Máquina virtual de Self

Ejercicio Final

Objetivos	● Afianzar los conocimientos adquiridos durante la cursada. ● Poner en práctica la coordinación de tareas dentro de un grupo de trabajo. ● Realizar un aplicativo de complejidad media con niveles aceptables de calidad y usabilidad.
Instancias de Entrega	**Pre-Entrega:** clase 14 (15/11/2016). **Entrega:** clase 16 (29/11/2016).
Temas de Repaso	● Aplicaciones Cliente-Servidor multi-threading. ● Interfaces gráficas con *gtkmm* ● Manejo de errores en C++
Criterios de Evaluación	● Criterios de ejercicios anteriores. ● Construcción de un sistema Cliente-Servidor de complejidad media. ● Empleo de buenas prácticas de programación en C++. ● Coordinación de trabajo grupal. ● Planificación y distribución de tareas para cumplir con los plazos de entrega pautados. ● Cumplimiento de todos los requerimientos técnicos y funcionales. ● Facilidad de instalación y ejecución del sistema final. ● Calidad de la documentación técnica y manuales entregados. ● Buena presentación del trabajo práctico y cumplimiento de las normas de entrega establecidas por la cátedra (revisar criterios en sitio de la materia).

Índice

Introducción

Self es un lenguaje de programación orientado a objetos, prototipado y dinámico desarrollado a finales de los 80. A pesar de no haber llegado nunca a ser un lenguaje de producción, es fuente de inspiración de lenguajes como Javascript y IO asi como también innovó en tecnología de máquinas virtuales, tecnologías que hoy pueden encontrarse en las máquinas virtuales de Java o Python con el *garbage collector generacional* y el *just in time*.

Descripción

Self es un lenguaje de programación orientado a objetos, prototipado y dinámico por lo tanto el concepto de objeto es central.
Un objeto en Self puede tener cero o mas slots y opcionalmente codigo asociado. Los slots servirán para definir los atributos de los objetos como los argumentos de un método.
A diferencia de su predecesor, Smalltalk, en Self no existe una separación entre clases e instancias de clases. En Self sólo hay objetos y la herencia entre objetos no se basa en una estructura jerárquica (clases) sino en prototipos.

A continuación se detallan los distintos elementos que constituyen un programa hecho en Self, desde su sintaxis hasta su entorno de desarrollo pasando por algunos detalles de su máquina virtual.
El presente trabajo consiste en implementar un *subconjunto* del lenguaje.

Slots

Todo objeto en Self puede tener cualquier número de slots.
Cada slot está identificado por un nombre y pueden ser de tipo inmutables (de solo lectura) o bien mutables.

El valor que puede contener cada slot puede ser *cualquier* objeto: un int, un string, un método.

Por ejemplo, el siguiente código define 2 slots, uno inmutable llamado `x` y el otro mutable llamado `y`.
Ambos inicializados con los objetos 3 y 4 respectivamente.

```
x = 3.
y <- 4.
```

Una vez definido un slot mutable se le puede cambiar el valor con la sintaxis:

```
y: 8.
```

Si un slot no se lo inicializa por default Self crea un slot mutable inicializado con el objeto `nil`.
Por ejemplo:

```
a.
```

es equivalente a:

```
a <- nil.
```

Aunque los slots de un objeto se definen al momento de crearlo, en algunos casos es necesario crear y agregar nuevos slots a un objeto ya existente. Para ello se invoca el método `_AddSlots`: con la siguiente sintaxis (durante el resto del enunciado se explicará en más detalle esta sintaxis):

```
objeto _AddSlots: (| x = 3. y <- 4. |).
```

En este caso se agregaron 2 slots más: `x` e `y`.

Del mismo modo a un objeto se le puede sacar uno o varios slots con:

```
objeto _RemoveSlots: (| x. y. |).
```

Mensajes

Todo código en Self es una serie de mensajes que les son enviados a los objetos y que son ejecutados en secuencia.
Todo mensaje tiene un destinatario o receptor (receiver) que puede estar explícito en el código o no, dependiendo del contexto.

Hay tres tipos de mensajes basados en la cantidad de argumentos que reciben: los unarios, los binarios y los

basados en keywords (n-arios)
A continuación un ejemplo de cada uno de ellos:

```
x print.
x + 3.
x min: 1 Max: 2.
```

Los mensajes unarios no reciben ningún argumento y su sintaxis es simple:

```
receiver message.
```

Los mensajes por keywords, en cambio, reciben 1 o más argumentos y su sintaxis es

```
receiver keyword1: argumento1 Keyword2: argumento2 ... KeywordN: argumentoN.
```

Nótese como la primer keyword va en minúsculas mientras que las subsecuentes van en mayúsculas. Viniendo de otros lenguajes como C/C++, Python o Javascript puede resultar una situación extraña pues el nombre del mensaje está compuesto por múltiples palabras.

