

Análisis Semántico

Tabla de Símbolos y Árbol de Sintaxis Abstracto MiniJava

Compiladores e Intérpretes 2014

Introducción

- ¿Qué realizamos en el análisis semántico?
- Control de Declaraciones/Definiciones
- Chequeo de Tipos
- Resolución de Nombres

Introducción - MiniJava

- En general, la semántica de MiniJava es la de Java
- Aun así hay algunas diferencias, por ejemplo:
 - Una clase **no** puede tener dos **métodos** con el **mismo nombre**
 - No se puede definir más de **un constructor** por clase
 - Si un método es **sobre-escrito** toda su **signatura** debe **coincidir** con la de su **ancestro**.

Para más detalles ver el apunte de Semántica del lenguaje en la página web de la materia

Introducción - Motivación

- ¿Cómo construimos el Analizador Semántico?
- Una **possible estrategia** es diseñar **todos** los controles semánticos en una **EDT** para la **gramática** en la que basamos el **Analizador Sintáctico**
 - en **cada producción** se hacen todos los **controles semánticos** relacionados a esa producción
 - Luego, se **implementan todas** las acciones semánticas sobre el Analizador Sintáctico

Compilador de 1 Pasada

Introducción - Motivación

- ¿Es posible realizar **todos los controles** semánticos en **MiniJava** siguiendo esta estrategia?

NO

- En MiniJava casi ningún control semántico puede realizarse por completo de esta manera porque:
 - **Se permiten referencias hacia adelante**
- Seguiremos una estrategia de **2 Pasadas**

Introducción - Motivación

- Como parte del Analizador Sintáctico, vamos a **crear las estructuras** necesarias para realizar los chequeos semánticos
- El módulo de **Análisis Semántico** toma las estructuras y realiza todos los chequeos



Estructuras de Datos para el Análisis Semántico

Tabla de Símbolos

Tabla de Símbolos - Estructura

- La tabla de símbolos es la **estructura** central para el análisis semántico
- Mantiene información de **todas las entidades declaradas** (clases, métodos, variables, etc.)
- Cada **entidad** estará asociada a una **entrada** en la tabla
- Una **entrada** tendrá toda la **información de la entidad** que representa, tanto para el análisis semántico como para la generación de código

Tabla de Símbolos - Estructura

- ¿Cómo **estructuramos** las entradas en la TS para **facilitar** los controles semánticos?
- Tener en cuenta que, **por ejemplo**:

Ciertos nombres sólo son **visibles/accesibles** en **ciertos contextos**

```
class A{
    var int X;
    dynamic void m1(){ ... }
}
class B{
    dynamic void m1(){ ... }
}
```

Es posible referenciar directamente a la variable de instancia X

No es posible referenciar directamente a la variable de instancia X

Tabla de Símbolos - Estructura

- ¿Cómo **estructuramos** las entradas en la TS para **facilitar** los controles semánticos?
- Tener en cuenta que, **por ejemplo**:

Ciertos nombres sólo son **visibles/accesibles** en **ciertos contextos**

Un nombre puede representar **entidades diferentes** en **diferentes contextos**.

```
class A{
    var int X;
    dynamic void m1(){ ... }
    dynamic void m2(int X){ ... }
}
```

En este método el nombre X corresponde a la **variable de instancia**

En este método el nombre X corresponde al **parámetro**

Tabla de Símbolos - Estructura

- En general, debemos estructurar la tabla de símbolos de manera tal que en un **contexto**, como el cuerpo de un método, sea claro a qué **nombres** se puede **referenciar**.
- Buscaremos que la estructura de la tabla respete lo mejor posible los **ambientes** de declaración determinados por la **sintaxis de Minijava**

Tabla de Símbolos - Estructura

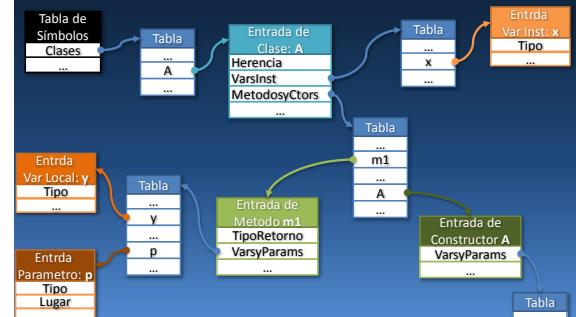


Tabla de Símbolos Implementación

Tabla de Símbolos - Implementación

- Cada tipo de **entrada** (clase, método, variable, etc.) se puede **representar** mediante una **clase**
- Las **variables de instancia** de la clase representan los diferentes **atributos** que caracterizan a la **entidad** asociada a la entrada.



