PROYECTO DE LENGUAJES DE PROGRAMACIÓN

Tema

Intérprete de instrucciones básicas de Clojure con PLY

Integrantes

Steven Encalada
Andrea Mero
Milca Valdez

Primer Parcial

2022 - II PAO

Tabla de Contenido

Introducción

Clojure es un lenguaje de programación funcional que se caracteriza por combinar la accesibilidad y desarrollo interactivo de un lenguaje de scripting con una infraestructura eficiente y robusta para la programación multiproceso. Además de poseer una estructura de datos inmutable y persistente.

Al ser un dialecto de Lisp, está todo escrito en forma de lista, por ello la abundancia de los paréntesis, que denotan la lista "( )", además cuenta con símbolos. Este lenguaje utiliza la notación de prefijo en su totalidad. Dentro de las palabras reservadas podemos encontrar "def", "println", "count", "get", "map" entre otras. Por ejemplo para "(+ 34)" su sintaxis sería "( )" que es una lista, " + " es el símbolo y "34" son los números. En Clojure se definen los siguientes tipos de datos: integer, floating point, char, boolean, String, Nil y Atom [3]. Además podemos contar las siguientes estructuras de control: "if", "if and do", "when", "cond", "cond and else", "case", "case with else-expression", "dotimes", "doseq", "doseq with multiple bindings", "loop and recur", "defn and recur" and "recur for recursion"[4]. Las funciones dentro de Clojure se definen mediante el macro "defn", pero también existen las funciones anónimas que no tienen asociadas un nombre.

PLY es una implementación de herramientas de análisis LEX (Generador de Tokens) y YACC (Parseador de Gramática) para Python, con el objetivo de construir compiladores [9]. Su principal tarea como analizador léxico consiste en leer caracteres de entrada, agruparlos en lexemas, que es una secuencia de de caracteres del programa fuente que coincide con un patrón especificado previamente, y generar una secuencia de tokens identificados. Por otro lado, como analizador sintáctico, obtiene un secuencia de tokens y verifica que la cadena de caracteres posea el orden que caracteriza al lenguaje de programación en cuestión [8].

El objetivo de este proyecto es desarrollar un Analizador Léxico, Sintáctico y Semántico del Lenguaje de Programación Clojure empleando la herramienta PLY.

Componentes Léxicos

Identificadores

Los nombres de las variables deben empezar con una letra, ya sea mayúscula o minúscula. Además, solo pueden contener letras, número y guión bajo.
Ejemplos:

- Lista_2
- var24
- VAR_2
- LISTA_24
- NUMEROS_PARES

Tipos de Datos Primitivos

Datos Primitivos	Descripción	Ejemplo
Integers	La función devuelve la suma de números. Puede especificar cualquier número de números como argumentos:	(+ 1 2) ;;=> 3 (+ 1 2 3) ;;=> 6 (+ 1.2 3.5) ;;=> 4.7
Floating point	Un número de punto flotante es un número con cero o más dígitos detrás del separador decimal.	12.34 -2.4 0.1 3.14 .16.984025 1024.0
Char	Esto define un literal de un solo carácter. Los caracteres se definen con el símbolo de contragolpe.	\c => "c" \A => "A"
Boolean	Esto representa un valor booleano, que puede ser verdadero o falso.	(> 4 2) ;;=> true (= 5 2) ;;=> false
String	Estos son literales de texto que se representan en forma de cadena de caracteres.	“Hello World”

Nil	Esto se usa para representar un valor NULL en Clojure.	user=> (if (> 0 2) 3) ;;=> nil

Palabras Reservadas

Como en la mayoría de lenguajes de programación, Clojure también tiene términos o palabras que son propias del lenguaje y realizan una función en particular dentro del mismo. Es indispensable no utilizar estas palabras como nombres de variables o de funciones, dado que el lenguaje podría considerarlo como una de sus palabras claves.

let	future	defn	def
fn	ns	class	if
do		println	new

Símbolos y Caracteres Especiales

Caracteres especiales	**Descripción**
,	Se lee como un espacio en blanco. A menudo se usa entre pares de clave/valor de mapa para mejorar la legibilidad.
'	quote: 'form → (quote form)
/	Separador de espacio de nombres
\	Carácter literal
:	Palabra clave
;	Comentario de una sola línea
^	Metadata
`	Cita de sintaxis
(	Lista literal
[	Vector literal

