# Proyecto Compiladores e Interpretes

Etapa 2

Analizador Sintáctico

Rodrigo Díaz Figueroa

LU 97400

# Clases utilizadas y decisiones de diseño

## Cambios desde la última etapa:

Se cambio el nombre de la clase Utilidades por Utl.
Se agregaron las clases AnalizadorSintactico y SintacticException

## Clases:

**AnalizadorLexico**
**AnalizadorSintáctico**
**Principal**
**EntradaSalida**
**Token**
**Utl**
**LexicoException**
**SintacticException**

**Nuevas clases:**

**AnalizadorSintáctico:**
Se encarga de analizar la sintaxis del archivo a través de los tokens que recibe del Analizador Léxico y de lanzar excepciones de tipo SintacticException.

**Utl:**
Previamente: Utilidades.
Esta clase es estática y en ella se encuentran las constantes de tipo entero que representan a los todos tipos de tokens y algunos métodos estáticos que utilizan otras clases.

**SintacticException:**
Clase que hereda de Exception y es utilizada por AnalizadorSintáctico para lanzar excepciones.

# Decisiones de diseño:

- Se optó por utilizar solo un tipo de excepción para los errores sintácticos y utilizar el mensaje para especificar el tipo de error

- El tipo de los tokens es un entero, y para compararlos se utilizan métodos provistos por la clase Token.

# Logros

Los logros que se intentan alcanzar en esta etapa son:

- **Imbatibilidad Sintáctica.**

# Compilación y Ejecución

Para compilar el programa desde consola se deberá ejecutar el siguiente comando

```
>javac Principal.java
```

*Observación: Si este comando no compila todas las clases del programa se deberá ejecutar entonces*

```
>javac Principal.java AnalizadorSintactico.java AnalizadorLexico.java
Utl.java Token.java EntradaSalida.java LexicoException.java
SintacticException.java
```

Para la ejecución del programa desde consola se deberá ejecutar

```
>java Principal <archivo fuente>
```

Donde <archivo fuente> es el path absoluto o relativo dependiendo de donde se encuentra el archivo fuente a analizar en el equipo.

**Ejemplos:**

```
>java Principal ./Test/Correctos/Test1.java
```

El archivo Test1.java debe estar dentro del directorio Test/Correctos/ que se encuentra en el mismo directorio que los archivos fuente

```
>java Principal C:/Testing/Incorrectos/Test2.java
```

El archivo Test2.java debe estar dentro del directorio que especifica el path absoluto

# Evolución de la gramática.

## Paso 0: Eliminación de símbolos EBNF (superíndices *, + y ?)

Inicio1 → Clase Inicio2
Inicio2 → Inicio1 | ε
Clase → class idClase Herencia { Miembros }
Herencia → extends idClase | ε
Miembros → Miembro Miembros | ε
Miembro → Atributo | Ctor | Metodo
Atributo → Visibilidad Tipo ListaDecVars ;
Metodo → FormaMetodo TipoMetodo idMetVar ArgsFormales Bloque
Ctor → idClase ArgsFormales Bloque
ArgsFormales → ( ListaArg )
ListaArg → ε | Arg , ArgFormales
ArgFormales → ε | Arg , ArgFormales
Arg → Tipo idMetVar
FormaMetodo → static | dynamic
Visibilidad → public | private
TipoMetodo → Tipo | void
Tipo → TipoPrimitivo | TipoReferencia
TipoPrimitivo → boolean | char | int
TipoReferencia → idClase | String | TipoPrimitivo [ ]
ListaDecVars → idMetVar
ListaDecVars → idMetVar , ListaDecVars
Bloque → { Sentencias }
Sentencias → Sentencia Sentencias | ε
Sentencia → ;
Sentencia → Asignacion ;
Sentencia → SentenciaLlamada ;
Sentencia → Tipo ListaDecVars ;
Sentencia → if ( Expresion ) Sentencia
Sentencia → if ( Expresion ) Sentencia else Sentencia
Sentencia → while ( Expresion ) Sentencia
Sentencia → Bloque
Sentencia → return Expresiones ;
Asignacion → AccesoVar = Expresion
Asignacion → AccesoThis = Expresion
SentenciaLlamada → ( Primario )
Expresiones → Expresion Expresiones | ε
Expresion → ExpOr
ExpOr → ExpOr || ExpAnd | ExpAnd
ExpAnd → ExpAnd && ExpIg | ExpIg
ExpIg → ExpIg OpIg ExpComp | ExpComp
ExpComp → ExpAd OpComp ExpAd | ExpAd
ExpAd → ExpAd OpAd ExpMul | ExpMul
ExpMul → ExpMul OpMul ExpUn | ExpUn
ExpUn → OpUn ExpUn | Operando
OpIg → == | !=
OpComp → < | > | <= | >=
OpAd → + | -
OpUn → + | - | !
OpMul → * | /
Operando → Literal
Operando → Primario
Literal → null | true | false | intLiteral | charLiteral | stringLiteral

