Guía práctica de ANTLR 2.7.2

Versión 1.0 (Septiembre 2003)

Alumno

Enrique José García Cota

(+34) 666901806

enriqueinformatico@yahoo.es

Tutor de proyecto

José Antonio Troyano Jiménez

(+34) 954552777

troyano@lsi.us.es

E.T.S. de Ingeniería Informática de la Universidad de Sevilla.

Departamento de Lenguajes y Sistemas Informáticos.

Agradecimientos

Agradezco a mi tutor, Jose Antonio Troyano, que me haya dado total libertad para elegir el formato del proyecto.

Así mismo quiero agradecer al equipo de desarrollo de ANTLR, y especialmente a Terence Parr, el tiempo y esfuerzo que han dedicado a desarrollar esta herramienta y mantener satisfecha a una extensa comunidad de usuarios, comunidad a la que también estoy agradecido.

Estoy muy agradecido a los desarrolladores de la *suite* de ofimática OpenOffice.org, sin cuyo procesador de textos (*Writer*) me habría muy penoso redactar este documento. Este agradecimiento se extiende a todas las asociaciones y desarrolladores que contribuyen con el movimiento del software libre, el código abierto y los estándares accesibles por todos, y en especial a Richard Stallman y la FSF.

Por último deseo mostrar mi gratitud a mi familia y amigos, que me han apoyado en todo momento, y a Petra, que ha tenido muchísima paciencia.

Índice de contenido

Capítulo 1: Preámbulos....1

Sección 1.1: Introducción....2
1.1.1: Objetivos – LeLi y antlraux.... 2
1.1.2: Requisitos.... 2
1.1.3: Enfoque.... 2
1.1.4: Estructura.... 3
1.1.5: Notación.... 4
Sección 1.2: Breve repaso a la teoría de compiladores....6
1.2.1: Conceptos básicos: software y hardware.... 6
1.2.2: Un poco de historia.... 7
1.2.3: Ensambladores y compiladores.... 8
1.2.4: Intérpretes y máquinas virtuales.... 9
1.2.5: El proceso de compilación.... 9
1.2.6: “Compiladores de compiladores”.... 12
Sección 1.3: Algoritmos de análisis.... 15
1.3.1: La sintaxis EBNF.... 15
1.3.2: “De arriba a abajo”; Analizadores recursivos descendentes.... 19
1.3.3: “De abajo a arriba”; analizadores LR.... 23
1.3.4: Comparación entre LR(k) y LL(k).... 25
Sección 1.4: Conclusión....28

Capítulo 2: Presentación de ANTLR....29

Sección 2.1: Introducción....31
2.1.1: ¿Qué es y cómo funciona ANTLR?.... 31
2.1.2: Propósito y estructura de éste capítulo.... 31
Sección 2.2: Los analizadores en ANTLR....33
2.2.1: Especificación de gramáticas con ANTLR.... 33
2.2.2: La zona de código nativo.... 34
Sección 2.3: Los flujos de información....35
2.3.1: Flujo de caracteres.... 35
2.3.2: Flujo de Tokens.... 35
2.3.3: Los ASTs.... 38
Sección 2.4: Reglas EBNF extendidas....44
2.4.1: Introducción.... 44
2.4.2: Declarando variables locales en una regla.... 44
2.4.3: Utilizando las etiquetas.... 46
2.4.4: Pasando parámetros a una regla.... 46
2.4.5: Devolviendo valores en una regla.... 47
2.4.6: Utilizando rangos de caracteres en el analizador léxico.... 48
2.4.7: Utilizando patrones árbol en el analizador semántico.... 48
Sección 2.5: Construcción de los ASTs....50
2.5.1: Comportamiento por defecto.... 50
2.5.2: El sufijo de enraizamiento (^).... 51
2.5.3: Sufijo de filtrado (!).... 52
2.5.4: Desactivando la construcción por defecto.... 53
2.5.5: Construyendo el AST en una acción.... 53
2.5.6: Casos en los que la construcción por defecto no basta.... 55
Sección 2.6: Analizadores de la Microcalculadora.... 56
2.6.1: El fichero MicroCalc.g y el paquete microcalc.... 56
2.6.2: El analizador léxico.... 56
2.6.3: El analizador sintáctico.... 57

2.6.4: Nivel semántico ... 58
2.6.5: El fichero MicroCalc.g ... 59
2.6.6: Generando los analizadores de MicroCalc ... 61
Sección 2.7: Ejecutando Microcalc ... 62
2.7.1: La clase Calc ... 62
2.7.2: Clase Calc refinada ... 63
2.7.3: Utilizando microcalc ... 65
Sección 2.8: Conclusión ... 67

Capítulo 3: LeLi: un Lenguaje Limitado ... 68

Sección 3.1: Introducción ... 69
Sección 3.2: Nivel léxico ... 70
3.2.1: Blancos ... 70
3.2.2: Comentarios ... 70
3.2.3: Literales ... 70
3.2.4: Identificadores ... 70
3.2.5: Palabras reservadas ... 71
Sección 3.3: Niveles sintáctico y semántico ... 72
3.3.1: Características básicas de LeLi ... 72
3.3.2: Clases especiales del lenguaje ... 72
3.3.3: Declaración de una clase ... 73
3.3.4: Métodos de una clase ... 73
3.3.5: Constructores ... 74
3.3.6: Métodos abstractos ... 76
3.3.7: Variables, atributos y parámetros ... 76
3.3.8: Expresiones ... 79
Sección 3.4: Instrucciones de LeLi ... 86
3.4.1: Separación de instrucciones ... 86
3.4.2: Asignaciones ... 86
3.4.3: Bucles : mientras, hacer-mientras, desde ... 86
3.4.4: Condicionales: si ... 87
3.4.5: Instrucción volver ... 87
Sección 3.5: Otros aspectos de LeLi ... 88
3.5.1: Herencia de clases ... 88
3.5.2: Gestión de la memoria ... 89
Sección 3.6: Conclusión ... 90

Capítulo 4: Análisis léxico de LeLi ... 91

Sección 4.1: Introducción ... 92
Sección 4.2: Estructura general del analizador ... 93
4.2.1: Cuerpo del fichero ... 93
4.2.2: Header: El paquete leli ... 93
4.2.3: Opciones del analizador ... 93
4.2.4: Zona de tokens ... 94
Sección 4.3: Zona de reglas ... 96
4.3.1: ¡Argh! ... 96
4.3.2: Blancos ... 97
4.3.3: Identificadores ... 99
4.3.4: Símbolos y operadores ... 101
4.3.5: Enteros y reales ... 102
4.3.6: Comentarios ... 102
4.3.7: Literales cadena ... 106
Sección 4.4: Compilando el analizador ... 107

Sección 4.5: Ejecución: presentación de la clase Tool ... 108
4.5.1: Introducción ... 108
4.5.2: Clase Tool imprimiendo el flujo “tokens” ... 108
4.5.3: Ejemplo ... 109
Sección 4.6: Añadiendo el nombre de fichero a los tokens ... 110
4.6.1: La incongruencia de antlr.CommonToken ... 110
4.6.2: Tokens homogéneos y heterogéneos ... 110
4.6.3: Modificaciones en la clase Tool ... 112
4.6.4: Segundo problema ... 113
Sección 4.7: El fichero LeLiLexer.g ... 116
Sección 4.8: Conclusión ... 120

Capítulo 5:
Análisis sintáctico de LeLi ... 121

Sección 5.1: Introducción ... 122
Sección 5.2: Definición del analizador sintáctico ... 123
5.2.1: Opciones del analizador ... 123
5.2.2: Importando los tokens ... 123
5.2.3: Zona de tokens ... 124
Sección 5.3: Zona de reglas ... 126
5.3.1: Programa ... 126
5.3.2: Definición de clases ... 126
5.3.3: Azúcar sintáctica ... 127
5.3.4: Definición de atributos ... 131
5.3.5: Definición de métodos ... 131
5.3.6: Expresiones ... 133
5.3.7: Instrucciones ... 137
Sección 5.4: Fichero LeLiParser.g ... 142
Sección 5.5: Compilación del analizador ... 149
Sección 5.6: Ejecución: modificaciones en la clase Tool ... 150
5.6.1: Introducción ... 150
5.6.2: Interpretando la línea de comandos: el paquete antlraux.clparse ... 150
5.6.3: La línea de comandos inicial: el método leli.Tool.LeeLC() ... 153
5.6.4: El método leli.Tool.imprimeAyuda() ... 154
5.6.5: El nuevo método main ... 154
5.6.6: El método leli.Tool.trabaja() ... 155
Sección 5.7: Otros aspectos de los ASTs ... 161
5.7.1: Fábricas de ASTs ... 161
5.7.2: ASTs heterogéneos ... 161
Sección 5.8: Conclusión ... 164

Capítulo 6:
Recuperación de errores ... 165

Sección 6.1: Introducción ... 167
6.1.1: Situación ... 167
6.1.2: La controversia ... 167
6.1.3: Fases ... 168
6.1.4: Gestión de errores en bison y flex ... 168
6.1.5: Errores en analizadores recursivos descendentes ... 169
6.1.6: Errores como excepciones ... 169
6.1.7: Manejadores de excepciones ... 170
Sección 6.2: Estrategias de recuperación ... 172
6.2.1: Introducción ... 172
6.2.2: Estrategia basada en SIGUIENTE ... 173
6.2.3: Conjuntos PRIMERO y SIGUIENTE ... 174

6.2.4: Estrategia basada en PRIMERO + SIGUIENTE ... 174
6.2.5: Aprovechando el modo pánico ... 175
6.2.6: Tokens de sincronismo ... 176
6.2.7: Implementación en ANTLR ... 178
6.2.8: Trampas para excepciones ... 181
6.2.9: Retardo en el tratamiento de errores ... 184
Sección 6.3: Implementación - Herencia de gramáticas ... 189
6.3.1: El problema ... 189
6.3.2: Presentando la herencia de gramáticas en ANTLR ... 189
6.3.3: Herencia ANTLR != Herencia java ... 190
6.3.4: Línea de comandos ... 193
6.3.5: ¿Cómo se relaciona todo esto con la RE? ... 193
6.3.6: Importación de vocabulario ... 193
Sección 6.4: Control de mensajes: La clase Logger ... 195
6.4.1: El problema ... 195
6.4.2: La clase Logger del paquete antlraux ... 196
Sección 6.5: Mejorando los mensajes de error ... 199
6.5.1: Introducción ... 199
6.5.2: Cambiando el idioma de los mensajes de error ... 199
6.5.3: Alias de los tokens ... 201
Sección 6.6: Aplicación en el compilador de LeLi ... 205
6.6.1: Acotación del problema: el fichero de errores ... 205
6.6.2: Aplicando los símbolos de sincronismo ... 206
6.6.3: Colocando las trampas para excepciones ... 209
6.6.4: Retardando el tratamiento de errores ... 212
6.6.5: Errores frecuentes – acentuación ... 214
6.6.6: El resultado ... 219
Sección 6.7: Código ... 221
6.7.1: Fichero LeLiLexer.g ... 221
6.7.2: Fichero LeLiParser.g ... 225
6.7.3: Fichero LeLiErrorRecoveryParser.g ... 232
Sección 6.8: Compilando y ejecutando el analizador ... 240
6.8.1: Compilación ... 240
6.8.2: Ejecución ... 240
Sección 6.9: Conclusión ... 243

Capítulo 7: Análisis semántico de LeLi ... 244

Sección 7.1: Introducción ... 247
7.1.1: Errores semánticos estáticos ... 247
7.1.2: Errores semánticos dinámicos ... 250
7.1.3: Añadiendo información léxica a los ASTs ... 250
Sección 7.2: Iterador simple de árboles ... 253
7.2.1: Estrategia de implementación del análisis semántico ... 253
7.2.2: Estructura del fichero de gramática ... 254
7.2.3: Regla raíz: Programa ... 254
7.2.4: Definición de clases ... 254
7.2.5: Definición de atributos ... 255
7.2.6: Expresiones ... 256
7.2.7: Instrucciones ... 257
7.2.8: Código completo del iterador ... 258
7.2.9: Los errores de construcción del AST ... 262
Sección 7.3: El sistema Ámbito/Declaración/Tipo ... 263
7.3.1: Introducción ... 263
7.3.2: Presentación de el sistema Ámbito/Declaración/Tipo (ADT) ... 263
7.3.3: Implementación del sistema ADT: el paquete antlraux.context ... 266

7.3.4: La clase antlraux.context.Scope ... 268
7.3.5: La clase antlraux.context.Declaration ... 272
7.3.6: El sistema de tipos (antlraux.context.types.*) ... 273
7.3.7: ASTs especializados ... 279
7.3.8: Resumen ... 283
Sección 7.4: ADT y LeLi ... 284
7.4.1: Introducción ... 284
7.4.2: Las declaraciones en LeLi ... 284
7.4.3: Los ámbitos en LeLi ... 284
7.4.4: El sistema de tipos de LeLi ... 286
Sección 7.5: Comprobación de tipos - Primera pasada ... 292
7.5.1: Comprobación de tipos en dos pasadas ... 292
7.5.2: Definición del analizador ... 294
7.5.3: Fase 1: Creación de ámbitos ... 296
7.5.4: Fase 2: Preparación de las expresiones ... 302
7.5.5: Fase 3: Preparación de los tipos ... 303
7.5.6: Fase 4: Preparación del ámbito global ... 304
7.5.7: Fichero LeLiSymbolTreeParser.g ... 307
7.5.8: Notas finales sobre el analizador ... 316
Sección 7.6: Comprobación de tipos – segunda pasada ... 317
7.6.1: Introducción ... 317
7.6.2: Definición del analizador ... 317
7.6.3: Fase 1 – Mantener el ámbito actual ... 319
7.6.4: Fase 2: Adición de las variables locales a los ámbitos ... 322
7.6.5: Fase 3: Expresiones ... 322
7.6.6: Fase 4: Accesos ... 331
7.6.7: Fichero LeLiTypeCheckTreeParser.g ... 341
7.6.8: Notas finales sobre el analizador ... 358
Sección 7.7: Compilación y ejecución ... 359
7.7.1: Compilación ... 359
7.7.2: Ejecución ... 359
Sección 7.8: Conclusión ... 364

Capítulo 8: Generación de código ... 365

Sección 8.1: Introducción ... 367
8.1.1: No generaremos código ... 367
8.1.2: Estructura del capítulo ... 367
Sección 8.2: Las Máquinas ... 368
8.2.1: Definiciones ... 368
8.2.2: Componentes principales ... 368
8.2.3: Juego de órdenes ... 370
8.2.4: Las otras máquinas ... 374
Sección 8.3: Gestión de la memoria ... 376
8.3.1: Definición ... 376
8.3.2: División clásica de la memoria ... 376
8.3.3: Acerca de la pila ... 378
8.3.4: Acerca del motículo ... 386
8.3.5: Acerca del offset ... 391
Sección 8.4: Código intermedio ... 395
8.4.1: Introducción ... 395
8.4.2: Proposiciones ... 395
8.4.3: Objetivos y ventajas del código intermedio ... 396
Sección 8.5: Generación de instrucciones y expresiones ... 397
8.5.1: Introducción ... 397
8.5.2: Instrucciones condicionales ... 397

8.5.3: Bucles ... 398
8.5.4: Instrucciones/expresiones ... 399
Sección 8.6: Optimización de código ... 402
8.6.1: Introducción ... 402
8.6.2: Optimizaciones independientes de la máquina objetivo ... 402
8.6.3: Optimizaciones dependientes de la máquina objetivo ... 404
8.6.4: Portabilidad vs. eficiencia ... 406
Sección 8.7: Miscelánea ... 408
8.7.1: Introducción ... 408
8.7.2: Mapeado de tipos ... 408
8.7.3: Librerías y tipos básicos del sistema ... 410
Sección 8.8: Conclusión ... 412

Capítulo 9: Conclusiones ... 413

Sección 9.1: Introducción ... 414
Sección 9.2: ANTLR y el análisis léxico ... 415
9.2.1: La primera impresión ... 415
9.2.2: Usabilidad ... 415
9.2.3: Eficiencia ... 415
9.2.4: Prestaciones ... 416
9.2.5: Carencias ... 416
9.2.6: Conclusión ... 417
Sección 9.3: ANTLR y el análisis sintáctico ... 418
9.3.1: La primera impresión ... 418
9.3.2: Usabilidad ... 418
9.3.3: Eficiencia ... 418
9.3.4: Prestaciones ... 418
9.3.5: Carencias ... 419
9.3.6: Conclusión ... 419
Sección 9.4: ANTLR y el análisis semántico ... 420
9.4.1: La primera impresión ... 420
9.4.2: Usabilidad ... 420
9.4.3: Eficiencia ... 420
9.4.4: Prestaciones ... 420
9.4.5: Carencias ... 421
9.4.6: Conclusión ... 422
Sección 9.5: Conclusiones finales ... 423
9.5.1: Sobre ANTLR ... 423
9.5.2: Sobre el manejo de los ASTs ... 423
9.5.3: Sobre la eficiencia ... 423
9.5.4: Sobre el futuro de ANTLR ... 425
9.5.5: ¿Merece la pena ANTLR? ... 425
9.5.6: Final ... 426

Apéndice A: Referencias ... 427

Apéndice B: Glosario ... 428

Apéndice C: Cuestiones técnicas ... 435

Sección C.1: Instalación de ANTLR sobre Windows ... 436
Sección C.2: Internacionalización ... 438
C.2.1: El problema ... 438
C.2.2: Defensa de la internacionalización ... 438
C.2.3: Editando los ficheros ... 438
C.2.4: Recompilando ... 439

Sección C.3: Interacción de ANTLR con otros programas..... 441
C.3.1: GNU Emacs/Xemacs..... 441
C.3.2: La plataforma Eclipse..... 442
C.3.3: Otros programas..... 443

Apéndice D: Contenido del CD-ROM.....445

Índice de ilustraciones

Ilustración 1.1 Estructura de este documento.....4
Ilustración 1.2 Un ordenador muy básico..... 6
Ilustración 1.3 Funcionamiento de un ensamblador..... 8
Ilustración 1.4 Funcionamiento de un compilador.....8
Ilustración 1.5 Estructura básica de un compilador..... 10
Ilustración 1.6 Estructura de un AST sencillo..... 11
Ilustración 1.7 Comparación de las herramientas..... 14
Ilustración 1.8 Autómata LR para una gramática de 4 reglas..... 25
Ilustración 2.1 Funcionamiento de ANTLR.....31
Ilustración 2.2 Primer flujo de información: caracteres..... 35
Ilustración 2.3 Flujo de tokens entre el lexer y el parser.....36
Ilustración 2.4 Flujo (izquierda) y árbol (derecha) de información.....38
Ilustración 2.5 Intercambio de ASTs entre las diferentes partes de un compilador..... 39
Ilustración 2.6 #(A B C)..... 39
Ilustración 2.7 #(A B #(C D E))..... 39
Ilustración 2.8 Árbol limitado (derecha) y corriente (izquierda)..... 42
Ilustración 2.9 #(OP_MAS IDENT IDENT)..... 50
Ilustración 2.10 AST para una expr_suma simplificada..... 50
Ilustración 2.11 Árbol degenerado para expr_producto..... 51
Ilustración 2.12 Árbol degenerado para expr_suma..... 51
Ilustración 2.13 #(A #(B C D))..... 52
Ilustración 2.14 AST del bucle while..... 53
Ilustración 5.1 Árbol AST con diabetes sintáctica..... 128
Ilustración 5.2 Árbol AST deseable..... 128
Ilustración 5.3 AST de un acceso..... 137
Ilustración 5.4 AST degenerado de un acceso..... 137
Ilustración 5.5 Ventana SWING mostrando el AST de "Hola mundo"..... 158
Ilustración 6.1 Recuperación de errores en el reconocimiento..... 167
Ilustración 6.2 Pila de llamadas antes y después de recuperar un error..... 178
Ilustración 6.3 Relación jerárquica de las excepciones de ANTLR..... 199
Ilustración 6.4 AST sin recuperación de errores..... 206
Ilustración 6.5 AST con la recuperación de errores funcionando..... 220
Ilustración 7.1 Análisis semántico dividido en subtareas..... 253
Ilustración 7.2 Esquema de clases para el análisis semántico..... 254
Ilustración 7.3 Jerarquía de antlraux.context.ContextException..... 267
Ilustración 7.4 Ámbitos y enmascaramiento..... 270
Ilustración 7.5 Doble ordenamiento de las declaraciones en los ámbitos..... 271
Ilustración 7.6 ASTs sin mostrar información adicional..... 282
Ilustración 7.7 ASTs con información adicional mostrada..... 282

Ilustración 7.8 Esquema de paquetes de antlraux.context....283
Ilustración 7.9 Relación entre LeLiType y LeLiMetaType....290
Ilustración 7.10 AST de un acceso....331
Ilustración 7.11 AST de un "Hola mundo"....364
Ilustración 8.1 La última fase del proceso....367
Ilustración 8.2 Esquema de una CPU....368
Ilustración 8.3 División clásica de la memoria....376
Ilustración 8.4 Típica línea de código....377
Ilustración 8.5 Orden en varias líneas de memoria....377
Ilustración 8.6 Comportamiento de la pila con gosub....379
Ilustración 8.7 Estado inicial de la máquina....381
Ilustración 8.8 Va a empezar a ejecutarse f2....381
Ilustración 8.9 Primeros preparativos de f2....382
Ilustración 8.10 Listo para ejecutar f2....382
Ilustración 8.11 Las vtables....384
Ilustración 8.12 Instancias de una clase apuntando a su descriptor....385
Ilustración 8.13 Montículo con 5 elementos alojados....388
Ilustración 8.14 Montículo con segmentos libres y ocupados....388
Ilustración 8.15 Disposición en memoria de Persona....393
Ilustración 8.16 Lugar del código intermedio....396
Ilustración 8.17 Desenrollado de un bucle....405
Ilustración 8.18 El formato de número flotante de 32 bits del IEEE....408

Capítulo 1: Preámbulos

"El lenguaje es el vestido de los pensamientos"
Samuel Johnson

Capítulo 1: Preámbulos 1

Sección 1.1: Introducción 2
1.1.1: Objetivos – LeLi y antlraux 2
1.1.2: Requisitos 2
1.1.3: Enfoque 2
1.1.4: Estructura 3
1.1.5: Notación 4

Sección 1.2: Breve repaso a la teoría de compiladores 6
1.2.1: Conceptos básicos: software y hardware 6
1.2.2: Un poco de historia 7
1.2.3: Ensambladores y compiladores 8
1.2.4: Intérpretes y máquinas virtuales 9
1.2.5: El proceso de compilación 9
Lenguaje 9
Fases del proceso de compilación 10
1.2.6: "Compiladores de compiladores" 12

Sección 1.3: Algoritmos de análisis 15
1.3.1: La sintaxis EBNF 15
Reglas básicas : BNF 15
EBNF 17
Acciones 18
Otros 19
1.3.2: "De arriba a abajo"; Analizadores recursivos descendentes 19
Lookahead > 1 21
Enriqueciendo el algoritmo: pred-LL(k) 21
1.3.3: "De abajo a arriba"; analizadores LR 23
El lookahead 24
Implementación 24
1.3.4: Comparación entre LR(k) y LL(k) 25
Potencia 25
Velocidad 26
Facilidad de uso 27

Sección 1.4: Conclusión 28

Sección 1.1: Introducción

1.1.1: Objetivos – LeLi y antlraux

Este documento pretende presentar la herramienta ANTLR, una herramienta escrita en java y dedicada al desarrollo de intérpretes. Podría considerarse como un tutorial sobre dicha herramienta.

La estrategia que se va a seguir para lograr este propósito será la de desarrollar, comentando cada paso, un compilador para un lenguaje de una envergadura media. Dicho lenguaje se llamará LeLi; las diferentes fases de su desarrollo conformarán el hilo argumental de este libro.

Paralelamente al desarrollo del compilador de LeLi se desarrollará una librería de clases genéricas, que al no estar directamente relacionadas con LeLi podrán ser utilizadas en otros desarrollos con ANTLR. El nombre de esta librería será antlraux.

Tanto el compilador de LeLi como la librería antlraux irán incluidos en el CD-ROM que acompaña a este documento. Para versiones más recientes de antlraux, consúltese el sitio **http://antlraux.sourceforge.net**. Es posible (aunque improbable) que aparezcan nuevas versiones de LeLi; si se da el caso, éstas estarán disponibles en mi sitio web personal, **http://imaginatica.us.es/~enrique**.

1.1.2: Requisitos

Dado que las herramientas más populares en desarrollo de intérpretes (especialmente de compiladores) son bison++ y flex++, las compararé con ANTLR en más de una ocasión. Por lo tanto es deseable, aunque no imprescindible, que el lector los conozca de antemano.

A pesar de que en este capítulo se hace una introducción a la teoría de compiladores, es recomendable que el lector ya tenga ciertos conocimientos de compiladores antes de enfrentarse a este documento.

El único requisito imprescindible será tener un conocimiento básico del lenguaje java, que será el utilizado para implementar tanto el compilador de LeLi como la librería antlraux. También serán necesarios conocimientos relacionados con la programación orientada a objetos, como la herencia o el polimorfismo.

1.1.3: Enfoque

A la hora de escribir un libro sobre cualquier tema técnico hay muchas formas de presentar la información al lector, o "enfoques". Algunos de los enfoques que más me molestan son:

- El enfoque "matemático-formal": El lenguaje que se emplea para presentar la información es del estilo de " $\forall a \in B \exists c \in D / a + c = n$ ". En ocasiones este lenguaje es el más indicado para presentar un concepto, pero esto rara vez ocurre en el ámbito de los compiladores. Un documento típicamente matemático-formal sería aquel que define el conjunto PRIMERO de una regla con símbolos matemáticos.
- "Orientado a diagramas": la mayoría de los conceptos se presentan en forma de diagramas con muchos círculos y flechas, presumiblemente porque facilitan la comprensión de conceptos. Los diagramas son potentes herramientas para fijar conceptos, pero por lo general no son buenas para definirlos; para eso es mejor usar palabras y frases. Un texto orientado a diagramas es aquel que define los diferentes tipos de reglas BNF utilizando diagramas de flujo.
- "Guías de referencia": son listados muy voluminosos de datos que no sirven para aprender,

sino para buscar detalles particulares sobre algo que ya se conoce.

En oposición a estos enfoques, mi objetivo es tratar de utilizar un enfoque tipo "tutorial de programación". Intentaré usar lenguaje natural en combinación con código cuando sea posible. Trataré de que todos los conceptos se interrelacionen de manera que el documento pueda leerse capítulo a capítulo de una forma más o menos amena, utilizando un tono más próximo al lector de lo que es habitual en la bibliografía existente.

Por último, trataré por todos los medios de no utilizar ecuaciones matemáticas, sino que definiré los conceptos utilizando lenguaje natural, apoyándome en código, ejemplos y algún que otro diagrama para fijar conocimientos.

1.1.4: Estructura

El texto estará dividido según la usual estructura en capítulos, cada uno versando sobre las materias que se presentan a continuación.

Actualmente usted está leyendo el capítulo de "Preámbulos". En este capítulo presentaré someramente la teoría actual de intérpretes y compiladores. Los conceptos que definiré serán análisis léxico, sintáctico y semántico, recuperación de errores, generación de código y algoritmos LL, pred-LL, LR y LALR. Si alguno de ellos le resulta extraño o no lo recuerda claramente le sugiero que eche un vistazo a este documento. En otro caso, puede pasar directamente al capítulo 2.

En el segundo capítulo se presenta la herramienta ANTLR de una manera breve. Si no conoce ANTLR, debería echarle un vistazo antes de continuar. No se preocupe, no será una lista interminable de capacidades técnicas de ANTLR.

Uno de los detalles que he observado leyendo los tutoriales de ANTLR que se encuentran por la red es que la mayoría utilizan un lenguaje "simple": una "calculadora", un "listado de libros"[1], etc. Estos lenguajes son demasiado limitados para tratar todos los temas que son necesarios al crear un compilador real, o al menos "realista", como son el manejo de ámbitos o la orientación a objetos[2]. Por lo tanto examinaré ANTLR mediante el uso de un lenguaje un poco menos simple. En el capítulo 3 definiré dicho lenguaje y en los sucesivos expondré los analizadores léxico (cap. 4) y sintáctico (cap. 5).

En el capítulo 6 presentaré algunas estrategias para implementar la recuperación de errores sintácticos son ANTLR. Definiré conceptos conocidos en el ámbito de los intérpretes, como los conjuntos PRIMERO y SIGUIENTE, o los tokens de sincronismo. Además añadiré nuevos conceptos como las trampas para excepciones y el retraso de tratamiento de errores.

El análisis semántico, que será llevado a cabo por **varios** analizadores sintácticos, se expondrá en el capítulo 7.

El capítulo 8 se centrará en la generación de código.

El capítulo 9, el último será una lista de las conclusiones que haya obtenido sobre ANTLR.

Seguidamente aparecen la bibliografía y un glosario de términos en los apéndices A y B.

En el apéndice C discuto cuestiones técnicas sobre ANTLR como son la instalación, recompilación e interacción con otros programas (especialmente IDEs).

Por último, en el apéndice D detallo el contenido del CD-ROM que acompaña a esta obra.

1 Salvo algunas honrosas excepciones, por supuesto.
2 Si bien la orientación a objetos que se implementará será tremendamente simple.

1.1.5: Notación

Seguiré la siguiente notación:

- Esto es texto normal.
- Los fragmentos de código y representaciones de la interfaz de comandos utilizarán letra `courier new` e irán delimitados con líneas. Se podrán señalar zonas de código de especial interés con un sombreado y utilizando letras en negrita:

```
/* Ejemplo de código */

class HolaMundo
{
   public static void main(String args[])
   {
      System.out.println("Hola mundo"); // Código resaltado
   }
}
```

- Los términos en otros idiomas o conceptos importante irán *en cursiva*.
- Este documento se organiza en capítulos, secciones, apartados y, si el asunto lo requiere, subapardados. Es decir, la estructura del documento es la siguiente:

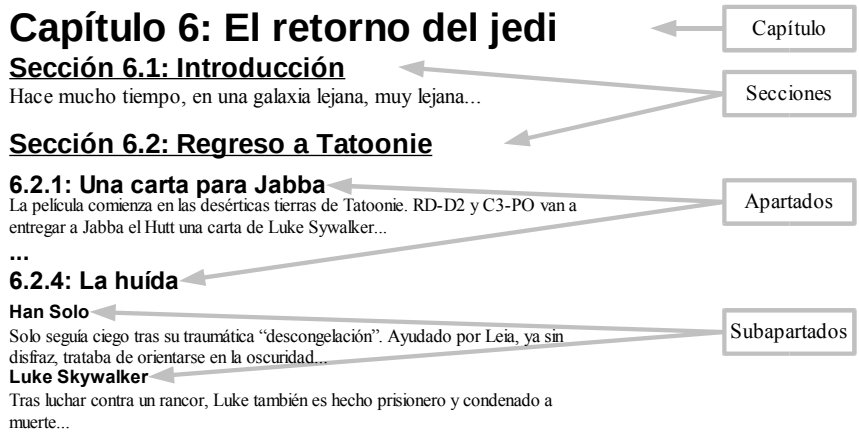

Ilustración 1.1 Estructura de este documento

- Ocasionalmente podrán encontrarse fragmentos de texto en los que el usuario debe prestar especial atención (por ejemplo, al hablarse de un error que puede cometerse). Estos fragmentos irán señalizados de la manera siguiente:

> Los párrafos representados de esta forma sirven para indicar una información que merece especial atención por parte del lector.

- Algunas secciones del documento son traducciones de partes de los manuales de ANTLR o de artículos relacionados con él. Estos fragmentos irán precedidos de una anotación como la siguiente:

Las secciones precedidas con una anotación como ésta son extractos de manuales o artículos.

Es decir, el texto que usted está leyendo en estos momentos debería de ser una traducción o copia de algún libro o manual. La anotación precedente debería indicar con precisión el origen del texto.

El final de cada una de estas secciones se marcará con 5 asteriscos, así:

*****

- A la hora de escribir este documento, no tenía claro si utilizar el plural de cortesía o un tono más personal en primera persona (es un proyecto de fin de carrera, pero a la vez pretendo que sea un tutorial sobre ANTLR y sobre los compiladores). Esta lucha interna aún se mantiene, por lo que a veces utilizaré la primera opción (diciendo cosas como "Hemos visto" o "esto lo implementaremos así") y a veces la segunda ("Yo lo hago así, pero usted debería hacerlo de esta otra manera").
- De igual manera, me referiré individualmente al lector, ya sea en segunda ("Usted puede hacer esto así") o tercera persona ("El lector debe saber que..."), aunque también podré dirigirme a todo el colectivo de potenciales lectores ("los que no sepan cómo hacer esto, que revisen la sección anterior").
- Un apunte final, esta vez en cuanto a los nombres de las clases en java: hay dos maneras de hacer referencia a las clases del dicho lenguaje; por un lado está el "nombre completo" de la clase, que incluye todos los paquetes y sub paquetes de la clase. Esta nomenclatura puede ser muy complicada de utilizar, porque tiende a generar nombres muy largos (por ejemplo, en una ocasión tendremos que utilizar la clase `antlraux.context.types.CommonType`). Así que en muchas ocasiones utilizaré el "nombre reducido", que permite referirse a la clase utilizando únicamente su nombre (es decir, escribiré simplemente `CommonType`). Es muy improbable que un nombre de una clase se repita en más de un paquete, así que esta medida no debería suponer ningún problema.

Sección 1.2: Breve repaso a la teoría de compiladores

1.2.1: Conceptos básicos: software y hardware

Un compilador es, muy a *grosso modo*, "un programa que sirve para hacer otros programas, mediante el uso de un lenguaje de programación". Un programa es "una serie de órdenes en la memoria de un ordenador".

Un ordenador es la unión de una o más unidades de memoria, una o más unidades de cálculo y uno o más registros de cálculo intermedios, unidos entre sí por una serie de buses de información, de tal manera que el conjunto sea capaz de ejecutar programas, codificados en forma de instrucciones en la memoria.

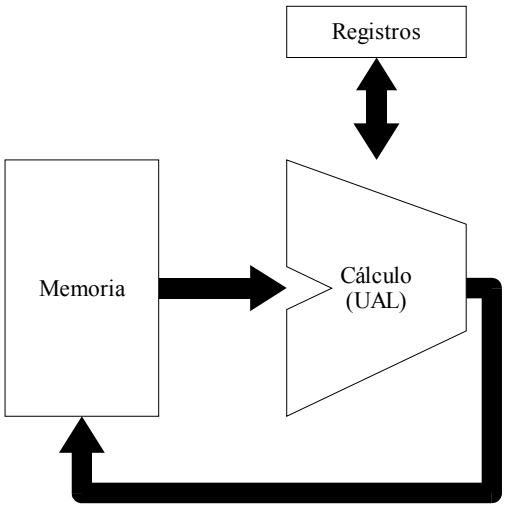

Ilustración 1.2 Un ordenador muy básico

La memoria es una zona en la que se almacena la información. Los microprocesadores actuales vienen dotados de una unidad de memoria, físicamente dentro del procesador, que se llama memoria caché. Esta memoria es bastante más rápida que la memoria RAM, y bastante más pequeña. Hay diversas técnicas que permiten guardar en la memoria caché los datos más utilizados de la RAM, incrementando enormemente la velocidad de procesado. No obstante, cuando la memoria caché no contiene un dato necesitado tiene que recurrir a la RAM convencional, o incluso al disco duro, si el sistema operativo está utilizando técnicas de *swapping*.

Para acelerar los accesos de memoria se utiliza también una zona de memoria extremadamente rápida dividida en "registros". El número de registros de cada máquina y su capacidad depende de cada modelo.

La zona en la que se realizan los cálculos se suele llamar UAL (Unidad Aritmético – Lógica). Por lo general la UAL trabaja directamente con los registros en lugar de acceder directamente a la memoria: suele ser más rápido.

Por último, todos los componentes de un microprocesador están unidos por líneas de transmisión de información llamadas "buses". Las características más importantes de un bus son su velocidad de transmisión (que en los buses internos de un procesador es altísima) y su capacidad, es decir, el número de bits simultáneos que pueden transmitir. Cuando se dice que "un procesador es de

64 bits" se suele hacer referencia a la capacidad de los buses internos.

Los programas que un ordenador debe ejecutar se almacenan codificados en la memoria en forma de números binarios. La forma de codificar la información también varía según el modelo de la máquina, aunque existen algunas reglas generales: los bits más significativos suelen codificar el tipo de operación (suma, resta) que la UAL debe realizar. El resto de los bits se utilizan para codificar los "operandos", es decir, los registros o direcciones de memoria que se van a utilizar en la operación. Los códigos binarios de las operaciones suelen agruparse y presentarse en hexadecimal. Así, el número "0xFFAC" podría significar para un procesador procesador "toma el valor del registro A, súmale el del registro B y guarda el resultado en A".

Generalmente al soporte físico (UAL, buses, memoria + otros dispositivos, como la fuente de alimentación o el teclado) del ordenador se le llama *hardware*, mientras que a los programas que se almacenan en su memoria se le llama *software*.

1.2.2: Un poco de historia

En los albores de la programación no había ningún tipo de abstracción. La "programación" en aquel entonces era completamente diferente a como la conocemos ahora. En general el hardware proporcionaba un medio de escribir códigos hexadecimales en su memoria, y una vez en memoria los códigos eran procesados.

El primer ordenador personal que jamás se vendió fue el MITS ALTAIR 8800. Esta pieza de tecnología venía equipada con 256 flamantes *bytes* (no, no *Kbytes*) de memoria RAM – ampliables, eso sí, a 2, 4, 8 ¡e incuso 12 KB!

Con la configuración por defecto no se podía hacer mucho. No podía conectarse a una televisión, y no disponía de ningún dispositivo de salida a excepción de 35 LEDs en su parte frontal. Tampoco existía ningún teclado, y la introducción de códigos hexadecimales se realizaba utilizando algunas "palancas", que al ser accionadas modificaban el valor de los bits en la memoria, mostrándose el cambio en los LEDs.

La primera evolución fueron los ensambladores. En lugar de escribir el código directamente en binario, el programador utilizaba nemotécnicos. Así, se pasó de escribir "0xFFAC" a escribir algo parecido a "ADD A,B" (suma los registros A y B, y guarda el resultado en A). Para ello era necesario contar con un aparato extra, llamado teletipo. Este artilugio estaba dotado de un teclado, y enviaba la pulsación de teclas al Altair. Por supuesto, hacía falta un ensamblador. El ensamblador era un programa que permitía traducir el lenguaje "con nemotécnicos", que a la sazón también se llamó ensamblador, en código binario interpretable por la máquina. Los nemotécnicos no aislan al programador del hardware; para programar en lenguaje ensamblador es necesario conocer la máquina tan bien como cuando se programa directamente en código máquina. Sin embargo, para un ser humano normal resulta mucho más cómodo utilizar los nemotécnicos que el código binario de la máquina – que desde entonces se denomina simplemente "código máquina".

Para poder utilizar un ensamblador era necesario contar, al menos, con la ampliación de memoria de 4KB. Ahora bien, con 4KB ya era posible utilizar un lenguaje de programación de más alto nivel: el BASIC.

BASIC proporcionaba un (limitado) entorno de programación que permitía "ignorar" el hardware. Conociendo una serie de conceptos abstractos, cualquiera podría escribir software para el Altair. `ADD A,B` pasó a convertirse en LET `speed=speed+aux`.

Con una ampliación de 8KB se podía ejecutar 8K BASIC. 8K BASIC proporcionaba un juego de

instrucciones más grande que 4K BASIC. Por último, Extended BASIC solamente funcionaba con el módulo de 12KB.

Extended BASIC fue el primer software comercial que desarrolló un tal Bill Gates.

1.2.3: Ensambladores y compiladores

Un ensamblador es a su vez un programa. Dicho programa se encarga de traducir automáticamente el lenguaje ensamblador (`ADD A, B`) a código "máquina" (`0xFFAC`).

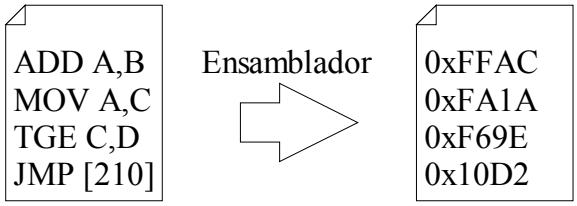

Ilustración 1.3 Funcionamiento de un ensamblador

Un ensamblador es un programa relativamente simple, dado que traducir los nemotécnicos a código máquina es una acción muy directa.

Los compiladores, por regla general, se limitan a traducir un lenguaje de programación de alto nivel a ensamblador, y después un ensamblador se encarga de generar el código máquina. Al lenguaje de alto nivel que define un programa de software se le suele llamar "código fuente".

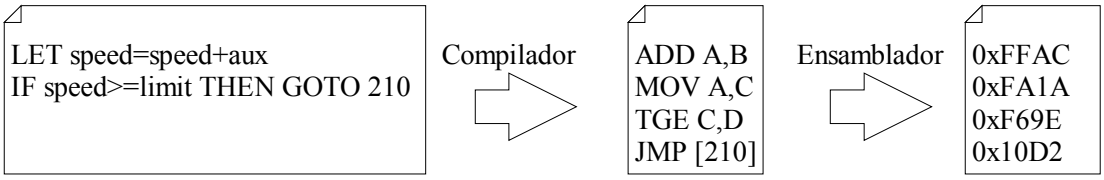

Ilustración 1.4 Funcionamiento de un compilador

Con el BASIC del Altair seguía siendo sencillo traducir. El lenguaje de programación aún recordaba bastante la máquina subyacente.

Con el tiempo, no obstante, los lenguajes de programación han ido evolucionando más y más, añadiendo más y más abstracción, de manera que hoy en día es posible encontrar programadores que desconocen por completo la arquitectura de la máquina sobre la que están programando[3].

Los lenguajes compilados más utilizados fueron, primeramente, FORTRAN, y más tarde C.

El incremento de la abstracción y de las prestaciones de los lenguajes de programación repercuten en la complejidad de los compiladores, que cada vez tienen que realizar más tareas para poder finalmente "traducir" a código ensamblador. Se hizo necesario dividir los programas en varias unidades de compilación independientes (usualmente ficheros de código diferentes). Esta división permitía recompilar una sola de las unidades de compilación sin tener que recompilar el resto, a la vez que permitió la aparición de "librerías", permitiendo a un programador aprovechar el código escrito por otros.

Para obtener un ejecutable que dependía de más de una unidad de compilación se hacía necesaria una fase más, en la que se "enlazaban" las diferentes partes para formar el programa final. Esta última fase no la realizaba el compilador, sino una herramienta externa llamada "enlazador"

3 ¿Qué porcentaje de programadores de java conoce la arquitectura de la máquina virtual de java?

(*linker*). Dado que nosotros trabajaremos con una única unidad de compilación no utilizaremos enlazadores, así que los ignoraremos a partir de ahora.

1.2.4: Intérpretes y máquinas virtuales

Los lenguajes de programación que requieren ser traducidos por compilador a código máquina se llaman "lenguajes compilados". En oposición a los lenguajes compilados encontramos los lenguajes interpretados.

Los programas escritos en lenguajes interpretados no se traducen hasta el momento de ser ejecutados. Para ejecutarlos es necesario que un software adicional, llamado "intérprete", los lea, y ejecute las instrucciones que en ellos se codifican.

Un programa escrito en un lenguaje interpretado se ejecutará, por tanto, más lentamente que su equivalente compilado, principalmente porque tendrá que estar compartiendo el hardware con el intérprete.

BASIC ha sido tradicionalmente un lenguaje interpretado, aunque han existido tantas versiones de BASIC que es muy difícil asegurar que todas sean interpretadas.

Un concepto parecido al de los lenguajes interpretados lo encontramos en las máquinas virtuales. Una máquina virtual es un "simulador": sirve para simular un ordenador dentro de otro ordenador.

Considérese el siguiente escenario: nosotros disponemos de un PC convencional y, en un ataque de nostalgia, deseamos ejecutar sobre él un programa diseñado para el ZX Spectrum 48k de Sinclair, que a la sazón fue escrito en BASIC para el spectrum 48k. Si dispusiéramos del código fuente en BASIC del software, podríamos buscar un compilador para nuestro ordenador del mismo programa, y obtendríamos un ejecutable para el PC.

No obstante, imagínese el lector que no dispone del código fuente, sino solamente del código máquina del Spectrum. Aún hay una manera de ejecutar el software: hay que buscar un programa que interprete el código máquina del Spectrum como si se tratara de un lenguaje interpretado en el PC. Dicho programa existe, y se llama emulador[4].

Otro ejemplo de máquina virtual lo encontramos en la máquina virtual de java: java es un lenguaje que, al compilarse, genera un código máquina para una máquina "inexistente". Los ficheros *.class son instrucciones en código máquina pensadas para ejecutarse sobre una máquina virtual: cada sistema operativo dispone de su propio "emulador" de la máquina virtual de java, y así es como se consigue que java sea un lenguaje capaz de ejecutarse en varias plataformas. Volveremos a este punto más adelante.

1.2.5: El proceso de compilación

Lenguaje

Todo lenguaje de programación está formado por un conjunto de "símbolos básicos", que se agrupan para formar los elementos de un "vocabulario", de la misma manera que en la lengua española las letras se agrupan para formar las palabras. Los símbolos de los lenguajes de programación son los caracteres que forman el código, y a sus "palabras" les llamaremos *tokens*.

Las "reglas" de un lenguaje indican cómo pueden o no agruparse los diferentes tokens. A estas reglas se las llama "reglas sintácticas". Es frecuente que el vocabulario se defina implícitamente al

4 La posesión de emuladores en general no constituye delito, pero sí suele serlo estar en posesión de copias no legítimas del software de máquinas emuladas.

definir las reglas sintácticas de un lenguaje. Al conjunto de reglas sintácticas de un lenguaje se la llama "gramática".

El conjunto de las reglas sintácticas del castellano es la gramática castellana.

Por último, un lenguaje suele ir acompañado de ciertas reglas que complementan a las reglas sintácticas. En el lenguaje natural utilizamos la lógica y la memoria para saber si una frase tiene "sentido". Por ejemplo, aunque en español la frase "El sediento bebió un vaso de tierra" es perfectamente válida a nivel gramatical, detectamos un error gracias a nuestro sentido común.

Fases del proceso de compilación

Lo primero que debe hacer un compilador es comprobar que la información que se le suministra pertenece a su lenguaje (no hay errores léxicos, sintácticos ni semánticos). Si es así, el intérprete debe representar de alguna manera la información que le se le suministró para poder trabajar con ella, y finalmente traducir dicha información a código máquina.

Un esquema de dicho funcionamiento es el que se muestra en la siguiente figura.

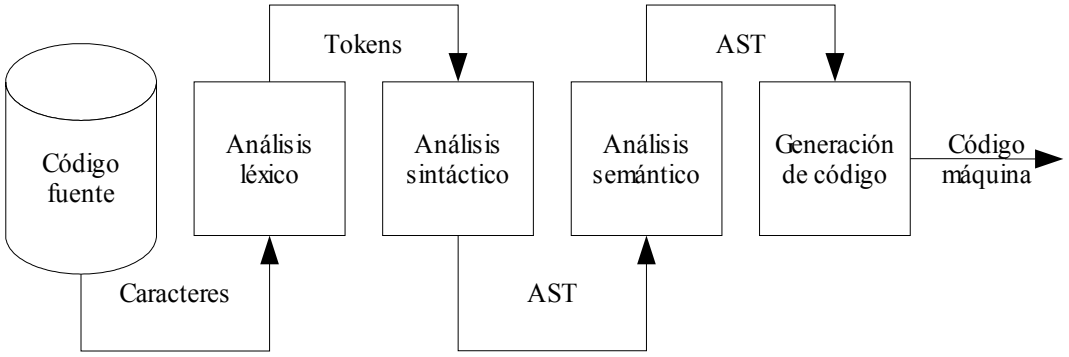

Ilustración 1.5 Estructura básica de un compilador

A primera vista podemos distinguir que hay dos tipos de elemento en dicho gráfico: los "elementos activos" (figuras cerradas) y los "flujos de datos", representados como flechas que unen los diferentes elementos activos. Si entre los elementos activos "A" y "B" hay una flecha llamada "C", eso quiere decir que "A" produce el flujo de datos "C", que es usado por "B". Analicemos brevemente cada elemento y cada flujo:

- Código fuente: Es información almacenada en la memoria de un ordenador. Suele tratarse de uno o varios ficheros de texto, normalmente en el disco duro de la máquina[5]. En estos ficheros hay cierta información cuyo fin es provocar ciertas acciones en una máquina objetivo (que puede no ser la que está "interpretándolos"). Para ello, los ficheros son leídos del disco y pasados a la memoria, conformando el flujo denominado "Caracteres"[6].
- Análisis léxico: Esta fase tiene que ver con el "vocabulario"del que hablábamos más arriba. El proceso de análisis léxico agrupa los diferentes caracteres de su flujo de entrada en *tokens*. Los tokens son los símbolos léxicos del lenguaje; se asemejan mucho a las palabras del lenguaje natural. Los tokens están identificados con símbolos (tienen "nombres") y suelen contener información adicional (como la cadena de caracteres que los originó, el fichero en el que están y la línea donde comienzan, etc). Una vez son identificados, son transmitidos al siguiente nivel de análisis. El programa que permite realizar el análisis léxico es un analizador léxico. En inglés se le suele llamar *scanner* o *lexer*.

