PARSEO Y GENERACIÓN DE CÓDIGO – 2DO SEMESTRE 2017

LICENCIATURA EN INFORMÁTICA CON ORIENTACIÓN EN DESARROLLO DE SOFTWARE

UNIVERSIDAD NACIONAL DE QUILMES

Trabajo práctico 2

Compilador para el lenguaje **Remolacha**

Fecha de entrega: 23 de noviembre

Índice

1. Introducción

Remolacha es un lenguaje minimalista con algunas características básicas de orientación a objetos. Este TP consiste en implementar el generador de código para compilar el lenguaje Remolacha al lenguaje C++[1]. Para analizar sintácticamente el lenguaje Remolacha, contamos con una gramática de Remolacha ya escrita en Lleca[2], que se les proveerá en el archivo `remolacha.ll`.

2. Sintaxis de Remolacha

2.1. Sintaxis concreta (descripción informal)

Un programa Remolacha es una lista de declaraciones de clases. Cada declaración de clase puede incluir una lista de variables locales (variables de instancia). Además, cada clase tiene una lista de definiciones de métodos. Un método tiene un nombre, que representa el mensaje que responde, y una lista de parámetros. El cuerpo de un método está dado por una lista de expresiones. Las expresiones incluyen: variables, constantes, asignaciones, envíos de mensajes, referencia al objeto actual (`self`), y creación de nuevos objetos (`new`). Por ejemplo:

[1] Los compiladores que traducen un lenguaje de alto nivel a otro lenguaje de alto nivel haciendo transformaciones meramente locales o *cosméticas* en la estructura del código a veces se conocen como *transpiladores*. En el caso de este TP preferimos evitar esta terminología.

[2] Generador de parsers desarrollado para el TP 1.

```
class Contador
  local valor

  def inicializar(valorInicial)
    set valor = valorInicial
    self

  def incrementar()
    self.incrementarEn(1)

  def incrementarEn(x)
    set valor = valor.add(x)

  def valorActual()
    valor

class Main

  local c

  def main()
    set c = new Contador.inicializar(0)
    c.incrementar().valorActual().print()       /* 1 */
    c.incrementar().valorActual().print()       /* 2 */
    c.incrementarEn(10).valorActual().print()   /* 12 */
```

Notas.

- Las variables y métodos son todos de instancia: no hay variables ni métodos `static`.
- Las clases son primitivas (no son objetos).
- Los tipos de datos básicos (números y cadenas) sí son objetos.
- No hay mecanismos primitivos de subclasificación, herencia, *mixins*, etc.
- No hay estructuras de control (`if`, `while`, `for`, ...).
- No hay bloques ni clausuras.
- No se espera que se implemente el reclamo automático de memoria (*garbage collection*). Este es un tema complejo que excede el alcance de este TP.
- El pasaje de mensajes es siempre sincrónico y la comunicación entre objetos usa el mecanismo *call/return*.

La sintaxis concreta formal de Remolacha está dada en el archivo `remolacha.ll`. En la siguiente sección se describe la sintaxis abstracta, con la que trabajarán en este TP.

2.2. Árbol de sintaxis abstracta

El árbol de sintaxis abstracta es un término como el que se construyó para el TP 1. Un programa es una lista de definiciones de clases:

$$\begin{array}{rcl} \textit{programa} & ::= & \texttt{Nil} \\ & | & \texttt{Cons}(\textit{clase}, \textit{programa}) \end{array}$$

La definición de una clase tiene asociados: el nombre de la clase (un identificador), una lista de definiciones locales (variables de instancia) para esa clase, y una lista de definiciones de métodos para esa clase.

$$\textit{clase} \quad ::= \quad \texttt{DefClass}(\texttt{ID}, \textit{locales}, \textit{metodos})$$

Las definiciones locales están dadas por una lista de pares que le asocian un valor inicial a cada variable local:

$$\begin{array}{rcl} \textit{locales} & ::= & \texttt{Nil} \\ & | & \texttt{Cons}(\texttt{Local}(\texttt{ID}, \textit{expresion}), \textit{locales}) \end{array}$$

Las definiciones de métodos para una clase están dadas por una lista de definiciones de métodos:

$$\begin{array}{rcl} \textit{metodos} & ::= & \texttt{Nil} \\ & | & \texttt{Cons}(\textit{metodo}, \textit{metodos}) \end{array}$$

La definición de un método tiene asociados: el nombre del mensaje que responde (un identificador), la lista de parámetros y un bloque que representa el cuerpo del método.

