assigment : list_var EQUAL_SYMBOL args_method'''

def p_list_var(p):
    '''list_var : variable COMMA
                | variable COMMA list_var
                | variable'''
```

se define **p_control_structure()** que define todas las posibles estructuras de control de nuestro lenguaje propuesto y las respectivas funcionalidades de cada una.

```
def p_control_structure(p):
    '''control_structure : if
                         | if_modifier
                         | unless
                         | unless_modifier
                         | and
                         | or
                         | not
                         | range_expressions
                         | while
                         | while_modifier
                         | until
                         | until_modifier
                         | iterator
                         | for
                         | yield
                         | begin_expression
```

```
                        | retry
                        | return
                        | break
                        | next
                        | redo
                        | BEGIN
                        | END
                        | case'''
                        # TODO
                        # | raise TODO'''


def p_if(p):
    '''if : IF expression expression END
            | IF expression THEN expression END
            | IF expression expression elsif END
            | IF expression THEN expression elsif END
            | IF expression expression else END
            | IF expression THEN expression else END
            | IF expression expression elsif else END
            | IF expression THEN expression elsif else END'''
    global resultado
    resultado = "expresión if correcta"

def p_elsif(p):
    '''elsif : ELSIF expression expression END
            | ELSIF expression THEN expression END'''
def p_else(p):
    '''else : ELSE expression'''

def p_if_modifier(p):
    'if_modifier : expression IF expression'

def p_unless(p):
    '''unless : UNLESS expression expression END
            | UNLESS expression THEN expression END
            | UNLESS expression THEN expression else END'''

def p_unless_modifier(p):
    '''unless_modifier : expression UNLESS expression'''

def p_case(p):
    ''' case : CASE expression END
            | CASE expression when END
            | CASE expression else END
            | CASE expression when else END'''

def p_when(p):
    ''' when : WHEN it_expression THEN expression
            | WHEN it_expression THEN expression when
            | WHEN it_expression expression
            | WHEN it_expression expression when '''

def p_it_expression(p):
    ''' it_expression : expression
                        | expression it_expression'''


def p_and(p):
    'and : expression AND expression'
```

```
def p_or(p):
    'or : expression OR expression'

def p_not(p):
    '''not : NOT expression
           | NOT_SYMBOL expression
           | expression NOTEQUAL expression
           | expression OPPOSITE_MATCHED_STRINGS_OP expression'''

def p_range_expressions(p):
    '''range_expressions : expression RANGE_INCLUSIVE expression
                         | expression RANGE_EXCLUSIVE expression'''

def p_while(p):
    '''while : WHILE expression expression END
             | WHILE expression DO expression END'''
    global resultado
    resultado = "expresión while correcta"

def p_while_modifier(p):
    'while_modifier : expression WHILE expression'

def p_until(p):
    '''until : UNTIL expression DO expression END
             |  UNTIL expression expression END'''

def p_until_modifier(p):
    'until_modifier : expression UNTIL expression'

def p_iterator(p):
    '''iterator : expression DO OR_SYMBOL expression OR_SYMBOL
expression END
                | expression LKEY OR_SYMBOL expression OR_SYMBOL
expression RKEY'''

def p_for(p):
    '''for : FOR IDENTIFIER IN expression DO expression END
        | FOR IDENTIFIER IN expression expression END'''

    global resultado
    resultado = "expresión for correcta"

def p_yield(p):
    '''yield : YIELD LPAREN expression RPAREN
             | YIELD expression'''

def p_begin_expression(p):
    '''begin_expression : BEGIN expression END
                        | BEGIN expression RESCUE expression END
                        | BEGIN expression ENSURE expression END
                        | BEGIN expression RESCUE expression ENSURE
expression END
                        | BEGIN expression RESCUE expression ELSE
expression END
                        | BEGIN expression ELSE expression ENSURE
expression END
                        | BEGIN expression RESCUE expression ELSE
expression ENSURE expression END'''

def p_retry(p):
    'retry : RETRY'
```

```
def p_return(p):
    '''return : RETURN
              | RETURN args_method'''

def p_break(p):
    'break : BREAK'

def p_next(p):
    'next : NEXT'

def p_redo(p):
    'redo : REDO'

def p_begin(p):
    '''begin : BEGIN LKEY expression RKEY'''

def p_end(p):
    '''end : END RKEY expression LKEY'''
```

se definen metodos para las clases normales y clases singleton. Así mismo, las expresiones de modulos, métodos y demás palabras reservadas.

