Trabajo Práctico sobre Clojure

Evaluación de un Lenguaje de Programación

Cátedra: Lenguajes de Programación

Jueves 30 de Septiembre de 2010

Fontán, Emmanuel

INTRODUCCIÓN

El presente trabajo trata de hacer un abordaje al lenguaje de programación **Clojure**, analizando las características funcionales y sintácticas tratadas en clase y encontradas en el lenguaje, y la incidencia de cada una de estas sobre los criterios de Legibilidad, Facilidad de Escritura, Confiabilidad y Costos.

Antes, para comprender mejor los aspectos de este lenguaje y lo que conlleva abordar un paradigma nuevo desde cero, se considera apropiado hacer una introducción a la **Programación Funcional**, describiendo sus principales características, incluyendo *transparencia referencial*, *funciones puras* y *carencia de efectos secundarios*, entre otras que consideramos importantes.

Luego, se hará una breve introducción al lenguaje, analizando muy brevemente su origen, su sintaxis, y explicando el ciclo de vida de un programa en Clojure a partir de que se traduce el programa fuente hasta que se genera el código intermedio interpretable por la máquina virtual.

Finalmente, se analizará en detalle cada una de las características y criterios mencionados, en los cuales se han incluido únicamente los más pertinentes al lenguaje, agregando el de *Concurrencia* por ser este uno de los objetivos del lenguaje.

PROGRAMACIÓN FUNCIONAL

La **programación funcional** es un *paradigma de programación* declarativa basado casi enteramente en la utilización de **funciones** para la manipulación de listas y conjuntos de datos.

Los programas escritos en un lenguaje funcional *puro* están constituidos únicamente por **definiciones de funciones**, entendiendo éstas no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas, en las que se verifican ciertas propiedades como la **transparencia referencial** y, por tanto, la **carencia** total de "**efectos secundarios**" o "**efectos colaterales**".

En los **lenguajes funcionales puros** un programa consiste exclusivamente en la aplicación de distintas funciones a un valor de entrada para obtener un valor de salida.

Otras características propias de estos lenguajes son la no existencia de asignaciones de variables y la falta de construcciones estructuradas como la secuencia o la iteración (lo que obliga en la práctica a que todos los *bucles* de instrucciones se lleven a cabo por medio de **funciones recursivas**).

A pesar de lo dicho antes, cabe aclarar que existen dos grandes categorías de lenguajes funcionales: los funcionales **puros** (Haskell y Miranda) y los **híbridos** (Clojure, Scala, Lisp, Scheme, Ocaml, SAP, Standard ML y R). La **diferencia** entre ambos estriba en que los lenguajes *funcionales híbridos* son menos estrictos o "conservadores" que los puros, al admitir conceptos tomados de los lenguajes imperativos, como las **secuencias de instrucciones** o la **asignación de variables**. En contraste, **los lenguajes funcionales** puros tienen una mayor potencia expresiva, conservando a la vez su transparencia referencial, algo que no se cumple siempre con un lenguaje funcional híbrido.[1]

En los lenguajes funcionales en general, además, las **estructuras de datos** suelen ser *"inmutables"*, es decir, que una vez que fueron creadas no pueden ser modificadas. Esto es algo necesario en este tipo de lenguajes, sobre todo para mantener las funciones puras, pero también favorece el soporte a la programación concurrente

PRINCIPIOS DE LA PROGRAMACIÓN FUNCIONAL

EFECTOS SECUNDARIOS:

Se dice que una función o expresión tiene **efecto colateral** o **efecto secundario** si ésta, además de retornar un valor, modifica el estado de su entorno. Por ejemplo, una función puede modificar una variable global o estática, modificar uno de sus argumentos, escribir datos a la pantalla o a un archivo, o leer datos de otras funciones que tienen efecto secundario. Los efectos secundarios **frecuentemente hacen que el comportamiento de un programa sea más difícil de predecir**.

La **programación imperativa** generalmente emplea funciones con efecto secundario para hacer que los programas funcionen, la **programación funcional** en cambio se caracteriza por minimizar estos efectos.

