[MiniJava]

Manual técnico

Ramiro Agis - LU. 96248
Victoria Martínez de la Cruz - LU. 87620

Índice

Introducción

Información sobre el manual

El presente manual detalla las características de los módulos que componen al compilador presentado. Se incluye documentación sobre,

- Analizador Léxico,
  En esta parte indicamos el alfabeto, la gramática, los *tokens* reconocidos, el diseño del autómata para reconocimiento de *tokens* y detalles de implementación, incluyendo la estructura de clases, las decisiones de diseño y la batería de tests.

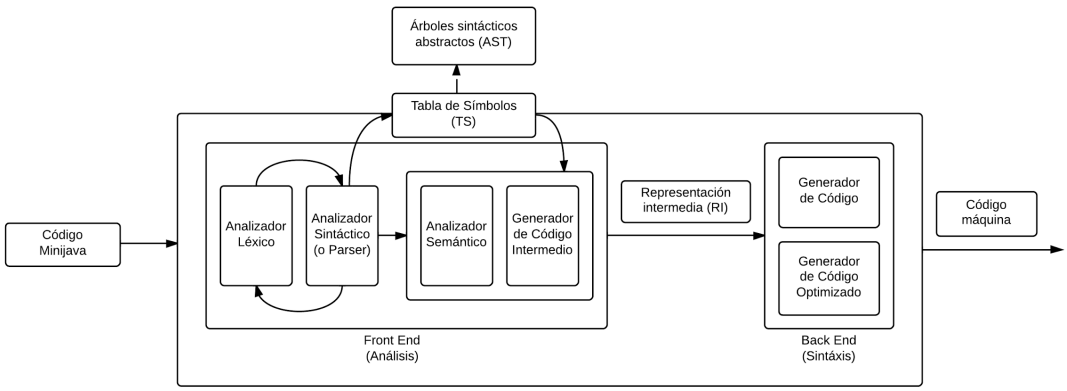
- Analizador Sintáctico,
  En esta sección realizamos un análisis exhaustivo sobre la gramática y mostramos, en un proceso iterativo, la evolución de la misma al aplicar algoritmos que la transforman en LL(1). Este proceso incluye la eliminación de los símbolos EBNF, la eliminación de la recursión a izquierda (directa e indirecta), la factorización y la detección de ambigüedades. A continuación estudiamos si efectivamente la gramática obtenida es LL(1) y proponemos la versión final para la implementación.
  También se incluyen detalles de implementación, incluyendo la estructura de clases, las decisiones de diseño y la batería de tests.

- Analizador Semántico,
  En esta parte desarrollamos un esquema de traducción (EDT) que da inicio a la construcción de la tabla de símbolos (TS) y los árboles sintácticos abstractos (AST) y detallamos el proceso realizado al realizar los controles semánticos (chequeo de declaraciones y chequeo de sentencias).
  También se incluyen detalles de implementación, incluyendo la estructura de clases, las decisiones de diseño y la batería de tests.

- Generación de Código Intermedio,
  En la sección final mostramos el proceso de generación de código intermedio. También se incluyen detalles de implementación, incluyendo la estructura de clases, las decisiones de diseño y la batería de tests.

Estructura del compilador

El compilador presentado puede esquematizarse como sigue,



Analizador Léxico

Alfabeto de entrada

∑ = { ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ '\0' '\n' '\t' <blank>}

El alfabeto de entrada utilizado para MiniJava está conformado por los caracteres imprimibles del ASCII básico y por tres caracteres de control: '\0', '\n' y '\t'.

Gramática

<*Inicial*> → <*Clase*>+

<*Clase*> → **class** *identificador* <*Herencia*>? **{** <*Miembro*>* **}**

<*Herencia*> → **extends** *identificador*

<Miembro> → <*Atributo*> | <*Ctor*> | <*Metodo*>

<*Atributo*> → **var** <*Tipo*> <*ListaDeVars*> **;**

<*Metodo*> → <*ModMetodo*> <*TipoMetodo*> *identificador* <*ArgsFormales*> <*VarsLocales*> <*Bloque*>

<*Ctor*> → *identificador* <*ArgsFormales*> <*VarsLocales*> <*Bloque*>

<*ArgsFormales*> → **(** <*ListaArgsFormales*>? **)**

<*ListaArgsFormales*> → <*ArgFormal*>
<*ListaArgsFormales*> → <*ArgFormal*> **,** <*ListaArgsFormales*>

<*ArgFormal*> → <*Tipo*> *identificador*

<*VarsLocales*> → <*Atributo*>*

<*ModMetodo*> → **static** | **dynamic**

<*TipoMetodo*> → <*Tipo*> | **void**

<*Tipo*> → <*TipoPrimitivo*> | *identificador*

<*TipoPrimitivo*> → **boolean** | **char** | **int** | **String**

<*ListaDecVars*> → *identificador*
<*ListaDecVars*> → *identificador* **,** <*ListaDecVars*>

<*Bloque*> → **{** <*Sentencia*>* **}**

<*Sentencia*> → **;**
<*Sentencia*> → <*Asignacion*> **;**
<*Sentencia*> → <*SentenciaSimple*> **;**
<*Sentencia*> → **if (** <*Expresion*> **)** <*Sentencia*>
<*Sentencia*> → **if (** <*Expresion*> **)** <*Sentencia*> **else** <*Sentencia*>
<*Sentencia*> → **while (** <*Expresion*> **)** <*Sentencia*>
<*Sentencia*> → **for (** <*Asignacion*> **;** <*Expresion*> **;** <*Expresion*> **)** <*Sentencia*>
<*Sentencia*> → <*Bloque*>
<*Sentencia*> → **return** <*Expresion*>? **;**

<*Asignacion*> → *identificador* **=** <*Expresion*>

<*SentenciaSimple*> → **(** <*Expresion*> **)**

<*Expresion*> → <*Expresion6*>

<*Expresion6*> → <*Expresion6*> || <*Expresion5*>
<*Expresion6*> → <*Expresion5*>

<*Expresion5*> → <*Expresion5*> && <*Expresion4*>
<*Expresion5*> → <*Expresion4*>

<*Expresion4*> → <*Expresion4*> <*Operador4*> <*Expresion3*>
<*Expresion4*> → <*Expresion3*>

<*Expresion3*> → <*Expresion3*> <*Operador3*> <*Expresion2*>
<*Expresion3*> → <*Expresion2*>

<*Expresion2*> → <*Expresion2*> <*Operador2*> <*Expresion1*>
<*Expresion2*> → <*Expresion1*>

<*Expresion1*> → <*Expresion1*> <*Operador1*> <*Expresion0*>
<*Expresion1*> → <*Expresion0*>

<*Expresion0*> → <*OperadorUnario*> <*Primario*>
<*Expresion0*> → <*Primario*>

<*OperadorUnario*> → **!** | **+** | **-**

<*Operador1*> → ***** | **/** | **%**
<*Operador2*> → **+** | **-**
<*Operador3*> → **<** | **>** | **>=** | **<=**
<*Operador4*> → **==** | **!=**

<*Primario*> → **this**
<*Primario*> → <*Literal*>
<*Primario*> → **(** <*Expresion*> **)** <*Llamada*>*
<*Primario*> → *identificador* <*Llamada*>*
<*Primario*> → **new** *identificador* <*ArgsActuales*> <*Llamada*>*
<*Primario*> → *identificador* <*ArgsActuales*> <*Llamada*>*

<*Llamada*> → **.** *identificador* <*ArgsActuales*>

<*Literal*> → **null** | **true** | **false** | intLiteral | charLiteral | stringLiteral

<*ArgsActuales*> → **(** <*ListaExps*>? **)**

<*ListaExps*> → <*Expresion*>
<*ListaExps*> → <*Expresion*> **,** <*ListaExps*>

Tokens reconocidos

Token	Patrón	Tipo
class	class	Palabra clave
extends	extends	Palabra clave
var	var	Palabra clave
static	static	Palabra clave
dynamic	dynamic	Palabra clave
void	void	Palabra clave
boolean	boolean	Palabra clave
char	char	Palabra clave
int	int	Palabra clave
String	String	Palabra clave
if	if	Palabra clave
else	else	Palabra clave
while	while	Palabra clave
for	for	Palabra clave
return	return	Palabra clave
this	this	Palabra clave
new	new	Palabra clave
id	[a-z,A-Z]|_ ([a-zA-Z] | [0-9] | _)*	Identificador
intLiteral	[1-9][0-9]* | 0	Literal Entero
charLiteral	‘ REF_CH ‘	Literal Caracter
StringLiteral	“ REF_STR* “	Literal String
booleanLiteral	true | false	Literal Booleano
null	null	Literal Nulo
(	(	Puntuación
)	)	Puntuación
{	{	Puntuación
}	}	Puntuación
;	;	Puntuación
,	,	Puntuación
.	.	Puntuación
>	>	Operador
<	<	Operador
!	!	Operador
==	==	Operador
>=	>=	Operador
<=	<=	Operador

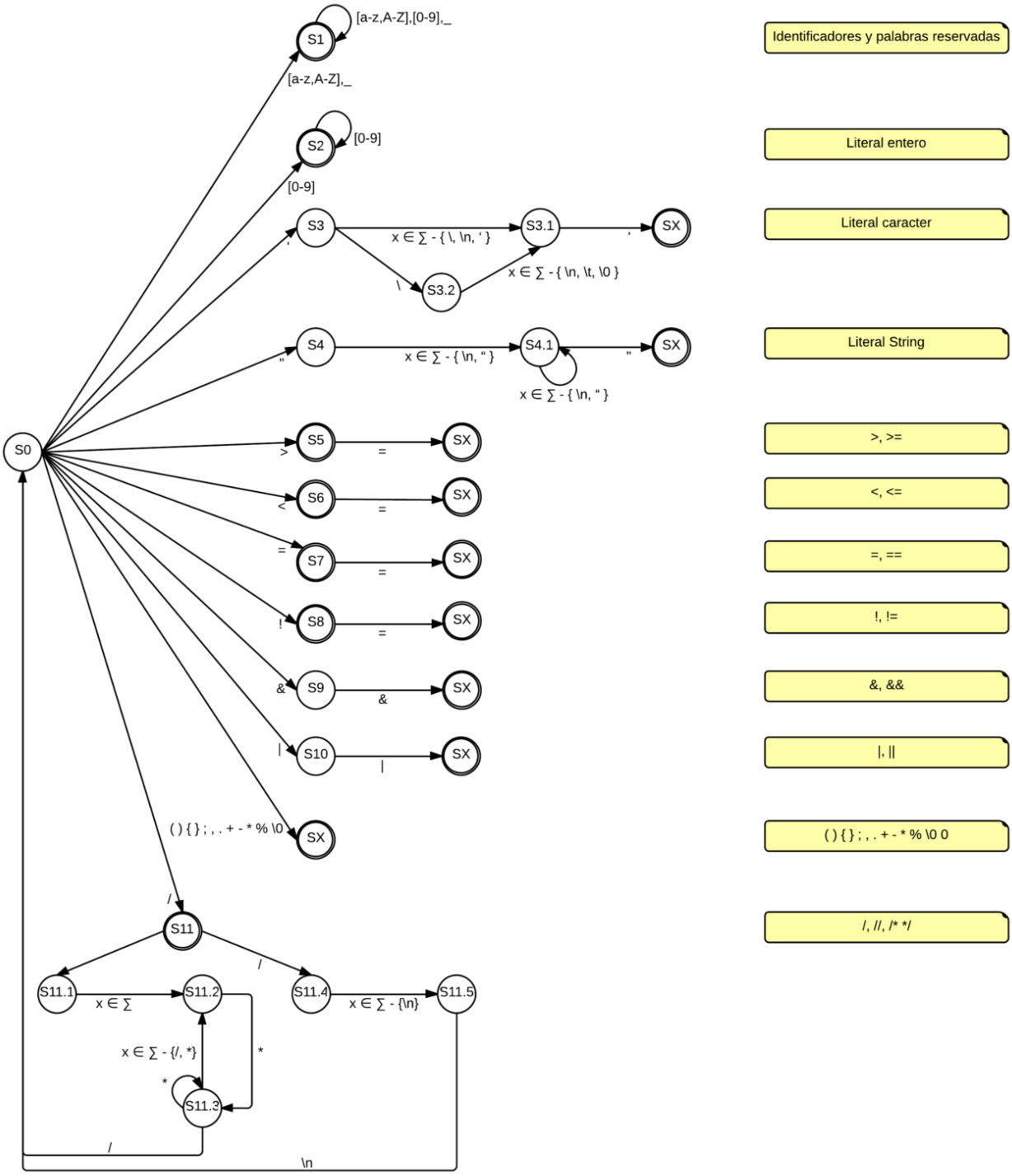
!=	!=	Operador
+	+	Operador
-	-	Operador
*	*	Operador
/	/	Operador
&&	&&	Operador
||	||	Operador
%	%	Operador
=	=	Asignación

REF_CH: x ∈ ∑ - { \, \n, ‘ } | \x ∈ ∑ - { ‘\n’, ‘\t’, ‘\0’ } | \t | \n
REF_STR: x ∈ ∑ - { \n, “ }

Implementación

Funcionalidad principal

Cada token presentado en la tabla anterior puede ser representado por un autómata que luego servirá para la implementación del analizador léxico.



Patrón encontrado	Acción
[a-z, A-Z],_	Cambia al estado S1. De ser una cadena válida se tratará de un identificador o de una palabra reservada.
[0-9]	Cambia al estado S2. De ser una cadena válida se tratará de un entero.
‘	Cambia al estado S3. De ser una cadena válida se tratará de un caracter.
“	Cambia al estado S4. De ser una cadena válida se tratará de un String.
>	Cambia al estado S5. Este caracter puede representar al token ‘>’ o a ‘>=’.
<	Cambia al estado S6. Este caracter puede representar al token ‘<’ o a ‘<=’.
=	Cambia al estado S7. Este caracter puede representar al token ‘=’ o a ‘==’
!	Cambia al estado S8. Este caracter puede representar al token ‘!’ o a ‘!=’.
&	Cambia al estado S9. Este caracter puede representar al token ‘&’ o a ‘&&’.
|	Cambia al estado S10. Este caracter puede representar al token ‘|’ o a ‘||’.
/	Cambia al estado S11. Este caracter puede representar al token ‘/’, a ‘//’ o a ‘/* */’ [*].
( ) { } ; , . + - * % \0	Se retorna el token asociado. \0, el fin de archivo, también es considerado como token.

Los espacios y tabs son ignorados. De encontrar cualquier otro caracter que no sea parte del alfabeto ocurrirá un error.

[*] //, /* y */ no son tokens, pero forman parte de la entrada esperada.

S0 - Estado inicial: A partir de este estado se procede a derivar el procesamiento a diferentes subrutinas según el primer caracter que sea leído. Posibles entradas válidas:

S1 - Identificadores y palabras reservadas: Al llegar a este estado ya se ha consumido una letra [a-z,A-Z] o un ‘_’. Se sigue leyendo del buffer y concatenando al lexema actual mientras que los caracteres consumidos pertenezcan al conjunto de caracteres válidos para un identificador (es decir, [a-z,A-Z], [0-9] ,_).
En caso de encontrar un caracter que no pertenece al conjunto de caracteres válidos para S1, se determina si el lexema formado es una palabra reservada o no, se retorna el token correspondiente y se retrocede el puntero del buffer de entrada.

S2 - Literal entero: Al llegar a este estado ya se ha consumido un dígito. Se sigue leyendo del buffer y concatenando al lexema actual mientras que los caracteres consumidos pertenezcan al conjunto de

caracteres válidos para un literal entero (es decir, [0-9]).
En caso de encontrar un caracter que no pertenece al conjunto de caracteres válidos para S2, se retorna el token correspondiente y se retrocede el puntero del buffer de entrada.

S3 - Literal caracter: Al llegar a este estado ya se ha consumido un ‘. Se lee del buffer el próximo caracter:
- Si el caracter consumido pertenece al conjunto de caracteres imprimibles (es decir, si x es el caracter consumido y x ∈ ∑ - { \, \n, ‘ }), cambia al estado S3.1.
- Si el caracter consumido corresponde a \, cambia al estado S3.2.
S3.1 - Se lee del buffer el próximo caracter. Si el caracter corresponde a ’, se retorna el token correspondiente.
S3.2 - Se lee del buffer el próximo caracter. Si el caracter pertenece al conjunto de caracteres imprimibles (es decir, si x es el caracter consumido y x ∈ ∑ - { \, \n, ‘ }), cambia al estado S3.1.

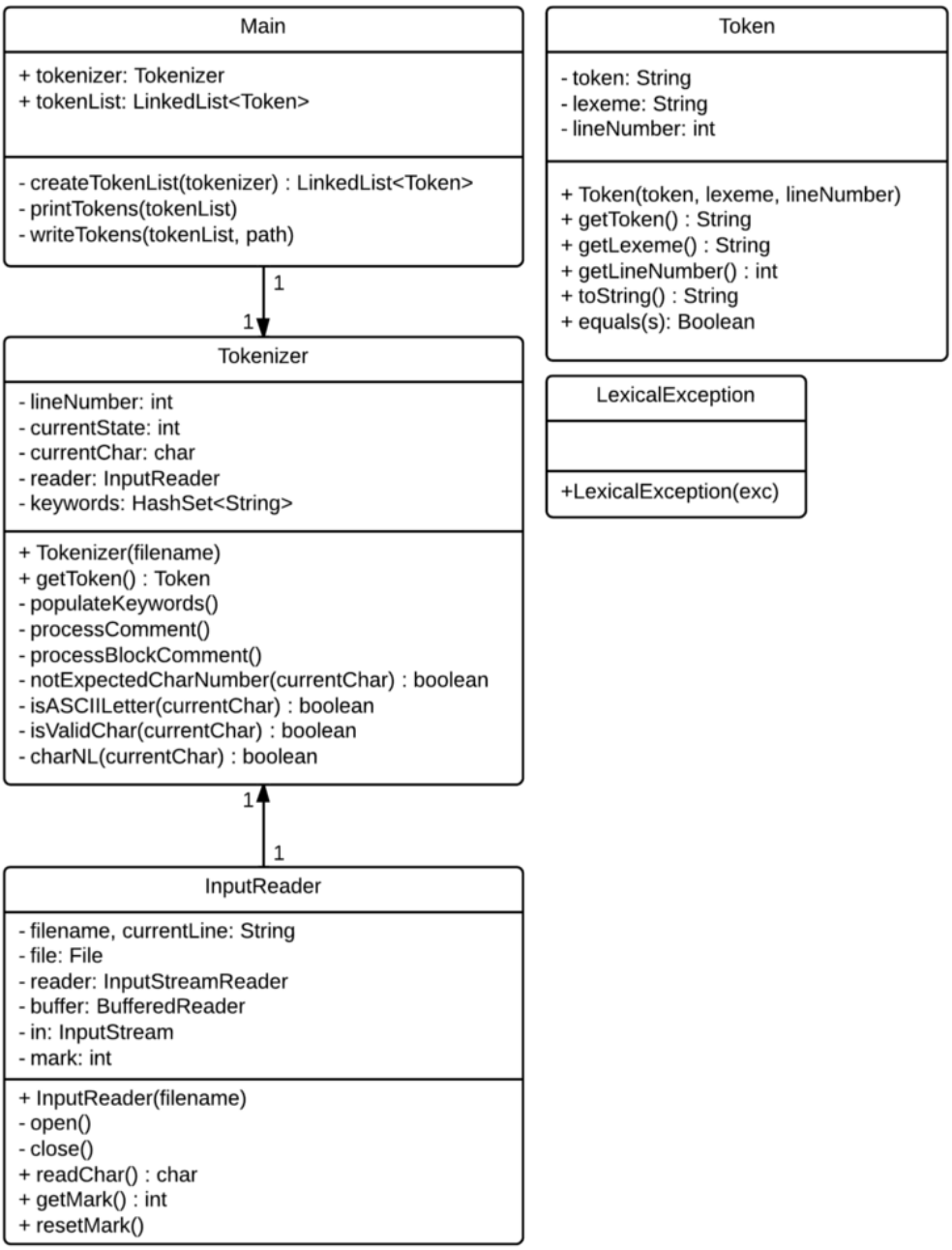
S4 - Literal String: Al llegar a este estado ya se ha consumido un “. Se lee del buffer el próximo caracter:
- Si el caracter consumido pertenece al conjunto de caracteres válidos (es decir, si x es el caracter consumido y x ∈ ∑ - { \n, “ }), cambia al estado S4.1.
S4.1 - Se sigue leyendo del buffer y concatenando al lexema actual mientras que los caracteres consumidos pertenezcan al conjunto de caracteres válido (es decir, si x es el caracter consumido y x ∈ ∑ - { \n, “ }). Si el caracter corresponde a “, se retorna el token correspondiente.