Tabla de Símbolos - Implementación

- Además, se deben proveer **métodos** para **consultar/administrar** la tabla de símbolos y las distintas **entradas**.
- Por ejemplo:
 - La **tabla de símbolos** debería tener un **método** para ver si cierto **nombre** es una **clase declarada** y devolver su **entrada**
 - Una **entrada de clase** debería tener un **método** para ver si determinado **nombre** es un **método suyo**, y devolver la correspondiente **entrada de método**.

Tabla de Símbolos - Implementación

Una consideración importante con respecto al diseño de la TS es cómo se representarán los tipos

EntradaMetodo
Tipo TipoRetorno
Hash<VariableM> VarsYParams
...

Tipo Primitivo VS Tipo Clase

Son **diferentes semánticamente** y por lo tanto deben modelarse por **separado** (los tipos primitivos **NO** son clases)

Por otra parte es importante **modelarlos de manera uniforme** para diseñar adecuadamente las **entidades que los utilizan** (métodos y variables)

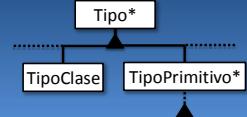


Tabla de Símbolos – Creando la TS

- La **tabla de símbolos** se creará a medida que se va **realizando el Análisis Sintáctico**.
- Haremos una **EDT** sobre la gramática utilizada en el Analizador Sintáctico, donde las **acciones semánticas construirán** la TS
- Luego, modificaremos los **métodos** del **Analizador Sintáctico** para reflejar las **acciones semánticas** de la EDT

Tabla de Símbolos – Creando la TS

- Consideremos la regla:

<ArgFormal> → <Tipo> identificador

EDT

```
<ArgFormal> → <Tipo> identificador {
  <ArgFormal>.Var =
    new EntradaParametro( <Tipo>.tipo,
                          identificador.lex);
}
```

<ArgFormal> usa el **atributo sintetizado Var** para devolver la entrada del parámetro reconocido

<Tipo> usa el **atributo sintetizado tipo** para devolver el tipo reconocido

identificador utiliza el **atributo intrínseco lex** para dar el lexema asociado al identificador

Tabla de Símbolos – Creando la TS

- Consideremos la regla:

EDT	$\langle \text{ArgFormal} \rangle \rightarrow \langle \text{Tipo} \rangle \text{ identificador}$	Código Sintáctico
<pre><ArgFormal> -> <Tipo> identificador { <ArgFormal>.Var = new EntradaParametro(<Tipo>.tipo, identificador.lex); }</pre>	<pre>void ArgFormal(){ Tipo(); match('id'); }</pre>	

Tabla de Símbolos – Creando la TS

- Consideremos | Devuelve la entrada del parámetro reconocido

EDT	$\langle \text{ArgFormal} \rangle \rightarrow \langle \text{Tipo} \rangle \text{ identificador}$	Código Sintáctico
<pre><ArgFormal> -> <Tipo> identificador { <ArgFormal>.Var = new EntradaParametro(<Tipo>.tipo, identificador.lex); }</pre>	<pre>EntradaParametro ArgFormal(){ Tipo(); match('id'); }</pre>	

Tabla de Símbolos – Creando la TS

- Consideremos | Devuelve el tipo reconocido

EDT	$\langle \text{ArgFormal} \rangle \rightarrow \langle \text{Tipo} \rangle \text{ identificador}$	Código Sintáctico
<pre><ArgFormal> -> <Tipo> identificador { <ArgFormal>.Var = new EntradaParametro(<Tipo>.tipo, identificador.lex); }</pre>	<pre>EntradaParametro ArgFormal(){ Tipo T = Tipo(); match('id'); }</pre>	

Tabla de Símbolos – Creando la TS

- Consideremos la regla:

EDT	$\langle \text{ArgFormal} \rangle \rightarrow \langle \text{Tipo} \rangle \text{ identificador}$	Código Sintáctico
<pre><ArgFormal> -> <Tipo> identificador { <ArgFormal>.Var = new EntradaParametro(<Tipo>.tipo, identificador.lex); }</pre>	<pre>EntradaParametro ArgFormal(){ Tipo T = Tipo(); String Nom = Lookahead.lex(); match('id'); }</pre>	

Tabla de Símbolos – Creando la TS

- Consideremos la regla:

EDT	$\langle \text{ArgFormal} \rangle \rightarrow \langle \text{Tipo} \rangle \text{ identificador}$	Código Sintáctico
<pre><ArgFormal> -> <Tipo> identificador { <ArgFormal>.Var = new EntradaParametro(<Tipo>.tipo, identificador.lex); }</pre>	<pre>EntradaParametro ArgFormal(){ Tipo T = Tipo(); String Nom = Lookahead.lex(); match('id'); return new EntradaParametro(T,Nom); }</pre>	<p>Se construye la entrada y se retorna</p>

Tabla de Símbolos

- Con esta estructura y los métodos adecuados el analizador semántico podrá realizar el **control de declaraciones**.
- El **cuerpo de los métodos** lo representaremos mediante **Árboles Sintácticos Abstractos**

Estructuras de Datos para el Análisis Semántico Árbol Sintáctico Abstracto (AST)

Árbol Sintáctico Abstracto (AST)

El **AST** es una forma arbórea de **representar** la **estructura sintáctica** de un programa.