$$\textit{metodo} \quad ::= \quad \texttt{DefMethod}(\texttt{ID}, \textit{parametros}, \textit{bloque})$$

Los nombres de los parámetros están dados una lista de identificadores:

$$\begin{array}{rcl} \textit{parametros} & ::= & \texttt{Nil} \\ & | & \texttt{Cons}(\texttt{ID}, \textit{parametros}) \end{array}$$

Un bloque es una lista de expresiones. Observar que una expresión puede tener efectos secundarios.

$$\begin{array}{rcl} \textit{bloque} & ::= & \texttt{Nil} \\ & | & \texttt{Cons}(\textit{expresion}, \textit{bloque}) \end{array}$$

Una expresión puede ser alguna de las siguientes:

expresion	::=	`Variable(ID)`	lectura de variable o parámetro
	|	`Set(ID, expresion)`	asignación a variable o parámetro
	|	`ConstantNumber(NUM)`	constante numérica
	|	`ConstantString(STRING)`	constante de cadena
	|	`Self`	autorreferencia (palabra clave `self`)
	|	`Send(expresion, ID, argumentos)`	envío de mensaje
	|	`New(ID)`	construcción de un nuevo objeto

Las listas de argumentos en `Send` son listas de expresiones:

$$\begin{array}{rcl} \textit{argumentos} & ::= & \texttt{Nil} \\ & | & \texttt{Cons}(\textit{expresion}, \textit{argumentos}) \end{array}$$

3. Compilación

El compilador puede estar hecho en el lenguaje que quieran, pero debe generar código en C++. Asumiremos también que la arquitectura subyacente es de 64 bits, lo que no debería traer problemas. No obstante, como el objetivo del TP es entender el mecanismo mediante el cual podría funcionar un compilador más complejo, **no** está permitido usar la funcionalidad de clases, objetos y métodos ya disponible en C++. Nos limitaremos a usar estructuras de datos elementales como arreglos, registros (`structs`), punteros y valores nativos de C++ (números y cadenas).

3.1. Tipos de datos para representar clases, objetos y métodos

En esta sección se detallan los tipos de datos sugeridos para representar clases, objetos y métodos. Pueden cambiar los nombres o utilizar otra representación si les resulta conveniente, siempre que se base únicamente estructuras de datos elementales.

El compilador debe generar código para definir los tipos `Num` y `String` que representan valores nativos numéricos y cadenas:

```
typedef unsigned long long int Num;
typedef char* String;
```

El compilador también debe definir el siguiente tipo `PTR`, que representa un puntero a alguna posición de la memoria:

```
typedef void* PTR;
```

Una **clase** estará representada mediante un arreglo de métodos:

```
struct Clase {
    PTR* metodos;
};
```

Un **objeto** estará representado mediante un puntero a la clase del que es instancia, y un arreglo de punteros que están asociados a las variables de instancia de dicho objeto:

```
struct Objeto {
    Clase* clase;
    PTR* varsInstancia;
};
```

Un **método** es un puntero a una función[3] que recibe como parámetros una cierta cantidad de punteros a objetos (0, 1, 2, o más) y devuelve un puntero a un objeto.

```
typedef Objeto* (*Metodo)(...);
```

Además, el compilador necesitará reinterpretar el tipo `PTR` ("*castear*"), para lo cual usaremos los siguientes macros:

```
#define NUM_TO_PTR(N)    ((PTR)(N))     /* Convierte Num -> PTR */
#define PTR_TO_NUM(P)    ((Num)(P))     /* Convierte PTR -> Num */

#define STRING_TO_PTR(S) ((PTR)(S))     /* Convierte String -> PTR */
#define PTR_TO_STRING(P) ((String)(P))  /* Convierte PTR -> String */

#define METHOD_TO_PTR(M) ((PTR)(M))     /* Convierte Metodo -> PTR */
#define PTR_TO_METHOD(P) ((Metodo)(P))  /* Convierte PTR -> Metodo */

#define OBJECT_TO_PTR(O) ((PTR)(O))     /* Convierte Objeto* -> PTR */
#define PTR_TO_OBJECT(P) ((Objeto*)(P)) /* Convierte PTR -> Objeto* */
```

