Taller 3 - Semántica lenguajes de programación: Interpretador
Fundamentos de Lenguajes Programación

Carlos Andres Delgado S, Msc
carlos.andres.delgado@correounivalle.edu.co

Noviembre de 2021

Importante: El no cumplimiento de alguna de las normas aquí expuestas le traerá reducción en la nota o la anulación de su taller.

1. El taller debe ser entregado antes del día **Domingo, 21 de Noviembre de 2021 a las 23:59:59** hora de Colombia del por el enlace dispuesto en el campus virtual. Se permiten entregas tardías, pero se descuenta 0.15 en la nota por hora o fracción de retraso. Por ejemplo, si entrega a partir delas 12:00:01 am se aplicará una penalización de 0.15, si lo entrega a partir de las 01:00:01 se aplicará 0.3 y así sucesivamente.

2. Entregue un sólo archivo comprimido. No entregue archivos comprimidos dentro de archivos comprimidos.

3. Debe entregar el código fuente organizado en carpetas dentro del primer nivel del archivo comprimido, no cree una jerarquía compleja difícil de revisar.

4. No se permite copiar código de Internet ni de sus compañeros. Si se encuentra código copiado el taller será anulado por completo a todas las partes involucradas.

5. Debe implementar el taller en Dr Racket y recuerde que debe usar lambda para especificar sus funciones.

6. El taller puede ser realizado por grupos de máximo 4 personas previa inscripción al campus virtual. Es requisito inscribirse a los grupos del campus para realizar la entrega.

7. Las primeras líneas de cada archivo deben contener los nombres y códigos de los estudiantes.

```
;Autores: Juanito Perez, 1902321. Pepita Gomez, 1954545, Juanita Delgado, 1914547
```

8. En ese mismo archivo, vendrán comentados los procedimientos que llevan al código de la declaración, las operaciones, la expresión BNF de las estructuras que se están utilizando, y algunos ejemplos de prueba. Por ejemplo, si se pidiera construir el procedimiento remove-first, deberá existir un código como:

```
;; <lista-de-simbolos> := ({<exp-simbolo>}*)
;; <exp-simbolo> := <simbolo> | <lista-de-simbolos>
;;
;; remove-first : simbolo * lista-de-simbolos -> lista-de-simbolos
;;
;; Proposito:
;; Procedimiento que remueve la primera ocurrencia de un simbolo
;; en una lista de simbolos.
;;

(define remove-first
   (lambda (s los)
      (if (null? los)
          '()
          (if (eqv? (car los) s)
              (cdr los)
              (cons (car los)
                    (remove-first s (cdr los)))))))

;; Pruebas
(remove-first 'a '(a b c))
(remove-first 'b '(e f g))
(remove-first 'a4 '(c1 a4 c1 a4))
(remove-first 'x '())
```

9. Para la evaluación de los puntos tenga en cuenta los ejemplos de la última sección.

1. Modificaciones al interpretador de clase [30 puntos]

Estos puntos debe enviarlos en un archivo llamado **interpretadorClase.rkt**. Para construirlos debe basarse en el interpretador: **Interpretador asignación, paso por valor**, disponible en el campus virtual.

1. Agregue las expresiones de listas en el lenguaje. Creando las siguientes sentencias, estas obedecen la siguiente gramática:

```
<expresion> := cons ( <expresion> <expresion> )
               list-exp(valor lista)
            := empty
               list-empty-exp
```

   - **empty** Es la lista vacía
   - **cons(n,l)** Crea una lista, donde n es una expresión y l es una lista
   - **length(l)** Retorna el tamaño de la lista l
   - **first(l)** Retorna el primer elemento de la lista l
   - **rest(l)** Retorna el resto de la lista l
   - **nth(l,n)** Retorna el elemento n de la lista l

   Ejemplo se ingresa en el interpretador:

```
cons(1 cons(2 cons(3 empty)))
```

Esto retorna

```
(1 2 3)
```

Si ingresa:

```
empty
```

Esto retorna

```
()
```

Si ingresa:

```
cons(cons(1 empty) cons(2 empty))
```

Esto retorna

```
((1) 2)
```

Si ingresa:

```
cons(
    let x = 3
    in
    x
    cons(3 empty)
)
```

Esto retorna

```
(3 3)
```

Si ingresa:

```
cons(1 2)
```

```
"Error: 2 no es una lista"
```

Pista: Implemente los casos en las expresiones de la lista vacía y lista no vacía, donde se deban verificar

```
(cases expression exp
    (list-exp ... )
    (list-empty-exp  ...)
    (else eopl:error ...)
)
```

2. Implementar la sentencia **cond** que tiene la siguiente estructura:

```
expresion ::= cond {expresion ==> expresion}* else ==> expresion end
```

Suponiendo que X es igual a 2, el siguiente ejemplo. Recuerde que en nuestro lenguaje, se asume como verdadero cualquier valor diferente que 0.

```
cond
<=(x,1) ==> 1
<=(x,2) ==> 2
<=(x,3) ==> 4
else ==> 9
end
```

Retorna 2

```

cond
else ==> 9
end
```

Retorna 9

2. ¡Un pequeño Python! [70 puntos]

En este punto vamos a empezar a implementar nuestra versión de Python, este debe entregarlo en un archivo llamado **interpretadorPython.rkt**

Para empezar, podemos utilizar la especificación léxica del interpretador visto en clase y modificar los comentarios. Recuerde que los comentarios en Python inician con #.

