Parallel Lisp

Ejercicio N° 2

Objetivos	• Solución orientada a objetos en C++ • Diseño y construcción de sistemas con procesamiento concurrente • Encapsulación y manejo de Threads • Protección de recursos compartidos
Instancias de Entrega	**Entrega 1:** clase 6 (12/4/2016). **Entrega 2:** clase 8 (12/4/2016).
Temas de Repaso	• Implementación de herencia y polimorfismo en C++ • Manejo de librerías básicas de C++: std::string, std::fstream, std::iostream • POSIX Threads • RAII
Criterios de Evaluación	• Criterios de ejercicios anteriores • Buenas prácticas de diseño OOP • Correcto uso de estructuras C++ para el diseño de clases • Ausencia de secuencias concurrentes que permitan interbloqueo • Ausencia de condiciones de carrera en el acceso a recursos • Buen uso de Mutex, Condition Variables y Monitores para el acceso a recursos compartidos

Índice

Introducción

Se nos encarga la construcción de un intérprete de operaciones matemáticas con objetivos científicos. En esta ocasión se pretende utilizar las particularidades de Lisp, un conocido lenguaje funcional mixto, con el agregado de ejecuciones paralelas que permitan mejorar la performance general.
Si bien la tarea de implementación de un intérprete de programación no es sencilla, se pretende soportar un conjunto reducido de *keywords* del lenguaje con el fin de evaluar la nueva performance conseguida. Luego, dependiendo de los resultados, se decidirá si es conveniente la inversión total.

Descripción

El lenguaje Lisp [1] fue creado a fines de la década del ‘50 convirtiéndose en uno de los primeros lenguajes de alto nivel. Si bien se basa en fundamentos de programación funcional, su evolución fue marcada por la inclusión variantes y atajos que resultan prácticos pero lo alejan de la teoría funcional pura. En efecto, se considera a Lisp como un lenguaje multiparadigma pero a su vez el precursor de los lenguajes funcionales.
En la actualidad existen distintos intérpretes de Lisp. Uno de los más conocidos es Common Lisp [2] que será utilizado como referencia para el intérprete a construir.

Funciones y entorno

Las funciones utilizadas en Lisp guardan una importante relación con las funciones matemáticas y más precisamente con el cálculo Lambda [3]. En resumidas cuentas, una función es una expresión que recibe un entorno (o argumentos), realiza cierto proceso y retorna un nuevo entorno (o resultado). En caso de ser necesario, el entorno recibido o retornado puede ser vacío, pero esta decisión puede afectar a las funciones subsiguientes ya que, en Lisp, las funciones pueden encadenarse para que la ejecución de una se base en el resultado de otra.
Ejemplos de funciones estándares de Lisp:

- **+**: realiza la suma de todos los enteros indicados
- **-**: toma el primer entero y le resta todos los enteros indicados a continuación
- ***:** multiplica todos los números enteros indicados
- */*: toma el primer entero y lo divide por todos los enteros indicados a continuación. Notar que el resultado puede ser un número flotante.
- **=, <, >:** toman dos enteros y los comparan retornando 1 en caso positivo y una lista vacía en caso negativo.
- **car**: dada una lista de entrada, toma su primer valor y lo retorna en el entorno de salida. Si no hay

entrada o la lista está vacía, retorna una lista vacía.
- **cdr**: dada una lista de entrada, toma todos los valores salvo el primero y los retorna en el entorno de salida. Si no hay entrada o la lista está vacía, retorna una lista vacía.
- **append**: dadas una o más listas como entrada, toma todos los valores y los concatena en una nueva lista que se retorna como resultante.

Las funciones estándares permite contar con un conjunto de instrucciones para construir algoritmos mínimos, pero gracias a función especial **defun**, es posible definir nuevas funciones y resolver situaciones más complejas.

Elementos: Átomos y listas

En Lisp, las construcciones más simples para almacenar elementos son los átomos y las listas.
Los **átomos** son elementos básicos que pueden consistir en números, cadenas o símbolos. Los elementos constantes como números o cadenas no requieren evaluación, sin embargo los símbolos refieren a elementos variables, cuyo valor se define en tiempo de ejecución.
A fines prácticos, se define para el sistema a utilizar que los **símbolos** se tratan únicamente de variables que pueden almacenar átomos o listas (o tal vez una función definida con **defun**). A su vez, asumiremos que los **símbolos** siempre serán evaluados si es que la expresión que los contenga requiere ser evaluada. Si bien, esto no es cierto en Lisp, el tratamiento de símbolos a funciones y otras variantes puede resultar muy complejo.