La precondición de los macros `PTR_TO_X(foo)` es que `foo` tenga almacenado un valor de tipo `X`. Por ejemplo, podríamos hacer algo como esto:

```
PTR caja1 = NUM_TO_PTR(42);
PTR caja2 = STRING_TO_PTR("hola");

Num n = PTR_TO_NUM(caja1);        /* Recupera el valor 42 */
String s = PTR_TO_STRING(caja2);  /* Recupera el valor "hola" */
```

Pero el siguiente código viola la precondición de `PTR_TO_NUM`:

```
PTR caja3 = STRING_TO_PTR("hola");
Num mal = PTR_TO_NUM(caja3);      /* Comportamiento indefinido */
```

A continuación se ilustra también cómo usarlo para trabajar con métodos:

```
Objeto* foo(Objeto* obj0, Objeto* obj1) {
    return obj0;
}

void main() {
    PTR caja = METHOD_TO_PTR(foo);
    PTR_TO_METHOD(caja)(NULL, NULL);
}
```

En la Sec. 4.1 se muestra el encabezado completo que deberían incluir todos los programas compilados.

[3] En C++, un puntero a una función es un puntero a la dirección de memoria en la que se encuentra ubicado el código nativo correspondiente a dicha función.

3.2. Selectores, mensajes y métodos

En Remolacha, los objetos se comunican enviándose mensajes. Un **selector** es un *nombre* que sirve para identificar cuál es el mensaje particular que se está enviando. En Remolacha, los selectores son de la forma `identificador/número` donde el número indica el número de parámetros que recibe el mensaje. Por ejemplo la clase `Reloj` definida abajo acepta los mensajes identificados por los selectores `empezar/0` y `empezar/2`. Esta versión de sobrecarga está permitida en Remolacha y no ocasiona ningún tipo de ambigüedad:

```
class Reloj
    local hh
    local mm

    def empezar()
      self.empezar(0, 0)

    def empezar(horas, minutos)
      set hh = horas
      set mm = minutos
```

Un **mensaje** es un selector acompañado de argumentos. Por ejemplo, en la siguiente línea se le envía al objeto `foo` el mensaje identificado por el selector `empezar/2` y acompañado de la lista de argumentos `(18, 30)`:

```
foo.empezar(18, 30)
```

Un **método** es la implementación de un mensaje. Por ejemplo, las clases `A` y `B` tienen, cada una, un método para responder el mensaje identificado por el selector `mostrar/0`.

```
class A
  def mostrar()
    "a".print()

class B
  def mostrar()
    "b".print()
```

La resolución de métodos al enviar un mensaje es **dinámica**. Por ejemplo, al momento de compilar el siguiente código, el compilador no tiene manera de determinar, en general, cuál es el método que se ejecutará como respuesta al envío del mensaje `empezar(18, 30)`. Esto depende de la clase del objeto `foo`, y en principio podría haber varias clases que respondan al mensaje identificado por el selector `empezar/2`.

```
foo.empezar(18, 30)
```

3.3. Etapas del compilador

3.3.1. Verificación de buena formación

La primera tarea del compilador es verificar que el programa se encuentre bien formado. Esta etapa es importante no solamente para proveer un reporte de errores adecuado al usuario, sino también para que el código generado sea correcto.

Por ejemplo, la expresión `new A` crea una nueva instancia de la clase `A`. El compilador debe asegurarse de que la clase `A` se encuentre definida. Si el compilador no realizara ese chequeo, podría generar código erróneo en C++, lo que es inaceptable.

A continuación se listan las condiciones que debe verificar el compilador. No es necesario que esta etapa se haga *por separado* (es decir *antes*) de las demás etapas del compilador. Los chequeos pueden realizarse a medida que se compila el programa. Sin embargo, es imprescindible que estos chequeos se realicen en algún momento.

1. Debe haber una clase `Main` y una implementación el método `main/0` en dicha clase.

2. No puede haber clases con nombre repetido. Por ejemplo, el siguiente programa se rechaza porque hay dos definiciones de la clase `A`:

   ```
   class A
   class A
   ```

3. Una misma clase no puede tener dos métodos distintos con el mismo selector. Por ejemplo, el siguiente programa se rechaza porque hay dos métodos que implementan el mensaje `foo/1` para la clase `A`:

```
class A
  def foo(x)
  def foo(y)
```

4. Una clase no puede tener nombres de variables de instancia repetidos. Por ejemplo, el siguiente programa se rechaza porque en la clase `A` se declara dos veces la variable de instancia `x`:

```
class A
  local x
  local x
```

