Manual Técnico

Mini Java

26/12/2018

Germán A. Gómez

Índice

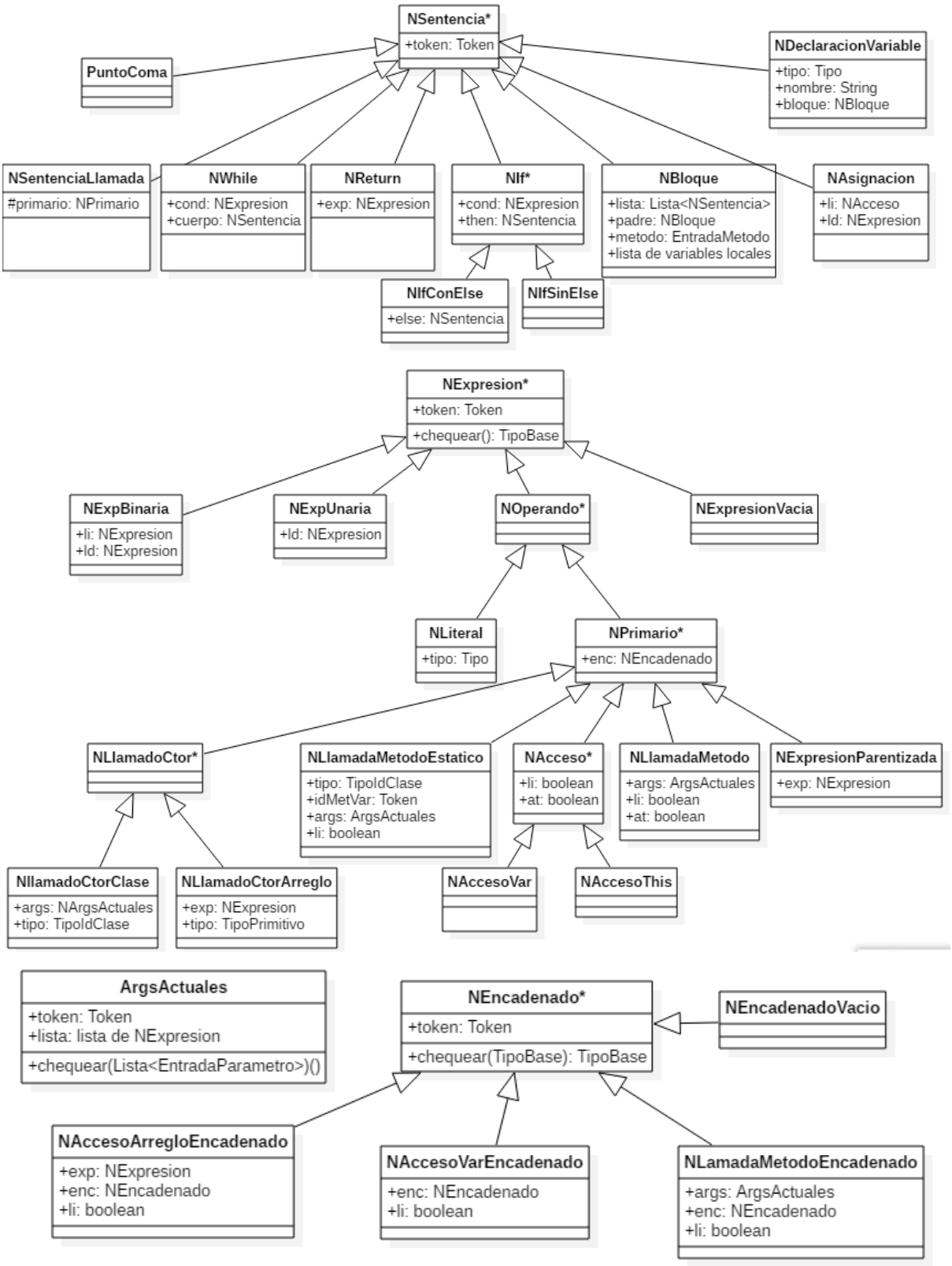
MiniJava

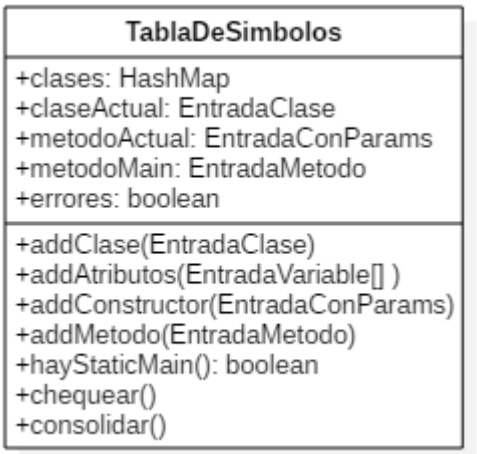
MiniJava es una simplificación del lenguaje Java. En este lenguaje no se considerarán elementos avanzados de Java tales como genericidad, interfaces, métodos abstractos, excepciones, hilos y sincronización, entre otros. MiniJava impone algunas restricciones sintácticas adicionales al lenguaje, como por ejemplo el uso de palabras reservadas especiales para declarar métodos de instancia, o el uso de la asignación como una sentencia en lugar de ser parte de una expresión. Por otra parte, algunos constructores tendrán una semántica diferente al lenguaje Java, como por ejemplo el modificar de visibilidad protected.