- Nosotros utilizaremos **AST** sólo para representar el **cuerpo** de los **métodos y constructores**, y se generarán durante el análisis sintáctico.
- Luego, el **analizador semántico** recorrerá los AST y realizará los **chequeos** correspondientes al código.

Árbol Sintáctico Abstracto (AST) – Estructura

- Dado que los **AST** van a representar el código vamos a tener **varios tipos de nodos** en el árbol
- Por ejemplo, tipos de nodo para:
 - Bloques**,
 - Sentencias** (asignación, while, if, return, etc.)
 - Expresiones** (operadores binarios y unarios)
 - Primario** (variables, llamadas, literales, etc.)

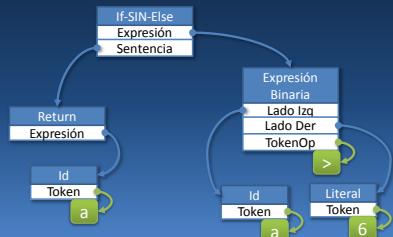
Árbol Sintáctico Abstracto (AST) – Estructura – Ejemplo

$a = 4 * (b+2);$



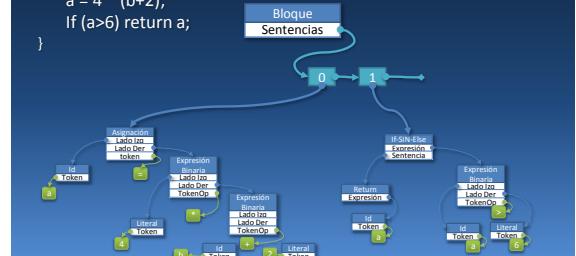
Árbol Sintáctico Abstracto (AST) – Estructura – Ejemplo

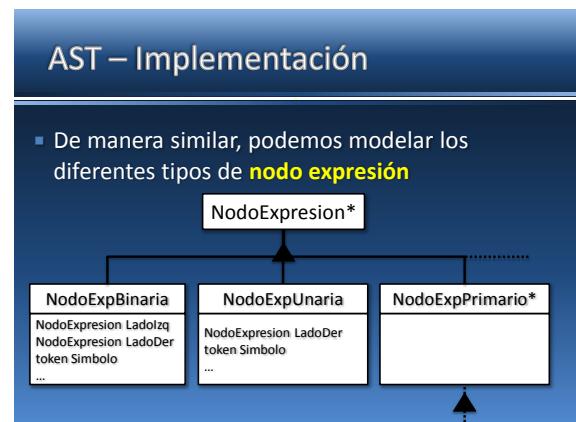
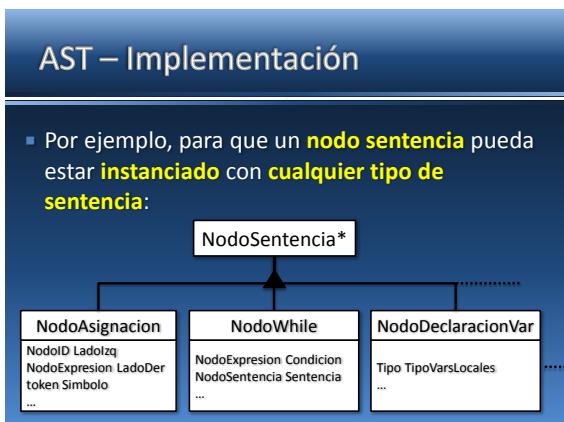
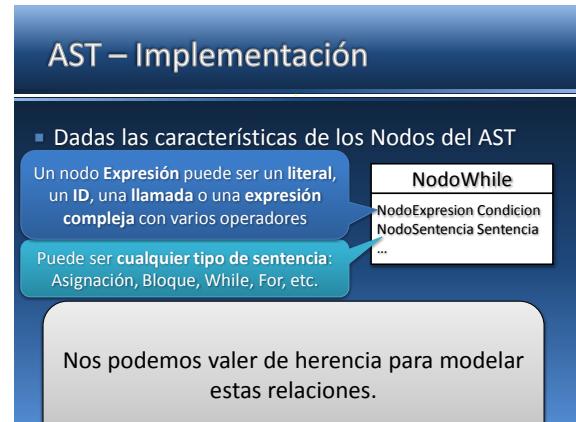
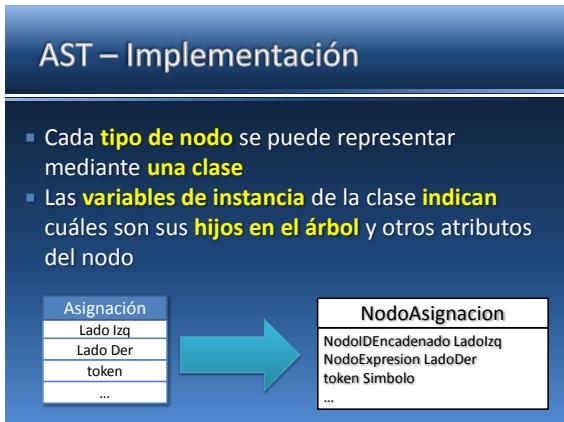
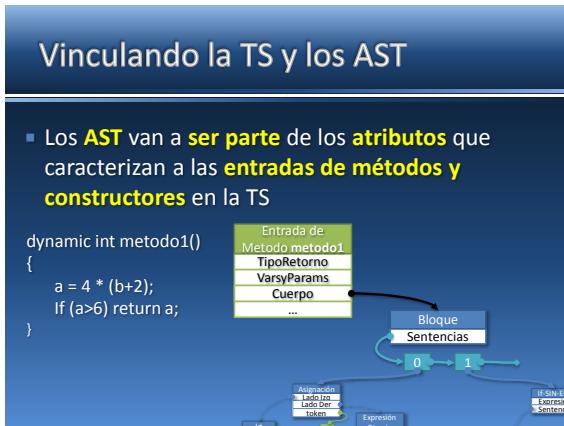
$a = 4 * (b+2);$
If ($a > 6$) return a;