---

[1] http://en.wikipedia.org/wiki/Functional_programming

De todos modos, y según (Volkmann, Java News Brief, 2010),

> *In practice, applications need to have some side effects. **Simon Peyton-Jones**, a major contributor to the functional programming language Haskell, said the following: "In the end, any program must manipulate state. A program that has no side effects whatsoever is a kind of black box. All you can tell is that the box gets hotter." (http://oscon.blip.tv/file/324976) The key is to limit side effects, clearly identify them, and avoid scattering them throughout the code.*

Transparencia referencial:

Que una función **no tenga efecto secundario** o colateral es una condición necesaria pero no suficiente para que sea **transparente referencialmente**. El concepto de transparencia referencial significa que una expresión (por ejemplo, una llamada a una función) puede ser reemplazada por su valor; esto requiere que la expresión no tenga efectos colaterales y que sea **pura**, o sea, que siempre retorne siempre el mismo resultado para los mismos valores de entrada.

Funciones Puras:

Los programas funcionales están hechos en base a funciones puras. Una función pura es la que no tiene **efectos secundarios**; el resultado depende **únicamente** de sus argumentos de entrada y su única influencia sobre el exterior es su **valor de retorno**. Una función pura retorna siempre el mismo resultado con los mismos valores de entrada.
Las funciones matemáticas son funciones puras: $log_2(4)$ es siempre *2*, no importa cuántas veces se ejecute, y la función siempre va a limitarse a devolver el mismo resultado.

Funciones de Primera Clase:

Otro concepto sumamente importante para la programación funcional. El concepto de funciones de **primera clase** se refiere al uso de funciones como si de un valor cualquiera se tratara. Una función de primera clase es aquella que puede ser pasada como parámetro a otras funciones y puede ser devuelta como un valor de retorno por éstas.

Funciones de Orden Superior:

Son funciones que pueden:

- Aceptar funciones como parámetros y ejecutarlas dentro más de una vez
- Devolver como resultado otra función

Átomos y Listas:

Según **(Cárdenas, 1997)**, en la mayoría de los lenguajes funcionales, existen dos tipos básicos de palabras: los **átomos** y las **listas**. Todas las estructuras definidas posteriormente son basadas en éstas.

Átomos
Los átomos pueden ser palabras, tal como CASA, SACA, ATOMO, etc. o cualquier disparate como EDSDS, DFKM454, etc. En general, un átomo puede ser cualquier combinación de las 26 letras del alfabeto (excluyendo obviamente la “ñ”) en conjunto con los 10 dígitos.
Al igual que en otros sistemas, por lo general no son átomos aquellas combinaciones que comienzan con dígitos.

Ejemplos de átomos:

- Hola
- Casa
- Uno34

Listas

Una lista es puede ser una secuencia de átomos separados por un espacio y encerrados por paréntesis redondos, incluyendo la posibilidad de que una lista contenga una sublista que cumple con las mismas características.

Ejemplos de listas:

- (ESTA ES UNA LISTA)
- (ESTALISTAESDISTINTAALAANTERIOR)
- (ESTA LISTA (TAMBIEN) ES DISTINTA)
- ((ESTA ES OTRA) (POSIBILIDAD DE LISTA))

En general, definiremos **término** de una lista como un elemento de una lista, ya sea un átomo o una sublista. Así, lista quedaría definida como la secuencia:

($término_1$ $término_2$ … $término_k$)

Donde **k** es el número de elementos de la lista.

FUNCIONES:

El modelo de evaluación para las funciones es muy simple. Cuando el evaluador encuentra una forma **(F A1 A2…)** entonces se asume que el símbolo nombrado **F** es uno de los siguientes:

- Un **operador especial** (fácilmente comprobado contra una lista fija)
- Un **macro operador** (debe haber sido definido previamente)
- El **nombre de una función** (por defecto), que puede ser un símbolo, o un principio de subforma con el símbolo lambda.

Si F es el nombre de una función, después las argumentos A1, A2,…, son evaluados en orden de izquierda a derecha, y la función es encontrada e invocada con esos valores suministrados como parámetros.