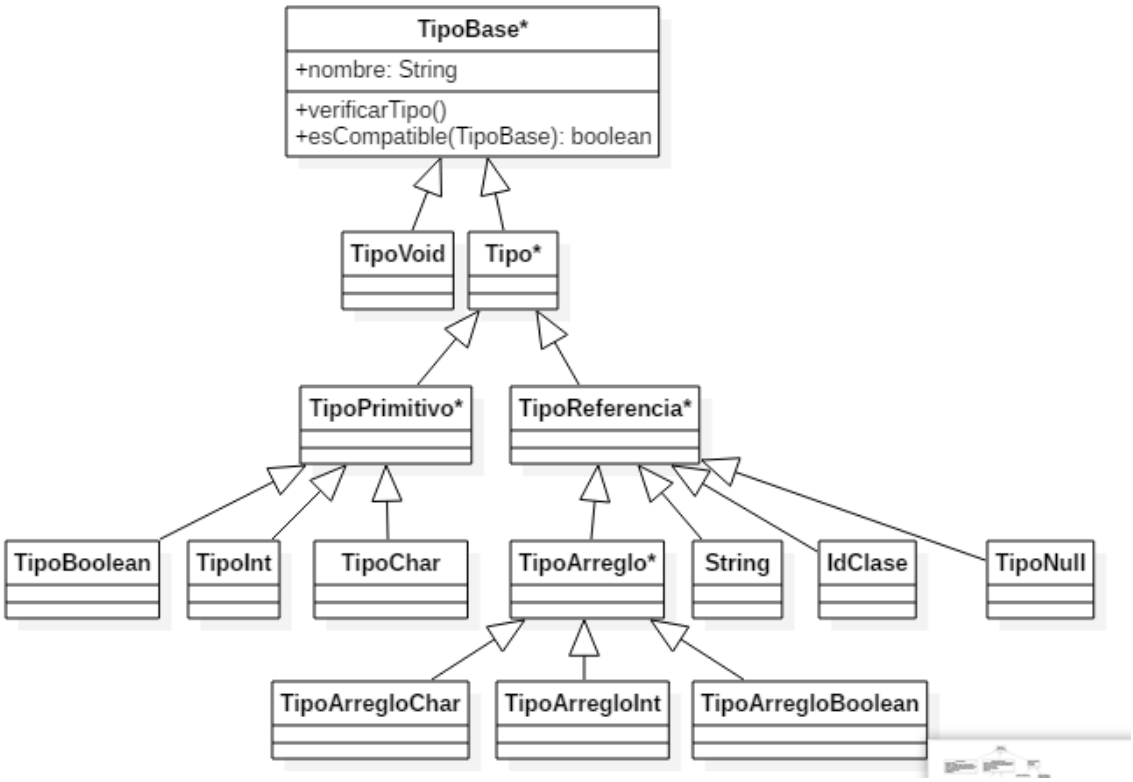
Arquitectura del compilador

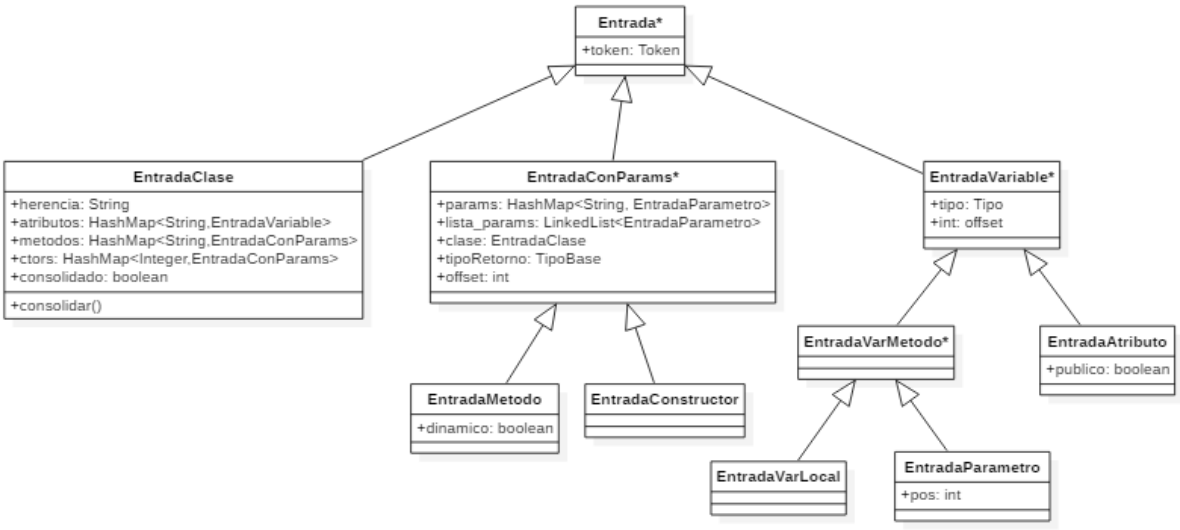
El compilador recibe un programa fuente de MiniJava. El analizador léxico le va brindando Tokens al analizador sintáctico, y este último, una vez consumido el Token, le pide otro al analizar léxico. El analizador Sintáctico genera una estructura denominada Tabla de símbolos. Luego, el analizador semántico, verifica las declaraciones y sentencias recolectadas en la etapa del analizador sintáctico. Por último, en una nueva pasada, genera código de representación intermedia.

Diagrama de clases




TablaDeSimbolos
+clases: HashMap
+claseActual: EntradaClase
+metodoActual: EntradaConParams
+metodoMain: EntradaMetodo
+errores: boolean
+addClase(EntradaClase)
+addAtributos(EntradaVariable[])
+addConstructor(EntradaConParams)
+addMetodo(EntradaMetodo)
+hayStaticMain(): boolean
+chequear()
+consolidar()


TipoBase*
+nombre: String
+verificarTipo()
+esCompatible(TipoBase): boolean
TipoVoid
Tipo*
TipoPrimitivo*
TipoReferencia*
TipoBoolean
TipoInt
TipoChar
TipoArreglo*
String
IdClase
TipoNull
TipoArregloChar
TipoArregloInt
TipoArregloBoolean


Entrada*
+token: Token
EntradaClase
+herencia: String
+atributos: HashMap<String,EntradaVariable>
+metodos: HashMap<String,EntradaConParams>
+ctors: HashMap<Integer,EntradaConParams>
+consolidado: boolean
+consolidar()
EntradaConParams*
+params: HashMap<String, EntradaParametro>
+lista_params: LinkedList<EntradaParametro>
+clase: EntradaClase
+tipoRetorno: TipoBase
+offset: int
EntradaVariable*
+tipo: Tipo
+int: offset
EntradaMetodo
+dinamico: boolean
EntradaConstructor
EntradaVarMetodo*
EntradaAtributo
+publico: boolean
EntradaVarLocal
EntradaParametro
+pos: int

Análisis Léxico

Tiene como propósito recibir un archivo de entrada, analizar su contenido y entender su significado para generar Tokens.

Los Tokens serán de vital importancia en las siguientes partes del compilador. Nos ayudan a guardar información adicional de una simple cadena de caracteres:

- La cadena analizada
- Numero de línea y columna
- Tipo de la cadena analizada

Por lo cual su propósito es limpiar los espacios en blanco, salto de línea y comentarios, e ir entregando Tokens que otro componente del compilador le va exigiendo.

Esta etapa usa un autómata finito para construir los Tokens e ir encontrando errores en el código fuente. Un error es encontrado cuando un Token no se puede ligar al alfabeto del lenguaje MiniJava.

Finaliza cuando se encuentra un error o cuando ya se leyó todo el archivo de entrada.

Alfabeto

El alfabeto reconocido por el analizador léxico, es el código ASCII exceptuando al 0, por lo cual son válidos los caracteres entre 1 a 255 inclusive.

Tokens

A continuación se encuentran los tipos de Tokens que se puede generar a través del archivo de entrada de código fuente.

Palabras reservadas

- ‘class’: class
- ‘string’: string
- ‘public’: public
- ‘if’: if
- ‘this’: this
- ‘extends’: extends

- ‘boolean’: boolean
- ‘private’: prívate
- ‘else’: else
- ‘new’: new
- ‘static’: static
- ‘char’: char
- ‘void’: void
- ‘while’: while
- ‘true’: true
- ‘false’: false
- ‘dynamic’: dynamic
- ‘int’: int
- ‘null’: null
- ‘return’: return

Expresión regular = class | String | public | if |this | extends |boolean |private | else | new | static | dynamic | void | while |true |int | null | return | false.

Identificadores

- IdClase: identificador de clase. Es una letra mayúscula seguida de cero o más letras (mayúsculas o minúsculas), dígitos y underscores.

Expresión regular = [a..z]([a..z] | [A..Z] | [0..9] | ‘_’)*

- IdMetVar: identificador de método y variable. Es una letra minúscula seguida de cero o más letras (mayúsculas o minúsculas), dígitos y underscores. Expresión regular = [A..Z]([a..z] | [A..Z] | [0..9] | ‘_’)*

Comentarios

- '/* comentario */’: Comentarios multi-línea. Todos los caracteres desde /* hasta */ son ignorados. No son Tokens necesarios para el analizador, una vez que se reconoce uno de ellos, no lo guarda.