5 ¡Los ficheros binarios que se interpretan también existen!
6 Que también podría ser un flujo de bytes.

Para ejemplificar cómo funciona un lexer vamos a usar un ejemplo: Dado el fragmento de código imaginario siguiente, que podría servir para controlar un robot en una fábrica de coches (por simplicidad, suponemos que es el ordenador que realiza el análisis es el del propio robot, es decir, no hay compilación cruzada):

```
Apretar (tuercas);
Pintar(chasis+ruedas);
```

El analizador léxico del robot produciría la siguiente serie de tokens:

```
RES_APRETAR PARENT_AB NOMBRE PARENT_CE PUNTO_COMA
RES_PINTAR PARENT_AB NOMBRE SIM_MAS NOMBRE PARENT_CE PUNTO_COMA
```

- Análisis sintáctico: En la fase de análisis sintáctico se aplican las reglas sintácticas del lenguaje analizado al flujo de tokens. En caso de no haberse detectado errores, el intérprete representará la información codificada en el código fuente en un Árbol de Sintaxis Abstracta, que no es más que una representación arbórea de los diferentes patrones sintácticos que se han encontrado al realizar el análisis, salvo que los elementos innecesarios (signos de puntuación, paréntesis) son eliminados. En adelante llamaremos AST a los Árboles de Sintaxis Abstracta.

  El código que permite realizar el análisis sintáctico se llama "analizador sintáctico". En inglés se le llama *parser*, que significa "iterador" o directamente *analyzer* ("analizador").

  Continuando con el ejemplo anterior, el análisis léxico del robot produciría un AST como el siguiente:

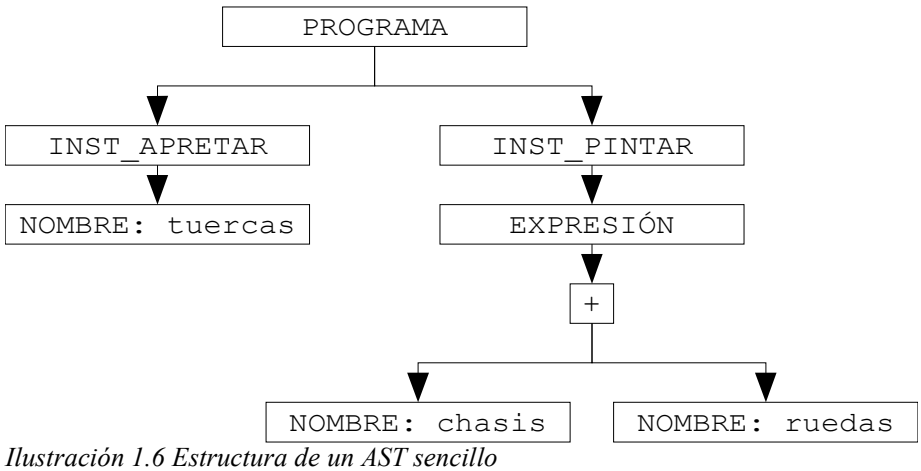

Ilustración 1.6 Estructura de un AST sencillo

  El robot ha identificado dos instrucciones, una de "apretar" y otra de "pintar". Los tokens del tipo "NOMBRE" tienen información adicional. Para operar sobre varios objetos a la vez (como sobre el chasis y las ruedas) se pueden escribir varios nombres unidos por el símbolo '+'.

- Análisis semántico: El análisis semántico del árbol AST empieza por detectar incoherencias a nivel sintáctico en el AST. Si el AST supera esta fase, es corriente enriquecerlo para realizar un nuevo análisis semántico. Es decir, es corriente efectuar varios análisis semánticos, cada uno centrado en aspectos diferentes. Durante éstos análisis el árbol es enriquecido y modificado.

  Cualquier herramienta que realice un análisis semántico será llamada "analizador semántico" en este texto. En la bibliografía inglesa suelen referirse a los analizadores semánticos como *tree parsers* (o "iteradores de árboles").

  En el caso de nuestro robot, podríamos considerar que aunque el chasis de un coche se pueda pintar, las ruedas no son "pintables". Por lo tanto se emitiría un mensaje de error (a pesar de que el código suministrado fuese gramaticalmente válido) y no continuaría el análisis. Imaginemos ahora

que el operario del robot se da cuenta de su error y sustituye "ruedas" por "llantas", que sí es reconocido por el robot como algo que se puede pintar. En tal caso, dada la simplicidad del AST, no es necesario enriquecerlo de ninguna forma, por lo que se pasaría a la siguiente fase.

- Generación de código: En esta fase se utiliza el AST enriquecido, producto del proceso de análisis semántico, para generar código máquina. Nuestro robot, una vez eliminado el error sintáctico detectado, generaría el código binario necesario para apretar y pintar los elementos adecuados, pudiendo ser utilizado en el futuro. Dicho código (en ensamblador-para-robot) sería parecido al siguiente:

```
mientras queden tuercas sin apretar
   busca tuerca sin apretar.
   aprieta tuerca.
sumergir chasis en pozo de pintura.
colocar llanta1 bajo ducha de sulfato. Esperar. Despejar ducha.
colocar llanta2 bajo ducha de sulfato. Esperar. Despejar ducha.
colocar llanta3 bajo ducha de sulfato. Esperar. Despejar ducha.
colocar llanta4 bajo ducha de sulfato. Esperar. Despejar ducha.
Desconectar.
```

¡Recordemos que el código anterior está codificado en binario, todo generado a partir de dos líneas de código fuente! El compilador es un "puente" entre el operario y el robot, que genera por el ser humano todo el código repetitivo en binario insertando cuando es necesario elementos auxiliares (como la orden de desconexión, que el operario podría olvidarse de introducir). Como vemos el uso de los compiladores aumenta la productividad.

En este documento nos vamos a centrar en las tres primeras fases del análisis, que son las cubiertas por ANTLR de una manera novedosa con respecto a los enfoques anteriores. Una vez realizado el análisis semántico, la generación de código no se diferenciará mucho de como se haría antiguamente.

1.2.6: "Compiladores de compiladores"

Los compiladores son las herramientas que sirven para hacer programas, pero también son programas.

¿Cómo se hace, entonces, un compilador?

En la primera época de la programación, no era inusual ver ensambladores construidos directamente en código máquina. De todas maneras en cuanto el primer ensamblador decente aparecía, todo el mundo programaba en ensamblador. Los primeros compiladores de basic estaban, por lo tanto, desarrollados en ensamblador.

Rápidamente la potencia de los ordenadores y de los nuevos lenguajes permitió escribir los ensambladores y compiladores en lenguajes de alto nivel. Hoy en día el ensamblador se usa muy puntualmente, en controladores de hardware y en ciertas rutinas gráficas. El resto del software es escrito en lenguajes de programación de más alto nivel, como C, C++ o java.

Volviendo al pasado, la mayoría del software empezó a escribirse en lenguajes de más alto nivel, y la misma suerte corrieron los compiladores.

Así, cuando Bjarne Strongstroup creó el primer compilador de C++, lo hizo programando en un lenguaje "intermedio" entre C y C++, llamado "C con Objetos" (*C with Objects*). El compilador del lenguaje C con Objetos estaba hecho en C, y el primer compilador de C++ estaba hecho en C with Objects, que prácticamente no se utilizó para otra cosa. Otros compiladores de C++ han sido construidos directamente en C, ¡y algunos compiladores de C++ han sido escritos

directamente en C++!

Ya fuera en ensamblador o con utilizando un lenguaje intermedio, el problema de escribir un compilador "a mano" es que hay que realizar muchas tareas muy repetitivas. Uno no puede evitar tener la impresión de que todo se podría automatizar enormemente.

Cuando se hizo patente esta necesidad de automatización aparecieron las primeras herramientas de ayuda a la construcción de compiladores. Lo que hacen estas herramientas es generar código en un lenguaje de programación (C, C++ y más tarde java) para ahorrar al programador la parte repetitiva de la programación de compiladores, pudiendo éste dedicarse al diseño.

Varias universidades construyeron herramientas de este tipo, pero fueron `yacc` y `lex` las que más se han extendido, llegando a considerarse "estándares" a la hora de realizar un compilador.

Al principio de los 70, Stephen C. Johnson desarrolló `yacc` (*Yet Another Compiler Compiler*) laboratorios Bell, usando un dialecto portable del lenguaje C. `yacc` es una herramienta capaz de generar un analizador sintáctico en C a partir de una serie de reglas de sintaxis que debe cumplir. Dichas reglas se especifican en un lenguaje muy sencillo.

`yacc` se apoya en la herramienta `lex` para el análisis léxico. `lex` fue desarrollada por Eric Schmidt. `lex` también fue desarrollado en C, y también genera un analizador en C.

`lex` y `yacc` sirvieron como base a `flex` y `bison`, que se consideran sus herederas. `flex` y `bison` son dos productos de la FSF (*Free Software Foundation*). Actualmente son las herramientas más utilizadas en el desarrollo de compiladores.

`flex` (*Fast LEXical analysis*) es el "relevo" de `lex`. Fue implementado por Vern Parxson y Kevin Gong, basándose en la implementación original de Jef Poskanzer, y con mucha ayuda de Van Jacobson.

`bison` fue escrito principalmente por Robert Corbett; Richard Stallman lo hizo compatible con `yacc`. Wilfred Hansen de la Carnegie Mellon University le añadió otras prestaciones como los literales cadena de más de un carácter.

Mucho ha llovido desde que bison y flex fueron escritos. Ahora el proceso de compilación está más formalizado; se admite ampliamente que es necesario crear un árbol de sintaxis abstracta si se quiere realizar un análisis semántico correctamente. Es necesario crear y recorrer de una forma estandarizada los árboles de sintaxis abstracta. Varias universidades, entre ellas la de Sevilla, utilizan herramientas para la construcción y manejo de árboles de sintaxis abstracta (`hecta` y `esa`), pero ninguna ha compartido el éxito de `flex` y `bison`.

ANTLR es un software desarrollado en java por varios individuales, aunque la idea inicial y las decisiones principales de diseño son de Terence Parr. En su proyecto de fin de carrera, Terence presentaba una manera eficiente de implementar los analizadores LL (más adelante explicaremos lo que son). Los hallazgos presentados en esta tesis fueron los que le llevaron a implementar PCCTS, que puede considerarse como la "semilla" de ANTLR.

PCCTS permite generar analizadores léxicos y sintácticos. Para recorrer los árboles de sintaxis abstracta, se desarrolló un programa compañero llamado SORCERER.

ANTLR ha sufrido dos reescrituras completas desde su inicio, incluyendo el cambio del lenguaje de programación utilizado (inicialmente fue C) y varios cambios de nombre. La versión actual (2.7.2) data de enero de 2003. Es el resultado de unir PCCTS y SORCERER en un único software.

Gracias a esta unión, mientras que flex y bison son herramientas dedicadas a una sola fase del análisis, ANTLR es capaz de actuar a *tres* niveles a la vez (cuatro si tenemos en cuenta la

generación de código):

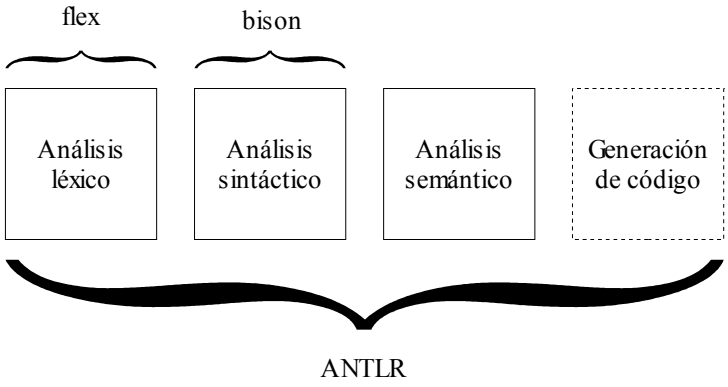

Ilustración 1.7 Comparación de las herramientas

El uso de una sola herramienta para todos los niveles tiene varias ventajas. La más importante es la "estandarización": con ANTLR basta con comprender el paradigma de análisis una vez para poder implementar todas las fases de análisis. Con `flex+bison` es necesario comprender y saber utilizar herramientas completamente diferentes (flex está basado en autómatas finitos deterministas y bison en un analizador LALR; pero más sobre ésto en la siguiente sección), además de necesitar de otras herramientas para realizar el análisis semántico.

Mostraré otras ventajas de utilizar ANTLR en lugar de los métodos más "tradicionales" a lo largo del libro.

Sección 1.3: Algoritmos de análisis

1.3.1: La sintaxis EBNF

Reglas básicas : BNF

En la sección anterior hemos mencionado que los lenguajes pueden ser especificados mediante un conjunto de reglas que llamaremos gramática. En este apartado veremos cómo vamos a especificar las gramáticas.

Definiremos las gramáticas utilizando un lenguaje especial, llamado EBNF. EBNF es una extensión del lenguaje BNF.

Una gramática es un conjunto de *reglas*. Toda regla comienza con un *nombre* (o "parte izquierda") seguido por el carácter de los dos puntos ":". A continuación aparece el *cuerpo* (o "parte derecha") de la regla. Todas las reglas terminan con el carácter de punto y coma ";".

```
nombre : cuerpo ;
```

Las reglas pueden ser de tres tipos: alternativas, enumeraciones y referencias a símbolos básicos del vocabulario o a otras reglas.

El tipo de regla más flexible es la alternativa. Las alternativas sirven para expresar "elección entre varias opciones". Las diferentes opciones de la regla se separan con un carácter de barra vertical "|". Por ejemplo, para indicar que "un perro" puede ser "un foxterrier" o "un caniche" o "un chucho" podemos escribir algo así.

```
perro: FOXTERRIER | CANICHE | CHUCHO ;
```

Hemos supuesto que los nombres de tipos de perro son símbolos básicos del vocabulario. Utilizaremos mayúsculas para referirnos a los símbolos del vocabulario. Aquí, la regla "perro" se satisface cuando en la entrada aparecen el símbolo `FOXTERRIER`, `CANICHE` o `CHUCHO`[7].

Dado que es posible insertar caracteres de separación en cualquier parte de la gramática, es usual insertar saltos de línea en las alternativas, para facilitar la lectura:

```
perro: FOXTERRIER
     | CANICHE
     | CHUCHO
     ;
```

La barra vertical es un operador introducido por comodidad. Si es necesario, una regla con alternativas puede definirse también así[8]:

```
perro: FOXTERRIER ;
perro: CANICHE ;
perro: CHUCHO ;
```

Además símbolos del vocabulario, se pueden utilizar otras reglas. Por ejemplo, obsérvese cómo se utiliza la regla `perro` en la regla `cuadrúpedo`:

7 Es usual que los símbolos terminales, y especialmente los tokens en el analizador sintáctico, se escriban con mayúsculas, mientras que los nombres de las reglas se escriben con minúsculas.

8 Esta sintaxis no puede emplearse en ANTLR, pero sí en bison.

```
cuadrúpedo : perro
           | gato
           | caballo
           ;
perro : ... /* definir perro */ ;
gato : ... /* definir gato */ ;
caballo : ... /* definir caballo */ ;
```

Obsérvese que hemos incluido comentarios del estilo de C. Admitiremos el uso de comentarios de una sola línea (con la doble barra, "//") o de varias líneas (entre "/*" y "*/").

Otro tipo de regla muy utilizado es la *enumeración*. Una enumeración no es más que una lista ordenada de referencias (a otras reglas o a elementos del vocabulario). Sirve para reconocer series de elementos. Los elementos simplemente se escriben unos detrás de otros, en el orden deseado, y separados por espacios, retornos de carro o tabulaciones.

```
frase: EL cuadrúpedo se_movía DEPRISA;
se_movía : CORRÍA
         | GALOPABA
         ;
```

En este caso los símbolos básicos del lenguaje son `EL`, `DEPRISA`, `CORRÍA` y `GALOPABA` (además de los nombres de perros citados anteriormente). frase y se_movía son reglas de la gramática.

La regla "frase" permite reconocer muchas entradas. Por ejemplo reconocerá la entrada "`EL FOXTERRIER CORRÍA DEPRISA`", y también "`EL PERCHERÓN GALOPABA DEPRISA`", asumiendo que la regla "cuadrúpedo" admita `FOXTERRIER` y `PERCHERÓN`. Nótese que otras entradas no tendrán tanto sentido; por ejemplo, si "cuadrúpedo" admite `GATO_SIAMÉS`, entonces la entrada "`EL GATO_SIAMÉS GALOPABA DEPRISA`" será considerada válida[9].

En BNF es posible utilizar enumeraciones como sub reglas de alternativas:

```
frase: EL perro PERSEGUÍA AL gato
     | EL caballo COME PIENSO
     | UN cuadrúpedo TIENE CUATRO PATAS
     ;
```

Por último hablemos de los patrones repetitivos. Los patrones repetitivos sirven para reconocer "uno o más elementos" o "cero o más elementos".

En BNF los patrones repetitivos deben implementarse con la recursión, es decir, con una regla que se llame a sí misma. Por ejemplo, para reconocer una llamada a una función con una lista de cero o más parámetros, había que escribir algo así:

```
llamada: NOMBRE PARENT_AB listaParametros PARENT_CE ;
listaParametros : parametro listaParametros // regla recursiva
                | /* nada */
                ;
parametro : ENTERO | NOMBRE ;
```

Suponiendo que `ENTERO` represente cualquier número entero, `NOMBRE` cualquier identificador, `PARENT_AB` el paréntesis abierto ("(") y `PARENT_CE` el paréntesis cerrado (")"), tendremos que la regla anterior permitirá reconocer entradas como `f(x)`, `max(a,10)` o `getMousePos()`. Nótese que para poder implementar este conjunto de reglas hemos tenido que utilizar una alternativa vacía. Si se desea que las llamadas a funciones no puedan tener valores vacíos, es decir, que su

9 Es decir, sintácticamente válida. La incoherencia aquí presentada es de tipo semántico, y debería ser reconocida en la fase de análisis semántico.

lista de parámetros sea de uno o más en lugar de de cero o más, hay que escribir lo siguiente:

```
llamada: NOMBRE PARENT_AB listaParametros PARENT_CE ;
listaParametros : parametro listaParametros
                | parametro
                ;
parametro : ENTERO | NOMBRE ;
```

EBNF

En BNF no se permite el uso de alternativas como sub reglas de una enumeración. EBNF si lo permite. Para ello es recomendable utilizar paréntesis:

```
orden: PINTAR (RUEDAS|CHASIS) DE (AZUL|AMARILLO|VERDE);
```

No conviene abusar de esta capacidad, porque aunque las sub reglas-enumeraciones se entienden muy bien, las sub reglas-alternativas oscurecen mucho el sentido de las reglas. La regla anterior se comprendería mejor así:

```
orden: PINTAR partePintable DE color ;
partePintable : RUEDAS|CHASIS ;
color : AZUL|AMARILLO|VERDE ;
```

Las sub reglas, además, admiten cualquier nivel de enraizamiento en EBNF (puede insertarse una sub regla dentro de una sub regla hasta donde se desee).

Además de la capacidad de "sub reglización", EBNF supera ampliamente a BNF en el reconocimiento de patrones repetitivos. Para ello introduce dos operadores nuevos :la *clausura positiva* (que se representa con el símbolo "+") y *cierre de Kleene* (que se representa con el asterisco, "*"). Estos dos operadores se emplean en conjunción con los paréntesis para indicar repetición. La clausura positiva indica "ésto se repite una o más veces" mientras que el cierre de Kleene indica "cero o más veces". Para el ejemplo anterior de la llamada a función podemos escribir

```
llamada: NOMBRE PARENT_AB (parametro)* PARENT_CE ;
parametro : ENTERO | NOMBRE ;
```

si deseamos "cero o más parámetros" y

```
llamada: NOMBRE PARENT_AB (parametro)+ PARENT_CE ;
parametro : ENTERO | NOMBRE ;
```

si deseamos "uno o más".

Por supuesto, también podemos utilizar la recursión (una regla puede llamarse a sí misma) aunque en la mayoría de los casos bastará con los cierres y clausuras.

Dada la naturaleza multi-sub-regla de EBNF, se pueden insertar opciones e incluso otras clausuras dentro de una clausura. En otras palabras, se pueden escribir reglas como la siguiente:

```
regla : (UN ( perro | gato ) NO ES UN caballo ) +
      ;
```

Esta regla reconoce una o más frases negando que perros o gatos sean caballos. Por ejemplo, serviría para reconocer entradas como "`UN CANICHE NO ES UN PERCHERÓN UN PEQUINÉS NO ES UN PURASANGRE`".

Anidar más de 3 niveles de reglas es posible (¡incluso se puede escribir todo un analizador en una sola regla!), pero no es recomendable porque la gramática resultante sería extremadamente difícil

de manejar.

EBNF permite dos tipos adicionales de sub reglas: la opcionalidad y la negación.

Una sub regla opcional se representa con el operador de opcionalidad, que es la interrogación ('?'). Normalmente se utiliza en conjunción con los paréntesis. La sub regla opcional es una regla que "puede estar o puede no estar en la entrada". Por ejemplo:

```
regla : UN perro (GRANDE)? NO ES UN caballo ;
```

Esta regla admite la entrada "UN CANICHE NO ES UN PURASANGRE" y también "UN CANICHE **GRANDE** NO ES UN PURASANGRE". Es decir, "GRANDE" puede "estar" o "no estar" en la entrada. Es "opcional". El operador de opcionalidad no está limitado a símbolos básicos del lenguaje: se puede insertar sub reglas, referencias a otras reglas, etc.

A nivel de implementación, ANTLR convierte las reglas opcionales en alternativas con una alternativa vacía. Así, ANTLR representa internamente el ejemplo anterior de esta manera:

```
regla : UN perro (GRANDE | /*nada*/) NO ES UN caballo ;
```

El símbolo de la negación es la tilde de la ñ ('~'). Sirve para indicar que se espera una entrada que sea "cualquier entrada que NO satisfaga esta regla". Normalmente se utiliza en los analizadores léxicos. Por ejemplo, una regla simple para reconocer comentarios en el analizador léxico será[10]:

```
comentario : "/*"
             (~("*/"))*
             "*/" ;
```

Un comentario comienza con la cadena "/*", seguido de cero o más veces (el cierre de Kleene) cualquier cosa que no sea la cadena de terminación, "*/". Finalmente hay una cadena de terminación.

Estos son los operadores básicos de EBNF; ANTLR utiliza un dialecto de EBNF que añade algunos más; los iremos presentando conforme nos hagan falta.

Acciones

La sintaxis EBNF sirve para definir la parte "analítica" de los reconocedores. Bastan las reglas EBNF para comprobar la adecuación de una entrada.

No obstante, en la práctica, un reconocimiento se realiza con una intención práctica que va más allá de la comprobar que la entrada se adapta a unas reglas sintácticas. Por ejemplo, en muchas ocasiones deseamos traducir (a código máquina o a otra cosa) el código fuente. Para poder llevar a término la traducción, tanto BNF como EBNF introducen un nuevo tipo de elementos: las acciones.

Las acciones son pedazos de código nativo que deben ejecutarse al llegar a ellas. Están separadas por llaves del resto del código, así:

```
r : A {System.out.println("Ha llegado una A");}
  | B {System.out.println("Ha llegado una B");}
  ;
```

Si al analizador definido por la regla r se le pasa la entrada "ABBA", se obtendrá la siguiente salida por la consola:

10 Esta regla no tiene en cuenta los saltos de línea.

```
Ha llegado una A
Ha llegado una B
Ha llegado una B
Ha llegado una A
```

Aunque es lo usual, las acciones no tienen por qué estar al final de una regla; pueden estar mezcladas con los símbolos y nombres de reglas, para ser invocadas cuando el analizador las encuentre.

Tanto bison como ANTLR soportan reglas (aunque, como veremos, bison tiene algunos problemas de ambigüedad con ellas).

Una de las principales funciones de una acción es la de trabajar con información sintáctica que se asocia a los elementos de su regla (principalmente árboles AST).

En bison el trabajo con árboles dentro de las acciones es muy explícito. Primero se debe declarar qué tipo de información semántica guardará cada regla (utilizando `%type`). No se proporciona ningún tipo de implementación "por defecto" de los árboles. Para referenciar la información semántica de una regla en bison se utilizan los "comodines" del carácter dolar ($).

```
%type <ArbolSA> regla A B // ArbolSA debe ser implementado
...
regla: A B C {$$=f($1,$3);} ;
```

En el código anterior, escrito según la notación de bison, se está especificando que "el `ArbolSA` de la regla `r` se obtiene al aplicar la función `f` sobre los `ArbolSA` de `A` y `B`".

ANTLR es diferente en varios aspectos. Para empezar, la construcción de un árbol AST se realiza por defecto (se puede desactivar con la opción `buildAST` global o localmente a una regla). Además ANTLR proporciona una implementación por defecto de los árboles (que también puede cambiarse global o localmente), y una manera estandarizada de recorrerlos[11]. Cuando se desea referenciar el árbol de una regla, basta con utilizar una etiqueta de símbolo. La etiqueta de un símbolo aparece delante de él, uniéndose a él con el carácter de los dos puntos (':'). De esta forma, el código equivalente al anterior en ANTLR será el siguiente:

```
r: a:A B c:C {#r = f (#a,#c);} ;
```

En éste caso se han utilizado dos etiquetas, `a` y `c`. Como puede verse, la información de la regla también puede referenciarse utilizando la cabeza de la regla. Nótese que a veces es necesario colocar el carácter almohadilla delante de las referencias en la acción (hacerlo o no significa acceder a diferentes objetos; más sobre esto cuando sea necesario).

Otros

Además de las reglas básicas y las acciones, ANTLR permite utilizar predicados sintácticos y semánticos, devolver valores desde las reglas y pasarles parámetros. Presentaré los predicados sintácticos en más abajo, y el resto de las funcionalidades cuando sean necesarias.

1.3.2: "De arriba a abajo"; Analizadores recursivos descendentes

Adaptación del artículo "Exactly 1800 words about compilers" disponible en http://www.antlr.org.

Los algoritmos de análisis llamados "de arriba a abajo" son los algoritmos de análisis más intuitivos. Por ejemplo, supongamos que se desea codificar un analizador (sintáctico) para la siguiente gramática:

11 Para más información sobre construcción y recorrido de árboles de información sintáctica, véanse los capítulos 6 y 7.

```
expresion : factor ;
factor : termino ( "+" termino )* ;
termino : atomo ( "*" atomo )* ;
atomo : "(" expresion ")"
      | ENTERO
      | IDENDIFICADOR
      ;
```

Donde `ENTERO` e `IDENTIFICADOR` son abreviaturas que simbolizan respectivamente "cualquier entero" y "cualquier identificador", y devueltas adecuadamente en forma de tokens por el nivel léxico.

Supondremos que el análisis léxico ya está realizado, y nos devuelve un flujo de tokens con los espacios filtrados y con toda la información necesaria para enteros e identificadores. Cada token será devuelto por un método que llamaremos "siguienteToken", que devolverá los enteros `ENTERO, IDENTIFICADOR, PARENTAB, PARENTCER, PRODUCTO` y `SUMA.`

Como veremos, es mucho más sencillo realizar el análisis si el resultado de `siguienteToken` es almacenado en una variable, cada vez que un nuevo token es identificado. A esta variable la llamaremos "variable de lookahead", y como veremos tiene mucha importancia.

Tenemos que encontrar cómo codificar el análisis sintáctico. Tras pensarlo un poco, lo más lógico es codificar el reconocimiento de cada regla en una función independiente.

Por ejemplo, el método en java para analizar un átomo será parecido al siguiente:

```
public void atomo()
{
   switch ( tokenActual ) {
      case LPAREN : // -> "(" expr ")"
         tokenActual = siguienteToken();
         expresion();
         if ( TokenActual != PARENTAB ) error;
         tokenActual = siguienteToken();
      break;
      case ENTERO : // -> ENTERO
         tokenActual = siguienteToken();
      break;
      case IDENTIFICADOR :   // -> IDENTIFICADOR
         tokenActual = siguienteToken();
      break;
      default :
         error ("falta PARENTAB, ENTERO O IDENTIFICADOR");
      break;
   }
}
```

Los analizadores programados de esta forma se llaman *analizadores recursivos descendentes.*

*****

Los analizadores recursivos descendentes son un conjunto de métodos mutuamente recursivos (que se llaman unos a otros).

A la hora de determinar qué alternativa utilizar, nuestro analizador se basa en el siguiente símbolo que hay en la entrada. A dicho símbolo se le llama símbolo de *lookahead*. Dado que solamente necesita un símbolo, decimos que el lookahead es 1.

Los analizadores recursivos descendentes se llaman "de arriba a abajo" porque, si se mira el análisis que realizan de forma arbórea, empiezan con la "raíz" de dicho árbol y van "hacia abajo",

identificando estructuras cada vez más simples hasta llegar a los símbolos terminales, u "hojas del árbol".

Los analizadores recursivos descendentes entran también dentro de la categoría de LL(1), que significa que la entrada se recorre de derecha a izquierda utilizando un símbolo de lookahead.

Lookahead > 1

Los analizadores LL(1) deben poder predecir qué patrón coincidirá con la entrada utilizando únicamente el primer token que podría ser utilizado en cada alternativa. Al conjunto de los primeros tokens de cada alternativa de una regla se le llama PRIMERO(regla). Así, $PRIMERO(atomo)=\{PARENTAB\ ,\ ENTERO\ ,\ IDENTIFICADOR\}$.

Cuando los conjuntos PRIMERO de las diferentes alternativas de una regla no son disjuntas, se dice que hay una ambigüedad, o que la gramática no es determinista. Es decir, que al repetirse uno de los tokens al principio de más de una alternativa, no es posible saber cuál de las dos alternativas es la que se debe seguir utilizando un solo símbolo de lookahead.

Por ejemplo, la siguiente gramática es ambigua sobre el token A con un lookahead de 1:

```
a : A B C ;
a : A D E ;
```

Dado que las dos alternativas de la regla a comienzan con A, un analizador LL(1) no podría saber cual de las dos alternativas utilizar. Si el analizador pudiera "ver" el token de después de la A, entonces no habría ningún problema.

El concepto de LL(1) puede extenderse a LL(k), con k>1. Así, con un analizador LL(2) no tendría dificultades con la gramática anterior. k es la letra por antonomasia para hablar del lookahead.

Cuando para una gramática dada se puede definir un analizador LL(k) sin ambigüedades como la anterior, se dice que dicha gramática es LL(k). Un lenguaje para el que existe una gramática LL (k) también se llama LL(k).

Cuando el lookahead no es suficiente para reconocer una regla, existen diversos procedimientos para descomponerla. Algunos de ellos son la factorización de prefijos comunes y la eliminación de la recursión. El problema de estos procedimientos es que transforman la gramática de manera que la resultante es menos comprensible para los seres humanos.

Enriqueciendo el algoritmo: pred-LL(k)

Ya hemos visto los fundamentos teóricos más básicos de los analizadores LL(k). Esta visión debe completarse con lo que sabemos de la práctica.

Una de las primeras cosas que sabemos de la práctica es que en los lenguajes de programación hay un nivel semántico que viene después del sintáctico. Utilizando hábilmente la información semántica de un lenguaje podemos evitar incrementar k sin caer en ambigüedades. Considérese el siguiente ejemplo:

```
instruccion : bucle
            | asignacion
            | "break" PUNTO_COMA
            ;
asignacion : IDENT ASIG espresion PUNTO_COMA;
bucle : MIENTRAS PARENT_AB expresion PARENT_CE listaInstrucciones ;
listaInstrucciones : LLAVE_AB (instruccion)* LLAVE_CE ;
```

En este caso estamos diciendo que una instrucción puede ser un bucle, o una asignación o la palabra reservada "break", que solamente sirve para salir de un bucle.

El programador puede desear restringir el uso de `break` instrucciones utilizadas en los bucles, de manera que la palabra reservada solamente se pudiera utilizar dentro de un bucle, detectándose un error en caso contrario. Una primera opción sería utilizar reglas diferentes para las instruccines de dentro y fuera de los bucles, así:

```
instruccion : bucle
            | asignacion
            ;
instruccion_bucle : instruccion
                  | "break" PUNTO_COMA
                  ;
bucle : MIENTRAS PARENT_AB expresion PARENT_CE listaInstrucciones_bucle
      ;
listaInstrucciones_bucle : LLAVE_AB (instruccion_bucle)* LLAVE_CE ;
asignacion : IDENT ASIG espresion PUNTO_COMA;
```

Sin embargo ANTLR ofrece otra posibilidad: los *predicados semánticos*. Podemos utilizar una variable que indique si la instrucción que estamos reconociendo está dentro de un bucle o no. Supongamos que dicha variable se declara en la zona de código nativo del analizador como miembro de la clase. En tal caso podría implementarse así:

```
{
   int nivelBucle = 0 ; // break será válido si nivelBucle > 0
}

instruccion : bucle
            | asignacion
            | {nivelBucle>0}? "break" PUNTO_COMA
            ;

bucle : MIENTRAS PARENT_AB expresion PARENT_CE
        { nivelBucle++; }
        listaInstrucciones
        { nivelBucle--; }
      ;
```

Al algoritmo de análisis LL(k) que utiliza este tipo de predicados se le llama pred-LL(k).

Pred-LL(k) puede además utilizar otro tipo de predicados, que a pesar de haber dejado para el final no deben considerarse menos importantes: los *predicados sintácticos*.

Considérese la siguiente gramática:

```
a : (A)* B
  | (A)* C
  ;
```

Sea cual sea el lookahead empleado, el algoritmo LL(k) común siempre encontrará una ambigüedad (para una entrada en la que A se repita más de k veces, no podrá elegir entre la primera y la segunda regla). Se dice entonces que esta gramática es no-LL(k) para todo k.

Sin embargo planteémonos escribir la gramática de esta otra forma:

```
a : ( (A)* B )=> (A)* B
  | (A)* C
  ;
```

Lo que se ha añadido delante de la primera alternativa de la regla es un *predicado sintáctico*. Éste en concreto permite al analizador decidir algo que a nosotros nos parece obvio: que para elegir la primera opción. En otras palabras, *el predicado sintáctico está extendiendo el lookahead hasta donde hace falta* (en este caso indefinidamente).

1.3.3: "De abajo a arriba"; analizadores LR

Extraído del artículo "Exactly 1800 words about compilers" en http://www.antlr.org.

El algoritmo LR es un algoritmo de análisis "de abajo a arriba" (o *bottom-up*) debido al orden en el que analiza el flujo de símbolos de entrada: empieza por los símbolos "más terminales" de la gramática, que pueden verse como "hojas" del árbol de análisis, y va construyendo nodos cada vez más complejos hasta llegar a la regla raíz.

Una manera simple de ilustrar el algoritmo LR es considerando un lenguaje simple como el de la siguiente gramática:

```
a : A B C
  | A B D
  ;
```

Por comodidad vamos a reformularla de esta otra manera[12]:

```
a : A B C ;
a : A B D ;
```

Un analizador LR consta de una "pila de símbolos" en la que puede "apilar y desapilar" los símbolos que le llegan por la entrada y las "cabezas" de las reglas de la gramática que analiza. En nuestro caso sería capaz de apilar y desapilar A,B,C,D y a.

Básicamente, lo que hace el algoritmo LR es ir "consumiendo" símbolos de la entrada , intentando hacer corresponder los elementos apilados con alguna de las reglas de la gramática, para hacer una sustitución.

Como ejemplo, supongamos que la entrada que se le suministra al fichero es "ABD". El algoritmo transcurriría entonces de la siguiente manera:

1. El analizador "consume" A de la entrada.
2. Como A es el principio de las dos alternativas posibles, se consume el siguiente símbolo, que resulta ser B.
3. De nuevo estamos ante el mismo problema: las dos alternativas comienzan por AB, así que se

12 Esta sintaxis no es válida en ANTLR, pero sí en bison.

consume el siguiente símbolo de la entrada, es decir, D.

4. En este punto el analizador ya está en disposición de identificar la entrada actual con la segunda alternativa, así que elimina los tres símbolos (ABD) de la pila y en su lugar coloca una a.

Al proceso de "consumo" de un símbolo se le llama desplazamiento *(shift)*. Un desplazamiento consiste pues en retirar un símbolo de la entrada y apilarlo en la pila de símbolos.

Al proceso de "sustitución de unos símbolos por la parte derecha de una regla" se le denomina reducción *(reduce)*. Una reducción consiste en desapilar algunos símbolos de la pila y apilar la parte derecha de una regla.

*****

El lookahead

Existen gramáticas más complejas que la que hemos utilizado como ejemplo en las que no es posible averiguar qué regla se debe aplicar mirando únicamente el contenido de la pila; a veces es necesario tener en cuenta varios elementos de la entrada antes de poder hacer una reducción. Al número de elementos que hay que tener en cuenta se le llama también *lookahead*, y también se representa con la letra k. A las gramáticas LR con un lookahead k se les denomina LR(k).

Dado que en el ejemplo nos ha bastado con mirar los elementos de la pila para hacer las reducciones, podemos concluir que la gramática de ejemplo es LR(0).

Implementación

A un nivel muy básico podemos decir que tanto LR como LL pueden implementarse usando autómatas: en las dos implementaciones aparecen "estados", y para pasar de estado a estado, "transiciones". La diferencia está pues en cómo están codificados dichos estados y transiciones.

En LL es sencillo: cada estado es una función que representa una alternativa. Las transiciones a otros estados se solucionan con un simple "`switch`" (o una serie de "if-else-if"s anidados, si k>1). Cada regla EBNF se corresponde unívocamente a un estado del autómata, de manera que el número de estados del autómata no aumenta significativamente al crecer el lookahead (aunque la complejidad de las comprobaciones a realizar en cada autómata sí que aumenta exponencialmente con k). Las transiciones se representan con llamadas a métodos.

En oposición a LL, donde solamente es posible estar reconociendo una regla cada vez, la naturaleza del algoritmo LR y sus derivados hace posible que, en un instante determinado, una entrada pueda satisfacer varias reglas simultáneamente. Cada estado de un analizador LR contiene de esta manera varias reglas que "por el momento" son cumplidas por la entrada. El "grado de satisfacción" de cada regla se representa con un carácter especial, un punto ('.') que comienza estando al principio del cuerpo de la regla hasta llegar al extremo derecho, momento en el que se considera que la entrada satisface completamente la regla. Las transiciones entre estados son operaciones de consumición de elementos en la entrada.

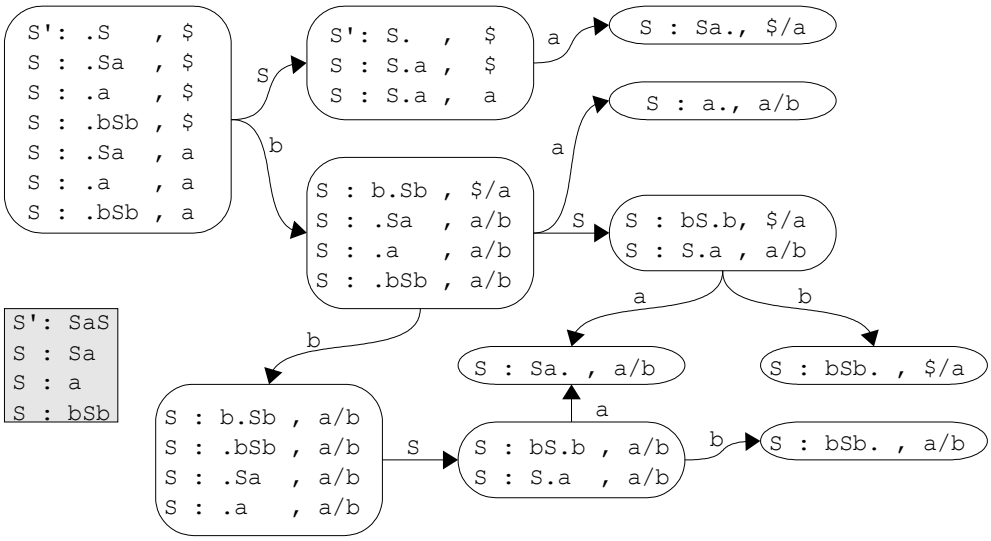

Ilustración 1.8 Autómata LR para una gramática de 4 reglas

En la figura anterior se muestra el autómata LR para un conjunto de 4 reglas simples (representadas en el cuadro). Nótese que el lookahead de los estados también es un elemento diferenciador: dos estados con las mismas reglas pero diferentes conjuntos de elementos en el lookahead se consideran dos estados diferentes.

El número de estados de un autómata LR es elevado: aumenta exponencialmente al crecer el número de reglas y al aumentar el lookahead.

La implementación más común del autómata LR es una tabla bidimensional en la que se codifican las transiciones, siendo los estados las filas de la tabla, y los símbolos las columnas. Si en la casilla en la que interseccionan el símbolo S y el estado E1 aparece una E2, eso quiere decir que si el analizador se encuentra en el estado E1 y en la entrada aparece el símbolo S, entonces se reduce y se pasa al estado E2. Una transición se implementa de una forma muy eficiente; se trata simplemente de cambiar el valor de un puntero.

Por esta implementación a los analizadores LR se les suele llamar *manejados por tablas (table-driven)*.

Las ambigüedades en un analizador LR ocurren cuando en un estado dado y para una entrada dada el analizador es capaz de hacer un desplazamiento o una reducción. A esto se le llama *conflicto shift-reduce*. Los conflictos shift-reduce no impiden que el analizador se genere; éste se genera, con el correspondiente mensaje de error, colocándose un comportamiento "por defecto" al analizador cuando se llega a ése estado y con esa entrada (el analizador *siempre desplaza o siempre reduce*). Si éste es el adecuado, la gramática funciona perfectamente, pero si se necesita el contrario estamos en un pequeño aprieto.

Bison utiliza una variante del algoritmo LR(1) llamada LALR(1), que reduce ostensiblemente el número de estados necesarios sin disminuir demasiado la potencia del análisis.

1.3.4: Comparación entre LR(k) y LL(k)

Potencia

Los analizadores LR(k) son más potentes que los analizadores **LL(k)** porque la estrategia LR utiliza un *contexto* más grande. El contexto para un analizador LR(k) son todas las reglas de la

gramática que concuerdan con la entrada que se está viendo. Intuitivamente: intenta hacer coincidir varias alternativas a la vez y pospone la toma de decisiones hasta que se haya visto suficiente entrada. Para un analizador LL el contexto es menor; está restringido a la secuencia de reglas que ya han sido analizadas y la posición dentro de la regla que se está analizando. Por lo tanto, los analizadores LL(k) dependen mucho del lookahead. Nótese que la gramática utilizada como ejemplo para el algoritmo LR es LR(0) y LL(3).

Por otro lado, los analizadores **pred-LL(k)** (no solamente LL(k), sino pred-LL(k)) son más potentes que los analizadores LR(k) por varias razones. Primera: los predicados semánticos pueden ser utilizados para analizar lenguajes dependientes del contexto. Segunda: los analizadores pred-LL(k) tienen lookahead ilimitado. Tercera: añadir acciones a una gramática LR puede producir ambigüedades, cosa que no ocurre con los analizadores LL(k).

Velocidad

Por lo general, un analizador LL(k) es algo más lento que su equivalente LR.

Al aumentar el lookahead, aumenta exponencialmente la complejidad del analizador. Y este aumento hace que el compilador sea cada vez más grande (lo que no importa demasiado) y más lento (y esto sí que importa). Con los ordenadores actuales se puede trabajar más o menos cómodamente con un lookahead de 3 o 4.

ANTLR usa una versión ampliada de LL(k) llamada pred-LL(K). El algoritmo pred-LL(k) fue presentado por primera vez por Terence Parr en su tesis de final de carrera. Lo primero que hizo Terence fue un estudio estadístico sobre diferentes gramáticas de 20 lenguajes de programación corrientes. Lo que descubrió fue que más del 98% de las reglas utilizadas necesitaban de un lookahead 1.

La idea que siguió al análisis es clara: Terence implementó un analizador con lookahead variable, que normalmente trabaja con un valor de k pequeño (por defecto 1), ampliándolo solamente en las reglas que lo requerían. Para ello añadió predicados (de ahí el sufijo “pred”.) especiales a las reglas que necesitaban un lookahead mayor. Esto aumentó enormemente la potencia del algoritmo, pudiéndose leer algunas gramáticas no-LL(k) para ningún k.

Sin embargo el algoritmo seguía siendo demasiado lento: En cada estado pred-LL(k) es necesario efectuar comparaciones con m^k elementos, siendo m el número de símbolos que podían seguir a la regla en la entrada (el conjunto SIGUIENTE de la regla).

Terence diseñó una nueva manera de calcular el conjunto SIGUIENTE de manera que el número de comprobaciones a realizar crecía de forma lineal al aumentar k, en lugar de de forma exponencial. Es decir, el número de comprobaciones a realizar se hizo m*k. Esta transformación del lookahead tiene el inconveniente de que hace que el analizador no sea LL(k) puro, es decir, que no reconoce todas las gramáticas LL(k). Al analizador LL(k) que utiliza el lookahead modificado de Terence se le llama Strong-LL(k), o SLL(k). Por lo tanto, el algoritmo que utiliza antlr debería llamarse pred-SLL(k). Hemos obviado este hecho porque las gramáticas de los lenguajes de programación son en su gran mayoría SLL(k).

Analicemos ahora el algoritmo que utiliza bison. Los ordenadores que existían cuando fue creado yacc, el predecesor de bison, no eran capaces de “aguantar” ni siquiera el número de estados de una gramática LR(1). Por lo tanto para implementar yacc se utilizó un derivado de LR(1) llamado LALR(1). LALR(1) utiliza menos estados que LR(1), y por lo tanto es menos potente. No obstante, las gramáticas de los lenguajes de programación suelen ser LALR.

De esta forma los analizadores basados en bison obtienen una gran rapidez de transición (cambios en un puntero sobre una table) y un número relativamente bajo de estados, al utilizarse

como algoritmo de análisis LALR y estar fijado el lookahead a 1[13].

Como media se observa un incremento de alrededor 20% en el tiempo de compilación de un compilador LL frente a su equivalente LR, disminuyendo esta diferencia al aumentar el número de estados.

Sin embargo esta medición es bastante teórica: supone un análisis sin acciones. Como señalan los creadores de LUA[14], la complejidad de las acciones también ha de ser tenida en cuenta a la hora de evaluar la velocidad. LUA es un proyecto que comenzó siendo desarrollado con bison y flex, pero que al cabo del tiempo se decidió utilizar autómatas recursivos descendientes hechos a mano (muy similares a los que genera ANTLR). Una de las razones que dieron para ello es el manejo de atributos heredados y sintetizados: las peculiaridades del algoritmo LR les obligaron a utilizar una pila de datos, en la que apilaban la información que debían pasar a las sub reglas. Utilizando ANTLR se prescinde de dicha pila: las variables locales, parámetros y valores devueltos por las reglas la hacen completamente innecesaria.

En casos como el anterior una transición entre estados LL puede ser incluso más rápida que una entre estados LR: mientras que LR se utiliza una pila "hecha a mano", LL usa la propia pila de ejecución del programa, que suele estar más optimizada.

Facilidad de uso

Las características de los algoritmos LR(k) y pred-LL(k) hacen que la forma de trabajar con ellos sea diferente.

Un analizador LR(k) oculta más información de cómo actúa al programador; éste simplemente se limita a escribir las reglas, y el programa funciona. El programador no necesita conocer el algoritmo LR(k) para escribir una gramática. Ocasionalmente el generador de analizadores comunicará conflictos de shift-reduce, pero por regla general se pueden ignorar sin mayor problema: la acción por defecto del generador suele ser la correcta. De esta forma es muy normal que un compilador LR tenga varios conflictos shift-reduce, sobre todo si se trata de un compilador LALR(1).

Un analizador pred-LL(k) requiere, sin embargo, un poco más de atención; en cuanto el programador escriba dos producciones con el mismo conjunto PRIMERO el analizador se negará a compilar; será necesario incrementar el lookahead, bien subiendo k o bien añadiendo predicados sintácticos a dichas reglas. Ésto hace que escribir con ANTLR sea (un poco) más laborioso que hacerlo con bison. De todas formas hay que tener en cuenta que en más del 98% de las ocasiones no necesitaremos aumentar k.

Además, arreglar estos conflictos es muchos más fácil que arreglar un conflicto shift-reduce de un compilador LR. Como un analizador LL no compila si tiene conflictos, es más "seguro" que un analizador LR (en el que es posible generar un analizador con conflictos).

Otro punto interesante de la facilidad de uso son las acciones: en LR, las acciones introducen nuevos estados en la tabla de estados que pueden añadir incoherencias. En otras palabras, un reconocedor LR sin conflictos puede dejar de funcionar correctamente al añadirle acciones. Los reconocedores LL, sin embargo, son impermeables a éste tipo de problemas.