Con esto, tratamos de dejar un poco en claro los conceptos de Programación Funcional que creemos necesarios para comprender mejor el resto del trabajo.
Vamos a definir a continuación un poco de qué trata Clojure, el lenguaje en particular en que nos enfocamos y cuáles son sus características.

Breve introducción a Clojure

Clojure es un *dialecto* del lenguaje de programación *Lisp*, pero no es compatible con éste. Es un *lenguaje de programación funcional*, creado por Rich Hickey en el año 2007, adaptado a la programación moderna, y diseñado especialmente para facilitar la **programación concurrente**.

Fue ideado para trabajar con plataformas y librerías existentes, y se ejecuta sobre la JVM, la máquina virtual de Java, y puede acceder a las librerías del paquete java.lang y demás librerías provistas por Java. El proceso es el mismo que en Java, es decir, que los programas se compilan a un *bytecode*, que luego es interpretado por la JVM.

Su **licencia** es Open Source, está liberado bajo la *Eclipse Public License (EPL)*, que está aprobada por la *Open Source Initiative* (OSI) y también por la *Free Software Foundation* (FSF), aunque no es compatible con la GPL.

Como mencionamos antes, algunos lenguajes de programación son "puramente funcionales" (como Haskel). Otros, como Clojure o Scala, sin dejar de ser funcionales también, se los considera "**multiparadigma**", ya que proveen la funcionalidad para implementar más de un *paradigma de programación*.
Clojure, en este sentido, es multiparadigma, ya que soporta los paradigmas **Imperativo**, **Funcional** y **Concurrente**.

Ciclo de vida de un programa en Clojure

Clojure es un lenguaje **homoicónico**[2], que es un término que describe el hecho de que los programas están representados por las **estructuras de datos**. Esta es una diferencia muy importante entre Clojure (y Common Lisp) y otros lenguajes de programación. Clojure se define en términos de la evaluación de estructuras de datos y **no** en términos de la sintaxis de los caracteres.
Cuando se ejecuta un programa en Clojure, una componente llamado **"lector"** ("Reader" en inglés) lee el código del programa en trozos llamados *"formularios"* (que se corresponden con los tipos de datos), y los traduce en **estructuras de datos**.
Estas estructuras pasan luego a un **Evaluador/Compilador**, el cual realiza un proceso de "*expansión de macros*". Las macros son algo similar a las funciones, pero tienen un funcionamiento diferente, y entender tales diferencias es clave para usarlas. **Una función produce un resultado, pero una macro produce una expresión que al ser evaluada produce un resultado**.
Veamos un **ejemplo**. Supongan que queremos escribir la macro *nil!* que asigna a su argumento el valor *nil*. Queremos que *(nil! x)* tenga el mismo efecto que *(setq x nil)*. Lo que hacemos es definir *nil!* como una macro que trasforma casos de la primera forma en casos de la segunda forma

```
defmacro nil! (var)
(list 'setq var nil))
NIL!
```