Expresión regular = /*(carac_ascii)**/

- ‘// comentario’: Comentario simple. Todos los caracteres de desde // hasta el final de la línea son ignorados. No son Tokens necesarios para el analizador, una vez que se reconoce uno de ellos, no lo guarda.

Expresión regular = //(carac_ascii)* ’/n’

Literales enteros

- ‘entero’: secuencia de uno o más dígitos. El valor de un literal entero corresponde a su interpretación estándar en base 10.

Expresión regular = ([0..9])+

Literales caracteres

- ‘caracter’: es un carácter encerrado entre comillas simples. Por ejemplo: ‘a’, ‘/n’, ‘/t’.

Expresión regular = ‘(carac_ascii | (/carac_ascii))’

Strings

- ‘string’: Un literal string se representa mediante una comilla doble (") seguida de una secuencia de caracteres y finaliza con otra comilla doble ("). El valor del literal corresponde a la cadena de caracteres entre las comillas. Se permite el uso de ‘/n’ para salto de línea y ‘/t’ para tab.

Expresión regular = “(carac_ascii) + “

Puntuación

- ‘(’: paréntesis que abre (‘(’).
- ‘)’: paréntesis que cierra (‘)’).
- ‘{’: llave que abre (‘{’).
- ‘}’: llave que cierra (‘}’).
- ‘;’: punto y coma (‘;’).
- ‘.’: punto (‘.’).
- ‘,’: coma (‘,’).
- ‘[’: corchete que abre (‘[’).
- ‘]’: corchete que cierra (‘]’).

Operadores

- ‘>’: mayor (‘>’).

- ‘<’ : menor (‘<’).
- ‘!’: negación (‘!’).
- ‘==’: igualdad(‘==’).
- ‘>=’: mayor o igual (‘>=’).
- ‘<=’: menor o igual (‘<=’).
- ‘!=’: distinto (‘!=’).
- ‘+’: más (‘+’).
- ‘-’: menos (‘-’).
- ‘*’: multiplicación (‘*’).
- ‘/’: división (‘/’).
- ‘&&’: and (‘&&’).
- ‘||’: or (‘||’).
- ‘=’: asignación (‘=’).

Fin de archivo

- ‘eof’: fin de archivo.

Autómata finito

El analizar léxico utiliza un autómata finito para construir Tokens y encontrar errores. Puede ver una imagen del autómata finito en el siguiente link: http://bit.ly/2rYgk97

Consideraciones de diseño

El analizador léxico se implementó de la forma switch case, dado que su eficiencia sobre su alternativa subprogramas. En código fuente tiene una cantidad considerables de comentarios para mejorar la legibilidad.

El programa elimina las comillas dobles que contiene a un String, por ejemplo: “hola” se guarda como hola. De forma contraria, se decidió guarda las comillas simples en los caracteres, por lo cual ‘a’ se guarda como ‘a’.

El lector del archivo de entrada, utiliza la función read() que nos facilita la clase BufferedReader. Si bien la documentación especifica que cuando no hay más caracteres devuelve el entero -1, en muchas ocasiones devuelve 65535. Se tomó la convención de que en el caso que se reciba -1 o 65535, archivo devolverá un 0 como próximo carácter, lo cual el analizador léxico lo interpretará con fin de archivo.

Análisis Sintáctico

La gramática original de MiniJava está hecha en formato EBNF. Este tipo de gramática no es apta para el desarrollo de un analizador sintáctico. Por lo cual es necesario hacer transformaciones:

- Pasar de una gramática EBNF a BNF.
- Eliminar recursión a izquierda.
- Factorización de la Gramática.

Luego se usó esta gramática en el compilador de MiniJava para garantizar que el programa era correcto sintácticamente.

El analizador sintáctico está hecho de forma descendiente recursiva, la cual simula el proceso de derivación partiendo desde el primer no terminal hasta llegar a la cadena de entrada.

Etapas de la gramática

Etapa 0: EBNF a BNF

<Inicial> ::= <Clase><MasClase>

<MasClase>::= ε | <Clase><MasClase>

<Clase> ::= **classidClase**<Herencia>**{** <Miembro>**}**

<Miembro> ::= <Atributo><Miembro> | <Ctor><Miembro> | <Metodo><Miembro> | ε

<Herencia> : := ε | **extendsidClase**

<Atributo> ::= <Visibilidad><Tipo><ListaDecVars> **;**

<Metodo> ::= <FormaMetodo><TipoMetodo>**idMetVar**<ArgsFormales><Bloque>

<Ctor> ::=**idClase**<ArgsFormales><Bloque>

<ArgsFormales> ::= (<ListaArgsFormales>)

<ListaArgsFormales> ::= ε | <ArgFormal>

<ListaArgsFormales> ::= <ArgFormal> **,**<ListaArgsFormales>

<ArgFormal> ::= <Tipo>**idMetVar**

<FormaMetodo> ::= **static** | **dynamic**

<Visibilidad> ::= **public** |**prívate**

<TipoMetodo> ::= <Tipo> | **void**

<Tipo> ::= <TipoPrimitivo> | <TipoReferencia>

<TipoPrimitivo> ::=**boolean** | **char** | i**nt**

<TipoReferencia> ::=idClase | S**tring** | <TipoPrimitivo>**[]**

<ListaDecVars> ::= **idMetVar**

<ListaDecVars> ::= **idMetVar ,**<ListaDecVars>

<Bloque> ::= **{**<MasSentencia>**}**

<MasSentencia> ::= ε | <Sentencia><MasSentencia>

<Sentencia> ::= **;**

<Sentencia> ::= <Asignacion>**;**

<Sentencia> ::= <SentenciaLlamada>**;**

<Sentencia> ::= <Tipo><ListaDecVars>**;**

<Sentencia> ::= **if**(<Expresion>) <Sentencia>

<Sentencia> ::= **if**(<Expresion>) <Sentencia>**else**<Sentencia>

<Sentencia> ::= **while** (<Expresion>) <Sentencia>

<Sentencia> ::= <Bloque>

<Sentencia> ::=**return**<ExpresionOpcional>>;