S5 a S10 - >, >=, <, <=, =, ==, !, !=, &, &&, | y ||: En todos estos casos puede darse el caso de que sea un token u otro. Dependiendo del próximo caracter leído se retornara el token correspondiente.

S11 - /, comentario de una línea o apertura de un bloque de comentarios: Si el escáner reconoce el caracter / pueden darse tres opciones:
- O bien es el operador división, para el cual se retornará el token correspondiente.
- O bien es un comentario de una línea, el cual se procederá a consumir y a continuar con la tokenización.
- O bien es la apertura de un bloque de comentarios, el cual se procederá a consumir hasta encontrar el cierre del mismo y a continuar con la tokenización.

Para cualquier otro caracter x tal que x ∈ ∑, se retornará el token correspondiente.

Diagrama de clases



Main - Clase principal del programa. Puede usarse a través de línea de comandos con uno o dos parámetros.
Si se indica un argumento se considerará que se trata del nombre del archivo con el código fuente a procesar y se mostrará por pantalla, si es que no encuentra un error, el listado de tokens generado.
Si se indican dos argumentos se considerará que el primero se trata del nombre del archivo con el código fuente a procesar y el segundo de un archivo de salida en el que se escribirá, si es que no encuentra un error, el listado de tokens generado.

Tokenizator - Clase encargada de llevar adelante el proceso de tokenización. Aquí se implementa el autómata presentado en la sección anterior. Cuenta con métodos adicionales para saltear comentarios simples y comentarios en bloque.

InputReader - *Handler* del archivo. Se encarga de abrir, leer y retornar los caracteres leídos a la clase Tokenizator y, posteriormente, cerrar el archivo.
Por cuestiones de eficiencia se implementó con un buffer BufferedReader el cual procesa el archivo línea por línea.
Al realizarlo de esta manera nos aseguramos de que el encoding quede encapsulado dentro de la clase y que el analizador léxico funcione correctamente para diferentes plataformas.

Token - Representación lógica de un token. Cada token tiene asociado un nombre, un lexema y el número de línea en el que se encontró.

LexicalException - Manejo de excepciones para el analizador léxico.

Decisiones de diseño

- El analizador léxico identifica 3 tipos de errores
  - Lexemas malformados.
  - Símbolos/caracteres extraños (es decir, no pertenecientes al lenguaje).
  - Comentarios mal cerrados (i.e. '/*' que carece del '*/').

- En nuestro diseño no consideramos la existencia de palabras prohibidas por cuestiones de eficiencia en el uso del espacio. Es decir, private (es una palabra reservada de Java) podrá ser utilizado como identificador.

  Un código como el que sigue es totalmente válido:

  ```
  public int private;
  private = 1;
  ```

  En este caso, `private` es un entero público que fue inicializado con el valor 1.

- De procesar una cadena que sea reconocida como un entero seguido inmediatamente de un identificador se considerará que es un número mal formado. E.g. “123hola” es considerado error. Los tests fail/ numero_mal_formado02.java y fail/numeros_y_cadenas.java controlan estos casos.
  No ocurrirá lo mismo para otras cadenas en las que un entero está seguido de otros símbolos, en las cuales se procederá a devolver los tokens correspondientes. E.g. “123{“hola”}”, “123[“ o “123{“.

- Dentro del conjunto de caracteres consideramos como error las secuencias ‘\<tab>’, ‘\<salto>’ donde <tab> y <salto> son los valores de un tab y un salto de línea. Los tests fail/caracter_no_soportado05.java y fail/caracter_no_soportado06.java controlan estos casos.

Batería de tests y manejo de errores

La batería de tests está compuesta de un conjuntos de tests exitosos y de un conjunto de tests que deben fallar. Se incluye un script en Bash para correr todos los tests de una sola pasada (run_tests.sh).

Tests que deben terminar exitosamente

- *archivo_vacio.java*
Controla la validez de un programa completamente vacío.

- *caracteres.java*
Controla la asignación de diferentes caracteres ASCII a una variable.

- *comentario_multiple_dentro_cadena.java*
Controla que el comienzo de un comentario dentro de una cadena no se considere como un comentario efectivo.

- *digitos.java*
Controla la asignación de diferentes enteros a una variable.

- *operadores.java*
Controla el uso de diferentes operadores válidos.

- *programa_generico_exitoso01.java*
Controla la validez de un programa con diferentes variaciones sintácticas del lenguaje.

- *programa_generico_exitoso02.java*
Controla la validez de un programa con diferentes variaciones sintácticas del lenguaje.

Tests que deben fallar

- *cadena_con_salto_linea.java*
Falla en la línea (3): String s = "salto de linea
Ocurre por incluir un salto de línea antes de cerrar el comentario.
Muestra el mensaje de error: “Cadena mal formada (“salto de linea
)."

- *cadena_sin_cerrar.java*
Falla en la línea (3): String s = "
Ocurre al haber comillas dobles que indican el comienzo de una cadena pero no aparecen las comillas dobles de cierre en esa línea.
Muestra el mensaje de error: “Cadena mal formada (“
).”.

- *caracter_no_soportado01.java*
Falla en la línea (3): ~
Ocurre por escribir en el código un caracter no soportado.
Muestra el mensaje de error: “Caracter no soportado (~)”.

- *caracter_no_soportado02.java*

Falla en la línea (3): ?
Ocurre por escribir en el código un caracter no soportado.
Muestra el mensaje de error: “Caracter no soportado (?)”.

- *caracter_no_soportado03.java*
Falla en la línea (3): :
Ocurre por escribir en el código un caracter no soportado.
Muestra el mensaje de error: “Caracter no soportado (:)”.

- *caracter_no_soportado04.java*
Falla en la línea (3): $
Ocurre por escribir en el código un caracter no soportado.
Muestra el mensaje de error: “Caracter no soportado ($)”.

- *caracter_no_soportado05.java*
Falla en la línea (2): c= ' \ '
Ocurre por escribir en el código un caracter no soportado.
Muestra el mensaje de error: “Caracter no soportado ( )”.

- *caracter_no_soportado06.java*
Falla en la línea (3): c=’
‘
Ocurre por escribir en el código un caracter no soportado.
Muestra el mensaje de error: “Caracter no soportado (
”.
- *caracter_mal_formado01.java*
Falla en la línea (3): c=’\\n‘
Ocurre por escribir en el código un caracter no soportado.
Muestra el mensaje de error: “Caracter mal formado (‘\\n).”.

- *caracter_mal_formado02.java*
Falla en la línea (3): c=’\\t‘
Ocurre por escribir en el código un caracter no soportado.
Muestra el mensaje de error: “Caracter mal formado (‘\\t).”.

- *caracter_sin_cerrar.java*
Falla en la línea (3): char c = 'a
Ocurre al haber comillas simples que indican el comienzo de una cadena pero no aparecen las comillas simples de cierre en esa línea.
Muestra el mensaje de error: “Caracter mal formado (‘a).”.

- *comentario_multilinea_sin_cerrar.java*
Falla en la línea (6): EOF
Ocurre al haber un /* que indica el comienzo de una comentario multilínea pero no aparecen su respectivo */ en ninguna parte del resto del archivo.
Muestra el mensaje de error: "El bloque de comentario no está cerrado y se alcanzó el fin de archivo."

- *numero_mal_formado01.java*
Falla en la linea (3): int n1 = 00000777;
Ocurre cuando los un número está compuesto de dígitos 0 al principio.

Muestra el mensaje de error: “Número mal formado. Un número no puede empezar con 0 (00).”

- *numero_mal_formado02.java*
Falla en la línea (3): 123hola = 100;
Ocurre cuando un número está compuesto de letras.
Muestra el mensaje de error: “Número mal formado (123h).”

- *numeros_y_cadenas.java*
Falla en la línea (1): hola123 123hola;
Ocurre cuando un número está compuesto de letras.
Muestra el mensaje de error: “Número mal formado (123h).”

- *operador_prohibido01.java*
Falla en la línea (3): boolean b = true & false;
Ocurre cuando se usa el operador prohibido &.
Muestra el mensaje de error: “Operador no soportado (&)”

- *operador_prohibido02. java*
Falla en la línea (3): boolean b = true | false;
Ocurre cuando se usa el operador prohibido |.
Muestra el mensaje de error: “Operador no soportado (|)”

Analizador sintáctico

Proceso de transformación de la gramática de MiniJava

0. Gramática inicial

<Inicial> → <Clase>+

<Clase> → **class** identificador <Herencia>? **{** <Miembro>* **}**

<Herencia> → **extends** identificador

<Miembro> → <Atributo> | <Ctor> | <Metodo>

<Atributo> → **var** <Tipo> <ListaDeVars> **;**

<Metodo> → <ModMetodo> <TipoMetodo> identificador <ArgsFormales> <VarsLocales> <Bloque>

<Ctor> → identificador <ArgsFormales> <VarsLocales> <Bloque>

<ArgsFormales> → **(** <ListaArgsFormales>? **)**

<ListaArgsFormales> → <ArgFormal>
<ListaArgsFormales> → <ArgFormal> **,** <ListaArgsFormales>

<ArgFormal> → <Tipo> identificador

<VarsLocales> → <Atributo>*

<ModMetodo> → **static** | **dynamic**

<TipoMetodo> → <Tipo> | **void**

<Tipo> → <TipoPrimitivo> | identificador

<TipoPrimitivo> → **boolean** | **char** | **int** | **String**

<ListaDecVars> → identificador
<ListaDecVars> → identificador **,** <ListaDecVars>

<Bloque> → **{** <Sentencia>* **}**

<Sentencia> → **;**
<Sentencia> → <Asignacion> **;**
<Sentencia> → <SentenciaSimple> **;**
<Sentencia> → **if (** <Expresion> **)** <Sentencia>
<Sentencia> → **if (** <Expresion> **)** <Sentencia> **else** <Sentencia>
<Sentencia> → **while (** <Expresion> **)** <Sentencia>
<Sentencia> → **for (** <Asignacion> **;** <Expresion> **;** <Expresion> **)** <Sentencia>
<Sentencia> → <Bloque>
<Sentencia> → **return** <Expresion>? **;**

<Asignacion> → identificador **=** <Expresion>

<SentenciaSimple> → **(** <Expresion> **)**

<Expresion> → <Expresion6>

<Expresion6> → <Expresion6> || <Expresion5>

<*Expresion6*> → <*Expresion5*>

<*Expresion5*> → <*Expresion5*> && <*Expresion4*>
<*Expresion5*> → <*Expresion4*>

<*Expresion4*> → <*Expresion4*> <*Operador4*> <*Expresion3*>
<*Expresion4*> → <*Expresion3*>

<*Expresion3*> → <*Expresion3*> <*Operador3*> <*Expresion2*>
<*Expresion3*> → <*Expresion2*>

<*Expresion2*> → <*Expresion2*> <*Operador2*> <*Expresion1*>
<*Expresion2*> → <*Expresion1*>

<*Expresion1*> → <*Expresion1*> <*Operador1*> <*Expresion0*>
<*Expresion1*> → <*Expresion0*>

<*Expresion0*> → <*OperadorUnario*> <*Primario*>
<*Expresion0*> → <*Primario*>

<*OperadorUnario*> → **!** | **+** | **-**

<*Operador1*> → ***** | **/** | **%**
<*Operador2*> → **+** | **-**
<*Operador3*> → **<** | **>** | **>=** | **<=**
<*Operador4*> → **==** | **!=**

<*Primario*> → **this**
<*Primario*> → <*Literal*>
<*Primario*> → **(** <*Expresion*> **)** <*Llamada*>*
<*Primario*> → *identificador* <*Llamada*>*
<*Primario*> → **new** *identificador* <*ArgsActuales*> <*Llamada*>*
<*Primario*> → *identificador* <*ArgsActuales*> <*Llamada*>*

<*Llamada*> → **.** *identificador* <*ArgsActuales*>

<*Literal*> → **null** | **true** | **false** | intLiteral | charLiteral | stringLiteral

<*ArgsActuales*> → **(** <*ListaExps*>? **)**

<*ListaExps*> → <*Expresion*>
<*ListaExps*> → <*Expresion*> **,** <*ListaExps*>

1. Eliminación de elementos de la gramática EBNF (BNF Extendida)

Los símbolos que pertenecen a la gramática EBNF y debemos eliminar de nuestra gramática son * (0 o más veces), ? (0 o 1 vez) y + (1 o más veces). Los eliminamos agregando las reglas necesarias (en color azul) para mantener la sintaxis que estos definen.

```
<Inicial> → <Clase> <ListaClases>

<ListaClases> → <Clase> <ListaClases>
<ListaClases> → λ

<Clase> → class identificador <Herencia> { <ListaMiembros> }

<Herencia> → extends identificador
<Herencia> → λ

<Miembro> → <Atributo> | <Ctor> | <Metodo>

<ListaMiembros> → <Miembro> <ListaMiembros>
<ListaMiembros> → λ

<Atributo> → var <Tipo> <ListaDecVars> ;

<Metodo>  →  <ModMetodo>  <TipoMetodo>  identificador  <ArgsFormales>  <VarsLocales>
<Bloque>

<Ctor> → identificador <ArgsFormales> <VarsLocales> <Bloque>

<ArgsFormales> → ( <ListaArgsFormales> ) | ( )

<ListaArgsFormales> → <ArgFormal> , <ListaArgsFormales>
<ListaArgsFormales> → <ArgFormal>

<ArgFormal> → <Tipo> identificador

<VarsLocales> → <ListaAtributos>

<ListaAtributos> → <Atributo> <ListaAtributos>
<ListaAtributos> →  λ

<ModMetodo> → static | dynamic

<TipoMetodo> → <Tipo> | void

<Tipo> → <TipoPrimitivo> | identificador

<TipoPrimitivo> → boolean | char | int | String

<ListaDecVars> → identificador
<ListaDecVars> → identificador , <ListaDecVars>

<Bloque> → { <ListaSentencias> }

<ListaSentencias> → <Sentencia> <ListaSentencias>
<ListaSentencias> →  λ

<Sentencia> → ;
<Sentencia> → <Asignacion> ;
<Sentencia> → <SentenciaSimple> ;
<Sentencia> → if ( <Expresion> ) <Sentencia>
<Sentencia> → if ( <Expresion> ) <Sentencia> else <Sentencia>
<Sentencia> → while ( <Expresion> ) <Sentencia>
<Sentencia> → for ( <Asignacion> ; <Expresion> ; <Expresion> ) <Sentencia>
<Sentencia> → <Bloque>
```

<Sentencia> → **return** <Expresion> **;** | **return ;**

<Asignacion> → *identificador* **=** <Expresion>

<SentenciaSimple> → **(** <Expresion> **)**

<Expresion> → <Expresion6>

<Expresion6> → <Expresion6> || <Expresion5>
<Expresion6> → <Expresion5>

<Expresion5> → <Expresion5> && <Expresion4>
<Expresion5> → <Expresion4>

<Expresion4> → <Expresion4> <Operador4> <Expresion3>
<Expresion4> → <Expresion3>

<Expresion3> → <Expresion3> <Operador3> <Expresion2>
<Expresion3> → <Expresion2>

<Expresion2> → <Expresion2> <Operador2> <Expresion1>
<Expresion2> → <Expresion1>

<Expresion1> → <Expresion1> <Operador1> <Expresion0>
<Expresion1> → <Expresion0>

<Expresion0> → <OperadorUnario> <Primario>
<Expresion0> → <Primario>

<OperadorUnario> → **!** | **+** | **-**

<Operador1> → ***** | **/** | **%**
<Operador2> → **+** | **-**
<Operador3> → **<** | **>** | **>=** | **<=**
<Operador4> → **==** | **!=**

<Primario> → **this**
<Primario> → <Literal>
<Primario> → **(** <Expresion> **)** <ListaLlamadas>
<Primario> → *identificador* <ListaLlamadas>
<Primario> → **new** *identificador* <ArgsActuales> <ListaLlamadas>
<Primario> → *identificador* <ArgsActuales> <ListaLlamadas>

<ListaLlamadas> → <Llamada> <ListaLlamadas>
<ListaLlamadas> → λ

<Llamada> → **.** *identificador* <ArgsActuales>

<Literal> → **null** | **true** | **false** | *intLiteral* | *charLiteral* | *stringLiteral*

<ArgsActuales> → **(** <ListaExps> **)** | **( )**

<ListaExps> → <Expresion>
<ListaExps> → <Expresion> **,** <ListaExps>

2. Eliminación de recursión a izquierda

Una gramática es recursiva a izquierda si podemos encontrar algún no-terminal A que eventualmente derive a una forma sentencial cuyo símbolo a la izquierda es sí mismo.

La recursión a izquierda puede clasificarse de dos formas: recursión a izquierda directa (2.1) y recursión a izquierda indirecta (2.2).

Hablamos de recursión a izquierda directa cuando existen reglas con la forma

$$A \rightarrow A\alpha \mid \beta$$

donde α y β son secuencias de no-terminales y terminales, y β no empieza con A. Por ejemplo, la regla <*Expresion*> → <*Expresion*> + <*Termino*>, es recursiva a izquierda directa.

Por otro lado, se considera recursión a izquierda indirecta cuando en la gramática vemos que para un conjunto de no-terminales $A_0, A_1, \ldots A_n$ existe un conjunto de reglas con la forma

$$\begin{array}{c} A_0 \rightarrow A_1\alpha_1 \mid \ldots \\ A_1 \rightarrow A_2\alpha_2 \mid \ldots \\ \ldots \\ A_n \rightarrow A_0\alpha_{n+1} \mid \ldots \end{array}$$

donde $\alpha_1, \alpha_2, \ldots, \alpha_n$ son secuencias de no-terminales y terminales.

La forma más simple de recursión a izquierda indirecta podría ser definida como,

$$\begin{array}{c} A \rightarrow B\alpha \mid C \\ B \rightarrow A\beta \mid D \end{array}$$

generando la derivación $A \vdash B\alpha \vdash A\alpha\beta \vdash \ldots$

2.1 Recursión a izquierda directa

La única parte de la gramática en donde se presenta recursión a izquierda directa es en las reglas utilizadas para modelar la precedencia de los operadores. Se detallan a continuación dichas reglas (en verde) seguidas de la reescritura correspondiente. Se resalta en un color más oscuro en donde se detecta la recursión.

```
<Expresion6> → <Expresion6> || <Expresion5>          // Convención
<Expresion6> → <Expresion5>                  // Todas las reglas reescritas
                                             // se identifican con
<Expresion6> → <Expresion5> <Expresion6_>            // el caracter '_' al final
<Expresion6_> → || <Expresion5> <Expresion6_>
<Expresion6_> → λ

<Expresion5> → <Expresion5> && <Expresion4>
<Expresion5> → <Expresion4>

<Expresion5> → <Expresion4> <Expresion5_>
<Expresion5_> → && <Expresion4> <Expresion5_>
<Expresion5_> → λ

<Expresion4> → <Expresion4> <Operador4> <Expresion3>
```

```
<Expresion4> → <Expresion3>

<Expresion4> → <Expresion3> <Expresion4_>
<Expresion4_> → <Operador4> <Expresion3> <Expresion4_>
<Expresion4_> → λ

<Expresion3> → <Expresion3> <Operador3> <Expresion2>
<Expresion3> → <Expresion2>

<Expresion3> → <Expresion2> <Expresion3_>
<Expresion3_> → <Operador3> <Expresion2> <Expresion3_>
<Expresion3_> → λ

<Expresion2> → <Expresion2> <Operador2> <Expresion1>
<Expresion2> → <Expresion1>

<Expresion2> → <Expresion1> <Expresion2_>
<Expresion2_> → <Operador2> <Expresion1> <Expresion2_>
<Expresion2_> → λ

<Expresion1> → <Expresion1> <Operador1> <Expresion0>
<Expresion1> → <Expresion0>

<Expresion1> → <Expresion0> <Expresion1_>
<Expresion1_> → <Operador1> <Expresion0> <Expresion1_>
<Expresion1_> → λ
```

2.2 Recursión a izquierda indirecta

En la gramática obtenida no se encuentra ninguna regla que produzca una recursión a izquierda indirecta.