{	Map literal

Reglas Sintácticas

Operadores de Asignación

Operador	Descripción	Ejemplo
def	Se puede definir una variable (“var”) mediante def	def x 1 x ;=> 1

Operadores de Comparaciones

Operadores de Comparaciones	Descripción	Ejemplo
=	Igualdad. Devuelve verdadero si x es igual a y, falso en caso contrario.	(= 2 3) ;;=> false (= 2 3 4) ;;=> false
not=	Same as (not (= obj1 obj2)). Permite negar una expresión booleana.	(not (not false)) ;;=> false (not (not (> 4 2))) ;;=> true
>	Devuelve un valor no nulo si los números están en orden monótonamente decreciente; de lo contrario, es falso.	(> 2 4) ;;=> false (> 5 3) ;;=> true
<	Devuelve un valor no nulo si los números están en orden monótonamente creciente; de lo contrario, es falso.	(< 4 8) ;;=> true (< 9 4) ;;=> false
<=	Devuelve un valor no nulo si los números están en orden monótono no decreciente; de lo contrario, es falso.	(<= 4 4) ;;=> true (<= 7 3) ;;=> false
>=	Devuelve un valor no nulo si los números están en un orden monótonamente no	(>= 5 5) ;;=> true

	creciente; de lo contrario, es falso.	(>= 7 9) ;;=> false

Estructuras de Control

Número	Estructura de Control	Descripción	Ejemplo
1	if	Es la expresión condicional más importante: consta de una condición, un "then" y un "else"; "if" sólo evaluará la rama seleccionada por el condicional. La estructura se engloba entre paréntesis, y se coloca el "if". Después del "if" siguen dos paréntesis donde va la comparación, y fuera de este es el bloque del "then" o "else".	user=> (str "2 is " (if (even? 2) "even" "odd")) 2 is even
2	if and do	El "if" solo toma una sola expresión para "then" y "else". Usar "do" para crear bloques más grandes que son una sola expresión. La única razón para hacer el "if and do" es si los bloques tienen efectos secundarios. La estructura se engloba entre paréntesis, y se coloca el "if" dentro. Después se debe englobar en paréntesis la condición y luego le siguen entre paréntesis el bloque del "do " que contiene a su vez entre paréntesis la instrucción y después el valor booleano a comparar con respecto a condición.	(if (even? 5) (do (println "even") true) (do (println "odd") false))
3	when	Es un "if" con solo una rama entonces. Comprueba una condición y luego evalúa cualquier cantidad de declaraciones como un cuerpo (por lo que no se requiere hacer). Se devuelve el valor de la última expresión. Si la condición es falsa, se devuelve "nil". Cuando le comunica a un lector que no hay una rama "else". La estructura se engloba entre paréntesis, y se coloca "when" dentro, luego la condición entre paréntesis y después de estos la instrucción a ejecutar.	(when (neg? x) (throw (RuntimeException. (str "x must be positive: " x))))
4	cond	Es una serie de pruebas y expresiones. Cada prueba se evalúa en orden y la expresión se evalúa y	(let [x 5] (cond (< x 2) "x is less than 2"