Finalmente, el código generado de un autómata recursivo descendente es fácilmente legible por un humano, al corresponderse cada regla con un estado. Uno puede hacerse una idea muy exacta de lo que está haciendo el analizador en cada instante, porque genera un código parecido al que una persona generaría al implementar el compilador a mano.

13 Un lookahead tan pequeño tiene el inconveniente de la pérdida de potencia: pred-SLL(k) reconoce más lenguajes que LALR(1)
14 LUA es un lenguaje embebible y libre que se desarrolla con ANTLR. Su sitio oficial es www.lua.org

Sección 1.4: Conclusión

En este capítulo he explicado algunos conceptos sobre compiladores. No era mi intención hacerlo de una manera rigurosa o formal; muchas cosas han sido sobreentendidas o directamente ignoradas. Una disertación de este tamaño se saldría completamente de los propósitos de este documento. Si el lector desea ampliar sus conocimientos en el tema le recomiendo que lea otros textos más especializados (puede encontrar algunos en la bibliografía.)

En el capítulo siguiente voy a dejar de lado la "visión general sobre los compiladores" que he tenido en este capítulo y voy a concentrarme en ANTLR, explicando cómo se trabaja con él.

Capítulo 2: Presentación de ANTLR

"La mayoría de las ideas fundamentales de la ciencia son esencialmente sencillas y, por regla general pueden ser expresadas en un lenguaje comprensible para todos"
Albert Einstein

Capítulo 2: Presentación de ANTLR.....29

Sección 2.1: Introducción.....31
2.1.1: ¿Qué es y cómo funciona ANTLR?..... 31
2.1.2: Propósito y estructura de éste capítulo..... 31
Sección 2.2: Los analizadores en ANTLR.....33
2.2.1: Especificación de gramáticas con ANTLR..... 33
2.2.2: La zona de código nativo..... 34
Sección 2.3: Los flujos de información.....35
2.3.1: Flujo de caracteres..... 35
2.3.2: Flujo de Tokens..... 35
La clase antlr.Token.....36
La clase antlr.CommonToken.....36
La interfaz antlr.TokenStream.....37
Construyendo y ejecutando el analizador.....37
2.3.3: Los ASTs..... 38
Situación de los ASTs.....38
Notación.....39
Nodos del AST: La interfaz antlr.collections.AST.....39
Las clases antlr.BaseAST y antlr.CommonAST.....41
Construcción y obtención de ASTs en un analizador.....42
Sección 2.4: Reglas EBNF extendidas.....44
2.4.1: Introducción..... 44
2.4.2: Declarando variables locales en una regla..... 44
2.4.3: Utilizando las etiquetas..... 46
2.4.4: Pasando parámetros a una regla..... 46
2.4.5: Devolviendo valores en una regla..... 47
2.4.6: Utilizando rangos de caracteres en el analizador léxico..... 48
2.4.7: Utilizando patrones árbol en el analizador semántico..... 48
Sección 2.5: Construcción de los ASTs.....50
2.5.1: Comportamiento por defecto..... 50
2.5.2: El sufijo de enraizamiento (^)..... 51
2.5.3: Sufijo de filtrado (!)..... 52
2.5.4: Desactivando la construcción por defecto..... 53
2.5.5: Construyendo el AST en una acción..... 53
2.5.6: Casos en los que la construcción por defecto no basta..... 55
Sección 2.6: Analizadores de la Microcalculadora.....56
2.6.1: El fichero MicroCalc.g y el paquete microcalc..... 56
2.6.2: El analizador léxico..... 56
2.6.3: El analizador sintáctico..... 57
2.6.4: Nivel semántico..... 58
2.6.5: El fichero MicroCalc.g..... 59
2.6.6: Generando los analizadores de MicroCalc..... 61
Sección 2.7: Ejecutando Microcalc.....62
2.7.1: La clase Calc..... 62
2.7.2: Clase Calc refinada..... 63
2.7.3: Utilizando microcalc..... 65
Sección 2.8: Conclusión.....67

Sección 2.1: Introducción

2.1.1: ¿Qué es y cómo funciona ANTLR?

ANTLR es un programa está escrito en java, por lo que se necesita alguna máquina virtual de java para poder ejecutarlo. Es software libre, lo que quiere decir que al descargarlo de la página oficial (**http://www.antlr.org**) obtendremos tanto los ficheros compilados `*.class` como el código fuente en forma de ficheros `*.java`.

ANTLR es un *generador de analizadores*. Mucha gente llama a estas herramientas *compiladores de compiladores*[15], dado que ayudar a implementar compiladores es su uso más popular. Sin embargo tienen otros usos. ANTLR, por ejemplo, podría servir para implementar el intérprete de un fichero de configuración.

ANTLR es capaz de generar un analizador léxico, sintáctico o semántico en varios lenguajes (java, C++ y C# en su versión 2.7.2) a partir de unos ficheros escritos en un lenguaje propio. Dicho lenguaje es básicamente una serie de reglas EBNF y un conjunto de construcciones auxiliares.

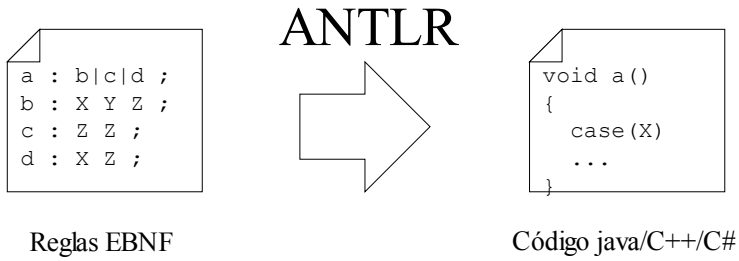

Ilustración 2.1 Funcionamiento de ANTLR

ANTLR genera analizadores pred-LL(k), y él mismo utiliza un analizador pred-LL(k) para leer los ficheros en los que están escritas las reglas EBNF. ANTLR admite acciones en sus reglas, además de otras prestaciones como paso de parámetros, devolución de valores o herencia de gramáticas.

2.1.2: Propósito y estructura de éste capítulo

El propósito de este capítulo es ilustrar el funcionamiento de ANTLR implementando un *intérprete* (que no un compilador) para un lenguaje muy sencillo: el de una calculadora simbólica.

La calculadora será capaz de realizar operaciones aritméticas simples (suma, resta, multiplicación y división) con números enteros o con decimales, e imprimirá el resultado por pantalla. Los cálculos los introducirá el usuario por la consola, y la calculadora los presentará por pantalla. La precedencia de los operadores podrá cambiarse mediante el uso de paréntesis.

Podemos definir dos grandes bloques en este capítulo. Las secciones 2.2, 2.3, 2.4 y 2.5 son más bien teóricas. Preparan el terreno para el segundo bloque, formado por las secciones 2.6 y 2.7, en las que utilizamos lo aprendido en el primer bloque para implementar rápidamente la microcalculadora.

Hay dos maneras de leer este capítulo. Cada uno debe decidir cuál utilizar dependiendo de qué tipo de lector sea.

- "El corredor de fondo": Consiste en ir leyendo todas las secciones, una tras otra, hasta el final.

15 Ver el capítulo anterior para una descripción más detallada de los compiladores de compiladores.

Requiere una gran resistencia, pues el primer bloque de teoría es bastante denso.

- "El aventurero": Es el que empieza leyendo por la sección 2.6, intentando entenderlo todo basándose en su sentido común. Utilizará el primer bloque de secciones como consulta cuando se quede atascado.

Aún se me ocurre un tercer tipo:

- "El corredor arrepentido": Es el que empieza en el orden indicado, pero antes de terminar con la teoría, exhausto, decide comenzar a leer la sección 2.6, y a partir de ahí va "intercalando" teoría y práctica.

Estos son los tipos de lector que se me han ocurrido, pero seguro que hay más. No importa el orden; lo que importa es poder leer todo el capítulo. Que cada cual lo haga como mejor pueda. Recuérdese que más adelante se van a emplear los conceptos vistos aquí.

Sección 2.2: Los analizadores en ANTLR

Ya mencionamos que ANTLR genera sus analizadores a partir de unos ficheros de entrada en los que se especifica el funcionamiento de dichos analizadores mediante el uso de reglas EBNF. Para poder comprender los analizadores necesitaremos conocer la sintaxis de dichos ficheros de entrada.

2.2.1: Especificación de gramáticas con ANTLR

Los ficheros con los que trabaja ANTLR tienen la terminación *.g, y en adelante los llamaremos *ficheros de especificación de gramáticas* o, directamente, *ficheros de gramáticas*.

Un fichero de gramática contiene la definición de uno o varios analizadores. Cada uno de estos analizadores se traducirá a código nativo (java, C++ o C#, dependiendo de ciertas opciones) en forma de clases. Es decir, por cada analizador descrito en el fichero de gramáticas se generará una clase.

Todo fichero de gramática tiene la siguiente estructura:

```
header{
   /* opciones de cabecera */
}
options
{
/* opciones generales a todo el fichero */
}

// A continuación la definición de el(los) analizadore(s).
```

Cabecera: Esta zona es opcional (puede aparecer o no). Delimitada por las partículas "`header` {" y "}", en esta zona incluimos elementos en código nativo (java, C++ o C#) que deben preceder a la definición de las diferentes clases de los analizadores. Esta sección se utiliza para incluir otros ficheros (`import` e #include), definir el paquete al que pertenecerá la clase del analizador (`package`) etc.

- Opciones generales del fichero: Esta zona es opcional. Permite controlar algunos parámetros de ANTLR mediante "opciones". Las opciones se representan como asignaciones : `nombreOpcion=valor;`. Se utilizan mucho en ANTLR. Las opción más importante de esta zona es la que permite elegir el lenguaje nativo en el que se generarán los analizadores (java,C++,C#). Su valor por defecto es "java". Dado que vamos a generar reconocedores en java, no necesitaremos esta zona. En el manual de ANTLR aparecen todas las opciones que se pueden incluir en esta zona.

Tras las opciones generales del fichero vienen las definiciones de analizadores. Es muy común que en un mismo fichero se especifiquen varios analizadores (en la mayoría de los ejemplos que acompañan a ANTLR se utiliza esta técnica). Sin embargo también es posible definir cada analizador en un fichero, sobre todo cuando se trata de analizadores extensos[16].

Dado que nuestro lenguaje en este capítulo será muy sencillo, definiremos los tres analizadores en el mismo fichero.

En ANTLR, cada analizador tiene la siguiente estructura:

```
class nombreAnalizador extends tipoAnalizador; // definición del analizador
```

16 Utilizar varios ficheros requiere un trabajo especial (importar y exportar vocabularios). Más información sobre esto en los capítulos 5 y 6.

```
options {
   /* Zona de opciones del analizador*/
}
tokens {
   /* Zona de definición de tokens */
}
{
   /* Zona de código nativo */
}
/* Zona de reglas */
```

Definición del analizador: En la primera línea definimos el nombre del analizador (`nombreAnalizador`) y su tipo (`tipoAnalizador`). El tipo puede será `Lexer` para analizadores léxicos, `Parser` para analizadores sintácticos y `TreeParser` para analizadores semánticos.

Zona de opciones: Esta zona es opcional, aunque casi siempre interesa utilizarla. En esta zona se definen propiedades muy importantes del analizador: se define el lookahead (k), si se va a generar un AST o no, en el caso de Parsers y TreeParsers, la importación/exportación de vocabulario, la activación/desactivación del tratamiento automático de errores etc. Para más información consúltese el manual de ANTLR.

Zona de definición de tokens: Esta zona es opcional. Permite definir nuevos tokens, que se añaden a los que se hayan importado en la zona de opciones del analizador.

Zona de código nativo: (Ver más abajo)

Zona de definición de reglas: En esta zona se encontrarán las reglas que definirán la gramática. Se admiten reglas EBNF extendidas (para más información sobre las reglas EBNF extendidas, véase la sección dedicada a ellas en este mismo capítulo).

2.2.2: La zona de código nativo

No hay que olvidar que para ANTLR cualquier analizador es una instancia de una clase. En ocasiones es muy útil declarar métodos y atributos para dicha clase.

Para añadir métodos y variables a una clase de un analizador basta con escribirlos, entre la zona de opciones y la zona de reglas, *entre llaves*. Por ejemplo,

```
class MyParser extends Parser; // definición de un analizador sintáctico
options {...}
tokens {...}
{
  private String a= "Hola "; // atributo privado

  public void imprimir(String s) // método público
  { System.out.println(s); }
}

regla1 : i:IDENT { imprimir(i.getText()); } ;
regla2 : e:ENTERO { imprimir(a+e.getText()); } ;
```

En este ejemplo hemos añadido un atributo y un método a la clase. Podríamos haber añadido otros. También podríamos haber incluido uno o varios constructores. Como puede verse, estos elementos son directamente utilizables en las acciones de las reglas.

Sección 2.3: Los flujos de información

Ya sabemos qué tipos de analizadores hay (léxico, sintáctico o semántico). Pero ¿cómo se transmite la información entre los diferentes niveles? En esta sección responderemos a esa pregunta.

Terence Parr ha organizado la información de una manera muy interesante que merece ser explicada.

(Como siempre, me voy a referir a la versión en Java de los analizadores).

2.3.1: Flujo de caracteres

El primer flujo que voy a explicar es el que modela la entrada de datos desde el exterior hasta la primera fase del compilador, es decir, al analizador léxico. En otras palabras, voy a explicar qué es para ANTLR el flujo que en el primer capítulo llamé "Caracteres".

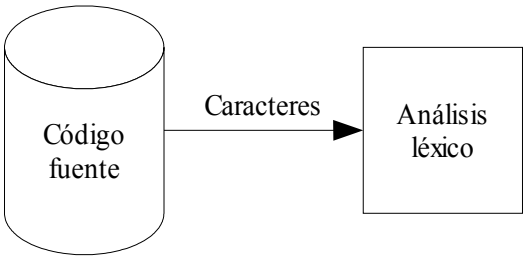

Ilustración 2.2 Primer flujo de información: caracteres

Para ANTLR el flujo caracteres es, simplemente "cualquier subclase de `java.io.InputStream`". Lo más normal es utilizar un flujo de caracteres provinentes de un fichero (con un `FileInputStream`) pero pueden utilizarse otras fuentes, como una cadena (`StringBufferStream`) o una página web (`URL.openStream`).

El `InputStream` que proporciona los caracteres al analizador léxico se le pasa como parámetro en su constructor.

Un pequeño inconveniente de utilizar éste método es que se pierde el concepto de "fichero"; los datos del fichero de texto perduran en forma de flujo `FileInputStream`, *pero no ocurre así con el nombre del fichero*. Éste puede ser especificado con la función `setFilename`:

```
// Convertir el nombre de fichero en un flujo
FileInputStream fis = newFileInputStream(fileName);

// Crear el lexer utilizando dicho flujo
LeLiLexer lexer = new LeLiLexer(fis);

// Proporcionar además el nombre de fichero al analizador
lexer.setFilename(f);
```

2.3.2: Flujo de Tokens

En éste caso estamos hablando del flujo de información existente entre el nivel léxico y sintáctico, es decir, el flujo que anteriormente hemos llamado "Tokens".

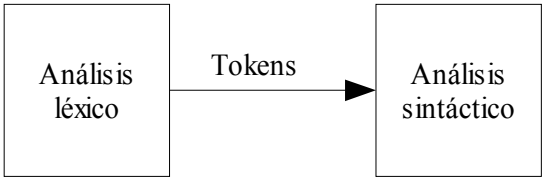

Ilustración 2.3 Flujo de tokens entre el lexer y el parser.

La clase antlr.Token

Los tokens se representan en ANTLR utilizando una clase llamada `antlr.Token`. He aquí el código íntegro de dicha clase (salvo por algunos comentarios):

```
public class Token {
  // constants
  public static final int MIN_USER_TYPE = 3;
  public static final int INVALID_TYPE = 0;
  public static final int EOF_TYPE = 1;
  public static final int SKIP = -1;

  // each Token has at least a token type
  int type=INVALID_TYPE;

  // the illegal token object
  public static Token badToken =
    new Token(INVALID_TYPE, "");

  public Token() {;}
  public Token(int t) { type = t; }
  public Token(int t, String txt) {
    type = t; setText(txt);
  }

  public void setType(int t) { type = t; }
  public void setLine(int l) {;}
  public void setColumn(int c) {;}
  public void setText(String t) {;}

  public int getType() { return type; }
  public int getLine() { return 0; }
  public int getColumn() { return 0; }
  public String getText() {...}
}
```

Esta clase está "casi vacía"; no hay una verdadera implementación de los métodos – están ahí para ser sobreescritos por una subclase.

A pesar de ser casi una interfaz, esta clase nos da una idea de cómo se tratan los tokens en ANTLR: un token es un "tipo" (un entero) un texto (una cadena) y una línea y columna (sendos enteros). Esta clase base garantiza que siempre que utilicemos un token en ANTLR podremos obtener estas informaciones.

La clase antlr.CommonToken

La clase `antlr.Token` por sí misma no es muy práctica debido a sus "métodos vacíos". El

analizador sintáctico no la utilizará directamente para representar los tokens; en su lugar se utiliza una subclase de `antlr.Token` llamada `antlr.CommonToken`. Su código es el siguiente:

```
public class CommonToken extends Token {
  // most tokens will want line, text information
  int line;
  String text = null;

  public CommonToken() {}
  public CommonToken(String s)  { text = s; }
  public CommonToken(int t, String txt) {
    type = t;
    setText(txt);
  }

  public void setLine(int l)     { line = l; }
  public int  getLine()          { return line; }
  public void setText(String s) { text = s; }
  public String getText()        { return text; }
}
```

Esta clase no hace más que rellenar los "huecos" que faltan en su superclase.

Uno de los "huecos" que `CommonToken` no implementa es la conservación del nombre de fichero (filename) en el token (`CommonToken` no tiene ningún atributo llamado "filename", y los métodos `getFilename` y `setFilename` no hacen nada o devuelven null).

Más adelante en el capítulo de análisis léxico veremos cómo solucionar esto.

La interfaz antlr.TokenStream

La clase `antlr.CharScanner` (de la que heredan los analizadores léxicos que hacemos en ANTLR) implementa la interfaz `antlr.TokenStream`. El código completo de esta interfaz es el siguiente:

```
package antlr;

/* ANTLR Translator Generator
 * Project led by Terence Parr at http://www.jGuru.com
 * Software rights: http://www.antlr.org/RIGHTS.html
 *
 * $Id: //depot/code/org.antlr/main/main/antlr/TokenStream.java
 */

public interface TokenStream {
    public Token nextToken() throws TokenStreamException;
}
```

Así que lo único que tiene que hacer el analizador sintáctico es ir llamando al método `nextToken` del analizador léxico.

Construyendo y ejecutando el analizador

Por eso el constructor principal de cualquier analizador sintáctico generado por ANTLR toma como único parámetro un objeto que cumpla la interfaz `antlr.TokenStream`. Por ejemplo, nuestro analizador léxico:

```
// Crear un analizador sintáctico utilizando el léxico
LeLiParser parser = new LeLiParser(lexer);
```

Como ocurría anteriormente, el nombre del fichero es irrecuperable por el analizador, con lo que de nuevo es necesario pasárselo al analizador:

```
// Proporcionar el nombre del fichero al analizador sintáctico
parser.setFilename(fileName);
```

Una consecuencia muy interesante de esta arquitectura es que podemos utilizar cualquier clase que implemente la interfaz `antlr.TokenStream` como analizador léxico; si no nos gusta la implementación con autómatas LL del analizador léxico, podemos utilizar nuestra propia implementación. Bastará que cumpla la interfaz para poder pasársela al analizador sintáctico en la construcción.

Para que un analizador sintáctico comience a analizar una entrada, es necesario llamar explícitamente al método de la primera clase que se ha declarado. A dicha clase también se le llama "regla inicial" o "regla raíz".

Si la regla raíz de nuestro compilador se llama "programa", entonces para iniciar el análisis sintáctico sobre una entrada será necesario escribir algo así:

```
parser.programa();
```

En ese momento el analizador sintáctico comenzará a pedir tokens al analizador léxico, haciendo que éste a su vez comience a consumir caracteres de la entrada.

2.3.3: Los ASTs

Situación de los ASTs

Los ASTs (*Abstract Syntax Trees*, o Árboles de Sintaxis Abstracta) sirven para manejar la información semántica de un código. La forma más eficiente de manejar la información provinente de un lenguaje de programación es la forma arbórea; por éso la estructura de datos elegida es un árbol. Además, construyendo ASTs a partir de un texto podemos obviar mucha información irrelevante; si un AST se construye bien, no habrá que tratar con símbolos de puntuación o azúcar sintáctica en el nivel semántico.

Al contrario que los flujos, una estructura en árbol puede especificar la relación jerárquica entre los símbolos de una gramática.

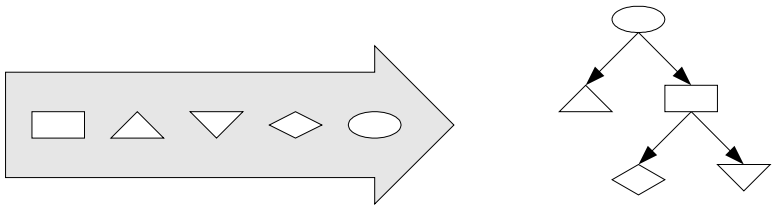

Ilustración 2.4 Flujo (izquierda) y árbol (derecha) de información

Los ASTs pueden intervenir en varias fases del análisis: como producto del análisis sintáctico, como elemento intermedio en sucesivos análisis semánticos y como entrada para la generación de código. Podemos visualizarlo gráficamente de la siguiente manera:

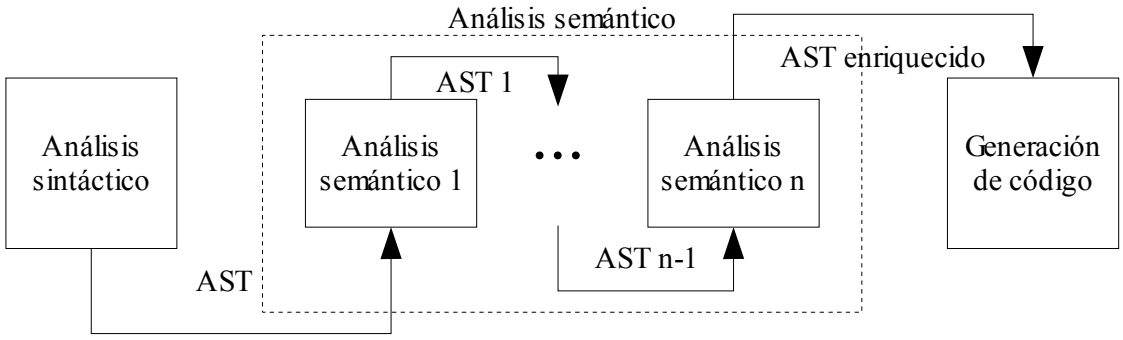

Ilustración 2.5 Intercambio de ASTs entre las diferentes partes de un compilador

Notación

Traducido del manual en inglés de ANTLR. Fichero trees.htm. Muchas modificaciones.

La forma normal de representar un árbol en ANTLR utiliza una notación basada en LISP. Por ejemplo:

```
#(A B C)
```

Es un árbol con **"A" en la raíz**, y los hijos B y C.

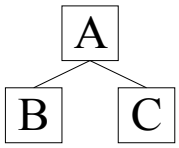

Ilustración 2.6 #(A B C)

El elemento que hace de raíz en el árbol #(X Y Z) es el que aparece primero, es decir, "X".

Esta notación puede anidarse para describir árboles de estructuras más complejas. Así:

```
#(A B #(C D E))
```

Representa el siguiente árbol:

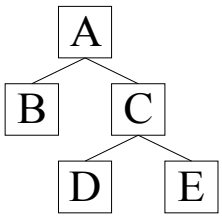

Ilustración 2.7 #(A B #(C D E))

*****

Nodos del AST: La interfaz `antlr.collections.AST`

Ya he establecido que un AST almacena la información semántica en una estructura arbórea. Queda por aclarar *cómo* se almacena dicha información.

Para ANTLR, un árbol AST es cualquier clases que implemente la interfaz `antlr.collections.AST`. Voy a explicarla brevemente. Por comodidad me referiré a los métodos de la interfaz como si ya estuvieran implementados, es decir, en lugar de escribir "este método deberá hacer esto" escribiré "este método hace ésto". Pero téngase en cuenta que es una manera de hablar.

Una aclaración previa: la interfaz es *minimal*, y por ésa razón solamente almacena dos datos en cada nodo: un texto y un entero, que se utilizarán probablemente para representar un token. No obstante el usuario es libre de añadir más datos en cada nodo ampliando la interfaz.

Paso a explicar el fichero en el que se define `antlr.collections.AST`, línea a línea.

```
package antlr.collections;

import antlr.Token;

public interface AST {

   public void  addChild(AST c);
```

Tras las declaraciones de paquete e importaciones y de la interfaz, encontramos el primer método, `addChild`. Éste sirve para "añadir un hijo por el final de la lista de hijos (por la derecha)".

```
   public boolean equals(AST t);
   public boolean equalsList(AST t);
   public boolean equalsListPartial(AST t);
   public boolean equalsTree(AST t);
   public boolean equalsTreePartial(AST t);
```

Éste conjunto de métodos sirven para comparar árboles AST entre sí.

- El primero de ellos (`equals`) no es "recursivo"; hace una comparación a nivel de texto del nodo (`toString`) y a nivel de tipo de información entre los dos.
- El segundo (`equalsList`) los compara, además, estructuralmente (recursivamente). Incluye hermanos e hijos.
- El tercero (`equalsListPartial`) comprueba si t es una versión aumentada del árbol que llama al método. Es decir, si t es el AST llamador con algunas árboles más colgando de algunas hojas.
- El cuarto (`equalsTree`) es como equalsList salvo que supone que el árbol tiene una única raíz (no es degenerado). Es decir, ignora los hermanos.
- El último método (`equalsTreePartial`) hace un `equalsListPartial(t)` sobre el árbol suponiendo que es la raíz del árbol (ignorando los hermanos).

```
   public ASTEnumeration findAll(AST tree);
   public ASTEnumeration findAllPartial(AST subtree);
```

Estos dos métodos implementan búsquedas. El primero devuelve todos los subárboles que son `equalsList(tree)` mientras que el segundo devuelve los que son `equalsListPartial(tree)`.

```
   public AST getFirstChild();
   public AST getNextSibling();
```

Estos dos métodos permiten la "navegación" por el árbol. `getFirstChild` devolverá el primer hijo del AST, mientras que `getNextSibling` devolverá el "hermano" (el siguiente hijo del padre

del nodo) del árbol. Ambos métodos devuelven `null` si no hay hijos/hermanos. Nótese que la interfaz solamente prevee la navegación "de arriba a abajo" y de "izquierda a derecha".

```
public String getText();
public int getType();
```

Estos dos métodos sirven para obtener la única información estrictamente necesaria para los ASTs: un entero (tipo) y una cadena (texto). En los nodos ASTs como mínimo hay un texto y un tipo de nodo. Por defecto estas informaciones se obtendrán de Tokens. No obstante, podemos hacer que se obtengan de cualquier forma, siempre y cuando implementemos `getText` y `getType`.

Esta información es importante: todo nodo AST tiene (al menos) un tipo y un texto, y se obtienen con los métodos `getType` y `getText`. Es muy común iniciar dichos valores con un `Token`(que también posee un tipo y un texto). Pero hay que tener presente que las dos clases son diferentes: un AST no tiene por qué contener un Token; un AST puede tener un tipo o un texto diferente a los de cualquier token.

```
public int getNumberOfChildren();
```

Devuelve el número de hijos del AST.

```
public void  initialize(int t, String txt);
public void  initialize(AST t);
public void  initialize(Token t);
```

Diversos métodos de inicialización (ya que en una interfaz no pueden especificarse constructores obligatorios, se ha optado por esta solución).

```
public void  setFirstChild(AST c);
public void  setNextSibling(AST n);
public void  setText(String text);
public void  setType(int ttype);
```

Métodos de modificación del primer hijo, el hermano por la derecha, el texto y el tipo del Token de la raíz.

```
public String toString();
public String toStringList();
public String toStringTree();
}
```

Métodos para convertir a cadenas. El primero de ellos no es recursivo, devolviendo solamente información sobre el token de la raíz. El segundo obtiene recursivamente una representación en una sola línea todo el árbol, utilizando una sintaxis muy parecida a la del lenguaje LISP. El último método es idéntico al anterior, pero solamente tiene en cuenta el primer hijo del AST.

En resumen, la interfaz proporciona métodos para añadir hijos, buscar, convertir a cadenas y comparar. Además están los métodos `getText()` y `getType()`, para obtener el texto y tipo de un nodo.

Las clases antlr.BaseAST y antlr.CommonAST

Ya he dicho que ANTLR permite utilizar como árbol cualquier clase que implemente adecuademante `antlr.Collections.AST`. La clase que se utiliza para generar el AST en el analizador sintáctico se puede especificar mediante la opción `ASTLabelType` en la sección `options` del analizador. El valor por defecto de `ASTLabelType` es `antlr.CommonAST`.

`antlr.CommonAST` es la implementación por defecto que ANTLR ofrece para los ASTs. Esta clase no se limita a implementar completamente `antlr.Collections.AST`, sino que añade mucha funcionalidad extra (como serialización xml o codificación para transmitir por internet). Sin embargo no es en `antlr.CommonAST` donde esta funcionalidad se implementa, sino en suclase padre, `antlr.BaseAST`. En esta clase abstracta se implementan todos los métodos de `antlr.Collections.AST` excepto los relacionados con la información contenida en las raíces (que tienen una implementación por defecto en `antlr.CommonAST`).

Si bien es posible utilizar una clase hecha desde cero que implemente `antlr.Collections.AST` para los compiladores, lo más usual es utilizar directamente `antlr.CommonAST` o bien utilizar una subclase de `antlr.BaseAST`.

Uno de los inconvenientes de utilizar las implementaciones de ASTs proporcionadas por ANTLR es que la visión que se tiene del árbol es muy reducida; cada nodo solamente tiene constancia de su primer hijo y de su "hermano". Es decir, se trabaja con un árbol como el de la izquierda en la siguiente figura:

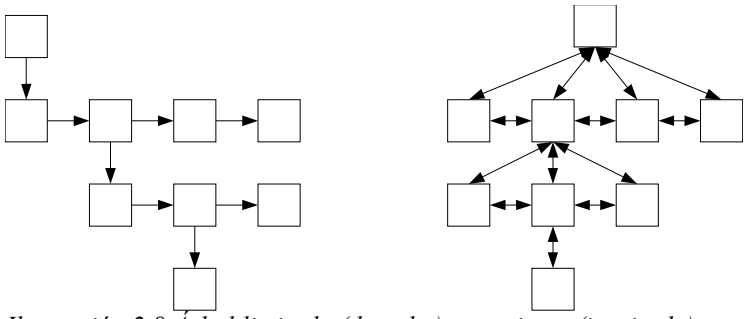

Ilustración 2.8 Árbol limitado (derecha) y corriente (izquierda)

Mientras que lo usual es tener más referencias, de manera que cada nodo conoce a *todos* sus hijos y además a su padre y sus hermanos, como ocurre en el de la izquierda. En los comentarios de `antlr.BaseAST` se hace referencia a que "recorrer los árboles es más fácil así si tienen esta estructura", aunque pienso que con una estructura con más enlaces no se habría perdido ninguna facilidad.

De todas maneras, en la mayoría de las operaciones que se realicen con árboles basta con un recorrido abajo/derecha. Sólamente he descrito su funcionamiento interno porque es conveniente tenerlo en cuenta a la hora de trabajar con ASTs. Por ejemplo:

- Si a un árbol se le van a insertar sucesivamente varios hijos, es más eficiente guardar el último y llamar a `setNextSibling` que utilizar `addChild` en el padre.
- Cuando un AST "A" se hace hijo de otro AST "B" (`B.addChild(A)`) todos los hermanos de A se harán también hijos de B.
- Cuando un AST "A" se fija como primer hijo de otro AST "B" (`B.setFirstChild(A)`) todos los hijos que B tuviera previamente se perderán (siendo sustituidos por los hermanos de A)

Construcción y obtención de ASTs en un analizador

Para que un analizador léxico (o semántico) genere un AST a partir de la entrada, es necesario que la opción del analizador `buildAST` esté activada. Es decir, en el analizador en cuestión debemos encontrar algo como ésto:

```
class MiParser extends Parser/TreeParser; // Parser que construye ASTs
```

```
options{   buildAST=true; }
...
```

Una vez el análisis haya acabado es posible obtener el AST generado utilizando el método `getAST()`. El árbol así obtenido es el que se utilizará en el análisis semántico.

```
AST ast = parser.getAST();
```

Sección 2.4: Reglas EBNF extendidas

2.4.1: Introducción

ANTLR utiliza un tipo de reglas que llamaremos EBNF extendidas. Las reglas EBNF extendidas se diferencian de las reglas EBNF en los siguientes aspectos:

- Pueden tener acciones
- Pueden tener predicados sintácticos y semánticos
- Pueden empezar con una acción, para declaración de variables locales.
- Los elementos de la regla son utilizables en el cuerpo de las acciones, y para ello se utilizan "etiquetas".
- Pueden devolver valores
- Pueden tomar parámetros
- Pueden codificar rangos de caracteres en el analizador léxico
- Pueden codificar patrones árbol en el analizador semántico
- Pueden utilizarse operadores especiales para construir el AST (véase sección dedicada a los ASTs en este mismo capítulo)

Ya hemos explicado qué son las acciones y los predicados sintácticos y semánticos en el capítulo anterior. Vamos a ver el resto de los puntos.

Las funcionalidades que voy a explicar pueden utilizarse en cualquier analizador generado por ANTLR (ya sea léxico, sintáctico o semántico). No obstante, por simplicidad, los ejemplos que voy a utilizar van a ser los de un analizador sintáctico.

Esta sección es tanto informativa como de consulta. En ella se explican la mayoría de las visicitudes de la gramática de ANTLR. Si en algún momento no comprende la construcción de una regla, lo más probable es que se explique en esta sección. Los únicos elementos sintácticos que no se explican aquí son:

- La herencia de gramáticas, que se explicará cuando se utilice, en el capítulo 6.
- Los predicados sintácticos y semánticos, que se explicaron el capítulo 1.
- Los operadores de construcción del AST, que se explican en la siguiente sección.

2.4.2: Declarando variables locales en una regla

Un error que he cometido mucho al principio de mi aprendizaje con ANTLR ha sido intentar declarar variables locales dentro de una acción "normal". Considérese por ejemplo la regla que aparece en el siguiente fichero:

```
// Fichero BananaParser.g
header { package leli; } // paquete leli (para poder importar el vocabulario)

class BananaParser extends Parser;
options { importVocab=LeLiLexerVocab; } // importa el vocabulario léxico de
LeLi

regla1 :  {String a;} i:IDENT {a=i.getText();} ;
```

Este analizador, es muy simple pero sirve para ilustrar perfectamente el problema. ANTLR compilará `BananaParser.g` sin problemas[17], generando el analizador en `BananaParser.java`. Sin embargo, cuando intentemos compilar este segundo fichero, java emitirá un error, indicando que la variable "`a`" no ha sido declarada.

El problema está provocado en su raíz por la forma que tiene ANTLR de analizar la gramática. Además del estado "normal", antlr tiene un estado especial, llamado *guessing* (o "adivinando"), en el que entra en ciertos momentos para ver si una regla es aplicable o no (concretamente mientras se evalúa un predicado sintáctico). Pero las reglas no se están reconociendo realmente. En realizad se está "probando a ver si se pueden reconocer". Por lo tanto, las acciones no se deben ejecutar, así que todas las acciones van precedidas de un `if(!guessing)` parecido a ésto:

```
public void regla1()
{
   try{
      if(!guessing) {
         String a;
      }
      Token i = match(IDENT);
      if(!guessing){
         a = i.getText();
      }
   }catch (RecognitionException ex)
   { ... }
}
```

En java, una variable declarada dentro del cuerpo de un `if` es eliminada cuando éste se acaba. Por lo tanto, la declaración `String a` no tiene ningún efecto, y efectivamente la variable `a` no puede encontrarse en el segundo `if`.

En éste caso la opción es muy fácil de arreglar. Bastaría con declarar a dentro de su misma acción:

```
regla1 :  i:IDENT {String a=i.getText();} ;
```

El problema llega cuando queremos utilizar la misma variable en dos acciones. Por ejemplo, así:

```
regla2 : i:IDENT {a=i.getText();}
       | e:LIT_ENTERO {a=e.getText();}
       ;
```

ANTLR permite especificar una tipo especial de acción que se ejecuta *siempre* (esté o no el analizador en estado *guessing*) al principio de la regla. En ella es posible declarar variables cuyo ámbito será todo el método. Para ello basta declarar las acciones *a la izquierda de los dos puntos*, en lugar de a la derecha:

17 Suponiendo que en el mismo fichero haya un analizador léxico en el que se defina "IDENT".

```
regla2 { String a; } : i:IDENT {a=i.getText();}
                     | e:LIT_ENTERO {a=e.getText();}
                     ;
```

En resumen:

Se pueden realizar acciones y declarar variables locales antes de entrar en la fase de reconocimiento de una regla. Para ello hay que colocar las acciones izquierda de los dos puntos en la definición de la regla.

2.4.3: Utilizando las etiquetas

Se pueden colocar etiquetas en los distintos elementos de una regla, tanto en tokens como en referencias a otras reglas. Una etiqueta no es más que un nombre que se le da a un elemento de la regla. Se coloca delante de dicho elemento y separada con dos puntos (':').

Considérese el siguiente ejemplo de regla sintáctica:

```
llamada: i:IDENT PARENT_AB e:expresion PARENT_CE ;
```

En el ejemplo anterior,

- `i` es una etiqueta de un token, mientras que
- `e` es una etiqueta de una referencia a otra regla.

En un analizador léxico no hay diferencias entre “token” y “referencia a otra regla”.

Las etiquetas nos sirven para poder utilizar información importante de los diferentes elementos de cada regla en las acciones y predicados semánticos. Así, en las acciones y predicados semánticos, siendo `etiqueta` una etiqueta cualquiera, podremos escribir:

- `etiqueta` : En las acciones de un analizador sintáctico, utilizar una etiqueta sin más sirve para hacer referencia al **token** que representa. O más formalmente, a la instancia de la clase `antlr.Token` -o alguna de sus subclases- que ha sido utilizada para representar el token en cuestión. En los analizadores semánticos representa un nodo AST (la instancia de una clase que implementa la interfaz AST y que representa el nodo AST).

```
// Imprime un identificador
regla : i:IDENT {System.out.println(i.getText());} ;
```

- `#etiqueta` : La etiqueta precedida del carácter almohadilla (#), sirve para referenciar **el AST** generado por un analizador sintáctico o semántico que esté generando un nuevo AST a partir de `etiqueta`. Todos los métodos de la interfaz `antlr.collections.AST` serán por tanto invocables:

```
// Imprime un AST en forma de cadena
regla : e:expresion {System.out.println(#e.toStringList());} ;
```

2.4.4: Pasando parámetros a una regla

Una evolución evidente del autómata recursivo descendente es poder pasar parámetros a los métodos que representan las reglas. En ANTLR a una regla se le añaden parámetros utilizando los corchetes. Los parámetros de una regla pueden utilizarse tanto en las acciones como en los predicados semánticos

```
regla[int a]
  : {a<0}? IDENT
  | {a==0)? RES_ENTERO
  | {a>0}? RES_REAL
  | RES_CADENA {System.out.println("valor de a: " + a); }
  ;
```

La regla del ejemplo anterior sirve para conocer varias cosas, dependiendo del valor de su parámetro entero `a`. Si `a` es menor que 0, puede reconocer un `IDENT` o una cadena. Si `a` es igual a 0, un entero o una cadena. Si a es mayor que 0, un real o una cadena. Si la entrada es una cadena la reconoce siempre, y se imprime por pantalla el valor de `a`.

Se pueden utilizar elementos etiquetados como parámetros de otras reglas. Por ejemplo:

```
lista_idents[AST tipo] : (IDENT)+
                           { ... }
                       ;
// Pasa como parámetro el AST generado por "tipo" a lista_idents
declaracion : t:tipo lista_idents [#t];
```

2.4.5: Devolviendo valores en una regla

El tipo de los métodos del autómata recursivo descendente generado por ANTLR es por defecto `void`. Sin embargo se puede hacer que una regla devuelva un valor para poder utilizarlo en otros sitios. Para ello hay que utilizar la palabra reservada `returns`:

```
regla3 returns [String a]
   : i:IDENT {a=i.getText();}
   | e:LIT_ENTERO {a=e.getText();}
   ;
```

`regla3` será idéntica a `regla2` salvo por que devolverá la variable a y su tipo será, consecuentemente, `String` en lugar de `void`.

Para poder utilizar el valor devuelto por una regla basta con declarar una variable que guarde dicho valor (por ejemplo como se indica en el apartado anterior) y asignarle el valor con el operador "=".

```
regla4 {String mens} : mens=regla3 {System.out.println(mens);} ;
```

¡No hay que confundir todo lo anterior con las variables que se utilizan para guardar los valores que devuelven las reglas con la cláusula `returns`!:

```
expresion returns [float f] : ... ;

programa
{ float result; }
   : result=e:expresion
   {
      System.out.println("AST: " + #e.toStringList());
      System.out.println("result: " + result);
   }
;
```

En el ejemplo anterior, result sirve para guardar el valor devuelto por `expresion`, mientras que `#e` sirve para referenciar el AST que ha creado.

Frecuentemente los valores devueltos deben ser iniciados. Pueden inicarse en la declaración

`returns`:

```
expresion returns [float f=0] : ... ;
```

Un último apunte: en las reglas de los analizadores léxicos, es conveniente limitar el uso de la cláusula `returns` y del paso de parámetroslas reglas declaradas como `protected`[18]; las reglas públicas no deberían pasar o devolver valores pues los analizadores de niveles posteriores no podrán recuperarlos (pues se limitan a utilizar la interfaz `TokenStream` llamando a `nextToken`).

2.4.6: Utilizando rangos de caracteres en el analizador léxico

Los analizadores léxicos utilizan reglas EBNF para reconocer y agrupar en tokens rangos de caracteres. Por lo tanto, es necesario poder introducir rangos y reconocimiento de caracteres individuales en el analizador. Así:

```
class MiLexer extends Lexer;

OP_MAS : '+';           // Carácter individual
ENTERO : ('0'..'9')+ ;  // Rango de caracteres
```

Además es posible especificar cadenas y palabras reservadas; lo veremos en el capítulo 4.

2.4.7: Utilizando patrones árbol en el analizador semántico

Los analizadores semánticos pueden utilizar en sus reglas cualquier construcción de los otros analizadores (excepto rangos de caracteres o caracteres individuales) y además una exclusiva: el patrón árbol.

Esta sección es una traducción libre de algunos pasajes del manual de ANTLR. Fichero sor.html

La gramática de especificación de analizadores semánticos[19] se basa en la gramática EBNF con acciones y predicados sintácticos y semánticos incrustados.

```
regla : alternativa1
      | alternativa2
     ...
      | alternativaN
      ;
```

Donde cada alternativa es una enumeración de elementos. Cada uno de estos elementos puede ser cualquiera de los presentes en las gramáticas de los analizadores sintácticos (tokens, referencias a otros reglas, cierres...) pero además se añade un nuevo tipo de elemento: el*patrón árbol*, que tiene la siguiente forma:

```
#( token-raíz hijo1 hijo2 ... hijoN )
```

Por ejemplo, el siguiente patrón árbol reconoce un simple AST con el token `OP_MAS` como raíz y dos `LIT_ENTERO`s como hijos:

```
#( OP_MAS LIT_ENTERO LIT_ENTERO)
```

La raíz del patrón árbol tiene que ser una referencia a un token, pero los hijos pueden ser sub reglas. Por ejemplo, la estructura común `si-entonces-sino` podría representarse así:

18 Más información sobre reglas protegidas en el capítulo 4.
19 En la documentación se les llama *treeparsers*, o iteradores de árboles.

```
#( RES_SI expresion listaInstruciones (listaInstrucciones)? )
```

Un concepto importante a tener en cuenta al especificar patrones árbol es que el reconocimiento *no se hace de manera exacta, sino de manera suficiente*. Tan pronto como un AST satisface un patrón, es reconocido, sin importar cuánto del árbol queda por reconocer. Por ejemplo, el patrón `#(A B)` reconocerá cualquier árbol más grande, como `#(A (#B #C) #D)`.

Si se desea diferenciar entre árboles con estructura parecida (por ejemplo, si los dos árboles pueden ser reconocidos de manera suficiente, aunque no exacta) es posible modificar k, aunque dada la propia naturaleza de los ASTs (que requieren un token como raíz) normalmente se deja k=1, y se recurre a los predicados sintácticos.

Supongamos que utilizamos el token OP_MENOS como base tanto para las substracciones normales entre números (2-1) como para el cambio de signo (-6). A la hora de reconocer los ASTs tendríamos que utilizar un predicado sintáctico para diferenciar:

```
expresion : #(OP_MENOS expresion expresion)=> #( OP_MENOS expresion expresion )
          | #(OP_MENOS expresion)
          ...
          ;
```

El orden de los elementos es importante: la primera alternativa tiene que ser siempre “la más grande”.

Los predicados semánticos funcionan exactamente igual que en los otros niveles.

*****

Sección 2.5: Construcción de los ASTs

La función principal del analizador léxico es construir el AST. El cometido de algunos analizadores semánticos también puede ser la construcción del AST. En esta sección explicaremos cómo se construye un AST. Las explicaciones que daremos serán válidas tanto para un analizador sintáctico como uno léxico (los operadores funcionan de la misma forma en los dos tipos de analizadores). Sin embargo los ejemplos que voy a utilizar serán de un analizador sintáctico.

2.5.1: Comportamiento por defecto

La opción `buildAST` activa y desactiva la construcción del árbol AST, en el caso del analizador sintáctico, y la transformación del árbol en el analizador semántico.

Los "ladrillos" con los que ANTLR construye los ASTs son los tokens (aunque éste comportamiento por defecto puede modificarse y extenderse). Consideremos por ejemplo una versión simple de la expresión del producto:

```
expr_producto : IDENT OP_PRODUCTO IDENT ;
```

Lo que se hace con las expresiones binarias (y unarias) es colocar como raíz del AST generado el operador (en nuestro caso, `OP_PRODUCTO`) para poder identificar el tipo de expresión, y como hijos los operandos. Es decir, se debería generar un AST como éste:

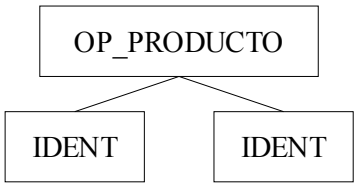

Ilustración 2.9 #(OP_MAS IDENT IDENT)

Repito que los elementos básicos de construcción de ASTs son los tokens. Una regla puede, además de tener tokens, hacer referencia a otras reglas. Por ejemplo, en el siguiente conjunto de reglas:

```
expr_producto : IDENT OP_PRODUCTO IDENT ;
expr_suma : expr_producto OP_SUMA expr_producto ;
```

`expr_suma` hace referencia a `expr_producto` en dos ocasiones. Lo que se hace es sustituir estas referencias por el árbol correspondiente. Es decir, el árbol para `expr_suma` debería estar formado por dos árboles `expr_producto` enraizados con un `OP_SUMA`. Algo así:

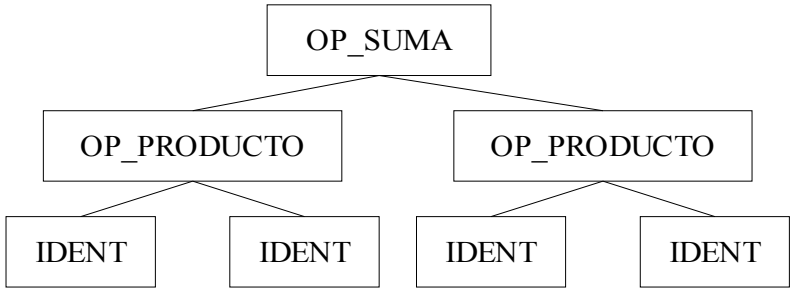

Ilustración 2.10 AST para una expr_suma simplificada

Nótese no obstante que no he dicho que se construyan así: he dicho que *deberían* construirse así. Por desgracia, tal y como están escritas la reglas anteriores, no ocurrirá así. Ello implicaría que ANTLR habría sido capaz de deducir por sí solo las relaciones jerárquicas entre los tokens. ANTLR no es tan listo.

El comportamiento por defecto de ANTLR con respecto a los tokens es *crear un árbol degenerado con todos ellos*.

El concepto de "árbol degenerado" puede ser difícil de entender. Los árboles degenerados, carecen de nodo raíz. Son una secuencia de nodos "hermanos" sin raíz que los identifique. Éste tipo de árbol puede implementarse gracias a que los nodos de los ASTs de antlr "guardan" un enlace con su hermano por la derecha. Éstos enlaces aparecen en línea discontinua.