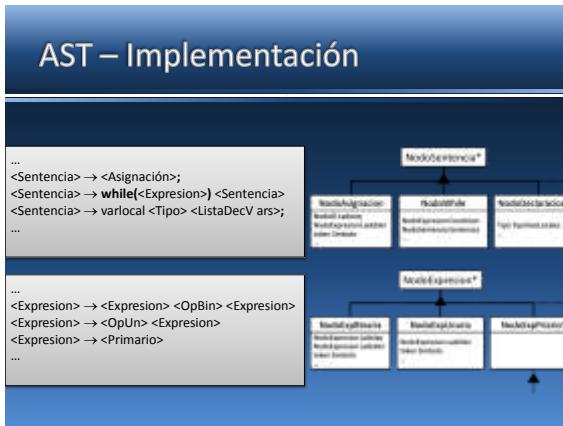


Árbol Sintáctico Abstracto (AST) – Estructura – Ejemplo

```
dynamic int metodo1()
{
    a = 4 * (b+2);
    If (a>6) return a;
}
```

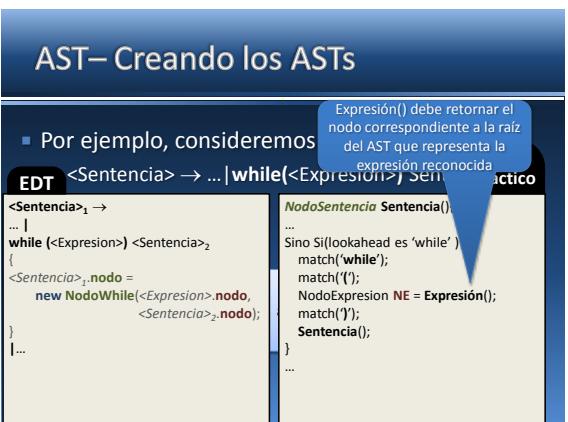
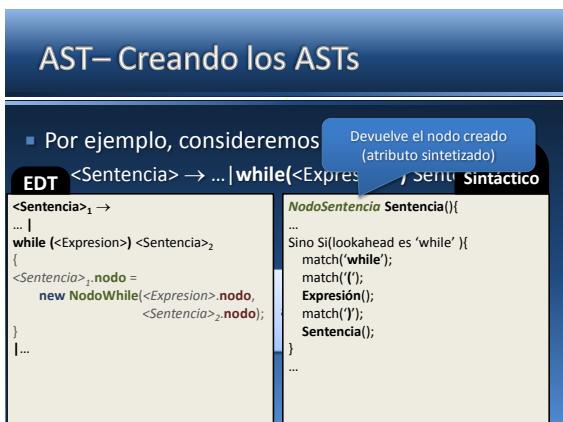
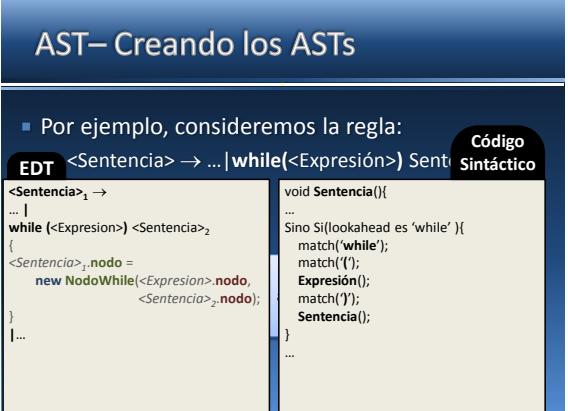
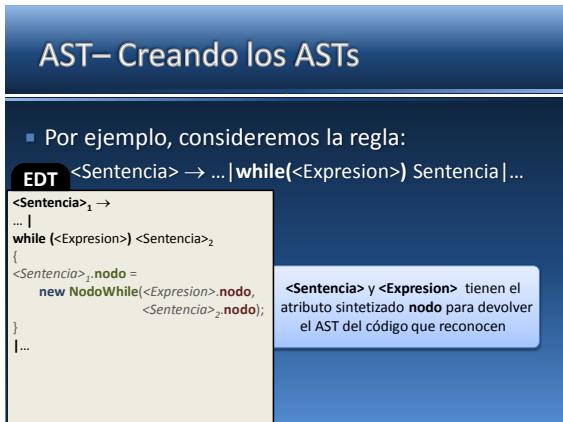






