Taller 2 FLP

- Probado y desarrollado en la versión 6.7 de racket
- Este archivo contiene las pruebas de el taller

El presente proyecto intenta implementar un lenguaje de programación lo mas parecido a phython que sea posible, pero por supuesto por las limitaciones de las herramientas que se estan usando para la realización de este no es posible cumplir con todos los requerimientos, ademas se han modificado algunos requerimientos a los expresados en el taller (para bien por supuesto) los cuales son importantes de describir en el presente documento.

1.1 Se han adicionado los flotantes tal y como indica el punto, adicionalmente se han añadido los flotantes negativos, pero para poder representarlos el simbolo "-" debe estar seguido de el número, si no el parse devolvera un error.
Ejemplo:

evaluate x - - 1.5 Forma incorrecta
evluate x - -1.5 Forma correcta

1.2 Se han adicionado también los booleanos, externamente funcionan de la manera que se especifica en el taller, internamente se manejan como caracteres ('True 'False).

2 - 3. El punto se ha desarrollado como se ha especificado en el taller para las primitivas de multiplicacion, suma, resta, división y modulo, al igual que para las comaparaciones entre números, (<, >, <=, >=, == !=), y para el and y or, pero el not tiene la siguiente representación elegida a conveniencia:

not bool-exp

Ejemplos:

not True ;False
not evaluate 3 < 5 ;False
not evaluate 3 < evaluate 5 % 20 ;False

4. El print esta correctamente implementado, bajo la salvedad de que cuando se imprime una expresión directamente el el interpetador, imprimira una linea adicional con la salida "#void" esto se supone que es por el interpretador que usamos, el cual siempre imprime lo que devuelve la función, y como print no devuelve nada mas que #void esto se imprime al final, adicional a los display que hace.

5. El if se ha implementado con la sintaxis que se ha pedido, con la salvedad de que el return literalmente terminara con la ejecución de el programa, y este puede definirse donde se quiera, dentro de un elif, en el else, en el if, funciona en todas partes.

El if-else funciona como se ha pedido.

6. En el punto de ambientes se ha roto la regla, la definicion de variables no es solo global, si no que al contrario de esto se pueden definir variables al interior de los procedimientos o estructuras de control y estas se eliminan al salir de esta.

Un ejemplo de esto:

Suponiendo que se tiene un ambiente inicial con una variable x que toma el valor 10, y el siguiente prcedimiento:

```
def cuadrado(y):
        var x = evaluate y * y
        return x
end
```

Al evaluar execute cuadrado(20) este retornara 400 y la variable x definida en el ambiente inicial (ambiente global) seguira siendo 10.

Si por ejemplo se tiene otra finción con el mismo ambiente inicial, una unica variable x = 10

```
def potencia3(y):
        var x = evaluate y * y
        var z = evaluate x * y
        return z
end
```

El siguietne ejemplo retornara la variable e, pero afuera de la función este no es visible:

```
def ejemplo():
        if True:
                var e = 10
        else:
                var r = 50
        end
return e
end
```

Al ejecutar potencia3 con el valor de 5 se retornara 27, y al llamar a la variable z, esta debera arrojar un error por que su definición fue temporal dentro de la función.

Todas las variables que se definan por linea de comandos seran adicionadas al ambiente inicial (ambiente global) y seran visibles al interior de los procedimientos.

Si por ejemplo ejecutamos "var n = 10" en el interprete de comandos y luego tenemos la funcion

```
def nCuadrado():
        var m = execute n * n
        return m
end
```

La función retornara 100 y m no sera visible fuera de la función.

Lo anterior se ha logrado gracias a la definición de ambientes utilizada, existe una estructura llamada macroambiente la cual tiene la siguiente definición gramatical:

```
environment  := empty-env
                        := extended-env
```

Donde <enpty-env> no tiene argumentos y <extended-env> esta compuesto por por un gvector de simbolos, un gvector de valores y un entero que representa la posición dentro de el macroambiente (el cual resulta ser un gvector de ambientes) en al que se encuentra el ambiente de el que extiende.

macroambiente := pos-current-env empty-env extended-env extended-env*

La primera posición indica en sobre que ambiente se esta trabajando en un momento dado de la ejecución de el programa, el segundo es un ambiente vacio el cual funciona como caso base para varias de las funciones que estan definidas sobre los ambientes y mantienen las propiedades de la recursividad sirviendo como caso base, un ambiente obligatorio que es el ambiente "global", sobre el cual las ejecuciones en la linea de comandos actua, esto es si se hace un var y = 25 en la linea de comandos esta y estar en ese primer y obligatorio ambiente extendido, el resto de ambientes se va creando y eliminando conforme la ejecución del programa se va dando, creando o eliminando en llamados recursivos y en la finalización de la ejecución de procedimientos, <<Actualmente no es posible el paso por referencias ni modificar variables globales desde dentro de los procedimientos, aunque si se pueden ver desde dentro de estos>>

Varias funciones trabajan sobre el macroambiente, por ejemplo get-current-env el cual retorna <pos-curren-env>, get-env-n la cual retorna el ambiente n definido dentro de el

macroambiente, get-extend-from el cual con un ambiente extendido como entrada retorna el número que identifica al ambiente del que extiende, etc etc.

Es importante decir que las funciones se toman como ciudadanos de primera clase como se hace en scheme, estas no son mas que un id con un closure asociado, estas funciones se alamacenan igual que las variables.