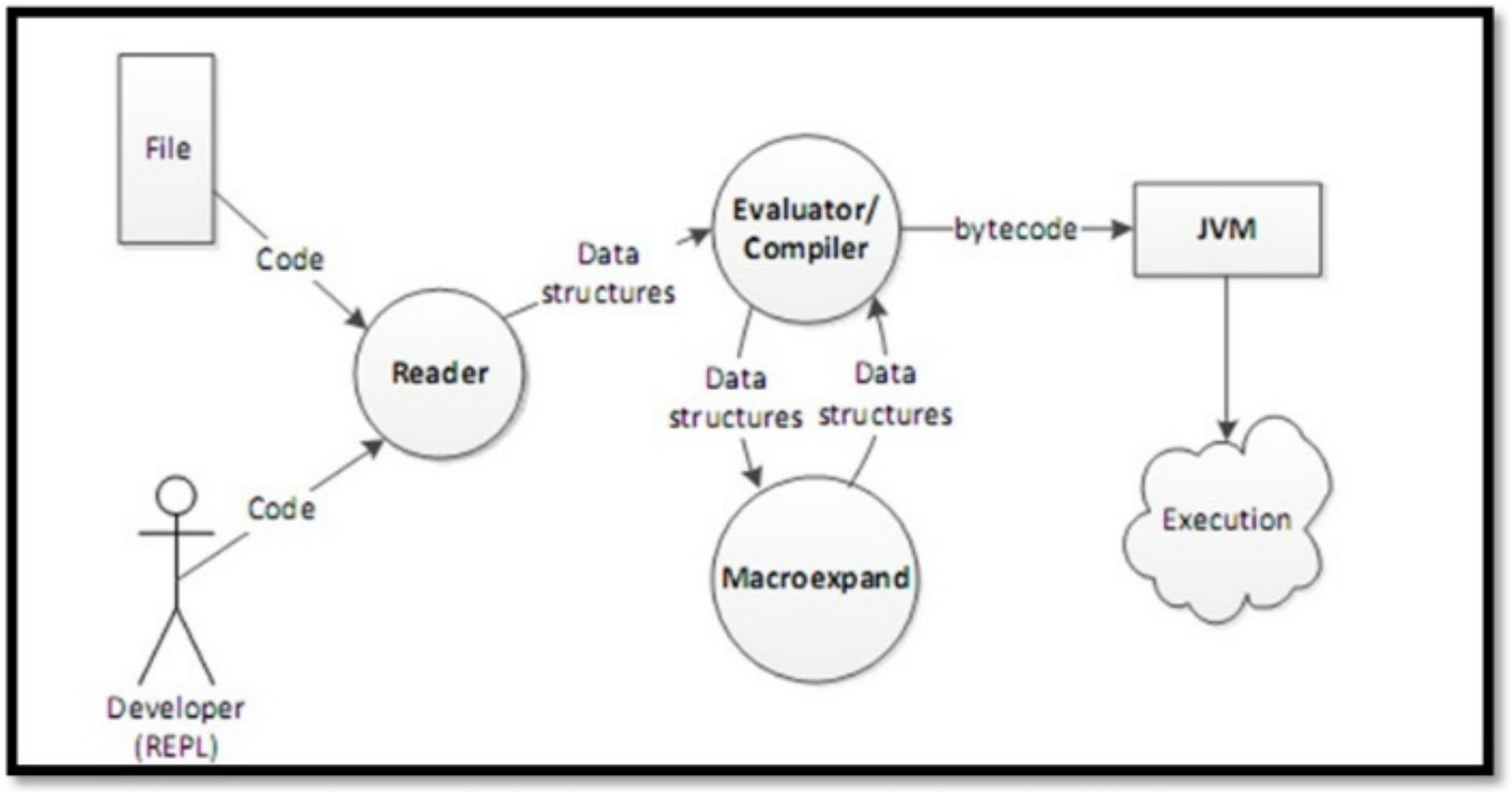
Parafraseada al español, la definición anterior le dice a Lisp *"Siempre que encuentres una expresión de la forma (nil! var), conviértela en una expresión de la forma (setq var nil) antes de evaluarla".*
La expresión generada por la macro será evaluada en lugar de la llamada original a la macro.

---

[2] http://en.wikipedia.org/wiki/Homoiconicity

¿Qué sucede cuando llamamos a la macro con (nil! x)?
Construye la expresión especificada por la definición de la marco, entonces evalúa la expresión en lugar de la llamada original a la macro. El paso que construye la nueva expresión a ser evaluada se llama **macro-expansión**. Tras la macro-expansión, el segundo paso es la evaluación. A grandes rasgos, la idea de macros es similar al del **preprocesador** en C, C++ o C#. Luego, el programa fuente se compila y se genera el bytecode que es interpretado por la maquina virtual de Java.