AST – Creando los AST

- Los **AST** se **crearán** a medida que se va **realizando** el **Análisis Sintáctico**.
- Haremos una **EDT** sobre la gramática utilizada en el Analizador Sintáctico donde las **acciones semánticas** construirán los **ASTs**
- Luego, modificaremos los **métodos** del Analizador Sintáctico para reflejar las **acciones semánticas** de la EDT



AST – Creando los ASTs

■ Por ejemplo, consideremos

```
EDT <Sentencia> → ... | while(<Expresión>) Sentencia Sintáctico
```

```
<Sentencia>_1 →
...
| while (<Expresión>) <Sentencia>_2
{
  <Sentencia>_2.nodo =
    new NodoWhile(<Expresión>.nodo,
                  <Sentencia>_2.nodo);
}
```

Sentencia() retorna el nodo correspondiente a la raíz del AST que representa la sentencia reconocida

```
NodoSentencia Sentencia(
...
Sino Si(lookahead es 'while')
  match('while');
  match(')');
  NodoExpresión NE = Expresión();
  match(')');
  NodoSentencia NS = Sentencia();
)
...
```

AST – Creando los ASTs

■ Por ejemplo, consideremos la regla:

EDT	<Sentencia> → ... while(<Expresión>) Sentencia Sintáctico	Código Sintáctico
		NodoSentencia Sentencia(){ ... Sino Si(lookahead es 'while'){ match('while'); match(')'); NodoExpresión NE = Expresión(); match(')'); NodoSentencia NS = Sentencia(); } ... }

Analizador Semántico

Analizador Semántico

■ Para realizar los **chequeos** de manera adecuada el análisis semántico se divide en **dos etapas**:

1. Chequeo de Declaraciones
2. Chequeo de Sentencias

■ Es importante que se realicen en ese orden

Analizador Semántico – Chequeo de Declaraciones

- Chequear que toda declaración fue **Correctamente Declarada**
- En general implica:
 - Chequear que no haya **nombres repetidos** en el mismo contexto (*ej:* dos clases con el mismo nombre)
 - Chequear que todo **nombre usado** en una declaración **haya sido declarado** (*ej:* el tipo de un método)
 - Chequear que no haya **herencia circular**
 - Chequear que todo **método redefinido** tenga exactamente la **misma firma** que el **ancestro**

Analizador Semántico – Chequeo de Declaraciones

- Chequear que toda declaración fue **Correctamente Declarada**
- En general implica:
 - Chequear que no haya **nombres repetidos** en el mismo contexto (*ej:* dos clases con el mismo nombre)
 - Chequear que todo **nombre usado** en una declaración **haya sido declarado** (*ej:* el tipo de un método)
 - Chequear que no haya **herencia circular**
 - Chequear que todo **método redefinido** tenga exactamente la **misma firma** que el **ancestro**

Pueden ocurrir conflictos entre la TS y el AST, por lo tanto se debe realizar la TS en el Análisis Sintáctico!

Analizador Semántico – Chequeo de Declaraciones

- Estos **controles** y tareas se realizarán **directamente** sobre la **Tabla de Símbolos**
- Para esto se analizarán las entradas de
 - Clases
 - Métodos
 - Constructores
 - Variables de Instancia y Parámetros

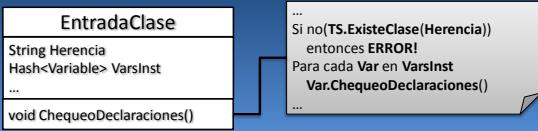
Los controles de **Variables locales** deben realizarse en el **Chequeo de Sentencias!!!**