```
def p_class_definitions(p):
    '''class_definitions : CLASS IDENTIFIER expression END
                         | CLASS IDENTIFIER LESSERTHAN SUPER expression
END'''

def p_singleton_class_definitions(p):
    'singleton_class_definitions : CLASS BINARY_LEFT_SHIFT_OP
expression expression END'

def p_module_definitions(p):
    'module_definitions : MODULE IDENTIFIER expression END'

def p_method_definitions(p):
    '''method_definitions : DEF function expression END'''

def p_singleton_method_definitions(p):
    '''singleton_method_definitions : DEF expression DOT IDENTIFIER
expression END
                         | DEF expression DOT IDENTIFIER LPAREN
args_method RPAREN expression END'''

def p_alias(p):
    'alias : ALIAS expression expression'

def p_undef(p):
    'undef : UNDEF expression'

def p_defined(p):
    'defined : DEFINED_OP expression'
```

Por último, se define el método **p_expression_operations()** donde se especifica las diferentes formas de especificar una operación matemática de dos o más elementos.

```
def p_expression_operations(p):
    '''expression_operations : opmate
                             | LPAREN opmate RPAREN
                             | expression_operations op
expression_operations
                             | expression_operations op opmate
                             | LPAREN opmate RPAREN op
expression_operations
                             | expression_operations op LPAREN opmate
RPAREN
                '''
    if (len(p) > 2):
        if (p[1] != "Semantic error in input!" and p[3] != "Semantic
error in input!"):
            if p[2] == '+':
                p[0] = (p[1] + p[3])
            elif p[2] == '-':
                p[0] = (p[1] - p[3])
            elif p[2] == '*':
                p[0] = (p[1] * p[3])
            elif p[2] == '/':
                p[0] = (p[1] / p[3])
            elif p[2] == '%':
                p[0] = (p[1] % p[3])
        else:
            p[0] = "Semantic error in input!"
    else:
        if (p[1] == "Semantic error in input!"):
            p[0] = "Semantic error in input!"
        else:
            p[0] = "sintaxis Valida"
            print(p[0])

def p_opmate(p):
    '''opmate : data op data'''
    if (len(p) > 2):
        if (p[1] != "Semantic error in input!" and p[3] != "Semantic
error in input!"):
            if p[2] == '+':
                p[0] = (p[1] + p[3])
            elif p[2] == '-':
                p[0] = (p[1] - p[3])
            elif p[2] == '*':
                p[0] = (p[1] * p[3])
            elif p[2] == '/':
                p[0] = (p[1] / p[3])
            elif p[2] == '%':
                p[0] = (p[1] % p[3])
        else:
            p[0] = "Semantic error in input!"
    else:
        if (p[1] == "Semantic error in input!"):
            p[0] = "Semantic error in input!"
        else:
            print(p[0])
            p[0] = "sintaxis Valida"

def p_op(p):
    '''op : PLUS
          | MINUS
          | TIMES
```

```
            | DIVIDE
            | MOD
            | EQUAL
            | NOTEQUAL
            | GREATERTHAN
            | GREATERTHANEQUAL
            | LESSERTHAN
            | LESSERTHANEQUAL
        '''
```

Por último, se define **p_data()** donde se indica que puede ser una variable, numero o string.

```
def p_data(p):
    '''data : NUMBER
            | STRING
            | variable'''
```

COMPONENTE SEMANTICO

Para el caso de los booleanos se define un método llamado **boolean_operations()** en el cual se hace la validación respectiva de si los parámetros son booleanos, se procede con la operación lógica que se ingrese por teclado.