Debido a las limitaciones de la librería EOPL. no forzaremos la indentación, en su lugar finalizaremos las sentencias con **end**

1. [**5 puntos**] Agregar a la especificación léxica los números flotantes y los booleanos, los cuales tienen la siguiente gramática:

```
<flotante> := <digito>* <digito> . <digito> <digito>*
```

   Ejemplo:

```
1.0
10.56
13.5
```

   Agregar también los booleanos **True** y **False**

```
<booleano> := True | False
```

   Ejemplo:

```
True
False
```

2. Agregar las primitivas numéricas suma, resta, multiplicación y módulo, las cuales funcionan en notación inflija. Debe funcionar (valor y respuesta respectivamente). La gramática asociada es:

```
<expresion> := evaluate expresion primitiva expresion
               evaluate-exp(exp1 exp2)
<primivita> := "+"
               suma-prim
               := "-"
               resta-prim
               := "*"
               mult-prim
               := "/"
               div-prim
               := "mod"
               mod-prim
```

   (.[ev]aluate.[ex]pression primitive expression)

```
evaluate 5+7
12
evaluate 5 + evaluate 7+1
13
evaluate evaluate 2*3+1
7
evaluate evaluate 2%2*7
0
evaluate 1 + evaluate 2 + evaluate 3 + evaluate 4+5
15
```

Recuerde que la operación modulo tiene prioridad sobre el resto de operaciones. La operación multiplicación y división tiene prioridad sobre la resta

3. Agregar las primitivas relacionales <, >, <=, >=, == y ! = estas deben retornar **True** o **False** según el caso.

```
evaluate 5 < 3
False
evaluate 5 > 3
True
evaluate 5 == 5
True
evaluate 5 != 3
True
```

Así mismo implementar los operadores lógicos **and**, **or** y **not**

```
evaluate True and False
evaluate False or evaluate True or False
```

Retorna:

```
False
True
```

4. Generar la primitiva **print** de Python, la cual tiene la siguiente estructura:

```
print (expresion)
```

5. **[10 puntos]** Agrega los condicionales que tienen la siguiente estructura:

```
if expresion: expresion+
<elif expresion: expresion* end>*
else: expresion*
end
```

Por ejemplo:

```
if evaluate 5 < 3:
    evaluate 4 + 3
    evaluate 4 < 5
else:
    evaluate 8 + 9
    evaluate 7 < 3
end
```

Retorna

```
False
```

```
if evaluate 5 < 3:
    evaluate 4 + 3
    evaluate 4 < 5
elif True:
    evaluate 4 + 9
    evaluate 6 + 8
end
else:
    evaluate 8 + 9
    evaluate 7 < 3
end
```

Retorna

```
14
```

6. Definiciones y ambientes en Python. Diseñe una representación de ambientes para Python (a su gusto) y permita la definición de variables. Python maneja alcance dinámico de variables, sin embargo, está por fuera del alcance del curso. Por lo tanto, usaremos **alcance estático**. En nuestro Python **no manejaremos definiciones locales**

```
var x = 9
print(x)
var x = 4
print(x)
```

Retorna

```
9
4
```

La definición de variables es una operación de efecto, por lo que no tiene retorno.

7. Diseñe las funciones en Python y sus respectivos llamados. Recuerde que estas pueden ser llamadas recursivamente.

```
expresion := def identificador(identificador* ','): expresion* return expresion end
expresion := execute identificador(expresion*',')
```

Ejemplo

```
def funcion(a):
    if evaluate a == 0:
        return 1
    else
        return 1 + execute funcion(a - 1)
    end
end
```

Si se llama:

```
print(execute funcion(2))
```

Retorna

```
3
```

2.1. Ejemplos

Las siguientes expresiones deben funcionar en su lenguaje:

2.1.1. Ejemplo 1

```
var a = 3
var b = 2
var c = 3
def funcion(x,y,z):
    if evaluate x < 3 :
```

```
        var t = 4
        evaluate evaluate t * x + evaluate y - z
    else:
        var t = 2
        evaluate evaluate t * y - z
    end
end
print(execute funcion(a,b,c))
```

Retorna:

```
1
```

2.1.2. Ejemplo 2

```
var a = 4
print (a)
def funcion(a):
    var a = evaluate 7 + a
    return evaluate a + 2
end
print(execute funcion(a))
print(a)
```

Retorna:

```
13
4
```

Pista: No se preocupe si retorna más cosas (producto de las otras expresiones), es importante que el print sea la clave del retorno de un valor.