Analizador Semántico – Chequeo de Declaraciones

- Por ejemplo, para la **entrada de una clase** en la TS se controla que:
 - Si tiene herencia explícita **herede de una clase declarada**
 - No tenga herencia circular** (i.e. que en su línea de ancestros no aparezca ella misma)
 - No tenga dos métodos o variables** de instancia con el **mismo nombre**
 - Que todos sus **métodos**, sus **variables** de instancia y su **constructor** se encuentren **correctamente declarados**

Analizador Semántico – Chequeo de Declaraciones

- La **implementación** de todos estos **controles** puede diseñarse de **varias maneras**
- Por ejemplo, se pueden agregar **métodos** para realizar los controles directamente en las **entradas** de la TS



Analizador Semántico – Chequeo de Declaraciones

- Otra tarea importante que debe realizarse en el chequeo de declaraciones es la **Actualización de las Tablas de Métodos en las clases en base a Herencia**
- Esto implica agregar todos los métodos heredados (accesibles) a una clase
- Con que hay que tener cuidado al realizar esto?
 - Métodos sobre-escritos**: una clase sólo tiene acceso a la **última versión** de los métodos que tiene y hereda
 - Además, podemos aprovechar y mientras actualizamos **controlamos** si un método es **correctamente redefinido**

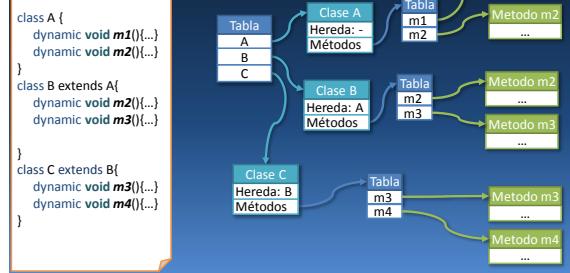
Analizador Semántico – Chequeo de Declaraciones

```

class A {
    dynamic void m1(){...}
    dynamic void m2(){...}
}
class B extends A{
    dynamic void m2(){...}
    dynamic void m3(){...}
}
class C extends B{
    dynamic void m3(){...}
    dynamic void m4(){...}
}
  
```

Analizador Semántico – Chequeo de Declaraciones

Actualización de las Tablas de Métodos en las clases en base a Herencia



Análizador Semántico – Chequeo de Declaraciones

Actualización de las Tablas de Métodos en las clases en base a Herencia

```
class A {
    dynamic void m1() {...}
    dynamic void m2() {...}
}
class B extends A {
    dynamic void m2() {...}
    dynamic void m3() {...}
}
class C extends B {
    dynamic void m3() {...}
    dynamic void m4() {...}
}
```

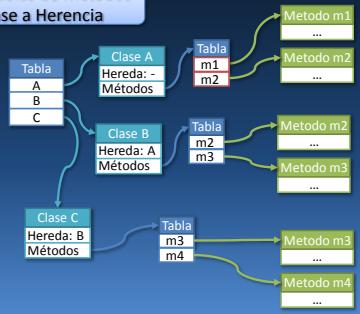


B hereda de A, por lo tanto hay que agregar los métodos no sobre-escritos y controlar que los sobre-escritos están bien

Analizador Semántico – Chequeo de Declaraciones

Actualización de las Tablas de Métodos en las clases en base a Herencia

```
class A {  
    dynamic void m1(){...}  
    dynamic void m2(){...}  
}  
class B extends A{  
    dynamic void m2(){...}  
    dynamic void m3(){...}  
}  
class C extends B{  
    dynamic void m3(){...}  
    dynamic void m4(){...}  
}
```



Análizador Semántico – Chequeo de Declaraciones

Actualización de las Tablas de Métodos en las clases en base a Herencia

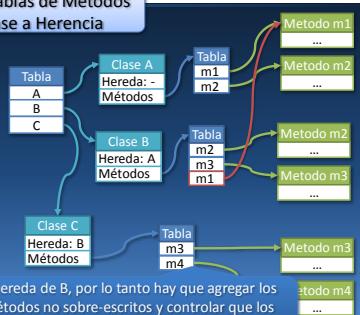
```
class A {
    dynamic void m1() {...}
    dynamic void m2() {...}
}
class B extends A {
    dynamic void m2() {...}
    dynamic void m3() {...}
}
class C extends B {
    dynamic void m3() {...}
    dynamic void m4() {...}
}
```



Analizador Semántico – Chequeo de Declaraciones
Análisis de la Tabla de Múltiples

Actualización de las Tablas de Métodos en las clases en base a Herencia

```
class A {  
    dynamic void m1(){...}  
    dynamic void m2(){...}  
}  
class B extends A{  
    dynamic void m2(){...}  
    dynamic void m3(){...}  
}  
class C extends B{  
    dynamic void m3(){...}  
    dynamic void m4(){...}  
}
```