Por consiguiente el árbol que se creará para `expr_producto` será:

Ilustración 2.11 Árbol degenerado para expr_producto

Por extensión, el AST para expr_suma será también degenerado:

Ilustración 2.12 Árbol degenerado para expr_suma

2.5.2: El sufijo de enraizamiento (^)

Está claro que es necesario poder especificar a ANTLR qué tokens van a utilizarse como raíz de un árbol. Para ello se utiliza el *sufijo de enraizamiento*, que se representa con un acento circunflejo (^). Cuando aparece colocado tras un token, se está indicando a ANTLR que "utilice ese token para enraizar el árbol que hasta ahora ha construido". Es decir, para que nuestras expresiones funcionen adecuadamente hay que añadir el sufijo tras los dos operadores:

```
expr_producto : IDENT OP_PRODUCTO^ IDENT ;
expr_suma : expr_producto OP_SUMA^ expr_producto ;
```

Si dentro de una misma regla hay varios tokens sucedidos con el sufijo de enraizamiento, se van haciendo alternativamente raíces, de forma que se crean sub-árboles. Por ejemplo, la regla:

```
a: A^ B^ C D
```

generará el árbol `#(A #(B C D))`. Éste árbol tiene una raíz en A, y luego un sólo hijo que tiene una raíz en B, y finalmente dos hojas, C y D. Es decir:

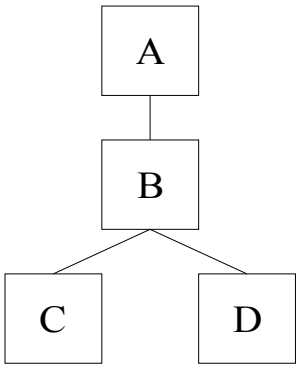

Ilustración 2.13 #(A #(B C D))

El sufijo de enraizamiento solamente puede seguir a tokens (que siguiera a una referencia a una regla no tendría sentido; no se puede enraizar con un árbol de más de un nodo).

Cuando un `Token` se utiliza como raíz, se crea un nuevo AST y se inicializa con los datos del token. En la implementación por defecto se crea un `CommonAST`, y solamente se "guardan" el texto y el tipo del token. La instancia de `Token` no se guarda en el AST; solamente se copian algunos parámetros de ésta.

2.5.3: Sufijo de filtrado (!)

La información que debe guardar en los ASTs es la mínima imprescindible; en muchos casos no es necesario guardar todos los tokens de una regla, pues algunos no aportan información semántica (a pesar de que sintácticamente tengan un papel primordial).

El *sufijo de filtrado* (carácter !) sirve exactamente para eso, para filtrar. Cualquier símbolo sucedido por ! será ignorado a la hora de realizar el AST. El carácter de filtrado puede seguir tanto a tokens como a referencias a otras reglas.

Imaginemos por ejemplo un bucle while de C. La regla que lo representaría sería algo así:

```
listaInst : ... ; // Una lista de instrucciones

bucleWhile: RES_WHILE PARENT_AB expr PARENT_CE
            LLAVE_AB listaInst LLAVE_CE
          ;
```

Supondremos que `lista_inst` es una regla ya definida que devuelve un árbol formado por una serie de instrucciones enraizadas con el token imaginario `LISTA_INST`[20].

Para el AST del bucle tendrá como raíz la palabra reservada `RES_WHILE`, y tendrá dos hijos: la expresión de salida (suponemos que el árbol de `expr` ya se conoce) y la lista de instrucciones:

20 Más abajo explicaré que para poder tener una lista de estas características será necesario enraizar con un token imaginario.

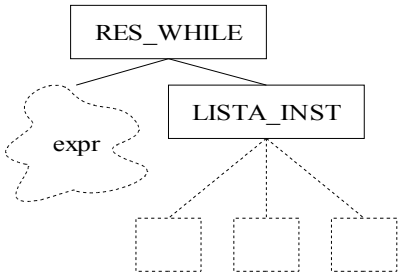

Ilustración 2.14 AST del bucle while

En otras palabras, tenemos hacer `RES_WHILE` raíz del árbol (utilizando el operador de enraizamiento) y tenemos que filtrar todas las llaves y paréntesis, que no aportan información semántica. Lo haremos así:

```
bucleWhile: RES_WHILE^ PARENT_AB! expr PARENT_CE!
            LLAVE_AB! listaInst LLAVE_CE!
          ;
```

2.5.4: Desactivando la construcción por defecto

Ya vimos que es posible activar y desactivar la construcción automática del AST con la opción del analizador `buildAST` (poniéndola a `false` se desactiva la construcción, y poniéndola a `true` se activa).

Con `buildAST` activada, además, es posible desactivar la construcción localmente en una regla. Para ello se utiliza el operador de filtrado. Puede hacerse de dos formas:

- colocándolo tras la parte izquierda (nombre) de la regla se desactiva la construcción por defecto de ASTs para esa regla:

```
regla ! : A B C ; // desactiva la construcción para toda la regla
```

- colocándolo delante de una opción se desactiva la construcción del AST para esa opción.

```
regla : opcion1
      | ! opcion2 // desactiva la construcción para la opción 2
      | opcion3
      ;
```

El árbol resultante de una regla o sub regla precedidas con el prefijo de desactivación de la construcción será un árbol vacío. A efectos de la construcción del AST, será como si las entradas que generaron estas reglas nunca hubieran existido.

2.5.5: Construyendo el AST en una acción

Es posible construir el AST de una regla u opción en una acción. Esto es especialmente útil cuando se ha desactivado la construcción por defecto del AST. Por ejemplo, para construir a mano la expresión simplificada del producto:

```
expr_producto ! : // desactivamos la construcción por defecto
   e1:IDENT op:OP_PRODUCTO e2:IDENT
   {
      #expr = new antlr.CommonAST(#op);
      expr.addChild(#e1);
      #e1.setNextSibling(#e2);
   }
```

Nótese que hemos tenido que añadir etiquetas a los diferentes elementos de la regla para poder referenciarlos en la acción.

El método anterior es poco utilizado. De acuerdo con el manual de ANTLR, es mejor utilizar una factoría. El código necesario para hacerlo es muy largo y tedioso de escribir, así que no lo mostraré aquí. Además, ANTLR proporciona una manera más adecuada de construir un AST en las acciones.

Este método consiste en utilizar una serie de abreviaturas que construyen el AST de la manera más adecuada (además de permitir a la regla permanecer independente del lenguaje generado). Estas abreviaturas son:

- La doble almohadilla, `##`, para representar la raíz de la regla actual.
- La almohadilla seguida de una serie de árboles entre paréntesis, `##(...)`, crea un árbol cuya raíz es el primer árbol y los demás son sus hijos.
- La almohadilla seguida de corchetes, `##[...]`, permite crear un nodo. Puede especificarse un token o un token y una cadena. Veremos cómo utilizar esta construcción un poco más adelante.

Por ejemplo, el código equivalente al anterior utilizando las abreviaturas de ANTLR es el siguiente:

```
expr! : e1:ENTERO op:OP_MAS e2:ENTERO
   { ## = #(#op, #e1, #e2); }
```

Esta sencilla acción realiza la construcción completa del árbol. De todas maneras téngase presente que no es necesaria; el código anterior es equivalente a éste otro, en el que se aprovecha la construcción por defecto de ANTLR:

```
expr : ENTERO OP_MAS^ ENTERO;
```

Una nota sobre el sufijo de filtrado:

En un analizador sintáctico, una referencia a una regla que esté seguida por el sufijo de filtrado *sigue construyendo el AST*, simplemente éste no se añade a la regla padre. Sin embargo sigue estando disponible para utilizarse en las acciones.

Es decir, en la siguiente regla:

```
a: regla1 regla2! regla3;
```

El AST que se construye con `regla2` no se incluye en el AST de a, *pero sí se crea*, pudiéndose utilizar en una acción:

```
a: r1:regla1 r2:regla2! r3:regla3
   { ## = #(#r1, #r2, #r3); }
 ;
```

2.5.6: Casos en los que la construcción por defecto no basta

Las acciones no solamente pueden crear completamente el árbol de una acción; también pueden modificar el árbol que se crea automáticamente en una acción.

A menudo bastará con servirse de sufijos para construir los ASTs. Sin embargo existen tres casos bastante frecuentes en los que la mejor solución es valerse de la construcción por defecto y modificarla adecuadamente en una acción.

- Caso 1: Ninguno de los símbolos de la regla es una raíz adecuada, y debe utilizarse un token imaginario como raíz.

```
clase MyParser extends Parser;

tokens{
   LLAMADA;      // Un token imaginario
   LISTA_IDENTS; // Otro (para el siguiente ejemplo)
}

llamada: IDENT PARENT_AB! lista_idents PARENT_CE!
         { ## = #( #[LLAMADA,"LLAMADA"], ## ); }
```

Si la regla `llamada` no tuviera acciones, se obtendría un árbol degenerado (pues ningún nodo tiene el sufijo ^) con un identificador hermanado de lo que devuelva `lista_idents`. Esta estructura no ofrece información suficiente: ¿cómo saber que se trata de una `llamada`? Un identificador seguido de una lista de identificadores puede ser casi cualquier cosa (una llamada, un acceso o una declaración de variables). Es necesario algún tipo de información (preferentemente en la raíz del árbol) que especifique "esto es una llamada, no una declaración o un acceso".

Precisamente ésto es lo que se hace en la acción de la regla. Primero se crea un nodo ficticio (`#[LLAMADA, "LLAMADA"]`). El primer parámetro proporciona el tipo de token para el árbol, mientras que el segundo es una cadena cualquiera. Se suele utilizar el nombre del token para que sea más fácil hacer el debug.

Una vez creado el nodo ficticio, se utiliza como raíz del árbol que se había generado automáticamente. Esto es muy interesante: el propio árbol generado (referenciado con el comodín `##`) puede ser reutilizado en su propia redefinición. En éste caso, nos ha venido muy bien; podemos eliminar los paréntesis en la generación por defecto (con el sufijo de filtrado, "!") y luego enraizar el AST degenerado resultante con el nodo ficticio de `LLAMADA`.

- Caso 2: listas de elementos, ya sean generadas por clausuras positivas o por cierres de Kleene.

```
lista_idents: IDENT (COMA! IDENT)*
              { ## = #( #[LISTA_IDENTS,"LISTA_IDENTS"], ##); }
```

Si no tuviera la acción, la regla generaría un árbol degenerado consistente en una lista de identificadores, de longitud variable. Este tipo de árboles es muy poco manejable, así que se enraíza con una raíz ficticia como la del ejemplo.

- Caso 3: Cuando es necesario copiar ciertos nodos para utilizarlos en otros sitios. (Veremos este tipo de comportamiento más abajo, al resolver el problema del azúcar sintáctica).

Sección 2.6: Analizadores de la Microcalculadora

2.6.1: El fichero MicroCalc.g y el paquete microcalc

Ha llegado el momento de empezar con nuestra microcalculadora. Vamos a generar tres analizadores (uno léxico, uno sintáctico y uno semántico) por nivel, en java, y todos en el mismo fichero.

A la hora de desarrollar una nueva aplicación en java es muy recomendable introducir todas las clases en un paquete java nuevo. Nosotros incluiremos todos los analizadores en un paquete que llamaremos `microcalc`.

El comienzo del fichero `MicroCalc.g` será, por lo tanto, el siguiente:

```
header {
   package microcalc;
}
```

Para que más tarde el intérprete funcione será necesario que los ficheros en código nativo generados se encuentren situado en un directorio llamado microcalc.

Dado que generaremos código java, no será necesario modificar las opciones globales del fichero, así que las omitiremos. Pasaremos directamente al analizador.

2.6.2: El analizador léxico

Nuestro analizador léxico será sencillo: solamente es capaz de reconocer los enteros y los reales, los operadores aritméticos básicos y los paréntesis.

Comenzaremos definiendo el analizador:

```
class MicroCalcLexer extends Lexer ;
```

No vamos a utilizar la zona de opciones, la de tokens o la de código nativo, así que pasaremos directamente a las reglas.

Lo primero que se suele hacer en un analizador léxico es definir los “blancos”. Los blancos son caracteres que no tienen relevancia a nivel sintáctico, pero que sirven para diferenciar unos tokens de otros. En nuestro caso nos estamos refiriendo a los espacios y los caracteres de tabulación.

```
BLANCO : (' ' | '\t')
         { $setType(Token.SKIP); };
```

Obsérvese lo que hacemos en la acción. `$setType` es una acción especial que permite modificar el tipo del token que estamos creando. El tipo `Token.SKIP` quiere decir que el token no debe ser pasado al nivel sintáctico.

Es usual utilizar caracteres en mayúsculas para denominar las reglas de los analizadores léxicos, porque generan tokens. Más adelante (en los niveles sintáctico y semántico) será muy útil para diferenciar los tokens de las referencias a reglas.

Lo siguiente que vamos a reconocer serán los operadores aritméticos y los paréntesis. Las reglas son muy fáciles de entender, y no creo que requieran explicación:

```
OP_MAS      : '+';
OP_MENOS    : '-';
OP_PRODUCTO : '*';
OP_COCIENTE : '/';

PARENT_AB   : '(';
PARENT_CE   : ')';#
```

Por último, tenemos que definir los números. En nuestro lenguaje de microcalculadora vamos a tratar todos los números como flotantes. Pero externamente debemos permitir utilizar enteros. Así que nuestra definición de `NUMERO` será así:

```
NUMERO : ('0'..'9')+ ('.' ('0'..'9')+)?;
```

Analicemos la entrada. En primer lugar podemos tener 1 o más dígitos (caracteres del 0 al 9). Después tenemos una sub regla opcional (tiene el operador '?'), es decir, que puede aparecer o no. Dicha regla reconoce el carácter del punto ('.') y luego 1 o más dígitos. Así conseguimos que la regla reconozca tanto enteros (sin decimales) como reales (con decimales).

Esta era la última regla que había que definir en el analizador léxico. Pasemos al siguiente nivel.

2.6.3: El analizador sintáctico

El objetivo del analizador sintáctico es reconocer patrones en el flujo de tokens que le llega del nivel léxico y organizar la información en un AST.

El analizador sintáctico comenzará así:

```
class MicroCalcParser extends Parser ;
```

Dado que deseamos utilizar el nivel semántico, vamos a hacer que el analizador construya el AST:

```
options
{
   buildAST = true;
}
```

No utilizaremos la zona de tokens o la de código nativo.

El analizador debe reconocer una serie de expresiones aritméticas. La naturaleza del algoritmo LL(k) hace que reconocer correctamente dichas expresiones no sea trivial, porque al contrario que en bison no es posible definir prioridad en los operadores. Es decir, ante la siguiente entrada:

```
1+3*4/2
```

tenemos que encontrar una forma de hacer que el analizador reconozca `1+(3*(4/2))`, y no `(1+3)*(4/2)` o `1+(3*4)/2`.

El problema de las expresiones está ya muy estudiado y resuelto. La solución consiste en comenzar reconociendo las expresiones de más baja prioridad utilizando como “operadores” las de más alta prioridad. Los operadores se colocan siempre como raíces del AST. Las operaciones de suma, que son las de más baja prioridad, se reconocerían en primer lugar:

```
expresion : expSuma; //expresion es la regla “base”, la primera que se reconoce
expSuma : expResta (OP_SUMA^ expResta)*;
```

Obsérvese que,en el caso de que aparezca un token `OP_SUMA`, éste se utiliza como raíz del AST. `expSuma` se define en función de `expResta`, cuyo operador tiene mayor prioridad. De la misma

forma, se definirá `expResta` en función de `expProducto` y `expProducto` en función de `expCociente`.

```
expResta      : expProducto (OP_MENOS^ expProducto)* ;
expProducto   : expCociente (OP_PRODUCTO^ expCociente)* ;
expCociente   : expBase  (OP_COCIENTE^ expBase)* ;
```

Queda por ver qué es `expBase`.

`expBase` es el tipo de expresión de más alta prioridad. Por lo tanto, debe ser el último en evaluarse. Lo último que debe evaluarse son los propios números y las expresiones entre paréntesis. Así que `expBase` quedará así:

```
expBase : NUMERO
        | PARENT_AB! expresion PARENT_CE!
        ;
```

(Nótese que los paréntesis no se incluyen en el AST – se utiliza el operador de filtrado)

El nivel sintáctico está acabado. Pasemos al nivel semántico.

2.6.4: Nivel semántico

Nuestro nivel semántico se limitará a realizar los cálculos aritméticos oportunos a partir del AST, devolviendo el resultado en un `float`.

Comenzaremos definiendo el analizador:

```
class MicroCalcTreeParser extends TreeParser ;
```

No utilizaremos la sección de opciones, la de tokens o la de código nativo. Pasemos, pues, a la sección de reglas.

La única regla que va a tener el analizador semántico es excepcion. excepcion devuelve un `float` (inicialmente con valor 0.0).

```
expresion returns [float result=0]
```

Como veremos, en muchas ocasiones vamos a necesitar guardar resultados parciales de "expresiones hijas" en sendos flotantes, así que declararemos dos variables locales, `izq` y `der`, para guardar los resultados de los cálculos del "hijo derecho" y el "hijo izquierdo":

```
expresion returns [float result=0]
{ float izq=0, der=0; } :
```

`result` será calculado a partir del AST. Los cálculos serán:

- El valor del número, si el AST es un `NUMERO`
- La suma de los hijos izquierdo y derecho, si la raíz es `OP_SUMA`
- La resta de los hijos izquierdo y derecho, si la raíz es `OP_MENOS`
- El producto si la raíz es `OP_PRODUCTO`
- El cociente si la raíz es `OP_COCIENTE`. En este último caso tendremos en cuenta que el divisor puede ser 0.

Así, el código completo de `expresion` será:

```
expresion returns [float result=0]
{ float izq=0, der=0; }
   : n:NUMERO
     { result = new Float(n.getText()).floatValue(); } // se devuelve el float
   | #(OP_MAS      izq=expresion der=expresion)
     { result = izq + der ; }                 // se suman izq y der
   | #(OP_MENOS    izq=expresion der=expresion)
     { result = izq - der ; }                 // se restan
   | #(OP_PRODUCTO izq=expresion der=expresion)
     { result = izq * der ; }                 // se multiplican
   | #(OP_COCIENTE izq=expresion der=expresion)
     {
      if(der==0.0)
         throw new ArithmeticException("División por cero");
      result = izq / der;              // se dividen (o se lanza una excepción)
     }
   ;
```

Nótese que podríamos haber realizado el cálculo en el nivel sintáctico. Sin embargo esto no es aconsejable: la función más adecuada del nivel sintáctico es construir el AST.

2.6.5: El fichero MicroCalc.g

A continuación se muestra un listado completo del fichero `MicroCalc.g`, con algunos comentarios explicativos añadidos.

```
header {
   package microcalc;
/*-------------------------------------------------*\
 |    Un intérprete para una microcalculadora     |
 | _____________________________________________  |
 |                                                |
 |                   MICROCALC                    |
 | _____________________________________________  |
 |                                                |
 |            Enrique J. Garcia Cota              |
\*-------------------------------------------------*/
}

/** Analizador léxico de MicroCalc **/
class MicroCalcLexer extends Lexer ;

/** Ignorar los espacios en blanco **/
BLANCO : (' ' | '\t')
         { $setType(Token.SKIP); };

/** Operador suma **/
OP_MAS      : '+';
/** Operador resta **/
OP_MENOS    : '-';
/** Operador multiplicación **/
OP_PRODUCTO : '*';
/** Operador cociente **/
OP_COCIENTE : '/';

/** Paréntesis abierto **/
```

```
PARENT_AB   : '(';
/** Paréntesis cerrado **/
PARENT_CE   : ')';

/** Número (real o decimal) **/
NUMERO : ('0'..'9')+('.' ('0'..'9')+)?;

/** Analizador sintáctico de Microcalc **/
class MicroCalcParser extends Parser ;
options
{
  buildAST = true; // Construir el AST
}

/** Regla raíz **/
expresion : expSuma ;

/** Expresión suma (nivel 4) **/
expSuma       : expResta (OP_MAS^ expResta)* ;
/** Expresión resta (nivel 3) **/
expResta      : expProducto (OP_MENOS^ expProducto)* ;
/** Expresión producto (nivel 2) */
expProducto   : expCociente (OP_PRODUCTO^ expCociente)* ;
/** Expresión cociente (nivel 1)**/
expCociente   : expBase  (OP_COCIENTE^ expBase)* ;
/** Expresión base (nivel 0) **/
expBase : NUMERO
        | PARENT_AB! expresion PARENT_CE!
        ;

/** Analizador semántico de MicroCalc **/
class MicroCalcTreeParser extends TreeParser ;

/**
 * Esta regla parsea el AST y devuelve el resultado de los cálculos.
 * También es capaz de lanzar una excepción si se encuentra una
 * división por cero.
 **/
expresion returns [float result=0]
{ float izq=0, der=0; }
   : n:NUMERO
     { result = new Float(n.getText()).floatValue(); }
   | #(OP_MAS      izq=expresion der=expresion)
     { result = izq + der ; }
   | #(OP_MENOS    izq=expresion der=expresion)
     { result = izq - der ; }
   | #(OP_PRODUCTO izq=expresion der=expresion)
     { result = izq * der ; }
   | #(OP_COCIENTE izq=expresion der=expresion)
     {
      if(der==0.0)
         throw new ArithmeticException("División por cero");
      result = izq / der;
     }
   ;
```

2.6.6: Generando los analizadores de MicroCalc

Una vez hemos definido el fichero `MicroCalc.g`, el siguiente paso será "compilarlo". Los ficheros de gramáticas se "compilan" cuando se traducen para generar el código nativo (java, C++ o C#) de los analizadores.

La compilación la realizaremos utilizando la línea de comandos del sistema operativo MS Windows. Para compilar en cualquier otro sistema operativo los pasos serán muy similares (una vez se haya instalado el compilador de java y ANTLR).

Para compilar `MicroCalc.g` como vamos a mostrar en este apartado será necesario que ANTLR esté convenientemente instalado en el `CLASSPATH` de java. En el apéndice "Cuestiones técnicas" hay información sobre cómo instalar ANTLR.

La forma más común de compilar un fichero de gramáticas es mediante la siguiente orden:

```
C:\> java antlr.Tool path_del_fichero
```

Así, si deseamos compilar el fichero `MicroCalc.g` que se encuentra en el directorio `C:\microcalc`, deberemos hacer algo así:

```
C:\> cd microcalc
C:\microcalc\> java antlr.Tool MicroCalc.g
```

ANTLR se pondrá entonces en funcionamiento. Si todo ha ido bien, aparecerán varios ficheros de extensión *.java en el directorio `c:\microcalc`:

- `MicroCalcLexer.java`
- `MicroCalcParser.java`
- `MicroCalcTreeParser.java`
- `MicroCalcLexerTokenTypes.java`

Los tres primeros ficheros son los analizadores propiamente dichos. El último es un interfaz java que contiene todos los "tipos de tokens" del analizador léxico (más información sobre ésto en el capítulo 4).

Queda un único paso para tener unos analizadores completamente funcionales: compilar los ficheros *.java para obtener los ficheros *.class. Esto se realiza de la misma manera que compilamos cualquier fichero *.java. Dado que vamos a compilar un paquete completo, la manera más sencilla de hacerlo es:

```
C:\microcalc\> cd ..
C:\> java microcalc\*.java
```

Como hemos incluido todos los analizadores dentro del paquete microcalc, es necesario que se los ficheros estén en un directorio llamado microcalc para que puedan compilarse.

Para poder compilar el paquete microcalc, será necesario que ANTLR se encuentre en el `CLASSPATH`.

Sección 2.7: Ejecutando Microcalc

Hemos pasado la fase más difícil en el desarrollo de microcalc: generar los analizadores del nuevo microlenguaje. Sin embargo todavía nos queda un paso muy importante por dar: hacer que los analizadores "funcionen".

Hablando en la jerga de C y C++, diríamos que nos hace falta la función `main`. En la jerga de java, nos hace falta el método `main`. Será en dicho método donde obtendremos datos de entrada y donde se los pasaremos a los analizadores, que tendremos que haber creado.

2.7.1: La clase Calc

Incluiremos el método main en una nueva clase, de la que será en único método disponible. Llamaremos `Calc` a dicha clase, así que la implementaremos en el fichero `Calc.java`.

Inicialmente la clase Calc permitirá leer por la entrada estándar un cálculo a realizar, e imprimirá el resultado por la pantalla, terminando.

La clase `Calc` formará parte del paquete `microcalc`, así que `Calc.java` comenzará así:

```
package microcalc;
```

Después vendrá la sección de importaciones (con `import`). La única importación realmente imprescindible – aunque luego añadamos más – será la de la interfaz AST de ANTLR:

```
import antlr.collections.AST;

   public static void main(String args[])
   {
```

He aquí un esquema de los pasos que vamos a realizar con `Calc`:

- Paso 1: Leer una cadena de texto por la entrada estándar.
- Paso 2: Construcción de un analizador léxico para dicha cadena.
- Paso 3: Construcción de un analizador sintáctico asociado al analizador léxico.
- Paso 4: Lanzar el análisis sintáctico
- Paso 5: Obtener el AST construido en el análisis sintáctico.
- Paso 6: Construir un analizador semántico
- Paso 7: Recorrer el AST para así obtener el valor del cálculo.

En su versión inicial, `Calc` deberá leer una línea que el usuario tecleará. Dado que los analizadores léxicos necesitan un flujo y no una cadena para constuirse, guardaremos la entrada en forma de `StringReader`. Será necesario importar las clases que vamos a utilizar.

```
...
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.StringReader;
...

     try
     {
        // PASO 1. Obtener una línea de texto por la entrada estándard
        BufferedReader in =
```

```
          new BufferedReader(new InputStreamReader(System.in));
        StringReader sbr = new StringReader(in.readLine());
```

Después creará un analizador léxico al que asociará la entrada:

```
        // PASO 2. Crear un analizador léxico para dicha entrada
        MicroCalcLexer lexer   = new MicroCalcLexer(sbr);
```

Luego un analizador sintáctico para dicho analizador léxico. El análisis se iniciará llamando al método `expresion()` - la regla raíz del análisis sintáctico.

```
        // PASO 3. Crear un analizador sintáctico asociado al léxico
        MicroCalcParser parser = new MicroCalcParser(lexer);
        // PASO 4. Lanzar el análisis léxico-sintáctico
        parser.expresion();
```

Por último, crearemos un analizador semántico. Llamaremos al método `expresion` de dicho analizador pasándole el AST obtenido del analizador sintáctico. `expresion` devolverá el resultado de los cálculos en forma de `float`. Imprimiremos dichos resultados:

```
        // PASO 5. Obtener el AST
        AST ast = parser.getAST();

        // PASO 6. Crear el analizador semántico
        MicroCalcTreeParser treeParser = new MicroCalcTreeParser();

        // PASO 7. Recorrer el AST
        float result = treeParser.expresion(ast);

        // Imprimimos el resultado
        System.out.println("Resultado: " + result);
    } catch (Exception e) {
       e.printStackTrace(System.err);
    }
}
```

2.7.2: Clase Calc refinada

Vamos a hacer una calculadora un poco más útil.

- En lugar de poder leer una sola vez la entrada, haremos que la calculadora lea una y otra vez la entrada estándar, leyendo cada línea e imprimiendo el resultado por pantalla.
- El proceso terminará cuando el usuario escriba “salir”. En caso de que se introduzca un valor incorrectamente construído, la calculadora deberá ser capaz de pedir una nueva entrada, sin que se aborte la ejecución del programa.

Para poder realizar el primer punto vamos a tener que utilizar un bucle do-while, en el cual efectuaremos los 7 pasos una y otra vez. Las variables que utilizaremos las crearemos fuera del bucle para ganar algo de eficiencia, aunque las iniciaremos en el bucle.

En cuanto al segundo punto, lo que haremos será capturar las excepciones “de reconocimiento” de ANTLR. Todas las excepciones de este tipo son subclases de `antlr.ANTLRException`[21], así que bastará con capturar cualquier excepción de dicho tipo en el bucle principal.

A continuación se encuentra el código completo de la nueva clase `Calc`:

21 Más información sobre errores y excepciones en el capítulo 7 (Recuperación de errores)

```
package microcalc;

/*-----------------------------------------------*\
 |    Un intérprete para una microcalculadora    |
 | _____________________________________________ |
 |                                               |
 |                   MICROCALC                   |
 | _____________________________________________ |
 |                                               |
 |            Enrique J. Garcia Cota             |
\*-----------------------------------------------*/

// Importar clases para leer en la entrada estándar
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.StringReader;

// Importar AST
import antlr.collections.AST;

// Importar excepcines de ANTLR
import antlr.ANTLRException;

public class Calc
{
   public static void main(String args[])
   {
      try
      {
         // Leer de la entrada
         BufferedReader in =
            new BufferedReader(new InputStreamReader(System.in));

         // Iniciar variables que se utilizarán en el bucle
         String entrada                = null;
         MicroCalcLexer lexer          = null;
         MicroCalcParser parser        = null;
         MicroCalcTreeParser treeParser = null;
         AST ast                       = null;
         float result                  = 0;
         boolean bSalir                = false;

         // Comienza el bucle principal del método
         do
         {
            System.out.print("\n? ");

            // PASO 1. Leer la entrada
            entrada = in.readLine();

            // PASO 1b. Si comienza con "salir", salir
            if(entrada.startsWith("salir"))
               bSalir=true;
            else // si no...
            {
               try
               {
                  // PASOS 2 y 3 Crear lexer y parser
```

```
                lexer      = new MicroCalcLexer(new StringReader(entrada));
                parser     = new MicroCalcParser(lexer);

                // PASO 4. Lanzar el análisis sintáctico
                parser.expresion();

                // PASO 5. Obtener el AST
                ast = parser.getAST();

                // PASO 6. Crear el treeParser
                treeParser = new MicroCalcTreeParser();

                // PASO 7. Lanzar el análisis semántico
                result = treeParser.expresion(ast);

                // Imprimir el resultado
                System.out.println("Resultado: " + result);
              // Capturar excepciones de ANTLR
              }catch(ANTLRException re){
                System.err.println( "No entendí la entrada: '"+
                                    entrada + "'");
                System.err.println(re.getMessage());
              }
              // Capturar división por cero
              catch(ArithmeticException ae)
              {
                System.err.println(ae.getMessage());
              }
            }

        } while(!bSalir);

      // Capturar otras excepciones
      }catch (Exception e) {
        e.printStackTrace(System.err);
      }
    }
```

He señalado dos zonas del código que me parecen importantes:

- La primera, porque ilustra cómo hacemos para salir del bucle – no modificamos los analizadores para un simple comando de salida, en lugar de ello comprobamos directamente si la entrada comienza con “salir”.
- La segunda muestra cómo capturar cualquier excepción lanzada durante el reconocimiento de los analizadores o durante su creación. Este bloque catch permite capturar cualquiera de ellas, dejando pasar otras como errores de entrada salida, etc. Además capturamos los errores de división por cero (capturando `ArithmeticException`).

2.7.3: Utilizando microcalc

Una vez guardado el fichero Calc.java en el directorio c:\microcalc, solamente hay que recompilar:

```
c:\> javac microcalc\*.java
```

Para conseguir una microcalculadora completamente funcional.

Solamente nos quedará ejecutar la calculadora. Tenemos dos opciones.

La primera consiste en ejecutar microcalc desde C:\>. Si el directorio actual ('.') se encuentra en el `CLASSPATH`, podremos ejecutar microcalc así:

```
c:\>java microcalc.Calc
```

En otro caso tendremos que añadir c:\ a `CLASSPATH`. Esto puede hacerse justo antes de invocar la microcalculadora:

```
c:\algun\sitio\> set CLASSPATH=%CLASSPATH%;C:\
c:\algun\sitio\> java microcalc.Calc
```

o puede hacerse en la línea de comandos con la extensión -cp:

```
c:\algun\sitio\> java -cp %CLASSPATH%;C:\ microcalc.Calc
```

He aquí una sesión con microcalc:

```
? 1+2
Resultado: 3
? 1+2*3
Resultado: 7
? (1+2)*3
Resultado: 9
? 3.0/2.0
Resultado: 1.5
? 4.5/(1-1)
División por cero
? salir
c:\>
```

Sección 2.8: Conclusión

En este capítulo hemos visto los conceptos esenciales para trabajar con ANTLR; cualquiera que tenga curiosidad podría comenzar a implementar su propio compilador.

Los conceptos que hemos visto son:

- Función de ANTLR (para qué sirve)
- El flujo de información entre los niveles de análisis en ANTLR
- EBNF extendida
- Concepto, creación y recorrido de ASTs
- Compilación y ejecución de un sistema de análisis basado en ANTLR

Muchas cosas quedan por decir. Éste es, al fin y al cabo, un capítulo introductorio, en el que se presentan las capacidades más básicas de ANTLR. Por "capacidades básicas de ANTLR" quiero decir el conjunto mínimo de capacidades que permiten construir y ejecutar un intérprete de tres niveles (léxico, sintáctico y semántico). Así que ¡cuidado!: por ser "básicas" no tienen por qué ser "las únicas que merezca la pena aprender".

En capítulos venideros estudiaremos otras capacidades que, si bien no son absolutamente imprescindibles para desarrollar analizadores con ANTLR, son tremendamente útiles. Algunas serán de uso muy frecuente, como los predicados sintácticos y semánticos; otras serán menos usuales pero muy potentes, como la herencia de gramáticas.

Estudiaremos las funciones de ANTLR que nos quedan por aprender (además de repasar las que ya hemos comenzado a utilizar) en los próximos capítulos. Para ello vamos a utilizar un ejemplo práctico: vamos a construir un compilador de un lenguaje "serio", orientado a objetos y fuertemente tipado.

Pero antes deberemos definir dicho lenguaje.

Capítulo 3:
LeLi: un Lenguaje Limitado

"Los límites de mi lenguaje son los límites de mi mente."
Ludwin Wittgenstein

Capítulo 3:
LeLi: un Lenguaje Limitado....68

Sección 3.1: Introducción....69
Sección 3.2: Nivel léxico....70
3.2.1: Blancos.... 70
3.2.2: Comentarios.... 70
3.2.3: Literales.... 70
3.2.4: Identificadores.... 70
3.2.5: Palabras reservadas.... 71
Sección 3.3: Niveles sintáctico y semántico.... 72
3.3.1: Características básicas de LeLi.... 72
3.3.2: Clases especiales del lenguaje.... 72
La clase Sistema....72
La clase Objeto....72
La clase Inicio....73
3.3.3: Declaración de una clase.... 73
3.3.4: Métodos de una clase.... 73
3.3.5: Constructores.... 74
3.3.6: Métodos abstractos.... 76
3.3.7: Variables, atributos y parámetros.... 76
Tipos básicos....76
Forma general de declaración....77
Atributos de una clase....77
Variables locales de un método....77
Parámetros de un método o constructor....78
3.3.8: Expresiones.... 79
Expresiones aritméticas....79
Expresiones de comparación....80
Expresiones lógicas....80
Asignaciones....80
Expresiones con cadenas....80
Acceso a variables, atributos y parámetros. Enmascaramiento....81
Métodos de los tipos básicos....83
Trabajando con tipos: convertir y esUn....83
Prioridad en las expresiones....85
Sección 3.4: Instrucciones de LeLi.... 86
3.4.1: Separación de instrucciones.... 86
3.4.2: Asignaciones.... 86
3.4.3: Bucles : mientras, hacer-mientras, desde.... 86
3.4.4: Condicionales: si.... 87
3.4.5: Instrucción volver.... 87
Sección 3.5: Otros aspectos de LeLi.... 88
3.5.1: Herencia de clases.... 88
Raíz del sistema: clase Objeto....89
3.5.2: Gestión de la memoria.... 89
Sección 3.6: Conclusión....90

Sección 3.1: Introducción

El lenguaje que vimos en el capítulo 2 es el típico ejemplo que podemos encontrar en un tutorial sobre ANTLR colgado en la red: un lenguaje "pequeño" que queda muy lejos de un verdadero lenguaje de programación. En este capítulo definiré un lenguaje más parecido a los "reales", estableciendo la base para los futuros capítulos, en los cuales le iremos construyendo un compilador.

El lenguaje que vamos a implementar se llama Lenguaje Limitado, o LeLi.

No intentaremos construir un super-lenguaje que revolucione el mundo de la programación. LeLi es un lenguaje orientado a objetos, pero con muchas limitaciones, incluyendo (pero no limitándose a):

- No contempla el manejo de tablas o de otros tipos de datos abstractos (tablas hash, listas...) para manejar conjuntos de variables.
- No hay "referencias" (punteros) de tipo alguno. Una variable apunta a la misma dirección de memoria desde que se crea hasta que se destruye.
- No hay pasos de parámetro por valor, solamente por referencia.
- No se pueden redefinir operadores; los operadores están reservados a los tipos básicos.
- No se contempla el uso de interfaces, ni la herencia múltiple.
- No se contempla el manejo de excepciones; los errores aritméticos (desbordamiento de pila, división por cero) provocan una parada irremediable del programa.
- No se contempla el uso de tipos parametrizados (`templates`)
- No se contempla la distribución de las clases en paquetes y librerías. En general no se admiten compilaciones que involucren a más de un fichero.
- No habrá un recolector de basura propiamente dicho: todas las variables, excepto las devueltas por un método, serán destruídas cuando se acabe su ámbito. Dado que no se pueden utilizar referencias de ningún tipo, no será necesario realizar ninguna otra operación.

En este capítulo describiremos todas las características de LeLi. Para ello empezaremos con los aspectos léxicos del lenguaje, y una vez definidos pasaremos a los aspectos sintácticos y semánticos.

Sección 3.2: Nivel léxico

LeLi se parece mucho a otros lenguajes conocidos, como C o PHP, y esto incluye el nivel léxico. Por lo tanto no hay muchas sorpresas.

3.2.1: Blancos

El carácter de espacio, la tabulación y los caracteres de salto de línea[22] serán ignorados a partir del nivel léxico. Pueden ser utilizados para hacer más fácil la lectura al programador.

3.2.2: Comentarios

Los comentarios serán similares a los de C y C++: agrupaciones de líneas delimitadas por las cadenas "/*" y "*/", o líneas individuales precedidas de la doble barra ("//"):

```
// Estos comentarios solamente pueden ocupar una línea.

/* Estos comentarios pueden ocupar una
   o varias
   líneas,
   según la necesidad del programador.
*/
```

Los comentarios de varias líneas **no pueden anidarse**.

3.2.3: Literales

Las cadenas se expresan utilizando las dobles comillas. No existen los literales carácter: en su lugar se usa una cadena de longitud 1.

No se admiten tabuladores o saltos de línea o comillas dentro de las cadenas; en lugar de eso el lenguaje proporciona las siguientes palabras reservadas:

- `nl`: Cadena con el carácter de nueva línea.
- `tab`: Cadena con el carácter de tabulación.
- `com`: Cadena con las comillas (").

No se admiten caracteres de escape, ni siquiera para especificar caracteres unicode.

No existen literales de tipo "carácter". Un carácter entre dos comillas será considerado un literal cadena.

Los literales enteros se representan utilizando los dígitos del 0 al 9. No se pueden representar números en octal o hexadecimal como en C++ o java.

Para representar cantidades no enteras están los literales flotantes, que se representan como dos series de dígitos unidos por el carácter punto ("."). Para representar cantidades menores que la unidad será obligatorio comenzar el literal con el dígito 0. No será posible utilizar la nomenclatura mantisa-exponente (xxxeyyy) de C++ y java.

3.2.4: Identificadores

Un identificador válido será cualquier cadena de caracteres, dígitos y el carácter de subrayado, siempre y cuando no sea una palabra reservada del lenguaje y no empiece por un dígito.

22 Por "saltos de línea" entenderemos el carácter \n de linux, el \r de Macintosh y el \n\r de Microsoft.

3.2.5: Palabras reservadas

LeLi es un lenguaje sensible a las mayúsculas. Todas las palabras reservadas deberán escribirse en minúsculas, excepto los nombres de los tipos básicos, que tienen la primera letra en mayúscula.

Las palabras reservadas de LeLi son las siguientes:

- `Entero`
- `Real`
- `Booleano`
- `Cadena`
- `cierto`
- `falso`
- `nl`
- `tab`
- `com`
- `clase`
- `extiende`
- `método`
- `constructor`
- `atributo`
- `abstracto`
- `parámetro`
- `convertir`
- `mientras`
- `hacer`
- `si`
- `otras`
- `volver`
- `esUn`
- `y`
- `o`
- `no`
- `super`

Sección 3.3: Niveles sintáctico y semántico

3.3.1: Características básicas de LeLi

LeLi será un lenguaje **orientado a objetos** y **fuertemente tipado**. Sus características básicas serán:

- Toda variable deberá ser inicializada antes de su utilización (mediante una asignación).
- El compilador será sensible a las mayúsculas y minúsculas.
- Los tipos básicos serán los usuales, salvo por el hecho de que el tipo carácter no existirá.
- Se implementará la herencia de clases a un nivel básico (todas las funciones serán virtuales, menos los constructores. La herencia no podrá ser múltiple).
- Se implementarán métodos y atributos abstractos.
- El manejo de referencias será limitado: se permitirá el paso por referencia y por valor, **pero no las asignaciones de referencias**: todas las asignaciones serán "por valor". De hecho, no se permiten asignaciones entre instancias de tipos definidos por el usuario.
- La vida de una variable terminará al terminar el entorno (iteración, condición, función o clase) en el que fue creada.
- Existirá la clase básica `Objeto`.
- Se permitirán clases recursivas.
- Se permitirán métodos recursivos.

Ciertas capacidades que se dan en otros lenguajes de programación orientados a objetos no se implementarán:

- No se implementarán interfaces.
- No se podrán definir operadores para una clase.
- No habrá ocultación de atributos. Todos los atributos de una clase serán públicos, al igual que sus métodos.

3.3.2: Clases especiales del lenguaje

La clase Sistema

La clase sistema será una clase definida por defecto que proporcionará el método abstracto polimórfico `imprime`:

- `método abstracto imprime(Cadena mensaje)`: Imprimirá mensaje por pantalla.
- `método abstracto imprime(Booleano booleano)`: Imprimirá `cierto` o `falso` por pantalla, dependiendo del valor de `booleano`.
- `método abstracto imprime(Entero entero)`: Imprimirá `entero` por pantalla.
- `método abstracto imprime(Real real)`: Imprimirá `real` por pantalla.

La clase Objeto

Toda clase creada por el programador heredará automáticamente de la clase Objeto, que solamente dispone del método `aserto`:

- `método abstracto aserto(Booleano condición; Cadena mensaje)`: Si `condición` es `falso`, se imprimirá por pantalla `mensaje` y se abortará la ejecución del programa.

Al tratarse de un método abstracto de la clase `Objeto`, toda clase creada por el usuario podrá utilizarlo directamente (podrá escribirse `aserto` en lugar de `Objeto.aserto`).

La clase Inicio

El compilador buscará una clase llamada `Inicio` (la primera I mayúscula y el resto minúsculas) y, dentro de éste un método abstracto llamado `inicio`(con minúsculas y sin parámetros). Éste será el método que se invoque al ejecutar la aplicación.

3.3.3: Declaración de una clase

El esqueleto de una declaración de clase es el siguiente:

```
clase NombreClase [extiende NombreClasePadre]
{
   // Declaraciones de atributos y métodos
}
```

3.3.4: Métodos de una clase

Un método de una clase es un fragmento de código, en el que se especifica mediante instrucciones (más adelante veremos qué son y qué instrucciones permite LeLi) un algoritmo a ejecutar. LeLi permite tres tipos diferentes de métodos.

El tipo más usual de método de LeLi es el método básico. Un método básico tiene la siguiente forma:

```
método [Tipo] NombreMetodo( [ListaParametros] )
{
   // Cuerpo del método
}
```

Donde `ListaParametros` es una lista de declaraciones de parámetros del método (ver más abajo). La lista de parámetros puede ser vacía.

Tipo es el nombre del tipo que devuelve el método. Normalmente se le llama "tipo del método". Si un método no devuelve nada, es de tipo vacío, por lo que `Tipo` se omite. A los métodos que no devuelvan algo les llamaremos *métodos vacíos* y, por simetría a los que devuelvan algo los llamarenos *no vacíos*.

En todo método no vacío se declarará automáticamente una variable del mismo tipo del método, que podrá ser modificada o cambiada. A esta variable la llamaremos *variable de retorno del método.* Dicha variable inicialmente tomará el valor por defecto de su tipo. Esta manera de manejar los valores de retorno de una función es similar a la que utiliza Visual Basic. Veámosla:

```
método Entero suma (Entero A; Entero B)
{
   suma = A + B;
}
```

No tiene mucho misterio, ¿verdad?. Como ya hemos dicho, el valor de la variable de retorno del método al iniciarse éste es el valor por defecto. Es decir, al iniciarse el método suma el valor de la variable `suma` es 0, que es el valor por defecto de los enteros.

Dado que las variables de método deben poder iniciarse con valores por defecto, los únicos tipos válidos para los métodos son los tipos básicos y aquellos tipos definidos por el programador que posean un constructor por defecto.

Pero ¿qué es un "constructor por defecto"?. Es más ¿qué es un *constructor*?.

3.3.5: Constructores

Decíamos que LeLi proporciona tres tipos diferentes de método. Pues bien, los constructores son el segundo tipo.

Los constructores permiten crear instancias de la clase, y normalmente se utilizan para inicializar los atributos de la clase. En LeLi los constructores se definen con la palabra clave `constructor`.

A continuación se ejemplifica el uso del constructor mediante la clase Persona, que utilizaremos repetidamente a lo largo de este capítulo.

```
clase Persona
{
   atributo Cadena Nombre, Apellidos;
   constructor (Cadena Nombre; Cadena Apellidos)
   {
      atributo.Nombre = Nombre;
      atributo.Apellidos = Apellidos;
   }
}
```

(Para más información sobre la palabra clave "atributo", léase el apartado "Expresiones", un poco más abajo)

Una clase puede tener un número ilimitado de constructores, que deberán diferenciarse en su tipo de parámetros.

Para poder instanciar una clase (para poder crear objetos de dicha clase) es necesario que ésta tenga por lo menos un constructor.

Los constructores son métodos vacíos (no devuelven valor alguno) y solamente sirve para iniciar los atributos de cada clase. Los atributos que no sean inicializados en el constructor de la clase tomarán sus valores por defecto.

Veamos entonces qué es el constructor por defecto.

Un constructor por defecto es, simplemente, el constructor de una clase que carece de parámetros. Un nombre más adecuado sería "constructor sin parámetros". Creo que su nombre viene de C++.

En C++, todas las clases tienen implementado el constructor sin parámetro. El constructor se llamó, "por defecto" por eso, porque todas las clases lo tienen "por defecto". El programador puede proporcionar su propio constructor por defecto, simplemente escribiendo un constructor sin parámetros.

La inclusión del constructor por defecto ha sido polémica desde su inicio, principalmente por lo que implica: que toda clase tiene unos valores "por defecto". Esta afirmación es matemáticamente falsa: existen casos en los que una instancia de una clase no tiene sentido mientras que no se le asignen unos valores lógicos a sus parámetros. Los métodos de estas clases simplemente no pueden funcionar correctamente si los atributos tienen valores "por defecto", y

eso se traduce en errores que no se detectan en tiempo de compilación[23].

Por lo tanto, en LeLi no hay "implementación por defecto del constructor por defecto": el programador tiene que escribir el constructor. Escribir el constructor no conlleva mucho trabajo, ya que solamente es necesario inicializar aquellos atributos que requieran una inicialización especial, y los demás tomarán valores por defecto:

```
clase Persona
{
   atributo Cadena Nombre, Apellidos, Dirección;
   constructor() // Constructor por defecto
   { } // Nombre, Apellidos y Dirección toman el valor ""
}
```

De esta forma el constructor por defecto sirve sobre todo para indicar que la clase puede inicializarse sin parámetros, algo que nunca debería haberse extendido.

El valor por defecto de un objeto definido por el usuario es el que se obtiene al llamar al constructor por defecto de dicho objeto. Si la clase de dicho objeto carece de constructor por defecto, entonces el objeto no tiene valor por defecto.

De esta última apreciación obtendremos una curiosa consecuencia: si una clase A tiene un atributo de la clase B y esta última no tiene constructor por defecto, entonces el atributo de clase B deberá ser debidamente inicializado en todos los constructores de A (o de lo contrario LeLi generará un error).

Una última puntualización sobre los constructores: para iniciar los atributos de una clase que no son básicos se utiliza la palabra clave `constructor`:

```
clase ClaseA
{
...
   constructor(Entero e1; Entero e2) // Único constructor de ClaseA
   {
      ...
   }
}

clase ClaseB
{
   atributo ClaseA a; // ClaseB posee un atributo de tipo ClaseA
   constructor(Entero e1)
   {
      a.constructor(e1,0); // Utilizamos constructor para iniciar "a"
   }
}
```

Nótese que si a no fuera inicializado con su único constructor en la construcción de ClaseB LeLi emitiría un error.