Por ejemplo:

```
x min: 1 Max: 2.
```

Al objeto `x` se le envía el mensaje `min:Max:` con los argumentos 1 y 2 respectivamente.

Los mensajes binarios no son más que mensajes con una sintaxis más conveniente que es:

```
receiver operator argument
```

Esto resulta particularmente útil para los operadores aritméticos:

```
x + 3.
```

que sin embargo son equivalentes a los mensajes por keyword de un solo argumento. Para el ejemplo anterior:

```
x plus: 3.
```

Por simplicidad los únicos operadores que soporta nuestra implementación son: `+, -, *, /, ==` y `!=.` Recibiendo siempre objetos numéricos.

Cada tipo de mensaje tiene su propio nivel de precedencia y de asociatividad pero nuestra implementación requerirá el *uso de paréntesis explícito* para desambiguar.

Por ejemplo, en Self:

```
x twice print.
```

es equivalente a:

```
(x twice) print.
```

porque los mensajes unarios tienen asociatividad de izquierda a derecha. Sin embargo, y con fines de simplificar el trabajo, el parser no forzará ninguna asociatividad ni precedencia y es responsabilidad del programador de usar los paréntesis explícitamente:

```
(x twice) print.
```

Por ejemplo:
```
(x plus: 3) minus: 4.
```
es muy distinto a:
```
x plus: (3 minus: 4).
```

Mas ejemplos:
```
3 + (4 * 7)
aList append: 1.
aList insert: 2 At: 0.
aList extend: (otherList clone).
aList search: (1 + 3).
(aList append: (2 + (3 factorial))) print.
```

Objetos

En Self todo es un objeto compuesto por cero o más slots, un slot adicional implícito llamado `self` (véase la sección de lookup) y opcionalmente código asociado.
Los data objects son objetos de Self que no tienen código sino solamente slots.

Por ejemplo el objeto punto (1,2) lo podríamos escribir como:
```
(| x = 1. y = 2. |)
```

Si quisiéramos que los slots fueran mutables escribimos:
```
(| x <- 1. y <- 2. |)
```

Los objetos con código asociado se los conoce como *method objects* porque sirve justamente para implementar los métodos de otros objetos.
Por ejemplo, el método retorna siempre un valor constante se podría implementar como:
```
(| |
  42.
)
```

En Self, el operador ^ es quien indica el valor a retornar pero en nuestra implementación haremos que los métodos retornen el valor de la última expresión ejecutada.

Y si quisiéramos un método que multiplique por 2 su argumento:
```
(| :arg. |
   arg * 2.
)
```

Nótese como la sintaxis es igual a la de los data objects en donde el argumento es un slot mas solo identificado por "`:`" al principio de su nombre (y no puede estar inicializado como los slots normales).

Y si queremos crear un objeto punto que tenga un método que retorna su norma al cuadrado?

```
(|
   x <- 1.
   y <- 2.
   square_norm = (| | ((x * x) + (y * y)).).
|)
```

Nótese como el objeto punto tiene 3 slots: `x, y, square_norm` donde los primeros 2 son mutables e inicializados con los números 1 y 2 mientras que el tercer slot es inmutable inicializado con otro objeto. Este segundo objeto no tiene ningún slot propio ni recibe ningún parámetro y tiene código (es un method object).

Parent slots

Un tipo especial de slot son los llamados *parent slots* o prototipos. Funcionan igual que cualquier otro slot solo que además intervienen en la cadena de lookup y son indicados con un asterisco a continuación de su nombre.

Por ejemplo:

```
obj = (| padre* = punto. x = 1. |).
(obj x) print.
(obj y) print.
```

En el primer `print`, el objeto tiene un slot x, así que el primer print imprimirá “1”.
En siguiente `print`, el objeto no tiene un slot y así que el slot y se buscará en su (o sus) parents o prototipos. En otras palabras se enviará el mensaje `obj padre y`.