Primario → ExpresionParentizada
Primario → AccesoThis
Primario → AccesoVar
Primario → LlamadaMetodo
Primario → LlamadaMetodoEstatico
Primario → LlamadaCtor
ExpresionParentizada → ( Expresion ) Encadenado
AccesoThis → this Encadenado
AccesoVar → idMetVar Encadenado
LlamadaMetodo → idMetVar ArgsActuales Encadenado
LlamadaMetodoEstatico → idClase . LlamadaMetodo Encadenado
LlamdaCtor → new idClase ArgsActuales Encadenado
LlamdaCtor → new TipoPrimitivo [ Expresion ] Encadenado
ArgsActuales → ( ListaExpresiones )
ListaExpresiones → Expresion ListaExp | ε
ListaExp → , Expresion ListaExp | ε
ListaExps → Expresion
ListaExps → Expresion , ListaExps
Encadenado → ε | . Acceso | AccesoArregloEncadenado
Acceso → LlamadaMetodoEncadenado | AccesoVarEncadenado
LlamadaMetodoEncadenado → idMetVar ArgsActuales Encadenado
AccesoVarEncadenado → idMetVar Encadenado
AccesoArregloEncadenado → [ Expresion ] Encadenado

## Paso 1: Eliminación de recursión a izquierda

Inicio1 → Clase Inicio2
Inicio2 → Inicio1 | ε
Clase → class idClase Herencia { Miembros }
Herencia → extends idClase | ε
Miembros → Miembro Miembros | ε
Miembro → Atributo | Ctor | Metodo
Atributo → Visibilidad Tipo ListaDecVars ;
Metodo → FormaMetodo TipoMetodo idMetVar ArgsFormales Bloque
Ctor → idClase ArgsFormales Bloque
ArgsFormales → ( ListaArg )
ListaArg → ε | Arg , ArgFormales
ArgFormales → ε | Arg , ArgFormales
Arg → Tipo idMetVar
FormaMetodo → static | dynamic
Visibilidad → public | private
TipoMetodo → Tipo | void
Tipo → TipoPrimitivo | TipoReferencia
TipoPrimitivo → boolean | char | int
TipoReferencia → idClase | String | TipoPrimitivo [ ]
ListaDecVars → idMetVar
ListaDecVars → idMetVar , ListaDecVars
Bloque → { Sentencias }
Sentencias → Sentencia Sentencias | ε
Sentencia → ;
Sentencia → Asignacion ;
Sentencia → SentenciaLlamada ;
Sentencia → Tipo ListaDecVars ;