Las **listas**, son estructuras que almacenan cero, uno o más elementos de forma ordenada para su posterior uso. Para construir una lista, sólamente se requiere encerrar entre paréntesis al símbolo **list** junto con los elementos deseados, separándolos con uno o más caracteres en blanco.
Cabe aclarar que los elementos pueden ser átomos o listas, siendo la anidación de listas un concepto de abstracción muy poderoso que da lugar a estructuras N-dimensionales de ser necesario. A continuación se presentan algunos ejemplos de átomos y listas:

- `2` `;número 2`
- `"Texto"` `;cadena presentando la palabra "Texto"`
- `(list 1 2 3 4)` `;lista almacenando los números 1, 2, 3, 4`
- `(list 1 "Texto" 2 (list 3 4))` `;lista almacenando el número 1, la cadena "Texto", el número 2 y la lista (3 4)`
- `(list)` `;lista vacía`

> **<u>Nota:</u> la posibilidad de utilizar cadenas de caracteres se plantea como un opcional para el alcance del trabajo**

La lista vacía en Lisp posee un doble significado. Por un lado, indica una colección vacía de elementos y por otro define el equivalente a **FALSO** para lenguajes imperativos. Esto es, una condición NO VERDADERA frente a una función condicional como **if** (ver más adelante).

Definición y evaluación de funciones

La invocación de funciones en Lisp consiste simplemente en encerrar el símbolo que represente la función en cuestión seguido de los argumentos que requiere como entorno. Veamos algunos ejemplos con las funciones anteriores:

- `(+ 1 2 3)` `;retorna 6`

- `(/ 10 5)` `;retorna 2`
- `(list 1 2 3)` `;en realidad, **list** es una función que crea listas con el entorno`

Las expresiones pueden ser escritas de forma secuencial, separadas por \n y se evaluarán de dicha manera:

- `(+ 1 2 3)`
  `(- 4 2)` `;ejecuta primero (+ 1 2 3) y luego (- 4 2)`

A su vez, si recordamos que todo símbolo debe ser evaluado si se está evaluando la expresión que lo contiene, podemos anidar funciones:

- `(+ (* 2 3) (*3 4))` `;retorna 18`
- `(car (list 1 2 3))` `;retorna 1`
- `(cdr (list 1 2 3))` `;retorna (2 3)`
- `(car (cdr (list 1 2 3)))` `;retorna 2`
- `(car (list))` `;retorna ()`

Existe una función estándar llamada **defun** que nos permite definir nuevas funciones. Para tener efecto, requiere el nombre de la función a definir, seguida por los argumentos y el cuerpo. En este caso especial, argumentos y cuerpo son encerrados en paréntesis y NO son evaluados hasta que lo requiera la función definida. Ejemplos:

- `(defun primero (ENV) (car ENV))` `;pendiente de evaluación`
  `(primero (list 1 2 3 4))` `;retorna 1`
- `(defun lista_con_primero (ENV) (list (car ENV)))` `;pendiente de evaluación`
  `(lista_con_primero (list 1 2 3 4))` `;retorna (1)`

Para el presente trabajo, utilizaremos una simplificación importante: la única variable que puede recibir una función definida con **defun** es el entorno de ejecución. Este parámetro será indicado siempre con ENV y corresponderá a utilizado al momento de evaluar la función. Ejemplo:

- `(defun calculo (ENV) (+ (car ENV) (car (cdr ENV))))` `;ENV no está definido aún`
  `(calculo (list 1 2 3 4))` `;ENV=(list 1 2 3 4) con lo`
  `;cual se ejecuta (+ 1 2)`
- `(calculo (list 3 4))` `;ENV=(list 3 4) con lo`
  `;cual se ejecuta (+ 3 4)`

> **<u>Nota:</u> la posibilidad de admitir defun para declaración de funciones se plantea como un opcional para el alcance del trabajo**

Condiciones, variables y recursividad

El uso de expresiones condicionales en Lisp es muy discutido ya que este concepto es el pilar de la programación imperativa en contraposición con los preceptos de programación funcional. De cualquier forma, las expresiones condicionales permite flexibilizar el uso de Lips permitiendo atacar rápidamente un nuevo conjunto de problemas:

- **if**: recibe 3 expresiones, si la ejecución de la primera no es FALSO (es decir, no es una lista vacía), ejecuta y retorna la segunda. En caso contrario, ejecuta y retorna la tercera.