5. Los nombres de los parámetros de un método no pueden estar repetidos. Por ejemplo, el siguiente programa se rechaza porque el método `foo/2` de la clase `A` repite el parámetro `x`:

```
class A
  def foo(x, x)
```

6. Los nombres de los parámetros no pueden coincidir con nombres de variables locales. Por ejemplo, el siguiente programa se rechaza porque `x` no puede ser al mismo tiempo el nombre de un parámetro y de una variable local:

```
class A
  local x
  def foo(x)
```

7. **Puede** haber nombres de variables en común entre distintas clases, y nombres de parámetros en común entre distintos métodos. Por ejemplo, el siguiente programa está bien formado:

```
class A
  local x
  def foo(y)
  def bar(y)
class Main
  local x
  def main()
```

8. Cada vez que se accede o se asigna una variable, el nombre de dicha variable debe ser el nombre de una variable de instancia o de un parámetro que se encuentren en *scope*. Por ejemplo, el siguiente programa se rechaza porque la asignación a la variable `x` no corresponde al nombre de una variable de instancia de la clase `Main` ni a un parámetro en *scope*.

```
class A
  local x
class Main
  def foo(x)
  def main()
    set x = 1
```

9. Cada vez que se hace un `new`, el identificador debe corresponder a una clase existente. Por ejemplo, el siguiente programa se rechaza porque `A` no es una clase definida:

```
class Main
  def main()
    new A
```

10. Cuando se envía un mensaje, el compilador **no** debe fallar en tiempo de compilación si el mensaje no existe. La falla debe postergarse hasta el momento de la ejecución. Por ejemplo, el siguiente programa está bien formado y falla solamente una vez que se lo ejecuta.

```
class A
class Main
  def main()
    new A.mensajeInexistente()
```

3.3.2. Recolección de clases y selectores

A continuación, el compilador debe recolectar los nombres de todas las clases y selectores que se utilizan a lo largo del programa. Por ejemplo, en el siguiente programa:

```
class Coordenada
  local x
  local y
  def init()
    self.init(0, 0)
  def init(xx, yy)
    set x = xx
    set y = yy
  def mostrar()
    "(".print() x.print() ",".print() y.print() ")".print()
class Main
  local c
  def main()
    set c = new Coordenada.init()
    c.mostrar()
    c.mensajeInexistente("foo", "bar")
```

Hay dos clases: `Coordenada` y `Main`, y cuatro selectores: `init/0`, `init/2`, `mostrar/0`, `print/0`, `main/0` y `mensajeInexistente/2`. El compilador debe armar una lista de todas las clases disponibles y una lista de todos los selectores disponibles, y asignarle un identificador único a cada uno.

Además, Remolacha tiene clases primitivas que aceptan mensajes. Por ejemplo, si suponemos que las clases primitivas son `Int` y `String`, y que los selectores de los mensajes que aceptan son `print/0` y `add/1`, la tabla podría quedar así:

Nombre en Remolacha	Identificador único
`Int`	`cls0`
`String`	`cls1`
`Coordenada`	`cls2`
`Main`	`cls3`
`print/0`	`sel0`
`add/1`	`sel1`
`init/0`	`sel2`
`init/2`	`sel3`
`mostrar/0`	`sel4`
`main/0`	`sel5`
`mensajeInexistente/2`	`sel6`

El compilador debe estar organizado de tal manera que sea razonablemente fácil agregar otras clases primitivas además de `Int` y `String`, y agregar otros mensajes a dichas clases además de `print/0` y `add/1`.

3.3.3. Enteros y cadenas

Los enteros y cadenas en Remolacha son objetos, que pertenecen a sus respectivas clases. Las clases primitivas se declaran así:

```
Clase* cls0;  /* Int */
Clase* cls1;  /* String */
```

Además, las clases primitivas tienen sus respectivos constructores. Cada constructor crea un objeto con una única variable de instancia, en la que se almacena el valor nativo de C++ asociado a ese objeto:

```
/* Construye un objeto de clase Int */
Objeto* constructor_cls0(Num valor) {
    Objeto* obj = new Objeto;
    obj->clase = cls0; /* Int */
    obj->varsInstancia = new PTR[1];
    obj->varsInstancia[0] = NUM_TO_PTR(valor);
    return obj;
}

/* Construye un objeto de clase String */
Objeto* constructor_cls1(String valor) {
    Objeto* obj = new Objeto;
    obj->clase = cls1; /* String */
    obj->varsInstancia = new PTR[1];
    obj->varsInstancia[0] = STRING_TO_PTR(valor);
    return obj;
}
```