No está prohibido invocar explícitamente a los constructores de un objeto en el cuerpo de un método común (fuera de los constructores). De esta forma pueden reinicializarse los valores de un objeto:

23 Para detectar estos errores en tiempo de compilación, Scott Meyers aconsejaba en su "Effective C++" declarar explícitamente como privado el constructor por defecto de las clases en las que dicho constructor no debe ser utilizado.

```
clase ClaseA {...}

clase ClaseB
{
   atritubo ClaseA a;
   constructor ...
   método reiniciar_a(Entero e1)
   {
      a.constructor(e1,0);
   }
}
```

¡Cuidado! Recordemos que la manera de declarar variables locales objeto es diferente: `Tipo`, `nombreVariable` y entre paréntesis los parámetros del constructor:

```
método cualquiera(Entero e1)
{
   ClaseA a(e1,0); // crear variable local de tipo ClaseA llamada a
   atributo.a.constructor(e1,0); // Reiniciar el atributo a
}
```

3.3.6: Métodos abstractos

Veamos ahora el tercer y último tipo de método admitido por LeLi: los métodos abstractos. Comienzan con la cadena `método abstracto`, y son como los métodos abstractos usualmente encontrados en C++ o Java[24]:

```
clase MiClaseAbstracta
{
   método abstracto sumar(Entero A, Entero B)
   {
      Sistema.imprime("La suma de " + A + "+" + B + " es " + A + B + nl);
   }
}
```

Los métodos abstractos son propios de la clase en la que son declarados, y no necesitan instancias de dicha clase para ser invocados. Por ejemplo, el métodos sumar del ejemplo anterior puede ser invocado así:

```
...
MiClaseAbstracta.sumar(1,2);
...
```

Y el resultado será que por pantalla aparecerá el mensaje:

```
La suma de 1 y 2 es 3
```

En el cuerpo de un método abstracto solamente se pueden utilizar los atributos abstractos de la clase a la que pertenecen (para más información sobre atributos abstractos, siga leyendo.)

3.3.7: Variables, atributos y parámetros

Tipos básicos

En LeLi existen los tipos básicos usuales, a saber:

24 Los métodos abstractos de Java se llaman métodos estáticos (usan la palabra clave static).

- `Entero`: Los números enteros, con signo utilizando 32 bits.
- `Real`: Números reales (coma flotante). Equivalentes al tipo double de C.
- `Cadena`: Cadenas de caracteres. Este tipo se parece mucho al tipo `String` de java; tiene un método `longitud()` y un método `subCadena()`, que son explicados más abajo.
- `Booleano`: El tipo booleano solamente puede ser representado con los valores `cierto` y `falso`.

En LeLi no hay tipo carácter, en su lugar se utilizan cadenas de longitud 1.

En LeLi no hay tablas.

Forma general de declaración

En LeLi los atributos de una clase, las variables locales de un método y los parámetros se especifican de la misma forma:

```
NombreClase (listaDecs)+;
listaDec = NombreVariable ('=' expresión)?;
```

Donde `ListaDecs` es una lista de declaraciones separadas por comas.

Atributos de una clase

Los atributos de clases se declaran con la palabra reservada *atributo*:

```
clase Empresa
{
   atributo Cadena NIF;
}

clase Persona
{
   atributo Cadena nombre, apellidos;
   atributo Entero edad;
   atributo Empresa empresa;
}
```

Es posible definir atributos abstractos de una clase utilizando también la palabra reservada `abstracto`:

```
clase Trabajador
{
   atributo abstracto Entero EdadMínima = 18;
}
```

Los atributos abstractos pueden ser inicializados a un valor, como en el ejemplo. Los atributos no abstractos no pueden ser inicializados de esta forma. Ese mismo efecto se consigue con un constructor.

Los atributos y métodos llamados “abstractos” en LeLi son los métodos “propios de la clase en la que se declaran”, y no dependen de la instancia. En otros lenguajes, como C++ y java, se les suele llamar “estáticos”.

Variables locales de un método

Las variables locales de un método se declaran utilizando la forma general de declaración:

```
clase MiClase
{
   método miMétodo()
   {
      Entero A = 2, B;
      Sistema.imprime("El valor de A es "+ A +" y el de B es " + B + nl);
   }
}
```

Las variables locales pueden ser declaradas en cualquier parte del código. Son destruidas al terminar su ámbito (que en LeLi es el conjunto de instrucciones entre '{' y '}' en el que se declararon).

Si el programador no especifica un valor específico al crear una variable (por ejemplo, en el caso anterior la variable A valdrá 2) se le asignará un valor por defecto (como en el caso de la variable B, que por defecto valdrá 0). El valor por defecto de los enteros y reales es 0, el de las cadenas es la cadena vacía y el de los booleanos es falso.

Al instanciar una clase es posible que sea necesario pasar parámetros a su constructor. Ésto se realiza exactamente igual que en C++:

```
Persona GeorgeBush("George", "Bush");
```

En LeLi no hay variables locales estáticas.

Parámetros de un método o constructor

La declaración de los parámetros de un método se realiza de una forma similar a C++:

```
clase Persona
{
   atributo Cadena nombre, apellidos;
   atributo Entero edad;
   ...
   método cumplirAños(Entero años)
   {
      edad = edad + cantidad;
   }
}
```

Hay que resaltar una diferencia importante entre LeLi y C++:

En LeLi los parámetros son pasados por **referencia**, excepto los literales, que son pasados por copia.

Uno de los usos más habituales del paso de parámetros por referencia es el constructor copia:

```
clase Persona
{
   atributo Cadena Nombre, Apellidos;
   constructor (Persona p)
   {
      Nombre = p.Nombre;
      Apellidos = p.Apellidos;
   }
}
```

Como en la mayoría de los lenguajes, las variables pasadas por referencia a un método pueden ser modificadas por dicho método (pueden, por ejemplo, ser utilizadas como parámetros de entrada-salida).

Por último, señalar que la separación entre parámetros es un tanto diferente que C++: se utiliza el punto y coma ";" para separar parámetros de distinto tipo. Los parámetros del mismo tipo pueden reutilizar el nombre de su tipo y estar separados por comas. Por ejemplo, un método que admite dos Enteros y una Cadena tendrá la siguiente especificación:

```
clase Parámetros
{
   método abstracto parámetros (Entero a,b; Cadena mensaje)
   {
      Sistema.imprime("Los enteros son " + a + " y " + b + nl);
      Sistema.imprime("El mensaje es " + com + mensaje + com);
   }
}
```

¡Atención! Aunque se separen con punto y coma en la declaración, se separan con comas exclusivamente en la llamada.

```
clase Inicio
{
   método abstracto inicio()
   {
      // ¡Aquí se separan con comas, no con punto y coma!
      Parámetros.parámetros(1, 2, "Un mensaje");
   }
}
```

3.3.8: Expresiones

Expresiones aritméticas

Las expresiones aritméticas son las usuales. Se realizan entre enteros y reales. Los operadores que se utilizarán son: + (mas), - (menos), * (multiplicación), / (división), ^ (potencia), ++ (postincremento) y --(postdecremento).

El tipo de la expresión cambiará en función de los tipos que se usen:

- Entre `Real` y `Real`, la expresión será siempre de tipo `Real`.
- Entre `Real` y `Entero`, la expresión será siempre `Real`.
- Entre `Entero` y `Entero`, la expresión será siempre `Entera`, salvo la división, que será `Real`.

Expresiones de comparación

Los operadores de comparación son los mismos que en C y C++, a saber: == (igualdad), < (menor que), > (mayor que), <= (menor o igual que), >= (mayor o igual que) y != (distinto de).

Estos operadores son aplicables entre expresiones numéricas de cualquier tipo y devuelven siempre un Booleano.

Expresiones lógicas

Los operadores lógicos son `y`, `o`, y `no`. Tienen la función usual. Hay tres cosas a destacar:

- Las expresiones lógicas solamente pueden utilizar subexpresiones de tipo booleano. En C se podía hacer algo como `if (entero)`; el equivalente de dicha expresión en LeLi es `if (entero!=0)`
- La evaluación de las condiciones lógicas será perezosa: Si el primer término de una expresión "`y`"es falso, no se evaluará la segunda parte de la expresión (que se considerará falsa). Igualmente si el primer término de una expresión "`o`" es cierto, toda la expresión se considerará cierta, sin evaluar la segunda parte.
- El operador "`o`" no es exclusivo: Si ambas partes de una expresión "`o`"son ciertas, la expresión será considerada cierta (de hecho, solamente se evaluará la primera).

Asignaciones

Las asignaciones se realizarán con el operador de asignación (`=`).

Un aspecto importante de las asignaciones es que, al contrario que en C y C++,**las asignaciones no devuelven nada**. Por ejemplo, no será posible hacer `e=3+(a=b).`

No existen operadores aritmético-asignatorios como += o -=. Aunque sí existen los de postincremento(++) y postdecremento(--). Los operadores de postincremento y postdecremento son a todos los efectos idénticos a utilizar una asignación (a++ es lo mismo que a=a+1). Por lo tanto tampoco éstos devuelvan nada.

Las asignaciones entre objetos se hacen por copia, es decir, si se escribe `A=B`, siendo A una variable objeto y siendo B una expresión que devuelva un objeto de una clase o subclase de A, todos los valores de atributos de B se copiarán en los de A. Lo recomendable es que B sea del mismo tipo que A, y no una subclase (pues para hacer uso de los atributos y métodos específicos de las clases deberemos hacer una conversión de tipos; seguir leyendo)

Expresiones con cadenas

LeLi permite realizar ciertas operaciones con las cadenas. La más básica de todas, la concatenación, se realiza con el operador +:

```
Cadena saludo = "¡Hola" + " " + "mundo!";
```

Es posible comparar cadenas con los operadores ==, !=, <, >, <= y >=. La comparación será de tipo alfanumérico, basada en el código ASCII de cada carácter de la cadena.

Por último, es posible sumar cualquier tipo básico a una cadena, obteniéndose una nueva cadena. Al sumar una cadena a:

- Un `Entero`, el resultado es una cadena con dicho entero en su interior, "traducido" a caracteres. En el ejemplo siguiente la cadena `película` toma el valor "12 monos".

```
Cadena película = 12 + " monos";
```

- Un `Real`, el resultado es una cadena con dicho real en caracteres. Por ejemplo:

```
Cadena valorDePi = "PI es " + 3.14159; // valorDePi = "PI es 3.14159"
```

- Un `Booleano`, a la cadena se le sumará la cadena "`cierto`" o "`falso`", dependiendo del valor de dicho `Booleano`. Por ejemplo:

```
Cadena misterio = "1>2 es " +(1>2); // misterio = "1>2 es falso"
```

Una consecuencia interesante de lo anterior podemos deducir que para convertir un `Entero`, `Real` o `Booleano` en `Cadena` bastará con sumarles la cadena vacía ("").

Acceso a variables, atributos y parámetros. Enmascaramiento.

Para utilizar una variable local, un atributo de clase (tanto normal como abstracto) o un parámetro de un método en una expresión basta con escribir su nombre.

Utilizaremos de nuevo la clase `Persona` para ilustrar cómo se utilizan.

```
clase Persona
{
   atributo Cadena Nombre;
   atributo Entero Edad;
   constructor(...)

   método cambiar(Cadena Nombrep; Entero Edadp)
   {
      Entero Edadv = Edadp+1;
      Edad = Ev;
      Nombre = N;
   }
}
```

En el ejemplo anterior se ilustra perfectamente el principal problema de esta nomenclatura: el enmascaramiento de variables. El programador ha tenido que cambiar los nombres de sus variables, añadiendo una "p" minúscula en el caso de los parámetros o una "v"en el caso de las variables locales, para poder hacer referencia sin problemas a cada tipo de variable.

Esta solución aportada por el programador es una de las posibles, pero queda lejos de ser la mejor; al no estar estandarizada la solución de este problema, cada programador podría utilizar una nomenclatura diferente. Por eso LeLi provee un mecanismo de "desenmascaramiento de variables". Para desenmascarar variables basta con seguir dos reglas.

La primera regla a seguir es el "orden de enmascaramiento": la variable que prevalece es "la última en ser declarada". Es decir, en este orden: variables locales, parámetros y atributos de una clase. Veamos esto con un ejemplo:

```
clase Persona
{
   atributo Cadena Nombre;
   constructor{...}
   método muestraEnmascaramiento(Cadena Nombre)
   {
      Cadena Nombre = "Manuel";
      Sistema.imprime(Nombre);
   }
}

clase Inicio
{
   método abstracto inicio()
   {
      Persona Pedro("Pedro");
      Pedro.muestraEnmascaramiento("José");
   }
}
```

En este ejemplo se imprimirá "Manuel" por pantalla, independientemente del valor del parámetro pasado o del atributo. Si no existiera la variable local `Nombre` en el método `muestraEnmascaramiento` se imprimiría "José". Finalmente, si el parámetro tampoco existiera, se imprimiría el atributo de la clase, "Pedro".

¿Cómo hacer referencia en el ejemplo anterior al atributo de la clase o al parámetro? Haciendo uso de la segunda regla: la palabra clave `atributo` sirve para hacer referencia a los atributos de una clase, y la palabra clave `parámetro` sirve para los parámetros. Tras ellas debe utilizarse el punto, igual que en C++ o Java se utiliza la palabra clave `this`. Así, si la línea del método `muestraEnmascaramiento` hubiera sido así:

```
Sistema.imprime(atributo.Nombre);
```

El programa habría imprimido el atributo de la clase Persona, es decir, "Pedro". De la misma forma, habiendo escrito:

```
Sistema.imprime(parámetro.Nombre);
```

Lo que se mostraría sería el parámetro de la función, es decir, "Manuel".

No existe ninguna palabra reservada para hacer referencia a las variables locales, éstas solamente pueden ser referenciadas con su nombre.

El problema del enmascaramiento de variables está solventado casi completamente. No obstante, sigue habiendo un caso en el cual una variable puede enmascarar a otra: definiendo una nueva variable local con el mismo nombre. Es decir, así:

```
método ultimaRegla()
{
   Cadena Mensaje1 = "Primera variable";
   Cadena Mensaje1 = "Segunda variable";
   Sistema.imprime(Mensaje1);
}
```

Tomando estrictamente la primera regla, debería imprimirse "Segunda variable" por pantalla, al haberse declarado la segunda variable más tarde. Podría por tanto suponerse que LeLi permite declarar varias variables locales con el mismo nombre. Y sin embargo no es así.

Utilizar variables locales con el mismo nombre tiene el inconveniente de abrir la puerta a un elevado número de errores de programación, principalmente al "copiar y pegar"[25]. En el ejemplo anterior, es probable que el programador hubiera querido crear dos mensajes diferentes, `Mensaje1` y `Mensaje2`, y trabajar con ellas de manera diferente. Copió y pegó la primera línea, olvidándose de cambiar el nombre de la variable. El programa compilaría perfectamente, y tendríamos un bug perfectamente oculto.

Además, en el improbable caso de que el programador quisiera verdaderamente utilizar `Mensaje1` de esa forma, su código sería confuso e inútil: no hacía falta redeclarar la variable, bastaba con cambiarle la definición.

De todo lo anterior podemos deducir la tercera y última regla de enmascaramiento. Más que una regla es una restricción sobre las declaraciones: en LeLi no es posible declarar dos variables locales con el mismo nombre.

Como consecuencia directa de la última regla de enmascaramiento, no se pueden declarar variables locales en el cuerpo de un bucle.

Métodos de los tipos básicos

Como ya se ha dicho, `Objeto` es el tipo básico de la jerarquía de objetos de LeLi. Los tipos básicos no son una excepción. Por lo tanto cualquier tipo básico posee los métodos heredados de dicha clase. Así, el método siguiente:

```
método cualquiera()
{
   Entero e = 1;
   Sistema.imprime(e.nombreClase());
}
```

Imprimirá por pantalla la cadena "Entero".

El tipo Cadena tiene además una serie de métodos propios de su clase, a saber:

- `método Entero longitud()`: Devuelve la longitud
- `método Cadena subCadena(Entero inicio, Entero longitud)`: Devuelve una subcadena. En LeLi las cadenas se indexan en 0.

Una propiedad curiosa es que es posible invocar los métodos de los tipos básicos directamente desde los literales:

```
1.nombreTipo(); // Devuelve "Entero"
"Hola mundo".subCadena(0,4); // Devuelve "Hola"
```

Trabajando con tipos: convertir y esUn

Para hacer conversiones entre tipos (*castings*) es necesario utilizar el método especial `convertir`, al que se le pasa el objeto a convertir y el tipo al que se quiere convertir. La conversión es posible entre los tipos básicos.

25 Según mi experiencia, "copiar y pegar" es de por sí una de las principales fuentes de errores de programación.

```
convertir(1, Cadena);          // Entero-Cadena. devuelve "1"
convertir(1.0, Entero);        // Real-Entero. devuelve 1
convertir(falso, Entero);      // Booleano-Entero. devuelve 0
convertir("1", Entero);        // Cadena-Entero. Devuelve 1
convertir("3.14159", Real);    // Cadena-Real. Devuelve 3.14159
convertir("cierto", Booleano); // Cadena-Booleano. Devuelve cierto
convertir("uno", Entero); // Error en tiempo de ejecución
```

Al contrario que en C++, una conversión errónea de tipos (como la última) no arrojará el valor por defecto de los enteros (0), sino que provocará un error – y la parada inmediata del programa.

También es posible convertir objetos de clase a superclase, y viceversa; el primer paso siempre es posible, mientras que el segundo puede provocar errores en tiempo de ejecución o de compilación.

```
clase ClaseA {...}

clase ClaseB extiende ClaseA {...}

clase ClaseC extiende ClaseA {...}

clase Inicio
{
   método abstracto inicio()
   {
      ClaseA a; ClaseB b; ClaseC c;
      convertir(c, ClaseA); // Siempre posible
      convertir(a, ClaseB); // Falso, a no esUn ClaseB
   }
}
```

De la misma forma que en C#, existe un mecanismo de *boxing* y *unboxing* que permite hacer transformaciones de tipo `Objeto` a tipo básico (`Entero`, `Real`, `Cadena`).

```
clase Unboxing
{
   método imprimir(Objeto o)
   {
      Si(o esUn Entero)
      {
         Entero e=convertir(o, Entero);
         Sistema.imprime("Es un Entero de valor "+e);
      }
      | (o esUn Real)
      {
         Real r=convertir(o,Real);
         Sistema.imprime("Es un Real de valor "+r);
      }
      | (o esUn Cadena)
      {
         Cadena c = convertir(Entero,o);
         Sistema.imprime("Es una cadena de valor " + com + c + com);
      }
      | (o esUn Booleano)
      {
         Booleano b = convertir(o, Booleano);
         Sistema.imprime("Es un booleano de valor "+ b);
      }
```

```
      | otras
      {
         Sistema.imprime("No es un tipo básico");
      }
   }
}
```

Como puede verse, `esUn` permite saber en tiempo de ejecución si un objeto es una instancia de una clase o superclase dada. Es la única palabra reservada del lenguaje que tiene una letra mayúscula.

Todas las conversiones entre tipos (con `convertir`) se comprueban en tiempo de compilación excepto la conversión dinámica (de clase a subclase) que se comprueba en tiempo de ejecución. Las expresiones que utilicen `esUn` se evalúan siempre en tiempo de ejecución.

Prioridad en las expresiones

La prioridad de los operadores es idéntica a la empleada en C++ y Java. Los paréntesis sirven para cambiarla, de la forma usual.

Sección 3.4: Instrucciones de LeLi

3.4.1: Separación de instrucciones

La separación de instrucciones se realiza con el carácter punto y coma (;), exactamente de la misma manera que se haría en java y C.

3.4.2: Asignaciones

Las asignaciones en LeLi se realizan mediante el operador de asignación "=". Solamente se permiten sobre objetos de tipos básico (`Entero`, `Real`, `Cadena`, `Booleano`). La asignación está prohibida para el resto de los objetos del lenguaje, porque LeLi no admite ningún tipo de "punteros" o "referencias" a objetos, aparte del paso de parámetros por referencia.

3.4.3: Bucles : mientras, hacer-mientras, desde

Los bucles en LeLi se expresan utilizando la palabra reservada `mientras`, la combinación de palabras reservadas `hacer` y `mientras` o la palabra reservada `desde`.

Un bucle "`mientras`" es equivalente a un bucle "`while`" de java o C:

```
metodo buclemientras()
{
   Entero i = 1;

   // Imprime 10 mensajes por pantalla
   mientras(i<=10)
   {
      Sistema.imprime("Iteración (mientras) número " + i + nl);
      i++;
   }
}
```

Un bucle "`hacer-mientras`" es equivalente a un "`do-while`" de java o C:

```
método buclehacermientras()
{
   // Imprime nueve mensajes por pantalla
   i=1;
   hacer
   {
      Sistema.imprime("Iteración (hacer-mientras) número " + i + nl);
      i++;
   }mientras(i<=10);
```

Como es habitual, los dos bucles se comportarán de manera diferente cuando la condición del bucle sea inicialmente falsa:

```
método condicionesFalsas()
{
   // Este bucle jamás se ejecutará
   mientras(1>2)
   { Sistema.imprime("Es imposible llegar aquí" + nl); }

   // Este bucle se ejecutará una vez, aunque su condición sea falsa
   hacer
   { Sistema.imprime("Esto solamente se imprirá una vez" + nl); }
```

```
   mientras (1>2);
}
```

En LeLi las instrucciones de un bucle deben ir **obligatoriamente** delimitadas por llaves ({}), al contrario que en java y C++.

Por último, un bucle `desde` es virtualmente idéntico a un bucle `for` de java y C++:

```
método bucleDesde()
{
   Entero i;
   desde(i=1; i<=10; i++)
   {
      Sistema.imprime("esta es la iteración " + i + nl);
   }
}
```

Dentro de la primera parte de un bucle `desde` solamente puede haber expresiones; no puede haber declaraciones de variables.

3.4.4: Condicionales: si

En LeLi solamente hay una instrucción condicional, llamada "`si`". Su sintaxis es una mezcla entre la de su homónima en LEA[26] y la de la instrucción `if` de java y C.

La estructura de la instrucción si es la siguiente:

```
Si(condicion1){ instrucciones1 }
...
| (condicionN){ instruccionesN }
| otras { instrucciones_otras }
```

Solamente las dos primeras partes (`si(condicion){instrucciones}`) son necesarias. Al cuerpo principal se le pueden añadir tantas comprobaciones adicionales como se quieran.. La partícula "`otras`" puede no aparecer, pero en caso de que aparezca debe hacerlo una sola vez y al final.

En LeLi es **obligatorio** utilizar las llaves ({}) para delimitar las instrucciones a ejecutar al cumplirse una condición en la instrucción *si*.

3.4.5: Instrucción volver

Para salir de un método sin haber llegado al final de dicho método puede utilizarse la palabra reservada volver:

```
método Entero métodovolver(Booleano quiero1)
{
   Si(quiero1)
   {
      métodovolver= 1;
      volver;
   }
   métodovolver= 2; // Si la ejecución llega hasta aquí, se devuelve un 2
}
```

26 El lenguaje que se enseña en los primeros cursos de ingeniería informática en la Escuela Técnica Superior de Sevilla

Sección 3.5: Otros aspectos de LeLi

3.5.1: Herencia de clases

LeLi implementa un tipo muy básico de herencia, parecido al de PHP4.

Para que una clase herede de otra basta con especificarlo de esa forma en la declaración de la clase, utilizando la palabra reservada `extiende`:

```
clase A extiende B
{
...
}
```

Todos los métodos de la clase B pasarán automáticamente a formar parte de la clase A (los constructores no). La clase A puede sobreescribir cualquier método de B:

```
clase B
{
   método m() {Sistema.imprime("m en clase B" + nl);}
}

clase A extiende B
{
   método m() {Sistema.imprime("m en clase A" + nl);}
}

clase Inicio
{
   método abstracto inicio()
   {
      A a; B b;
      a.m(); // Imprime "m en clase A"
      b.m(); // Imprime "m en clase B"
      convertir(a,B).m(); // Imprime "m en clase B"
      a.super.m(); // Imprime "m en clase B"
   }
}
```

Cuando se necesita llamar a un método de la superclase se utiliza la palabra reservada `super`. Ésta puede ser empleada de dos maneras: la primera es como acabamos de ver en el ejemplo anterior: simplemente se sigue el nombre de la variable de un punto y super. El otro se utiliza cuando es el objeto actual el que precisa llamar a un método del padre. En tal caso, super se usa directamente, sin escribir nada delante. esta capacidad es especialmente útil en los constructores de las clases:

```
clase A
{
   atributo Entero x;
   constructor(Entero x) {atributo.x = x;}
}

clase B extiende A
{
   atributo Cadena n;
   constructor (Entero x, Cadena n)
   {
```

```
      super.constructor(x); // Inicia x
      atributo.n = n;
   }
}
```

La herencia solamente se aplica a los métodos abstractos y no abstractos; los atributos no pueden "sobreescribirse", porque en general esta capacidad solamente añade confusión al lenguaje:

```
clase Base
{
   atributo Entero a;
}

clase Heredada extiende Base
{
   atributo Cadena a; // Provoca un error: el atributo "a" ya existe
}
```

Raíz del sistema: clase Objeto

Todas las clases del sistema son subclases de la clase `Objeto`. Esta clase solamente dispone de un método abstracto, `aserto`, que sirve para detener la aplicación en tiempo de ejecución si una condición no se cumple. Los parámetros que admite aserto son dos: una condición booleana y un mensaje de texto en forma de cadena:

```
clase Objeto
{
   método abstracto aserto(Booleano condición, Cadena mensaje)
}
```

Gracias a la herencia, aserto puede ser utilizada directamente en cualquier método de cualquier classe de LeLi (pues todas las clases del lenguaje son en última instancia sub clases de `Objeto`).

3.5.2: Gestión de la memoria

No se especifica ningún mecanismo de gestión de la memoria; dado el carácter académico del lenguaje, no supone ninguna ventaja incluir un sistema de recolección de basura avanzado (ya que solamente se declararán una o dos clases en cada ejemplo, con 10 o 20 variables como máximo). Aunque el lector es libre de intentarlo.

Como norma general, se recomienda utilizar la pila para todas las variables (la no existencia de punteros permite utilizarla fácilmente) excepto para las cadenas, que deberían implementarse con un pseudo-montículo. Los atributos abstractos de las clases pueden guardarse en la pila o en una zona de datos especial.

Sección 3.6: Conclusión

En este capítulo he definido de una manera más o menos estricta el lenguaje LeLi, que usaré para ilustrar cómo crear un compilador con ANTLR. Esto es exactamente lo que haré en los próximos capítulos.

Un pequeño avance: como señalé en el capítulo 2, con ANTLR se trabaja utilizando una serie de ficheros de definición de gramáticas. Aunque es posible escribir el analizador léxico, sintáctico y semántico en el mismo fichero, es mucho más flexible tener varios[27] ficheros separados. Ésta será la estrategia que seguiremos para construir un compilador para LeLi.

Cada uno de los siguientes capítulos tendrá como objetivo principal explicar la construcción de cada uno de dichos ficheros, y explicar cómo hacerlo de la mejor manera posible (en el caso de que haya varias).

27 Dos, tres o *más*.

Capítulo 4:
Análisis léxico de LeLi

"Cuando hables, procura que tus palabras sean mejores que el silencio."
Proverbio hindú

Capítulo 4:
Análisis léxico de LeLi.....91

Sección 4.1: Introducción.....92
Sección 4.2: Estructura general del analizador.....93
4.2.1: Cuerpo del fichero.....93
4.2.2: Header: El paquete leli.....93
4.2.3: Opciones del analizador.....93
4.2.4: Zona de tokens.....94
Sección 4.3: Zona de reglas.....96
4.3.1: ¡Argh!.....96
Pero... ¡Nosotros siempre utilizábamos un autómata para el análisis léxico!.....96
4.3.2: Blancos.....97
Opciones dentro de una regla.....98
Reglas protegidas.....99
4.3.3: Identificadores.....99
4.3.4: Símbolos y operadores.....101
4.3.5: Enteros y reales.....102
4.3.6: Comentarios.....102
4.3.7: Literales cadena.....106
Sección 4.4: Compilando el analizador.....107
Sección 4.5: Ejecución: presentación de la clase Tool.....108
4.5.1: Introducción.....108
4.5.2: Clase Tool imprimiendo el flujo "tokens".....108
4.5.3: Ejemplo.....109
Sección 4.6: Añadiendo el nombre de fichero a los tokens.....110
4.6.1: La incongruencia de antlr.CommonToken.....110
4.6.2: Tokens homogéneos y heterogéneos.....110
Primera clase de antlraux: LexInfoToken.....111
La interfaz LexInfo.....111
4.6.3: Modificaciones en la clase Tool.....112
4.6.4: Segundo problema.....113
Sección 4.7: El fichero LeLiLexer.g.....116
Sección 4.8: Conclusión.....120

Sección 4.1: Introducción

En este capítulo y los siguientes voy a escribir, completamente desde 0, todos y cada uno de los ficheros de especificación de gramática que serán necesarios para crear nuestro compilador de LeLi. Dado que voy a hablar continuamente de salidas por pantalla, compilaciones y demás, es muy conveniente que el lector posea ANTLR instalado en su sistema, y codifiquemos a la vez estos ficheros.

Es un buen momento para que el lector revise su instalación de ANTLR. Tanto en este capítulo como en los siguientes voy a utilizar la última versión disponible (2.7.2).

Para poder compilar los ficheros correctamente puede ser necesario recompilar ANTLR para que admita el juego de caracteres extendido del español(con versiones previas a la 2.7.2). Explico el proceso de internacionalización en el apéndice C.

Hay dos decisiones a tomar en cuanto a los lenguajes empleados, tanto el natural como el de programación.

ANTLR proporciona la posibilidad de generar compiladores en Java, C++ y C#[28]. Por indicación de mi tutor de proyecto el lenguaje utilizado será Java.

Un compilador genera código ensamblador, que se ensambla para convertirse en código máquina. En mi caso prescindiré de generar código alguno (así que verdaderamente lo que generaremos será un intérprete, no un compilador). No obstante el capítulo 8 está dedicado a la generación de código.

El lenguaje natural a emplear también es problemático: tenemos que generar y tratar un "lenguaje en español" (LeLi) utilizando "lenguajes en inglés" (java y el propio ANTLR). No podemos, por ejemplo, nombrar una clase "AnalizadorSintáctico", pues java no admitiría el carácter acentuado. Por lo tanto he seguido las siguientes reglas:

1. Todos los elementos escritos en los ficheros de definición de gramática de ANTLR estarán en español, **excepto los nombres de tokens y reglas**. Éstos se convierten en variables de interfaces y métodos java, que no pueden utilizar caracteres no ASCII. Estarán en español, pero sustituyendo los caracteres "españoles" por otros ASCII.
2. El código (nombres de variables, comentarios...) de las clases java que haya que crear para ayudar exclusivamente al compilador de LeLi (paquete `leli`) estarán en español (eliminando caracteres especiales cuando sea necesario), y las que puedan servir para hacer otros compiladores (como las que integran el paquete `antlraux`) estarán en inglés.
3. Los nombres de los ficheros estarán en inglés.
4. Los ficheros de ejemplo escritos en LeLi estarán en español.

28 En fase de pruebas.

Sección 4.2: Estructura general del analizador

4.2.1: Cuerpo del fichero

El fichero que escribiremos en este capítulo se llamará `LeLiLexer.g`. Su estructura será parecida a la que ya presentábamos en el capítulo 2:

```
header{
   /* opciones de cabecera */
}
class LeLiLexer extends Lexer;
options {
   /* opciones de generación para este analizador*/
}
tokens {
   /* Sección de tokens */
}
/* Reglas de generación */
```

4.2.2: Header: El paquete leli

He organizado las clases que utilizaré para el compilador en varios paquetes; en el paquete leli irán todos los analizadores del lenguaje, y la clase que permitirá invocar las diferentes etapas del análisis.

Todos los analizadores deberán, por lo tanto, incluirse dentro del paquete leli. Esto se realiza normalmente en la cabecera (header) de cada fichero:

```
header
{
   package leli;
}
```

Todos los analizadores para LeLi formarán parte del paquete leli, así que todas las cabeceras de analizadores incluirán esta línea de código.

4.2.3: Opciones del analizador

Tras la cabecera existe la posibilidad de especificar opciones globales para todo el fichero. La única opción que por ahora es válida en dicha sección es la opción languaje, que permite especificar el lenguaje nativo en el que será generado el analizador (java,C++,C#). Como el lenguaje por defecto es java, y es el que vamos a utilizar, no especificaremos ninguna opción global (omitiendo esta sección) y pasaremos directamente a especificar las opciones relativas al analizador léxico. Éstas se encuentran inmediatamente después de la declaración del analizador, que tiene la siguiente forma:

```
class LeLiLexer extends Lexer;
```

La zona de opciones del analizador comienza con la palabra reservada *options* seguida de una llave abierta ({) y termina con una llave cerrada (}). Las opciones que vamos a utilizar son las siguientes:

```
options
{
   charVocabulary = '\3'..'\377';
   exportVocab=LeLiLexerVocab;
   testLiterals=false;
   k=2;
}
```

La primera opción que se modifica es `charVocabulary`. Esta opción indica qué juego de caracteres serán válidos para el analizador. En el caso de LeLi el rango de caracteres está entre el carácter unicode 3 y el 377, lo que permite todos los caracteres ASCII mas todas las letras acentuadas con todos los acentos posibles, mas algunos símbolos especiales de puntuación.

La segunda opción, `exportVocab`, es la que permite, en conjunción con su compañera, importVocab, separar en diferentes ficheros diferentes analizadores que se comunican (ver el apartado de opciones en el nivel sintáctico).

La siguiente opción es `testLiterals`. Esta opción sirve para manejar las palabras reservadas del lenguaje de programación. Cuando está activada, antes de aplicarse una regla se comprueba si existe una palabra reservada (o token[29]) que concuerde con la nueva palabra.

Por último, k sirve para modificar el lookahead "por defecto" del analizador. Lo hemos hecho 2 para poder prescindir de algunos predicados sintácticos que sería muy tedioso escribir.

Las opciones `charVocabulary` y `testLiterals` son exclusivas de los analizadores léxicos. `ExportVocab` y k son comunes a todos los tipos de analizadores.

4.2.4: Zona de tokens

Al contrario que flex, ANTLR proporciona una manera estándar de especificar las palabras reservadas de un lenguaje y su token asociado. Para ello se utiliza la sección de tokens del analizador léxico. Así es como comienza la sección de tokens de nuestro analizador léxico:

```
tokens
{
// Tipos basicos
   TIPO_ENTERO   = "Entero"   ;
   TIPO_REAL     = "Real"     ;
   TIPO_BOOLEANO = "Booleano" ;
   TIPO_CADENA   = "Cadena"   ;
```

Como vemos esta parte no es muy complicada: se define una palabra reservada por cada tipo básico del lenguaje de programación.

```
// Literales booleanos
   LIT_CIERTO = "cierto" ; LIT_FALSO = "falso" ;

// Literales cadena
   LIT_NL = "nl"; LIT_TAB = "tab" ; LIT_COM = "com";
```

Con estas dos líneas se definen los literales booleanos y las palabras reservadas que LeLi utiliza en lugar de las secuencias de escape en las cadenas.

29 Para más información sobre palabras reservadas y tokens, lea un poco más adelante la sección "tokens".

```
// Palabras reservadas
   RES_CLASE       = "clase"       ;
   RES_EXTIENDE    = "extiende"    ;
   RES_METODO      = "método"      ;
   RES_CONSTRUCTOR = "constructor" ;
   RES_ATRIBUTO    = "atributo"    ;
   RES_ABSTRACTO   = "abstracto"   ;
   RES_PARAMETRO   = "parámetro"   ;
   RES_CONVERTIR   = "convertir"   ;
   RES_MIENTRAS    = "mientras"    ;
   RES_HACER       = "hacer"       ;
   RES_DESDE       = "desde"       ;
   RES_SI          = "si"          ;
   RES_OTRAS       = "otras"       ;
   RES_SALIR       = "volver"      ;
   RES_ESUN        = "esUn"        ;
   RES_SUPER       = "super"       ;
```

En esta parte se define las palabras reservadas. Como ya hemos dicho, LeLi es un lenguaje sensible a las mayúsculas. Atención a los acentos en “método” y “parámetro”.

```
// Operadores que empiezan con letras;
   OP_Y          = "y"  ;
   OP_O          = "o"  ;
   OP_NO         = "no" ;
```

A pesar de ser operadores lógicos, hemos de tratar a “`y`”, “`o`” y “`no`” como palabras reservadas en lugar de operadores, porque empiezan por una letra, y generarían conflictos con la regla `IDENT` expuesta más abajo.

```
// Los literales real y entero son "devueltos" en las
// acciones del token LIT_NUMERO
   LIT_REAL ; LIT_ENTERO;
}
```

Finalmente hemos declarado dos tokens “imaginarios”. Al trabajar con ANTLR he tenido ocasión de trabajar muchas veces con este tipo de tokens, especialmente al hacer el análisis sintáctico. Los tokens imaginarios son tokens ficticios que se utilizan por comodidad. En éste caso los declaramos aquí para poder devolverlos adecuadamente con la regla LIT_NUMERO (siga leyendo para ver de qué estoy hablando). Los tokens imaginarios nunca tienen nombre, es decir, no van sucedidos de un igual y una cadena.

Los lectores observadores se han dado cuenta de que no he hecho referencia alguna a los comentarios que aparecen; como ya he indicado en otras partes del texto, éstos pueden escribirse en cualquier lugar del código, y su sintaxis es idéntica a la de java y C.

Sección 4.3: Zona de reglas

4.3.1: ¡Argh!

Ya lo he anunciado anteriormente, pero lo volveré a escribir aquí: Una de las primeras sorpresas que se encuentran al escribir el primer analizador léxico con ANTLR es que no se utiliza un autómata finito determinista (DFA, *Deterministic Finite Authomata*) para realizar el análisis, como en flex. En lugar de ello se utiliza un autómata pred-LL(k) que utiliza caracteres como símbolos.

Cuando me dí cuenta de ésto lo primero que pensé fue "¡Argh!". Ya me veía haciendo factorizaciones por la izquierda, inmerso en las tormentosas aguas de LL(k)...

Pensando en los que, como yo, sufrieran escalofríos al oír la idea, Terence Parr incluyó un apartado llamado "Pero... ¡Nosotros siempre utilizábamos un autómata para el análisis léxico!"[30] en el manual de ANTLR.Voy a permitirme incluir una traducción de dicha sección aquí.

Pero... ¡Nosotros siempre utilizábamos un autómata para el análisis léxico!

Traducido del manual en inglés de ANTLR. Fichero lexer.htm.

Todos los analizadores léxicos eran construidos a mano en los inicios de los compiladores hasta que los Autómatas Finitos Deterministas (DFAs[31]) se impusieron. Los DFAs tienen ciertas ventajas:

- Pueden construirse fácilmente a partir de simples expresiones regulares.
- Pueden hacer factorizaciones a la izquierda automáticamente de los prefijos. En un analizador léxico hecho a mano, hay que encontrar y factorizar todos los prefijos comunes. Por ejemplo, considérese un analizador que acepte enteros y flotantes. Las expresiones regulares son directas:

```
entero : "[0-9]+" ;
real   : "[0-9]+{.[0-9]*}|.[0-9]+" ;
```

Escribir un escáner para ésto requiere factorizar el [0-9]+ común. El código que habría que escribir sería de éste estilo:

```
Token nextToken() {
  if ( Character.isDigit(c) ) {
    match(entero)
    if ( c=='.' ) {
      match(entero)
      return new Token(REAL);
    }
    else {
      return new Token(ENTERO);
    }
  }
  else if ( c=='.' ) { // Real que comienza con punto
    match(entero)
    return new Token(REAL);
  }
  else ...
```

30 "But...We've Always Used Automata For Lexical Analysis!"
31 N del T: DFA, *Deterministic Finite Authomata*

```
}
```

Alternativamente, los analizadores escritos a mano tienen las siguientes ventajas sobre los implementados sobre un DFA:

- No están limitados a los tipos normales de lenguajes. Pueden utilizar información semántica y llamar a métodos durante el reconocimiento, mientras que un DFA no tiene pila y se caracteriza por no poder tener predicados semánticos.
- Los caracteres Unicode (16 bit values) son manejados fácilmente mientras que los DFAs usualmente están limitados a caracteres de 8 bits.
- Los DFAs son tablas de enteros y son, por lo tanto, difíciles de examinar y mantener.
- Un analizador hecho a mano, si está bien realizado, puede ser más rápido que un DFA.

Así que, ¿qué solución propone ANTLR 2.xx? ¡Ninguna de las dos! ANTLR permite especificar elementos léxicos con expresiones, pero genera un analizador que se parece mucho a uno hecho a mano. La única pega es que sigue siendo necesario hacer la factorización a la izquierda en algunas definiciones de tokens (pero al menos ésto se hace con expresiones, y no con código). Esta aproximación híbrida permite construir analizadores léxicos que son mucho más rápidos y potentes que los basados en DFAs, evitando tener que escribir los analizadores a mano.

En resumen, especificar expresiones regulares es más simple y más corto que escribir un analizador a mano, pero éstos últimos son más rápidos, potentes, capaces de manejar unicode y más fáciles de mantener. Este análisis ha llevado a muchos programadores a escribir analizadores léxicos a mano a pesar de la existencia de herramientas de generación basadas en DFA como lex y dlg. Como justificación final, nótese que escribir analizadores léxicos es trivial comparado con construir analizadores sintácticos; y que, una vez que haya construido su primer analizador léxico, usted lo reutilizará en el futuro añadiéndole diversas modificaciones.

*****

Una vez justificado el uso del algoritmo LL(k) para el análisis léxico, vamos a lo importante: las reglas. Las reglas se colocan tras la sección de tokens, y no terminan hasta que no se termina el fichero o aparece otra declaración de analizador. La zona de reglas no está delimitada por "`rules{ }`" ni nada parecido.

4.3.2: Blancos

Usualmente la primera regla que se define en un analizador léxico es la de los caracteres de separación, también llamados espacios en blanco o, simplemente, blancos.

Los blancos en LeLi son el espacio, el carácter de tabulación y el salto de línea. Los dos primeros son sencillos de implementar, mientras que el segundo requiere algo más de trabajo.

La regla que permite identificar los blancos es la siguiente:

```
BLANCO :
   ( ' '
   | '\t'
   | NL
   ) { $setType(Token.SKIP); } // La acción del blanco: ignorar
   ;
```

Una de las primeras cosas que resalta es el comentario. ANTLR permite insertar comentarios tipo javadoc delante de sus analizadores y delante de cada regla del analizador.

La alternativa es fácilmente legible: un espacio, un tabulador o un `NL` (Nueva Línea). Lo que es

un poco menos legible es la acción que viene inmediatamente después de la alternativa.

`$setType` es una función que permite cambiar el tipo de token que devuelve la regla en la que se encuentra. Si hubiéramos escrito "`$setType(LIT_CIERTO);`", por ejemplo, el analizador léxico devolvería `LIT_CIERTO` por cada blanco que encontrase. `Token.SKIP` es un valor especial que puede darse al tipo del token que indica al analizador que dicho token debe ser filtrado, es decir, no debe ser pasado al analizador sintáctico.

Ya casi hemos definido qué son los blancos. Para definirlo completamente es necesario que definamos `NL`.

Como ya hemos dicho los saltos de línea son algo más complicados que los otros dos separadores. Para empezar, el analizador debe "contar"cuántos saltos de línea ha pasado, sobre todo de cara a la emisión de mensajes de error. Para ello el analizador cuenta con el método newline(), que debe ser añadido dentro de una acción después de cada salto de línea.

Ahora bien, el concepto de "salto de línea" cambia de sistema operativo en sistema operativo. En los sistemas operativos de UNIX se representa con el carácter '\n'. En los sistemas Macintosh es '\r', y en los sistemas operativos y similares de Microsoft es la secuencia de caracteres "\r\n". De esta manera, hay que diferenciar los casos en los que hay "un \n en solitario", "un \r en solitario" y "un \r seguido de \n". En teoría la regla que habría que escribir sería la siguiente:

```
NL :
   (
     "\r\n"  // MS-DOS
   | '\r'    // MACINTOSH
   | '\n'    // UNIX
   ) { newline(); }
   ;
```

Por desgracia esta regla es incoherente en un analizador LL(k), para cualquier k. ANTLR se encargará de recordárnoslo si intentamos compilar una gramática con esta regla. Afortunadamente hay una manera maravillosamente simple de solucionar el problema. Los que hayan leído el capítulo 1 ya sabrán cuál es: un predicado sintáctico.

```
protected NL :
   (
     ("\r\n") => "\r\n"  // MS-DOS
   | '\r'                // MACINTOSH
   | '\n'                // UNIX
   ) { newline(); }
   ;
```

Añadiendo dicho enunciado a la opción se está forzando al analizador a intentarlo primero con "\r\n", y si no lo consigue, considerar las otras dos opciones. Si no entiende qué está pasando aquí, necesita releer el primer capítulo.

Opciones dentro de una regla

Existe otra manera de implementar los saltos de línea, muy extendida, que a mí no me gusta demasiado. Consiste en utilizar la capacidad de ANTLR para definir opciones dentro de alternativas de una regla.

La sintaxis para definir dichas excepciones es realmente muy engorrosa. Es algo así:

```
regla : alternativa1
      | ( options {
            nombre1=valor1;
            nombre2=valor2;
            ...
          }
          alternativa2 )
     ...
      | alternativaN
      ;
```

Lo curioso del asunto es que ANTLR ofrece un comportamiento "por defecto" cuando se encuentra con una regla como la anterior: consume caracteres lo más pronto posible (o reconoce la primera alternativa listada). Si se desactivan los warnings, la regla se reconoce perfectamente. Para desactivarlos basta con utilizar la opción `generateAmbigWarnings`:

```
NL :
   ( options {generateAmbigWarnings=false;}:
     "\r\n"  // MS-DOS
   | '\r'    // MACINTOSH
   | '\n'    // UNIX
   ) { newline(); }
   ;
```

Esta manera de implementar los saltos de lína se extiende por la mayoría de los ficheros de ejemplo de ANTLR[32]. Esta regla funciona más rápido que la anterior, al no tener que ejecutar un predicado sintáctico. No obstante prefiero solucionar las ambigüedades en lugar de ocultarlas, así que me decanto por mi solución.

Reglas protegidas

Tanto BLANCO como NL son reglas léxicas con una característica interesante: el analizador sintáctico que se conecte a nuestro LeLiLexer jamás tendrá constancia de estas reglas, pues no producen ningún token; son de uso "interno" del analizador léxico.

ANTLR permite optimizar este tipo de reglas en los analizadores léxicos con la palabra reservada `protected`. Ésta se coloca delante del nombre de la regla, así:

```
protected NL :
   ( options {generateAmbigWarnings=false;}:
     "\r\n"  // MS-DOS
   | '\r'    // MACINTOSH
   | '\n'    // UNIX
   ) { newline(); }
   ;
```

Las reglas protegidas son optimizadas para no crear tokens, de manera que el análisis sea más rápido.

Tanto `BLANCO` como `NL` deberán ir precedidos de protected, al igual que otras reglas internas que ya iremos viendo.

4.3.3: Identificadores

Un identificador en LeLi es una letra o carácter de subrayado seguidos de una secuencia (que puede ser vacía) de letras, dígitos y caracteres de subrayado. Empecemos por definir qué es una

32 Que ofrecen soluciones para casi cualquier problema de reconocimiento que se pueda presentar.

letra.

```
/**
 *  Letras españolas.
 */
protected LETRA
   : 'a'..'z'
   | 'A'..'Z'
   | 'ñ' | 'Ñ'
   | 'á' | 'é' | 'í' | 'ó' | 'ú'
   | 'Á' | 'É' | 'Í' | 'Ó' | 'Ú'
   | 'ü' | 'Ü'
   ;
```

La regla en cuestión no es difícil de entender: una letra es cualquier letra del abecedario español, incluyendo la ñ y las vocales acentuadas, en mayúsculas y minúsculas. Nótese que al especificar una regla de analizador léxico es posible utilizar el operador doble punto (..) para referenciar rangos de caracteres. Es posible utilizar caracteres unicode para representar los caracteres, sean éstos ASCII o no-ASCII; por ejemplo, puede utilizarse '\u00f1' o '\241' en lugar de 'ñ', pero utilizar este último es mucho más intuitivo[33]. Nótese que `LETRA` es una regla protegida.

Una vez definidas las letras definamos los dígitos:

```
protected DIGITO : '0'..'9';
```

Nota: si no fuera por que hemos hecho k=2 en las opciones del analizador, las reglas `LETRA` e `IDENT` entrarían en conflicto, así como `DIGITO` y `RES_NUMERO` (ver más abajo)

De nuevo una regla privada, aún más sencilla que la anterior. Pasemos, por fin, a los identificadores:

```
IDENT
options {testLiterals=true;} // Comprobar palabras reservadas
:
   (LETRA|'_') (LETRA|DIGITO|'_')*
;
```

El cuerpo de la regla es la transcripción exacta de lo que decíamos al principio de la sección : una letra o carácter de subrayado seguida de cero o más letras, dígitos o caracteres de subrayado.