3. Factorización

Una gramática está factorizada si no existen producciones alternativas de un mismo no-terminal A que empiezan de la misma forma. Si este no fuera el caso, al llevar adelante el proceso de análisis sintáctico no se sabrá por cuál de ellas seguir.

Para solucionar esto debemos reescribir las reglas de A de modo que se retrase la decisión hasta haber analizado lo suficiente de la entrada y ser capaces de elegir la opción correcta.

Se detallan a continuación dichas reglas (en rojo) seguidas de la reescritura correspondiente. Se resalta en un color más oscuro en donde se detecta el no determinismo.

```
<ArgsFormales> → ( <ListaArgsFormales> ) | ( )

<ArgsFormales> → ( <ArgsFormales_>
<ArgsFormales_> → <ListaArgsFormales> )
<ArgsFormales_> → )

<ListaArgsFormales> → <ArgFormal>
<ListaArgsFormales> → <ArgFormal> , <ListaArgsFormales>

<ListaArgsFormales> → <ArgFormal> <ListaArgsFormales_>
<ListaArgsFormales_> → , <ListaArgsFormales>
<ListaArgsFormales_> → λ

<ListaDecVars> → identificador
<ListaDecVars> → identificador , <ListaDecVars>

<ListaDecVars> → identificador <ListaDecVars_>
<ListaDecVars_> → , <ListaDecVars>
<ListaDecVars_> → λ

<Sentencia> → if ( <Expresion> ) <Sentencia>
<Sentencia> → if ( <Expresion> ) <Sentencia> else <Sentencia>

<Sentencia> → if ( <Expresion> ) <Sentencia> <Sentencia_>
<Sentencia_> → else <Sentencia>
<Sentencia_> → λ

<Sentencia> → return <Expresion> | return ;

/* El nombre Sentencia_ (un guión bajo) fue usado para la reescritura de otra regla,
por lo que usamos Sentencia__ (dos guiones bajos) para mantener la convención de
reescrituras. */

<Sentencia> → return <Sentencia__> ;
<Sentencia__> → <Expresion>
<Sentencia__> → λ

<Primario> → identificador <ListaLlamadas>
<Primario> → identificador <ArgsActuales> <ListaLlamadas>

<Primario> → identificador <ListaLlamadas_>
<ListaLlamadas_> → <ListaLlamadas>
<ListaLlamadas_> → <ArgsActuales> <ListaLlamadas>

<ArgsActuales> → ( <ListaExps> ) | ( )

<ArgsActuales> → ( <ArgsActuales_>
<ArgsActuales_> → <ListaExps> )
<ArgsActuales_> → )
```

```
<ListaExps> → <Expresion>
<ListaExps> → <Expresion> , <ListaExps>

<ListaExps> → <Expresion> <ListaExps_>
<ListaExps_> → , <ListaExps>
<ListaExps_> → λ
```

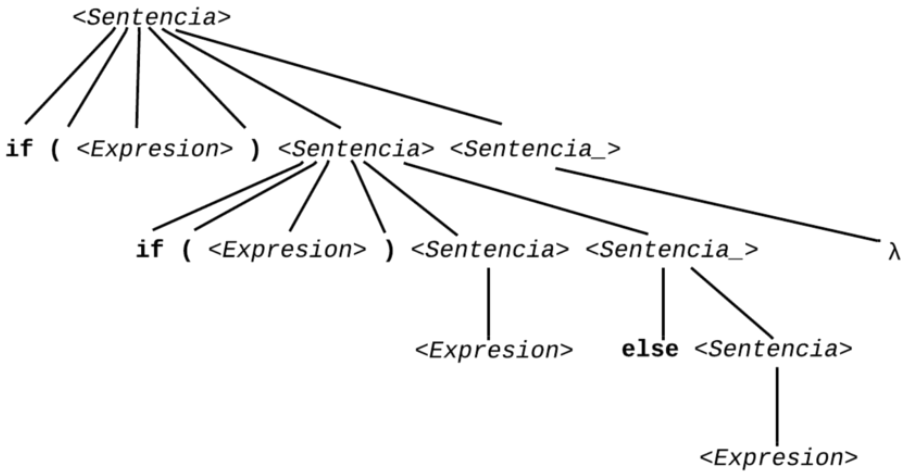
4. Ambigüedades

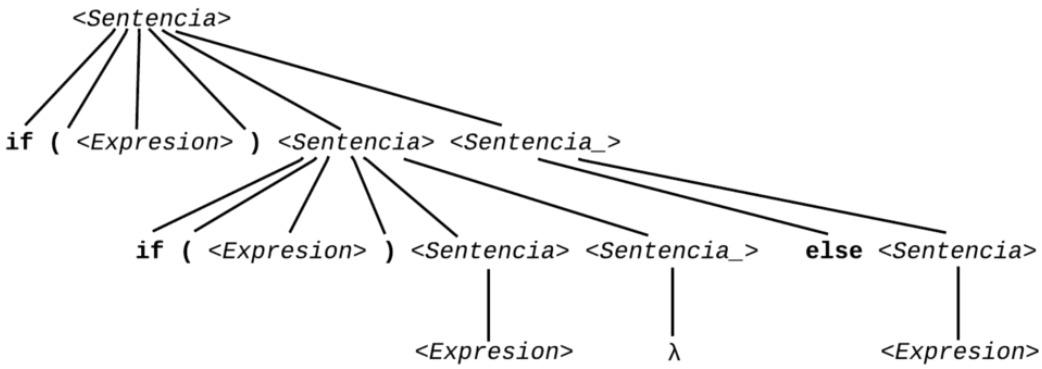
La gramática original hasta este punto es ambigua ya que existen dos árboles de derivación diferentes para una misma cadena del lenguaje.

Por ejemplo, la cadena

if (<Expresion>) if (<Expresion>) <Expresion> else <Expresion>

se puede obtener a partir de los siguientes árboles de derivación.





Esto se debe a que las reglas de producción,

<Sentencia> → **if** (*<Expresion>*) *<Sentencia>* *<Sentencia_>*
<Sentencia_> → **else** *<Sentencia>*

generan ambigüedad.

Es posible encontrar reglas de producción que solucionen este problema, pero de esta forma la gramática se volvería innecesariamente compleja. Por lo tanto consideramos lo más adecuado es adoptar la siguiente convención:
"Cada *else* corresponde al *if* inmediatamente superior que todavía no tenga su correspondiente

else.”

Es decir, para solucionar esta ambigüedad la implementación del analizador sintáctico siempre seguirá el primer árbol de derivación.

5. Gramática resultado

<Inicial> → <Clase> <ListaClases>

<ListaClases> → <Clase> <ListaClases>
<ListaClases> → λ

<Clase> → **class** identificador <Herencia> **{** <ListaMiembros> **}**

<Herencia> → **extends** identificador
<Herencia> → λ

<Miembro> → <Atributo> | <Ctor> | <Metodo>

<ListaMiembros> → <Miembro> <ListaMiembros>
<ListaMiembros> → λ

<Atributo> → **var** <Tipo> <ListaDecVars> **;**

<Metodo> → <ModMetodo> <TipoMetodo> identificador <ArgsFormales> <VarsLocales> <Bloque>

<Ctor> → identificador <ArgsFormales> <VarsLocales> <Bloque>

<ArgsFormales> → **(** <ArgsFormales_>
<ArgsFormales_> → <ListaArgsFormales> **)**
<ArgsFormales_> → **)**

<ListaArgsFormales> → <ArgFormal> <ListaArgsFormales_>
<ListaArgsFormales_> → , <ListaArgsFormales>
<ListaArgsFormales_> → λ

<ArgFormal> → <Tipo> identificador

<VarsLocales> → <ListaAtributos>

<ListaAtributos> → <Atributo> <ListaAtributos>
<ListaAtributos> → λ

<ModMetodo> → **static** | **dynamic**

<TipoMetodo> → <Tipo> | **void**

<Tipo> → <TipoPrimitivo> | identificador

<TipoPrimitivo> → **boolean** | **char** | **int** | **String**

<ListaDecVars> → identificador <ListaDecVars_>
<ListaDecVars_> → , <ListaDecVars>
<ListaDecVars_> → λ

<Bloque> → **{** <ListaSentencias> **}**

<ListaSentencias> → <Sentencia> <ListaSentencias>
<ListaSentencias> → λ

<Sentencia> → **;**
<Sentencia> → <Asignacion> **;**
<Sentencia> → <SentenciaSimple> **;**
<Sentencia> → **if (** <Expresion> **)** <Sentencia> <Sentencia_>
<Sentencia_> → **else** <Sentencia>
<Sentencia_> → λ
<Sentencia> → **while** (<Expresion> **)** <Sentencia>
<Sentencia> → **for (** <Asignacion> **;** <Expresion> **;** <Expresion> **)** <Sentencia>
<Sentencia> → <Bloque>

<Sentencia> → **return** <Sentencia__> **;**
<Sentencia__> → <Expresion>
<Sentencia__> → λ

<Asignacion> → *identificador* **=** <Expresion>

<SentenciaSimple> → **(** <Expresion> **)**

<Expresion> → <Expresion6>

<Expresion6> → <Expresion5> <Expresion6_>

<Expresion6_> → || <Expresion5> <Expresion6_>
<Expresion6_> → λ

<Expresion5> → <Expresion4> <Expresion5_>

<Expresion5_> → && <Expresion4> <Expresion5_>
<Expresion5_> → λ

<Expresion4> → <Expresion3> <Expresion4_>

<Expresion4_> → <Operador4> <Expresion3> <Expresion4_>
<Expresion4_> → λ

<Expresion3> → <Expresion2> <Expresion3_>

<Expresion3_> → <Operador3> <Expresion2> <Expresion3_>
<Expresion3_> → λ

<Expresion2> → <Expresion1> <Expresion2_>

<Expresion2_> → <Operador2> <Expresion1> <Expresion2_>
<Expresion2_> → λ

<Expresion1> → <Expresion0> <Expresion1_>

<Expresion1_> → <Operador1> <Expresion0> <Expresion1_>
<Expresion1_> → λ

<Expresion0> → <OperadorUnario> <Primario>
<Expresion0> → <Primario>

<OperadorUnario> → **!** | **+** | **-**

<Operador1> → ***** | **/** | **%**
<Operador2> → **+** | **-**
<Operador3> → **<** | **>** | **>=** | **<=**
<Operador4> → **==** | **!=**

<Primario> → **this**
<Primario> → <Literal>
<Primario> → **(** <Expresion> **)** <ListaLlamadas>
<Primario> → *identificador* <ListaLlamadas_>

<Primario> → **new** *identificador* <ArgsActuales> <ListaLlamadas>
<Primario> → *identificador* <ArgsActuales> <ListaLlamadas>

<ListaLlamadas> → <Llamada> <ListaLlamadas>
<ListaLlamadas> → λ

<ListaLlamadas_> → <ListaLlamadas>
<ListaLlamadas_> → <ArgsActuales> <ListaLlamadas>

<Llamada> → **.** *identificador* <ArgsActuales>

<*Literal*> → **null** | **true** | **false** | *intLiteral* | *charLiteral* | *stringLiteral*

<*ArgsActuales*> → **(** <*ArgsActuales_*>

<*ArgsActuales_*> → <*ListaExps*> **)**
<*ArgsActuales_*> → **)**

<*ListaExps*> → <*Expresion*> <*ListaExps_*>

<*ListaExps_*> → **,** <*ListaExps*>
<*ListaExps_*> → λ

Análisis: ¿Es una gramática LL(1)?

Un analizador sintáctico es LL(1) si sólo usa un token cuando ve hacia delante de la sentencia para hacer el análisis de sus decisiones. Las gramáticas LL(1) tienen varias propiedades distintivas: ninguna gramática ambigua o recursiva por la izquierda puede ser LL(1).
La gramática de MiniJava sin recursión a izquierda y ya factorizada es ambigua por la forma en la que modela las sentencias **if-then-else**.

Existen dos árboles de derivación diferentes para la cadena:

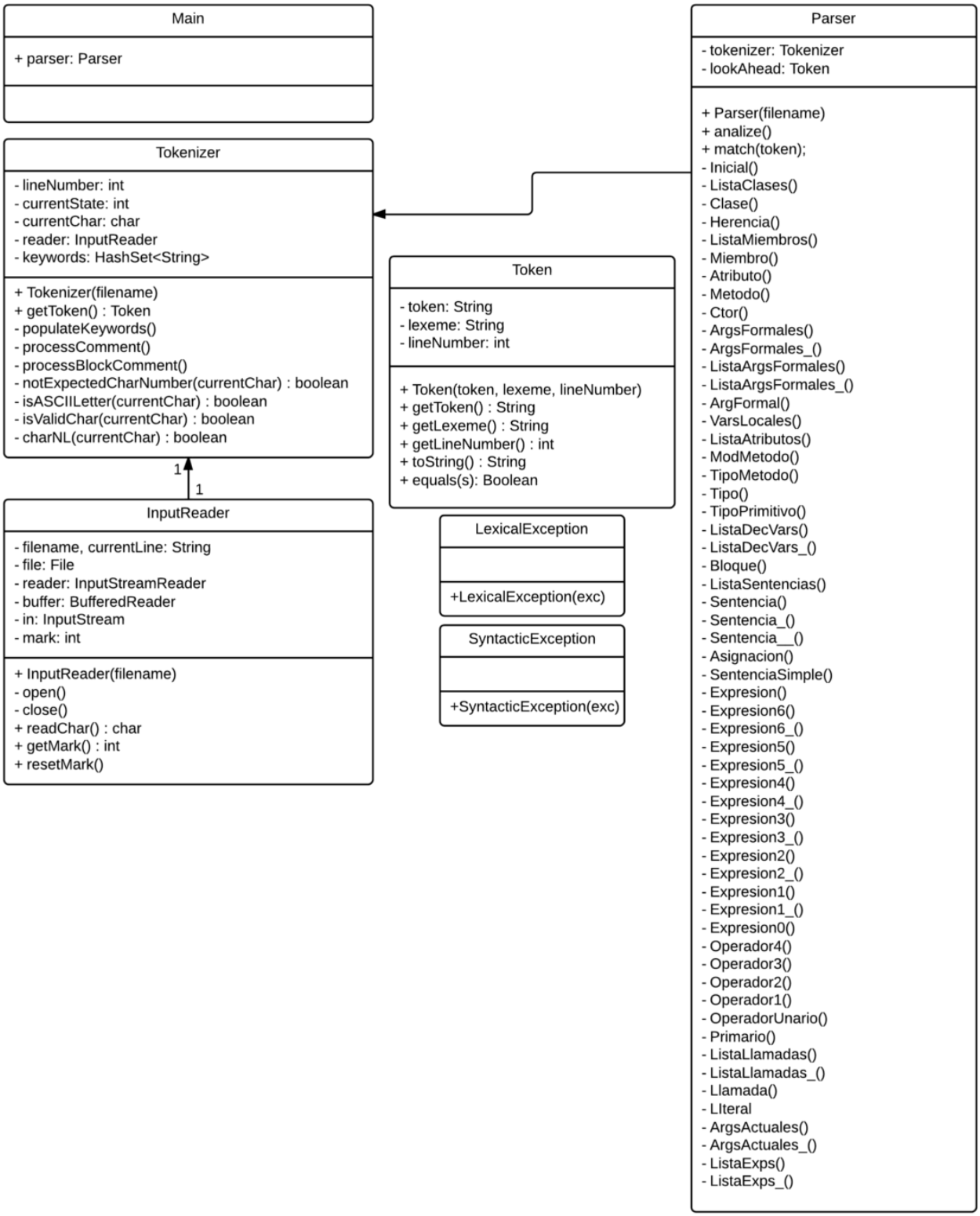
if (*<Expresion>* **)** if **(** *<Expresion>* **)** *<Expresion>* **else** *<Expresion>*

Luego la gramática de MiniJava **no es LL(1)** porque es ambigua.

A pesar de ésto, la ambigüedad se solucionó en la implementación del analizador optando por sólo una de los dos árboles de derivación. Por lo tanto el analizador sintáctico si es LL(1).

Implementación

Diagrama de clases



Main - Clase principal del programa. Puede usarse a través de línea de comandos indicando como argumento el nombre del archivo con el código fuente a procesar. Se mostrará por pantalla si el

análisis sintáctico fue exitoso, y en caso contrario, notificará al usuario con el mensaje con el error correspondiente.

Tokenizer - Clase encargada de llevar adelante el proceso de tokenización. Cuenta con métodos adicionales para saltear comentarios simples y comentarios en bloque.

InputReader - *Handler* del archivo. Se encarga de abrir, leer y retornar los caracteres leídos a la clase Tokenizer y, posteriormente, cerrar el archivo.
Por cuestiones de eficiencia se implementó con un buffer BufferedReader el cual procesa el archivo línea por línea.
Al realizarlo de esta manera nos aseguramos de que el encoding quede encapsulado dentro de la clase y que el analizador léxico funcione correctamente para diferentes plataformas.

Token - Representación lógica de un token. Cada token tiene asociado un nombre, un lexema y el número de línea en el que se encontró.

LexicalException - Manejo de excepciones para el analizador léxico.

Parser - Clase encargada de llevar adelante el análisis sintáctico del código fuente pasado por parámetro. Implementa un analizador sintáctico descendente donde la entrada se analiza de izquierda a derecha con derivación a izquierda.
Se implementa un método por cada no-terminal de la gramática modificada de MiniJava, encargados de llevar un flujo recursivo de análisis correspondiente.. Para esto requiere los tokens generados por la clase Tokenizer.

SyntacticException - Manejo de excepciones para el analizador sintáctico.

Decisiones de diseño

- Para **todos** los nombres de los símbolos no-terminales que surgieron de la transformación de la gramática por factorización o eliminación de la recursión a izquierda, se optó por utilizar el caracter **_** (guión bajo).
  Esto permite que no haya diferencias entre la gramática y la codificación de la misma.
  E.g:
  ```
  private void ListaExps_() throws LexicalException, SyntacticException {
  ```

- Para solucionar la ambigüedad de la gramática generada por las reglas de producción

  <Sentencia> → **if (** *<Expresion>* **)** *<Sentencia>* *<Sentencia_>*
  <Sentencia_> → **else** *<Sentencia>*

  en la implementación del analizador sintáctico se tomó la siguiente convención:

  "Cada *else* corresponde al *if* al inmediatamente anterior que todavía no tenga su correspondiente *else.*"

- La implementación es case sensitive (sensible a las mayúsculas). E.g. no es lo mismo "Class" que "class".

- En el caso de que en una misma sentencia se encuentren dos operadores que tengan el mismo nivel de precedencia, se evaluará de izquierda a derecha.