<Asignacion> ::=<AccesoVar>**=**<Expresion>

<Asignacion> ::= <AccesoThis>**=** <Expresion>

<SentenciaLlamada> ::= **(** <Primario> **)**

<ExpresionOpcional> ::= ε | <ExpOr>

<Expresion> ::= <ExpOr>

<ExpOr> ::= <ExpOr>**||**<ExpAnd> | <ExpAnd>

<ExpAnd> ::=<ExpAnd>**&&**<ExpIg> | <ExpIg>

<ExpIg> ::=<ExpIg><OpIg><ExpComp> | <ExpComp>

<ExpComp> ::=<ExpAd><OpComp><ExpAd> | <ExpAd>

<ExpAd> ::=<ExpAd><OpAd><ExpMul> | <ExpMul>

<ExpMul> ::=<ExpMul><OpMul><ExpUn> | <ExpUn>

<ExpUn> ::= <OpUn><ExpUn> | <Operando>

<OpIg> ::= **==** | **!=**

<OpComp> ::= **<** | **>** | **<=** | **>=**

<OpAd> ::= **+** | **-**

<OpUn> ::= **+** | **-** | **!**

<OpMul> ::= ***** | **/**

<Operando> ::= <Literal>

<Operando> ::= <Primario>

<Literal> ::=null | true | false | intLiteral| charLiteral| stringLiteral

<Primario> ::= <ExpresionParentizada>

<Primario> ::= <AccesoThis>

<Primario> ::= <AccesoVar>

<Primario> ::=<LlamadaMetodo>

<Primario> ::=<LlamadaMetodoEstatico>

<Primario> ::=<LlamadaCtor>

<ExpresionParentizada> ::= (<Expresion>) <Encadenado>

<AccesoThis> ::=**this**<Encadenado>

<AccesoVar> ::= **idMetVar**<Encadenado>

<LlamadaMetodo> ::=**idMetVar**<ArgsActuales><Encadenado>

<LlamadaMetodoEstatico> ::= **idClase** . <LlamadaMetodo><Encadenado>

<LlamdaCtor> ::= **new idClase**<ArgsActuales><Encadenado>

<LlamdaCtor> ::= **new**<TipoPrimitivo>**[** <Expresion>**]** <Encadenado>

<ArgsActuales> ::= **(**<ListaExps>**)**

<ListaExps> ::= ε | <Expresion>

<ListaExps> ::= ε| <Expresion>**,**<ListaExps>

<Encadenado> ::= ε | **.**<LlamadaMetodoEncadenado>

<Encadenado> ::= ε | **.**<AccesoVarEncadenado>

<Encadenado> ::= ε | <AccesoArregloEncadenado>

<LlamadaMetodoEncadenado> ::=**idMetVar**<ArgsActuales><Encadenado>

<AccesoVarEncadenado> ::=**idMetVar**<Encadenado>

<AccesoArregloEncadenado> ::= **[**<Expresion>**]**<Encadenado>

Etapa 1: eliminar recursión a izquierda

<Inicial> ::= <Clase><MasClase>

<MasClase>::= ε | <Clase><MasClase>

<Clase> ::= **classidClase**<Herencia>**{** <Miembro>**}**

<Miembro> ::= <Atributo><Miembro> | <Ctor><Miembro> | <Metodo><Miembro> | ε

<Herencia> : := ε | **extendsidClase**

<Atributo> ::= <Visibilidad><Tipo><ListaDecVars> **;**

<Metodo> ::= <FormaMetodo><TipoMetodo>**idMetVar**<ArgsFormales><Bloque>

<Ctor> ::=**idClase**<ArgsFormales><Bloque>

<ArgsFormales> ::= (<ListaArgsFormales>)

<ListaArgsFormales> ::= ε | <ArgFormal>

<ListaArgsFormales> ::= <ArgFormal> **,**<ListaArgsFormales>

<ArgFormal> ::= <Tipo>**idMetVar**

<FormaMetodo> ::= **static** | **dynamic**

<Visibilidad> ::= **public** |**prívate**

<TipoMetodo> ::= <Tipo> | **void**

<Tipo> ::= <TipoPrimitivo> | <TipoReferencia>

<TipoPrimitivo> ::=**boolean** | **char** | i**nt**

<TipoReferencia> ::=idClase | S**tring** | <TipoPrimitivo>**[]**

<ListaDecVars> ::= **idMetVar**

<ListaDecVars> ::= **idMetVar ,**<ListaDecVars>

<Bloque> ::= **{**<MasSentencia>**}**

<MasSentencia> ::= ε | <Sentencia><MasSentencia>

<Sentencia> ::= **;**

<Sentencia> ::= <Asignacion>**;**

<Sentencia> ::= <SentenciaLlamada>**;**

<Sentencia> ::= <Tipo><ListaDecVars>**;**

<Sentencia> ::= **if**(<Expresion>) <Sentencia>

<Sentencia> ::= **if**(<Expresion>) <Sentencia>**else**<Sentencia>

<Sentencia> ::= **while** (<Expresion>) <Sentencia>

<Sentencia> ::= <Bloque>

<Sentencia> ::=**return**<ExpresionOpcional>>;

<Asignacion> ::=<AccesoVar>**=**<Expresion>

<Asignacion> ::= <AccesoThis>**=** <Expresion>

<SentenciaLlamada> ::= **(** <Primario> **)**

<ExpresionOpcional> ::= ε | <ExpOr>

<Expresion> ::= <ExpOr>

<ExpOr> ::= <ExpAnd><ExpOr2>

<ExpOr2> ::= || <ExpAnd><ExpOr2> | ε

<ExpAnd> ::= <ExpIg> <ExpAnd2>

<ExpAnd2> ::= && <ExpIg> <ExpAnd2> | ε

<ExpIg> ::= <ExpComp> <ExpIg2>

<ExpIg2> ::= <OpIg> <ExpComp> <ExpIg2> | ε

<ExpComp> ::=<ExpAd><OpComp><ExpAd> | <ExpAd>

<ExpAd> ::= <ExpMul> <ExpAd2>

<ExpAd2> ::= <OpAd> <ExpMul> <ExpAd2> | ε

<ExpMul> ::= <ExpUn><ExpMul2>

<ExpMul2> ::= <OpMul><ExpUn><ExpMul2> | ε

<ExpUn> ::= <OpUn><ExpUn> | <Operando>

<OpIg> ::= **==** | **!=**

<OpComp> ::= **<** | **>** | **<=** | **>=**

<OpAd> ::= **+** | **-**

<OpUn> ::= **+** | **-** | **!**

<OpMul> ::= ***** | **/**

<Operando> ::= <Literal>

<Operando> ::= <Primario>

<Literal> ::=null | true | false | intLiteral| charLiteral| stringLiteral

<Primario> ::= <ExpresionParentizada>

<Primario> ::= <AccesoThis>

<Primario> ::= <AccesoVar>

<Primario> ::=<LlamadaMetodo>

<Primario> ::=<LlamadaMetodoEstatico>

<Primario> ::=<LlamadaCtor>

<ExpresionParentizada> ::= (<Expresion>) <Encadenado>

<AccesoThis> ::=**this**<Encadenado>

<AccesoVar> ::= **idMetVar**<Encadenado>

<LlamadaMetodo> ::=**idMetVar**<ArgsActuales><Encadenado>

<LlamadaMetodoEstatico> ::= **idClase** . <LlamadaMetodo><Encadenado>

<LlamdaCtor> ::= **new idClase**<ArgsActuales><Encadenado>

<LlamdaCtor> ::= **new**<TipoPrimitivo>**[** <Expresion>**]** <Encadenado>

<ArgsActuales> ::= **(**<ListaExps>**)**

<ListaExps> ::= ε | <Expresion>

<ListaExps> ::= ε| <Expresion>**,**<ListaExps>

<Encadenado> ::= ε | **.**<LlamadaMetodoEncadenado>

<Encadenado> ::= ε | **.**<AccesoVarEncadenado>

<Encadenado> ::= ε | <AccesoArregloEncadenado>

<LlamadaMetodoEncadenado> ::=**idMetVar**<ArgsActuales><Encadenado>

<AccesoVarEncadenado> ::=**idMetVar**<Encadenado>

<AccesoArregloEncadenado> ::= **[**<Expresion>**]**<Encadena

Etapa 2: factorizar, agregar asignación inline y corrección de errores

<Empezar> ::= <Inicial> **eof**

<Inicial> ::= <Clase><MasClase>

<MasClase>::= <Clase><MasClase> | ε

<Clase> ::= **class idClase**<Herencia>**{** <Miembro>**}**

<Miembro> ::= <Atributo><Miembro> | <Ctor><Miembro> | <Metodo><Miembro> | ε

<Herencia> : := ε | **extends idClase**

<Atributo> ::= <Visibilidad><Tipo><ListaDecVars><InLine> **;**

<InLine> ::= = <Expresion> | ε

<Metodo> ::= <FormaMetodo><TipoMetodo>idMetVar<ArgsFormales><Bloque>

<Ctor> ::=**idClase**<ArgsFormales><Bloque>

<ArgsFormales> ::= **(** <ListaArgsFormales>**)**

<ListaArgsFormales> ::= <ArgFormal><F1> | ε

<F1> ::= **,**<ListaArgsFormales> | ε

<ArgFormal> ::= <Tipo>**idMetVar**

<FormaMetodo> ::= **static** | **dynamic**

<Visibilidad> ::= **public** |**private**

<TipoMetodo> ::= <Tipo> | **void**

<Tipo> ::= **boolean**<F10> | **char** <F10> | **int** <F10> | **idClase** | **String**