La parte interesante de esta regla no está en el cuerpo, sino antes: como puede verse, es posible especificar opciones individuales para cada regla de la gramática (además de para las sub reglas, como vimos en los `BLANCOS`). En este caso la opción `testLiterals` (que se desactivó para el analizador en general) se ha activado puntualmente en esta regla. Cuando esta opción está activa, el analizador actúa de la siguiente forma:

1. El analizador encuentra un token que satisface el cuerpo de la regla (en nuestro caso, un identificador)
2. El analizador comprueba si tiene en su lista de tokens algún token cuyo texto coincida con el de la regla que acaba de hacer coincidir. De ser así, se envía dicho token al analizador sintáctico y se da por concluida la regla.
3. En caso contrario, se procede normalmente (en nuestro caso, enviando un identificador).

33 Para poder utilizar esta notación puede ser necesario recompilar ANTLR para internacionalizarlo. Consulte el Apéndice B.

4.3.4: Símbolos y operadores

Empezaremos por definir las reglas que permiten al analizador reconocer los operadores (como suma, multiplicación, etc) y los símbolos de separación (paréntesis, llaves, etc.). Esta es probablemente la parte más fácil de hacer de todo el analizador.

Para definir los separadores utilizaremos las siguientes reglas:

```
// Separadores
PUNTO_COMA   : ';' ;
COMA         : ',' ;
CORCHETE_AB  : '[' ;
CORCHETE_CE  : ']' ;
LLAVE_AB     : '{' ;
LLAVE_CE     : '}' ;
PUNTO        : '.' ;
PARENT_AB    : '(' ;
PARENT_CE    : ')' ;
BARRA_VERT   : '|' ;
```

Los operadores, por su parte, se representan así:

```
//  operadores
OP_IGUAL       : "==" ;
OP_DISTINTO    : "!=" ;
OP_ASIG        : '='  ;
OP_MENOR       : '<'  ;
OP_MAYOR       : '>'  ;
OP_MENOR_IGUAL : "<=" ;
OP_MAYOR_IGUAL : ">=" ;
OP_MAS         : '+'  ;
OP_MENOS       : '-'  ;
OP_MASMAS      : "++" ;
OP_MENOSMENOS  : "--" ;
OP_PRODUCTO    : '*'  ;
OP_DIVISION    : '/'  ;
```

Nótese que no es necesario alinear los símbolos de dos puntos ni de punto y coma; lo he hecho así por pura estética, y porque alineando el código de esta forma es más fácil comprobar si falta o sobra algún punto y coma o algún paréntesis. Es posible incluso definir varias reglas en la misma línea, cosa que yo desaconsejo fervientemente.

La declaración de separadores y operadores es una tarea sencilla que no requiere mucha explicación. Solamente vamos a explicar dos pequeños detalles.

Primero: Dado que algunos operadores (como + y ++) comienzan con el mismo carácter, es necesario que el lookahead por defecto del analizador sea >= 2 (de otra manera habría sido necesario utilizar tokens imaginarios y predicados sintácticos, quedando el código realmente sucio)

Segundo: Incluir los operadores que empiezan con letras (y, o, no) en la sección de tokens y declarar los demás como reglas no es una decisión fortuita; se ha realizado así porque las palabras reservadas tienen una regla para dispararse (en el caso de LeLi dicha regla será IDENT; sigua leyendo para más información). No obstante no hay ninguna regla equivalente para los operadores - Ni debe haberla: una regla como la siguiente:

```
tokens{
   OP_MAS="+";
   OP_MENOS="-";
...
}
...
OPERADOR options {testLiterals=true}: ('*' | '+' | '-' | ...)+ ;
```

Funcionaría bien mientras la entrada fuera alguno de los operadores válidos. Sin embargo, un operador no válido (por ejemplo =>) no provocaría el lanzamiento de una excepción, sino que lanzaría el token imaginario `OPERADOR` al analizador léxico (como ocurre con `IDENT`).

Este inconveniente podría solucionarse haciendo que la regla `OPERADOR` "fracasara siempre" (por ejemplo, lanzando una excepción en una acción). Pero creo que es mucho más sencillo implementar los operadores como yo lo he hecho.

4.3.5: Enteros y reales

Aprovechando que estamos hablando de tokens imaginarios, vamos a volver sobre los dos tokens imaginarios que hemos definido en la sección de tokens. Para los que no tengan ganas de revolver páginas, los vuelvo a escribir aquí:

```
// Los literales real y entero son "devueltos" en las
// acciones del token LIT_NUMERO
   LIT_REAL ; LIT_ENTERO;
```

El comentario que los precede debería ahora resultar más comprensible que antes. Vamos a escribir una regla (`LIT_NUMERO`) que sea capaz de diferenciar los enteros de los reales, y de devolverlos convenientemente.

La regla en cuestión es la siguiente:

```
/**
 * Permite reconocer literales enteros y reales sin generar conflictos.
 */
LIT_NUMERO : (( DIGITO )+  '.' ) =>
                ( DIGITO )+ '.' ( DIGITO )* { $setType (LIT_REAL); }
           | ( DIGITO )+  { $setType (LIT_ENTERO); }
           ;
```

La regla es en realidad una opción con dos alternativas muy simples. Lo único que hay que mencionar es que la primera alternativa comienza con un **predicado sintáctico**. Éste predicado amplía indefinidamente el lookahead del analizador cuando encuentra una secuencia de dígitos, hasta que encuentra el carácter siguiente. Si dicho carácter es un punto, entonces se trata de un real (se cumple la primera regla). En otro caso es un entero, por lo que la regla que se cumple es la segunda.

Ya hemos visto el método `$setType` al hablar de los `BLANCO`s. En éste caso sirve para que el analizador devuelve un token de tipo `LIT_ENTERO` o `LIT_REAL` en lugar de `LIT_NUMERO`.

4.3.6: Comentarios

Es posible especificar dos tipos de comentarios en LeLi: de una sola línea y de varias líneas. Los comentarios serán totalmente ignorados a partir éste momento, así que deberán ser filtrados de la misma forma que los blancos.

Observemos, pues, la regla que permite reconocer los comentarios de una sola línea:

```
/**
 * Comentario de una sola línea
 */
protected COMENTARIO1
   :
      "//" (~('\n'|'\r'))*
      { $setType(Token.SKIP); }
   ;
```

Esta regla impone un poco al principio, pero analizándola poco a poco se puede entender sin dificultad. Comencemos por los extremos: la "//" del principio indica que un comentario debe empezar con la doble barra, mientras que la acción "`$setType(Token.SKIP);`" sirve para filtrar todo el token y que no llegue al analizador sintáctico.

Lo que queda entre la doble barra y la acción, `(~('\n'|'\r'))*`, quiere decir "cero o más veces `~('\n'|'\r')`". Literalmente, `~(expresión)` significa "cualquier elemento menos aquellos que cumplan expresión". En nuestro caso `expresión` es `'\n'|'\r'`, es decir, cualquier salto de línea.

Por lo tanto, lo que dice la regla es: "Un `COMENTARIO1` es la cadena "//" seguida de cero o más veces cualquier carácter que no sea un salto de línea. Esta regla debe filtrarse."

Pasemos sin más preámbulos a la construcción de una regla para los comentarios de varias líneas. En lugar de presentar mi solución y comentar lo que significa, en este caso voy a explicar cómo la construí, paso a paso.

Los comentarios multilínea van delimitados entre "/*" y "*/". La regla terminará también ejecutando la acción "`$setType(Token.SKIP);`". Por lo tanto, la regla tendrá la siguiente forma.

```
protected COMENTARIO2 :"/*" ??? "*/"   { $setType(Token.SKIP); };
```

Falta averiguar lo que hay que escribir en lugar de `???`. Podríamos pensar en que `???` es "una secuencia de cero o más caracteres cualesquiera", es decir:

```
protected COMENTARIO2 :"/*" (.)* "*/" { $setType(Token.SKIP); };
```

El punto (.) en ANTLR representa "cualquier carácter del vocabulario".

Pero las cosas no suelen ser tan fáciles. Si intentamos compilar la gramática con esa regla ANTLR emitirá un mensaje de error, indicando que "no hay determinismo léxico entre la primera alternativa y el final de la regla".

El problema es que no hemos planteado bien lo que hay entre las marcas de comentario: al decir "una secuencia de cero o más caracteres cualesquiera" estamos incluyendo la cadena del final del comentario, es decir "*/". Por lo tanto lo que deberíamos poner es "cualquier secuencia ce cero o más caracteres, **excepto la cadena de fin de comentario**". Es decir, algo así:

```
protected COMENTARIO2 :
   "/*"
   (
      ('*' ~'/') => '*' ~'/'
   | ~('*')
   )*
   "*/"
   { $setType(Token.SKIP); }
   ;
```

Utilizando un predicado sintáctico, y una disyunción puede conseguirse esta regla, que no es ambigua en absoluto. De hecho, si al analizador le pasamos una gramática con comentarios

multilínea, será capaz de reconocerlos y filtrarlos convenientemente. ¡Pero sigue habiendo un problema!.

Seguramente no nos daremos cuenta de él inmediatamente. El problema es el siguiente: El analizador no tiene en cuenta las líneas de los comentarios a la hora de hacer el conteo de líneas del fichero.

Mantener información sobre la línea actual que se está reconociendo es una tarea muy importante, porque influye notablemente en la claridad de los mensajes de error que daremos a los programadores de LeLi.

En el caso de `COMENTARIO1` no era necesario contar las líneas, pues se salía de la regla antes de llegar a los caracteres de fin de línea, que eran reconocidos en la regla `NL`, que se disparaba por `BLANCO` una vez que se acababa el `COMENTARIO1`, llamándose a newLine() cuando era necesario.

Ya hemos visto que contar el número de líneas no es trivial, y por eso lo hemos colocado en una regla aparte (`NL`). Tenemos que llamar a esa regla cuando sea necesario. Por ejemplo así:

```
protected COMENTARIO2  :
   "/*"
   |  ('*' ~('/')) => '*' ~('/')
   | NL
   | ~( '\n' | '\r' | '*' )
   )*
   "*/"
   { $setType(Token.SKIP); }
   ;
```

Hemos añadido `NL` como una alternativa más. Para que no hubiera conflictos, hemos tenido que quitar `PRIMERO(NL)` del predicado sintáctico de la tercera alternativa. El conteo de líneas parece efectuarse correctamente.

¡Por desgracia, aún queda un problema! Éste puede observarse al analizar el siguiente fichero:

```
/* Este ejemplo demuestra que probar todas las posibilidades es *MUY*
   importante. */

class Manolo { } // Error(línea 4); es "clase", no "class"
```

Hay un error en la línea 4. Misteriosamente, al compilar, el analizador señala que el error está en la línea 3.

Esto quiere decir que alguno de los saltos de línea anteriores a la línea 4 no ha disparado la regla `NL`. No pueden ser los de la línea 2 a la 3 y de la 3 a la 4, pues son `BLANCOS`, que funcionan correctamente. Por lo tanto solamente puede ser el comentario.

Efectivamente, hay un caso en el cual el comentario no ejecuta `newline()` al topar con un carácter de nueva línea: cuando éste va precedido por un asterisco. Esto ocurre por culpa de esta alternativa:

```
protected COMENTARIO2  :
   "/*"
   |  ('*' ~('/')) => '*' ~('/')
   | NL
   | ~( '\n' | '\r' | '*' )
   )*
   "*/"
   { $setType(Token.SKIP); }
   ;
```

Literalmente, lo que significa esta alternativa para el analizador léxico es "cuanto encuentres un asterisco seguido de *cualquier carácter que no sea la barra*, simplemente ignora ambos caracteres".

¡Pero "cualquier carácter que no sea la barra" incluye los caracteres de nueva línea! Así que tenemos que añadir una alternativa más, para el caso de un asterisco seguido de un carácter de nueva línea, asegurarnos de que los caracteres de nueva línea son reconocidos por esa nueva regla y no por la anterior. Para ello (y para evitar ambigüedades) deberemos modificar la regla expuesta más arriba. Una vez realizadas estas modificaciones, la regla quedará así:

```
protected COMENTARIO2  :
   "/*"
   ( ('*' NL) => '*' NL             // Nueva alternativa
   | ('*' ~('/'|'\n'|'\r')) => '*' ~('/'|'\n'|'\r') // Modificada
   | NL
   | ~( '\n' | '\r' | '*' )
   )*
   "*/"
   { $setType(Token.SKIP); }
   ;
```

Finalmente hay 4 alternativas. Éstas son:

1. Un asterisco seguido de un carácter de nueva línea.
2. Un asterisco seguido de cualquier carácter que no sea ni la barra (lo que generaría un conflicto con el final de la regla) ni '\n' o '\r' (lo que generaría un conflicto con la regla anterior).
3. Un carácter de nueva línea
4. Cualquier carácter que no sea ni asterisco, ni de nueva línea

Hay una pequeña aclaración a hacer en lo referente a los conjuntos `PRIMERO`: ¿por qué en los predicados sintácticos he utilizado '\n' y '\r' en lugar del mucho más natural `NL`? En teoría, podría haber hecho algo así:

```
protected COMENTARIO2 :
   "/*"
   ( ('*' NL) => '*' NL
   | ('*' ~('/'|NL)) => '*' ~('/'|NL) // NL en lugar de '\n'|'\r'
   | NL
   | ~( NL | '*' ) // NL en lugar de '\n'|'\r'
   )*
   "*/"
   { $setType(Token.SKIP); }
   ;
```

Ésto que parece tan lógico no se puede hacer por una pequeña limitación de ANTLR. ANTLR puede manejar bastante bien conjuntos negativos de "sólo caracteres" (como `~('a'|'b'|'c')`)

o de "sólo reglas" (como `~(IDENT|ENTERO)`), pero no de los dos a la vez (no puede manejar `~('a'|ENTERO)`). Por eso en el caso de tener que hacer predicados de éste tipo el programador tiene que trabajar un poco más y calcular el conjunto `PRIMERO` de las reglas que necesite. En nuestro caso es muy fácil, el conjunto `PRIMERO` de `NL` es `{'\n','\r'}`. Es siempre preferible utilizar reglas de "sólo caracteres", porque son más eficientes.

4.3.7: Literales cadena

Si usted ha conseguido entender las reglas de los comentarios, los literales cadena no deberían presentar problema alguno. Le recuerdo que LeLi no permite secuencias de escape; en lugar de ello están las palabras reservadas `nl` (para nueva línea), `tab` (para el carácter de tabulación) y `com` (para las comillas). Que no haya secuencias de escape simplifica la regla. Mi implementación ha sido la siguiente:

```
LIT_CADENA :
   '"' !
   ( ~('"'|'\n'|'\r'|'\t') )*
   '"' !
   ;
```

La regla no es difícil de entender: un literal cadena empieza y termina con unas comillas (""). Lo que hay entre estas dos comillas es, literalmente "cualquier carácter del vocabulario que no sea ni unas comillas (para evitar ambigüedades con el final de la cadena) ni un carácter de fin de línea, ni un tabulador".

Una pequeña aclaración para los observadores. Hay dos signos de admiración siguiendo a las comillas. El símbolo de admiración se utiliza en un analizador para filtrar. Es decir, el código anterior significa lo mismo que el siguiente:

```
LIT_CADENA :
   ('"' { $setType(Token.SKIP); } )
   ( ~('"'|'\n'|'\r'|'\t') )*
   ('"' { $setType(Token.SKIP); } )
   ;
```

El símbolo de admiración no solamente se puede utilizar en los analizadores léxicos; es aplicable en cualquier analizador. No obstante, su significado varía ligeramente dependiendo del tipo de analizador en el que se aplique : en una regla de un analizador léxico significa "este token no debe ser pasado al nivel sintáctico", mientras que en un analizador sintáctico o semántico significa "éste token no debe formar parte del árbol AST".

Sección 4.4: Compilando el analizador

Para compilar generar el analizador deberá situarlo en un directorio llamado `leli` (pues así se llama el paquete en el que se encuentra). Voy a suponer que el directorio leli se encuentra en `c:\languajes` (es decir, el path completo del paquete será `c:\languajes\leli`). Sitúese en `c:\languajes\leli` y escriba lo siguiente[34]:

```
c:\languajes\leli\> java antlr.Tool LeLiLexer.g
```

Si todo ha ido bien ANTLR no emitirá mensajes de error. En el directorio `leli` habrán aparecido los siguientes ficheros:

- `LeLiLexer.java`
- `LeLiLexerVocabTokenTypes.java`
- `LeLiLexerVocabTokenTypes.txt`

La función del primer fichero es evidente: es el fichero java que implementa el analizador (un analizador léxico recursivo descendente enriquecido con predicados pred-LL(k)).

¿Pero qué son los otros dos ficheros? Estos dos ficheros son los que permiten al analizador compartir su conjunto de tokens con el analizador sintáctico, que implementaremos después (más información sobre ésto en el siguiente capítulo).

Volvamos al fichero que nos interesa, `LeLiLexer.java`. Para que un fichero java nos sea útil, debe ser convertido a código compatible con la máquina virtual java, es decir, debe ser compilado. Dado que `LeLiLexer.java` necesita a `LeLiLexerVocabTokenTypes.java`, será necesario compilar los dos. El comando más fácil para hacerlo es el siguiente:

```
c:\languajes\leli> javac LeLiLexer*.java
```

El compilador de java no debería mostrar mensaje alguno, significando que todo ha ido bien.

34 Como siempre, para poder utilizar ANTLR igual que en los ejemplos será necesario que esté correctamente instalado. En el apéndice B explico cómo realizar la instalación.

Sección 4.5: Ejecución: presentación de la clase Tool

4.5.1: Introducción

Una vez implementado el nivel léxico del compilador, un buen paso es probarlo. Para poder hacerlo, sin embargo, necesitamos aún algo que aún no tenemos: necesitamos el método main(). Necesitaremos un equivalente a la clase `Calc` que vimos en el capítulo 2.

Siguiendo el ejemplo de la herramienta ANTLR, vamos a hacer que LeLi disponga de una "herramienta" *à la* `antlr.Tool`. En otras palabras, vamos a crear una clase que permita ejecutar nuestro analizador de la siguiente forma:

```
c:\home> java leli.Tool ejemplo.leli
```

La clase `leli.Tool` servirá para interpretar cualquier fichero escrito en LeLi. Para ello deberá manejar el análisis léxico, sintáctico y semántico, controlando los errores y finalmente generando código. Iremos añadiéndole capacidad al ir completando niveles.

Por ahora la funcionalidad de la clase Tool será muy limitada: se contentará con abrir el fichero de texto que se le pase por la línea de comandos, e imprimir un listado de los tokens que el analizador léxico sea capaz de reconocer.

4.5.2: Clase Tool imprimiendo el flujo "tokens"

Se pueden distinguir varios pasos importantes a realizar en esta primera implementación de la clase `leli.Tool`:

- Abrir el fichero cuyo nombre se suministra como primer argumento en la línea de comandos.
- Crear un analizador léxico que lea dicho fichero.
- Leer todos los tokens del fichero imprimiendo en la consola la información de cada token.

Además de estos tres pasos principales, existen otros pasos auxiliares que deberán darse, como la gestión de las excepciones que puedan producirse.

El código de la clase `Tool` será el siguiente:

```
package leli; // Tool también pertenece al paquete leli

import antlr.*;   // Incluir todo antlr
import java.io.*; // Para entrada-salida

public class Tool{
   public static void main(String [] args)
   {
      try
      {
         FileInputStream fis = new FileInputStream(args[0]);
         LeLiLexer lexer = new LeLiLexer(fis);
         lexer.setFilename(args[0]);

         Token tok = lexer.nextToken();
         while(tok.getType() != Token.EOF_TYPE)
         {
            System.out.println( tok.getFilename()+ ":" +
                                tok.getLine() +  ":" +
                                tok.getColumn() +  ": " +
```

```
                                tok.getText()+ ", " + tok.getType() );

            tok = lexer.nextToken();
         }

      }
      catch (FileNotFoundException fnfe)
      {
         System.err.println("No se encontró el fichero");
      }
      catch (TokenStreamException tse)
      {
         System.err.println("Error leyendo tokens");
      }
   }
}
```

4.5.3: Ejemplo

Para la siguiente entrada:

```
// Primer progama escrito en LeLi!

clase Inicio
{
   método inicio()
   {
      Sistema.imprime("Hola mundo!"+nl);
   }
}
```

La salida por pantalla será la siguiente:

```
null:3:1: clase, 13
null:3:7: Inicio, 35
null:3:14: extiende, 14
null:3:23: Objeto, 35
null:4:1: {, 38
null:5:9: método, 15
null:5:16: inicio, 35
null:5:22: (, 41
null:5:23: ), 42
null:6:9: {, 38
null:7:17: Sistema, 35
null:7:24: ., 40
null:7:25: imprime, 35
null:7:32: (, 41
null:7:33: ¡Hola mundo!, 62
null:7:47: +, 51
null:7:48: nl, 10
null:7:50: ), 42
null:7:51: ;, 36
null:8:9: }, 39
null:9:1: }, 39
```

Hay varios detalles interesantes a observar. El primero es que los tokens que debían ser filtrados (comentarios y blancos) no se muestran. El segundo detalle importante es que ¡los tokens no han conservado el nombre del fichero (se muestra "null")!

Sección 4.6: Añadiendo el nombre de fichero a los tokens

4.6.1: La incongruencia de antlr.CommonToken

`antlr.Token` es una clase abstracta; la clase que ANTLR utiliza por defecto para representar los tokens es `antlr.CommonToken`.

Hay dos maneras de ver los tokens. Por un lado, la única información "vital" que contienen es su texto y su tipo; si faltara cualquiera de los dos el análisis sintáctico no podría trabajar adecuadamente con los tokens – no habría manera de diferenciarlos.

Por comodidad, sin embargo, se ha dotado a la clase `Token` de una serie de métodos abstractos adicionales para obtener información adicional de tipo léxico, como son un número de línea y columna y un nombre de fichero.

La validez de incluir esta información adicional en la clase `Token` es discutible, porque implícitamente se está sobreentendiendo que un token procede de un fichero de texto, cuando ya hemos visto que podría proceder de cualquier flujo de datos (cualquier clase que implemente la interfaz `java.io.inputStream`), en el cual los conceptos "línea, columna y nombre de fichero" podrían no tener significado.

Este tipo de flujos de datos es, sin embargo, tan poco habitual, que por pragmatismo es más conveniente ignorarlo y suponer que todos los tokens poseerán un número de línea y columna y un nombre de fichero. Por eso la clase `antlr.Token` proporciona métodos (vacíos) para acceder a esta información. Las subclases serán las encargadas de sobreescribir dichos métodos para proporcionar la información.

Y esto es lo que se supone que tendría que hacer `antlr.CommonToken`; al ser la clase que ANTLR utiliza por defecto para implementar los tokens, debería de sobreescribir todos los métodos vacíos de `antlr.Token`.

¡Y sin embargo no es así! Por razones que desconozco, `antlr.CommonToken` no implementa los métodos ni atributos que permiten obtener el nombre de fichero del que proviene cada token; solamente proporcionan línea y columna. El método `setFilename()` no hace nada, y el método `getFilename()` devuelve siempre null – por eso aparecen nulls en el listado del ejemplo de la sección anterior.

4.6.2: Tokens homogéneos y heterogéneos

`antlr.CommonToken` no es la única clase que puede utilizarse para implementar los tokens en un analizador léxico. Existen mecanismos para utilizar cualquier otra subclase de `antlr.Token` definida por el usuario.

Nuestro objetivo será, por tanto, implementar una clase de tokens que sí contemple la posibilidad de guardar el nombre del fichero del que se leyó el token. Esta clase será la primera clase que incluiremos en el paquete antlraux.

La primera de ellas pasa por utilizar el método `setTokenObjectClass` en nuestro analizador léxico. Este método se llama tras crear el analizador léxico, así:

```
import enrique.tokens.MyToken;
...

MiLexer lexer = new MiLexer(new FileInputStream(nombreFichero));
lexer.setTokenObjectClass("enrique.tokens.MiToken");
```

Para utilizar este método es necesario especificar el nombre completo de la clase a utilizar (precediéndolo del prefijo de su paquete si está en un paquete java).

La clase que especifiquemos además debe cumplir dos condiciones:

- Ser subclase de `antlr.Token`.
- Tener accesible un constructor por defecto (sin parámetros)

Mientras que el primer método cambia el tipo por defecto de todos los tokens del analizador, el segundo puede modificar los tipos de los tokens expelidos por cada regla. Se Consiste en construir manualmente un token del tipo deseado en las acciones del analizador léxico y designarlo como producto de la regla utilizando el patrón `$setToken()`. Por ejemplo, supongamos un analizador léxico que lee enteros con la regla `ENTERO` y que se desea que la clase de los tokens devueltos por esta regla sea `enrique.tokens.EnteroToken`, pero que para el resto de las reglas se siga utilizando el tipo por defecto. El código necesario será:

```
header{
   import enrique.tokens.EnteroToken;
   ...
}
class MyLexer extends Lexer;
...
ENTERO: (DIGITO)+
{
   // getText() obtiene el texto de la regla
   EnteroToken et = new EnteroToken(ENTERO, getText());
   // $setToken designa un nuevo token como producto de la regla
   $setToken(et);
}
```

Las clases de tokens que se utilizan en este caso no precisan un constructor por defecto, aunque sí necesitan ser subclases de `antlr.Token`.

Cuando un analizador léxico genera tokens de diferentes clases se dice que está generando "tokens heterogéneos". Si todos son del mismo tipo, se llaman "homogéneos".

Primera clase de antlraux: LexInfoToken

Vamos a utilizar setTokenObjectClass para hacer que la clase por defecto de nuestros tokens no sea antlr.CommonToken, sino una que definiremos nosotros: `LexInfoToken`.

`LexInfoToken` será la primera clase que añadiremos a nuestro paquete de utilizades antlraux, del que ya hablamos en el capítulo 1. Estará dentro del subpaquete "util", así que su nombre completo será "antlraux.util.LexInfoToken".

Su implementación es muy sencilla: será una subclase de `CommonToken` con un atributo añadido dedicado a guardar el nombre de fichero. Además se sobreescribirán los métodos `setFilename` y `getFilename`.

La interfaz LexInfo

`LexInfoToken` implementará la interfaz `antlraux.util.LexInfo`. Esta interfaz contiene las siguientes definiciones de métodos:

```
public interface LexInfo{
   public int getLine();
   public int getColumn();
   public String getFilename();

   public void setLine(int line);
   public void setColumn(int column);
   public void setFilename(String filename);

   public void copyLexInfo(LexInfo other);
   public String getLexInfoString();
}
```

LexInfo representa cualquier objeto que permita identificar un punto cualquiera dentro de un fichero (con el nombre de fichero, línea y columna). Además de los usuales métodos de acceso (los tres "set" y los tres "get") hay dos métodos que facilitarán el trabajo con la interfaz:

- `copyLexInfo`: Que copia el nombre de fichero, línea y columna de otro LexInfo
- `getLexInfoString`: Que devuelve una cadena del estilo "nombrefichero:linea:columna"


Para poder utilizar antlraux será necesario incluirla en el classpath.

4.6.3: Modificaciones en la clase Tool

La clase `Tool` deberá modificarse para modificar la clase de tokens que usa el analizador léxico. El código de la nueva clase `Tool` será el siguiente:

```
package leli; // Tool también pertenece al paquete leli

import antlr.*;   // Incluir todo antlr
import java.io.*; // Para entrada-salida
import antlraux.util.LexInfoToken; // Nueva clase Token

public class Tool{
   public static void main(String [] args)
   {
      try
      {
         FileInputStream fis = new FileInputStream(args[0]);
         LeLiLexer lexer = new LeLiLexer(fis);
         lexer.setFilename(args[0]);
         lexer.setTokenObjectClass("antlraux.util.LexInfoToken");

         Token tok = lexer.nextToken();
         while(tok.getType() != Token.EOF_TYPE)
         {
            System.out.println( tok.getFilename()+ ":" +
                                tok.getLine() +  ":" +
                                tok.getColumn() +  ": " +
                                tok.getText()+ ", " + tok.getType() );

            tok = lexer.nextToken();
         }

      }
      catch (FileNotFoundException fnfe)
      {
```

```
            System.err.println("No se encontró el fichero");
         }
         catch (TokenStreamException tse)
         {
            System.err.println("Error leyendo tokens");
         }
      }
   }
```

4.6.4: Segundo problema

Una vez compilada la nueva clase Tool, la ejecutaremos sobre cualquier fichero de texto y ... ¡seguirán apareciendo "nulls" en lugar del nombre del fichero!

El problema es que aunque hemos resuelto un problema, queda otro por resolver: los nuevos tokens que se crean no almacenan el nombre del fichero. Si se coloca un mensaje de debug en el método `LexInfoToken.setFilename()`, veremos que éste no es invocado ni una sola vez.

La solución más inmediata consiste en añadir una acción a cada regla, invocando manualmente setFilename. Esta opción, sin embargo, no es la mejor.

Analicemos la situación. Lo que está ocurriendo es que, al crearse los tokens en el analizador, donde quiera que ésto se haga, se esá llamando a `setType`, `setText`, `setLine` y `setColumn`, pero no a `setFilename`. Hay que encontrar el punto en el que se crean los tokens para poder modificar la conducta del analizador en dicho punto.

En casos así lo mejor es mirar directamente el código generado para el analizador. Todos los analizadores generados por ANTLR tienen el mismo aspecto: son subclases de una clase de analizador de ANTLR (`antlr.CharScanner` para los analizadores léxicos, `antlr.Parser` para los sintácticos y `antlr.TreeParser` para los semánticos). Hay una serie de constructores y luego un método por cada regla que tuviera el analizador. Estos métodos suelen tener un "switch" en el que se decide la transición a realizar.

Centrándonos en los analizadores léxicos, hay un método que es llamado muy a menudo, y que tiene un nombre revelador: es `makeToken`. Este método, implementado en `antlr.CharScanner`, es el que crea los tokens y el que "se está olvidando" de añadirles el nombre de fichero. El código de dicho método es el siguiente:

```
    protected Token makeToken(int t) {
        try {
            Token tok = (Token)tokenObjectClass.newInstance();
            tok.setType(t);
            tok.setColumn(inputState.tokenStartColumn);
            tok.setLine(inputState.tokenStartLine);

            return tok;
        }
        catch (InstantiationException ie) {
            panic("can't instantiate token: " + tokenObjectClass);
        }
        catch (IllegalAccessException iae) {
            panic("Token class is not accessible" + tokenObjectClass);
        }
        return Token.badToken;
    }
```

Nótese que `makeToken` no llama a `setFilename` ni a `setText`. `setText` es llamado más tarde (el

el código que invoca a maketoken), pero no `setFilename`.

Se puede modificar `makeToken` añadiéndole la línea `setFilename(inputState.filename);` Pero en lugar de modificar el código de una clase de antlr vamos a hacer que nuestro analizador léxico, que al fin y al cabo es una subclase de `CharScanner`, sobreescriba `makeToken` y le añada la información del nombre de fichero.

En la lista de correo de ANTLR, algunas voces (entre ellas la mía) se han alzado pidiendo que se incluya al menos la línea que añade el nombre de fichero a los tokens en makeToken. Es posible que ANTLR la incluya en el futuro.

Para ello disponemos de la zona de código nativo en el fichero de definición del analizador. La zona de código nativo, que aún no hemos utilizado en este analizador, es una zona opcional en la que escribir métodos, atributos y constructores en lenguaje nativo (java, C++ o C#) para completar nuestro analizador. Cuando está presente, se encuentra justo antes de la zona de reglas, y entre llaves ({}). En esta zona también es posible sobreescribir métodos de la superclase del analizador, y eso es lo que vamos a hacer. El código a añadir en nuestro analizador léxico será el siguiente:

```
class LeLiLexer extends Lexer;
options {...} // Las opciones permanecen sin modificación
tokens  {...} // Sin modificación

{  // Comienza la zona de código nativo
   protected Token makeToken(int type)
   {
      // Usamos la implementación de la superclase...
      Token result = super.makeToken(type);
      // ... añadimos información del nombre de fichero
      result.setFilename(inputState.filename);
      // ... y devolvemos el token
      return result;
   }
}

// Zona de reglas (sin modificar)
```

Una vez recompilado el fichero `LeLiLexer.g` y los generados a partir de él (ver sección anterior), podremos lanzar nuestra clase `Tool` y obtener un listado con el nombre del fichero del que proceden los tokens:

```
test.leli:3:1: clase, 13
test.leli:3:7: Inicio, 35
test.leli:3:14: extiende, 14
test.leli:3:23: Objeto, 35
test.leli:4:1: {, 38
test.leli:5:9: método, 15
test.leli:5:16: inicio, 35
test.leli:5:22: (, 41
test.leli:5:23: ), 42
test.leli:6:9: {, 38
test.leli:7:17: Sistema, 35
test.leli:7:24: ., 40
test.leli:7:25: imprime, 35
test.leli:7:32: (, 41
test.leli:7:33: ¡Hola mundo!, 62
test.leli:7:47: +, 51
```

```
test.leli:7:48: nl, 10
test.leli:7:50: ), 42
test.leli:7:51: ;, 36
test.leli:8:9: }, 39
test.leli:9:1: }, 39
```

Sección 4.7: El fichero LeLiLexer.g

El código completo del fichero `LeLiLexer.g` que hemos escrito será por lo tanto el siguiente:

```
header{
package leli;
/*-----------------------------------------------*\
 | Un intérprete para un Lenguaje Limitado(LeLi) |
 | _____________________________________________ |
 |                                               |
 |              ANALISIS LÉXICO                  |
 | _____________________________________________ |
 |                                               |
 |            Enrique J. Garcia Cota             |
\*-----------------------------------------------*/

}

/**
 *  El objeto que permite todo el analisis léxico.
 *  notas: comentar todo esto al final del analex.
 */
class LeLiLexer extends Lexer;

options{
   charVocabulary = '\3'..'\377';  // Unicodes usuales
   exportVocab=LeLiLexerVocab;     // Comentar al hacer el parser
   testLiterals=false;             // comprobar literales solamente cuando se
diga exp
   k=2;                            // lookahead
}

tokens
{
// Tipos basicos
   TIPO_ENTERO   = "Entero"   ;
   TIPO_REAL     = "Real"     ;
   TIPO_BOOLEANO = "Booleano" ;
   TIPO_CADENA   = "Cadena"   ;

// Literales booleanos
   LIT_CIERTO = "cierto" ; LIT_FALSO = "falso" ;

// Literales cadena
   LIT_NL = "nl"; LIT_TAB = "tab" ; LIT_COM = "com";

// Palabras reservadas
   RES_CLASE       = "clase"       ;
   RES_EXTIENDE    = "extiende"    ;
   RES_METODO      = "método"      ;
   RES_CONSTRUCTOR = "constructor" ;
   RES_ATRIBUTO    = "atributo"    ;
   RES_ABSTRACTO   = "abstracto"   ;
   RES_PARAMETRO   = "parámetro"   ;
   RES_CONVERTIR   = "convertir"   ;
   RES_MIENTRAS    = "mientras"    ;
   RES_HACER       = "hacer"       ;
```

```
   RES_DESDE       = "desde"       ;
   RES_SI          = "si"          ;
   RES_OTRAS       = "otras"       ;
   RES_SALIR       = "volver"      ;
   RES_ESUN        = "esUn"        ;
   RES_SUPER       = "super"       ;

// Operadores que empiezan con letras;
   OP_Y           = "y"  ;
   OP_O           = "o"  ;
   OP_NO          = "no" ;

// Los literales real y entero son "devueltos" en las
// acciones del token "privado" LIT_NUMERO
   LIT_REAL ; LIT_ENTERO;

}

{ // Comienza la zona de código
   protected Token makeToken(int type)
   {
      // Usamos la implementación de la superclase...
      Token result = super.makeToken(type);
      // ...y añadimos información del nombre de fichero
      result.setFilename(inputState.filename);
      // y devolvemos el token
      return result;
   }
}

/**
 * Los tres tipos de retorno de carro.
 */
protected NL :
   (
     ("\r\n") => "\r\n"  // MS-DOS
   | '\r'                // MACINTOSH
   | '\n'                // UNIX
   )
   { newline(); }
   ;

/**
 * Esta regla permite ignorar los blancos.
 */
protected BLANCO :
   ( ' '
   | '\t'
   | NL
   ) { $setType(Token.SKIP); } // La accion del blanco: ignorar
   ;

/**
 *  Letras españolas.
 */
protected LETRA
   : 'a'..'z'
   | 'A'..'Z'
```

```
   | 'ñ' | 'Ñ'
   | 'á' | 'é' | 'í' | 'ó' | 'ú'
   | 'Á' | 'É' | 'Í' | 'Ó' | 'Ú'
   | 'ü' | 'Ü'
   ;
/**
 * Dígitos usuales
 */
protected DIGITO : '0'..'9';

/**
 * Regla que permite reconocer los literales
 * (y palabras reservadas).
 */
IDENT
options {testLiterals=true;} // Comprobar palabras reservadas
:
   (LETRA|'_') (LETRA|DIGITO|'_')*
;

// Separadores
PUNTO_COMA     : ';'  ;
COMA           : ','  ;
CORCHETE_AB    : '['  ;
CORCHETE_CE    : ']'  ;
LLAVE_AB       : '{'  ;
LLAVE_CE       : '}'  ;
PUNTO          : '.'  ;
PARENT_AB      : '('  ;
PARENT_CE      : ')'  ;
BARRA_VERT     : '|'  ;

//  operadores
OP_IGUAL       : "==" ;
OP_DISTINTO    : "!=" ;
OP_ASIG        : '='  ;
OP_MENOR       : '<'  ;
OP_MAYOR       : '>'  ;
OP_MENOR_IGUAL : "<=" ;
OP_MAYOR_IGUAL : ">=" ;
OP_MAS         : '+'  ;
OP_MENOS       : '-'  ;
OP_MASMAS      : "++" ;
OP_MENOSMENOS  : "--" ;
OP_PRODUCTO    : '*'  ;
OP_DIVISION    : '/'  ;


/**
 * Permite reconocer literales enteros y reales sin generar conflictos.
 */
LIT_NUMERO : (( DIGITO )+  '.' ) =>
                ( DIGITO )+ '.' ( DIGITO )* { $setType (LIT_REAL); }
           | ( DIGITO )+  { $setType (LIT_ENTERO); }
           ;

/**
 * Comentario de una sola línea
```

```
 */
protected COMENTARIO1
   :
      "//" (~('\n'|'\r'))*
      { $setType(Token.SKIP); }
   ;

/**
 * Comentario de varias líneas
 */
protected COMENTARIO2  :
   "/*"
   ( ('*' NL) => '*' NL
   | ('*' ~('/'|'\n'|'\r')) => '*' ~('/'|'\n'|'\r')
   | NL
   | ~( '\n' | '\r' | '*' )
   )*
   "*/"
   { $setType(Token.SKIP); }
   ;

/**
 * Literales cadena
 */
LIT_CADENA :
   '"' !
   ( ~('"'|'\n'|'\r') )*
   '"' !
   ;
```

Sección 4.8: Conclusión

Ya hemos visto cómo realizar un analizador léxico con ANTLR. Espero que no haya sido muy difícil de entender: los analizadores léxicos son la parte más fácil de hacer con ANTLR.

Se trata de un analizador sencillo, que no ilustra todas las capacidades de análisis léxico que posee ANTLR. Algunas de estas capacidades inexploradas son:

- Analizador léxico insensible a las mayúsculas (utilizando la opción `caseSensitive`)
- Utilizar otros métodos de Token además de `$setType`, como `$setToken`, `$append` o `$setText`.
- Utilizar predicados semánticos. En algunos casos podríamos haberlos utilizado, pero la gramática sería potencialmente menos portable (solamente podría generarse con seguridad código java).
- Análisis léxico de ficheros binarios.
- Utilización de otras opciones a nivel del analizador y de sus reglas, como `ignore` o `classHeaderSuffix`.
- Utilizar varios analizadores sintácticos simultáneamente, como ocurre con el analizador de comentarios javadoc en java (que funciona a la vez que el analizador de código).

Hemos visto que no es imposible implementar un lexer recursivo descendente, y que una vez que se comprende el funcionamiento de pred-LL(k) es incluso sencillo.

De momento hemos terminado con el análisis léxico. En el siguiente capítulo nos meteremos de lleno con el análisis sintáctico.

Capítulo 5:
Análisis sintáctico de LeLi

"Las palabras son como las hojas; cuando abundan, poco fruto hay entre ellas."
Alexander Pope

Capítulo 5:
Análisis sintáctico de LeLi.....121

Sección 5.1: Introducción.....122
Sección 5.2: Definición del analizador sintáctico.....123
5.2.1: Opciones del analizador.....123
5.2.2: Importando los tokens.....123
5.2.3: Zona de tokens.....124
Sección 5.3: Zona de reglas.....126
5.3.1: Programa.....126
5.3.2: Definición de clases.....126
5.3.3: Azúcar sintáctica.....127
Insulina semántica.....128
5.3.4: Definición de atributos.....131
5.3.5: Definición de métodos.....131
5.3.6: Expresiones.....133
Introducción.....133
Reglas de las expresiones.....133
Accesos.....135
5.3.7: Instrucciones.....137
Instrucción nula.....138
Instrucción volver.....138
Intrucción-expresión.....138
Declaración de variables locales.....138
Bucles.....139
Instrucción "si".....139
Sección 5.4: Fichero LeLiParser.g.....142
Sección 5.5: Compilación del analizador.....149
Sección 5.6: Ejecución: modificaciones en la clase Tool.....150
5.6.1: Introducción.....150
5.6.2: Interpretando la línea de comandos: el paquete antlraux.clparse.....150
Presentación.....150
Funcionamiento.....151
Ejemplo.....151
5.6.3: La línea de comandos inicial: el método leli.Tool.LeeLC().....153
5.6.4: El método leli.Tool.imprimeAyuda().....154
5.6.5: El nuevo método main.....154
5.6.6: El método leli.Tool.trabaja().....155
Pasos.....155
Inicio.....155
Nivel léxico.....156
Nivel sintáctico.....156
Obtención del AST.....157
Representación textual del AST.....157
Representación gráfica del AST.....157
Código completo.....158
Sección 5.7: Otros aspectos de los ASTs.....161
5.7.1: Fábricas de ASTs.....161
5.7.2: ASTs heterogéneos.....161
Sección 5.8: Conclusión.....164

Sección 5.1: Introducción

El análisis sintáctico es la esencia misma de la interpretación de lenguajes. Muchísimos programadores no lo distinguen (equivocadamente) del análisis semántico. El análisis léxico, por otra parte, es considerado una tarea más o menos "trivial"; se considera una molestia necesaria.

De los tres niveles del análisis, el análisis sintáctico es el que más satisfacción proporciona al ser implementado. Tiene mucho encanto poder definir las reglas de un nuevo lenguaje[35]. Además, el concepto de reglas sintácticas es muy intuitivo, y se amolda muy bien a la sintaxis BNF y todavía mejor a EBNF.

Todo esto provoca que a la hora de desarrollar un compilador se tienda a prestar más atención al nivel sintáctico que al resto de los niveles.

Otro tema importante a tener en cuenta es la recuperación de errores. Muchos programadores de compiladores "encastran" la recuperación de errores dentro del análisis sintáctico, lo que produce analizadores muy voluminosos y poco flexibles.

Nosotros vamos a separar estos dos conceptos; en este capítulo nos centraremos en el análisis sintáctico propiamente dicho, y en el siguiente le añadiremos la recuperación de errores de una forma completamente flexible y manejable (utilizando la herencia de gramáticas).

Bueno, vamos a empezar.

35 Las reglas sintácticas, claro. Las semánticas deberían definirse en el nivel semántico.

Sección 5.2: Definición del analizador sintáctico

El fichero donde definiremos nuestro analizador sintáctico se llamará `LeLiParser.g`. Tendrá la estructura usual:

```
header{
   package leli;
}
class LeLiParser extends Parser; // Se extiende Parser, y no Lexer
options {
   /* opciones del analizador */
}
tokens {
   /* tokens */
}
/* Reglas de generación */
```

Como podemos ver el analizador sintáctico también formará parte del paquete leli.

5.2.1: Opciones del analizador

Las opciones del analizador sintáctico serán las siguientes:

```
options
{
   k = 2;
   buildAST = true;
   importVocab = LeLiLexerVocab;
   exportVocab = LeLiParserVocab;
}
```

Las dos primeras no deberían ser un problema a estas alturas. k es el lookahead y `buildAST` activa la construcción del AST.

Las otras dos opciones, `importVocab` y `exportVocab`, son necesarias cuando se utilizan analizadores en ficheros diferentes (como en nuestro caso; el analizador léxico en un fichero, el sintáctico en otro, etc.). Se explican con detenimiento en el siguiente apartado.

5.2.2: Importando los tokens

Al final del capítulo anterior, cuando hablamos de compilar el fichero del analizador *léxico* (`LeLiLexer.g`) observamos que además del fichero `LeLiLexer.java` se generaban otros dos ficheros, `LeLiLexerVocabTokenTypes.java` y `LeLiLexerVocabTokenTypes.txt`. Estos dos ficheros son los que permiten al alanizador léxico "compartir" sus tokens con el analizador sintáctico. ¿Cómo lo hacen? Para averiguarlo analicemos su contenido.

El fichero `LeLiLexerVocabTokenTypes.java` no define una clase java, sino una interfaz llamada `LeLiLexerVocabTokenTypes`. Es una interfaz muy sencilla; carece de métodos, y solamente tiene atributos de tipo entero. Cada uno de éstos enteros representará un token del analizador léxico (excepto los primeros, que son utilizados internamente por ANTLR). El código de la interfaz viene a ser algo así:

```
package leli;

public interface LeLiLexerVocabTokenTypes {
   int EOF = 1;
   int NULL_TREE_LOOKAHEAD = 3;
   int TIPO_ENTERO = 4;
   int TIPO_REAL = 5;
  ...
   int OP_MAS = 55;
   int OP_MENOS = 56;
   int OP_PRODUCTO = 57;
   int OP_DIVISION = 58;
}
```

Si nos fijamos en el fichero `LeLiLexer.java` podemos ver que el analizador léxico implementa la interfaz:

```
public class LeLiLexer extends antlr.CharScanner
   implements LeLiLexerVocabTokenTypes, TokenStream
```

El fichero `LeLiLexerVocabTokenTypes.txt`, por su parte, es del siguiente estilo:

```
LeLiLexerVocab    // Nombre del vocabulario
TIPO_ENTERO="entero"=4
TIPO_REAL="real"=5
...
OP_MAS=55
OP_MENOS=56
OP_PRODUCTO=57
OP_DIVISION=58
```

El fichero codifica casi exactamente la misma información que el primero, con otra sintaxis. La única diferencia es que aquí, además del tipo de los tokens, se incluye su "nombre", en el caso de que lo tengan. Por ejemplo, el token `TIPO_ENTERO` tiene el tipo 4 y su nombre es "entero". `OP_MAS`, por su parte, carece de nombre.

En definitiva, lo que se le está diciendo al analizador sintáctico con la opción `importVocab=LeLiLexerVocab` es "el analizador debe implementar la interfaz `LeLiLexerVocabTokenTypes`. Busca los tipos y los nombres de los tokens en el fichero `LeLiLexerVocabTokenTypes.txt`".

La función de la opción `exportVocab` es exactamente la esperada: permite al analizador sintáctico "exportar" su conjunto de tokens a un fichero de manera que éstos puedan ser compartidos con el analizador semántico.

Las opciones `importVocab`, `exportVocab` y `k` son comunes a todos los tipos de analizador, mientras que `buildAST` solamente tiene sentido en los analizadores sintácticos (para construir el AST) y en los semánticos (para transformarlo).

5.2.3: Zona de tokens

Ya he comentado la función de la sección de tokens en un analizador sintáctico ANTLR; la sección de tokens permite declarar tokens imaginarios que se utilizarán para enraizar los árboles que lo necesiten (generalmente en reglas que no tengan un símbolo apropiado para la raíz o reglas con listas).

La sección de tokens que utilizaremos para LeLi es la siguiente:

```
tokens
{
// Tokens imaginarios para enraizar listas
   PROGRAMA ;
   LISTA_MIEMBROS;
   LISTA_EXPRESIONES;
   LISTA_INSTRUCCIONES;

// Tokens imaginarios que se utilizan cuando no hay raíces adecuadas
   OP_MENOS_UNARIO;
   INST_EXPRESION;
   INST_DEC_VAR;
   LLAMADA;
   ATRIBUTO;

// Otros
   TIPO_VACIO;
}
```

Sección 5.3: Zona de reglas

5.3.1: Programa

La primera regla que se escribe en una gramática es la que define un programa. Un programa no es más que una lista de una o más definiciones de clases. Así que literalmente:

```
programa : (decClase)+
           {## = #( #[PROGRAMA, "PROGRAMA"] ,##);}
         ;
```

Como se trata de una lista, enraizamos el árbol generado automáticamente con un nodo imaginario. En éste caso concreto no es estrictamente necesario, pero enraizar las listas es un hábito en el que más vale pecar por exceso que por defecto.

5.3.2: Definición de clases

Una definición de clase es:

- La palabra reservada `clase`.
- El nombre de la clase, opcionalmente seguido de `extiende` y el nombre de una clase padre.
- Un conjunto de cero o más miembros (atributos y métodos) encerrados entre llaves.

Así que escribimos:

```
decClase : RES_CLASE^ IDENT (RES_EXTIENDE IDENT)?
           LLAVE_AB! listaMiembros LLAVE_CE!
         ;
```

En este caso utilizamos el nodo del token `RES_CLASE` como raíz.