Sentencia → if ( Expresion ) Sentencia
Sentencia → if ( Expresion ) Sentencia else Sentencia
Sentencia → while ( Expresion ) Sentencia
Sentencia → Bloque
Sentencia → return Expresiones ;
Asignacion → AccesoVar = Expresion
Asignacion → AccesoThis = Expresion
SentenciaLlamada → ( Primario )
Expresiones → Expresion Expresiones | ε
Expresion → ExpOr
ExpOr → ExpAnd ExpOrR
ExpOrR → || ExpAnd ExpOrR | ε
ExpAnd → ExpIg ExpAndR
ExpAndR → && ExpIg ExpAndR | ε
ExpIg → ExpComp ExpIgR
ExpIgR → OpIgual ExpComp ExpIgR | ε
ExpComp → ExpAd OpComp ExpAd | ExpAd
ExpAd → ExpMul ExpAdR
ExpAdR → OpAd ExpMul ExpAdR | ε
ExpMul → ExpUn ExpMulR
ExpMulR → OpMul ExpUn ExpMulR | ε
ExpUn → OpUn ExpUn | Operando
OpIg → == | !=
OpComp → < | > | <= | >=
OpAd → + | -
OpUn → + | - | !
OpMul → * | /
Operando → Literal
Operando → Primario
Literal → null | true | false | intLiteral | charLiteral | stringLiteral
Primario → ExpresionParentizada
Primario → AccesoThis
Primario → AccesoVar
Primario → LlamadaMetodo
Primario → LlamadaMetodoEstatico
Primario → LlamadaCtor
ExpresionParentizada → ( Expresion ) Encadenado
AccesoThis → this Encadenado
AccesoVar → idMetVar Encadenado
LlamadaMetodo → idMetVar ArgsActuales Encadenado
LlamadaMetodoEstatico → idClase . LlamadaMetodo Encadenado
LlamdaCtor → new idClase ArgsActuales Encadenado
LlamdaCtor → new TipoPrimitivo [ Expresion ] Encadenado
ArgsActuales → ( ListaExpresiones )
ListaExpresiones → Expresion ListaExp | ε
ListaExp → , Expresion ListaExp | ε
ListaExps → Expresion
ListaExps → Expresion , ListaExps
Encadenado → ε | . Acceso | AccesoArregloEncadenado
Acceso → LlamadaMetodoEncadenado | AccesoVarEncadenado
LlamadaMetodoEncadenado → idMetVar ArgsActuales Encadenado
AccesoVarEncadenado → idMetVar Encadenado
AccesoArregloEncadenado → [ Expresion ] Encadenado

## Paso 2: Factorización

Inicio → Clase Clases $
Clases → Clase Clases
Clases → ε
Clase → class idClase Herencia { Miembros }
Herencia → extends idClase
Herencia → ε
Miembros → Miembro Miembros
Miembros → ε
Miembro → Atributo
Miembro → Ctor
Miembro → Metodo
Atributo → Visibilidad Tipo ListaDecVars ;
Metodo → FormaMetodo TipoMetodo idMetVar ArgsFormales Bloque
Ctor → idClase ArgsFormales Bloque
ArgsFormales → ( ListaArg )
ListaArg → Arg ArgFormales
ListaArg → ε
ArgFormales → , Arg ArgFormales
ArgFormales → ε
Arg → Tipo idMetVar
FormaMetodo → static
FormaMetodo → dynamic
Visibilidad → public
Visibilidad → private
TipoMetodo → Tipo
TipoMetodo → void
Tipo → boolean PosibleArreglo
Tipo → char PosibleArreglo
Tipo → int PosibleArreglo
Tipo → idClase
Tipo → String
PosibleArreglo → [ ]
PosibleArreglo → ε
TipoPrimitivo → boolean
TipoPrimitivo → char
TipoPrimitivo → int
TipoReferencia → idClase
TipoReferencia → String
TipoReferencia → TipoPrimitivo [ ]
ListaDecVars → idMetVar ListaDV
ListaDV → , idMetVar ListaDV
ListaDV → ε
Bloque → { Sentencias }
Sentencias → Sentencia Sentencias
Sentencias → ε
Sentencia → ;
Sentencia → if ( Expresion ) Sentencia SentenciaElse
Sentencia → while ( Expresion ) Sentencia
Sentencia → return Expresiones ;
Sentencia → Asignacion ;
Sentencia → SentenciaLlamada ;
Sentencia → Tipo ListaDecVars ;
Sentencia → Bloque
SentenciaElse → else Sentencia