```
def p_boolean_operations(p):
    '''boolean_operations : expression AND expression
                          | expression OR expression
                          | expression EQUAL expression
                          | expression NOTEQUAL expression
                          | expression GREATERTHAN expression
                          | expression GREATERTHANEQUAL expression
                          | expression LESSERTHAN expression
                          | expression LESSERTHANEQUAL expression
                          | NUMBER EQUAL NUMBER
                          | NUMBER NOTEQUAL NUMBER
                          | NUMBER GREATERTHAN NUMBER
                          | NUMBER GREATERTHANEQUAL NUMBER
                          | NUMBER LESSERTHAN NUMBER
                          | NUMBER LESSERTHANEQUAL NUMBER
                          | TRUE AND TRUE
                          | TRUE OR TRUE
                          | TRUE AND FALSE
                          | TRUE OR FALSE
                          | FALSE AND FALSE
                          | FALSE OR FALSE
                          | TRUE EQUAL TRUE
                          | TRUE NOTEQUAL TRUE
    '''
    global resultado
    if p[1] == "true" and p[3] == "true":
        p[1]=p[3]=True
    if p[1] == "false" and p[3] == "false":
        p[1]=p[3]=False
    if p[1] == "true" and p[3] == "false":
        p[1]=True
        p[3]=False
    if p[1] == "false" and p[3] == "true":
        p[1]=False
        p[3]=True

    # Semantic (prueba semantica)
    if p[2] == 'and':
        p[0] = p[1] and p[3]
        resultado = "true" if p[0] else "false"
    elif p[2] == 'or':
        p[0] = p[1] or p[3]
        resultado = "true" if p[0] else "false"

    elif p[2] == '==':
        if p[1] == p[3]:
            p[0] = True
        else:
            p[0] = False
        resultado = "true" if p[0] else "false"

    elif p[2] == '!=':
```

```
        if p[1] != p[3]:
            p[0] = True
        else:
            p[0] = False
        resultado = "true" if p[0] else "false"

    elif p[2] == '>':
        if p[1] > p[3]:
            p[0] = True
        else:
            p[0] = False
        resultado = "true" if p[0] else "false"

    elif p[2] == '>=':
        if p[1] >= p[3]:
            p[0] = True
        else:
            p[0] = False
        resultado = "true" if p[0] else "false"

    elif p[2] == '<':
        if p[1] < p[3]:
            p[0] = True
        else:
            p[0] = False
        resultado = "true" if p[0] else "false"

    elif p[2] == '<=':
        if p[1] <= p[3]:
            p[0] = True
        else:
            p[0] = False
        resultado = "true" if p[0] else "false"

    if not isinstance(p[1], bool) and not isinstance(p[2], bool) :
        print("Semantic error in input!")
```

Con respecto a las operaciones matemáticas, en la función definida **opmate()** se realiza las validaciones de los parámetros ingresados si pertenecen a un número para realizar las operaciones indicadas.

```
def p_opmate(p):
    '''opmate : data op data'''

    global resultado

    print(p[1])
    print(p[2])

    if not isinstance(p[1], int) and not isinstance(p[3], int) :
        resultado = "error semantico"
    else:
        print("aquí")
        if p[2] == '+':
            p[0] = (p[1] + p[3])
        elif p[2] == '-':
            p[0] = (p[1] - p[3])
        elif p[2] == '*':
            p[0] = (p[1] * p[3])
        elif p[2] == '/':
```

```
            p[0] = (p[1] / p[3])
        elif p[2] == '%':
            p[0] = (p[1] % p[3])
        resultado = p[0]
```

En las condicionales, como lo es **IF**, se valida que la expresión en la siguiente posición luego de la palabra reservada, sea un resultado booleano. Ya que de eso depende de si se continua con aquella estructura de control o no.

Así mismo, con los ciclos como **WHILE** y **UNTIL**.

```
def p_if(p):
    '''if : IF expression expression END
            | IF expression THEN expression END
            | IF expression expression elsif END
            | IF expression THEN expression elsif END
            | IF expression expression else END
            | IF expression THEN expression else END
            | IF expression expression elsif else END
            | IF expression THEN expression elsif else END'''
    global resultado
    if(p[2]!=True and p[2]!=False):
        resultado = "Condición debe dar un boolean"
    else:
        resultado = "expresión if correcta"

def p_elsif(p):
    '''elsif : ELSIF expression expression END
             | ELSIF expression THEN expression END'''
    global resultado
    if (p[2] != True and p[2] != False):
        resultado = "Condición debe dar un boolean"
    else:
        resultado = "expresión if correcta"
```