Esta regla reconocerá muy bien las definiciones de clases que le pasemos. No obstante presenta un problema: no construye el AST más adecuado. Cuando en la declaración no se expresa explícitamente su superclase, ésta debe ser la clase base, `Objeto`. Para hacer un AST lo más homogéneo posible, vamos a hacer que siempre se genera un árbol AST. Lo más cómodo es utilizar una regla "extra" que se encargue de gestionar la parte de creación de árboles:

```
/** Definición de una clase. */
decClase : RES_CLASE^ IDENT clausulaExtiende
           LLAVE_AB! listaMiembros LLAVE_CE!
         ;

/** Cláusula "extiende" **/
clausulaExtiende : RES_EXTIENDE^ IDENT
                 | /*nada*/ // Creamos un AST
                   { ## = #( #[RES_EXTIENDE,"extiende"],
                             #[IDENT, "Objeto"] ); }
                 ;
```

`listaMiembros` tendrá, por su parte, la siguiente implementación:

```
protected listaMiembros
   : (decMetodo|decConstructor|decAtributo)*
     {## = #( #[LISTA_MIEMBROS, "LISTA_MIEMBROS"] ,##);}
   ;
```

De nuevo enraizamos las listas con un token imaginario. Nótese la palabra clave `protected` que precede a la definición de la regla; declarar como protegidas las reglas auxiliares es muy común.

5.3.3: Azúcar sintáctica

Se dice que un lenguaje tiene azúcar sintáctica si tiene alguna regla que, aunque no sea estrictamente necesaria para codificar el lenguaje, facilita un poco la vida al programador. Un ejemplo claro de reglas con azúcar sintáctica son las declaraciones de atributos, parámetros y variables.

En LeLi, las declaraciones de parámetros, atributos y variables locales permiten al programador declarar varios elementos en una sola declaración. Por ejemplo, el siguiente código:

```
clase Azúcar
{
   atributo Booleano bEsMoreno;
   atributo Booleano bEsBlanca;
   atributo abstracto Entero caloriasCuchara=16;
   atributo abstracto Entero cucharadasMedias=2;

   método endulzar(Entero dulzura; Entero cucharadasExtra)
   {
      Entero calorias1 = caloriasCuchara * cucharadasMedias;
      Entero calorias2 = caloriasCuchara * cucharadasExtra;
      dulzura = calorias1 + calorias2;
   }
}
```

Es equivalente a este otro:

```
clase Azúcar
{
   atributo Booleano bEsMoreno, bEsBlanca;
   atributo abstracto Entero caloriasCuchara=16, cucharadasMedias=2;

   método endulzar(Entero dulzura, cucharadasExtra)
   {
      Entero calorias1 = caloriasCuchara * cucharadasMedias,
             calorias2 = caloriasCuchara * cucharadasExtra ;
      dulzura = calorias1 + calorias2;
   }
}
```

En una misma línea de declaración de atributos se declaran varios atributos, y lo mismo ocurre con los parámetros y variables locales.

La representación gráfica del AST para el ejemplo anterior, del cual solamente incluiremos los ASTs de los dos primeros atributos no abstractos (`bEsMoreno`, `bEsBlanca`) de la clase `Azúcar`, compartirán su raíz:

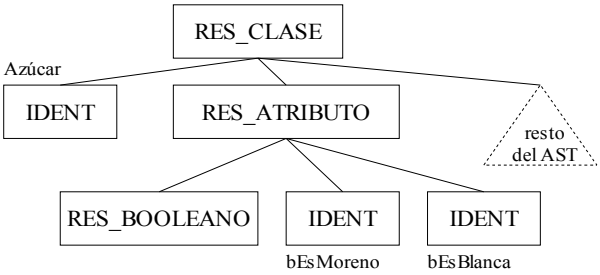

Ilustración 5.1 Árbol AST con diabetes sintáctica

He marcado éste arbol como "con diabetes sintáctica".

La razón es que un árbol así no es idóneo para la etapa siguiente del análisis (análisis semántico) es mucho más conveniente un árbol en el que los diferentes identificadores de los atributos tengan "un árbol para cada uno", por razones que veremos en el siguiente capítulo[36]. Es decir, que el árbol que genere la segunda implementación de la clase `Azúcar` debe ser igual al que genera la primera, en la que hay un árbol de declaración para cada identificador, como el mostrado en la siguiente gráfica:

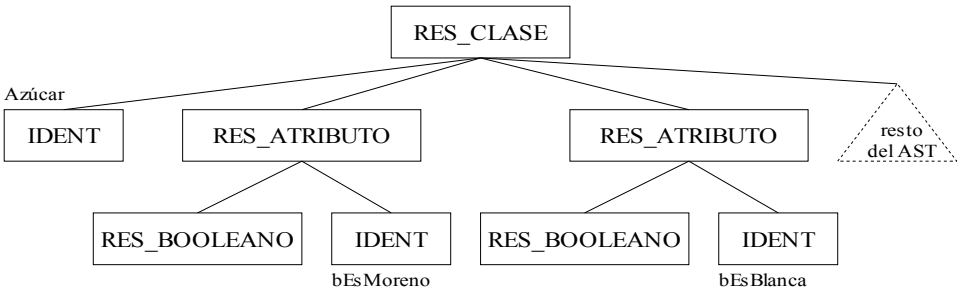

Ilustración 5.2 Árbol AST deseable

Insulina semántica

Para evitar los árboles diabéticos necesitaremos crear los árboles de una manera especial.

Vamos a tener que modificar la construcción del AST. He aquí algunas consideraciones importantes:

1. La misma problemática que se encuentra en las declaraciones de atributos (azúcar sintáctica) se encuentra en las declaraciones de variables locales y parámetros de métodos. Lo más deseable sería implementarlos todos con el mismo conjunto de reglas.
2. Los árboles deben ser creados al llegar a los identificadores, y no antes, pues solamente en ese momento se tiene toda la información (tipo del atributo/variable/parámetro, nombre y valor si está inicializado)
3. De alguna manera hay que crear sendas copias del nodo que representa el tipo del árbol de declaración (`RES_BOOLEANO`) y de su raíz (`RES_ATRIBUTO`) una vez por cada identificador.
4. Hay que continuar diferenciando si se puede o no añadir una variable especial, el valor inicial del objeto (recuérdese que las variables y los atributos abstractos pueden ser inicializados).

En definitiva "Los árboles deben ser creados al llegar a los identificadores" -punto 2-, "y hay que pasarles el tipo de declaración (atributo, variable o parámetro) y tipo de la declaración (`Entero`, `Cadena`...) que también debe ser copiado" -punto 3- "además de un booleano que indique si se

36 Para los impacientes: hace falta para poder asociar un árbol AST distinto a cada identificador en la etapa de enriquecimiento semántico.

pueden o no iniciar" -punto 4-.

La regla que utilizaremos para reconocer las declaraciones de los atributos, parámetros y variables de LeLi, así como de eliminar el azúcar sintáctica de las reglas, se llama `listaDeclaraciones`. Es una regla que acepta varios parámetros, que definirán su forma de actuar. El primer parámetro de la regla sirve para especificar la "raíz" de todas las declaraciones (en este caso vendrá proporcionada por el token `RES_ATRIBUTO`).

El segundo parámetro de la regla es un booleano que indica si las declaraciones se pueden "iniciar". Hay dos formas de iniciar una declaración: con un valor (por ejemplo, `Entero i=2`) o con "parámetros de constructor" (por ejemplo `Persona pepe("Pepe", "Sanchez", 35);`). En el lenguaje LeLi las variables y los atributos *abstractos* pueden iniciarse de esta forma, mientras que los parámetros y los atributos *no abstractos* no pueden. Por eso ha sido necesaria la variable local `abstracto` en la regla `decAtributo`.

Veamos entonces la regla `listaDeclaraciones`:

```
listaDeclaraciones [AST raiz, boolean inicializacion]
   : t:tipo!
     declaracion[raiz,#t,inicializacion]
     (COMA! declaracion[raiz,#t,inicializacion])*
   ;
```

La regla se limita a reconocer el tipo de la declaración y a pasárselo a la regla que hace todo el trabajo, es decir, `declaracion`. Esta nueva regla se implementa de la siguiente manera:

```
declaracion ! // desactiva construcción por def. del AST
[AST r, AST t, boolean inicializacion] // parámetros
{
   AST raiz = astFactory.dupTree(r);     // copia del arbol
   raiz.addChild(astFactory.dupTree(t)); // copia del árbol
}
   : i1:IDENT
     {
        raiz.addChild(#i1);
        ## = raiz;
     }
   | { inicializacion }? // pred. semántico
     i2:IDENT OP_ASIG valor:expresion
     {
        raiz.addChild(#i2);
        raiz.addChild(#valor);
        ## = raiz;
     }
   | { inicializacion }?
     i3:IDENT parentAb li:listaExpresiones parentCe
     {
        raiz.addChild(#i3);
        raiz.addChild(#li);
        ## = raiz;
     }
   ;
```

Nótese que es declaracion la que se encarga que realizar las copias de nodos AST necesarias para eliminar el azúcar sintáctica. Los dos predicados semánticos que preceden a las dos últimas alternativas de la regla impiden que éstas se reconozcan si el parámetro inicializacion es falso.

Nótese igualmente que `listaDeclaraciones` produce un AST degenerado de declaraciones.

Ya hemos explicado qué es lo que debe hacer la regla `declaracion` – duplicar ciertos nodos y enraizarlos de cierta forma. Veamos cómo lo hace. Para entenderlo hay que comprender tres conceptos: los predicados semánticos (que ya hemos explicado en los primeros capítulos de este documento) y la forma de duplicar y añadir hijos a los nodos AST. Estas dos últimas operaciones se realizan utilizando `astFactory`.

`astFactory` es un atributo del analizador sintáctico, concretamente una instancia de `ASTFactory`. El método `dupTree` de las `ASTFactory` permite copiar la estructura de un AST, nodo a nodo[37]. Hemos de crear cópias de los ASTs r y t para poder modificarlos a nuestro antojo (añadiéndoles hijos, principalmente) repetidas veces, sin que una influya en la otra.

Veamos ahora `addChild`. Hemos tenido que utilizarlo para no perder hijos que ya tenía r; normalmente, si se desea añadir un elemento a un AST, podemos escribirlo así:

```
// añadir el AST t2 a el AST t1
t1 = #(t1,t2);
```

Este método tiene, sin embargo, un importante problema: elimina todos los hijos de t1, situando en su lugar t2. Si lo que queremos es conservar los hijos de t1 y añadir un nuevo AST tendremos que utilizar `addChild`:

```
t1.addChild(t2); // añade t2 al final de la lista de hijos de t1 sin borrarla.
```

Téngase presente que los dos métodos son equivalentes si t1 no tiene hijos (el segundo es más rápido, pero el primero es más portable).

Es una pena que no existan abreviaturas como `#()`, `##`, y `#[]`, para hacer copias de nodos o para añadir hijos a un AST, de manera que tengamos que utilizar métodos poco portables.

Una última nota: algunos podrían pensar que hemos perdido tiempo creando una variable (raiz) para ir guardando el AST que se va creando, ya que podríamos utilizar ##:

```
declaracion ! // desactiva construcción por def. del AST
[AST r, AST t, boolean inicializacion] // parámetros
{
   ## = astFactory.dupTree(r);      // copia del arbol
   ##.addChild(astFactory.dupTree(t)); // copia del árbol
}
   : i1:IDENT
     {
        ##.addChild(#i1);
     }
   | { inicializacion }? // pred. semántico
     i2:IDENT OP_ASIG valor:expresion
     {
        ##.addChild(#i2);
        ##.addChild(#valor);
     }
   | { inicializacion }?
     i3:IDENT parentAb li:listaExpresiones parentCe
     {
        ##.addChild(#i3);
        ##.addChild(#li);
     }
   ;
```

¡También yo pensaba que un código así funcionaría! No obstante, una mirada al código generado

37 Consúltese la sección “Otros aspectos de los ASTs” en el capítulo 5 para más nformación sobre las fábricas de ASTs.

me hizo cambiar de idea: "##" se utiliza principalmente cuando ANTLR está construyendo automáticamente un AST. En cada acción, ANTLR inicializa "##" a "lo que lleva construido hasta ese momento". Dado que la construcción por defecto del AST está desactivada, ¡## se inicializa a null en cada acción!

Así que en general hay que tener cuidado a la hora de utilizar ## en varias acciones, sobre todo cuando la construcción por defecto está desactivada. Si ha de hacerse, es mejor utilizar una variable auxiliar y "asignarla"a ## al final.

5.3.4: Definición de atributos

Consideremos ahora cómo reconocer las declaraciones de atributos. Una declaración comienza por la palabra clave `atributo`, seguida (o no) por la palabra reservada `abstracto`. Después viene un tipo y finalmente una lista de declaraciones, que son identificadores seguidos opcionalmente de valores de inicialización (ya sean valores o parámetros de constructor) si los atributos son abstractos. Las declaraciones van separadas por comas, y finalmente hay un punto y coma.

Lo primero que hay que definir es la regla que reconoce el tipo con el que un atributo/variable/parámetro puede ser declarado. Éste puede ser `Entero`, `Real`, `Booleano`, `Cadena` o una clase definida por el usuario. Dicho y hecho:

```
tipo : TIPO_ENTERO   // tipo Entero
     | TIPO_REAL     // tipo Real
     | TIPO_BOOLEANO // tipo Booleano
     | TIPO_CADENA   // tipo Cadena
     | IDENT         // tipo no básico
     ;
```

Una vez leído el apartado anterior, reconocer las declaraciones de atributos debería ser muy sencillo:

```
decAtributo
{ boolean abstracto = false; }
   : raiz:RES_ATRIBUTO^
     ( a:RES_ABSTRACTO { abstracto=true; } )?
     listaDeclaraciones[#raiz, abstracto] PUNTO_COMA!
   ;
```

Analicemos la regla. En primer lugar, se declara la variable local `abstracto`. Como está declarada en una acción a la izquierda de los dos puntos, su ámbito será toda la regla.

La primera estructura que se reconoce es el token `RES_ATRIBUTO`. Tras él, aplicándosele el operador de opcionalidad (la interrogación) puede aparecer el token `RES_ABSTRACTO`. En tal caso la variable `abstracto` se hace cierta.

Después, solamente queda invocar `listaDeclaraciones` con los parámetros adecuados.

5.3.5: Definición de métodos

Los métodos pueden ser de tres tipos: constructores, métodos normales y métodos abstractos.

Un constructor no es más que la palabra reservada constructor seguida de una lista de parámetros entre paréntesis y una lista de instrucciones entre llaves:

```
decConstructor : RES_CONSTRUCTOR^ PARENT_AB! listaDecParams PARENT_CE!
                 LLAVE_AB! listaInstrucciones LLAVE_CE!
               ;
```

Los métodos abstractos y los no abstractos pueden tener un tipo de retorno. Por lo demás, solamente se diferencian en la utilización de la palabra reservada `abstracto`.

Un método abstracto o no abstracto comienza con la palabra reservada `método` seguida (o no) por la palabra reservada `abstracto`. Después viene el tipo de retorno, que es opcional. Luego viene el nombre del método, que es un `IDENT`. Finalmente hay una lista de parámetros entre paréntesis y una lista de instrucciones entre llaves.

```
decMetodo
   : RES_METODO^ (RES_ABSTRACTO)? tipoRetorno IDENT
     PARENT_AB! listaDecParams PARENT_CE!
     LLAVE_AB! listaInstrucciones LLAVE_CE!
   ;
```

Quedan algunas cosas por definir: `tipoRetorno`, `listaParams` y `listaInstrucciones`.

Para implementar `tipoRetorno` tenemos que considerar dos opciones: que el método devuelva algo o que no devuelva nada. La primera opción es sencilla: basta con reconocer `tipo`. La segunda opción consiste en reconocer "nada" (ningún token). Algunos podrían estar tentados de utilizar el operador de opción ("?") ya que se reconoce "un tipo o nada", y hacer algo así:

```
protected tipoRetorno : (tipo)? ;
```

Sin embargo hay una razón por la cual prefiero no utilizar esta regla. Pretendo que internamente todos los métodos tengan un tipo, incluso los que no devuelven nada. De esta forma el recorrido de árboles de la etapa de análisis semántico será más sencilla.

En otros lenguajes se utiliza el tipo `void` para especificar esta idea, pero no hay una palabra reservada "vacío" en LeLi. Pero éso no impide que exista el concepto aunque no exista la palabra. Lo que voy a hacer es utilizar un token imaginario e implementar `tipoRetorno` así:

```
protected tipoRetorno
   : tipo
   | /* nada */ {## = #[TIPO_VACIO,"TIPO_VACIO"]; }
   ;
```

Nótese que el conjunto `PRIMERO(tipoRetorno)` contiene a `IDENT`, y que `tipoRetorno` puede ser vacío y siempre va seguido de un `IDENT`. Gracias a la opción `k=2` del analizador podemos ignorar éste problema y ahorrarnos unos engorrosos predicados sintácticos.

`listaDecParams` es muy fácil de implementar si uno se basa en la regla `listaDeclaraciones`, que definí anteriormente. Una lista de parámetros es una lista de cero o más `listaDeclaraciones` separadas por el carácter punto y coma (";").

```
listaDecParams
{ final AST raiz = #[RES_PARAMETRO,"parámetro"] ;}
   :
   ( listaDeclaraciones[raiz, false]
     ( PUNTO_COMA! listaDeclaraciones[raiz, false])*
   )? // opcional
   { ## = #(#[LISTA_DEC_PARAMS,"LISTA_DEC_PARAMS"], ##); }
   ;
```

Como de costumbre, toda lista es enraizada con un token imaginario. Nótese el operador de opción ("?") casi al final de la regla.

Veremos `listaInstrucciones` un poco más adelante, en el apartado de instrucciones.

5.3.6: Expresiones

Introducción

Implementar el análisis sintáctico de las expresiones es una de las pocas cosas más fáciles de hacer con bison que con ANTLR, gracias a la directiva `%left` y a que las reglas tienen "precedencia de utilización" según el orden en el que se declaren. En ANTLR hay que seguir utilizando la técnica habitual de dividir las expresiones en "niveles de precedencia", de manera que si dos operadores tienen diferente precedencia están en niveles de precedencia diferentes.

Las expresiones en LeLi se organizan según la siguiente tabla:

Nivel	Nombre	Elementos
9	Asignaciones	`=`
8	"O" lógico	`o`
7	"Y" lógico	`y`
6	Comparaciones	`<=, >=, ==, !=, <, >`
5	Suma y resta aritmética	`+, -`
4	Producto y división	`*, /`
3	Cambio de signo unario	`+, -`
2	Postincremento y decremento	`++, --`
1	Negación	`no`
0	esUn	`esUn`
(-1)	accesos y paréntesis	`<accesos>, ()`

Por cada nivel hay una regla, que utiliza reglas del siguiente nivel.

Reglas de las expresiones

Las asignaciones son la entrada del sistema de análisis de expresiones:

```
expresion: expAsignacion;
```

Incluir la asignación dentro de las expresiones es una práctica corriente en el mundo de los compiladores; a pesar de tratarse de una instrucción, su relación con las expresiones es tan grande que incluirla dentro del grupo de las expresiones es más sencillo que no hacerlo. Además,

existen algunas expresiones que también pueden ser utilizadas como instrucciones. Por ejemplo el postdecremento (`i++`) o las llamadas a métodos son también expresiones.

Dado el sistema de precedencia que estamos usando, las asignaciones "incluyen" al resto de las expresiones. Una asignación es una expresión de nivel inferior que puede estar o no seguida del operador de asignación y otra expresión:

```
expAsignacion : expOLogico (OP_ASIG^ expOLogico)? ;
```

Las expresiones del siguiente nivel son las expresiones lógicas con el operador O. Una expresión de éste tipo son una o varias expresiones del siguiente nivel unidas con operadores O:

```
expOLogico : expYLogico (OP_O^ expYLogico)* ;
```

De una forma parecida se definen el resto de las expresiones binarias, ya sea el Y lógico,

```
expYLogico : expComparacion (OP_Y^ expComparacion)*;
```

o la comparación,

```
expComparacion
   : expAritmetica
   (
      ( OP_IGUAL^ | OP_DISTINTO^ | OP_MAYOR^ | OP_MENOR^
      | OP_MAYOR_IGUAL^ | OP_MENOR_IGUAL^
      )
      expAritmetica
   )*
  ;
```

o la suma y resta aritmética,

```
expAritmetica : expProducto ((OP_MAS^ | OP_MENOS^) expProducto)* ;
```

o el producto y la división.

```
expProducto : expCambioSigno
              ((OP_PRODUCTO^ | OP_DIVISION^) expCambioSigno)*
            ;
```

A partir del nivel 3 la estructura de las reglas cambia un poco porque las expresiones pasan de ser binarias a ser unarias. Por ejemplo, la regla para los cambios de signo es:

```
expCambioSigno :
   ( OP_MENOS! expPostIncremento
     { ## = #( #[OP_MENOS_UNARIO, "OP_MENOS_UNARIO"], ## ) ;}
   )
   | (OP_MAS!)? expPostIncremento
   ;
```

Como vemos esta regla es un poco más compleja que las anteriores. Lo que estamos haciendo es que si encontramos un signo "menos" ("-") delante de una expresión, ignoramos el signo y enraizamos la expresión con el token imaginario `OP_MENOS_UNARIO`. El signo "más" ("+"), si se encuentra, es directamente ignorado.

El postincremento y postdecremento se manejan fácilmente:

```
expPostIncremento :  expNegacion (OP_MASMAS^|OP_MENOSMENOS^)? ;
```

En el nivel 1 de nuestra jerarquía de expresiones encontramos el operador de negación:

```
expNegacion : (OP_NO^)* expEsUn
            ;
```

Y en el último nivel de nuestra jerarquía tenemos las expresiones que utilizan el operador `esUn`:

```
expEsUn : acceso (RES_ESUN^ tipo)*
        ;
```

Falta por definir qué es un `acceso`.

Accesos

Las expresiones de un lenguaje sirven para manejar *valores*. Estos valores pueden venir dados por:

- Un literal
- Un atributo de la clase actual o de otra clase
- Una llamada a un método o al constructor de la clase actual o de otra clase
- Una variable local

A éste conjunto de valores lo llamaremos `acceso`.

El aspecto general de un acceso es el de un identificador seguido opcionalmente de una serie de "sub-accesos" separados por puntos. No obstante existen algunas excepciones:

- Llamadas a métodos o constructores del objeto actual (solamente se escribe el nombre del método seguido de sus parámetros entre paréntesis. En LeLi no hay un equivalente a "this").
- Los literales, que aunque pueden llamar a algunos métodos (por ejemplo, `"Hola munto".subCadena(4)` devuelve la cadena `"Hola"`) no comienzan por un identificador.
- Accesos realizados con las palabras reservadas `parámetro` y `atributo`, que tampoco empiezan con un identificador.
- Conversiones entre tipos (con `convertir`)
- La palabra reservada `super`.
- Una expresión entre paréntesis.

Además, algunas de las raíces del acceso (las palabras reservadas `parámetro`, `atributo` y `super`) exigen ir seguidas de subaccesos, mientras que el resto pueden aparecer en solitario.

En definitiva, puede decirse que un acceso es "una raíz seguida de cero o más de sub-accesos separados por puntos". Si la raíz necesita obligatoriamente de al menos un acceso entonces irá seguida de uno o más. Es decir:

```
acceso : r1:raizAcceso { ## = #(#[ACCESO], #r1); }
           ( PUNTO! sub1:subAcceso! { ##.addChild(#sub1); } )*
       | r2:raizAccesoConSubAccesos { ## = #(#[ACCESO], #r2); }
           ( PUNTO! sub2:subAcceso! { ##.addChild(#sub2); } )+
       ;
```

Hemos tenido que modificar un poco la manera que tiene ANTLR por defecto de construir los árboles: la raíz es el primer hijo de un nodo ficticio de tipo `ACCESO`, y los `subAccesos` se van añadiendo a la lista de accesos, utilizando `addChild`. Nótese que en la primera alternativa se utiliza un cierre de Kleene mientras que en la segunda se utiliza una clausura positiva

La regla `raizAccesoConSubAccesos` es, como ya hemos indicado antes, la palabra reservada `parámetro`, la palabra reservada `atributo` o la palabra reservada `super`:

```
raizAccesoConSubAccesos
   : RES_PARAMETRO
   | RES_ATRIBUTO
   | RES_SUPER
   ;
```

La regla `raizAcceso` debe reconcer el resto de las posibles raíces que hemos enunciado:

```
raizAcceso : IDENT
           | literal
           | llamada
           | conversion
           | PARENT_AB! expresion PARENT_CE!
           ;
```

Los literales pueden ser de los tres tipos básicos: `Entero`, `Real` y `Cadena`. Los literales cadena pueden ser una cadena normal o bien las palabras reservadas `nl`, `tab` o `com`.

```
literal : LIT_ENTERO
        | LIT_REAL
        | LIT_CADENA
        | LIT_NL
        | LIT_TAB
        | LIT_COM
        | LIT_CIERTO
        | LIT_FALSO
        ;
```

Una llamada es muy fácil de especificar: un identificador seguido de una lista de expresiones entre paréntesis y separadas por comas, en el caso de los métodos, o la palabra reservada `constructor` seguida de los correspondientes parámetros.

```
llamada : IDENT PARENT_AB! listaExpresiones PARENT_CE!
          { ## = #(#[LLAMADA,"LLAMADA"],##); }
        | RES_CONSTRUCTOR^ PARENT_AB! listaExpresiones PARENT_CE!
        ;
protected listaExpresiones
   : ( expresion (COMA! expresion)* )?
     { ## = #(#[LISTA_EXPRESIONES,"LISTA_EXPRESIONES"],##); }
  ;
```

Estas dos reglas ilustran los dos casos básicos en los que viene bien modificar el árbol que se genera automáticamente: cuando no hay un token que pueda ser empleado como raíz y en las listas.

Por su parte, la conversión entre tipos es sencilla:

```
conversion : RES_CONVERTIR^
             PARENT_AB! expresion COMA! tipo PARENT_CE!
           ;
```

Por último, vamos a analizar los subAccesos. Un subAcceso puede ser de tres tipos: una llamada a un método, un acceso a un atributo o la palabra reservada `super`:

```
subAcceso
   : llamada
   | IDENT
   | RES_SUPER
   ;
```

En la segunda opción se utiliza un token imaginario para obtener una jerarquía de accesos. esta jerarquía, junto con la empleada en las llamadas, permite que el acceso `objeto.atributo1.metodo1()` genere un árbol como éste:

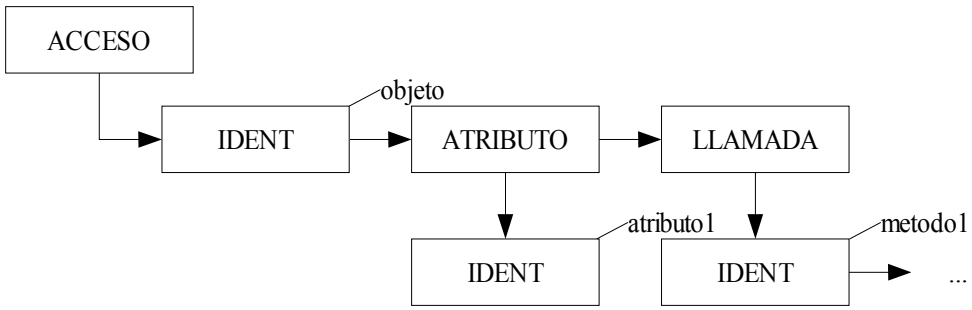

Ilustración 5.3 AST de un acceso

`IDENT`, `ATRIBUTO` y `LLAMADA` son árboles hijos de `ACCESO` (y son hermanos entre ellos). `IDENT` es la *raíz* del acceso, y el resto son *subaccesos*.

Y no el inmanejable árbol degenerado que se produciría sin este enraizamiento:

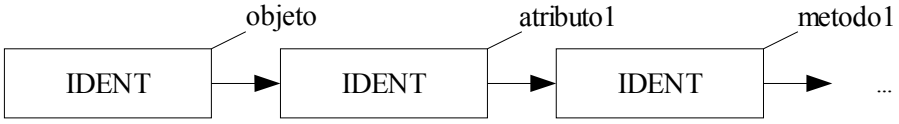

Ilustración 5.4 AST degenerado de un acceso

Antes de terminar con los accesos, una nota final: la que hemos presentado no es más que una de las formas de implementarlos. Pueden encontrarse otras formas de implementarlos en los ejemplos que acompañan a ANTLR.

5.3.7: Instrucciones

Anteriormente hemos hecho referencia a la regla `listaInstrucciones`, que representa una lista de cero o más instrucciones. Su implementación será la habitual para las listas:

```
listaInstrucciones
   : (instruccion)*
     {## = #( #[LISTA_INSTRUCCIONES,"LISTA_INSTRUCCIONES"], ##);}
   ;
```

Por su parte una instrucción puede ser de diferentes tipos: una expresión (una asignación, postincremento o llamada a función) o una instrucción de control (como un bucle o una instrucción si). Las declaraciones de variables también son instrucciones. Además está la instrucción `volver` y la instrucción nula (que solamente tiene un punto y coma).

```
instruccion : instDecVar                // declaración
            | instExpresion             // Instrucción - expresión
            | instMientras              // bucle mientras
            | instHacerMientras         // bucle hacer-mientras
            | instDesde                 // bucle desde
            | instSi                    // Instrucción Si
            | instVolver                // Instrucción volver
            | instNula                  // Instrucción Nula
            ;
```

Un poco más abajo (en el apartado “declaraciones de variables locales”) veremos que esta especificación no es suficiente, pues hay que colocar un predicado sintáctico para evitar una ambigüedad.

Vamos a ver uno a uno los diferentes tipos de instrucciones, del más sencillo al más complicado.

Instrucción nula

La instrucción nula, al tener solamente un `PUNTO_COMA`, es la más sencilla de implementar:

```
instNula : PUNTO_COMA! ;
```

Nótese que al no añadirse ningún nodo al AST con la instrucción nula, la regla produce un árbol vacío, de manera que es totamente filtrada al análisis semántico.

Instrucción volver

La instrucción volver es también muy fácil de comprender. Se trata de la palabra reservada `volver` seguida de un `PUNTO_COMA`. Dicha palabra reservada será el único nodo del árbol.

```
instVolver : RES_VOLVER PUNTO_COMA! ;
```

Intrucción-expresión

Una instrucción-expresión es una expresión seguida de `PUNTO_COMA`.

```
instExpresion: expresion PUNTO_COMA!
              {## = #( #[INST_EXPRESION,"INST_EXPRESION"], ##);}
             ;
```

He añadido la raíz ficticia `INST_EXPRESION` al árbol de las expresiones para que toda instrucción tenga una raíz que la identifique por su tipo.

Declaración de variables locales

Una declaración de variable local a nivel sintáctico no es más que un tipo seguido de uno o varios identificadores, a los que opcionalmente se le puede asignar un valor. Exactamente éso es la regla `tipoListaIdentsValores`, que anteriormente definí al hablar de las definiciones de parámetros de los métodos. No hay que olvidar añadir el `PUNTO_COMA`:

```
instDecVar
{ final AST raiz = #[INST_DEC_VAR,"variable"]; }
   : listaDeclaraciones[raiz,true] PUNTO_COMA! ;
```

Atención: introducir esta regla provoca que el analizador no diferente correctamente las instrucciones-expresion de las declaraciones, porque los conjuntos `PRIMERO(expresion)` y `PRIMERO(instDecVar)` contienen a `IDENT`, y el analizador es SLL(2) pero no LL(2), lo que

hace que en algunos momentos no sea suficiente con k=2.

Es necesario añadir un predicado sintáctico a `instruccion` para solucionar esta ambigüedad:

```
instruccion : (tipo IDENT)=>instDecVar // declaración
            | instExpresion            // Instrucción - expresión
            | instMientras             // bucle mientras
            | instHacerMientras        // bucle hacer-mientras
            | instDesde                // bucle desde
            | instSi                   // Instrucción Si
            | instVolver               // Instrucción volver
            | instNula                 // Instrucción Nula
            ;
```

Bucles

Los bucles `mientras` y `hacer-mientras` son muy fáciles de implementar:

```
instMientras : RES_MIENTRAS^ PARENT_AB! expresion PARENT_CE!
               LLAVE_AB! listaInstrucciones LLAVE_CE!
             ;
instHacerMientras : RES_HACER^ LLAVE_AB! listaInstrucciones LLAVE_CE!
                    RES_MIENTRAS PARENT_AB! expresion PARENT_CE!
                    PUNTO_COMA!
                  ;
```

El bucle `desde` es un poco más complicado porque tras la palabra reservada desde, entre los paréntesis, hay tres listas de expresiones separadas por por `COMA`s, y pueden no tener ningún elemento. Ya hemos implementado antes éste tipo de listas al hablar de llamadas a métodos: la regla `listaExpresiones` es exactamente lo que buscamnos. Por lo tanto un bucle `desde` es lo siguiente:

```
instDesde : RES_DESDE^ PARENT_AB! listaExpresiones PUNTO_COMA!
                                  listaExpresiones PUNTO_COMA!
                                  listaExpresiones PARENT_CE!
            LLAVE_AB! listaInstrucciones LLAVE_CE!
          ;
```

Instrucción “si”

La instrucción `si` parece sencilla, y sin embargo ha resultado ser la más difícil de codificar . Comienza con la palabra reservada `RES_SI`, y tras ella una expresión y una lista de instrucciones. Después hay cero o más alternativas, que pueden ser “normales” (una barra, una expresión y una lista de instrucciones) o la alternativa “otras”. En principio podemos escribir algo así:

```
instSi
   : RES_SI^ PARENT_AB! expresion PARENT_CE!
     LLAVE_AB! listaInstrucciones LLAVE_CE!
     (altSiNormal)*
     (altSiOtras)?
   ;

protected altSiNormal : BARRA_VERT^ PARENT_AB! expresion PARENT_CE!
                        LLAVE_AB! listaInstrucciones LLAVE_CE!
                      ;
protected altSiOtras : BARRA_VERT! RES_OTRAS^
                       LLAVE_AB! listaInstrucciones LLAVE_CE!
                     ;
```

Nótese que mientras que las alternativas normales tienen como raíz el token `BARRA_VERT`, la alternativa otras la tiene en `RES_OTRAS`.

Éste conjunto de reglas, que debería ser interpretado perfectamente con k=2 por ANTLR, no lo es. Posiblemente por un bug de implementación en ANTLR, el lookahead se reduce a 1 al evaluar la sección `(altOtras)?` de la regla, lo que provoca un incómodo mensaje de aviso de indeterminismo en la sub regla `(altSiNormal)*`. De todas maneras las instrucciones condicionales se reconocen bien, así que el mensaje de error puede eliminarse simplemente poniendo la opción `generateAmbigWarnings` a `false`:

```
instSi :
   RES_SI^ PARENT_AB! expresion PARENT_CE!
   LLAVE_AB! listaInstrucciones LLAVE_CE!
   (options {generateAmbigWarnings=false;}: altSiNormal)*
   (altSiOtras)?
   ;
```

Sin embargo a mí no me gusta desactivar `generateAmbigWarnings`. Prefiero tener una gramática sin ambigüedades. Hay varias gramáticas equivalentes a la anterior que no generan mensajes de aviso de ambigüedad. Lo primero que puede ocurrirsenos es utilizar un predicado semántico, así:

```
instSi  :
   RES_SI^ PARENT_AB! expresion PARENT_CE!
   LLAVE_AB! listaInstrucciones LLAVE_CE!
   alternativasSi;

alternativasSi
   { boolean hayOtras=false; } // variable local
   :
   ( {hayOtras==false}? altSiNormal
   | {hayOtras==false}? altSiOtras { hayOtras=true; }
   )*
   ;
```

En éste ejemplo se declara una variable local (nótese que hay que declararla *antes* de los dos puntos de la regla) y se utiliza como predicado semántico: solamente se pueden añadir alternativas a la instrucción `si` si no se ha llegado a la alternativa `otras`.

El problema que tiene utilizar predicados semánticos y variables locales es que la gramática ya no es independiente del lenguaje. Por ejemplo, en C++ no existe el tipo `boolean` (se utiliza la versión más reducida `bool`). Es recomendable buscar una alternativa independiente del lenguaje generado.

Hay muchas otras maneras de hacerlo que sí son independientes del lenguaje generado. Aquí presento una, que se basa en el viejo método de la recursión.

```
instSi :
   RES_SI^ PARENT_AB! expresion PARENT_CE!
   LLAVE_AB! listaInstrucciones LLAVE_CE!
   alternativasSi
   ;

protected alternativasSi
   : altSiNormal alternativasSi
   | altSiOtras
   | /* nada */
   ;
```

Éste conjunto de reglas funciona perfectamente igual que el anterior, es casi igual de fácil de entender y además no provoca ninguna situación de ambigüedad.

Sección 5.4: Fichero LeLiParser.g

El fichero `LeLiParser.g` al completo tendrá el siguiente aspecto:

```
header{

package leli;

/*-----------------------------------------------*\
 | Un intérprete para un Lenguaje Limitado(LeLi) |
 | _____________________________________________ |
 |                                               |
 |              ANALISIS SINTÁCTICO              |
 | _____________________________________________ |
 |                                               |
 |            Enrique J. Garcia Cota             |
\*-----------------------------------------------*/

}

/**
 *  El objeto que permite todo el analisis sintactico.
 */
class LeLiParser extends Parser;

options
{
   k = 2;
   importVocab = LeLiLexerVocab;
   exportVocab = LeLiParserVocab;
   buildAST = true;
}

tokens
{
// Tokens imaginarios para enraizar listas
   PROGRAMA ;
   LISTA_MIEMBROS;
   LISTA_EXPRESIONES;
   LISTA_INSTRUCCIONES;

// Tokens imaginarios que se utilizan cuando no hay raíces adecuadas
   OP_MENOS_UNARIO;
   INST_EXPRESION;
   INST_DEC_VAR;
   LLAMADA;
   ATRIBUTO;

// Otros
   TIPO_VACIO;
}

/**
 * Un programa esta formado por una lista de definiciones de clases.
 */
programa : (decClase)+
           {## = #( #[PROGRAMA, "PROGRAMA"] ,##);}
       ;
```

```
/** Definición de una clase. */
decClase : RES_CLASE^ IDENT clausulaExtiende
           LLAVE_AB! listaMiembros LLAVE_CE!
         ;

/** Cláusula "extiende" **/
clausulaExtiende : RES_EXTIENDE^ IDENT
                 | /*nada*/
                   { ## = #( #[RES_EXTIENDE,"extiende"],
                             #[IDENT, "Objeto"] ); }
                 ;

/** Auxiliar para definición de clase */
protected listaMiembros
   : (decMetodo|decConstructor|decAtributo)*
     { ## = #( #[LISTA_MIEMBROS, "LISTA_MIEMBROS"] ,##);}
   ;

/**
 * Declaración de los diferentes tipos que se pueden usar en LeLi
 */
tipo : TIPO_ENTERO   // tipo Entero
     | TIPO_REAL     // tipo Real
     | TIPO_BOOLEANO // tipo Booleano
     | TIPO_CADENA   // tipo Cadena
     | IDENT         // tipo no básico
     ;

declaracion ! // desactiva construcción por def. del AST
[AST r, AST t, boolean inicializacion] // parámetros
{
   AST raiz = astFactory.dupTree(r);     // copia del arbol
   raiz.addChild(astFactory.dupTree(t)); // copia del árbol
}
   : i1:IDENT
     {
        raiz.addChild(#i1);
        ## = raiz;
     }
   | { inicializacion }? // pred. semántico
     i2:IDENT OP_ASIG valor:expresion
     {
        raiz.addChild(#i2);
        raiz.addChild(#valor);
        ## = raiz;
     }
   | { inicializacion }?
     i3:IDENT parentAb li:listaExpresiones parentCe
     {
        raiz.addChild(#i3);
        raiz.addChild(#li);
        ## = raiz;
     }
   ;

listaDeclaraciones [AST raiz, boolean inicializacion]
   : t:tipo!
```

```
     declaracion[raiz,#t,inicializacion]
     (COMA! declaracion[raiz,#t,inicializacion])*
   ;

/**
 * Definición de un atributo (abstracto o no abstracto)
 */
decAtributo
{ boolean abstracto = false; }
   : raiz:RES_ATRIBUTO^
     ( a:RES_ABSTRACTO { abstracto=true; } )?
     listaDeclaraciones[#raiz, abstracto] PUNTO_COMA!
   ;

/** Definición de un constructor */
decConstructor : RES_CONSTRUCTOR^ PARENT_AB! listaDecParams PARENT_CE!
                 LLAVE_AB! listaInstrucciones LLAVE_CE!
               ;

/** Definición de un método normal */
decMetodo
   : RES_METODO^ (RES_ABSTRACTO)? tipoRetorno IDENT
     PARENT_AB! listaDecParams PARENT_CE!
     LLAVE_AB! listaInstrucciones LLAVE_CE!
   ;

/** Regla auxiliar que codifica el tipo de retorno de un método */
protected tipoRetorno
   : tipo
   | /* nada */ {## = #[TIPO_VACIO,"TIPO_VACIO"]; }
   ;

/** Lista de parámetros. */
listaDecParams
{ final AST raiz = #[RES_PARAMETRO,"parámetro"] ;}
   :
   ( listaDeclaraciones[raiz, false]
     ( PUNTO_COMA! listaDeclaraciones[raiz, false])*
   )? // opcional
    ;

/**
 * Regla que sirve para empezar a reconocer las expresiones de LeLi
 */
expresion: expAsignacion;

/** Asignaciones (nivel 9) */
expAsignacion : expOLogico (OP_ASIG^ expOLogico)? ;

/** O lógico (nivel 8) */
expOLogico : expYLogico (OP_O^ expYLogico)* ;

/** Y lógico (nivel 7) */
expYLogico : expComparacion (OP_Y^ expComparacion)* ;

/** Comparación (nivel 6) */
expComparacion
   : expAritmetica
```

```
   (
      ( OP_IGUAL^ | OP_DISTINTO^ | OP_MAYOR^ | OP_MENOR^
      | OP_MAYOR_IGUAL^ | OP_MENOR_IGUAL^
      )
      expAritmetica
   )*
  ;

/** Suma y resta aritmética (nivel 5) */
expAritmetica : expProducto ((OP_MAS^ | OP_MENOS^) expProducto)* ;

/** Producto y división (nivel 4) */
expProducto : expCambioSigno
              ((OP_PRODUCTO^ | OP_DIVISION^) expCambioSigno)*
            ;

/** Cambio de signo (nivel 3) */
expCambioSigno :
   ( OP_MENOS! expPostIncremento
     { ##=#(#[OP_MENOS_UNARIO, "OP_MENOS_UNARIO"], ##) ;}
   )
   | (OP_MAS!)? expPostIncremento
   ;

/** Postincremento y postdecremento (nivel 2) */
expPostIncremento :  expNegacion (OP_MASMAS^|OP_MENOSMENOS^)? ;

/** Negación y accesos (nivel 1) */
expNegacion : (OP_NO^)* expEsUn
            ;

/** EsUn + accesos (nivel 0) **/
expEsUn : acceso (RES_ESUN^ tipo)*
        ;

/**
 * Regla que permite reconocer los accesos de las expresiones de LeLi.
 * Los accesos son los valores que se utilizan en las expresiones:
 * literales, variables, llamadas a métodos, etc.
 */
acceso : r1:raizAcceso { ## = #(#[ACCESO], #r1); }
         ( PUNTO! sub1:subAcceso! { ##.addChild(#sub1); } )*
       | r2:raizAccesoConSubAccesos { ## = #(#[ACCESO], #r2); }
         ( PUNTO! sub2:subAcceso! { ##.addChild(#sub2); } )+
       ;

/**
 * Raíz de los accesos que no son llamadas a un método de la
 * clase "actual"
 */
raizAcceso : IDENT
           | literal
           | llamada
           | conversion
           | PARENT_AB! expresion parentCe
           ;
/**
 * Raíz de los accesos que no son llamadas a un método de la
```

```
 * clase "actual" y que obligatoriamente van sucedidos de un subacceso
 */
raizAccesoConSubAccesos
   : RES_PARAMETRO
     | RES_ATRIBUTO
     | RES_SUPER
     ;

/** Regla que reconoce los literales */
literal : LIT_ENTERO
        | LIT_REAL
        | LIT_CADENA
        | LIT_NL
        | LIT_TAB
        | LIT_COM
        | LIT_CIERTO
        | LIT_FALSO
        ;

/**
 * Esta regla se utiliza tanto para representar:
 * 1) Una llamada a un método del objeto actual
 * 2) Un subacceso en forma de llamada.
 * 3) Una llamada a un constructor del objeto actual
 * 4) Un subacceso en forma de constructor.
 */
llamada : IDENT PARENT_AB! listaExpresiones PARENT_CE!
          { ## = #(#[LLAMADA,"LLAMADA"],##); }
        | RES_CONSTRUCTOR^ PARENT_AB! listaExpresiones PARENT_CE!
        ;

/**
 *  Regla auxiliar que reconoce los parámetros de una llamada
 *  a un método y la inicialización del bucle "desde"
 */
protected listaExpresiones
   : ( expresion (COMA! expresion)* )?
     { ## = #(#[LISTA_EXPRESIONES, "LISTA_EXPRESIONES"],##); }
   ;

/** Conversión entre tipos */
conversion : RES_CONVERTIR^
             PARENT_AB! expresion COMA! tipo PARENT_CE!
           ;

/** Regla que reconoce los accesos a atributos y métodos de un objeto. */
subAcceso
   : llamada
   | IDENT
   | RES_SUPER
   ;

/** Una lista de 0 o más instrucciones */
listaInstrucciones
   : (instruccion)*
     {## = #( #[LISTA_INSTRUCCIONES,"LISTA_INSTRUCCIONES"], ##);}
   ;
```

```
/**
 * Las instrucciones. Pueden ser expresiones, instrucciones de control,
 * declaraciones de variables locales o la instrucción volver.
 */
instruccion : (tipo IDENT)=>instDecVar // declaración
            | instExpresion            // Instrucción - expresión
            | instMientras             // bucle mientras
            | instHacerMientras        // bucle hacer-mientras
            | instDesde                // bucle desde
            | instSi                   // Instrucción Si
            | instVolver               // Instrucción volver
            | instNula                 // Instrucción Nula
            ;

/** Instrucción nula */
instNula : PUNTO_COMA! ;

/** Instrucción volver */
instVolver : RES_VOLVER PUNTO_COMA! ;

/** Instrucción-expresión */
instExpresion: expresion PUNTO_COMA!
             {## = #( #[INST_EXPRESION,"INST_EXPRESION"], ##);}
             ;

/** Declaración de variables locales */
instDecVar
{ final AST raiz = #[INST_DEC_VAR,"variable"]; }
   : listaDeclaraciones[raiz,true] PUNTO_COMA! ;

/** Bucle "mientras" */
instMientras : RES_MIENTRAS^ PARENT_AB! expresion PARENT_CE!
               LLAVE_AB! listaInstrucciones LLAVE_CE!
             ;

/** Bucle "hacer-mientras" */
instHacerMientras : RES_HACER^ LLAVE_AB! listaInstrucciones LLAVE_CE!
                    RES_MIENTRAS PARENT_AB! expresion PARENT_CE!
                    PUNTO_COMA!
                  ;

/** Bucle "desde" */
instDesde : RES_DESDE^ PARENT_AB! listaExpresiones PUNTO_COMA!
                                  listaExpresiones PUNTO_COMA!
                                  listaExpresiones PARENT_CE!
            LLAVE_AB! listaInstrucciones LLAVE_CE!
          ;

/** Instruccion "si" muy parecida a la del lenguaje LEA. */
instSi :
   RES_SI^ PARENT_AB! expresion PARENT_CE!
   LLAVE_AB! listaInstrucciones LLAVE_CE!
   alternativasSi
   ;

/**
 * Auxiliar (reconoce las alternativas de la instrucción "si"
 * sin warnings de ambigüedad)
```

```
 */
protected alternativasSi
   : altSiNormal alternativasSi
   | altSiOtras
   | /* nada */
   ;

/** Auxiliar (alternativa normal de la instrucción "si") */
protected altSiNormal : BARRA_VERT^ PARENT_AB! expresion PARENT_CE!
                        LLAVE_AB! listaInstrucciones LLAVE_CE!
                      ;

/** Auxiliar (alternativa final "otras" de la instrucción "si") */
protected altSiOtras : BARRA_VERT! RES_OTRAS^
                       LLAVE_AB! listaInstrucciones LLAVE_CE!
                     ;
```

Sección 5.5: Compilación del analizador

No hay grandes diferencias entre compilar el analizador léxico y el sintáctico. Supondremos de nuevo que el paquete leli se encuentra en el directorio `c:\languajes\leli`. Sitúese en `c:\languajes\leli` y escriba lo siguiente[38]:

```
c:\languajes\leli\> java antlr.Tool LeLiParser.g
```

En el directorio `leli` deberán haber aparecido los siguientes nuevos ficheros:

- `LeLiParser.java`
- `LeLiParserVocabTokenTypes.java`
- `LeLiParserVocabTokenTypes.txt`

A estas alturas debería ser evidente la función de cada uno de éstos ficheros: el primero es el analizador propiamente dicho, mientra que los otros dos le sirven para pasarle al analizador semántico los tokens.