Lookup de un slot

Veamos en detalle cómo se hace un lookup de un slot:

```
obj x.
```

Primero, el objeto que recibe el mensaje es obj y el slot a buscar es x.
El lookup consiste en buscar en obj el slot x y de no encontrarse se buscara en los *parent slots* de forma recursiva con tres posibles resultados:

- se encontró un único slot x en alguno de los prototipos, ese es el slot a retornar
- se encontraron más de un slot x, emitir un error
- no se encontro ningun slot x, emitir un error.

Para simplificar no se implementara ningún esquema de excepciones. Emitir un error implicara imprimir por consola que hubo un error (o un mensaje gráfico en la interfaz grafica) y acto seguido se retornará un `nil`.

Veamos ahora otro ejemplo en donde tenemos definido el siguiente objeto punto:

```
(|
   x <- 1.
   y <- 2.
   square_norm = (|| ((x * x) + (y * y)).).
|).
```

Entonces si ahora se quiere enviar el mensaje `square_norm`:

```
punto square_norm.
```

El lookup de `square_norm` es trivial como lo vimos en el ejemplo anterior y retorna un slot que tiene código asociado el cual se ejecuta en ese momento.
El código a ejecutar es:

```
(|| ((x * x) + (y * y)).).
```

Para ejecutar el método el objeto `(|| ((x * x) + (y * y)).)` **se clona** de tal manera que si tiene slots propios estos también se clonan y **funcionan como variables locales** del método.
Luego se ejecuta el código interno: el lookup de x comienza por buscar un slot local (como si fuera una variable local de C) en los slots del método.
En este caso no hay ninguna slot local ni argumento.
El lookup de x continúa en el objeto **self**, un parent slot implícito que todo objeto tiene y apunta al objeto que inició el método, en nuestro caso "punto" y viene a ser el equivalente al **this** de C++ (salvando las distancias)
En otras palabras, x se transforma en `self x` o equivalentemente en `punto x`.
A partir de ahí, el lookup continúa de forma recursiva hasta encontrar un único slot x o emitir un error.
Nótese que haber usado x no es lo mismo que haber usado `self x`: en el primer caso el lookup busca el slot x primero en los slots del método (variables y argumentos del método) y luego en los slots del objeto `self`.

Lo mismo sucede con el slot `y`, así como con el resto de los mensajes.

Sintaxis BNF

A continuación se define la sintaxis de nuestra versión de Self usando la notación BNF. Como recordatorio, las reglas se leen de izquierda a derecha donde:
`A B` significa que se debe parsear `A` y luego `B`.
`A | B` significa que la regla a parsear puede ser `A` o `B` (primero intentar con `A` y si falla, luego con `B`)
`{ A }` significa que la regla a parsear `A` puede aparecer 0 o mas veces
`'A'` significa que se debe parsear el literal `A`
`A | vacio` significa que A es opcional, puede aparecer o no.

En todos los casos debe considerarse que puede haber un cero o más espacios vacíos o saltos de línea entre cada regla.

```
-script := { expression '.' }
-expression := keyword_message | binary_message |
               unary_message | expressionCP
-expressionCP := expressionP | constant
-expressionP := '(' expression ')'


-keyword_message := receiver lower_keyword ':'
                                        expressionCP { cap_keyword ':' expressionCP }
-binary_message := receiver operator expressionCP
-unary_message := receiver name
```

```
-receiver := expressionCP | vacio

-object  := ‘(|’ slot_list ‘|’  script ‘)’
-slot_list :=  { slot_name_extended [ ( ‘=’ | ‘<-’ )  expression ]  ‘.’ }
-slot_name_extended := ( ‘:’ name | name ‘*’ | name )

-constant := number | string | object | ‘nil’ | name

-operator := ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘!=’ | ‘==’

-name := cualquier palabra alfanumérica que empiece con una letra en minúsculas
-lower_keyword := cualquier palabra alfanumérica que empiece con una letra en
minúsculas o un guión bajo
-cap_keyword := cualquier palabra alfanumérica que empiece con una letra en mayúsculas
-number := un número de uno o varios dígitos tanto números enteros como flotantes,
positivos y negativos.
Tantos los nombres como los números no tendrán mas de 256 caracteres.
```

Objetos estándar

Los siguientes objetos deben ser implementados de forma nativa:

- los números (1, 3, 3.68, -22) y deben soportar los métodos `+, -, *, /, ==, !=`
- los booleanos (`true, false`)
- los strings de la forma `'hola-mundo'`. Por simplicidad se garantiza que los strings no contendrán comillas ni espacios.
- el `lobby`, es el objeto padre global y representa el lugar donde viven el resto de los objetos. Siempre existe y no puede ser borrado. Tiene un método `collect` que ejecuta el garbage collector.

