# Práctica 3: Árbol de sintaxis abstracta.

Compiladores 2017-1

Diana Olivia Montes Aguilar

October 7, 2016

## 1 Introducción

El analizador sintáctico que se tiene ya construido, únicamente valida que la cadena (el programa fuente) sea una de las que se pueden generar por la gramática del lenguaje. La presente práctica tiene como objetivo agregarle al reconocimiento sintáctico la generación del árbol de sintaxis abstracta (*AST Abstract Syntax Tree*). Nos basaremos en el patrón Compuesto para darle estructura al AST y en el despacho dinámico que ofrece Java para obtener el comportamiento específico de cada nodo.

### 1.1 Patrón Compuesto

Las partes del patrón compuesto son las siguientes:

1. *Componente.* Será nuestra clase Nodo. Define una interfaz para todos los elementos de la composición (el AST) e implementa un comportamiento por omisión.

2. *Compuesto.* Define el comportamiento específico para los nodos que tienen que manejar hijos.

3. *Hoja.* Representación de los elementos que no tienen hijos y definición del comportamiento específico para estos objetos.

4. *Cliente.* Manipula los objetos de la composición a través de la interfaz Componente.

El patrón compuesto ofrece una jerarquía de clases que van desde la más general hasta la más particular; entre más alejado de la raíz se encuentre la clase, tendrá un comportamiento más específico. Aunque es flexible para

implementar comportamiento específico a cada elemento, ofrece una interfaz al exterior, de tal modo que los objetos de las clases más particulares pueden tener el mismo comportamiento (los mismos métodos) que las otras clases.

Es importante recalcar que el patrón compuesto tiene muchas otras aplicaciones. Pero en esta práctica fue adaptado al *AST* y que con la palabra interfaz usada en el componente no hace referencia a una interfaz de Java (clase con todos los métodos abstractos, *interface*), sino a una plantilla de métodos que pueden ser aplicados a cualquier elemento de la composición. Sí es posible instanciar un objeto de la clase Nodo.

### 1.2 Despacho dinámico

Veamos el siguiente ejemplo:

```
class Nodo{
    public String getHijos(){
        return "";
    }
}
class Compuesto extends Nodo{
    public String getHijos(){
        return "Tengo n hijos";
    }
}
class Hoja extends Nodo{

}
class Prueba{
    public static void main(){
        Nodo h = new Hoja();
Nodo c = new Compuesto;
System.out.println(h.getHijos());
System.out.println(c.getHijos());
    }
}
```

Podemos observar en el ejemplo que tanto el objeto `c` y el objeto `h` en tiempo de compilación tienen tipo `Nodo`, pero al momento de ejecución, se hace el despacho dinámico, es decir, se resuelve el tipo en tiempo de ejecución y se determina que `h` es tipo `Hoja`, pero como no hay una implementación propia del método `getHijos()`, se ejecuta la que se definió por omisión en

la interfaz de la composición. En cambio para el nodo `c`, que se creó con el constructor de la clase `Compuesto`, sí hay una implementación específica.

### 1.3 Valores semánticos

En la versión actual del analizador sintáctico no hay manejo de tipos semánticos para símbolos terminales o no terminales. Dado que lo que nos interesa crear nodos e irlos relacionando unos con otros, necesitamos que los símbolos de la gramática tengan asociadas referencias a Nodos. Una manera de hacerlo, es provista por *byaccj*, mediante la opción de compilación *-Jsemantic=<SemanticValue>* que reemplaza el tipo *ParserVal* por lo que esté en *<SemanticValue>*. En nuestro caso, el tipo semántico que nos interesa es `Nodo`. Eso quiere decir que los valores que sean pasados por el analizador léxico en el atributo *yylval* del parser serán de tipo `Nodo`.

### 1.4 Recursividad izquierda y derecha

En la práctica 2, notamos que la recursividad derecha implica asociatividad derecha y la recursividad izquierda implica asociatividad izquierda, y en el momento de interpretar una expresión aritmética el resultado puede ser diferente en cada uno de los casos. También se observó que la pila crece más con la recursividad derecha porque no se hacen las reducciones hasta el final.