3.3.4. Generación de código para las clases y constructores

Cada clase definida por el usuario se representa con un puntero a una estructura `Clase`. Por ejemplo, si las clases definidas en el programa son `cls2` y `cls3`, se debe generar el siguiente código para declarar dichos punteros:

```
Clase* cls2;  /* Coordenada */
Clase* cls3;  /* Main */
```

Cada clase tiene asociado un constructor. Los constructores de las clases definidas por el usuario no reciben ningún parámetro y devuelven un objeto nuevo de dicha clase. Al crear un objeto, se reserva espacio para guardar tantas variables de instancia como defina la clase dentro de sus declaraciones locales. Las variables de instancia se inicializan siempre para que su valor sea un objeto de clase `Int`, con valor 0.

```
/* Construye un objeto de la clase Coordenada */
Objeto* constructor_cls2() {
    Objeto* obj = new Objeto;
    obj->clase = cls2; /* Coordenada */

    /* Reserva espacio para dos variables de instancia: "x" e "y" */
    obj->varsInstancia = new PTR[2];
    obj->varsInstancia[0] = constructor_cls0(0); /* Se inicializa en 0 */
    obj->varsInstancia[1] = constructor_cls0(0); /* Se inicializa en 0 */
    return obj;
}
```

3.3.5. Generación de código para los métodos

Cada método en Remolacha se compila a una función en C++. El método de la clase `X` correspondiente al selector `Y` se compila a una función `Objeto* met_clsX_selY(Objeto* o0, Objeto* o1, ..., Objeto* oN)`. Por ejemplo, el método de la clase `Coordenada` correspondiente al mensaje `init/0` de la Sec. 3.3.2 se compila a una función:

```
/* cls2 => Coordenada , sel2 => init/0 */
Objeto* met_cls2_sel2(Objeto* o0) {
  /* ... */
}
```

Y el método para el mensaje `init/2` se compila a una función:

```
/* cls2 => Coordenada , sel3 => init/2 */
Objeto* met_cls2_sel3(Objeto* o0, Objeto* o1, Objeto* o2) {
  ...
}
```

Notar que un mensaje que recibe n parámetros se compila a una función que recibe $n + 1$ parámetros. El primer parámetro que recibe la función representa el objeto que recibe el mensaje (`self`).

3.3.6. Inicialización de las clases

La función "`void main()`" de C++ debe incluir código para inicializar todas y cada una de las clases, antes de comenzar a ejecutar el programa. Recordemos que una clase es un `struct` con un campo `PTR* metodos` que es un arreglo de punteros.

Los arreglos de métodos de **todas** las clases miden lo mismo. En particular, si hay n selectores distintos, el arreglo de métodos debe tener n entradas, una para cada selector. Si la clase `X` implementa el mensaje asociado al selector `Y`, entonces tendremos `clsX->metodos[Y] = METHOD_TO_PTR(met_clsX_selY)`. Si la clase `Y` no implementa el mensaje asociado al selector `Y`, entonces tendremos `clsX->metodos[Y] = NULL`.

Mostramos cómo podría ser un fragmento del código para inicializar las clases de acuerdo con la tabla de la Sec. 3.3.2:

```
void main() {
   /* Inicialización la clase cls0 (Int) */
     /* ... */
   /* Inicialización de la clase cls1 (String) */
     /* ... */
   /* Inicialización de la clase cls2 (Coordenada) */
     cls2 = new Clase;
     cls2->metodos = new PTR[7];  /* Los selectores son sel0, ..., sel6 */
     cls2->metodos[0] = NULL;                          /* print/0 */
     cls2->metodos[1] = NULL;                          /* add/1 */
     cls2->metodos[2] = METHOD_TO_PTR(met_cls2_sel2);  /* init/0 */
     cls2->metodos[3] = METHOD_TO_PTR(met_cls2_sel3);  /* init/2 */
     cls2->metodos[4] = METHOD_TO_PTR(met_cls2_sel4);  /* mostrar/0 */
     cls2->metodos[5] = NULL;                          /* main/0 */
     cls2->metodos[6] = NULL;                          /* mensajeInexistente/2 */
   /* Inicialización de la clase cls3 (Main) */
     /* ... */
   /* Ejecución del programa principal */
     /* ... */
}
```

3.3.7. Compilación de expresiones

Se describe a continuación cómo compilar todas las construcciones del lenguaje:

1. **Variables.** El acceso a una variable `x` puede corresponder a un parámetro o a una variable de instancia.