  Es decir, sea <op> = {<, <=, >, >=}

  En expresiones del tipo

  ```
  X <op> Y <op> ... <op> Z
  ```

  el orden de asociatividad será de izquierda a derecha

  ```
  X <op> Y <op> ... <op> Z  → ((X <op> Y) <op>  ... ) <op> Z
  ```

- El analizador sintáctico captura los siguientes errores

  Sean A y B dos tokens tal que A ≠ B.
  Si el Parser al hacer el match entre dos tokens esperaba el token A y recibe el token B, falla mostrando el error:
  ```
  Línea: [Nro. Línea] - Error sintáctico: Se esperaba: A. Se encontró: B.
  ```

  Si luego del nombre de una clase el Parser no encuentra el comienzo del bloque de la clase { o la palabra reservada *extends*, falla mostrando el error:
  ```
  Línea: [Nro. Línea] - Error sintáctico: Se esperaba el comienzo de bloque de la clase o la especificación de herencia.
  ```

  Si dentro del bloque de una clase el Parser no encuentra las palabras reservadas *var* (la definición de un atributo), un identificador (la definición de un constructor) o *static* o *dynamic* (la definición de un método), falla mostrando el error:
  ```
  Línea: [Nro. Línea] - Error sintáctico: Se esperaba la definición de atributos, constructores o métodos.
  ```

  Si en los argumentos formales de un método el Parser no encuentra el cierre de argumentos) o una coma (nuevo argumento formal), falla mostrando el error:

```
Línea: [Nro. Línea] - Error sintáctico: Se esperaba un nuevo argumento formal o
el cierre de los argumentos formales.
```

Si luego de la especificación de los argumentos formales de un método o constructor el Parser no encuentra el comienzo del bloque del método { o la palabra reservada *var* (definición de variables locales), falla mostrando el error:

```
Línea: [Nro.Línea] - Error sintáctico: Se esperaba el comienzo de bloque del
método o la definición de variables locales.
```

Si luego de la palabra clave *var* (definición de variable local), o luego de las palabras claves *static* y *dynamic* (definición de un método) o en la especificación de un argumento formal el Parser no encuentra un tipo (ya sea un tipo primitivo o no), falla mostrando el error:

```
Línea: [Nro. Línea] - Error sintáctico: Se esperaba un tipo de dato.
```

Si en la lista de variables en la definición de atributos el Parser no encuentra el fin de la lista ; o una coma (nueva variable), falla mostrando el error:

```
Línea: [Nro. Línea] - Error sintáctico: Se esperaba un ;.
```

Si al esperar una sentencia el Parser no encuentra un ; (fin de sentencia), un identificador (comienzo de asignación), un (, un *if*, un *while*, un *for*, un { (comienzo de bloque) o un *return* (retorno de método), falla mostrando el error:

```
Línea: [Nro. Línea] - Error sintáctico: Se esperaba una sentencia.
```

Si al esperar un operador de precedencia 4 el Parser no encuentra == o !=, falla mostrando el error:

```
Línea: [Nro. Línea] - Error sintáctico: Se esperaba == o != .
```

Si al esperar un operador de precedencia 3 el Parser no encuentra <, >, >= o <=, falla mostrando el error:

```
Línea: [Nro.Línea] - Error sintáctico: Se esperaba <, >, >= o <= .
```

Si al esperar un operador de precedencia 2 el Parser no encuentra + o -, falla mostrando el error:

```
Línea: [Nro. Línea] - Error sintáctico: Se esperaba + o - .
```

Si al esperar un operador de precedencia 1 el Parser no encuentra *, /, o %, falla mostrando el error:

```
Línea: [Nro.Línea] - Error sintáctico: Se esperaba *, / o % .
```

Si al esperar un operador unario el Parser no encuentra !, - o +, falla mostrando el error:

```
Línea: [Nro. Línea] - Error sintáctico: Se esperaba !, - o + .
```

Si al esperar un literal el Parser no encuentra un *null*, un *true*, un *false*, un literal entero, un literal carácter o un literal String, falla mostrando el error:

```
Línea: [Nro. Línea] - Error sintáctico: Se esperaba un literal.
```

Batería de tests y manejo de errores

La batería de tests está compuesta de un conjuntos de tests exitosos y de un conjunto de tests que deben fallar. Se incluye un script en Bash para correr todos los tests de una sola pasada (run_tests.sh).

Tests que deben terminar exitosamente

- *clase_atributo.java*
Clase simple con atributos de diferentes tipos.

- *clase_metodo_sentencia_for.java*
Clase simple con un método conteniendo un for.

- *clase_metodo_sentencia_while.java*
Clase simple con un método conteniendo dos whiles anidados.

- *clase_constructor.java*
Clase simple con un constructor.

- *clase_metodo_sentencia_ifthenelse.java*
Clase simple con un método conteniendo un if-then.

- *clase_simple.java*
Clase simple conteniendo un constructor vacío.

- *clase_metodo.java*
Clase simple conteniendo varios métodos de diferentes modos de compilación y tipos.

- *clase_metodo_sentencia_puntoycoma.java*
Clase simple conteniendo un método y sentencia ;

- *clase_vacia.java*
Clase vacía con herencia.

- *clase_metodo_sentencia_asignacion.java*
Clase simple conteniendo un método con una asignación.

- *clase_metodo_sentencia_sentenciasimple.java*
Clase simple conteniendo un método con varias sentencias simples.

- *programa_sintacticamente_valido01.java* y *programa_sintacticamente_valido02.java*

El objetivo de estos test es que el el analizador sintáctico reconozca que esta clase es válida, es decir, que finalice su análisis sin la ocurrencia de un error léxico o sintáctico. Este test abarca los casos más raros de la gramática del lenguaje.

Tests que deben fallar

- *asignacion-primario-caractermalformado.java*
Falla en la línea (7): j='a;
Ocurre al haber comillas simples que indican el comienzo de una cadena pero no aparecen las comillas simples de cierre en esa línea.
Muestra el mensaje de error: Línea: 7 - Error léxico: Caracter mal formado ('a;).

- *asignacion-primario-faltapuntoycoma.java*
Falla en la línea (10): j='a'
Ocurre al encontrar un token de cierre de bloque } en lugar de un ; .
Muestra el mensaje de error: Línea: 10 - Error sintáctico: Se esperaba: ';'. Se encontró: '}'.

- *asignacion-primario-operadores.java*
Falla en la línea (6): n2 = n1 + * n3 - n4 / n5 * n6 % n7;
Ocurre al no encontrar un literal después del operador + .
Muestra el mensaje de error: Línea: 6 - Error sintáctico: Se esperaba un literal.

- *atributo01.java*
Falla en la línea (3): var char c1;c2;c3;
Ocurre por usar ';' para separar los identificadores en lugar de ','. El error se captura en "c2;" ya que "var char c1;" fue procesado como un atributo y ahora se encuentra reconociendo lo que se cree que es un constructor. Como un constructor viene seguido de una lista de argumentos formales, el error será que no se encontró esta lista.
Muestra el mensaje de error: Línea: 3 - Error sintáctico: Se esperaba: '('. Se encontró: ';'.

- *atributo02.java*
Falla en la línea (4): var int b
Ocurre al no encontrar ';' para separar la sentencia de la línea 3 de la 4. Al no encontrar el terminador queda a la espera de la definición de nuevas variables del mismo tipo.
Muestra el mensaje de error: Línea: 4 - Error sintáctico: Se esperaba una variable.

- *atributo-ctor-metodo01.java*
Falla en la línea (22): var char caracter
Ocurre al no encontrar el terminador ';' de la lista de de variables.
Muestra el mensaje de error: Línea: 22 - Error sintáctico: Se esperaba una variable.

- *atributo-ctor-metodo02.java*
Falla en la línea (11): String cadena;
Ocurre al no encontrar un *var* para la declaración de la variable local cadena.
Muestra el mensaje de error: Línea: 11 - Error sintáctico: Se esperaba el comienzo de bloque del metodo o la definicion de variables locales.

- *clase01.java*
Falla en la línea (7): class foo {
Ocurre al no encontrar el cierre del bloque de la clase foo.
Muestra el mensaje de error: Línea: 7 - Error sintáctico: Se esperaba la definicion de atributos, constructores o metodos.

- *clase02.java*
Falla en la línea (1): class class {
Ocurre al encontrar *class* en lugar del identificador de la clase.

Muestra el mensaje de error: Línea: 1 - Error sintáctico: Se esperaba: 'id'. Se encontró: 'class'.

- *for01.java*
Falla en la línea (8): for(n2 + 10; n2 != 20; true&&false%null) {
Ocurre porque el primer argumento del for no es una asignación, sino una expresión.
Muestra el mensaje de error: Línea: 8 - Error sintáctico: Se esperaba: '='. Se encontró: '+'.

- *for02.java*
Falla en la línea (15): for(n1 = null, n1<=54/2, n1||true) {
Ocurre porque los argumentos del for se separaron con , en lugar de ;
Muestra el mensaje de error: Línea: 15 - Error sintáctico: Se esperaba: ';'. Se encontró: ','.

- *herencia01.java*
Falla en la línea (1): class foo extends extends {
Ocurre al encontrar *extends* en lugar del identificador de la superclase.
Muestra el mensaje de error: Línea: 1 - Error sintáctico: Se esperaba: 'id'. Se encontró: 'extends'.

- *herencia02.java*
Falla en la línea (1): class foo extends {
Ocurre al no encontrar el identificador de la superclase.
Muestra el mensaje de error: Línea: 1 - Error sintáctico: Se esperaba: 'id'. Se encontró: '{'.

- *ifelse01.java*
Falla en la línea (15): EOF
Ocurre debido a que uno de los bloques del if-else está mal cerrado. Se continúa procesando hasta que se alcanza fin de línea.
Muestra el mensaje de error: Línea: 15 - Error sintáctico: Se alcanzo EOF durante el analisis sintáctico.

- *ifelse02.java*
Falla en la línea (12): else ;
Ocurre porque dentro de un bloque se espera una lista de sentencias, y else ; no es una sentencia.
Muestra el mensaje de error: Línea: 12 - Error sintáctico: Se esperaba una sentencia.

- *ifelse03.java*
Falla en la línea (9): (prueba.test1().test2(.test3().test4());
Ocurre debido a que se realiza una llamada mal formada.
Muestra el mensaje de error: Línea: 9 - Error sintáctico: Se esperaba un literal.

- *lista_incompleta.java*
Falla en la línea (2): static void test(int a,) {
Ocurre porque en la lista de argumentos formales aparece el terminal ‘,’ y luego de esto se espera un argumento formal.
Muestra el mensaje de error: Línea: 2 - Error sintáctico: Se esperaba un tipo de dato.

- *metodo.java*
Falla en la línea (3): static boolean method1();
Ocurre debido a que luego de la lista de argumentos formales se espera la declaración de variables locales o el comienzo de un bloque, y en este caso se encuentra el terminal ‘;’.
Muestra el mensaje de error: Línea: 3 - Error sintáctico: Se esperaba el comienzo de bloque del metodo o la definicion de variables locales.

- *return-expresion.java*
Falla en la línea (7): return = n1 + n2;
Ocurre porque no puede hacerse una asignación teniendo como lado izquierdo a un return. Es una palabra reservada, no identificador.
Muestra el mensaje de error: Línea: 7 - Error sintáctico: Se esperaba un literal.

- *sentencia-for.java*
Falla en la línea (6): for (j!=20%2>=6*3; j/i){
Ocurre debido a que la sentencia for espera una asignación como primer parámetro. En este caso se encuentra una expresión.
Muestra el mensaje de error: Línea: 6 - Error sintáctico: Se esperaba: '='. Se encontró: '!='.

- *sentenciasimple-expresion01.java*
Falla en la línea (7): (true&&false;
Ocurre porque no se cerró el paréntesis de la expresión.
Muestra el mensaje de error: Línea: 7 - Error sintáctico: Se esperaba: ')'. Se encontró: ';'.

- *sentenciasimple-expresion02.java*
Falla en la línea (6): (objeto ().test.new llamada());
Ocurre porque se realiza una llamada mal formada. Se esperan los parámetros de test y lo que se encuentra es el inicio de una nueva llamada.
Muestra el mensaje de error: Línea: 6 - Error sintáctico: Se esperaba: '('. Se encontró: '.'.

- *sentenciasimple-expresion03.java*
Falla en la línea (7): (n1 < + 3 > n2 == true + "cadena);
Ocurre al haber comillas dobles que indican el comienzo de una cadena pero no aparecen las comillas dobles de cierre en esa línea.
Muestra el mensaje de error: Línea: 7 - Error léxico: Cadena mal formada ("cadena);
).

- *sentenciasimple-expresion04.java*
Falla en la línea (6): (new prueba ('a', "cadena", 2, true, null, n1 + 8 / 2 % 2 * 8).);
Ocurre al encontrar) en lugar del identificador del receptor de la llamada después del punto.
Muestra el mensaje de error: Línea: 6 - Error sintáctico: Se esperaba: 'id'. Se encontró: ')'.

- *sentenciasimple-expresion05.java*
Falla en la línea (6): (objeto (('a',"cadena",1));
Ocurre al encontrar una coma después de la expresión del primer argumento actual, en lugar de el paréntesis necesario para cerrar la expresión.
Muestra el mensaje de error: Línea 6 - Error sintáctico: Se esperaba ‘)’. Se encontró: ‘,’.

- *sentenciasimple-expresion-operadorunario.java*
Falla en la línea (6): !new objeto();
Ocurre ya que esta no es una sentencia correcta, sino una expresión.
Muestra el mensaje de error: Línea 6 - Error sintáctico: Se esperaba una sentencia.

- *sentencia-while.java*
Falla en la línea (8): while (n1=n2){
Ocurre porque la condición del *while* es una asignación en lugar de una expresión.
Muestra el mensaje de error: Línea 8 - Error sintáctico: Se esperaba ‘)’. Se encontró: ‘=’.

- *while01.java*
Falla en la línea (9): while() {
Ocurre porque falta un literal en la condición del *while*.
Muestra el mensaje de error: Línea 9 - Error sintáctico: Se esperaba un literal.

- *while02.java*
Falla en la línea (8): while (n4==n3)
Ocurre porque falta una sentencia después del *while*.
Muestra el mensaje de error: Línea 8 - Error sintáctico: Se esperaba una sentencia.

Analizador semántico

Esquema de Traducción

Definición de atributos

Atributos sintetizados

from_s - Inicializiza el valor de from_h
type_s - Inicializa el valor de type_h
list_s - Lista de variables que luego serán, según sea el caso, argumentos actuales o formales

Atributos heredados

from_h - Determina desde donde se ha llegado a una determinada regla sintáctica. Es necesario para poder diferenciar entre Métodos y Clases, y hacer los controles sobre atributos de forma correcta.
type_h - Determina el tipo de una variable o conjunto de variables, o del retorno de un método.
sent_h - Sentencia heredada (necesario para la creación de *SentenceNode*. Parámetro de entrada para aquellos nodos con más de una sentencia)
expr_h - Expresion heredada (necesario para la creación de *ExpressionNode*. Parámetro de entrada para aquellos nodos con más de una expresión)

