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:

- +, -, *, /, =, <, >: operaciones aritméticas básicas que actúan sobre un entorno de dos o más números para retornar un entorno con un único elemento.
- **car**: dada una lista de entrada, toma su primer valor y lo retorna en el entorno de salida.
- **cdr**: dada una lista de entrada, toma todos los valores salvo el primero y los retorna en el entorno de salida.
- **append**: dadas dos listas como entrada, se concatenan las listas y se retorna la lista 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.

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. 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.
Existe un símbolo especial definido por **nil** que permite contar con un átomo distintivo para operaciones de verdad como veremos

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`

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`

Las expresiones pueden ser escritas de forma secuencial 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 (lista) (car lista))
  (primero (list 1 2 3 4))                    ;retorna 1
  ```
- ```
  (defun primera_lista (lista) (list (car lista)))
  (primera_lista (list 1 2 3 4))              ;retorna (1)
  ```

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 elementos, si la ejecución del primero no es una lista vacía, ejecuta y retorna el segundo. En caso contrario, ejecuta y retorna el tercero.
- **cond**: recibe una lista con listas de pares **(valor resultado)**. Recorre el listado de pares evaluando y verificando **valor**. En caso de obtener una lista no vacía, detiene el recorrido, evalúa **resultado** y retorna. En caso contrario, sigue con la evaluación hasta encontrar un caso positivo o el final de la lista en cuyo caso retorna con una lista vacía.

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)`
- `(cond (1 2) (3 (list 1 2 3)))` `;retorna 2`
- `(cond ((list) 2) (3 (list 1 2 3)))` `;retorna (1 2 3)`

Por último, si es posible definir funciones, parece apropiado permitir la definición de variables. Esto se hace con dos funciones: **setq** y **let**. En el caso de **setq** se requiere el símbolo de la variable y su valor como entorno. La definición será global a todo el programa. En el caso de **let**, se requiere una lista con una serie de pares **(símbolo valor)** que serán definidos de forma local seguidos por la expresión a ejecutar con esta definición. **let** define todos los símbolos en el entorno actual continúa con el ciclo de ejecución eliminando toda definición antes de su retorno. Veamos unos ejemplos:

- `(setq variable 1234)` `;define variable=1234 de forma global`
- `(let ((variable 12) (variable2 34)))` `;define variable=12 y variable2=34 locales`
- ```
  (setq variable 123)
  (car (list variable 456))                   ;retorna 123
  ```
- ```
  (let ((variable 12) (variable2 34))
        (cdr (list variable variable2)))      ;retorna (34)
  ```
- ```
  (setq variable 123)
  (let ((variable 12) (variable2 34))
  ```
  ```

```
      (car (list variable variable2)))  ;retorna 12
(car (list variable variable2))         ;retorna 123
(cdr (list variable variable2))         ;retorna ()
```

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

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

Impresión

La función **print** permite imprimir por salida estándar al átomo utilizado como argumento. Luego de la impresión, agrega un fin de línea. Otra variante es utilizar **prin1** que simplemente no agrega el fin de línea.

En nuestro caso, los argumentos aceptados serán números, cadenas o listas así como cualquier símbolo que almacene estos tipos de datos. No se admitirán funciones como argumentos de **print o prin1,** quedando indefinido el comportamiento en este caso. Veamos unos ejemplos:

- ```
  (print 1234)                                  ;imprime "1234\n"
  ```
- ```
  (setq variable 1234)
  (prin1 "variable=")                           ;imprime "variable="
  (prin1 variable)                              ;imprime "1234"
  (prin1 "\n")                                  ;imprime "\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)`
- `(prin1 "variable=")`
- `(sync)`
- `(prin1 variable)`

Como agregado, se requiere que la función **sync** pueda recibir una lista que será ejecutada sincrónicamente, luego de que todas las instrucciones paralelas pendientes hayan finalizado. De esta forma, la secuencia anterior se podría reescribir como:

- `(setq variable 1234)`
- `(setq variable2 5678)`
- `(setq variable (* variable 100))`
- `(setq variable2 (* variable2 100))`
- `(sync ((setq variable (+ variable variable2)) (prin1 "variable=") (prin1 variable))`

Esta sintaxis puede parecer engorrosa para operaciones que no son naturalmente paralelas. Sin embargo, aquellos cálculos que lo permiten, pueden obtener grandes ventajas:

- `(let ((var1 12) (var2 56)) (setq result1 (+ (* var1 10) (* var2 10))))`
- `(let ((var1 78) (var2 90)) (setq result2 (+ (* var1 10) (* var2 10))))`
- `(sync ((prin1 "results=") (prin1 result1) (prin1 ",") (prin1 result2) (prin1 "\n")))`

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** y **prin1.** 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, defun, print, prin1, if, cond, setq, let, sync

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 las instrucciones **print** y **prin1.**

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 (list) (append (reverse (cdr lista))
    (list (car lista)))))
(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 ((prin1 “Result1=”) (prin1 result1) (prin1 “\n”) (prin1 “Result2=”)
(prin1 result2) (prin1 “\n”) (prin1 “Result3=”) (prin1 result3) (prin1
“\n”) (prin1 “Result4=”) (prin1 result4) (prin1 “\n”)))
```

Salida

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

Ejemplo 2

Entrada

```
(defun max (lista) (if lista (if (> (car lista) (max (cdr lista))) (car
lista) (max (cdr lista))) (list)))
(defun min (lista) (if lista (if (< (car lista) (min(cdr lista))) (car
lista) (min (cdr lista))) (list)))
(setq valores (list 1 2 3 4 5 6 7 8 9))
(sync ((prin1 “Min=”) (prin1 (min valores)) (prin1 “ Max=”) (prin1 (max
valores)) (prin1 “\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