Tanto las variables como los metodos pueden ser seteados con la expresion var, esto es, si existe una variable x definida como 10, y luego se ejecuta un var x = 11 en un esenario similar al que se declaro x, esta se setea, ademas se puede definir un procedimiento con el nombre de x caso en el cual simplemente x pasaria a tomar el valor del closure representado por la función definida.

Un ejemplo:

```
var x = 10
print (x) := 10
var x = 11
print (x) := 11

def x(x):
var x = evaluate  x * x
return x
end

print(x) := #(struct:closure (x) (#(struct:set-exp x #(struct:evaluate #(struct:var-exp x) #(struct:mult-prim) #(struct:var-exp x))) #(struct:return-exp #(struct:var-exp x))) 2)

print(execute x(10)) := 100
```

Notese que no existe interferencia entre las variables y se ejecuta la sentencia correctamente,

7. Con respecto a este punto, ha sido posible implementar el retorno de valores dentro de el cuerpo de la función en cualquier parte, ya sea al final o no, esto se consigue alterando la estructura de la gramatica para la función, quedando entonces de la forma:

expression := def identificador (identificador* ','): expression

Es posible con esto entonces definir funciones que en realidad no retornen nada (void).

Para las funciones que si retornan algo se definio entonces una expresion retorno:

expression:= return expression

En la evaluación tanto de la estructura de control if como dentro de los elif o de los else puede estar una expresion de retorno, terminando con la ejecucion de esta, igual pasa con las funciones declaradas.

Es importante mencionar que el retorno no funciona para llamados a otras funciones, por lo tanto tampoco para la recursividad, si desea hacer recursividad puede usar el return cuando valla a retornar expresiones atomicas, como puede ser un execute, un numero o un booleano, si se hace return llamando a una expresión no funcionaria, un ejemplo:

```
if False:
        print (0)
elif True:
        print (1)
        return evaluate 3 % 5
        print (2)
end
else:
        print (17)
end
```

Lo anterior da como resultado la impresión por pantalla de un 1 y la impresion del retorno que es 3, y termina la ejecución, nunca se imprime el 2, este no tiene problemas por que la expresión del return no hace llamado a ninguna función.

Para el caso de la función fibonacci, no se puede usar retorno en el llamado recursivo como se deberia hacer, pero funciona haciendo el llamado recursivo con un simple evaluate y asignandoselo a una variable la cual luego sera retornada (si se ejecute execute fib(8) se retorna 21, etc):

Forma correcta:

```
def fib(n):
        if evaluate n <= 1:
                return 1
        elif evaluate n == 2:
                return 1
        end
        else:
                var ret =  evaluate execute fib(evaluate n - 1) + execute fib (evaluate n - 2)
        end
return ret
end
```

Es de notar que se puede returnar la variable de retorno dentro del else o afuera, funciona de cualquier forma:

```
def fib(n):
        if evaluate n <= 1:
                return 1
        elif evaluate n == 2:
                return 1
        end
        else:
                var ret =  evaluate execute fib(evaluate n - 1) + execute fib (evaluate n - 2)
                return ret
        end

end
```

Forma incorercta:

```
def fib(n):
        if evaluate n <= 1:
                return 1
        elif evaluate n == 2:
                return 1
        end
        else:
                 return evaluate execute fib(evaluate n - 1) + execute fib (evaluate n - 2)
        end
end
```

De manera analoga sucede en el caso de la siguietne funcion, el siguiente ejemplo imprime por pantalla el 0, el 1 y el retorno 2 al ejecutarse (con execute a()).

```
def a():
        print(0)
        print(1)
        return 2
        print(3)
        print(4)
end
```

Como ya se dijo las funciones pueden ser recursivas, algunos ejemplos de funciones provadas (funciones recursivas y no recursivas):

```
def exp(numero, exponente):
        if evaluate exponente == 0:
        return 1
        elif evaluate exponente == 1:
        return numero
        end
        else:
        var ret = evaluate numero * execute exp(numero , evaluate exponente - 1)
        end
return ret
end
```

```
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))

var a = 4
print (a)
```

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

print (execute funcion(a))

print (a)
```

Factorial:

```
def fac(n):
        if evaluate n == 1:
        return 1
        else:
        var ret = evaluate n * execute fac(evaluate n - 1)
        return ret
        end
end
```

```
def a(x):
        if evaluate x <= 5:
                print(1)
        elif evaluate evaluate x > 5 and evaluate x <=10:
                print(1)
                print(2)
        end
        elif evaluate evaluate x > 10 and evaluate x <=20:
                print(1)
                print(2)
                print(3)
        end
        else:
                print(1)
                print(2)
                print(3)
                print(4)
        end
end
```

Ejemplos finales:

```
def r():
        if False:
                print(1)
```

```
        elif  True:
                if False:
                        print (2)
                elif False:
                        print (3)
                end
                elif True:
                        return 4
                        print (6)

                end
                else:
                        print(7)
                end
        end
        else:
                print (8)
        end
return 666
end
```

Sobreescritura de variables:

```
var x = 1000
def  factAnidado(x):
        def fac(x):
                if evaluate x == 1:
                        return 1
                else:
                        var ret = evaluate x * execute fac(evaluate x - 1)
                end
        return ret
        end
var ret = execute fac(x)
return ret
end

print(execute facAnidado(5))
```