Atributos intrínsecos

lexeme - Cadena representando un token
token - Token

Especificación

```
<Inicial> →  { TS.initSymbolTable() } <Clase> <ListaClases>

<ListaClases> → <Clase> <ListaClases>

<ListaClases> → λ

<Clase> → class identificador
        {
                if (TS.getClass(identificador.lexeme) != null) then
                        Error: Ya existe una clase declarada con el nombre "..."
                else {
                        currentClass ← identificador.lexeme
                        TS.addClass(identificador.lexeme)
                }
        }
<Herencia>
        { ListaMiembros.from_h ← "class" }
{ <ListaMiembros>
        { TS.getClass(currentClass).controlDefaultConstructor() }
}

<Herencia> → extends identificador
        {
                TS.getClass(currentClass).setParentList(identificador.lexeme)
                TS.getClass(currentClass).setParent(identificador.lexeme)

                if (TS.controlInheritance(currentClass))
                        Error: Herencia circular
        }

<Herencia> → λ
        {
                TS.getClass(currentClass).setParentList("Object")
                TS.getClass(currentClass).setParent("Object")
        }

<Miembro> → { Atributo.from_h ← Miembro.from_h } <Atributo>
<Miembro> → <Ctor>
<Miembro> → <Metodo>


<ListaMiembros0> →   { Miembro.from_h ← ListaMiembros0.from_h }
                     <Miembro>
                     { ListaMiembros1.from_h ← ListaMiembros0.from_h }
                     <ListaMiembros1>

<ListaMiembros> →    λ

<Atributo> → var <Tipo>
        {
                ListaDecVars.type_h ← Tipo.type_s
                ListaDecVars.from_h ← Atributo.from_h
        }
<ListaDecVars> ;


<Metodo> → <ModMetodo> <TipoMetodo> identificador
        {
                currentMethod ← identificador.lexeme

                if (currentMethod.equals(currentClass)) {
                        Error: El método no puede tener el mismo nombre que la clase.
                }
```

```
                if (TS.getClass(currentClass).getMethod(currentMethod) != null) {
                        Error: Ya existe un método decladado con el nombre “...”
                } else {
                        TS.getClass(currentClass).addMethod(currentMethod,
                        TipoMetodo.type_s, ModMetodo.mod_s)
                }
        }
<ArgsFormales> { VarsLocales.from_h ← “method” }
<VarsLocales> <Bloque>
        {
                Metodo.body_s ← Bloque.tree_s
                TS.getClass(currentClass).getMethod(currentMethod).setBody(Metodo.body_s
        )
        }


<Ctor> → identificador
        {
                currentMethod ← identificador.lexeme
                if (currentClass != currentMethod) {
                        Error: El nombre del constructor no corresponde al nombre de
                        la clase.
                } else {
                        if (TS.getClass(currentClass).getConstructor() != null)) {
                                Error: Ya existe un constructor en la clase.
                        } else {
                                TS.getClass(currentClass).setConstructor(currentMethod)
                        }
                }
        }
<ArgsFormales> { VarsLocales.from_h ← “constructor” } <VarsLocales> <Bloque>
        {
                Ctor.body_s ← Bloque.tree_s
                TS.getClass(currentClass).getMethod(currentMethod).setBody(Ctor.body_s)
        }

<ArgsFormales> → ( <ArgsFormales_>
<ArgsFormales_> → <ListaArgsFormales> )
<ArgsFormales_> → )

<ListaArgsFormales> → <ArgFormal> <ListaArgsFormales_>
<ListaArgsFormales_> → , <ListaArgsFormales>
<ListaArgsFormales_> → λ

<ArgFormal> → <Tipo> identificador
        {
                if
                (TS.getClass(currentClass).getMethod(currentMethod).getParameter(identif
                icador.lexeme) != null)
                        Error: Ya existe una argumento formal con el mismo nombre en el
                método.
                else
                        TS.getClass(currentClass).getMethod(currentMethod)
                          .addParameter(identificador.lexeme, Type.type_s)

        }

<VarsLocales> → { ListaAtributos.from_h ← VarsLocales.from_h } <ListaAtributos>

<ListaAtributos0> → { Atributo.from_h ← ListaAtributos0.from_h } <Atributo>
{ ListaAtributos1.from_h ← ListaAtributos0.from_h } <ListaAtributos1>

<ListaAtributos> →  λ

<ModMetodo> → static { ModMetodo.mod_s ← “static” }
```

```
<ModMetodo> → dynamic { ModMetodo.mod_s ← “dynamic” }

<TipoMetodo> → <Tipo> { TipoMetodo.type_s ← Tipo.type_s }
<TipoMetodo> → void { TipoMetodo.type_s ← VoidType() }

<Tipo> → <TipoPrimitivo> { Tipo.type_s ← TipoPrimitivo.type_s }
<Tipo> → identificador { Tipo.type_s ← identificador.lexeme }

<TipoPrimitivo> → boolean { TipoPrimitivo.type_s ← BooleanType() }
<TipoPrimitivo> → char { TipoPrimitivo.type_s ← CharType() }
<TipoPrimitivo> → int { TipoPrimitivo.type_s ← IntegerType() }
<TipoPrimitivo> → String { TipoPrimitivo.type_s ← StringType() }

<ListaDecVars> → identificador
        {
                ListaDecVars_.from_h ← ListaDecVars.from_h
                ListaDecVars_.type_h ← ListaDecVars.type_h
        }
<ListaDecVars_>
        {
                if (ListaDecVars.from_h == “Clase”) {
                        if (!TS.getClass(currentClass)
                                .getInstanceVariable(identificador.lexeme)) {
                                TS.getClass(currentClass).addInstanceVariable(identificado
                                r.lexeme, ListaDecVars.type_h)
                        } else {
                                Error: Ya existe una variable de instancia con el mismo
                                nombre en la clase actual
                        }
                } else if (ListaDecVars.from_h == “Metodo”) {

                        if (TS.getClass(currentClass).getMethod(currentMethod)
                              .getParameter(identificador.lexeme) != null) {
                                Error: Ya existe una variable local con el mismo
                                nombre en el método actual.
                                } else if (TS.getClass(currentClass)
                                .getMethod(currentMethod).getLocalVariable(identificador.l
                                exeme) != null) {
                                Error: Ya existe un parámetro con el mismo nombre
                                en el método actual.
                                } else {
                                TS.getClass(currentClass).getMethod(currentMethod).
                                addLocalVariable(identificador.lexeme,
                                ListaDecVars.type_h)
                                }

                        }
                }

<ListaDecVars_> → , <ListaDecVars>
        {
                ListaDecVars.from_h ← ListaDecVars_.from_h
                ListaDecVars.type_h ← ListaDecVars_.type_h
        }
<ListaDecVars_> → λ

<Bloque> → { <ListaSentencias> }
        { Block.tree_s ← BlockNode(ListaSentencias.list_s) }

<ListaSentencias0> → <Sentencia>
        {
                if (ListaSentencias0.list_h == null) {
                        ListaSentencias0.list_h ← createList()
                        ListaSentencias0.list_h.add(Sentencia.tree_s)
                        ListaSentencias1.list_h ← ListaSentencias0.list_h
                } else {
```

```
                        ListaSentencias0.list_h.add(Sentencia.tree_s)
                        ListaSentencias1.list_h ← ListaSentencias1.list_s
                }
        }
<ListaSentencias1>
        { ListaSentencias0.list_h ← ListaSentencias1.list_s }

<ListaSentencias> → λ
        { ListaSentencias.list_s ← ListaSentencias.list_h }

<Sentencia> → ;
        { Sentencia.tree_s ← new SeparatorNode() }

<Sentencia> → <Asignacion> ;
        { Sentencia.tree_s ← Asignacion.tree_s }

<Sentencia> → <SentenciaSimple> ;
        { Sentencia.tree_s ← SentenciaSimple.tree_s }

<Sentencia0> → if ( <Expresion> ) <Sentencia1>
        {
                Sentencia_.expr_h ← Expresion.tree_s
                Sentencia_.sent_h ← Sentencia1.tree_s
                Sentencia0.tree_s ← Sentencia_.tree_s
        }
        <Sentencia_>

<Sentencia_> → else <Sentencia>
        { Sentencia_.tree_s = new IfThenElseNode(Sentencia_.expr_h, Sentencia_.sent_h,
        Sentencia.tree_s) }

<Sentencia_> → λ
        { Sentencia_.tree_s ← new IfThenNode(Sentencia_.expr_h, Sentencia_.sent_h) }

<Sentencia0> → while ( <Expresion> ) <Sentencia1>
        { Sentencia0.tree_s ← new WhileNode(Expresion.tree_s, Sentencia1.tree_s) }

<Sentencia0> → for ( <Asignacion> ; <Expresion0> ; <Expresion1> ) <Sentencia1>
        { Sentencia0.tree_s ← new ForNode(Asignacion.tree_s, Expresion0.tree_s,
Expresion1.tree_s, Sentencia1.tree_s) }

<Sentencia> → <Bloque>
        { Sentencia.tree_s ← Bloque.tree_s }

<Sentencia> → return <Sentencia__> ;
        {
                if (Sentencia__.tree_s == null)
                        Sentencia.tree_s ← new ReturnNode()
                else
                        Sentencia.tree_s ← new ReturnExpNode(Sentencia__.tree_s)
        }

<Sentencia__> → <Expresion>
        { Sentencia__.tree_s ← Expresion.tree_s }

<Sentencia__> → λ
        { Sentencia__.tree_s ← null }

<Asignacion> → identificador = <Expresion>
        { Asignacion.tree_s ← new AssignNode(new IdNode(identificador),
Expresion.tree_s) }

<SentenciaSimple> → ( <Expresion> )
        { SentenciaSimple.tree_s ← Expresion.tree_s }

<Expresion> → <Expresion6>
```

```
        { Expresion.tree_s ← Expresion6.tree_s }

<Expresion6> → <Expresion5>
        { Expresion6_.tree_h ← Expresion5.tree_s }

<Expresion6_>
        { Expresion6.tree_s ← Expresion6_.tree_s }

        <Expresion6_0> → || <Expresion5>  {
                Expresion6_1.tree_h ← new BinaryExpressionNode(Expresion6_0.tree_h,
                “||”, Expresion5.tree_s)

        }
         <Expresion6_1>
        { Expresion6_0.tree_s ← Expresion6_1.tree_s }


<Expresion6_> → λ { Expresion6_.tree_s ← Expresion6_.tree_h }

<Expresion5> → <Expresion4> <Expresion5_>
        {
                Expresion5_.tree_h ← Expresion4.tree_s
                Expresion5.tree_s ← Expresion5_.tree_s
        }

<Expresion5_0> → && <Expresion4> <Expresion5_1>
        {
                Expresion5_1.tree_h ← new BinaryExpressionNode(Expresion5_0.tree_h,
“&&”, Expresion4.tree_s)
                Expresion5_0.tree_s ← Expresion5_1.tree_s
        }

<Expresion5_> → λ { Expresion5_.tree_s ← Expresion5_.tree_h }

<Expresion4> → <Expresion3> <Expresion4_>
        {
                Expresion4_.tree_h ← Expresion3.tree_s
                Expresion4.tree_s ← Expresion4_.tree_s
        }

<Expresion4_0> → <Operador4> <Expresion3> <Expresion4_1>
        {
                Expresion4_1.tree_h ← new BinaryExpressionNode(Expresion4_0.tree_h,
Operador4.token, Expresion3.tree_s)
                Expresion4_0.tree_s ← Expresion4_1.tree_s
        }

<Expresion4_> → λ { Expresion4_.tree_s ← Expresion4_.tree_h }

<Expresion3> → <Expresion2> <Expresion3_>
        {
                Expresion3_.tree_h ← Expresion2.tree_s
                Expresion3.tree_s ← Expresion3_.tree_s
        }

<Expresion3_> → <Operador3> <Expresion2>
        {
                Expresion3_.tree_s ← new BinaryExpressionNode(Expresion3_0.tree_h,
                Operador3.token, Expresion2.tree_s)
        }

<Expresion3_> → λ { Expresion3_.tree_s ← Expresion3_.tree_h }

<Expresion2> → <Expresion1> <Expresion2_>
        {
                Expresion2_.tree_h ← Expresion1.tree_s
```

```
                Expresion2.tree_s ← Expresion2_.tree_s
        }

<Expresion2_0> → <Operador2> <Expresion1> <Expresion2_1>
        {
                Expresion2_1.tree_h ← new BinaryExpressionNode(Expresion2_0.tree_h,
                Operador2.token, Expresion1.tree_s)
                Expresion2_0.tree_s ← Expresion2_1.tree_s
        }

<Expresion2_> → λ { Expresion2_.tree_s ← Expresion2_.tree_h }

<Expresion1> → <Expresion0> <Expresion1_>
        {
                Expresion1_.tree_h ← Expresion0.tree_s
                Expresion1.tree_s ← Expresion1_.tree_s
                Expresion1.tree_s ← Expresion1_.tree_s
                Expresion1.this_s ← Expresion0.this_s
        }

<Expresion1_0> → <Operador1> <Expresion0> <Expresion1_1>
        {
                Expresion1_1.tree_h ← new BinaryExpressionNode(Expresion1_0.tree_h,
                Operador1.token, Expresion0.tree_s)
                Expresion1_0.tree_s ← Expresion1_1.tree_s

        }

<Expresion1_> → λ { Expresion1_.tree_s ← Expresion1_.tree_h }

<Expresion0_0> → <OperadorUnario> <Expresion0_1>
        {
                Expresion0_0.tree_s ← new UnaryExpressionNode(OperadorUnario.token,
                Expresion_0.tree_s)
                Expresion0_0.tree_s ← Expresion0_1.tree_s
        }

<Expresion0> → <Primario>
        {
                Expresion0.tree_s ← Primario.tree_s
                TS.getCurrentMethod()
        }

<OperadorUnario> → ! { OperadorUnario.token ← "!" }
<OperadorUnario> → + { OperadorUnario.token ← "+" }
<OperadorUnario> → - { OperadorUnario.token ← "-" }

<Operador1> → * { Operador1.token ← "*" }
<Operador1> → / { Operador1.token ← "/" }
<Operador1> → % { Operador1.token ← "%" }

<Operador2> → + { Operador2.token ← "+" }
<Operador2> → - { Operador2.token ← "-" }

<Operador3> → < { Operador3.token ← "<" }
<Operador3> → > { Operador3.token ← ">" }
<Operador3> → >= { Operador3.token ← ">=" }
<Operador3> → <= { Operador3.token ← "<=" }

<Operador4> → == { Operador4.token ← "==" }
<Operador4> → != { Operador4.token ← "!=" }

<Primario> → this
        {
                Primario.tree_s ← new ThisNode()
        }
```

```
<Primario> → <Literal> { Primario.tree_s ← new LiteralNode() }

<Primario> → ( <Expresion> ) <ListaLlamadas>
	{
		if (ListaLlamadas.list_s == null)
			Primario.tree_s ← new ExpressionCallNode(Expresion.tree_s)
		else
			Primario.tree_s ← new ExpressionCallNode(Expresion.tree_s,
ListaLlamadas.list_s)
	}

<Primario> → identificador { ListaLlamadas_.id_h ← "id" } <ListaLlamadas_> {
Primario.tree_s ← ListaLlamadas_.tree_s }

<Primario> → new identificador <ArgsActuales> <ListaLlamadas> { Primario.tree_s ← new
NewNode(new IdNode(identificador), ArgsActuales.list_s, ListaLlamadas.list_s) }

<ListaLlamadas0> → <Llamada>
	{
		if (ListaLlamadas0.list_h == null)
			ListaLlamadas0.list_h ← createList()
		ListaLlamadas0.list_h.add(Llamada.tree_s)
		ListaLlamadas1.list_h ← ListaLlamadas0.list_h
	}
<ListaLlamadas1> { ListaLlamadas0.list_s ← ListaLlamadas1.list_s }

<ListaLlamadas> → λ { ListaLlamadas.list_s ← ListaLlamadas.list_h }

<ListaLlamadas_> → <ListaLlamadas> { ListaLlamadas_.tree_s ← new
IdExpressionCallNode(new IdNode(ListaLlamadas_.id_h), ListaLlamadas.list_s) }

<ListaLlamadas_> → <ArgsActuales> <ListaLlamadas> { ListaLlamadas_.tree_s ← new
IdExpressionCallNode(new IdNode(ListaLlamadas_.id_h), ArgsActuales.list_s,
ListaLlamadas.list_s) }

<Llamada> → . identificador <ArgsActuales> { Llamada.tree_s ← new CallNode(new
IdNode(identificador), ArgsActuales.list_s) }

<Literal> → null { Literal.token ← ClassType("null") }
<Literal> → true { Literal.token ← BooleanType() }
<Literal> → false { Literal.token ← BooleanType() }
<Literal> → intLiteral { Literal.token ← IntegerType() }
<Literal> → charLiteral { Literal.token ← CharType() }
<Literal> → stringLiteral { Literal.token ← StringType() }

<ArgsActuales> → ( <ArgsActuales_> { ArgsActuales.list_s ← ArgsActuales_.list_s }

<ArgsActuales_> → { ListaExps.list_h ← createList() } <ListaExps> ) {
ArgsActuales_.list_s ← ListaExps.list_s }
<ArgsActuales_> → ) { ArgsActuales_.list_s ← null }


<ListaExps> → <Expresion>
	{
		ListaExps.list_h.add(Expresion.tree_s)
		ListaExps_.list_h ← ListaExps.list_h
	}
<ListaExps_>

<ListaExps_> → , <ListaExps> { ListaExps.list_s ← ListaExps_.list_h }

<ListaExps_> → λ { ListaExps_.list_s ← ListaExps.list_h }
```

Estructuras y operaciones auxiliares

createList()
add()
BooleanType()
CharType()
IntegerType()
StringType()
ClassType()
VoidType()

Chequeos semánticos

El proceso de compilación comienza con la ejecución de la clase principal *Main*, encargada de recibir el nombre del archivo fuente a compilar y el archivo de salida deseado.

Main invoca, con esta información, a la clase *SemanticAnalyzer*. Esta clase inicializa las estructuras necesarias para comenzar con el análisis semántico y la posterior generación de código intermedio.

Más precisamente, *SemanticAnalyzer* se encarga de llevar adelante el análisis léxico y sintáctico, creando de este modo una representación intermedia del código - tabla de símbolos y árboles AST - y luego efectuando sobre la misma el chequeo de declaraciones, el chequeo de sentencias y la generación de código final.

SemanticAnalyzer hace uso de los servicios provistos por las clases *ICEntry* e *ICGenerator* para el manejo de las instrucciones CeIASM y su impresión al archivo de salida.

La tabla de símbolos está compuesta principalmente por entradas de clase *ClassEntry*, cada una con sus respectivas variables de instancia *InstanceVariableEntry*, constructor *ConstructorEntry* y métodos *MethodEntry*.

A su vez, cada servicio - método o constructor -, tiene variables locales *localVariableEntry*, parámetros *parametersEntry* y cuerpo, que será un árbol AST (en nuestro modelo, *BlockNode*).

Esta representación nos permite realizar los chequeos de declaraciones y sentencias delegando el control a las diferentes unidades que componen un archivo de código fuente escrito usando el lenguaje Minijava.

En la *Figura 1* detallamos la relación entre las clases previamente mencionadas (diagrama de clases mínimo) y en la *Figura 2* se presenta la estructura de los árboles AST.

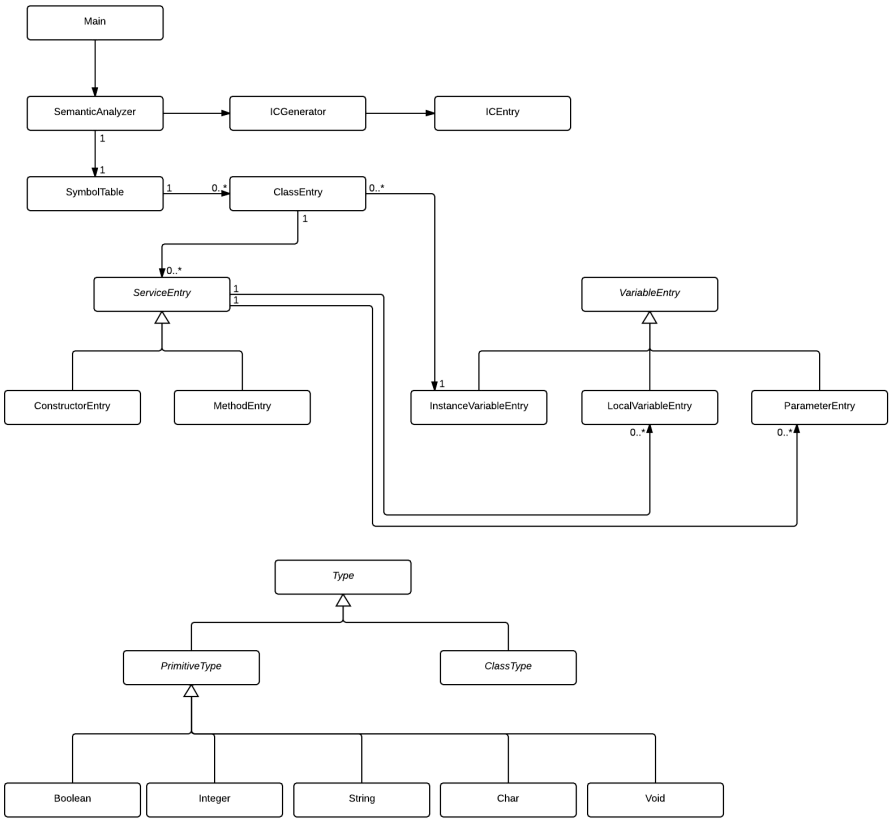


Figura 1

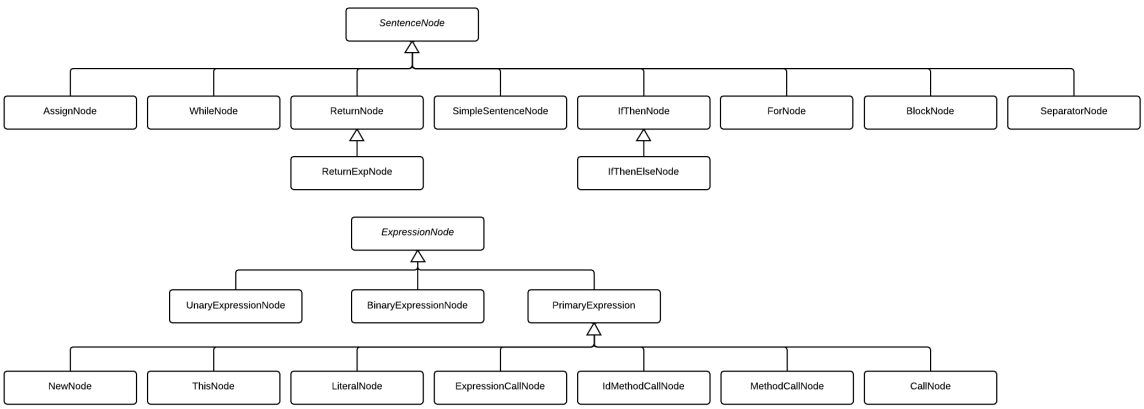


Figura 2

Control de declaraciones

El control de declaraciones se realiza sobre la tabla de símbolos construida durante la primer pasada del compilador. Este control se lleva adelante invocando a las siguientes rutinas.

declarationCheckInheritance()
Controla que exista en la tabla de símbolos la clase padre de todas las clases (excepto *Object* y *System*). En caso de no existir, se lanzará una excepción.

declarationCheckReturnType()
Controla que existan en la tabla de símbolos los tipos de retorno de todos los métodos de todas las clases. En caso de no existir, se lanzará una excepción.

declarationCheckVariables()
Controla que existan en la tabla de símbolos todos los tipo de las variables de instancia de todas las clases y todos los tipos de las variables locales y parámetros de los servicios que la componen. Dividimos este control en tres partes.

declarationCheckInstanceVariables()
Controla la existencia del tipo de las variables de instancia de una determinada clase.

declarationCheckParameters()
Controla la existencia del tipo de los parámetros de un determinado método.

declarationCheckLocalVars()
Controla la existencia del tipo de las variables locales de un determinado método.

En caso de encontrar una inconsistencia, se lanzará una excepción.

consolidateInheritance()
Se efectúa, de ser posible, la herencia por copia entre las clases que tienen una relación "es un". Se delega este control a la rutina *controlInheritance()*.

controlInheritance()
Si la clase padre no fue controlada, entonces se invoca recursivamente a la consolidación para la clase padre. En caso contrario, se delega el control de consolidación a la entrada de clase de la clase actual.
En este momento controlamos que los métodos heredados no oculten ninguno de los nombres de la clase actual y, de haber un método que intenta redefinirse, que las signaturas coincidan.
También aprovechamos este momento para hacer la inicialización de los offsets de los métodos, variables locales y parámetros de la clase actual.

consolidateConstructors()
Se inicializan los offsets de los parámetros y variables locales de los constructores.

declarationCheckMainExistence()
Controla que exista un método principal sin retorno (void) y estático (static) en alguna de las clases presentes en la tabla de clases.