   Acceso a parámetro:

Remolacha	**Compilado**

`class Foo`
`  def bar(x, y)`
`    y`
	`Objeto* met_cls2_sel3(Objeto* o0, Objeto* o1, Objeto* o2) {`
	`    return o2;`
	`}`

   Acceso a variable de instancia:

Remolacha	Compilado

`class Foo`
`  local x`
`  local y`
`  local z`
`  def bar()`
`    z`
	`Objeto* met_cls2_sel3(Objeto* o0) {`
	`    return PTR_TO_OBJECT(o0->varsInstancia[2]);`
	`}`

2. **Asignación.** La asignación de variables es similar al acceso: puede tratarse de una asignación a un parámetro o a una variable de instancia. El valor de una asignación es el entero 0.

   Asignación a variable de instancia:

Remolacha	Compilado

`class Foo`
`  local x`
`  def bar(y)`
`    set x = y`
	`Objeto* met_cls2_sel3(Objeto* o0, Objeto *o1) {`
	`    o0->varsInstancia[0] = o1;`
	`    return constructor_cls0(0);`
	`}`

3. **Constante numérica.**

Remolacha	Compilado

`class Foo`
`  def bar()`
`    42`
	`Objeto* met_cls2_sel3(Objeto* o0) {`
	`    return constructor_cls0(42);`
	`}`

4. **Constante de cadena.**

Remolacha	Compilado

`class Foo`
`  def bar()`
`    "hola"`
	`Objeto* met_cls2_sel3(Objeto* o0) {`
	`    return constructor_cls1("hola");`
	`}`

5. **Autorreferencia (`self`).**

Remolacha	Compilado

`class Foo`
`  def bar()`
`    self`
	`Objeto* met_cls2_sel3(Objeto* o0) {`
	`    return o0;`
	`}`

6. **Envío de mensaje.**

Remolacha	Compilado

class Foo
  local x
  def bar(o)
  o.mandar(x, 42)
	Objeto* met_cls2_sel3(Objeto* o0, Objeto* o1) {
	  Objeto* tmp1 = o0->varsInstancia[0]; /* Acceso a "x" */
	  Objeto* tmp2 = constructor_cls0(42);
	
	  /* sel4 => mandar/2 */
	  if (o1->clase->metodos[4] == NULL) {
	  fprintf(
	  stderr,
	  "El objeto no acepta el mensaje 'mandar/2'.\n"
	  );
	  exit(1);
	  }
	  return PTR_TO_METHOD(o1->clase->metodos[4])(o1, tmp1, tmp2);
	}

7. **Creación de nuevo objeto.**

Remolacha	Compilado

class Foo
  def bar()
  new Foo
	Objeto* met_cls2_sel3(Objeto* o0) {
	  /* Constructor de Foo */
	  return constructor_cls2();
	}

4. Código auxiliar

4.1. Encabezado del programa compilado

```
#include <cstdlib>
#include <cstdio>

typedef unsigned long long int Num;
typedef char* String;
typedef void* PTR;

struct Clase {
    PTR* metodos;
};

struct Objeto {
    Clase* clase;
    PTR* varsInstancia;
};

typedef Objeto* (*Metodo)(...);

#define NUM_TO_PTR(N)    ((PTR)(N))
#define PTR_TO_NUM(P)    ((Num)(P))
```

```
#define STRING_TO_PTR(S) ((PTR)(S))
#define PTR_TO_STRING(P) ((String)(P))

#define METHOD_TO_PTR(M) ((PTR)(M))
#define PTR_TO_METHOD(P) ((Metodo)(P))

#define OBJECT_TO_PTR(O) ((PTR)(O))
#define PTR_TO_OBJECT(P) ((Objeto*)(P))
```

5. Pautas de entrega

Para entregar el TP se debe enviar el código fuente por e-mail a la casilla `foones@gmail.com` hasta las 23:59:59 del día estipulado para la entrega, incluyendo `[TP lds-est-parse]` en el asunto y el nombre de los integrantes del grupo en el cuerpo del e-mail. No es necesario hacer un informe sobre el TP, pero se espera que el código sea razonablemente legible. Se debe incluir un README indicando las dependencias y el mecanismo de ejecución recomendado para que el programa provea la funcionalidad pedida. Se recomienda probar el programa con el conjunto de tests provistos.