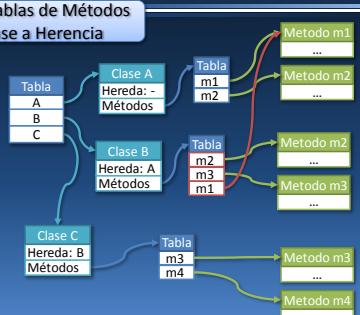


C hereda de B, por lo tanto hay que agregar los métodos no sobre-escritos y controlar que los sobre-escritos están bien

Analizador Semántico – Chequeo de Declaraciones

Actualización de las Tablas de Métodos en las clases en base a Herencia

```
class A {
    dynamic void m1() {...}
    dynamic void m2() {...}
}
class B extends A{
    dynamic void m2() {...}
    dynamic void m3() {...}
}
class C extends B{
    dynamic void m3() {...}
    dynamic void m4() {...}
}
```



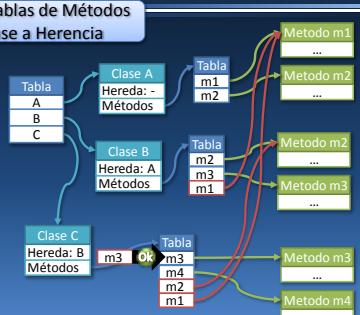
Analizador Semántico – Chequeo de Declaraciones

Actualización de las Tablas de Métodos en las clases en base a Herencia

```
class A {
    dynamic void m1(){...}
    dynamic void m2(){...}
}

class B extends A{
    dynamic void m2(){...}
    dynamic void m3(){...}
}

class C extends B{
    dynamic void m3(){...}
    dynamic void m4(){...}
}
```



Analizador Semántico Chequeo de Sentencias

Analizador Semántico – Chequeo de Sentencias

- El **Chequeo de Sentencias** consiste en chequear que el cuerpo de los métodos y constructores no tiene errores semánticos.
- Está centrado en el **chequeo de tipos** y la **resolución de nombres** en sentencias y expresiones

Analizador Semántico – Chequeo de Sentencias

- La **resolución de nombres** se refiere a que cuando se utiliza un **Identificador** en una expresión o sentencia, se controle que sea el nombre de la entidad adecuada.
- Por ejemplo:
- X() → X debe ser el nombre de un **método** en la **Clase Actual**
 - Y.X() → X debe ser el nombre de un **método** de la Clase declarada para Y (Y no puede ser de tipo primitivo)
 - X → X puede ser una **variable local** o un **parámetro del método actual**, o sino una **variable de instancia** de la clase actual
 - Y.X → X debe ser el nombre de una **variable de Instancia** de la Clase declarada para Y (Y no puede ser de tipo primitivo)

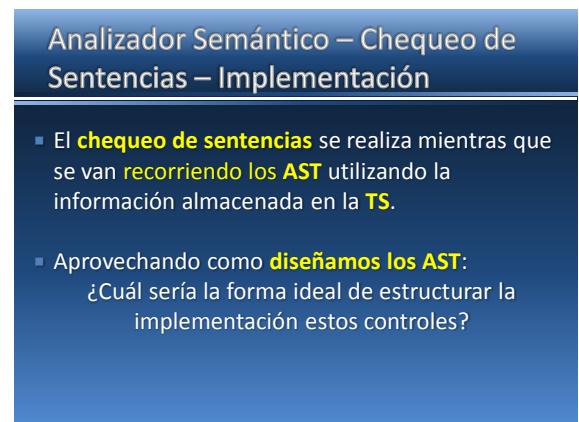
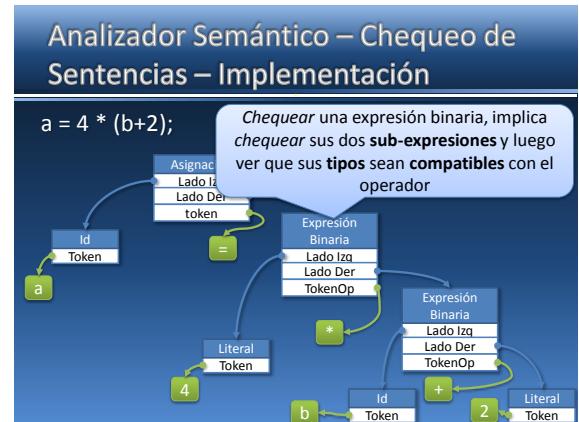
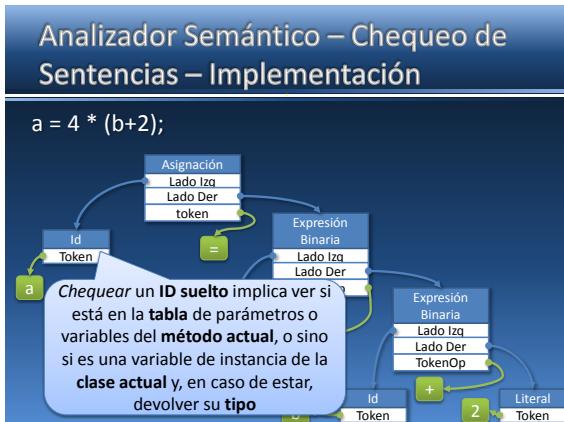
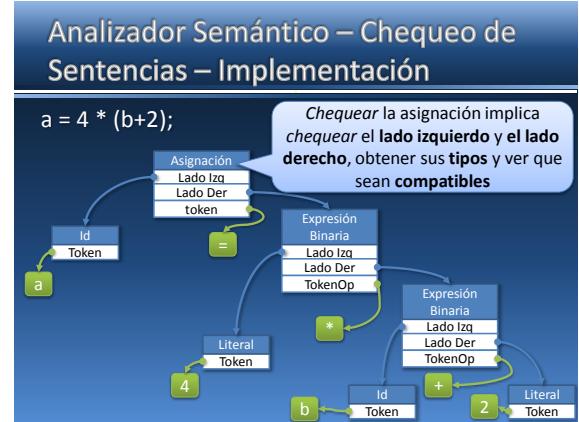
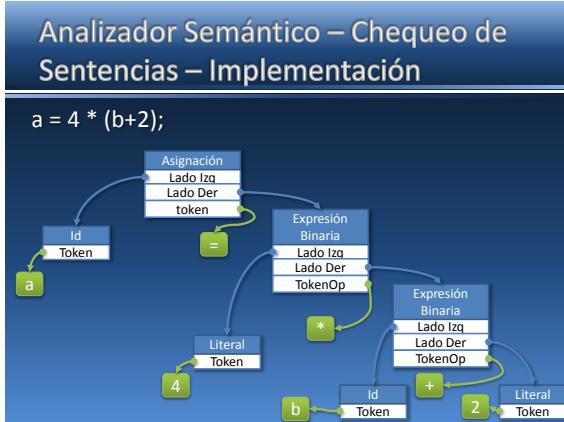
Analizador Semántico – Chequeo de Sentencias