<TipoPrimitivo> ::= **boolean** | **char** | **int**

<F10> ::= **[]** | ε

<ListaDecVars> ::= **idMetVar** <F2>

<F2> ::= **,**<ListaDecVars> | ε

<Bloque> ::= **{**<MasSentencia>**}**

<MasSentencia> ::= <Sentencia><MasSentencia> | ε

<Sentencia> ::= **;** | <Asignacion>**;** | <SentenciaLlamada>**;** | <Tipo><ListaDecVars ><InLine>**;**
|

if(<Expresion> **)** <Sentencia> <F3> | **while (** <Expresion> **)** <Sentencia> | <Bloque> |

return <ExpresionOpcional>**;**

<F3> ::= **else**<Sentencia> | ε

<Asignacion> ::= <AccesoVar>**=**<Expresion> | <AccesoThis>**=** <Expresion>

<SentenciaLlamada> ::= **(** <Primario> **)**

<ExpresionOpcional> ::= <ExpOr> | ε

<Expresion> ::= <ExpOr>

<ExpOr> ::=<ExpAnd><ExpOr2>

<ExpOr2> ::= **||** <ExpAnd><ExpOr2> | ε

<ExpAnd> ::=<ExpIg><ExpAnd2>

<ExpAnd2> ::=**&&**<ExpIg><ExpAnd2> | ε

<ExpIg> ::=<ExpComp><ExpIg2>

<ExpIg2> ::=<OpIg><ExpComp><ExpIg2> | ε

<ExpComp> ::=<ExpAd ><F8>

<F8> ::= <OpComp><ExpAd> | ε

<ExpAd> ::=<ExpMul><ExpAd2>

<ExpAd2> ::=<OpAd><ExpMul><ExpAd2> | ε

<ExpMul> ::=<ExpUn><ExpMul2>

<ExpMul2> ::=<OpMul><ExpUn><ExpMul2> | ε

<ExpUn> ::= <OpUn><ExpUn> | <Operando>

<OpIg> ::= **==** | **!=**

<OpComp> ::= **<** | **>** | **<=** | **>=**

<OpAd> ::= **+** | **-**

<OpUn> ::= **+** | **-** | **!**

<OpMul> ::= ***** | **/**

<Operando> ::= <Literal> | <Primario>

<Literal> ::=null | true | false | entero| caracter| string

<Primario> ::= <ExpresionParentizada> | <AccesoThis> | **idMetVar** <F4> | <LlamadaMetodoEstatico>

| **new** <F9>

<F4> ::= <ArgsActuales><Encadenado> | <Encadenado>

<LlamadaMetodo> ::=**idMetVar**<ArgsActuales><Encadenado>

<AccesoThis> ::=**this**<Encadenado>

<AccesoVar> ::= **idMetVar**<Encadenado>

<ExpresionParentizada> ::= **(** <Expresion> **)** <Encadenado>

<LlamadaMetodoEstatico> ::= **idClase .** <LlamadaMetodo>

<F9> ::= **idClase**<ArgsActuales><Encadenado> | <TipoPrimitivo>**[** <Expresion>**]** <Encadenado>

<ArgsActuales> ::= (<ArgsActualesAux>)

<ArgsActualesAux>::= <ListaExp> | ε

<ListaExps> ::= <Expresion> <F5>

<F5> ::= **,**<ListaExps> | ε

<Encadenado> ::= **.**<F6> | <AccesoArregloEncadenado> | ε

<F6> ::= **idMetVar** <F7>

<F7> ::= <ArgsActuales><Encadenado> | <Encadenado>

<AccesoArregloEncadenado> ::= **[**<Expresion>**]**<Encadenado>

Consideraciones de diseño

El método empezar() en la clase de Analizador Sintáctico es el que empieza a analizar los Tokens que se leen.

El analizador Sintáctico se realizo con el método recursivo. En toda producción que podía ser vacio, se calcularon los segundos para poder dar un error más preciso, por lo cual no se opto

por la posibilidad de usar un ‘else’ que deje “pasar un error” para que sea detectado luego en otra parte del analizador.

Los errores tratan de ser los más precisos posibles según el contexto que venían analizando.

En el analizador sintáctico, la función match lanza excepción cuando intenta comparar un tipo de Token con un tipo de Token actual diferente. Al no implementar modo pánico, se decidió propagar toda excepción para que sea capturada por la clase principal.

Análisis Semántico

Chequeo de declaraciones

Es el encargado de chequear las declaraciones del programa, por ejemplo, declaración de métodos, declaración de clases, declaración de atributos, verificar herencia circular, etc. Una forma de facilitar este proceso es la creación de una entidad llamada Tabla de Símbolos.

Para construir una Tabla de Símbolos (TS), utilizamos la gramática de atributos. La TS contiene toda entidad declarara en el código fuente MiniJava. La EDT es una forma de comprender como se va creando en la Tabla de Símbolos.

EDT

<Empezar> ::= { creo la tabla de símbolos } <Inicial> **eof**

<Inicial> ::= <Clase><MasClase>

<MasClase>::= <Clase><MasClase> | ε

<Clase> ::= **class idClase**{ creo una EntradaClase con el token IdClase y lo agrego a TS} <Herencia> **{** <Miembro>**}** { Agrego un constructor con 0 parámetros si no tiene constructor EntradaClase}

<Miembro> ::= <Atributo><Miembro> |

<Ctor> { agrego <Ctor>.clase a la EntradaClase actual } <Miembro> |

<Metodo><Miembro> | ε

<Herencia> ::= ε { herencia de clase actual es 'Object'} | **extends idClase** { herencia de clase actual es IdClase.lex }

<Atributo> ::= <Visibilidad>{ <visibilidad>.tipo = <visibilidad>.tipo } <Tipo> >{ <tipo>.tipo = <tipo>.tipo } <ListaDecVars> { guardarAtributos(<listaDecVars>.atributos) en ClaseActual } <InLine> **;**

<InLine> ::= = <Expresion> | ε

<Metodo> ::= <FormaMetodo><TipoMetodo>idMetVar {creo EntradaConParams con IdMetVar, <FormaMetodo>.tipo , <TipoMetodo>.dinamico } <ArgsFormales> { agrego EntradaMetodo a la clase Actual } <Bloque>

<Ctor> ::=**idClase** { <ctor>.clase = new EntradaClase(IdClase) } <ArgsFormales> { <ctor>.clase.guardar(<ArgsFormales>.paramatros) } <Bloque>