Control de sentencias

El control de sentencias se realiza sobre la tabla de símbolos y los árboles AST a continuación del control de declaraciones. Este control se lleva adelante invocando a la rutina *sentenceCheck()* de la tabla de símbolos.

sentenceCheck()
Invoca el control *checkClass()* para cada clase presente en la tabla de clases distinta de *Object* y *System*

checkClass()
Invoca el control *checkService()* para el constructor de la clase actual y para cada método presente en la tabla de métodos.

checkService()
Invoca el control *checkNode()* para cada uno de los servicios de la clase actual. No se controlan aquellos métodos que hayan sido heredados (evitamos controlar más de una vez)

checkNode()
Cada *BlockNode* está compuesto de una lista de sentencias. El *checkNode()* en un *BlockNode* invoca *checkNode()* para cada una de las sentencias que componen esa lista.
Dependiendo el tipo de nodo del árbol AST que constituye el cuerpo del servicio se realizarán los controles correspondientes. Ejemplo, si es una asignación, se verificará que el lado izquierdo sea un nombre visible (variable local, parametro o variable de instancia, si no se trata de un contexto estático) y que el lado derecho sea una expresión que conforme con el tipo del lado izquierdo.

El control de sentencias finaliza cuando se realizó el control correspondiente a la última sentencia del cuerpo del último servicio declarado en el código fuente.

Batería de tests y manejo de errores

La batería de tests está compuesta de un conjuntos de tests exitosos y de un conjunto de tests que deben fallar.

Tests que deben terminar exitosamente

- *clase_atributo.java*
Clase simple con atributos

- *clase_constructor.java*
Clase simple con un constructor

- *clase_herencia.java*
Herencia simple

- *clase_metodo_herencia.java*
Herencia con redefinición de métodos

- *clase_metodo.java*
Clase simple con métodos

- *clase_metodo_sentencia_asignacion.java*
Clases simple con asignaciones

- *clase_metodo_sentencia_for.java*
Clases simple con ciclos for

- *clase_metodo_sentencia_ifthenelse.java*
Clases simples con sentencias if-then-else

- *clase_metodo_sentencia_puntoycoma.java*
Clases simples con sentencias vacías

- *clase_metodo_sentencia_sentenciasimple.java*
Clases simples con sentencias simples

- *clase_metodo_sentencia_while.java*
Clases simples con ciclos while

- *clase_simple.java*
Clases simples

- *clase_simple_simple.java*
Clases simples

- *clase_vacia.java*
Clase vacia

- *herencia_invertido.java*
Clase hijo antes que clase padre

- *herencia.java*
Invocación de un método heredado

- *instancia_clase_Object.java*
Creación de una instancia de Object

- *invocacion_dinamico_metodo_static.java*
Invocación de un método dinámico en un contexto estático a través de una variable

- *invocacion_estatico.java*
Invocación de un método estático en un contexto dinámico a través del nombre de la clase

- *jerarquia_herencia.java*
Invocación de métodos heredados

- *llamada_estatica_cadena_llamadas_dynamic.java*
Invocación de métodos estáticos en una cadena de llamadas

- *llamada_estatica_variable_instancia.java*
Invocación de métodos estáticos a través de una variable de instancia

- programa_sintacticamente_valido01.java
Programa aleatorio

- programa_sintacticamente_valido02.java
Programa aleatorio

- sentenciasimple-expresion.java
Conjunto de sentencias simples

Tests que deben fallar

- heredar_clase_System.java
Output esperado:
```
IntermediateCodeGeneration.SemanticException: Linea: 3 - Error semantico: Una clase no puede heredar de la clase System.
```

- herencia_circular.java
Output esperado:
```
IntermediateCodeGeneration.SemanticException: Error semantico: Herencia circular. La clase B no puede heredar de si misma.
```

- herencia_conflictiva01.java
Output esperado:
```
IntermediateCodeGeneration.SemanticException: Linea: 19 - Error semantico:  El tipo de retorno del metodo 'methoda1' de la clase padre 'A' es diferente al tipo de retorno del metodo 'methoda1' de la clase hija 'B'.
En la clase padre el tipo de retorno del metodo es boolean y en la clase hija es int.
```

- herencia_conflictiva02.java
Output esperado:
```
IntermediateCodeGeneration.SemanticException: Linea: 19 - Error semantico: El modificador del metodo 'methoda1' de la clase padre 'A' es diferente al modificador del metodo methoda1 de la clase hija 'B'.
En la clase padre el modificador del metodo es dynamic y en la clase actual es static.
```

- herencia_conflictiva03.java
Output esperado:
```
IntermediateCodeGeneration.SemanticException: Linea: 19 - Error semantico: Los nombres de los parametros del metodo 'methoda1 de la clase padre 'A' son diferentes a los nombres de los parametros del metodo 'methoda1' de la clase hija 'B.
En la clase padre el parametro en la posicion 1 del metodo se llama c y en la clase hija se llama b.
```

- herencia_conflictiva04.java
Output esperado:
```
IntermediateCodeGeneration.SemanticException: Linea: 13 - Error semantico: La clase A tiene un metodo con el mismo nombre que la variable de instancia methoda2 de su subclase B.
```

- herencia_conflictiva05.java
Output esperado:
```
IntermediateCodeGeneration.SemanticException: Linea: 21 - Error semantico: La cantidad de parametros del metodo 'methoda1' de la clase padre 'A' es diferente a la cantidad de parametros del metodo 'methoda1' de la clase hija 'B.
En la clase padre la cantidad de parametros del metodo es 3 y en la clase hija es 2.
```

- *herencia.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 15 - Error semantico: La variable 'n' no es visible en el metodo 'printN'.
```

- *inconformidad_tipos_asignacion.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 41 - Error semantico: No puede asignarse una expresion de tipo Padre a una variable de tipo Hija.
```

- *inconformidad_tipos_parametros.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 8 - Error semantico: En la llamada al metodo 'conformityTest' el tipo del argumento actual en la posicion (1) no conforma con el tipo del argumento formal. El tipo del argumento actual es int y el tipo del argumento formal es String.
```

- *inconformidad_tipos_retorno.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 10 - Error semantico: El tipo de la expresion retornada no es conforme al tipo de retorno del metodo actual. El tipo de la expresion retornada es 'int' y el tipo de retorno del metodo es 'String'.
```

- *instancia_clase_System.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 7 - Error semantico: No se puede crear una instancia de la clase System.
```

- *invocacion_dinamico_metodo_static.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 20 - Error semantico: No puede hacerse una invocacion al metodo dinamico 'getN' en la clase Objeto en el contexto del metodo estatico 'main'.
```

- *invocacion_metodo_static.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 18 - Error semantico: No puede hacerse una invocacion al metodo dinamico 'methodi1' en la clase Objeto en el contexto del metodo estatico 'main'.
```

- *invocacion_parametros.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 23 - Error semantico: En la llamada al metodo 'methodb1' el tipo del argumento actual en la posicion (2) no conforma con el tipo del argumento formal. El tipo del argumento actual es Objeto2 y el tipo del argumento formal es Objeto.
```

- *llamada_dinamica_usando_nombre_clase.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 31 - Error semantico: No es posible invocar el metodo dinamico 'toString' teniendo como lado izquierdo el nombre de la clase.
```

- *llamada_dynamic_sin_id_desde_static.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 34 - Error semantico: No puede hacerse una invocacion al metodo dinamico 'ret1' en la clase Reverse en el contexto del metodo estatico 'm'.
```

- *metodo_constructor.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 4 - Error semantico: La variable 'foo' no es visible en el constructor 'A'.
```

- *numero_linea_correcto_error_for.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 7 - Error semantico: La condicion de la sentencia for debe ser de tipo boolean. Se encontro una expresion de tipo 'String'.
```

- *numero_linea_correcto_error_if.java*

Output esperado:

```
IntermediateCodeGeneration.LexicalException: Linea: 1 - Error lexico: Caracter no soportado ().
```

- *numero_linea_correcto_error_while.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 6 - Error semantico: La condicion de la sentencia while debe ser de tipo boolean. Se encontro una expresion de tipo 'String'.
```

- *retorno_constructor.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 5 - Error semantico: Un constructor no puede tener retorno.
```

- *retorno_en_void.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 21 - Error semantico: Un metodo de tipo void no puede retornar un valor.
```

- *sentencia_simple.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 27 - Error semantico: El metodo 'methodb1' no esta declarado en la clase Objeto.
```

- *this_metodo_static.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 20 - Error semantico: No se puede hacer referencia al objeto actual usando la palabra reservada 'this' en el contexto de un metodo estatico.
```

- *tipo_retorno_System.java*

Output esperado:

```
IntermediateCodeGeneration.SyntacticException: Linea: 8 - Error semantico: No se puede definir un metodo con tipo de retorno System.
```

- *variable_instancia_metodo_static.java*

Output esperado:

```
IntermediateCodeGeneration.SemanticException: Linea: 20 - Error semantico: No puede usarse la variable de instancia 'n' en un metodo estatico.
```

- *variable_tipo_System.java*
Output esperado:
```
IntermediateCodeGeneration.SyntacticException: Linea: 5 - Error semantico: No se puede declarar una variable de tipo System.
```

Errores semánticos que es capaz de detectar el compilador

A continuación serán detallados cada uno de los errores semánticos que el compilador es capaz detectar. Cada error será clasificado según la pasada en la que es detectado por el compilador, y si es detectado en la segunda pasada será subclasificado según si pertenece al control de declaraciones o al control de sentencias.
Luego se dará una breve descripción de cada error junto con la clase y el método en el que se realiza el control y finalmente se indicará el mensaje de error que es mostrado por pantalla al ser detectado.

Primer pasada: Construcción de la tabla de símbolos y árboles AST

Los siguientes errores son todos detectados en la clase *Parser* y dichos controles se reflejan en el EDT.

Colisión de nombres de clase

Descripción: Si en un mismo archivo fuente se declaran clases con el mismo nombre, ocurrirá un error. Se consideran también las clases predefinidas por el lenguaje, *Object* y *System*, que son insertadas en en la tabla de clases al comenzar el análisis semántico.
Este control se realiza en la primera pasada.
Clase: Parser
Método: Clase
Mensaje de error: "Error semántico: Ya existe una clase declarada con el nombre [nombre de la clase]."

Herencia de clase System

Descripción: Si una clase extiende a la clase System, ocurrirá un error. Este control se realiza en la primera pasada.
Clase: Parser
Método: Herencia
Mensaje de error: "Error semántico: Una clase no puede heredar de la clase System."

Variable de tipo System

Descripción: Si se declara una variable de tipo System, ocurrirá un error. Este control se realiza en la primera pasada.
Clase: Parser
Método: Atributo
Mensaje de error: "Error semántico: No se puede declarar una variable de tipo System."

Retorno de tipo System

Descripción: Si se define un método con tipo de retorno System, ocurrirá un error. Este control se realiza en la primera pasada.
Clase: Parser
Método: Método
Mensaje de error: "Error semántico: No se puede definir un método con tipo de retorno System."

Método con nombre de clase

Descripción: Si se define un método con el mismo nombre que la clase en la que se encuentra definido, ocurrirá un error. Este control es realizado en la primer pasada.
Clase: Parser
Método: Metodo
Mensaje de error: "Error semántico: El método [nombre del método] no puede tener el mismo nombre que su clase."

Sobrecarga de métodos

Descripción: MiniJava no soporta sobrecarga, por lo tanto ocurrirá un error si se declaran dos métodos con el mismo nombre dentro de una misma clase, por más que tengan diferente signatura (ya sea por el tipo, cantidad de parámetros o valor de retorno). Este control es realizado en la primer pasada.
Clase: Parser
Método: Metodo
Mensaje de error: "Error semántico: Ya existe un método [nombre del método] declarado en la clase [nombre de la clase]."

Nombre de constructor inválido

Descripción: Si se define un servicio cuya signatura corresponde a la de un constructor (es decir, sin tipo de retorno ni modificador) pero su nombre no coincide con el nombre de la clase, ocurrirá un error. Este control se realiza en la primer pasada.
Clase: Parser
Método: Constructor
Mensaje de error: "Error semántico: El nombre del constructor [nombre del constructor] no corresponde al nombre de la clase."

Múltiples constructores en una misma clase

Descripción: Si se define más de un constructor en una clase, ocurrirá un error. Este control se realiza en la primer pasada.
Clase: Parser
Método: Constructor
Mensaje de error: "Error semántico: Ya existe un constructor en la clase [nombre de la clase]."

Colisión de nombres entre variables de instancia

Descripción: Si en una clase se declara más de una variable de instancia con el mismo nombre, ocurrirá un error. Este control se realiza en la primer pasada.
Clase: Parser

Método: ListaDecVars
Mensaje de error: "Error semántico: Existe más de una variable de instancia con el nombre [nombre de la variable] en la clase [nombre de la clase]."

Colisión de nombres entre variable de instancia y clase

Descripción: Si se declara una variable de instancia cuyo nombre es igual al de su clase, ocurrirá un error. Este control es realizado en la primer pasada.
Clase: Parser
Método: ListaDecVars
Mensaje de error: "Error semántico: La variable de instancia [nombre de la variable de instancia] no puede tener el mismo nombre que la clase en la que se declara."

Colisión de nombres entre variable de instancia y método

Descripción: Si se declara una variable de instancia cuyo nombre es igual al de un método de su clase, ocurrirá un error. Este control es realizado en la primer pasada.
Clase: Parser
Método: ListaDecVars
Mensaje de error: "Error semántico: Existe un método previamente declarado con el nombre [nombre de la variable de instancia] en la clase [nombre de la clase]."

Colisión de nombres entre parámetros

Descripción: Si se define un método o un constructor con más de un parámetro con el mismo nombre, ocurrirá un error. Este control se realiza en la primer pasada.
Clase: Parser
Método: ListaDecVars
Mensaje de error: "Error semántico: Existe más de un parámetro con el nombre [nombre de la variable] en el [método | constructor] [nombre del método] en la clase [nombre de la clase]."

Colisión de nombres entre variable locales

Descripción: Si se define un método o un constructor con más de una variable local con el mismo nombre, ocurrirá un error. Este control se realiza en la primer pasada.
Clase: Parser
Método: ListaDecVars
Mensaje de error: "Error semántico: Existe más de una variable local con el nombre [nombre de la variable] en el [método | constructor] [nombre del método] de la clase [nombre de la clase]."

Segunda pasada: Control de declaraciones

Padre de clase inexistente

Descripción: Si se definió una clase cuya clase padre no existe, ocurrirá un error. Este control es realizado en la segunda pasada.
Clase: SymbolTable
Método: declarationCheckInheritance

Mensaje de error: "Error semántico: La clase [nombre de la clase padre], padre de la clase [nombre de la clase hija], no existe."

Herencia circular

Descripción: La herencia circular ocurre cuando una clase se tiene a sí misma como ancestro. En este caso, una clase estaría heredando su misma funcionalidad, lo cual no tiene sentido y es considerado un error. Este control es realizado en la segunda pasada.
Clase: SymbolTable
Método: controlCircularInheritance
Mensaje de error: "Error semántico: Herencia Circular. La clase [nombre de la clase] no puede heredar de sí misma."

Tipo de retorno de método inexistente

Descripción: Si se define un método cuyo tipo de retorno no existe, ocurrirá un error. Este control es realizado en la segunda pasada.
Clase: SymbolTable
Método: declarationCheckReturnType
Mensaje de error: "Error semántico: El tipo de retorno [tipo del retorno del método] del método [nombre del método] no existe."

Ausencia de método main

Descripción: Alguna clase tiene que tener un método estático *main* sin parámetros. Esta clase será considerada como principal. Si ninguna clase contiene el método estático *main* ocurrirá un error. Este control es realizado en la segunda pasada.
Clase: SymbolTable
Método: declarationCheckMainExistence
Mensaje de error: "Error semántico: El método 'main' no fue declarado en ninguna de las clases."

Tipo de variable de instancia inexistente

Descripción: Si se declara una variable de instancia cuyo tipo no existe, ocurrirá un error. Este control es realizado en la segunda pasada.
Clase: SymbolTable
Método: declarationCheckInstanceVariables
Mensaje de error: "Error semántico: El tipo [tipo de la variable de instancia] de la variable de instancia [nombre de la variable de instancia] no existe."

Tipo de parámetro inexistente

Descripción: Si al definir un método se declara un parámetro cuyo tipo no existe, ocurrirá un error. Este control es realizado en la segunda pasada.
Clase: SymbolTable
Método: declarationCheckInstanceParameters
Mensaje de error: "Error semántico: El tipo [tipo del parámetro] del parámetro [nombre del parámetro] no existe."

Tipo de variable local inexistente

Descripción: Si se declara una variable local cuyo tipo no existe, ocurrirá un error. Este control es realizado en la segunda pasada.
Clase: SymbolTable
Método: declarationCheckInstanceParameters
Mensaje de error: "Error semántico: El tipo [tipo de la variable local] de la variable local [nombre de la variable local] no existe."

Método main no estático

Descripción: Si se define un método *main* cuyo modificador no es *static*, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: ClassEntry
Método: hasMain
Mensaje de error: "Error semántico: El método main debe ser estático."

Método main con parámetros

Descripción: Si se define un método *main* cuya cantidad de parámetros no es 0, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: ClassEntry
Método: hasMain
Mensaje de error: "Error semántico: El método main no debe tener parámetros."

Colisión de nombre entre método de superclase y nombre de subclase

Descripción: Si en una superclase se define un método con el mismo nombre que el nombre de alguna de sus subclases, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: ClassEntry
Método: inheritMethods
Mensaje de error: "Error semántico: La clase [nombre de la superclase] tiene un método con el mismo nombre que su subclase [nombre de la subclase]."

Colisión de nombre entre método de superclase y variable de instancia de subclase

Descripción: Si en una superclase se define un método con el mismo nombre que una variable de instancia de alguna de sus subclases, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: ClassEntry
Método: addInheritedMethods
Mensaje de error: "Error semántico: La clase [nombre de la superclase] tiene un método con el mismo nombre que la variable de instancia [nombre de la variable] de su subclase [nombre de la subclase]."

Diferente modificador en redefinición de método

Descripción: Si en una superclase de define un método cuyo modificador es diferente al modificador de un método con el mismo nombre de alguna de sus subclases, ocurrirá un error. Este control se realiza en la segunda pasada.

Clase: MethodEntry
Método: compareModifier
Mensaje de error: "Error semántico: El modificador del método [nombre del método] de la clase padre [nombre de la clase padre] es diferente al modificador del método [nombre del método] de la clase hija [nombre de la clase hija]. En la clase padre el modificador del método es [dynamic | static] y en la clase hija es [dynamic | static]."

Diferente tipo de retorno en redefinición de método

Descripción: Si en una superclase de define un método cuyo tipo de retorno es diferente al tipo de retorno de un método con el mismo nombre de alguna de sus subclases, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: MethodEntry
Método: compareReturnType
Mensaje de error: "Error semántico: El tipo de retorno del método [nombre del método] de la clase padre [nombre de la clase padre] es diferente al tipo de retorno del método [nombre del método] de la clase hija [nombre de la clase hija]. En la clase padre el tipo de retorno del método es [tipo de retorno del método de la clase padre] y en la clase hija es [tipo de retorno del método de la clase hija]."

Diferente cantidad de parámetros en redefinición de método

Descripción: Si en una superclase de define un método cuya cantidad de parámetros es diferente a la cantidad de parámetros de un método con el mismo nombre de alguna de sus subclases, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: MethodEntry
Método: compareParameters
Mensaje de error: "Error semántico: La cantidad de parámetros del método [nombre del método] de la clase padre [nombre de la clase padre] es diferente a la cantidad de parámetros del método [nombre del método] de la clase hija [nombre de la clase hija]. En la clase padre la cantidad de parámetros del método es [cantidad de parámetros del método de la clase padre] y en la clase hija es [cantidad de parámetros del método de la clase hija].

Diferentes nombres de parámetros en redefinición de método

Descripción: Si en una superclase de define un método cuyos parámetros tienen diferente nombre a los parámetros de un método con el mismo nombre de alguna de sus subclases, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: MethodEntry
Método: compareParameters
Mensaje de error: "Error semántico: Los nombres de los parámetros del método [nombre del método] de la clase padre [nombre de la clase padre] son diferentes a los nombres de los parámetros del método [nombre del método] de la clase hija [nombre de la clase hija]. En la clase padre el parámetro en la posición [posición del parámetro] del método se llama [nombre del parámetro en la clase padre] y en la clase hija se llama [nombre del parámetro en la clase hija].

Diferente tipos de parámetros en redefinición de método

Descripción: Si en una superclase de define un método cuyos parámetros tienen diferente tipo a los parámetros de un método con el mismo nombre de alguna de sus subclases, ocurrirá un error. Este

control se realiza en la segunda pasada.
Clase: MethodEntry
Método: compareParameters
Mensaje de error: "Error semántico: Los tipos de los parámetros del método [nombre del método] de la clase padre [nombre de la clase padre] son diferentes a los tipos de los parámetros del método [nombre del método] de la clase hija [nombre de la clase hija]. En la clase padre el parámetro en la posición [posición del parámetro] del método es de tipo [tipo del parámetro en la clase padre] y en la clase hija es de tipo [tipo del parámetro en la clase hija]."

Segunda pasada: Control de sentencias

Error de tipo en asignación

Descripción: Al asignar una expresión a una variable, si el tipo de la expresión no conforma al tipo de la variable, se producirá un error se ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: AssignNode
Método: checkNode
Mensaje de error: "Error semántico: No puede asignarse una expresión de tipo [tipo de la expresión] a una variable de tipo [tipo de la variable]."

Variable no declarada en asignación

Descripción: Al asignar una expresión a una variable, si la variable no está declarada ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: AssignNode
Método: checkNode
Mensaje de error: "Error semántico: La variable [nombre de la variable] no está declarada."

Asignación a variable de instancia en método estático

Descripción: Si dentro de un método estático se asigna una expresión a una variable de instancia, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: AssignNode
Método: checkId
Mensaje de error: "Error semántico: No puede usarse la variable de instancia [nombre de la variable] en un método estático."

Error de tipo en operación aritmética binaria

Descripción: Los operadores binarios matemáticos +, -, *, / y operan sólo sobre subexpresiones de tipo entero y devuelven un resultado de tipo entero. Si las subexpresiones son de otros tipos ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: BinaryExpressionNode
Método: checkNode
Mensaje de error: "Error semántico: El operador binario [+ | - | * | / | %] no puede aplicarse a una subexpresión de tipo [tipo de la expresión]. Se esperaba una subexpresión de tipo entero."

Error de tipo en operación lógica binaria

Descripción: Los operadores binarios booleanos && y || operan sólo sobre subexpresiones de tipo boolean y devuelven un resultado de tipo boolean. Si las subexpresiones son de otros tipos ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: BinaryExpressionNode
Método: checkNode
Mensaje de error: "Error semántico: El operador binario [&& | ||] no puede aplicarse a una subexpresión de tipo [tipo de la expresión]. Se esperaba una subexpresión de tipo boolean."

Error de tipo en operación de equivalencia

Descripción: Los operadores de igualdad == y != operan sobre cualquier subexpresiones de tipos conformantes (es decir, relacionados a través de herencia) en cualquier dirección y devuelven un resultado de tipo boolean. Si los tipos de las subexpresiones no conforman entre si ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: BinaryExpressionNode
Método: checkNode
Mensaje de error: "Error semántico: Los tipos de las subexpresiones no son conformantes. La subexpresión de la izquierda es de tipo [tipo de la expresión de la izquierda]" y la subexpresión de la derecha es de tipo [tipo de la expresión de la derecha]."

Error de tipo en operación relacional

Descripción: Los operadores relacionales <, <=, >=, > operan sólo sobre subexpresiones de tipo entero y devuelven un resultado de tipo boolean. Si las subexpresiones son de otros tipos ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: BinaryExpressionNode
Método: checkNode
Mensaje de error: "Error semántico: El operador binario [> | >= | < | <=] no puede aplicarse a una subexpresión de tipo [tipo de la expresión]. Se esperaba una subexpresión de tipo entero."

Llamada a constructor inválida

Descripción: Si se llama al constructor de una clase fuera del contexto de la creación de un objeto (por ejemplo, realizar la llamada *Constructor()* en lugar de *new Constructor()*) ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: CallNode
Método: checkId
Mensaje de error: "Error semántico: No puede realizarse una llamada a un constructor."

Llamada a método inexistente

Descripción: Si se llama a un método sin indicar el objeto receptor del mensaje (aparenta ser un método de la clase en la que se realiza la llamada) y este método no está declarado en la clase, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: CallNode
Método: checkId
Mensaje de error: "Error semántico: El método [nombre del método] no está declarado en la clase [nombre de la clase]."

Llamada a método inválida por cantidad de parámetros

Descripción: Si al realizar la invocación a un método, la cantidad de parámetros actuales no coincide con la cantidad de parámetros formales del mismo, ocurrirá un error. Este control se realiza durante la segunda pasada.
Clase: CallNode
Método: controlFormalArgs
Mensaje de error: "Error semántico: Las listas de argumentos actuales y formales para el método [nombre del método] de la clase [nombre de la clase] tienen diferente longitud."

Llamada a método inválida por tipo de parámetros

Descripción: Si al realizar la invocación a un método, el tipo de la expresión correspondiente a cada parámetro actual no conforma al tipo asociado a cada parámetro formal del mismo, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: CallNode
Método: controlFormalArgs
Mensaje de error: "Error semántico: En la llamada al método [nombre del método] el tipo del argumento actual en la posición [posición del argumento] no conforma con el tipo del argumento formal. El tipo del argumento actual es [tipo del argumento actual] y el tipo del argumento formal es [tipo del argumento formal]."

Error de tipo en condición de sentencia For

Descripción: Una sentencia de la forma for (id = *e1*; *e2*; *e3*) S producirá un error si *e2* no es una expresión correctamente tipada de tipo boolean. Este control es realizado en la segunda pasada.
Clase: ForNode
Método: checkNode
Mensaje de error: "Error semántico: La condición de la sentencia for debe ser de tipo boolean. Se encontró una expresión de tipo [tipo de la expresión]."

Error de tipo en incremento de sentencia For

Descripción: Una sentencia de la forma for (*id* = *e1*; *e2*; *e3*) S producirá un error si *e3* no es una expresión correctamente tipada de tipo que conforma con el tipo de *id*. Este control es realizado en la segunda pasada.
Clase: ForNode
Método: checkNode
Mensaje de error: "Error semántico: La operación aplicada en el incremento no está definida para el tipo del contador."

Uso de variables de instancia en método estático

Descripción: Si en el cuerpo de un método estático aparecen referencias a variables de instancia, ocurirrá un error. Este control se realiza en la segunda pasada.
Clase: IdMethodCallNode
Método: checkId
Mensaje de error: "Error semántico: No puede usarse una variable de instancia en un método estático."

Error de tipo en condición de sentencia If-Then-Else

Descripción: Una sentencia de la forma if (*e*) S1 else S2 producirá un error si *e* no es una expresión correctamente tipada de tipo boolean. Este control se realiza en la segunda pasada.
Clase: IfThenElseNode
Método: checkNode
Mensaje de error: "Error semántico: La condición de la sentencia if-then-else debe ser de tipo boolean. Se encontró una expresión de tipo [tipo de la expresión]."

Error de tipo en condición de sentencia If-Then

Descripción: Una sentencia de la forma if (*e*) S1 producirá un error si *e* no es una expresión correctamente tipada de tipo boolean. Este control se realiza en la segunda pasada.
Clase: IfThenNode
Método: checkNode
Mensaje de error: "Error semántico: La condición de la sentencia if-then debe ser de tipo boolean. Se encontró una expresión de tipo [tipo de la expresión]."

Llamada a constructor inválida

Descripción: Si se llama al constructor de una clase fuera del contexto de la creación de un objeto (por ejemplo, realizar la llamada *Constructor()* en lugar de *new Constructor()*) ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: MethodCallNode
Método: checkId
Mensaje de error: "Error semántico: No puede realizarse una llamada a un constructor."

Llamada a método inexistente

Descripción: Si se llama a un método sin indicar el objeto receptor del mensaje (aparenta ser un método de la clase en la que se realiza la llamada) y este método no está declarado en la clase, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: CallNode
Método: checkId
Mensaje de error: "Error semántico: El método [nombre del método] no está declarado en la clase [nombre de la clase]."

Llamada a método dinámico desde método estático

Descripción: Si se realiza una llamada a un método dinámico desde un método estático, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: MethodCallNode
Método: checkId
Mensaje de error: "Error semántico: No puede hacerse una invocación al método dinámico [nombre del método dinámico] en la clase [nombre de la clase] en el contexto del método estático [nombre del método estático]."

Llamada a método inválida por cantidad de parámetros

Descripción: Si al realizar la invocación a un método, la cantidad de parámetros actuales no coincide con la cantidad de parámetros formales del mismo, ocurrirá un error. Este control se realiza durante la segunda pasada.
Clase: MethodCallNode
Método: controlFormalArgs
Mensaje de error: "Error semántico: Las listas de argumentos actuales y formales para el método [nombre del método] de la clase [nombre de la clase] tienen diferente longitud."

Llamada a método inválida por tipo de parámetros

Descripción: Si al realizar la invocación a un método, el tipo de la expresión correspondiente a cada parámetro actual no conforma al tipo asociado a cada parámetro formal del mismo, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: MethodCallNode
Método: controlFormalArgs
Mensaje de error: "Error semántico: En la llamada al método [nombre del método] el tipo del argumento actual en la posición [posición del argumento] no conforma con el tipo del argumento formal. El tipo del argumento actual es [tipo del argumento actual] y el tipo del argumento formal es [tipo del argumento formal]."

Llamada a constructor inexistente

Descripción: Si al realizar la invocación a un constructor, éste no está declarado en alguna de las clases, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: NewNode
Método: checkId
Mensaje de error: "Error semántico: El constructor invocado no está declarado."

Instanciación de clase System

Descripción: Si se intenta instanciar a la clase System, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: NewNode
Método: checkId
Mensaje de error: "Error semántico: No se puede crear una instancia de la clase System."

Llamada a constructor inválida por cantidad de parámetros

Descripción: Si al realizar la invocación a un constructor, la cantidad de parámetros actuales no coincide con la cantidad de parámetros formales del mismo, ocurrirá un error. Este control se realiza durante la segunda pasada.
Clase: NewNode
Método: controlFormalArgs
Mensaje de error: "Error semántico: Las listas de argumentos actuales y formales para el método [nombre del método] de la clase [nombre de la clase] tienen diferente longitud."

Llamada a método inválida por tipo de parámetros

Descripción: Si al realizar la invocación a un constructor, el tipo de la expresión correspondiente a cada parámetro actual no conforma al tipo asociado a cada parámetro formal del mismo, ocurrirá un

error. Este control se realiza en la segunda pasada.
Clase: NewNode
Método: controlFormalArgs
Mensaje de error: "Error semántico: En la llamada al método [nombre del método] el tipo del argumento actual en la posición [posición del argumento] no conforma con el tipo del argumento formal. El tipo del argumento actual es [tipo del argumento actual] y el tipo del argumento formal es [tipo del argumento formal]."

Retorno de expresión en método void

Descripción: Si un método cuyo tipo de retorno es void retorna una expresión, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: ReturnExpNode
Método: checkNode
Mensaje de error: "Error semántico: Un método de tipo void no puede retornar un valor."

Error de tipo en retorno de método

Descripción: Una sentencia de retorno de la forma return *e*; producirá un error si *e* no es una expresión correctamente tipada donde su tipo es T o no T conforma con el tipo de retorno del método que contiene la sentencia return. Este control se realiza en la segunda pasada.
Clase: ReturnExpNode
Método: checkNode
Mensaje de error: "Error semántico: El tipo de la expresión retornada no conforma al tipo de retorno del método actual. El tipo de la expresión retornada es [tipo de la expresión] y el tipo de retorno del método es [tipo de retorno del método]."

Constructor con retorno

Descripción: Si se define un constructor con retorno, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: ReturnExpNode
Método: checkNode
Mensaje de error: "Error semántico: Un constructor no puede tener un retorno."

Método no void con retorno vacío

Descripción: Una sentencia de retorno de la forma return; producirá un error si el método que la contiene no tiene como tipo de retorno void. Este control se realiza en la segunda pasada.
Clase: ReturnNode
Método: checkNode
Mensaje de error: "Error semántico: Se esperaba retornar una expresión de tipo [tipo de retorno del método]."

Uso de this en método estático

Descripción: Si en un método estático aparecen referencias a la palabra reservada *this*, ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: ThisNode
Método: checkNode

Mensaje de error: “Error semántico: No se puede hacer referencia al objeto actual usando la palabra reservada 'this' en el contexto de un método estático.”

Error de tipo en operación aritmética unaria

Descripción: Los operadores unarios + y - operan sólo sobre subexpresiones de tipo entero y retornan resultados de tipo entero. Si la subexpresión es de otro tipo ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: UnaryExpressionNode
Método: checkNode
Mensaje de error: “Error semántico: El operador unario [+ | -] no puede aplicarse a una subexpresión de tipo [tipo de la expresión]. Se esperaba una subexpresión de tipo entero.”

Error de tipo en operación lógica unaria

Descripción: El operador unario ! opera sólo sobre subexpresiones de tipo boolean y devuelve un resultado de tipo boolean. Si la subexpresión es de otro tipo ocurrirá un error. Este control se realiza en la segunda pasada.
Clase: UnaryExpressionNode
Método: checkNode
Mensaje de error: “Error semántico: El operador unario ! no puede aplicarse a una subexpresión de tipo [tipo de la expresión]. Se esperaba una subexpresión de tipo entero.”

Error de tipo en sentencia While

Descripción: Una sentencia de la forma while(e) S producirá un error si *e* no es una expresión correctamente tipada de tipo boolean.
Clase: WhileNode
Método: checkNode
Mensaje de error: “Error semántico: La condición de la sentencia while debe ser de tipo boolean. Se encontró una expresión de tipo [tipo de la expresión].”

Generación de código intermedio

La generación de código intermedio se realiza en conjunto al control de sentencias.

sentenceCheck()
Invoca el control *checkClass()* para cada clase presente en la tabla de clases distinta de *Object* y *System.* Establece el handler de código intermedio, *ICGenerator*, a la clase a controlar.

checkClass()
Inicializa la *Virtual Table* de la clase e invoca el control *checkService()* para el constructor de la clase actual y para cada método presente en la tabla de métodos. Establece el handler de código intermedio, *ICGenerator*, a cada servicio a controlar

checkService()
Inicializa el CI del servicio, invoca el control *checkNode()* y llama a la rutina *generateCode()* para cada uno de los servicios de la clase actual. No se controlan aquellos métodos que hayan sido heredados (evitamos controlar más de una vez). Establece el handler de código intermedio, *ICGenerator*, a cada nodo a controlar.

generateCode()
Cada *BlockNode* está compuesto de una lista de sentencias. El *generateCode()* en un *BlockNode* invoca *generateCode()* para cada una de las sentencias que componen esa lista.
Dependiendo el tipo de nodo del árbol AST que constituye el cuerpo del servicio se realizará la generación de código correspondientes. Ejemplo, si es un nodo binario se invocará a *generateCode()* para la expresión a izquierda y para la expresión a derecha, y también se pondrá en la pila el operador correspondiente (e.g., si el operador es '+' se hace un ADD).

La generación de código del programa finaliza cuando se realizó la generación de código correspondiente a la última sentencia del cuerpo del último servicio declarado en el código fuente.

Implementación

ICEntry - Representa una instrucción CeIASM. Esta compuesta por una instrucción, su offset (opcional) y un comentario (opcional).

ICGenerator - Clase encargada de generar las instrucciones CeIASM y de volcarlas en el archivo de salida.

InputReader - *Handler* del archivo. Se encarga de abrir, leer y retornar los caracteres leídos a la clase Tokenizer y, posteriormente, cerrar el archivo.
Por cuestiones de eficiencia se implementó con un buffer BufferedReader el cual procesa el archivo línea por línea.
Al realizarlo de esta manera nos aseguramos de que el encoding quede encapsulado dentro de la clase y que el analizador léxico funcione correctamente para diferentes plataformas.

LexicalException - Manejo de excepciones para el analizador léxico.

Main - Clase principal del programa. Puede usarse a través de línea de comandos indicando como argumentos el nombre del archivo con el código fuente a procesar y el nombre del archivo de salida. Se mostrará por pantalla si el análisis semántico fue exitoso, y en caso contrario, notificará al usuario con el mensaje con el error correspondiente. Si el análisis semántico fue exitoso se generará en el archivo de salida el código CeIASM correspondiente al código fuente del archivo de entrada.

Parser - Clase encargada de realizar el análisis sintáctico y la primer pasada del análisis semántico del código fuente pasado por parámetro. Implementa un analizador sintáctico descendente donde la entrada se analiza de izquierda a derecha con derivación a izquierda.
Se implementa un método por cada no-terminal de la gramática modificada de MiniJava, encargados de llevar un flujo recursivo de análisis correspondiente. Para esto requiere los tokens generados por la clase Tokenizer. Además crea la tabla de símbolos.

SemanticAnalyzer - Clase encargada de realizar la segunda pasada del análisis semántico (control de sentencias y control de declaraciones) utilizando la tabla de símbolos generada por el parser. Además se encarga de invocar al generador de código intermedio.

SemanticException - Manejo de excepciones para el analizador semántico.

SyntacticException - Manejo de excepciones para el analizador sintáctico.

Token - Representación lógica de un token. Cada token tiene asociado un nombre, un lexema y el número de línea en el que se encontró.

Tokenizer - Clase encargada de llevar adelante el proceso de tokenización. Cuenta con métodos adicionales para saltear comentarios simples y comentarios en bloque.

AssignNode - Nodo del AST que representa una asignación.

BinaryExpressionNode - Nodo del AST que representa una expresión binaria.

BlockNode - Nodo del AST que representa una bloque de sentencias.

CallNode - Nodo del AST que representa una llamada.

ExpressionCallNode - Nodo del AST que representa una llamada de la forma *(expresion).metodo() .*

ExpressionNode - Nodo del AST que representa una expresión.

ForNode - Nodo del AST que representa una sentencia For.

IdMethodCallNode - Nodo del AST que representa una llamada de la forma *id.metodo() .*

IfThenElseNode - Nodo del AST que representa una sentencia If-Then-Else.

LiteralNode - Nodo del AST que representa un literal.

MethodCallNode - Nodo del AST que representa una llamada de la forma *metodo() .*

NewNode - Nodo del AST que representa la palabra reservada *new .*

PrimaryNode - Nodo del AST que representa una expresión primaria.

ReturnExpNode - Nodo del AST que representa la sentencia *return (expresion) ;*

ReturnNode - Nodo del AST que representa la sentencia *return;* .

SentenceNode - Nodo del AST que representa una sentencia.

SeparatorNode - Nodo del AST que representa el separador de sentencias ; .

SimpleSentenceNode - Nodo del AST que representa una sentencia simple.

ThisNode - Nodo del AST que representa la palabra reservada *this*.

UnaryExpressionNode - Nodo del AST que representa una expresión unaria.

WhileNode - Nodo del AST que representa una sentencia While.

ClassEntry - Entrada de la tabla de símbolos que representa una clase.

ConstructorEntry - Entrada de la tabla de símbolos que representa un constructor.

InstanceVariableEntry - Entrada de la tabla de símbolos que representa una variable de instancia.

LocalVariableEntry - Entrada de la tabla de símbolos que representa una variable local.

MethodEntry - Entrada de la tabla de símbolos que representa un método.

ParameterEntry - Entrada de la tabla de símbolos que representa un parámetro.

ServiceEntry - Entrada de la tabla de símbolos que representa un servicio. Un servicio puede ser un constructor o un método.

SymbolTable - Representa la tabla de símbolos.

VariableEntry - Entrada de la tabla de símbolos que representa una variable. Una variable puede ser una variable de instancia, una variable local, o un parámetro.

BooleanType - Representa el tipo boolean.

CharType - Representa el tipo char.

ClassType - Representa un tipo clase (definido por el programa).

IntegerType - Representa el tipo entero.

PrimitiveType - Representa un tipo primitivo. Un tipo primitivo puede ser boolean, char, integer, String o void.

StringType - Representa el tipo String.

Type - Representa un tipo. Un tipo puede ser un tipo primitivo o un tipo clase.

VoidType - Representa un tipo void.

Batería de tests y manejo de errores

La batería de tests está compuesta de un conjuntos de tests exitosos y de un conjunto de tests que deben fallar.

Tests que deben terminar exitosamente

- *asignaciones.java*
Output esperado:
```
@cei
```

- *bug_labels_clases_metodos_case_sensitive.java*
Output esperado:

- *bug_VTs_herencia.java*
Output esperado:
```
1
2
3
```

- *clases_con_mismo_nombre.java*
Output esperado:
```
A :: A()
A :: m()
1
a :: a()
a :: m()
4
```

- *for.java*
Output esperado:
```
A :: A() 4
A :: forloop()
for [
while ( 0 1 2 3 ) Iteracion 0
while ( 0 1 2 3 ) Iteracion 1
while ( 0 1 2 3 ) Iteracion 2
while ( 0 1 2 3 ) Iteracion 3
]
```

- *herencia.java*
Output esperado:
```
A3 :: A3() :: i1
3
A3 :: A3() :: i3
4
A3 :: A3() :: i5
5