- El **chequeo de tipos**, implica chequear que:
- Los tipos de la **expresión** usada en una **sentencia** sean los requeridos
- Por ejemplo:
 - El tipo de la expresión de un **while** sea **booleano**
 - El tipo la expresión del lado derecho de una **asignación** sea **igual** o un **subtipo** del **tipo del identificador** al final de la cadena del lado izquierdo

Analizador Semántico – Chequeo de Sentencias

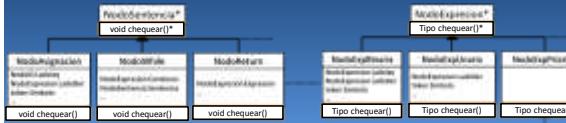
- El **chequeo de tipos**, implica chequear que:
- Los tipos de las **sub-expresiones** en una **expresión** sean los adecuados.
- Por ejemplo:
 - El **tipo de las sub-expresiones** de una expresión binaria con operador '+' sea **int**
 - El **tipo de la sub-expresión** de una expresión unaria con operador '!' sea **boolean**

Chequeo de Sentencias Implementación



Análizador Semántico – Chequeo de Sentencias – Implementación

- Una implementación posible:
- Agregamos a cada tipo de nodo del AST un **método** donde implementaremos los **controles asociados al nodo**



Análizador Semántico – Chequeo de Sentencias – Implementación

- Por Ejemplo (Pseudo código)

```

NodoWhile
NodoExpresion Condicion
NodoSentencia Sentencia
...
void chequear()
  
```

Si *Tipo* de **Condicion.chequear()** no es **boolean**
entonces **ERROR!**
sino **Sentencia.chequear()**

```

NodoAsignacion
NodoID LadoIzq
NodoExpresion LadoDer
token Simbolo
...
void chequear()
  
```

Si el tipo **LadoDer.chequear()** no es el mismo o
hereda del tipo de **LadoIzq.chequear()**
entonces **ERROR!**

Análizador Semántico – Chequeo de Sentencias – Otras Consideraciones

- Varios controles (los de ID) hacen referencia al **Método Actual** o a la **Clase Actual**
 - Es **importante** que la **tabla de símbolos** provea mecanismos para identificar estas entradas, y se deberá cambiarlas cuando se está analizando otro método o clase.
- Tener cuidado de **no chequear** el cuerpo de un método en **más de una clase!!!** (Herencia)

Resumen y Pautas

- Diseñar la TS y cada una de sus entradas
- Diseñar el AST identificando cada uno de los tipos de nodos y sus relaciones
- Estudiar cómo construir tanto la TS como los AST mientras se realiza el Análisis Sintáctico (EDT)
- Diseñar los métodos para hacer el Control de Declaraciones sobre la TS
- Diseñar los métodos para hacer el Control de Sentencias en cada nodo del AST
- Implementar un Analizador Semántico de dos pasadas para mini-Java basándose en estos diseños