Vemos entonces que una lista vacía tiene un significado especial muy parecido al concepto de FALSE en programación imperativa.

Veamos algunos ejemplos:

- `(if 1 2 3)` `;retorna 2`
- `(if (list) 2 3)` `;retorna 3`
- `(if 1 (car (list 1 2 3)) (cdr (list 1 2 3))` `;retorna 1`
- `(if (list) (car (list 1 2 3)) (cdr (list 1 2 3))` `;retorna (2 3)`

Por último, si es posible definir funciones, parece apropiado permitir la definición de variables. Esto se hace con la función **setq** donde se requiere el símbolo de la variable y su valor. La definición será global a todo el programa.
Veamos unos ejemplos:

- ```
  (setq variable 1234)                ;define variable=1234 de forma global
  (setq variable 123)                 ;sobreescribe variable con 123
  (car (list variable 456))           ;retorna 123
  ```

Por último, el lenguaje admite recursividad de llamadas dentro de la definición de funciones como muestra el siguiente ejemplo:

- ```
  (defun reverse (ENV) (if ENV (append (reverse (cdr lista)) (list (car lista)))
  (list)))
  ```

Notar que la función **if** necesariamente debe verificar la condición ANTES de evaluar la expresión primaria o secundaria. Si así no fuera y ambas expresiones se evaluaran a la vez, no ocurriría la condición de corte y **reverse** sería llamada recursivamente hasta el infinito.

> **<u>Nota:</u> la posibilidad de incluir funciones recursivas se plantea como un opcional para el alcance del trabajo**

Impresión

La función **print** permite imprimir por salida estándar al átomo utilizado como argumento y agrega un fin de línea. A fines de simplificar el presente trabajo, se define que la función print debe imprimir todos los elementos indicados en el entorno, tanto listas como átomos, separados por un espacio.
En caso de tratarse de listas, se deben encerrar los valores entre paréntesis y separarlos por espacios.
Veamos unos ejemplos:

- `(print 1234)` `;imprime "1234\n"`
- ```
  (setq variable 1234)
  (print "variable =" variable)        ;imprime "variable = 1234\n"
  ```
- `(print (list 1 2 3) 4 5` `;imprime "(1 2 3) 4 5\n"`

Existen otras formas de impresión en Lisp que no serán implementadas en esta fase del sistema.

Modificaciones y Paralelismo

A fin de poder controlar el paralelismo necesario para el cómputo científico se decidió automatizar la ejecución paralela pero permitiendo al usuario que agregue puntos de sincronización de forma explícita. Se define entonces la función **sync** que fuerza la espera de todo cómputo paralelo para avanzar con las siguientes instrucciones.
En otras palabras:

- Toda expresión a ejecutar será lanzada en paralelo de forma automática.
- La lucha por recursos compartidos (básicamente, el entorno de símbolos globales) no posee un control definido pudiendo las instrucciones competir por su acceso.
  ```

- Frente a una expresión **sync**, el sistema espera a todas las expresiones paralelas anteriores y luego retoma la tarea de interpretar el programa Lisp.

Veamos un ejemplo:

```
(setq variable 1234)
(setq variable2 5678)
(setq variable (* variable 100))
(setq variable2 (* variable2 100))
(sync)
(setq variable (+ variable variable2))
(sync)
(print "variable =" variable)
```

Esta sintaxis puede parecer engorrosa para operaciones que no son naturalmente paralelas. Sin embargo, existen cálculos con contextos de variables definidos en forma local que obtendrían grandes ventajas de esto. A fines de simplificar el trabajo, no trataremos estas variantes.

Simplificaciones y funciones permitidas

A fin de simplificar el alcance del sistema, se pueden asumir lo siguiente:

- Todas las expresiones a ejecutar serán escritas en una única línea.
- Las expresiones que no puedan ser interpretadas en una única línea se considerarán erróneas, fallando la ejecución y reportando el problema.
- No se admiten impresiones por consola salvo en las instrucciones **print.** Como excepción a la regla, se admite el reporte de los errores indicados en la sección "Códigos de Retorno y Salida de Error"
- La cantidad de hilos a utilizar para instrucciones en paralelo no está limitada. Aunque pueda obtenerse un programa de bajo rendimiento por estos inconvenientes, se tomó una decisión estratégica de delegar el control de concurrencia en el usuario programador.
- Se deben implementar únicamente las siguientes funciones Lisp:
    - +, -, *, /, list, append, car, cdr, print, if, setq, sync
- Los siguientes requerimientos se pueden considerar como opcionales:
    - Implementación de cadenas de caracteres.
    - Implementación de defun y funciones recursivas.
    - Implementación de operadores =, > y <.

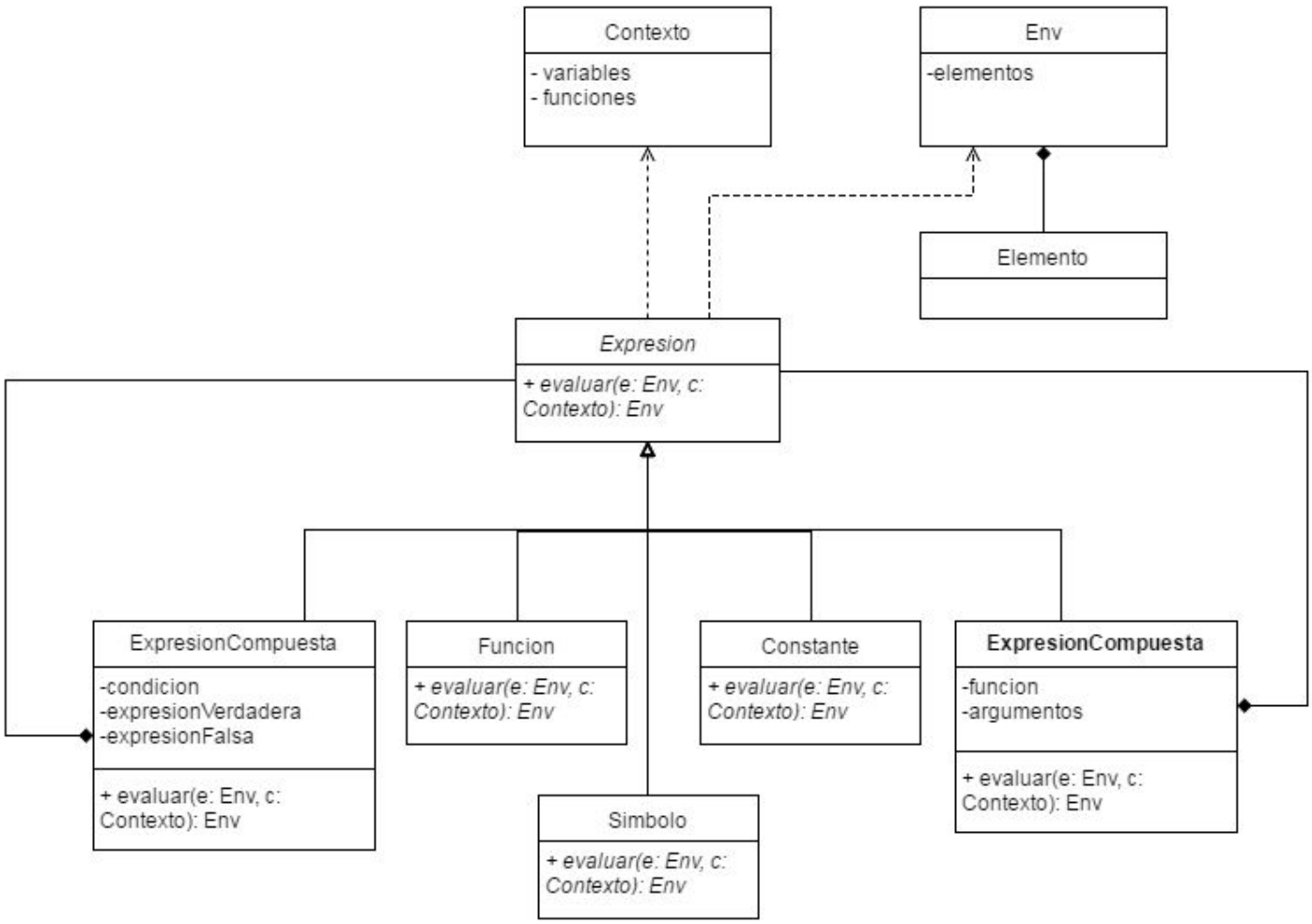
Sugerencias de implementación

A continuación se detalla la secuencia de implementación sugerida para facilitar el proceso de construcción:

1. Implementar un parser simple que permita identificar las funciones más usadas (+, car, etc) y sus argumentos constantes.
2. Construir en memoria la expresión parseada y permitir su evaluación usando los argumentos como entorno.
3. Implementar la función print y la composición de funciones. Probar su uso.
4. Permitir la definición de variables globales en el contexto y su uso mediante símbolos en lugar de constantes.
5. Implementar la definición de nuevas funciones sin recursividad.
6. Implementar el condicional (if) tomando en cuenta el orden de evaluación para permitir condiciones

de corte y recursividad.
7. Permitir que el intérprete resuelva muchas instrucciones en secuencia.
8. Encapsular los conceptos de hilos y realizar una prueba simple en un sistema auxiliar.
9. Lanzar la evaluación de expresiones en distintos hilos e implementar sync para esperarlas.

A continuación se presenta un diagrama de clases tentativo para comprender el concepto de árboles de expresión:



Formato de Línea de Comandos

El sistema se ejecutará sin ningún argumento, debiendo fallar con un código de error en caso detectar alguno:

```
./tp
```

Códigos de Retorno y Salida de Error

El sistema debe retornar 0 en caso de finalizar correctamente su ejecución. Si alguno de los siguientes errores fueran detectados, el sistema debe finalizar tan pronto como sea posible:

- Se detectaron argumentos de entrada. Código 1. Salida de error: “ERROR: argumentos\n”
- Se detectó una expresión inválida. Código 2. Salida de error: “ERROR: <línea inválida>\n”

Entrada y Salida Estándar

El programa será ingresado por entrada estándar e interpretado línea a línea.
La salida estándar sólo imprimirá el detalle indicado por la instrucción **print.**

Ejemplos de Ejecución

A continuación se presentan 2 ejemplos de ejecución con distintos programas.

Ejemplo 1

Entrada

```
(defun op1 (lista) (+ (car lista) (op1 (cdr lista))))
(defun op2 (lista) (* (car lista) (op2 (cdr lista))))
(defun reverse (lista) (if lista (append (reverse (cdr lista)) (list (car
    lista))) (list) ))
(setq valores (list 1 2 3 4 5 6 7 8 9))
(sync)
(setq result1 (op1 valores))
(setq result2 (op2 valores))
(setq result3 (op1 (reverse valores)))
(setq result4 (op2 (reverse valores)))
(sync)
(print "Result1 =" result1 "\nResult2 =" result2 "\nResult3 =" result3
"\nResult4 =" result4 "\n")
```

Salida

```
Result1 = 45
Result2 = 362880
Result3 = 45
Result4 = 362880
```

Ejemplo 2

Entrada

```
(defun max (ENV) (if ENV (if (> (car ENV) (max (cdr ENV))) (car ENV) (max
(cdr ENV))) (list)))
(defun min (ENV) (if lista (if (< (car ENV) (min(cdr ENV))) (car ENV)
(min (cdr ENV))) (list)))
(setq valores (list 1 2 3 4 5 6 7 8 9))
(sync)
(print "Min =" (min valores) " Max =" (max valores) "\n")
```

Salida

```
Min = 1 Max = 9
```

Restricciones

La siguiente es una lista de restricciones técnicas exigidas por el cliente:

1. El sistema debe desarrollarse en ISO C++98.
2. Está prohibido el uso de variables o funciones globales (salvo main)
3. Debe utilizarse pthread como librerías de soporte multithreading.

Referencias

[1] https://es.wikipedia.org/wiki/Lisp
[2] https://es.wikipedia.org/wiki/Common_Lisp
[3] https://es.wikipedia.org/wiki/C%C3%A1lculo_lambda