<ArgsFormales> ::= **(** <ListaArgsFormalesAux> { <ArgsFormales>.parametros = <listaArgsFormalesAux>.parametros } **)**

<ListaArgsFormalesAux> ::= <ListaArgsFormales> { <ListaArgsFormalesAux>.parametros = <listaArgsFormales>.parametros } | ε

<ListaArgsFormales> ::= <ArgFormal> { agrego <ArgFormal>.param a EntradaConParams actual } <F1>

<F1> ::= **,**<ListaArgsFormales> | ε

<ArgFormal> ::= <Tipo>**idMetVar** { <ArgFormal>.param = nuevo EntradaParametro con <Tipo>.tipo y token IdMetVar }

<FormaMetodo> ::= **static** { <formaMetodo>.dinamico = false} | **dynamic** {<formaMetodo>.dinamico = true }

<Visibilidad> ::= **public** {<visibilidad>.tipo = public} |**prívate** {<visibilidad>.tipo = private}

<ListaDecVars> ::= **idMetVar** { creo una EntradaAtributo con el token IdMetVar, <visibilidad>.tipo y <tipo>.tipo y lo agrego a ClaseActual } <F2>

<F2> ::= **,**<ListaDecVars> | ε

<TipoMetodo> ::= <Tipo> {tipo = <Tipo>.tipo} | **void** {tipo = void}

<Tipo> ::= **Boolean** {<Tipo>.tipo = boolean } <F10> {<Tipo>.tipo = <f10>.tipo } | **char** {<Tipo>.tipo = char } <F10> > {<Tipo>.tipo = <f10>.tipo } | **int** {<Tipo>.tipo = int } <F10> > {<Tipo>.tipo = <f10>.tipo } | **idClase** {<Tipo>.tipo = IdClase } | **String** {<Tipo>.tipo = String }

<TipoPrimitivo> ::= **boolean** | **char** | **int**

<F10> ::= **[]** {<f10>.tipo = TipoArreglo } | ε {<f10>.tipo = <Tipo>.tipo }

<Bloque> ::= **{**<MasSentencia>**}**

<MasSentencia> ::= <Sentencia><MasSentencia> | ε

<Sentencia> ::= **;** | <Asignacion>**;** | <SentenciaLlamada>**;** | <Tipo><ListaDecVars ><InLine>**;** |

if(<Expresion> **)** <Sentencia> <F3> | **while (** <Expresion> **)** <Sentencia> | <Bloque> |

return <ExpresionOpcional>**;**

<F3> ::= **else**<Sentencia> | ε

<Asignacion> ::= <AccesoVar>**=**<Expresion> | <AccesoThis>**=** <Expresion>

<SentenciaLlamada> ::= **(** <Primario> **)**

<ExpresionOpcional> ::= <ExpOr> | ε

<Expresion> ::= <ExpOr>

<ExpOr> ::=<ExpAnd><ExpOr2>

<ExpOr2> ::= **||** <ExpAnd><ExpOr2> | ε

<ExpAnd> ::=<ExpIg><ExpAnd2>

<ExpAnd2> ::=**&&**<ExpIg><ExpAnd2> | ε

<ExpIg> ::=<ExpComp><ExpIg2>

<ExpIg2> ::=<OpIg><ExpComp><ExpIg2> | ε

<ExpComp> ::=<ExpAd ><F8>

<F8> ::= <OpComp><ExpAd> | ε

<ExpAd> ::=<ExpMul><ExpAd2>

<ExpAd2> ::=<OpAd><ExpMul><ExpAd2> | ε

<ExpMul> ::=<ExpUn><ExpMul2>

<ExpMul2> ::=<OpMul><ExpUn><ExpMul2> | ε

<ExpUn> ::= <OpUn><ExpUn> | <Operando>

<OpIg> ::= **==** | **!=**

<OpComp> ::= **<** | **>** | **<=** | **>=**

<OpAd> ::= **+** | **-**

<OpUn> ::= **+** | **-** | **!**

<OpMul> ::= ***** | **/**

<Operando> ::= <Literal> | <Primario>

<Literal> ::=null | true | false | entero| caracter| string

<Primario> ::= <ExpresionParentizada> | <AccesoThis> | **idMetVar** <F4> | <LlamadaMetodoEstatico>

| **new** <F9>

<F4> ::= <ArgsActuales><Encadenado> | <Encadenado>

<LlamadaMetodo> ::=**idMetVar**<ArgsActuales><Encadenado>

<AccesoThis> ::=**this**<Encadenado>

<AccesoVar> ::= **idMetVar**<Encadenado>

<ExpresionParentizada> ::= **(** <Expresion> **)** <Encadenado>

<LlamadaMetodoEstatico> ::= **idClase .** <LlamadaMetodo>

<F9> ::= **idClase**<ArgsActuales><Encadenado> | <TipoPrimitivo>**[** <Expresion>**]** <Encadenado>

<ArgsActuales> ::= (<ArgsActualesAux>)

<ArgsActualesAux>::= <ListaExp> | ε

<ListaExps> ::= <Expresion> <F5>

<F5> ::= **,**<ListaExps> | ε

<Encadenado> ::= **.**<F6> | <AccesoArregloEncadenado> | ε

<F6> ::= **idMetVar** <F7>

<F7> ::= <ArgsActuales><Encadenado> | <Encadenado>

<AccesoArregloEncadenado> ::= **[**<Expresion>**]**<Encadenado>

Consideraciones de diseño

La EDT no utiliza el método formal.

Solo se permite un static void main() por archivo de minijava. Si el programador decide poner un dynamic void main() en un clase, le quita la posibilidad de agregar un static void main() y no arrojara error.

Si al momento de chequear que no haya herencia circular se encuentra con un caso de este tipo, la tabla de símbolos (TS) consolida la clase que hereda de la clase que genera conflicto. Por ejemplo si tenemos Clase1 extends Clase2, Clase2 extends Clase3, Clase3 extends Clase1, el compilador detectarte la herencia circular en Clase3, por lo cual lo marca como consolidado para que pueda seguir buscando otros errores. De igual manera ocurre que si por ejemplo, Clase1 extends Clase, y Clase no existe, el compilador muestra el error de que Clase no existe y a su vez consolida Clase1 para poder seguir consolidando.

Los constructores se guardan en tablas dentro de cada EntradaClase. Su clave es representada por la cantidad de parámetros que poseen.

Los métodos se guardan en tablas y listas dentro de cada EntradaClase. La clave en las tablas es generada a través de la concatenación del nombre del método, seguido de $ y la cantidad de parámetros. Por ejemplo ‘static void m1(int a)’ se almacena como ‘m1$1’.

En la clase TablaDeSimbolos se encuentra el método chequear el cual realiza:

- Verifica si existe un método main en alguna de las clases.
- Recorre todas las clases de la TS y verifica para cada una sí:
- Hay herencia circular y existe las clases a la que se extiende.
- Hay un tipo IdClase que no está definido.