SentenciaElse → ε
Expresiones → Expresion
Expresiones → ε
Asignacion → AccesoVar = Expresion
Asignacion → AccesoThis = Expresion
SentenciaLlamada → ( Primario )
Expresion → ExpOr
ExpOr → ExpAnd ExpOrR
ExpOrR → || ExpAnd ExpOrR
ExpOrR → ε
ExpAnd → ExpIg ExpAndR
ExpAndR → && ExpIg ExpAndR
ExpAndR → ε
ExpIg → ExpComp ExpIgR
ExpIgR → OpIgual ExpComp ExpIgR
ExpIgR → ε
ExpComp → ExpAd ExpCompR
ExpCompR → OpComp ExpAd
ExpCompR → ε
ExpAd → ExpMul ExpAdR
ExpAdR → OpAd ExpMul ExpAdR
ExpAdR → ε
ExpMul → ExpUn ExpMulR
ExpMulR → OpMul ExpUn ExpMulR
ExpMulR → ε
ExpUn → OpUn ExpUn
ExpUn → Operando
OpIgual → ==
OpIgual → !=
OpComp → <
OpComp → >
OpComp → <=
OpComp → >=
OpAd → +
OpAd → -
OpUn → +
OpUn → -
OpUn → !
OpMul → *
OpMul → /
Operando → Literal
Operando → Primario
Literal → null
Literal → true
Literal → false
Literal → intLiteral
Literal → charLiteral
Literal → stringLiteral
Primario → idMetVar MetodoVariable
Primario → ExpresionParentizada
Primario → AccesoThis
Primario → LlamadaMetodoEstatico
Primario → LlamadaCtor
MetodoVariable → ArgsActuales Encadenado
MetodoVariable → Encadenado
ExpresionParentizada → ( Expresion ) Encadenado
Encadenado → . idMetVar Acceso

Encadenado → AccesoArregloEncadenado
Encadenado → ε
Acceso → LlamadaMetodoEncadenado
Acceso → AccesoVarEncadenado
AccesoThis → this Encadenado
AccesoVar → idMetVar Encadenado
LlamadaMetodo → idMetVar ArgsActuales Encadenado
LlamadaMetodoEstatico → idClase . LlamadaMetodo
LlamadaCtor → new LlamadaCtorR
LlamadaCtorR → idClase ArgsActuales Encadenado
LlamadaCtorR → TipoPrimitivo [ Expresion ] Encadenado
ArgsActuales → ( ListaExpresiones )
ListaExpresiones → Expresion ListaExp
ListaExpresiones → ε
ListaExp → , Expresion ListaExp
ListaExp → ε
LlamadaMetodoEncadenado → ArgsActuales Encadenado
AccesoVarEncadenado → Encadenado
AccesoArregloEncadenado → [ Expresion ] Encadenado

# Errores sintácticos detectados.

Los errores detectados pueden ser agrupados en dos grupos:

- Errores simples de un token inesperado, que ocurren al intentar hacer un "*match*" del token actual con el token esperado. Estos errores solo brindan esa información.
- Errores específicos, que son capturados antes de realizar un "*match*" y brindan mas información sobre que tipo de error, los tokens que eran esperados y el token actual.

A continuación se listaran y explicaran brevemente los errores específicos que el compilador es capaz de detectar.

**Error de token no incluido en grupo (Token inesperado):**

Este es el error especifico mas simple. Es similar al error simple salvo que brinda la información de varios tipos de tokens que el analizador espera recibir y el token encontrado.