En el siguiente gráfico se aprecia el ciclo de vida completo de la ejecución de un programa en Clojure:



Ciclo de Vida de un programa en Clojure **(Ott, 2010)**

La sintaxis de Clojure

La **sintaxis** de Clojure es muy similar a la de Lisp. Las expresiones se caracterizan por empezar y acabar con paréntesis y por la **notación prefija Polaca Cambridge** por la que el símbolo que representa la función a evaluar siempre va en primer lugar (no hay operadores, ya que todas las operaciones se consideran funciones). Los programas en Clojure están compuestos de expresiones como las siguientes:

'(1 2 3) – Una lista en Clojure
(+ 2 3) – Devuelve 5
(- (+ 2 3) 5) – Devuelve 0
(+ 1 2 3) – Devuelve 6
(println (+ 1 2 3)) – Imprime en pantalla 6, devuelve *nil*
(hello "world") – Llama a la función hello y le pasa el string *"world"* como parámetro

Clojure es utilizado en proyectos comerciales desde 2008. Algunos de esos proyectos y consultoras que lo utilizan son, según (Ott, 2010):

- FlightCaster – machine learning, etc.
- Relevance, Inc. – consulting
- Runa – marketing services
- Sonian Networks – archiving solutions
- BackType – social media analytics
- DRW Trading Group – trading and finance
- DocuHarvest project by Snowtide Informatics Systems, Inc.
- ThorTech Solutions – scalable, mission-critical solutions
- TheDeadline project by freiheit.com (http://www.freiheit.com/)

Características del Lenguaje

Simplicidad

En Clojure hay un conjunto de **componentes básicos** (tipos de datos, estructuras de control, funciones ofrecidas por el lenguaje, etc.) apropiado para un lenguaje que ofrece soporte al paradigma de programación funcional. Al ser Clojure un dialecto de Lisp, es un poco más simple que éste y ofrece una sintaxis un poco más concisa.
El lenguaje nos brinda la posibilidad de definir **multimétodos**, que se utiliza para ofrecer *polimorfismo de expresiones*, con los cuales podemos referenciar distintas implementaciones según los datos de entrada que reciba este multimétodo. También podemos programar una nueva implementación de una función definida por el lenguaje o redefinirla. Esto nos permite poder llevar a cabo una sobrecarga de funciones[3].

Ortogonalidad

Clojure es un lenguaje que posee una alta ortogonalidad, debido a que, con el conjunto de constructores primitivos, podemos combinarlos de una gran cantidad de formas y así poder construir estructuras de control y de datos más complejas. Cabría hacer la aclaración de que si bien en los lenguajes de programación funcionales puros no hay estructuras de control, Clojure es un híbrido y por lo tanto posee dichas estructuras
Como resultado del análisis de esta característica, vemos que la ortogonalidad manejada por este lenguaje afecta **positivamente** a la legibilidad por su grado de regularidad; a la facilidad de escritura la afecta **negativamente** por la variedad de construcciones que podemos lograr, lo que complejizan la lectura del código.

Tipos de datos

En el paradigma funcional, los programas manipulan los datos. Básicamente, Clojure soporta **Números**, **Strings**, **Caracteres**, **Palabras claves**, **Símbolos**, y **Colecciones**.
Las "*colecciones*" pueden ser:

- Listas: '(1 2 3)
- Vectores: [1 2 3]
- Mapas: {:key1 "Value1" :key2 "Value2"}
- Conjuntos: #{"This" "is" "a" "Set"}
- Structs: (defstruct persona :name :surname :age :address)

Además de estas colleciones, el lenguaje también nos ofrece la posibilidad de **importar colecciones y clases** desde Java, aunque bajo ciertas restricciones (que una vez que se importa un tipo de dato, las reglas que lo manejan van a ser las de Java, entre otras).
A bajo nivel, todos los tipos de datos y colecciones (incluidas las de Java) son tratadas por el lenguaje como "**Secuencias**" (o *"seq"*)[4]. Las *secuencias* en Clojure son como las *listas* en Lisp, pero a un nivel de abstracción mucho más alto (Halloway, 2009).
Las secuencias (y en consecuencia, todos los tipos de datos) son **inmutables**, es decir, que una vez creados no se pueden modificar (esta es la razón por la que el lenguaje ofrece un gran soporte para la concurrencia, que trataremos más tarde).
En el cuadro siguiente, se muestran todos los tipos de datos soportados "nativamente":

---

[3] Hablamos de funciones, debido a que en Clojure no hay "operadores" como en otros lenguajes, sino que se los considera como *funciones*.