A3 :: print() :: i1
A3 :: geti1() :: i1
3

A3 :: print() :: i3
A3 :: geti3() :: i3
4
```

```
A3 :: print() :: i5
A3 :: geti5() :: i5
5

A3 :: A3() :: i1
1
A3 :: A3() :: i3
2
A3 :: A3() :: i5
3

A :: printA() :: i1 A3 :: geti1() :: i1
1
```

- *if.java*

Output esperado:

```
A :: A()
Parametros: 0 - a - Lorem - false
A :: setInteger()
A :: getInteger()
A :: print()A :: getInteger()
2
A :: less()
Valor i = 2
Valor n = 3
Es maximo! n es mayor que i
Valor a = 1
Valor b = 3
```

- *interaccion_clases_multiplesmains.java*

Output esperado:

```
A2 :: A2()
7
8
A2 :: print() :: i1
A2 :: geti1()
7
A2 :: print() :: i2
A2 :: geti2()
8
```

- *interaccion_clases.java*

Output esperado:

```
A2 :: A2()
1
2
A2 :: print() :: i1
A2 :: geti1()
1
A2 :: print() :: i2
A2 :: geti2()
2
A2 :: setO()
A :: A()
3
4
A :: print() :: i1
A :: geti1()
3
A :: print() :: i2
A :: geti2()
4
A :: setO()
```

```
A :: A()
5
6
A :: print() :: i1
A :: geti1()
5
A :: print() :: i2
A :: geti2()
6
```

- *llamada_directa_estatico.java*

Output esperado:

```
A1 :: main()
A :: A()
1
2
A2 :: A2()
3
4
A2 :: print() :: i1
A2 :: geti1()
3
A2 :: print() :: i2
A2 :: geti2()
4
A2 :: main()
5
```

- *llamada_encadenada_estatico.java*

Output esperado:

```
A :: calc() :: A.random() = 11
A :: calc() :: A2.random() = 100
A :: calc() :: A3.retA3().hello() = Hello, world!
A :: calc() :: foo1().foo2().foo3() = 120
```

- *llamada_estatico_this.java*

Output esperado:

```
A1 :: main()
A :: A()
1
2
A2 :: A2()
3
4
A2 :: print() :: i1
A2 :: geti1()
3
A2 :: print() :: i2
A2 :: geti2()
4
A2 :: main()
A2 :: A2()
1
2
5
```

- *llamadas_directas.java*

Output esperado:

```
A
A2
B
A
A
```

```
B
```

- *llamadas.java*

Output esperado:

```
A :: getName()
A :: getName()
A
A2 :: getName()
A2
B :: getName()
B
A :: getName()
A
A :: getName()
A
B :: getName()
B
B :: getName()
B
B :: getName()
B
B :: getName()
B
Main :: getName()
Main
B :: getName()
B
Main :: getName()
Main
Main :: getName()
Main
Main :: getName()
Main
```

- *null.java*

Output esperado:

```
A :: print() :: s1 Hola
A :: print() :: s2 Mundo
A2 :: print() :: i1 4
A2 :: print() :: i2 6
```

- *retorno_clase_inicializado.java*

Output esperado:

- *retorno_clase_sin_inicializar.java*

Output esperado:

- *retorno_string_inicializado.java*

Output esperado:

```
A :: main()
Hola, mundo!
Compiladores e Interpretes
Minijava
2013
```

- *retorno_string_sininicializar.java*

Output esperado:

```
×
```

- *sentencia_simple.java*

Output esperado:

```
A :: main()
A :: foo1()
A :: foo1()
A :: foo2()
A :: foo3()
```

- *subbloques.java*

Output esperado:

```
A :: print() :: i1 1
A :: print() :: i2 4
A2 :: A2()
i1 == 1
i2 == 6
B :: print() :: i1 1
B :: print() :: i2 6
```

- *this_sentencias.java*

Output esperado:

```
A :: _this()
A :: fooA()
A :: _sentencias()
10B :: fooB()
```

- *this.java*

Output esperado:

```
A :: main() :: ret
11
```

- *while.java*

Output esperado:

```
A :: A() Hello, world!
A :: whileloop()
Iteracion 1
Iteracion 2
Iteracion 3
Iteracion 4
Iteracion 5
Iteracion 6
Iteracion 7
Iteracion 8
Iteracion 9
Iteracion 10
Iteracion 11
Iteracion 12
Iteracion 13
Iteracion 14
Iteracion 15
Iteracion 16

Valor i = 20
Valor j = 20
```

- *bst.java*

Output esperado:

```
Se insertaran los siguientes animales:
15 -> Oso
45 -> Perro
56 -> Elefante
1 -> Koala
12 -> Leon
543 -> Tigre
```

```
156 -> Gato
34 -> Leopardo
26 -> Loro
11 -> Tucan
100 -> Pollo
320 -> Caballo
800 -> Vaca
810 -> Toro
901 -> Ardilla
43 -> Coyote
2 -> Lobo

Obtener: 543 > Tigre
Obtener: 11 > Tucan
Obtener: 2 > Lobo
Obtener: 810 > Toro
Obtener: 901 > Ardilla

Eliminar 543.
Eliminar 320.
Eliminar 901.

Obtener: 543 > No se ha encontrado.
Obtener: 901 > No se ha encontrado.
Obtener: 320 > No se ha encontrado.
Obtener: 43 > Coyote
Obtener: 12 > Leon
```

- *chars.java*

Output esperado:

```
       a
b
```

- *comparaciones.java*

Output esperado:

```
Equals (==)
Strings: NOT OK (EXPECTED)
Not Equals (!=)
Greater (>)
Greater Than (>=)
Lesser (<)
Lesser Than (<=)
```

- *empleados.java*

Output esperado:

```
Empleado   1000
```

- *estaticos.java*

Output esperado:

```
(7 * 5 + 5 ^ 2)/6 = 10
```

- *for.java*

Output esperado:

```
FOR:
0-1-2-3-4-5
```

- *invertir_numero.java*

Output esperado:

```
1231853211
```

- *jerarquia_herencia.java*

Output esperado:

```
123
```

- *LinkedSearchBinaryTree.java*

Output esperado:

```
Se insertaran los siguientes animales:
15 -> Oso
45 -> Perro
56 -> Elefante
1 -> Koala
12 -> Leon
543 -> Tigre
156 -> Gato
34 -> Leopardo
26 -> Loro
11 -> Tucan
100 -> Pollo
320 -> Caballo
800 -> Vaca
810 -> Toro
901 -> Ardilla
43 -> Coyote
2 -> Lobo

Obtener: 543 > Tigre
Obtener: 11 > Tucan
Obtener: 2 > Lobo
Obtener: 810 > Toro
Obtener: 901 > Ardilla

Eliminar 543.
Eliminar 320.
Eliminar 901.

Obtener: 543 > No se ha encontrado.
Obtener: 901 > No se ha encontrado.
Obtener: 320 > No se ha encontrado.
Obtener: 43 > Coyote
Obtener: 12 > Leon
```

- *listas.java*

Output esperado:

```
0
1
2
3
4
5
```

- *llamadas.java*

Output esperado:

```
Secuenciales
A(int)
A.m()
B(int)
B.m()
C(int)
C.m()
D(int)
D.m()
12345

Anidadas
A(int)
```

```
A.m()
B(int)
B.m()
C(int)
C.m()
D(int)
D.m()
54321
```

- *mayor.java*
Output esperado:
Ingrese un num:

- *piramide.java*
Output esperado:

```
*
**
***
****
*****
******
*******
********
*********
**********
```

- *polimorfismo.java*
Output esperado:

```
A.m1
111
A.m2
222

B.m1
222
A.m2
0

B.m1
0
C.m2
999
```

- *recursivos.java*
Output esperado:

```
13
28
5040
```

- *returnnull.java*
Output esperado:

- *sda.java*
Output esperado:

```
Objeto A:
VariableI A: 1

Objeto B (extiende a A):
VariableI A: 1
VariableI B: 2
```

```
Objeto C (extiende a B):
VariableI A: 1
VariableI B: 2
VariableI C: 3
```

- *strings.java*

Output esperado:

```
hola mundo
hola mundo
hola mundo
```

- *while.java*

Output esperado:

```
* if true: Met1: 10
* else true: Met1: 10
* if-else false: Met1: 10
```

Tests que deben fallar

- *acceso_invalido.java*

Output esperado:

```
Null pointer coming!

Error en tiempo de ejecución: Una instrucción intentó asignar al registro pc el valor
259 con DEP habilitada. Program counter (pc) = 239.
```

- *bst_nullroot.java*

Output esperado:

```
HOLA
Mariano
Ale
Feffo

Error en tiempo de ejecución: Desreferenciameinto de puntero nulo en LOADREF. Program
counter (pc) = 2323.
```

- *null_pointer.java*

Output esperado:

```
Error en tiempo de ejecución: Desreferenciameinto de puntero nulo en LOADREF. Program
counter (pc) = 655.
```

- *stack_overflow.java*

Output esperado:

```
A :: _this()
A :: fooA()
A :: _sentencias()
10A :: _this()
A :: fooA()
A :: _sentencias()
(...)

Error en tiempo de ejecución: Desbordamiento de pila (stack overflow). Program counter
(pc) = 124.
```

- *string_no_inicializada.java*

Output esperado:

ũΩÏũũũâ

Error de acceso a memoria en la locación 8192: Locación de memoria no implementada.
Program counter (pc) = 113.

Modo de uso

Requerimientos del sistema

- Java Runtime Environment 7
- Ejecutable de la máquina virtual de Compiladores e Intérpretes (CeIVM) v1.1.0 (CeIVM-cei2011.jar)

Compilación

Para ejecutar un programa MiniJava se debe abrir una terminal e ingresar el siguiente comando:

```
java -jar IntermediateCodeGeneration.jar <ARCHIVO_ENTRADA> [<ARCHIVO_SALIDA>]
```

Donde,
<ARCHIVO_ENTRADA> es el nombre o ruta del archivo fuente de MiniJava a compilar. Si se ingresa sólo el nombre del archivo, éste debe estar ubicado dentro de la misma carpeta donde se encuentra el jar del compilador.

[<ARCHIVO_SALIDA>] es un parámetro opcional que representa el nombre o ruta del archivo de código intermedio generado por el compilador. Si se omite este parámetro, se generará la salida en un archivo con un nombre igual al del archivo de entrada seguido de ".CeIASM.txt". El archivo de salida será generado en la misma carpeta en que se encuentra el jar del compilador.

Se espera que las rutas indicadas sean rutas absolutas. De ingresarse un nombre o ruta de archivo inválida o inexistente, se notificará al usuario con un error al intentar abrir el archivo.
Si la compilación del programa es exitosa, el compilador mostrará un mensaje informando esta situación y generará el archivo de código intermedio. En cambio, si se produce algún error durante el proceso de compilación, la salida del compilador será un mensaje informando el error y el número de línea donde se ha producido.

Ejemplos de uso:

```
java -jar MiniJava.jar input.txt output.txt
java -jar MiniJava.jar /home/minijava /home/ceiasm
java -jar MiniJava.jar input.txt (en este caso el nombre del archivo de salida será
input.txt.CEIASM.txt).
```

Ejecución

Para ejecutar ejecutar el archivo de código intermedio CeIASM generado por el compilador en la máquina virtual CeIVM se debe ingresar en la terminal el siguiente comando:

```
java -jar CeIVM-cei2011.jar <ARCHIVO_ORIGEN> [-v [<ARCHIVO_LISTADO>]]
```

Donde,
<ARCHIVO_ORIGEN> es el nombre o ruta del archivo que contiene el código intermedio a ejecutar. Éste debe estar ubicado dentro de la misma carpeta donde se encuentra el jar de la máquina virtual. Este

archivo de origen debería ser el archivo de salida de la compilación previamente hecha.

`[-v]` es un parámetro opcional que genera listados con información de ensamblado, linkeo y carga en un archivo con un nombre igual del archivo de origen seguido de ".list.txt".

`[-v [<ARCHIVO_LISTADO>]]` es un parámetro opcional que genera listados con información de ensamblado, linkeo y carga en un archivo con el nombre `<ARCHIVO_LISTADO>`.

Se espera que las rutas indicadas sean rutas absolutas. De ingresarse un nombre o ruta de archivo inválida o inexistente, se notificará al usuario con un error al intentar abrir el archivo.
Si la ejecución del programa es exitosa, se mostrarán por consola todas las salidas por pantalla del programa seguidas de un mensaje informando la ejecución exitosa. En cambio, si se produce algún error durante la ejecución, se mostrará por pantalla un mensaje informando el error.

Ejemplos de uso:

```
java -jar CeIVM-cei2011.jar input.CeIASM.txt
java -jar CeIVM-cei2011.jar input.CeIASM.txt -v
java -jar CeIVM-cei2011.jar input.CeIASM.txt -v listado.txt
```

Decisiones de diseño y limitaciones

Lenguaje

- En nuestra implementación pueden existir varias clases con el método principal *main*. Por defecto se elegirá la primera.

- Los métodos estáticos pueden ser invocados directamente si están definidos en la misma clase en la que se invocan, e.g. `métodoEstaticoEnClase()`, o a través del operador . (punto) teniendo como lado izquierdo,
    - El nombre de la clase en el que está definido el método estático. Ejemplo: `(A.metodoEstaticoEnA())`

  Si bien es desaconsejado debido a que se pierde la noción de métodos de clase, también puede tenerse como lado izquierdo,

    - Una instancia de la clase en la que está definido el método estático. Ejemplo: `var A a1 = new A(); (a1.metodoEstaticoEnA());`

    - Una expresión que retorne una instancia de la clase en la que está definido el método estático. Ejemplo: `((new A()).metodoEstaticoEnA()); ((this).metodoEstaticoEnA())`

  Naturalmente también es posible realizar llamadas encadenadas con método estáticos y dinámicos.
  Esta decisión de diseño se basa en los lineamientos de Java SE 7 (ver http://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html).

- No es posible declarar variables de instancia cuyo nombre sea igual al nombre de la clase en la que están definidas o igual al nombre de los métodos definidos dentro de la clase.

- Dado que la clase *System* es una clase constituida únicamente de variables y métodos estáticos, no es posible:
    - Extender a la clase System.
    - Crear instancias de la clase System.
    - Declarar variables de tipo System.
    - Definir métodos con tipo de retorno System.

Compilador

- Para la implementación de tabla de símbolos y sus diferentes entradas se hizo uso de la estructura *LinkedHashedMap*. Esta tabla hash posee la eficiencia de una tabla hash pero con la propiedad adicional de mantener el orden en el que se fueron agregando las entradas.

- No realizamos ningún control sobre sentencias inalcanzables, caminos sin retorno o múltiples retornos. Por ejemplo:

```
static int sinRetorno() {

}
```

La definición de este método es válida y su invocación retorna 0.

```
static int sentenciaInalcanzable()
var x;
{
	return 1;
	x = x + 1;
}
```

La invocación a este método retorna 1 y el compilador no informará que la sentencia `x = x + 1` es una sentencia inalcanzable.

```
static int multiplesRetornos() {
	return 0;
	return 1;
}

La definición de este método es válida y retorna 0.
```

- El compilador podría ser optimizado haciendo que los métodos de la clase System (`System.read, System.println, System.printI`, etc) sean llamados *en línea* (haciendo un PUSH de la instrucción a nivel máquina), en lugar de construir la estructura del método.

- Otra optimización sería posible implementando las llamadas de forma recursiva en lugar de hacerlo iterativamente. Es decir, en vez de contar con nodos AST con listas de llamadas, podríamos tener un nodo AST llamada que a su vez tenga un nodo AST llamada. De esa forma delegaríamos los controles y la generación de código a cada nodo interior en vez de hacer un recorrido exhaustivo como en el compilador presentado.

Cambios realizados para la defensa

- Dado que MiniJava es *case-sensitive* y CeIASM no lo es, debimos cambiar el formato de las etiquetas agregando *labels* secuenciales a las clases y a los métodos y así evitar la colisión de nombres. Con esta modificación, por ejemplo, es posible declarar una clase llamada “Clase” y otra llamada “clase”.

- Durante el *testing* del compilador detectamos un *bug* en el acceso a las VTs manifestado por un error en ejecución (precisamente, un stack overflow) al momento de correr el código CeIASM en la máquina virtual provista por la cátedra. Esto se debía a la forma en que se insertaban los métodos en la VT. Al realizar la consolidación de herencia luego de la inserción de todos los métodos de la clase y también debido a la estructura de datos utilizada para llevar los métodos, los métodos heredados se insertaban al final siendo que estos debían estar ordenados según su offset.

- Se corrigió el número de línea que se mostraba en los mensajes de error cuando ocurrían errores semánticos en la condiciones de las sentencias *While, For, If-Then* e *If-Then-Else.*

- Ahora se pueden crear objetos de tipo dinámico Object. Antes, al ejecutar una sentencia del tipo var o = new Object() ocurría excepción (a nivel Java) ya que no se le había establecido un constructor default a la clase Object. También se muestra el mensaje de error adecuado cuando se intenta definir una variable de tipo dinámico Object cuyo tipo estático no sea Object, ya que el tipo Object no conforma a otro tipo que no sea a si mismo.

- Se corrigió el mensaje de error que se mostraba cuando los parámetros actuales de un método no conformaban a sus parámetros formales. El problema era que no se estaba actualizando correctamente el índice que llevaba el control de la posición del parámetro que se estaba controlando, entonces si bien el control se realizaba correctamente, el mensaje de error mostraba, por ejemplo, el tipo del primer parámetro formal y el tipo del segundo parámetro actual.

- Se agregaron controles adicionales para la colisión de nombres: Ahora una variable de instancia no puede tener el mismo nombre que la clase o algún método definido dentro de la clase.

- Dado que la clase *System* es una clase constituida únicamente de variables y métodos estáticos, no es posible:
    - Extender a la clase System.
    - Crear instancias de la clase System.
    - Declarar variables de tipo System.
    - Definir métodos con tipo de retorno System.