En cada EntradaClase hay método llamado consolidar(), el cual consolida la clase. Su operatoria es la siguiente

---

Si ‘no está consolidada la clase’:

    Si ‘el padre no está consolidado’

        consolidar la clase padre.

    consolidarAtributos()
    consolidarMetodo()

Poner como consolidada la clase

ConsolidarAtributo()

Recorrer los atributos del padre

Si no hay un atributo en el hijo con el mismo nombre

Agrego atributo de padre al hijo

ConsolidarMetodo()

Recorer los métodos del padre

Si no hay un método compatible en el hijo

Agrego el método del padre al hijo

Sino

Si los tipos de retorno son iguales, los parámetros son del mismo tipo en el mismo orden

Agrego el método del padre al hijo si es compatible

Chequeo de Sentencias

Esta etapa usa los elementos que se guardaron en la Tabla de Símbolos. Es necesario tener una estructura adicional para mantener una referencia a la estructura de dichas sentencias. Para ello usamos el árbol sintáctico abstracto (AST). El AST nos permite almacenar una estructura del programa, eliminando elementos que son importantes pero son reemplazados por la jerarquía del AST, por ejemplo, comas, puntos, paréntesis, corchetes, etc.

Consideraciones de diseño

La EDT no utiliza el método formal.

La recuperación de errores de declaración no permite que se continúe con el chequeo de sentencias. Por ejemplo, si se encuentra un error semántico de declaración y además había un error semántico de sentencia, solo encontrará los errores de declaración.

Cada chequear() de cada nodo del AST, especifica en forma de comentarios que tipos de chequeos se realizaron.

EDT

<Empezar> ::= { creo la tabla de símbolos } <Inicial> **eof**

<Inicial> ::= <Clase><MasClase>

<MasClase>::= <Clase><MasClase> | ε

<Clase> ::= **class idClase**{ creo una EntradaClase con el token IdClase y lo agrego a TS} <Herencia> **{** <Miembro>**}** { Agrego un constructor con 0 parámetros si no tiene constructor EntradaClase}

<Miembro> ::= <Atributo><Miembro> |

<Ctor> { agrego <Ctor>.clase a la EntradaClase actual } <Miembro> |

<Metodo><Miembro> | ε

<Herencia> ::= ε { herencia de clase actual es 'Object'} | **extends idClase** { herencia de clase actual es IdClase.lex }

<Atributo> ::= <Visibilidad>{ <visibilidad>.tipo = <visibilidad>.tipo } <Tipo> >{ <tipo>.tipo = <tipo>.tipo } <ListaDecVars> { guardarAtributos(<listaDecVars>.atributos) en ClaseActual } <InLine> **;**

<InLine> ::= = <Expresion> {retorno el nodo expresion} | ε {retorno un nuevo nodo expresión vacio}

<Metodo> ::= <FormaMetodo><TipoMetodo>idMetVar {creo EntradaConParams con IdMetVar, <FormaMetodo>.tipo , <TipoMetodo>.dinamico } <ArgsFormales> { agrego EntradaMetodo a la clase Actual } <Bloque> {guardo el bloque que recibo en el bloque del metodo}

<Ctor> ::=**idClase** { <ctor>.clase = new EntradaClase(IdClase) } <ArgsFormales> { <ctor>.clase.guardar(<ArgsFormales>.paramatros) } <Bloque> {guardo el bloque que recibo en el bloque del metodo}

<ArgsFormales> ::= **(** <ListaArgsFormalesAux> { <ArgsFormales>.parametros = <listaArgsFormalesAux>.parametros } **)**

<ListaArgsFormalesAux> ::= <ListaArgsFormales> { <ListaArgsFormalesAux>.parametros = <listaArgsFormales>.parametros } | ε

<ListaArgsFormales> ::= <ArgFormal> { agrego <ArgFormal>.param a EntradaConParams actual } <F1>

<F1> ::= **,**<ListaArgsFormales> | ε

<ArgFormal> ::= <Tipo>**idMetVar** { <ArgFormal>.param = nuevo EntradaParametro con <Tipo>.tipo y token IdMetVar }

<FormaMetodo> ::= **static** { <formaMetodo>.dinamico = false} | **dynamic** {<formaMetodo>.dinamico = true }

<Visibilidad> ::= **public** {<visibilidad>.tipo = public} |**prívate** {<visibilidad>.tipo = private}

<ListaDecVars> ::= **idMetVar** { creo una EntradaAtributo con el token IdMetVar, <visibilidad>.tipo y <tipo>.tipo y lo agrego a ClaseActual } <F2>

<F2> ::= **,**<ListaDecVars> | ε

<TipoMetodo> ::= <Tipo> {tipo = <Tipo>.tipo} | **void** {tipo = void}

<Tipo> ::= **Boolean** {<Tipo>.tipo = boolean } <F10> {<Tipo>.tipo = <f10>.tipo } | **char** {<Tipo>.tipo = char } <F10> > {<Tipo>.tipo = <f10>.tipo } | **int** {<Tipo>.tipo = int } <F10> > {<Tipo>.tipo = <f10>.tipo } | **idClase** {<Tipo>.tipo = IdClase } | **String** {<Tipo>.tipo = String }

<TipoPrimitivo> ::= **boolean** | **char** | **int**

<F10> ::= **[]** {<f10>.tipo = TipoArreglo } | ε {<f10>.tipo = <Tipo>.tipo }

<Bloque> ::= **{** {recibo el bloque padre (null si es método el padre) **}** <MasSentencia>**}** {retorno el bloque que cree}

<MasSentencia> ::= <Sentencia> {recibo una lista de sentencias y la agrego a las sentencias ya encontradas} <MasSentencia> { le pido a masSentencias si hay mas sentecias} | ε {retorno la lista de sentencias}

<Sentencia> ::= **;** {retorno una sentencia vacia} |

<Asignacion>**;** {retorno una asignación}|

<SentenciaLlamada>**;** {retorno una sentencia llamada} |

<Tipo><ListaDecVars ><InLine>**;** {retorno una lista de declaración de variables y de asignaciones} |

if(<Expresion>**)** <Sentencia> {creo un nodo IfSinElse con expresión y sentencia } <F3> {retorno el NIf que retorna f3} |

while (<Expresion> **)** <Sentencia> { retorna un nodo while con espresion y sentencia } |

<Bloque> { retorno un bloque nuevo } |

return <ExpresionOpcional>**;** {retorno un nodo return}

<F3> ::= **else**<Sentencia> {retornar un NIfConElse } | ε {retorna NIfSinElse que recibió por parametro}

<Asignacion> ::= <AccesoVar>**=**<Expresion> {retorno nodo asignación con accesovar, igual y expresion} | <AccesoThis>**=** <Expresion> {retorno nodo asignación con accesothis, igual y expresion}

<SentenciaLlamada> ::= **(** <Primario> **)** {retorno nodo sentencia llamada con primario }

<ExpresionOpcional> ::= <ExpOr> {retorno el nodo expresión que retorna expOr } | ε {retorno ExpresionVacia}

<Expresion> ::= <ExpOr> {retorno el nodo expresión que retorna expOr }

<ExpOr> ::=<ExpAnd><ExpOr2> {retorno el nodo expresión que resulta de calcular ExpOr2 con el Nexpresion de ExpAnd}

<ExpOr2> ::= **||** <ExpAnd><ExpOr2> {retorno el nodo expresión que resulta de calcular ExpOr2 con el Nexpresion creada con la expresión pasada por parámetros y ExpAnd} | ε { retorna el nodo que recibe por parametro}

<ExpAnd> ::=<ExpIg><ExpAnd2> {retorno el nodo expresión que resulta de calcular ExpOr2 con el Nexpresion de ExpIg}