[4] Los *flujos* de entrada/salida y los árboles XML también son vistos a bajo nivel como colecciones **(Halloway, 2009)**

Form	Example(s)
Boolean	true, false
Character	\a
Keyword	:tag, :doc
List	(1 2 3), (println "foo")
Map	{:name "Bill", :age 42}
Nil	nil
Number	1, 4.2
Set	#{:snap :crackle :pop}
String	"hello"
Symbol	user/foo, java.lang.String
Vector	[1 2 3]

Tipos de datos en Clojure **(Halloway, 2009)**

Esta característica afecta **positivamente** a la legibilidad y a la facilidad de escritura, por la gran variedad de tipos de datos que ofrece el lenguaje para escribir programas y que hacen al código entendible y fácil de escribir.
Como la cantidad de tipos es amplia, no tenemos la necesidad de estar simulando tipos de datos (como por ej., simular booleanos con enteros), lo que hace al lenguaje mucho más confiable y legible.

Diseño de Sintaxis

Como se mencionó antes, el lenguaje que estamos analizando posee notación prefija Polaca Cambridge, la cual puede resultar difícil de leer. Como elementos de la sintaxis encontramos:

Comentarios

Hay dos formas de realizar comentarios:

```
;esto es u comentario
(esto también es un comentario)
```

El segundo comentario **devuelve un valor**, conocido como *nil* (valor nulo).

Tipo de datos Secuencias

En Clojure, las expresiones, las cuales pueden ser *símbolos* o *listas*.
Un **símbolo** es cualquier cosa que aparece en el programa (átomos o palabras). Una **lista** puede estar vacía o contener símbolos o listas, es decir, estas reglas son recursivas. Al permitir que una lista contenga a otra lista permite representar cualquier expresión compleja.

Macros

Las **Macros** pueden ser utilizadas por cualquier programador para modificar el código del programa o incluso generar código arbitrario durante la ejecución. Esto es lo que hace que Clojure sea tan diferente en otros lenguajes y tan poderoso a la vez.
Esta sintaxis de Clojure afecta de forma **positiva** a la legibilidad y a la facilidad de escritura, ya que, primero, siempre se sabe que *el primer símbolo de la expresión se va a evaluar como una función*, con lo cual siempre vamos a estar seguros que es una función; la dificultad radicaría en encontrar cuál es. Ahora para la facilidad de escritura, creemos que es fácil escribir programas con esta sintaxis, ya que gracias a la notación y a la

sintaxis, primero escribimos la función y seguido de esta todas los argumentos.
Estas facilidades también se extienden para la creación de tipos de datos o la creación/modificación de funciones.

SOPORTE PARA LA ABSTRACCIÓN

Clojure, mediante las colecciones mencionadas anteriormente, permite crear nuevos tipos de datos con los cuales podemos modelar de una mejor manera la realidad que intentamos reflejar en nuestro código. Dentro de éstas estructuras de conjuntos encontramos las listas, los vectores, los mapas, conjuntos y structs.
En cuanto a la **abstracción de procesos**, el lenguaje brinda la habilidad de crear funciones, las cuales pueden simplificarnos la programación, ya sea una función que realice el factorial de un número, o una función que realice un cálculo complejo sobre ecuaciones diferenciales.

EXPRESIVIDAD

La expresividad del lenguaje es sumamente alta, debido a que hay operadores muy poderosos que permiten lograr mucho cálculo con pocas líneas de código. A esto se lo atribuimos a la naturalidad con la que podemos expresarnos bajo este paradigma (funcional).
Si bien, en la sintaxis no es muy natural el hecho de que cada instrucción debe estar encerrada entre paréntesis, si son naturales las operaciones, manejo de las llamadas a funciones, las construcciones básicas que provee el lenguaje y su sintaxis natural.
Esta expresividad trae consigo efectos positivos para la legibilidad y la facilidad de escritura por las razones de que es simple realizar un programa bajo el paradigma funcional con Clojure, además de que es simple, también, entenderlo.