```
def p_while(p):
    '''while : WHILE expression expression END
             | WHILE expression DO expression END'''
    global resultado
    if(p[2]!=True and p[2]!=False):
        resultado = "la condición del while debe dar un boolean"
    else:
        resultado = "expresión while correcta"
```

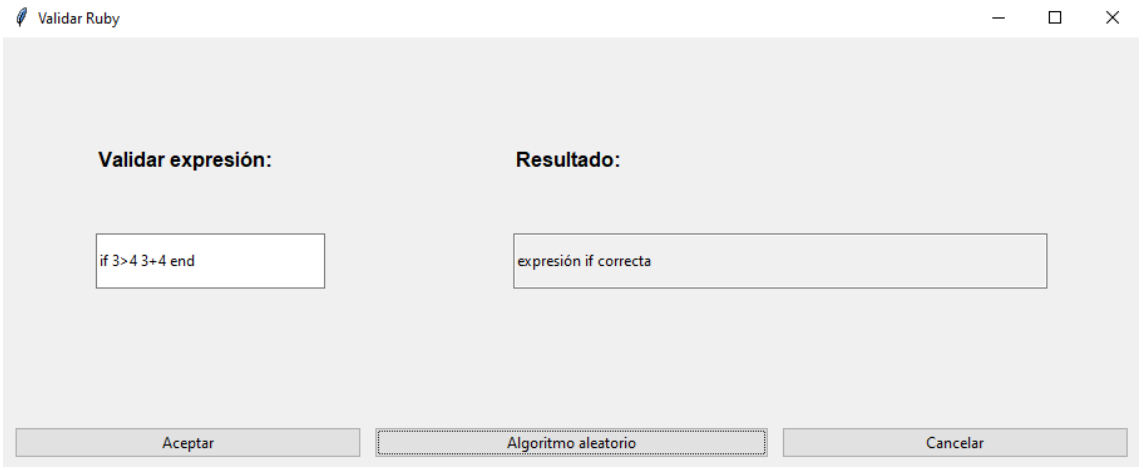
```
def p_until(p):
    '''until : UNTIL expression DO expression END
             |  UNTIL expression expression END'''
    global resultado
    if (p[2] != True and p[2] != False):
        resultado = "la condición del until debe dar un boolean"
    else:
        resultado = "expresión until correcta"
```

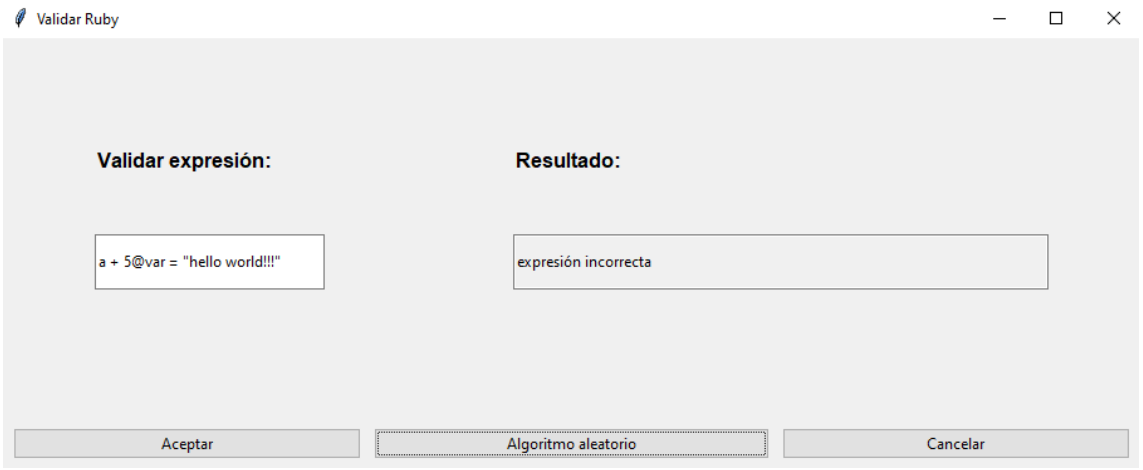
RESULTADOS

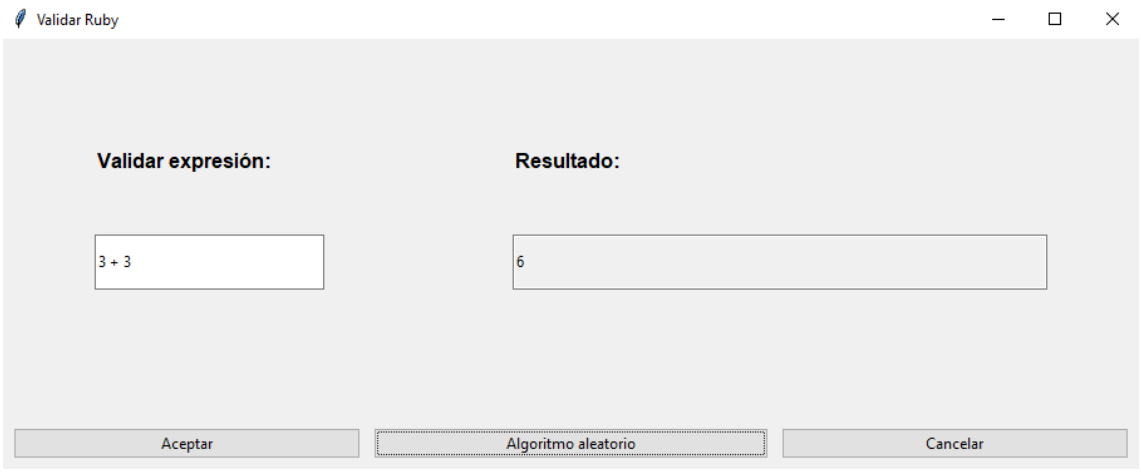
Para probar el funcionamiento del proyecto, se generó un array que contiene diferente código de fuente para probar su funcionamiento y el cual será llamado por un botón de la interfaz.

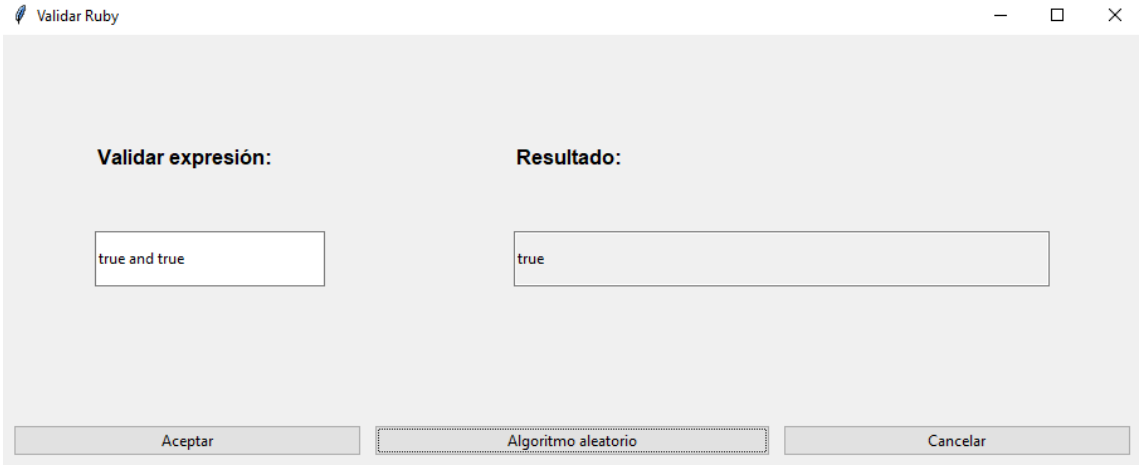
```
array_randoms= [
    '3 + 3',
    'true and true',
    'if 3>4 3+4 end',
    'if a+3 end',
    'true and a',
    'while 3 3+4 end',
    'a + 5'
    '@var = "hello world!!!"']
```

los resultados fueron los siguientes:









Las demás validaciones se encuentran dentro del código.

CONCLUSIONES

Ruby es un lenguaje radicalmente orientado a objetos. Lo que significa que su sintaxis y semántica presentan que en su mayoría todos los tipos son objetos. No tiene mucha variación respecto a otros lenguajes de programación. Sin embargo, tiene muy pocas excepciones respecto a sus palabras reservadas y definición de métodos.

El proyecto nos permitió profundizar en el ensamble del lenguaje propuesto, a conocer mas sobre él de como realmente funciona cada línea, el resultado que necesita cierta posición **p** para funcionar y de la lógica a manejar.

Con la ayuda de Tkinter como interfaz, se creó una pantalla de cómo funciona cada código fuente, y de predecir problemas que puedan solucionarse con sus respectivas validaciones.