Todos estos objetos deben poderse imprimir con el método `print` por consola y retorna el objeto `self` y deben soportar los métodos `_AddSlots:` y `_RemoveSlots:` para agregar y remover slots.
Ademas deben tener un slot mutable adicional llamado `_Name` inicializado con el string **“**`object`**”** que servirá para ponerle un nombre a los objetos.

Todos los objetos también deberán poderse clonar con el método `clone`.

Garbage collector

La máquina virtual de Self deberá implementar un mecanismo de liberación de recursos.
Dado que todos los objetos en Self son referenciados de forma directa o indirectamente por el objeto `lobby` es fácil saber qué objetos no están siendo referenciados y por lo tanto pueden ser recolectados.

Un algoritmo *posible* sería recorrer de forma recursiva desde el objeto lobby todos los objetos que son referenciados por él e ir marcándolos. Al concluir sólo los objetos que no pueden accederse quedarán sin marcar y por lo tanto pueden ser liberados.
En la segunda fase todos los objetos son visitados y aquellos que no fueron marcados en la fase anterior son borrados.

Ejemplo completo

Veamos ahora como todas las piezas se unen. Crearemos un prototipo de punto, luego 2 puntos y finalmente haremos algunas impresiones.
Como queremos crear nuestros objetos queden guardados los asignaremos como slots del objeto global `lobby`:

```
lobby _AddSlots: (| punto = (|
                              square_norm = (| | ((x * x) + (y * y)).).
                              print = (| |
                                          '(' print.
                                          x print.
                                          ';' print.
                                          y print.
                                          ')' print.
                                       ).
                              * = (| :scalar. temp. |
                                        temp: self clone.
                                        temp x: ((temp x) * scalar).
                                        temp y: ((temp y) * scalar).
                                        temp.
                                    )
                              addx:Addy: = (| :x2. :y2 |
                                              self x: ((self x) + x2).
                                              self y: ((self y) + y2).
                                              self.
                                            ).
                           |)
                  |).

lobby _AddSlots: (| punto1 = (| x <- 0. y <- 1. proto* = punto |).
                    punto2 = (| x <- 1. y <- 0. proto* = punto |).
                  |)


(punto1 * 2) print.
punto1 print.
punto2 addx: 3 Addy: 6.
punto2 print.

lobby collect.
```

Por consola debería imprimirse:

```
(0;2)
(0;1)
(4;6)
```

Y luego de ejecutarse el `collect`, solo los objetos `lobby`, `punto1` y `punto2` seguirán vivos. Los objetos temporales creados como `(punto1 * 2)` son destruidos.

Morphic

Self tiene un entorno de trabajo gráfico en donde los objetos pueden tener su propia representación conocida como *morph* y nos basaremos en la implementación original del entorno de Self.

Un morph puede tener cualquier forma pero nos limitaremos al morph básico que nos permitirá interactuar con los objetos de Self.

Un morph debe:

- mostrar el nombre del objeto
- mostrar sus slots, por cada slot:
    - el nombre del slot
    - su valor si es un int, boolean, nil o bien el nombre del objeto al que apunta
    - un botón que permite, haciéndole click crear un morph que mostrará el valor del slot de forma separada con una línea recta uniendo a ambos.
      La recta no solo conecta al segundo objeto sino que también permite cambiar de objeto referenciado con un *drag and drop*.
- un espacio para editar el código asociado al objeto (si es que tiene)
- un espacio para escribir mensajes que se enviarán al objeto junto con 2 botones `Get` y `Do`.
  Ambos ejecutan el código sólo que el primero además crea un morph del objeto resultado.
- tener un botón para sacar el morph de la vista (*dismiss*).

Los slots y el nombre del objeto deben poder ser editados.
Además se debe poder agregar y remover slots.

Se deberá poder tener siempre acceso al morph que representa al objeto `lobby` de tal manera que uno pueda ir agregandole objetos a él.

Nótese que si existe un morph representando a algún objeto `X`, este no puede ser recolectado por el garbage collector porque el objeto morph tiene una referencia al objeto `X` y el `lobby` tiene una referencia a ese morph y por lo tanto ninguno puede ser recolectado.
Sacar o hacer un *dismiss* de un morph supone que el morph deja de existir y por lo tanto el objeto `X` puede ser recolectado.