		devuelve para la primera prueba verdadera. La estructura se engloba entre paréntesis, y se coloca "cond" dentro, luego la condiciones englobadas en paréntesis y cada una seguida de la instrucción a ejecutar.	(< x 10) "x is less than 10"))
5	cond and else	Si no se satisface ninguna prueba, se devuelve cero. Un modismo común es usar una prueba final de ":else". Las palabras clave (como :else) siempre se evalúan como verdaderas, por lo que siempre se seleccionará como predeterminada. La estructura se engloba entre paréntesis, y se coloca "cond" dentro, luego la condiciones englobadas en paréntesis, cada una seguida de la instrucción a ejecutar y al final está ":else" seguida de la instrucción.	(let [x 11] (cond (< x 2) "x is less than 2" (< x 10) "x is less than 10" :else "x is greater than or equal to 10"))
6	case	Compara un argumento con una serie de valores para encontrar una coincidencia. ¡Esto se hace en tiempo constante (no lineal)! Sin embargo, cada valor debe ser literal en tiempo de compilación (números, cadenas, palabras clave, etc.). A diferencia de "cond", "case" generará una excepción si ningún valor coincide. La estructura se engloba entre paréntesis, y se coloca "case" dentro, luego el valor del caso y la instrucción a ejecutar.	user=> (defn foo [x] (case x 5 "x is 5" 10 "x is 10")) #'user/foo user=> (foo 10) x is 10 user=> (foo 11) IllegalArgumentException No matching clause: 11
7	case with else-expression	Este "case" puede tener una expresión final final que se evaluará si ninguna prueba coincide. La estructura se engloba entre paréntesis, y se coloca "case" dentro, luego el valor del caso y la instrucción a ejecutar y al final está ":else" seguida de la instrucción.	user=> (defn foo [x] (case x 5 "x is 5" 10 "x is 10" :else "x isn't 5 or 10")) #'user/foo user=> (foo 11) x isn't 5 or 10
8	dotimes	Evalúa la expresión "n" veces,retorna "nil" al final. La estructura se engloba entre paréntesis, después de "dotimes" se debe de poner el rango entre corchetes y al final la instrucción a ejecutar.	user=> (dotimes [i 3] (println i)) 0 1 2 nil

9	doseq	Itera sobre una secuencia, si es una secuencia "perezosa", fuerza la evaluación y retorna "nil" al final. Después de la palabra "doseq" se debe de poner el rango entre corchetes, dentro de estos se debe poner el rango entre paréntesis y al final la instrucción a ejecutar.	user=> (doseq [n (range 3)] (println n)) 0 1 2 nil
10	doseq with multiple bindings	Similar a los foreach anidados, procesa todas las permutaciones del contenido de la secuencia y retorna "nil" al final. La estructura se engloba entre paréntesis, y se coloca "doseq" dentro, a continuación el rango entre corchetes. Este rango puede tener algunas condiciones que van una detrás de la otra. Después la estructura de datos a iterar y al final la instrucción a ejecutar.	user=> (doseq [letter [:a :b] number (range 3)] ; list of 0, 1, 2 (prn [letter number])) [:a 0] [:a 1] [:a 2] [:b 0] [:b 1] [:b 2] nil
11	loop and recur	Construcción de bucle funcional, "loop" define enlaces y "recur" vuelve a ejecutar el bucle con nuevos enlaces. Preferiblemente usar funciones de biblioteca de orden superior en su lugar. La estructura se engloba entre paréntesis, después "loop" seguido del rango entre corchetes. A continuación, pueden ir más estructuras de control y al final entre paréntesis "recur" con la instrucción.	(loop [i 0] (if (< i 10) (recur (inc i)) i))
12	defn and recur	Los argumentos de función son enlaces de bucle implícitos. La estructura se engloba entre paréntesis, adentro se debe colocar "defn" pueden ir más estructuras de control o no y al final entre paréntesis "recur" con la instrucción.	(defn increase [i] (if (< i 10) (recur (inc i)) i))

Estructuras de Datos

Clojure posee las siguientes estructuras de datos:

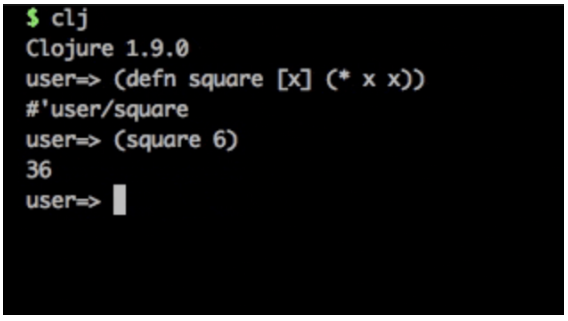
1	Listas	Las listas se usan más comúnmente en Clojure para representar código.	(list 1 2 3) ;=> (1 2 3) (list 1 (+ 1 1) 3)