<ExpAnd2> ::=**&&**<ExpIg><ExpAnd2> {retorno el nodo expresión que resulta de calcular ExpAnd2 con el NExpresion creado con la expresión pasada por parámetros y ExpIg } | ε { retorna el nodo que recibe por parametro}

<ExpIg> ::=<ExpComp><ExpIg2> {retorno el nodo expresión que resulta de calcular ExpIg2 con el Nexpresion de ExpComp}

<ExpIg2> ::=<OpIg><ExpComp><ExpIg2> { retorno el nodo expresión que resulta de calcular ExpIg2 con el NExpresion creado con la expresión pasada por parámetros y ExpComp } | ε { retorna el nodo que recibe por parametro}

<ExpComp> ::=<ExpAd ><F8> {retorno el nodo expresión que resulta de calcular f8 con el Nexpresion de ExpAd}

<F8> ::= <OpComp><ExpAd> { retorno el nodo expresión creado con la expresión pasada por parámetros y ExpAd } | ε { retorna el nodo que recibe por parametro}

<ExpAd> ::=<ExpMul><ExpAd2> {retorno el nodo expresión que resulta de calcular ExpAd2 con el Nexpresion de ExpMul}

<ExpAd2> ::=<OpAd><ExpMul><ExpAd2> { retorno el nodo expresión que resulta de calcular ExpAd2 con el NExpresion creado con la expresión pasada por parámetros y ExpMul } | ε { retorna el nodo que recibe por parámetro}

<ExpMul> ::=<ExpUn><ExpMul2> {retorno el nodo expresión que resulta de calcular ExpMul2 con el Nexpresion de ExpUn}

<ExpMul2> ::=<OpMul><ExpUn><ExpMul2> { retorno el nodo expresión que resulta de calcular ExpMul2 con el NExpresion creado con la expresión pasada por parámetros y ExpUn } | ε { retorna el nodo que recibe por parámetro}

<ExpUn> ::= <OpUn><ExpUn> { retorno el nodo expresión unario creado con el resultado de ExpUn } | <Operando> { retorna el nodo expresión de operando }

<OpIg> ::= **==** | **!=**

<OpComp> ::= **<** | **>** | **<=** | **>=**

<OpAd> ::= **+** | **-**

<OpUn> ::= **+** | **-** | **!**

<OpMul> ::= ***** | **/**

<Operando> ::= <Literal> { retorno un nodo literal que contiene el tipo como atr. } | <Primario> {retorno un nodo primario}

<Literal> ::=**null** | **true** | f**alse** | **entero**| c**aracter**| **string** {para todas las opciones de <literal>: retorna un nodo literal con el tipo al que representa }

<Primario> ::= <ExpresionParentizada> { retorna una expresión parentizada } | <AccesoThis> { retorna un acceso this } | **idMetVar** <F4> { retorna el resultado de <F4>} | <LlamadaMetodoEstatico> { retorna una llamada método estatico>}

| **new** <F9> {retorna el resultado de f9 }

<F4> ::= <ArgsActuales><Encadenado> {retorna una llamada método} | <Encadenado> { retorna un acceso var }

<LlamadaMetodo> ::=**idMetVar**<ArgsActuales><Encadenado> {retorna una llamada método}

<AccesoThis> ::=**this**<Encadenado> { retorna el nuevo nodo acceso this con encadenado y boolean li lugar (si esta a la izquierda del '=') }

<AccesoVar> ::= **idMetVar**<Encadenado> { retorna el nuevo nodo acceso var con encadenado y boolean li lugar (si esta a la izquierda del '=') }

<ExpresionParentizada> ::= **(** <Expresion> **)** <Encadenado> {retorna una expresión parentizada}

<LlamadaMetodoEstatico> ::= **idClase .** <LlamadaMetodo> {retorna una llamada método estático}

<F9> ::= **idClase**<ArgsActuales><Encadenado> { retorna una llamada Constructor de clase } | <TipoPrimitivo>**[** <Expresion>**]** <Encadenado> { retorna una llamada Constructor de arreglo }

<ArgsActuales> ::= (<ArgsActualesAux>) { retorna una clase con una lista de expresiones}

<ArgsActualesAux>::= <ListaExp> {devuelve lista de expresiones después de encontrar la ultima } | ε {devuelve una lista de expresiones }

<ListaExps> ::= <Expresion> <F5> {Encuentra una expresión, la agrega a la lista y le pide a f5 si hay alguna más}

<F5> ::= **,**<ListaExps> {Encuentra una coma por lo cual hay otro argumento, retorna la lista final de argumento que retorna listaExps} | ε { Retorna una lista de argumentos }

<Encadenado> ::= **.**<F6> {Retorna el resultado de f6}| <AccesoArregloEncadenado> {Retorna un acceso arreglo encadenado} | ε {retorna un encadenado vacio}

<F6> ::= **idMetVar** <F4> {Retorna el resultado de f7}

<F7> ::= <ArgsActuales><Encadenado> { retorna una llamada a un método encadenado } | <Encadenado> { retorna una llamada a una variable encadenada }

<AccesoArregloEncadenado> ::= **[**<Expresion>**]**<Encadenado> {Retorna un acceso arreglo encadenado}

Generación de código

Dada la estructura que se generó en etapas anteriores, se procede a crear el archivo de salida, el cual contiene una representación intermedia, que será ejecutada por la Maquina Virtual de MiniJava CeIVM.

Para lograr esto, fue necesario agregar más implementación al AST para crear y completar el archivo de salida. Se utilizó la estrategia de delegarle a una nueva pasada esta tarea, y no hacerlo mientras se realizaba la etapa anterior.

Además se calcularon los offsets correspondientes a los atributos, método y parámetros utilizados en las clases.

Calculo de offsets

Métodos

Al final de consolidar los métodos de la clase, se recorren todos los métodos dinámicos y se le asigna un offset, dependiendo de su contexto, ya sea que es heredado, redefinido o nuevo. El primer método tiene offset 1, el segundo 2, y el ultimo método tiene offset n.

Atributos de instancia

Al final de consolidar los métodos de la clase, se recorren todos los atributos y se le asigna un offset, dependiendo de su contexto, ya sea que es heredado o nuevo. El primer atributo tiene offset 0, el segundo 1, y el ultimo atributo tiene offset n-1.

Parámetros en métodos/constructores

Al final de consolidar los métodos de la clase, se recorren todos los parámetros y se le asigna un offset. El offset del parámetro se calcula como: cons + cantidad de parámetros del método – posición de parámetro, donde cons es 3 si el método es dinámico y 2 si es estático.

Variables locales

En la generación código se le asigna un offset. La primer variable local tiene offset 0, la segundo -1, y a si sucesivamente.

Consideraciones de diseño

El compilador hace una pasada para chequear el código a través de los AST creados en el analizador sintáctico. Luego de esto se procede a realizar una nueva pasada donde se genera el código maquina.

Las etiquetas de los métodos se generan a través de método getEtiqueta() que se encuentra en EntradaConParams, el cual se compone de “ mt_ <nombre_metodo> $ <Cantidad_parametros> _ <Clase_del_metodo> ”

Las etiquetas de los constructores se generan a través de método getEtiqueta() que se encuentra en EntradaConstructor, el cual se compone de “ ctor_ <Clase_del_ctor> $ <Cantidad_parametros> ”