Aplicaciones Requeridas

Dos aplicaciones son requeridas, una máquina virtual (servidor) y un entorno de desarrollo gráfico (cliente).
La máquina virtual podrá ejecutarse en dos modos: en uno de ellos podrá procesar un archivo con código Self, en el otro se comportará como un servidor que acepta conexiones desde los clientes.
El servidor al cerrarse debe persistir todos los `lobbys` y objetos creados en un archivo para que al levantarse pueda recrearlos y los programadores no hayan perdido sus trabajos.

Los clientes gráficos se conectaran al servidor y podrán crear un mundo o `lobby` nuevo en donde programar o unirse a un `lobby` ya creado y programar en simultáneo con otros clientes sobre un mismo `lobby` usando

la interfaz Morphic.
El programador podrá crear un `lobby` nuevo o bien crearlo copiandose de un `lobby` preexistente.
El programador podrá cambiar de `lobby` en cualquier momento que lo desee pero siempre mostrará un `lobby` a la vez sea este privado o compartido.
Ademas el programador podrá copiar un objeto de un `lobby` a otro `lobby.`

Como una analogía un mundo Self "privado" es como un blog personal mientras que un mundo "compartido" es como una página de la wikipedia o un Google doc.

Cuando la interfaz esté mostrando un mundo compartido deberá mostrar cómo los objetos se mueven y son editados por los otros programadores asi como también quiénes están interactuando.

Distribución de Tareas Propuesta

Con el objetivo de organizar el desarrollo de las tareas y distribuir la carga de trabajo, es necesario planificar las actividades y sus responsables durante la ejecución del proyecto. La siguiente tabla plantea una posible división de tareas de alto nivel que puede ser tomada como punto de partida para la planificación final del trabajo:

	Alumno 1 **Parser - Serialización**	**Alumno 2** **Máquina Virtual - Servidor**	**Alumno 3** **Cliente Gráfico**
Semana 1 (04/10/2016)	Parser de expresiones simples como: “`hola`” `print.`	Máquina virtual básica que permite crear objetos y ejecutar mensajes simples como: “`hola`” `print.`	Cliente gráfico básico con algunas primitivas para dibujar un morph.
Semana 2 (11/10/2016)	Parser de expresiones complejas como los objetos y el envío de mensajes.	Máquina virtual que permite crear objetos con sus slots.	Morphs gráficos más completos con el dibujado de sus slots y una edición básica.
Semana 3 (18/10/2016)	Parser completo.	Máquina virtual completa. Servidor basico	Morphs completos. Drag and Drop para el referenciado de objetos. Integración con el servidor.
Semana 4 (25/10/2016)	Serialización de los objetos.	Servidor completo con multiples lobbys.	Creación y unión a un lobby. Múltiples programadores interactuando en un mismo lobby.
Semana 5 (01/11/2016)	Garbage Collector	Servidor y Máquina virtual completa.	Cliente completo.
Semana 6 (08/11/2016)	- Correcciones y *tuning* del Servidor - Documentación	- Correcciones y *tuning* del Servidor - Documentación	- Correcciones y *tuning* del Cliente - Documentación

Preentrega			
Semana 7 (15/11/2016)	- Correcciones sobre Preentrega - Testing y corrección de bugs - Documentación	- Correcciones sobre Preentrega - Testing y corrección de bugs - Documentación	- Correcciones sobre Preentrega - Testing y corrección de bugs - Documentación
Semana 8 (22/11/2016) **Entrega**	- Testing - Documentación - Armado del entregable	- Testing - Documentación - Armado del entregable	- Testing - Documentación - Armado del entregable

Restricciones

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

1. El sistema se debe realizar en ISO C++ 11 utilizando librerías *gtkmm* u otras acordadas con la cátedra.
2. Con el objetivo de facilitar el desarrollo de las interfaces de usuario, se permite el uso de Glade.
3. Es condición necesaria para la aprobación del trabajo práctico la entrega de la documentación mínima exigida (consultar sitio de la cátedra). Es importante recordar que cualquier elemento faltante o de dudosa calidad pone en riesgo la aprobación del ejercicio.

Referencias

[1] http://handbook.selflanguage.org/ en particular la seccion Language Reference
[2] https://www.gnu.org/software/bison/ Bison es un generador de parsers que puede ser usado en el trabajo aunque la sintaxis de nuestra version de Self fue simplificada para que pueda implementarse con las herramientas de C++ estandar con facilidad.
[3] http://www.cplusplus.com/reference en particular los metodos `std::istream::get`, `std::istream::unget`, `std::istream::tellg` y `std::istream::seekg` pueden ser de interes a la hora de hacer el parser.