		Por lo general, si desea una Lista, construye una con la list función.	;=> (1 2 3) '(1 2 3) ;=> (1 2 3) ;; ok! '(1 (+ 1 1) 3) ;=> (1 (+ 1 1) 3)
2	Conjuntos	Los conjuntos tienen una sintaxis literal.	#{1 2 3} ;=> #{1 3 2}
3	Mapas	Los mapas son una de las colecciones más versátiles y es muy fácil integrarlas en nuestras funciones	{:a 1 :b 2 :c 3}
4	Vector	El vector es una colección y la declaramos de la siguiente manera.	[1 2 3 4]

Métodos de Impresión

En la programación funcional, se dice que un código que no tiene impacto en el mundo exterior, no tiene efectos secundarios (no side effects) . La distinción entre efectos secundarios y valor de retorno es una idea clave de la programación funcional. A continuación se presentan los métodos de impresión.

- *Con REPL* (standing for *Read-Eval-Print Loop*): se imprime el resultado de la expresión que se ejecute. No tiene side effects.



- Sin REPL: println, permite imprimir por medio de la consola. Tiene side effects.

```
user=> (println "Hello")
```

Funciones

Las funciones están compuestas por 5 partes principales:

1. La palabra reservada **defn**
2. El nombre de la función
3. Una pequeña descripción sobre el objetivo la función
4. Una lista de parámetros entre corchetes
5. El cuerpo de la función.

```
(defn nombreFuncion
        "Docstring que describe la función (opcional)"
        [ parametro1, parametro2, …, parametroN ]
        ( str "Hola Clojure" )
)
```

La forma de llamar una función, es la siguiente:

```
(nombreFuncion Parametro1 Parametro2 … ParametroN)
```

Ejemplo

```
(defn saludar
        "Retorna un saludo predeterminado con el nombre de la persona
ingresada como parámetro"
        [ name ]
        ( str "OH! Eres " name "?! Que emoción verte nuevamente! :D" )
)

(saludar "Zelda" )
;=> "OH! Eres Zelda?! Que emoción verte nuevamente! :D"
```

Input por teclado

Para poder leer una línea de caracteres ingresado por teclado, se utiliza *read-line*

```
user=> (read-line)
line to be read      ;Type text into console
"line to be read"
```

Bibliografía

[1] S. Tayon, “Cheatsheet,” *Clojure*. [Online]. Available: https://clojure.org/api/cheatsheet. [Accessed: 04-Nov-2022].

[2] “Learn clojure - syntax,” *Clojure*. [Online]. Available: https://www.clojure.org/guides/learn/syntax. [Accessed: 05-Nov-2022].

[3] “Clojure - data types,” *Tutorials Point*. [Online]. Available: https://www.tutorialspoint.com/clojure/clojure_data_types.htm. [Accessed: 05-Nov-2022].

[4] “Learn clojure - flow control,” *Clojure*. [Online]. Available: https://www.clojure.org/guides/learn/flow. [Accessed: 05-Nov-2022].

[5] “Clojure - defining a function,” *Tutorials Point*. [Online]. Available: https://www.tutorialspoint.com/clojure/clojure_defining_a_function.htm. [Accessed: 05-Nov-2022].

[6] “Clojure - anonymous functions,” *Tutorials Point*. [Online]. Available: https://www.tutorialspoint.com/clojure/clojure_anonymous_functions.htm. [Accessed: 05-Nov-2022].

[7] Practicalli, “Syntax,” *Syntax · Practicalli Clojure*. [Online]. Available: https://practical.li/clojure/reference/clojure-syntax/syntax.html. [Accessed: 05-Nov-2022].

[8] “Mi primer Proyecto Utilizando Ply Con Python 3,” *Erick Navarro*, 03-Jul-2021. [Online]. Available: https://ericknavarro.io/2020/02/10/24-Mi-primer-proyecto-utilizando-PLY/. [Accessed: 05-Nov-2022].

[9] *Ply (python lex-yacc)*. [Online]. Available: https://www.dabeaz.com/ply/. [Accessed: 05-Nov-2022].

[10] X. Lee, *Clojure tutorial*. [Online]. Available: http://xahlee.info/clojure/clojure_index.html.

[11] “Get Programming with clojure MEAP V02,” *Get Programming with Clojure MEAP V02*. [Online]. Available: https://livebook.manning.com/book/get-programming-with-clojure/welcome/v-2/.