Observemos las siguientes reglas de la gramática en su versión para *byaccj* y con recursividad izquierda y derecha:

```
1  stmt : aux0

2  aux0 : simple_stmt
3       | aux0 simple_stmt

4  simple_stmt : small_stmt NEWLINE

5  small_stmt : expr_stmt
6             | print_stmt
```

Recursividad derecha:

```
1  stmt : aux0

2  aux0 : simple_stmt
3       | simple_stmt aux0
```

```
4  simple_stmt : small_stmt NEWLINE

5  small_stmt : expr_stmt
6             | print_stmt
```

Pensemos que el archivo fuente es

```
print x
print y
z = 87 + y
print 'cadena'
```

Cada una de esas líneas de código serán eventualmente reducidas a un `small_stmt`, (un nodo para cada una de ellas) para su reducción con el `simple_stmt` no va a ser necesario la creación de un nuevo nodo, ya que el átomo `NEWLINE` no será integrado al árbol por que no proporciona información necesaria para los siguiente análisis. La meta es que al hacer la reducción de la regla 1, se tenga la referencia de la raíz del árbol. La cuál será un nodo `Compuesto` (o alguna de las clases que hereden de `Compuesto`) con una lista de hijos y en ella vivan cada uno de los 4 nodos antes mencionados. La diferencia sutil entre la manera en la que se construye el árbol en la gramática con recursividad izquierda y la de derecha radica en el orden en el que se van agregando los hijos. En la recursividad izquierda los hijos se van agregando en el orden que se van encontrando ya que las reducciones se hacen inmediatamente, en cambio en la recursividad derecha, los hijos conforme se van reconociendo se meten a la pila y el último que se reconoció se agrega primero a la lista, entonces si no se agregan con atención a la lista, el orden en el que se escribió el código fuente será el inverso.

# 2 Descripción

## 2.1 Ejercicios para el laboratorio (2 pts)

1. Deberán construir las siguientes 3 clases:
   - `Nodo`. Clase general
   - `Hoja`. Hereda de Nodo.
   - `HojaEntera`. Hereda de Hoja. Deberá de tener un atributo e implementar un método que dé acceso a ese atributo.

2. La estructura anterior de clases debe hacerse siguiendo el patrón compuesto, eso quiere decir que `Nodo` tiene los métodos y atributos de las clases específicas y para los métodos brinda un comportamiento por omisión.

3. Integrar al analizador léxico y sintáctico el código necesario para que se pueda reconocer un nodo `HojaEntera` y obtener su referencia al final del reconocimiento como raíz del AST más simple, que es únicamente un nodo. Se deberá imprimir el valor del nodo.

### 2.2 Ejercicios para llevar (9 pts)

1. Se deberá implementar una clase Nodo en la que se defina el un comportamiento(métodos) por omisión para todos los nodos. Los métodos y atributos que estarán en `Nodo`, son todos los métodos que las clases más particulares necesitan. Ejemplo, para el caso de los compuestos: `getHijoIzq()`, `getHijoDer()`, `setHijoDer()`... y para el caso de las hojas `getValor()`, `setValor()`...

2. Deberán implementar una clase para cada nodo específico, siguiendo la estructura del patrón compuesto. Recuerden que en algún punto generaremos código a partir de un recorrido del AST. Pero también recuerden que lo que se pide es un AST, es decir, un Árbol que guarde sólo la información necesaria de la derivación.

3. Se deberá integrar a las reglas del analizador sintáctico el código necesario para poder formar el AST, de tal modo que al terminar el reconocimiento se tenga la referencia a la raíz del AST.

4. Pueden crear todas las clases que juzguen necesarias, tanto para alojar a los hijos, como para alojar valores o cualquier otro detalle de diseño.

5. Deben implementar un método para imprimir de forma legible el árbol que están generando.

### 2.3 Administrativos(1 pto)

1. La práctica se entrega antes del sábado 22 de octubre.

2. El código deberá estar comentado.

3. Deberá tener un `makefile` con las siguientes etiquetas:

- compile: compilará todos los archivos que sean necesarios para la ejecución.
- test: probará el analizador sintáctico con el archivo `file`. Eso quiere decir que `file` es una bandera para `make`.
- clean: borrará todos los archivos generados en las dos etiquetas anteriores.