**Declaración de clase mal formada**

Este error se da cuando la declaración de una clase esta mal formada, es decir, no se comienza con la palabra reservada *class*.
Brinda información del token esperado y el token encontrado.

**Miembro mal formado**

Este error se da cuando un miembro de una clase esta mal formado, es decir, no se encontró un token valido que forme parte del comienzo de un atributo, un método, o un constructor.
Brinda información de los tokens esperados y el token encontrado.

**Método mal formado. Argumentos inexistentes**

Este error ocurre cuando en la declaración de un método falta la lista de argumentos.
Brinda información de los tokens esperados y el token encontrado.

**Método mal formado. Tipo de método inexistente**

Este error ocurre cuando en la declaración de un método no se encuentra el tipo.
Brinda información de los tokens esperados y el token encontrado.

**Lista de argumentos mal formada**

Este error se detecta cuando una lista de argumentos esta mal formada.
Brinda información de los tokens esperados y el token encontrado.

**Tipo invalido**

Este error ocurre cuando un tipo no es valido.
Brinda información de los tokens esperados y el token encontrado.

**Declaración mal formada**

Este error ocurre cuando una declaración esta mal formada.
Brinda información de los tokens esperados y el token encontrado.

**Sentencia mal formada**

Este error se detecta cuando una sentencia se encuentra mal formada.
Brinda información de los tokens esperados y el token encontrado.

**Expresión mal formada**

Este error se detecta cuando una expresión esta mal formada.
Brinda información de los tokens esperados y el token encontrado.

# Testing

Para el testing del programa se utilizaron varios casos de prueba tanto correctos como incorrectos.
Cada caso de test tanto correcto como incorrecto contiene al principio un comentario con la explicación, o el tipo de error para los incorrectos, para la cual esta hecho dicho caso.
A continuación se explicaran brevemente solo los casos de test correctos. Para los casos de test incorrectos, solo se agruparan por tipo de error que se espera obtener.

## Casos de test correctos:

**Test1.java**

Test general que intenta cubrir todos los posibles casos

**Test2.java**

Varias declaraciones de clases

**Test3.java**

Varias sentencias con expresiones entre paréntesis dobles

**Test4.java**

Varios atributos declarados de a uno por vez

**Test5.java**

Varios atributos declarados en forma de lista

**Test6.java**

Varios constructores con distintos argumentos

**Test7.java**

Varias declaraciones de clases con y sin herencia

**Test8.java**

Una clase con los tres tipos de miembros (atributos, métodos y constructores) en cualquier orden

**Test9.java**

Varios tipos de métodos con diferentes argumentos

**Test10.java**

Métodos con sentencias especiales ; y return;

**Test11.java**

Método con sentencias if else anidados

**Test12.java**

Método con sentencias while anidadas

**Test13.java**

Método con varias sentencias validas

## Casos de test incorrectos:

A continuación se listaran los tipos de errores y los números de test incorrectos que los producen (en total se crearon 49 casos de test incorrectos).

**Errores simples:**

Test n.º : 3, 4, 7 ,8, 9, 16, 20, 25, 26, 31 y 43.

**Declaración de clase mal formada:**

Test n.º : 1, 2 y 6.

**Miembro mal formado:**

Test n.º: 5, 10, 19, 29 y 34.

**Método mal formado. Argumentos inexistentes:**

Test n.º: 17.

**Método mal formado. Tipo inexistente:**

Test n.º: 15.

**Lista de argumentos mal formada:**

Test n.º: 18, 21, 23 y 24.

**Tipo invalido:**

Test n.º: 11 y 22.

**Declaración mal formada:**

Test n.º: 12, 13, 14, 45, 46, 47 y 48.

**Sentencia mal formada:**

Test n.º: 28, 33, 36, 37 y 49.

**Expresión mal formada:**

Test n.º: 27, 30, 32, 35, 38, 39, 40, 41, 42 y 44.