CHEQUEO DE TIPOS

Según (Rathore, 2010), durante la fase en la que el “Reader” lee el código fuente para traducirlo en las estructuras de datos, se hace un chequeo de tipos para que los tipos de datos coincidan con los esperados para las funciones y expresiones a evaluar. Las expresiones no válidas generan un “*error de lectura*”, con lo se muestra un mensaje de error apropiado y se aborta el programa.
En la segunda fase, se evalúan los las expresiones y sus valores de retorno. El código es compilado luego a bytecode, y se ejecuta sobre la JVM.
Este punto afecta **positivamente** la confiabilidad del lenguaje, ya que el chequeo de tipos que realiza el lenguaje es riguroso.

MANEJO DE EXCEPCIONES

En Clojure, el manejo de excepciones es muy similar a Java, pero está mucho más simplificado. Provee mecanismos que hacen que no sea tan necesario capturar explícitamente las excepciones que se generan en **tiempo de ejecución**, a menos que sea sumamente necesario. Las estructuras especiales *try* y *throw* ofrecen todas las funcionalidades del *try*, *catch*, *finally*, y el *throw* en Java.
Pero no es necesario utilizarlas muy a menudo, por las siguientes razones:

- No se necesita capturar las excepciones explícitamente en Clojure
- Se pueden utilizar las macros *with-open* que ofrecen una funcionalidad similar al *finally* en Java, que garantizan que después de utilizar un recurso, por más que se genere una excepción, el recurso sea borrado o cerrado (internamente , la macro genera una estructura try y la atiende de ser necesario)

Esta característica del lenguaje afecta **positivamente** su **Confiabilidad**, ya que el programador no tiene que estar tan pendiente de atender explícitamente al manejo de excepciones y a los errores que pueden ocurrir en la ejecución.

CONCURRENCIA

Como técnicas de Clojure para el soporte de concurrencia podemos incluir a los **agentes**, que realizan las acciones de manejo de estado y manejo de ejecución.

Como las variables son tratadas como trataría cualquier lenguaje a una *constante*, es decir, permanecen inmutables, es ahí donde entra en acción el **manejo de estados** ya que poco podríamos modelar si no podríamos modificar las variables. En el manejo de estados lo que realiza el agente es, como todos los elementos internamente los trata como secuencias, si necesitamos eliminar o insertar datos en una variable, crea una nueva variable y los enlaza o los borra temporalmente.

En el **manejo de ejecución**, los agentes son los que dan ejecución, pausa o muerte a los *subprogramas*, y su propósito es, obviamente, la concurrencia pero con un aprovechamiento eficiente de los procesadores por lo cual requiere de la JVM para no solo saber cuántos procesadores hay sino que además de los costos de los accesos remotos a los recursos.

REFERENCIAS

- **(Halloway, 2009)** *Programming Clojure*, Stuart Halloway, 2009, The Pragmatic Programmers, LLC.
- **(Rathore, 2010)** *Clojure in Action*, Amit Rathore , 2010, Manning Publications
- **(Fogus & Houser, 2010)** *The Joy of Clojure – Thinking the Clojure Way*, Michael Fogus, Chris Houser, 2010, Manning Publications
- **(VanderHart & Sierra, 2010)** *Practical Clojure*, Luke VanderHart, Stuart Sierra, 2010, ED books
- *http://clojure.org*, Sitio oficial del proyecto
- *Clojure and The Robot Apocalypse - Needfuls for Newbies*, slides de una conferencia dictada por la gente de http://www.porticosys.com, Agosto de 2010
- **(Volkmann, 2009)** *Clojure: dynamic, functional programming for the JVM*, slides de una conferencia dictada por R. Mark Volkmann, Mayo de 2009
- **(Volkmann, 2010)** *http://java.ociweb.com/mark/clojure/article.html*, R. Mark Volkmann , Partner Object Computing, Inc. (OCI), last updated on 21/07/2010
- **(Ott, 2010)** *Clojure: A small introduction*, slides de una conferencia dictada por Alex Ott, Julio de 2010, http://alexott.net
- **(Cárdenas, 1997)** *Manual de Referencia Rápida de LISP*, Rolando Burgos Cárdenas, Universidad de Concepción - Facultad de Ingeniería, Departamento de Ingeniería Informática Y Ciencias de la Computación
- *http://www.wikipedia.org*, Wikipedia Enciclopedia Libre