Para poder compilar `LeLiParser.java` será necesario que `LeLiParserVocabTokenTypes.java` esté compilado, **y también `LeLiLexerVocabTokenTypes.java`**. En general, para poder compilar el analizador de un nivel es necesario que todos los ficheros de los niveles anteriores estén perfectamente compilados y disponibles. Así que, si no lo ha hecho ya, compile el analizador sintáctico:

```
c:\languajes\leli\> java antlr.Tool LeLiParser.g
```

La herramienta antlr.Tool admite además varias opciones de compilación. Quizás una de las más interesantes sea trace, que permite seguir la traza del analizador:

```
c:\languajes\leli> java antlr.Tool -trace LeLiLexer.g
```

Una vez generados todos los ficheros java necesarios, se podrán compilar con la orden:

```
c:\languajes\leli> javac *.java
```

El compilador de java no debería mostrar mensaje alguno, significando que todo ha ido bien.

Si se requiere información de debug (algo muy deseable en las primeras fases del desarrollo) se debe añadir la opción -g al compilador de java:

```
c:\languajes\leli> javac -g *.java
```

38 Como siempre, para poder utilizar ANTLR igual que en los ejemples será necesario que esté correctamente instalado.

Sección 5.6: Ejecución: modificaciones en la clase Tool

5.6.1: Introducción

En el capítulo anterior hemos empezado a trabajar con la clase `leli.Tool`.

De momento hemos implementado los niveles léxico y sintáctico, y la construcción del AST inicial. Aunque no hayamos implementado todas las fases, ya se pueden empezar a hacer cosas más "interesantes" que simplemente imprimir el flujo de tokens. Por ejemplo, podemos mostrar por pantalla el AST (en lugar de un flujo de tokens, mostraremos una jerarquía de nodos AST).

En este apartado explicaré cómo implementar una clase Tool "incompleta", que se limite a analizar un fichero escrito en lenguaje LeLi y muestre su AST por pantalla. Imprimiremos un programa muy sencillo en la pantalla, como el siguiente:

```
clase Inicio
{
     método inicio()
     {
        Sistema.Imprime("¡Hola mundo!"+nl);
     }
}
```

ANTLR proporciona dos maneras de obtener una representación visual de un AST: con una cadena de texto simple, utilizando una notación parecida a la del lenguaje LISP, y una mucho más comprensible para los humanos, basada en un componente SWING de java.

5.6.2: Interpretando la línea de comandos: el paquete antlraux.clparse

Presentación

La antigua versión de `leli.Tool` no es muy flexible. Para modificar la forma en que se realizaba el análisis (por ejemplo, para activar/desactivar el análisis sintáctico) es necesario modificar el código de `leli.Tool` y recompilar.

En lugar de utilizar esta técnica, es mucho más simple utilizar la línea de comandos. Con unos cuantos parámetros, podremos controlar perfectamente el comportamiento de nuestra herramienta, sin tener que recompilar a cada vez.

Cuando se trata de un par de comandos diferentes es sencillo interpretarlos "a mano". Sin embargo, cuando el número de comandos a controlar aumenta, y cada uno tiene un número y un tipo de parámetros diferentes, hacerlo así produce un código realmente desagradable y difícilmente mantenible; varias instrucciones `switch` encadenadas llenas de condicionales (`if-elseif-elseif-else...`).

Jaime Cruz Mena y yo (Enrique José García Cota) nos encontramos con dicho problema mientras trabajábamos en una práctica de java, en los laboratorios del ISEP (*Institute Superieur d'Electronique de Paris*), durante el primer semestre de 2002. Decidimos implementar un paquete que permitiera interpretar la línea de comandos con pocas líneas de código, de una manera simple y elegante. Llamamos a dicho paquete `clparse` (por *Command Line PARSE*).

He incluido el paquete `clparse` dentro de `antlraux`, de forma que su nuevo nombre es `antlraux.clparse`.

Funcionamiento

El funcionamiento de antlraux.clparse es sencillo: todo gira en torno a la clase `CommandLineParser`. Ésta será la clase encargada de reconocer la línea de comandos y ejecutar las acciones oportunas. La mayoría de los usuarios tendrán suficiente con crear una sola instancia de `CommandLineParser`, aunque nada impide crear varias instancias diferentes.

Una vez creado el `CommandLineParser`, (pasándole como parámetro el nombre de la aplicación java que va a interpretar la línea de comandos) es necesario especificarle los diferentes comandos que puede reconocer, así como los parámetros que deben tomar y las acciones a efectuar con ellos.

Todo ello se realiza utilizando el método `addComand`. Éste tiene el siguiente prototipo:

```
public void addCommand( Object executer, String commandName,
                        String methodName, String paramTypes,
                        String description )
throws CommandLineParserException
```

Vamos a ver qué significa cada parámetro.

`methodName` será el nombre del nuevo comando a añadir. Por ejemplo, si un comando es "-f" entonces el nombre del comentario es "-f". El siguiente parámetro, `paramTypes`, sirve para especificar un parámetro. Así, si "-f" acepta una cadena como parámetro (como un nombre de fichero) entonces el parámetro `paramTypes` será la cadena "s" ("s" por "string", cadena). Es posible añadir parámetros de tipo cadena (s), entero(i), carácter(c), flotante(f) o booleano (b). También, aunque es infrecuente, es posible añadir más de un parámetro a cada comando, añadiendo simplemente más caracteres a la cadena. De esta manera, si "-f" tomara como parámetro una cadena, un entero y un booleano `paramTypes` sería "sib".

Los parámetros executer y `methodName` sirven para codificar las acciones a realizar tras reconocer un comando en la línea de comandos. executer es un objeto cualquiera (una instancia cualquiera de una clase), y methodName es el nombre de uno de los métodos de dicho objeto. Lo que hará el `CommandLineParser` cuando reconozca un comando será, en primer lugar, transformar los parámetros de dicho comando en variables de los tipos adecuados (`String`, `Integuer`, `Character`, `Float` o `Boolean`) e invocar el método llamado "methodName" del objeto executer, pasándole como parámetros las variables anteriormente mencionadas. Por lo tanto es necesario que el tipo de los parámetros del método coincida con los definidos en `paramTypes`.

Por último, description es una descripción de lo que hace el parámetro. Esta cadena será utilizada para generar el mensaje de ayuda.

Una vez se han añadido las especificaciones de comandos requeridas, la línea de comandos se puede leer invocando `parseWhilePossible`. El `CommandLineParser` "traducirá" la línea de comandos a una serie de tareas a realizar, y podrá ejecutarlas con el método `executeWhilePossible`.

Si en algún momento de todo el proceso se produce un error, se lanzará una excepción de tipo `CommandLineParserException`.

Ejemplo

Vamos a ver todo esto con un sencillo ejemplo. Considérese la siguiente clase de java:

```
// Es MUY importante que la clase ejecutora sea declarada pública
```

```
public class Persona
{
   private String Nombre="";
   private boolean esVaron=false;
   private int edad = 0;

   public Persona() {}
   public void ajustarNombre(String s)
   { Nombre = s; }

   // Hay que utilizar las clases de java (Boolean, Character, Float, Integer)
   // y *NO* los tipos básicos del lenguaje (boolean, char, float, int)
   public void ajustarSexo( Boolean b )
   { esVaron = b.booleanValue(); }

   public void ajustarEdad( Integer i )
   { edad = i.intValue(); }

   public String toString()
   {
      return Nombre + " es " + (esVaron?"un hombre de " : "una mujer de") +
             edad + " años");
   }
}
```

Ahora imaginemos que el método `main` está en la clase `Main`, y que tiene el siguiente aspecto:

```
import antlraux.clparse.*;

public class Main
{
   public static void main(String [] args)
   {
      CommandLineParser clp = new CommandLineParser( "Main" );
      Persona p = new Persona(); // "p" será el ejecutor de todos los comandos
      try
      {
         clp.addCommand(p, "-nombre", "ajustarNombre", "s", "Cambia el nombre");
         clp.addCommand(p, "-edad",   "ajustarEdad",   "i", "Cambia la edad");
         clp.addCommand(p, "-varon",  "ajustarSexo",   "b", "Cambia el sexo");

         clp.parseWhilePossible();
         clp.executeWhilePossible();
      } catch (CommandLineParserException clpe) {
         System.err.println(clpe);
         System.err.print("Usage: ");
         System.err.println(clp.getUsageString(true));
      }
      System.out.println(p.toString());
   }
}
```

He aquí una muestra de lo que obtendremos si ejecutamos el programa repetidas veces:

```
$ java Main -nombre Manuel
Manuel es una mujer de 0 años
$ java Main -nombre Juan -varon true -edad 30
Manuel es un hombre de 30 años
$ java Main -nombre Mercedes -edad false
Could not parse an integer from false
Usage: java Main [commands]

   -nombre String
     Cambia el nombre
   -edad Integuer
     Cambia la edad
   -varon Booleano
     Cambia el sexo
$
```

Como puede verse `CommanLineParser` se encarga incluso de generar un mensaje de uso adecuado, si se precisa. También se encarga de generar los errores adecuados si es necesario (por ejemplo, si se esperaba un entero y no se pudo leer).

El paquete `clparse` no está desarrollado con ANTLR: el análisis de cada elemento de la línea de comandos se hizo "a mano".

5.6.3: La línea de comandos inicial: el método leli.Tool.LeeLC()

Esta segunda versión de `leli.Tool` aceptará solamente de tres comandos:

- `-ventana booleano` : Activa/desactiva la ventana del AST (por defecto desactivada)
- `-f cadena` : Fija el nombre del fichero a reconocer (campo obligatorio)
- `-imprime booleano` : Activa/desactiva la impresión en modo texto del AST (por defecto desactivado).

La clase `Tool` proporcionará un método, llamado `LeeLC`, que tomará como parámetro la línea de comandos (`String [] args`) y la leerá, configurándose adecuadamente:

```
public class Tool
{
   public String mensajeAyuda="";
   public boolean mostrarVentana=false;
   public String nombreFichero="";
   public FileInputStream fis=null;
   public boolean imprimeArbol=false;

   public void fijarMostrarVentana(Boolean B)
   { mostrarVentana=B.booleanValue(); }

   public void fijarNombreFichero(String s)
   { nombreFichero=s;   }

   public void fijarImprimeArbol(Boolean B)
   { imprimeArbol = B.booleanValue(); }

   public Tool ()
   { }

   public void leeLC(String args[])
   throws CommandLineParserException
```

```
{
   CommandLineParser clp =
      new CommandLineParser("leli.Tool", args);
   clp.addCommand(this, "-ventana", "fijarMostrarVentana", "b",
                  "Enseña o no la ventana del AST (defecto no)");
   clp.addCommand(this, "-f", "fijarNombreFichero", "s",
                  "Fija el nombre del fichero a reconocer");
   clp.addCommand(this, "-imprime", "fijarImprimeArbol", "b",
                  "Imprime el AST en la salida estándar");
   mensajeAyuda = clp.getUsageMessage(true);
   clp.parseWhilePossible();
   clp.executeWhilePossible();

   if( nombreFichero==null ||
       nombreFichero.equals("") )
   {
      throw new
        CommandLineParserException("Se necesita un nombre de fichero");
   }

   try
   {
      fis = new FileInputStream(nombreFichero);
   }catch (FileNotFoundException fnfe){
        throw new CommandLineParserException(
           "El fichero '"+nombreFichero+"' no se pudo abrir");
   }
}
```

Nótese que en este caso el "ejecutor" que manejará el `CommandLineParser` es "this", es decir, la propia `Tool`.

5.6.4: El método leli.Tool.imprimeAyuda()

Este método es muy sencillo: su única función es imprimir un mensaje de uso, utilizando el mensaje de ayuda suministrado por el `CommandLineParser` y obtenido en `LeeLC` (atributo `mensajeAyuda`):

```
public void imprimeAyuda()
{
   System.err.println(mensajeAyuda);
}
```

5.6.5: El nuevo método main

Dado que ahora toda la funcionalidad está dentro de la propia clase `Tool`, el método main solamente tiene que crear una instancia de `leli.Tool` y llamar a sus métodos:

```
public static void main(String args[])
{

   Tool t = new Tool();

   try {
      t.leeLC(args); // Lee la línea de comandos
      t.trabaja();   // Ejecuta los comandos
   } catch (CommandLineParserException clpe) {
      System.err.println(clpe.getMessage());
      t.imprimeAyuda();
   }
}
```

Hemos hecho que las excepciones de la línea de comandos, que solamente pueden ser lanzadas por `leeLC`, se recuperen tras llamar a `Tool.trabaja`. Así, si hay un error durante la lectura de la línea de comandos, no se ejecuta ningún análisis. Este método no debería cambiar más de ahora en adelante.

Solamente nos queda por estudiar el método `trabaja`.

5.6.6: El método leli.Tool.trabaja()

En este nuevo método es en el que se realiza todo el "trabajo" de análisis, de acuerdo con los comandos que hemos proporcionado a la herramienta.

Pasos

Hay que tener claro lo que la herramienta debe hacer en este caso. Debe:

1. Leer comandos de la línea de comandos. Entre ellos debe aparecer el nombre del fichero a leer.
2. Crear un analizador léxico de LeLi, utilizando para crearlo el fichero al analizador léxico en forma de `FileInputStream`.
3. Proporcionar el nombre del fichero al analizador léxico con el método `setFilename()`, y especificar `LexInfoToken` como clase para el analizador léxico utilizando setTokenObjectClass
4. Crear un analizador sintáctico de LeLi, utilizando para crearlo el analizador léxico que se creó en el paso 2.
5. Proporcionar el nombre del fichero al analizador sintáctico con el método `setFilename()`.
6. Lanzar la primera regla del analizador sintáctico. En nuestro caso es `programa()`.
7. Obtener el AST generado por el analizador sintáctico con el método `getAST()`.
8. Mostrar el AST por pantalla.

El paso 1 ya está implementado en el método `LeeLC`. El método trabaja se encargará de los pasos del 1 al 8. Nota: si no se tiene claro cualquier paso entre el 1 y el 6 le sugiero que relea los capítulos 1 y 2.

Si ocurre cualquier error, la clase tool deberá mostrar un mensaje acorde con él.

Inicio

La clase comenzará con las directivas de importación y la declaración del paquete:

```
package leli; // Tool forma parte del paquete LeLi

import java.io.*;

import antlr.collections.AST;
import antlr.collections.impl.*;
import antlr.debug.misc.*;
import antlr.*;

import antlraux.clparse.*;
import antlraux.util.*;

public class Tool
{
   ... <métodos y atributos utilizados por LeeLC>
   public void LeeLC(String[] args) { ... }
   public class Tool() {} // Constructor de la clase (no hace nada)
   public void trabaja()
   {
      try{
      ...
      catch (TokenStreamException tse)
      {
         tse.printStackTrace(System.err);
      }
      catch (RecognitionException re)
      {
         re.printStackTrace(System.err);
      }
   }
```

Como puede verse, el método trabaja dispone de sus propios gestores de errores, independientes de los encontrados en el método main.

Nivel léxico

En los pasos 2 y 3 se prepara el nivel semántico. Solamente hay que copiar lo que ya hemos implementado anteriormente, cambiando ligeramente el nombre del fichero:

```
        System.out.println("Reconociendo el fichero '"+nombreFichero+"'");

        // PASOS 4 y 5. Crear analizador sintáctico y pasarle
        // nombre fichero
        LeLiLexer lexer = new LeLiLexer(fis);
        lexer.setFilename(nombreFichero);
        lexer.setTokenObjectClass("antlraux.util.LexInfoToken");
```

Nivel sintáctico

En el nivel sintáctico solamente tendremos que crear el analizador y lanzar el análisis.

```
        // PASOS 4 y 5. Crear analizador sintáctico y pasarle nombre Fichero
        LeLiParser parser = new LeLiParser(lexer);
        parser.setFilename(nombreFichero);

        // PASO 6. Comenzar el análisis
        parser.programa();
```

Obtención del AST

Obtendremos el AST invocando el método `getAST()` del parser.

```
        // PASO 7. Obtener el AST
        AST ast = parser.getAST();

        // PASO 7b. Activar la impresión de información extra a toString()
        BaseAST.setVerboseStringConversion(
           true,
           LeLiParser._tokenNames);
```

El paso 7b hace que se añada información adicional a los nodos AST cuando se impriman por pantalla.

Representación textual del AST

Los métodos `AST.toStringTree()` y `AST.toStringList()` sirven para representar un AST en forma de cadena. Si bien esta representación es útil en ciertas ocasiones, lo más normal es que nos decantemos por la representación gráfica, más cómoda de entender para un humano y más útil.

```
        // PASO 8. Mostrar el AST por pantalla (modo texto)
        if(imprimeArbol==true)
        {
           System.out.println(ast.toStringList());
        }
```

Representación gráfica del AST

Presentar la cadena que representa el árbol está bien, pero no es muy manejable para los humanos. La versión de java de ANTLR permite utilizar un sistema de representación muchísimo más interesante, en una ventana SWING. Para verlo, podemos añadir el paso 9 de representación gráfica del árbol AST:

```
// PASO 9. Representación en ventana del AST
        final ASTFrame frame = new ASTFrame(Filename, ast);
        frame.setVisible(true);
```

Una vez compilada la clase Tool, al pasarle el siguiente código:

```
// Fichero test.leli
clase Inicio
{
     método inicio()
     {
        Sistema.Imprime("¡Hola mundo!"+nl);
     }
}
```

y ejecutando la siguiente línea de comandos:

```
c:\> java leli.Tool test.leli
```

Obtendremos esta ventana:

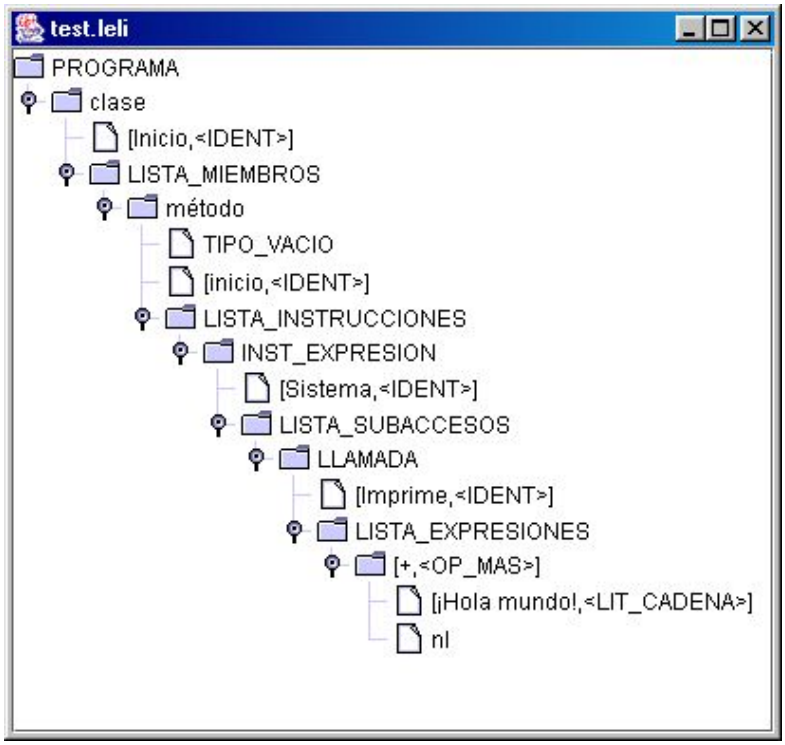

Ilustración 5.5 Ventana SWING mostrando el AST de "Hola mundo"

Una ventana es mucho más útil que una simple cadena porque:

- Presenta los datos de una manera más ordenada y comprensible para las personas.
- Cada nodo puede comprimirse o expandirse hasta las hojas.
- La ventana es de tamaño ajustable.

Código completo

A continuación se lista el código completo de la clase `leli.Tool` actual.

```
package leli; // Tool forma parte del paquete LeLi

import java.io.*;

import antlr.collections.AST;
import antlr.collections.impl.*;
import antlr.debug.misc.*;
import antlr.*;

import antlraux.clparse.*;
import antlraux.util.*;

public class Tool
{
   public String mensajeAyuda="";
   public boolean mostrarVentana=false;
   public String nombreFichero="";
   public FileInputStream fis=null;
   public boolean imprimeArbol=false;

   public void fijarMostrarVentana(Boolean B)
   { mostrarVentana=B.booleanValue(); }
```

```
public void fijarNombreFichero(String s)
{ nombreFichero=s;   }

public void fijarImprimeArbol(Boolean B)
{ imprimeArbol = B.booleanValue(); }

public Tool ()
{ }

public void leeLC(String args[])
throws CommandLineParserException
{
   CommandLineParser clp =
      new CommandLineParser("leli.Tool", args);
   clp.addCommand(this, "-ventana", "fijarMostrarVentana", "b",
                  "Enseña o no la ventana del AST (defecto no)");
   clp.addCommand(this, "-f", "fijarNombreFichero", "s",
                  "Fija el nombre del fichero a reconocer");
   clp.addCommand(this, "-imprime", "fijarImprimeArbol", "b",
                  "Imprime el AST en la salida estándar");
   mensajeAyuda = clp.getUsageMessage(true);
   clp.parseWhilePossible();
   clp.executeWhilePossible();

   if( nombreFichero==null ||
       nombreFichero.equals("") )
   {
      throw new
         CommandLineParserException("Se necesita un nombre de fichero");
   }

   try
   {
      fis = new FileInputStream(nombreFichero);
   }catch (FileNotFoundException fnfe){
        throw new CommandLineParserException(
          "El fichero '"+nombreFichero+"' no se pudo abrir");
   }
}

public class Tool() {} // Constructor de la clase (no hace nada)

public void trabaja()
{
   try{
      System.out.println("Reconociendo el fichero '"+nombreFichero+"'");

      // PASOS 4 y 5. Crear analizador sintáctico y pasarle
      // nombre fichero
      LeLiLexer lexer = new LeLiLexer(fis);
      lexer.setFilename(nombreFichero);
      lexer.setTokenObjectClass("antlraux.util.LexInfoToken");
```

```
        // PASOS 4 y 5. Crear analizador sintáctico y pasarle nombre Fichero
        LeLiParser parser = new LeLiParser(lexer);
        parser.setFilename(nombreFichero);

        // PASO 6. Comenzar el análisis
        parser.programa();

        // PASO 7. Obtener el AST
        AST ast = parser.getAST();

        // PASO 7b. Activar la impresión de información extra a toString()
        BaseAST.setVerboseStringConversion(
           true,
           LeLiParser._tokenNames);

        // PASO 8. Mostrar el AST por pantalla (modo texto)
        if(imprimeArbol==true)
        {
           System.out.println(ast.toStringList());
        }

        // PASO 9. Representación en ventana del AST
        final ASTFrame frame = new ASTFrame(Filename, ast);
        frame.setVisible(true);
     }
     catch (TokenStreamException tse)
     {
        tse.printStackTrace(System.err);
     }
     catch (RecognitionException re)
     {
        re.printStackTrace(System.err);
     }
  }
```

Sección 5.7: Otros aspectos de los ASTs

Aún hay más cosas que saber de los ASTs. En éste apartado se considerarán aspectos avanzados: fábricas y árboles heterogéneos. Los utilizaremos en el tema siguiente.

5.7.1: Fábricas de ASTs

Traducido del manual en inglés de ANTLR. Fichero trees.htm.

ANTLR utiliza el patrón de diseño fábrica para crear e interconectar los nodos AST. De esta forma se separa la construcción de los árboles del analizador sintáctico, proporcionando un "puente" entre el analizador y la construcción. Para implementar una fábrica de AST basta con hacer una subclase de `antlr.ASTFactory` y alterar el método `create`.

La clase que se utiliza para los AST puede cambiarse mediante el método `setASTNodeClass (String className)`. Como ya hemos dicho, la clase que se utiliza por defecto es `antlr.CommonAST`.

La clase `antlr.ASTFactory` tiene algunos métodos genéricos muy útiles:

```
public AST dup(AST t);
```

Sirve para copiar un AST. `clone()` no se utiliza porque queremos devolver un AST, no un Object. Además utilizando este método podemos controlar todos los ASTs que han sido creados por la fábrica.

```
public AST dupList(AST t);
```

Duplica además todos los hijos y hermanos del AST.

```
public AST dupTree(AST t);
```

Duplica un árbol completamente, asumiendo que tiene raíz (copia los hijos, pero no los hermanos de la raíz).

*****

Nótese que ya hemos utilizado algunos de estos métodos al eliminar el azúcar sintáctica de nuestra gramática.

5.7.2: ASTs heterogéneos

Traducido del manual en inglés de ANTLR. Fichero trees.htm.

Cada nodo en un árbol AST debe codificar información acerca del tipo de nodo que es; es decir, tiene que haber una manera de saber que se trata de un operador `OP_MAS` o de un `ENTERO`. Hay dos maneras de codificar dicha información: con un token o utilizando una clase (java, C++, C#) diferente para cada tipo. En otras palabras, se puede tener una sola clase de tokens con muchos tipos (muchos enteros diferentes) o bien muchas clases diferentes. Terence llamó a los ASTs "con una sola clase" ASTs *heterogéneos* y a los que tienen muchas clases *homogéneos*. La única razón para tener ASTs heterogéneos es para el caso en el que la información a almacenar varíe radicalmente de nodo a nodo.

ANTLR soporta los dos tipos de ASTs - ¡a la vez!. Si no se hace nada más que activar la opción `buildAST`, se optiene un árbol homogéneo. Más tarde, si se pretende utilizar clases físicamente serparadas para algunos de los nodos, simplemente hay que especificarlo en la regla que

construye el árbol. De esta manera se tiene lo mejor de ambos mundos: los árboles son construidos automáticamente, pero se pueden aplicar diferentes métodos y guardar información diferente sobre varios nodos. Nótese que la estructura del árbol permanece inalterada; solamente cambia el tipo de los nodos.

ANTLR aplica una especie de "algoritmo de contexto" para determinar el tipo de cada nodo particular que necesita crear. Por defecto el tipo es `antlr.CommonAST`, pero puede cambiarse con una invocación del método `setASTNodeClass` del analizador sintáctico:

```
myParser.setASTNodeClass("com.acme.MyAST");
```

En el fichero de definición de la gramática (el *.g) se puede cambiar el tipo cambiándolo para nodos creados a partir de un token particular. Para ello hay que utilizar la opción `<AST=nombretipo>` en la sección de tokens:

```
tokens {
   OP_MAS<AST=OP_MASNode>;
}
```

También es posible cambiar el tipo de un token dentro de una regla:

```
un_entero : ENTERO<AST=ENTERONode> ;
```

Esta capacidad es muy útil para tokens como `IDENT`, que podrían convertirse a `NOMBRETIPO` en unos contextos y a `VARIABLE` en otros.

ANTLR usa la fábrica de ASTs para crear nodos de los cuales no conoce el tipo específico (solamente sabe que deben cumplir la interfaz `antlr.collections.AST`). En otras palabras, ANTLR genera un código parecido al siguiente:

```
AST tmp2_AST = (AST)astFactory.create(LT(1));
```

Por otra parte, si se especifica una clase a utilizar, ya sea en la sección de tokens o dentro de una regla, ANTLR genera éste otro, más apropiado:

```
ENTERONode tmp3_AST = (ENTERONode)astFactory.create(LT(1), "ENTERONode");
```

Además de ser más rápido y evidente, este código aligera otro problema con los ASTs homogéneos: el tener que hacer conversiones de tipo (*castings*) entre `AST` y `ENTERONode` y a la inversa en cada regla (aunque ANTLR permite especificar el tipo de nodo por defecto con la opción `ASTLabelType`).

*****

Hay muchas razones para utilizar ASTs heterogéneos. Quizás la más importante tenga que ver con la organización de los nodos. Los nodos homogéneos proporcionados por defecto por ANTLR no se manejan muy facilmente en todos los casos. Por ejemplo, el AST homogéneo para el siguiente método de java:

```
public static void main(string [] args)
{
...
}
```

Tendría un nodo por cada modificador del método (`public`, `static`), otro seguidos del tipo de retorno(void) el nombre, etc.

La estructura de cada AST podrá cambiar dependiendo de la organización concreta que se utilice para representar los métodos, de manera que el nombre de un método podrá estar en primer

lugar en la lista de hijos de su AST, mientras que en otro esté en tercer o cuarto lugar; dependerá de si los modificadores preceden al nombre.

Para evitar estructuras diferentes en árboles del mismo tipo, lo mejor sería utilizar un AST heterogéneo que proporcionara la información de los modificadores (que permitiera hacer saber si el método es público o estático) mediante métodos en código nativo (java, C++, o C#), eliminándose así los nodos problemáticos del AST.

Otra razón es la mera necesidad de adjuntar información. Los nodos AST proporcionados por ANTLR no proporcionan información léxica sobre la información que contienen, de manera que a la hora de imprimir un mensaje de error semántico no se conoce el fichero, línea y columna donde se ha producido. Solucionaremos este problema utilizando la clase `LexInfoAST`. Pero no adelantemos acontecimientos.

Sección 5.8: Conclusión

En éste capítulo hemos visto cómo realizar un análisis sintáctico casi completo (salvo por la gestión de errores sintácticos). Como en el capítulo anterior, no hemos utilizado todas las capacidades de ANTLR; por ejemplo, no hemos aprovechado:

- Que ANTLR permite la herencia de analizadores sintácticos.
- Que se pueden utilizar varios analizadores léxicos con el mismo analizador sintáctico.
- Que se pueden declarar métodos y atributos java/C++/C# en el propio analizador.
- Que se pueden pasar parámetros a las reglas, y devolver parámetros con ellas.
- Los predicados semánticos (es mejor así; los predicados semánticos hacen la gramática menos portable).

Lo que sí hemos visto ha sido:

- Cómo implementar un analizador sintáctico en un fichero idenpendiente, y comunicarlo con un analizador léxico.
- Cómo crear un AST en un analizador sintáctico.
- Cómo eliminar el azúcar sintáctica.
- Cómo mostrar un AST en una ventana SWING.

En el siguiente capítulo no vamos a empezar con el análisis semántico de LeLi. En lugar de ello, vamos a extender un poco el análisis sintáctico para gestionar mejor los errores durante el reconocimiento, que en este capítulo hemos ignorado.

Capítulo 6: Recuperación de errores

"El hombre que ha cometido un error y no lo corrige comete otro error mayor."
Confucio

Capítulo 6: Recuperación de errores.....165

Sección 6.1: Introducción.....167
6.1.1: Situación..... 167
6.1.2: La controversia..... 167
6.1.3: Fases..... 168
6.1.4: Gestión de errores en bison y flex..... 168
6.1.5: Errores en analizadores recursivos descendentes..... 169
6.1.6: Errores como excepciones..... 169
6.1.7: Manejadores de excepciones..... 170
Sección 6.2: Estrategias de recuperación.....172
6.2.1: Introducción..... 172
6.2.2: Estrategia basada en SIGUIENTE..... 173
6.2.3: Conjuntos PRIMERO y SIGUIENTE..... 174
6.2.4: Estrategia basada en PRIMERO + SIGUIENTE..... 174
6.2.5: Aprovechando el modo pánico..... 175
6.2.6: Tokens de sincronismo 176
Ejemplo previo: el modo pánico fracasando estrepitosamente.....176
Presentación de los tokens de sincronismo.....178
6.2.7: Implementación en ANTLR..... 178
El conjunto de tokens de sincronismo.....179
El algoritmo de sincronización.....179
Problemas del algoritmo.....180
6.2.8: Trampas para excepciones..... 181
La amenaza fantasma.....181
¡Más manejadores!.....182
Presentación de la técnica de las trampas.....182
6.2.9: Retardo en el tratamiento de errores..... 184
El problema de la "nada" y NoViableAltException.....184
Capturando las excepciones.....186
Aún no hemos terminado.....187
Sección 6.3: Implementación - Herencia de gramáticas..... 189
6.3.1: El problema..... 189
6.3.2: Presentando la herencia de gramáticas en ANTLR..... 189
6.3.3: Herencia ANTLR != Herencia java..... 190
6.3.4: Línea de comandos..... 193
6.3.5: ¿Cómo se relaciona todo esto con la RE?..... 193
6.3.6: Importación de vocabulario..... 193
Sección 6.4: Control de mensajes: La clase Logger.....195
6.4.1: El problema..... 195
6.4.2: La clase Logger del paquete antlraux..... 196
Sección 6.5: Mejorando los mensajes de error.....199
6.5.1: Introducción..... 199
6.5.2: Cambiando el idioma de los mensajes de error..... 199
6.5.3: Alias de los tokens..... 201
Sección 6.6: Aplicación en el compilador de LeLi.....205
6.6.1: Acotación del problema: el fichero de errores..... 205
6.6.2: Aplicando los símbolos de sincronismo..... 206
Haciendo k=1.....206
Infraestructura para los tokens de sincronismo.....208

6.6.3: Colocando las trampas para excepciones..... 209
6.6.4: Retardando el tratamiento de errores..... 212
6.6.5: Errores frecuentes – acentuación..... 214
El problema..... 214
El método..... 215
Las estrategias..... 215
Modificaciones en las reglas..... 216
Vuelta a los tokens de sincronismo..... 217
6.6.6: El resultado..... 219
Sección 6.7: Código..... 221
6.7.1: Fichero LeLiLexer.g..... 221
6.7.2: Fichero LeLiParser.g..... 225
6.7.3: Fichero LeLiErrorRecoveryParser.g..... 232
Sección 6.8: Compilando y ejecutando el analizador..... 240
6.8.1: Compilación..... 240
6.8.2: Ejecución..... 240
Nuevos comandos..... 240
Nuevo código de la clase Tool..... 241
Sección 6.9: Conclusión..... 243

Sección 6.1: Introducción

6.1.1: Situación

La recuperación de errores no es una fase más del proceso de compilación. En lugar de eso, debería considerarse como una característica deseable del análisis léxico y semántico.

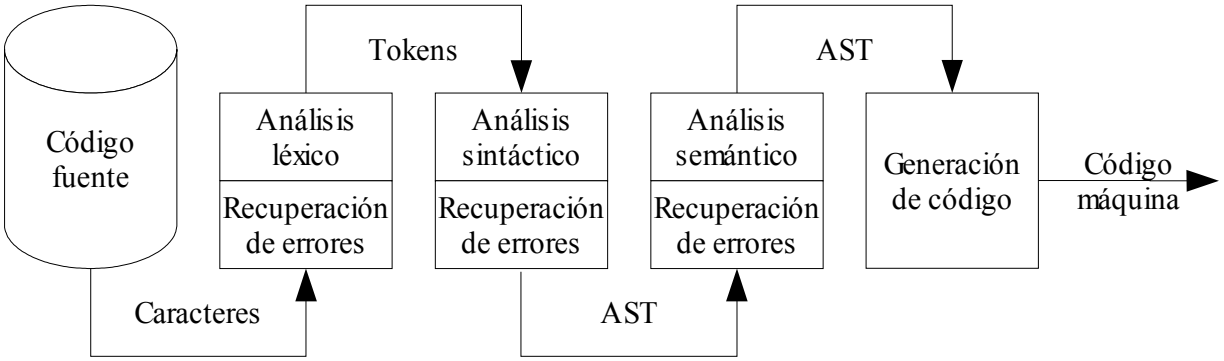

Ilustración 6.1 Recuperación de errores en el reconocimiento

La recuperación de errores permite al compilador detectar un error, dar parte de él*y seguir reconociendo la entrada*, detectando de esta manera más errores. Si el programador puede ver más errores en cada compilación, podrá corregir más errores en cada compilación, por lo que será más eficiente.

6.1.2: La controversia

Las opiniones con respecto a la recuperación de errores son muy variadas, y las discusiones que ocasiona están al mismo nivel de las discusiones sobre dónde colocar las llaves en C++ y java (¿debajo o al lado de la cabecera de los métodos?), o si los programas deberían tabularse con espacios o con tabulaciones[39].

La controversia gira en torno a la siguiente pregunta: ¿Es realmente necesaria la R.E.?

En internet he leído comentarios de gente que opina que la recuperación de errores es una pérdida de tiempo y dinero. La idea principal de éste argumento es que en los tiempos en los que los programas se codificaban en tarjetas perforadas tenía sentido proporcionar todos los errores de compilación posibles (porque cada compilación era lenta y cara – había que pagar las tarjetas!) mientras que en la actualidad las compilaciones son rápidas y virtualmente gratuitas (salvo por el coste en electricidad).

Es evidente que las personas que realizan este tipo de afirmaciones no han trabajado mucho en proyectos medianos o grandes. Por mi propia experiencia puedo decir que las compilaciones actuales de programas pequeños son actualmente casi instantáneas. Sin embargo, al aumentar el número de dependencias y librerías a recompilar, el tiempo de compilación aumenta rápidamente, sobre todo si además interviene un período de enlazado (o *linkage*, como dicen algunos). En una empresa en la que trabajé el programa principal (programado en C++) tardaba 30 minutos en compilar (en un servidor SUN especializado en compilación). Y eso sin contar con las librerías auxiliares. Éstas se solían compilar por la noche, en *batch*. Y no estaba garantizado que por la mañana todas estuvieran compiladas.

39 Parece que nunca llegaremos a un acuerdo con respecto a estas opciones. Por ejemplo, yo soy un programador “abajo y con tabulaciones”. Sin embargo Terence Parr es, a juzgar por el código de ANTLR, “derecha y con espacios”.

Así que las compilaciones actuales no son tan "rápidas"como algunos puedan pensar. Y tampoco tan baratas. Los 30 minutos que un técnico pierde esperando a que se compile un programa suponen 30 minutos de su sueldo perdido. Un sueldo que no es precisamente bajo, o al menos eso espero.

6.1.3: Fases

Las tres fases básicas de la gestión de errores son:

1. Detección. El error se localiza
2. Informe. Se archiva o muestra un mensaje informativo explicando la naturaleza del error
3. Recuperación. El analizador utiliza alguna técnica "superar" el error y poder seguir efectuando el análisis.

6.1.4: Gestión de errores en bison y flex

La gestión de errores en ANTLR es diferente de la gestión de errores en bison y flex, por la simple razón de que los reconocedores que generan son distingos. Mientras ANTLR genera reconocedores LL(k), bison y flex generan respectivamente autómatas finitos deterministas y reconocedores LALR(1).

Veamos cómo se gestionan los errores en bison y flex.

Para empezar, flex no tiene ningún mecanismo de recuperación automática de errores. Cuando durante el análisis léxico se encuentra algún carácter no esperado, simplemente se imprime un mensaje de error por pantalla y se termina la aplicación.

Bison permite un tratamiento mucho más personalizado de los errores. Dado que los analizadores que genera son LR, cada construcción sintáctica del lenguaje de programación puede ser reconocida por varias reglas a la vez; solamente una o dos sirven para reconocerla, mientras que las demás se utilizan para gestionar errores:

```
tipo : RES_ENTERO
     | IDENT
     ;

tipo_pto_coma
   : tipo PUNTO_COMA!
     {$$ = $1;}
   | tipo
     { $$ = $1; error(yylocation(),"Falta el ';' al final del tipo"); }
   | error PUNTO_COMA!
     {$$ = Lit_Error();yyerrok;}
   ;
```

La regla anterior debe reconocer un tipo seguido de un punto y coma. Solamente la primera opción reconoce verdaderamente la regla; las otras dos gestionan errores (un punto y coma que falta o un tipo erróneo, utilizando la palabra reservada de bison `error`)

Esta aproximación permite una granularidad muy fina en el tratamiento de errores; prácticamente cualquier error puede tratarse individualmente. El problema es que las gramáticas crecen rápidamente de tamaño y se hacen más ilegibles. Además, las acciones pueden potencialmente añadir conflictos *shift-reduce* al analizador.

6.1.5: Errores en analizadores recursivos descendentes

Un analizador LL es completamente diferente a uno LR. Ambos son "autómatas", en el sentido de que los dos tienen "estados" y "transiciones". La diferencia estriba en lo que hay en los estados.

En un estado LR hay "varias reglas que se están reconociendo desde atrás". Cuando una de las reglas se reconoce completamente se "reduce", y así sucesivamente hasta que se termina el análisis. Construcciones de reglas especiales (`error`, en el caso de bison) permiten "reconocer los errores en las reglas". Estas reglas son idénticas a las normales salvo por que provocan un informe de error y por las acciones semánticas que las sucedan.

En un autómata LL, sin embarco, cada estado se correspondería con una sola regla que se está reconociendo (un método del analizador recursivo). De esta forma, el analizador está "esperando" un determinado conjunto de símbolos en cada momento del análisis. Si el token que se encuentra no concuerda con dicho conjunto, hay un error. Así, la estrategia de detección de errores en LL no pasa por "reconocer los errores con reglas especiales". En lugar de ello, un error es "cualquier entrada que no esté en el conjunto esperado".

De esta manera, para escribir la regla anterior en un analizador de ANTLR tendríamos que "limpiar" todas las reglas de gestión de errores:

```
tipo_pto_coma: tipo PTO_COMA ;
```

Otro de los puntos interesantes de ANTLR es el siguiente ¿qué se hace cuando se detecta un error?

6.1.6: Errores como excepciones

La respuesta este interrogante es muy sencilla, teniendo en cuenta que utilizamos un analizador recursivo descendente: Excepciones.

En general, cualquier error de reconocimiento léxico o sintáctico en ANTLR implica el lanzamiento de una excepción.

En particular, los métodos para reconocer tokens en el analizador sintáctico (`match` y `al`) lanzan una excepción `antlr.MismatchedTokenException`. Por otra parte, cuando ninguno de los tokens del lookahead concuerda con los esperados en las alternativas de una expresión con varias alternativas, se lanza `antlr.NoViableAltException`. Los métodos análogos para reconocer caracteres en el analizador léxico (`match` y `al`) lanzan excepciones análogas.

ANTLR genera automáticamente código para gestionar los errores, aunque también puede especificarse uno propio. Ambas opciones desembocarán en la generación de un bloque `try/catch` envolviendo la parte de código generado que efectúa el reconocimiento. Lo que cambiaría sería el cuerpo del `catch`. Si no se especifica ningún gestor de errores (ni por defecto ni personalizado), la excepción se propagará hacia afuera, subiendo de método en método hasta salir del analizador. Recordemos que los reconocedores generados por ANTLR son reconocedores recursivos descendentes.

Para ilustrar cómo se gestionan los errores en ANTLR, veamos la función que se generaría para reconocer la regla anterior[40]:

40 El código está simplificado para que solamente muestre el código de gestión de errores de ANTLR.

```
protected void tipo_pto_coma()
{
   try{
        tipo();
        match(PTO_COMA);
   }catch (RecognitionException ex) {
      reportError(ex);
      consume();
      consumeUntil(_tokenSet_XX);
   }
}
```

Pueden observarse las 3 fases de la gestión de errores:

- La detección del error se realiza con el bloque `try/catch`
- El informe del error se realiza con la llamada a `reportError`, en el cuerpo del `catch`
- La recuperación es simple. Primero, se consume el token que ha provocado el error. Después, se van consumiendo tokens hasta que alguno de ellos concuerde con alguno de los existentes en el conjunto `_tokenSet_XX` (donde XX representa varios dígitos cualesquiera) He podido deducir que dicho conjunto es el conjunto `SIGUIENTE(tipo_pto_coma)`. Veremos un poco más adelante qué es este conjunto.

6.1.7: Manejadores de excepciones

Para modificar la manera en la que las excepciones son gestionadas en ANTLR hay que especificar *manejadores de excepciones*. Un manejador de excepciones es una acción que se ejecuta en la parte `catch` de la cláusula `try/catch` vista más arriba.

Una misma regla puede tener más de un manejador de excepciones simultáneamente, cada uno de ellos correspondiéndose a un bloque `catch` distinto. Un aspecto a resaltar es que *no tienen que situarse obligatoriamente al final de una regla.* También pueden cubrir una sola alternativa entre varias e incluso un solo elemento con una etiqueta.

```
exception [etiqueta]
catch [tipoExcepción variableExcepción]
  { acción }
catch ...
catch ...
```

etiqueta es utilizado solamente para los elementos que tienen etiqueta. `tipoExcepción` es el tipo de la excepción que vamos a capturar (a menudo `RecognitionException`), y `variableExcepción` es la variable en la que se almacena dicha excepción.

Vamos a ver un ejemplo de cada tipo de manejador de excepciones:

- Manejador para toda una regla:

```
exprSuma : exprProducto OP_SUMA^ exprProducto
         ;       // Nótese que los manejadores generales se colocan tras el ";"
         exception catch [RecognitionException ex]
         { ... <tratamiento> ... }
```

- Manejador para una alternativa:

```
instruccion : instFor
            | instExpresion
              exception catch [RecognitionException ex]
              { ... <tratamiento> ... }
            | instSi
            ;
```

- Manejador para un elemento con etiqueta:

```
instMientras : RES_MIENTRAS^ PARENT_AB! e:expresion PARENT_CE!
CORCHETE_AB! listaInst CORCHETE_CE!
             ;
             exception [e] catch [RecognitionException ex]
             { ... <tratamiento> ... }
```

Sección 6.2: Estrategias de recuperación

6.2.1: Introducción

En principio la fase de detección está estupendamente resuelta con el mecanismo de las excepciones[41]. En general bastará con añadir manejadores de excepciones para hacer una buena gestión. Pero en ciertas ocasiones resultará más cómodo utilizar una regla-error y lanzar una excepción personalizada.

Consideremos, por ejemplo, la instrucción `si` de LeLi. Dada la abundancia de lenguajes que utilizan la palabra inglesa `if` para representar los condicionales, suele utilizarse ésta por error en lugar de `si`. Es un error tan corriente que se podría pensar en incluir un mensaje exclusivo par él. Podríamos hacerlo con algo parecido a ésto:

```
instSi : si:RES_SI^ PARENT_AB! expresion PARENT_CE!
         LLAVE_AB! listaInst LLAVE_CE!
       ;
  exception [si] catch [RecognitionException ex]
  {
     Token t= LT(1); // Obtiene el token que ha provocado el error
     if(t.getText()==”if”) // Si es un “if”
     {
        // Utilizar la información léxica de la excepción,
        // pero cambiar el mensaje por otro.
        reportError(
           ex, “Se debe utilizar la palabra reservada 'si', no 'if'”);
        consume(); // consume el “if” y continúa la regla
     }
     else throw ex; // otro tipo de error. Relanza la excepción
  }
```

Sin embargo, el código es un poco difícil de leer. Es más sencillo es utilizar una regla-error:

```
{ // método auxiliar
  public void reportError(String msg, String fileName, int line, int column)
  { antlrTool.error(msg, fileName, line, column); }
}
...
si : RES_SI
   | siErroneo:IDENT {siErroneo.getText()== “if”}?
     {
        // Modificamos el AST para que contenga el nodo adecuado
        siErroneo.setType(RES_SI);
        siErroneo.set  Text(“si”);
        reportError(
           “Se debe utilizar la palabra reservada 'si', no 'if'”,
           getFileName(), getLine(), getColumn()-2);
     }
   ;

instSi : s:si! PARENT_AB! expresion PARENT_CE!
         LLAVE_AB! listaInst LLAVE_CE!
         // Hay que hacer de “si” la raíz. ¡No se puede utilizar
         // el operador ^ sobre referencias a otras reglas!
         { ## = #(#s, ##); }
```

41 Más adelante veremos que a veces detecta los errores “demasiado pronto”.

```
        ;
```

El método `antlrTool.error` es el que se llama "por debajo" en `Parser.reportError`. He tenido que utilizarlo porque por defecto no se suministra un `Parser.reportError` que permita incluir el nombre de fichero, línea y columna.

Centrémonos ahora en la recuperación de errores.

La fase de detección consiste en encontrar una discordancia entre la entrada del analizador y sus reglas. La fase de informe se limita a mostrar un mensaje informando sobre dicha discordancia. Pues bien, la recuperación de errores consiste en "resincronizar la entrada con respecto a las reglas", de manera que se pueda seguir con el análisis.

Se distinguen dos estrategias fundamentales para recuperarse de un error:

- Añadir, si es posible, los tokens que falten en la entrada (como se ha hecho en la sustitución del token "if" por un "si")
- Descartar tokens de la entrada hasta que se encuentre un token válido que permita "sincronizar". Ésta es la estrategia que utiliza ANTLR por defecto.

Hemos de mencionar que la estrategia de ANTLR es buena, y resulta difícil mejorarla. En esta sección vamos a presentar diferentes estrategias de recuperación, pero no las utilizaremos todas en nuestro compilador. Las que no utilizaremos están ahí por si alguien las necesita.

6.2.2: Estrategia basada en SIGUIENTE

Dada una expresión de reconocimiento (que puede ser una regla, una alternativa o una sub regla EBNF), se define el conjunto `SIGUIENTE(expresión)` como el conjunto de tokens que pueden seguir a dicha expresión en la gramática en la que se encuentra. Por ejemplo, en la gramática:

```
base : IDENT
     | LIT_ENTERO
     | LIT_REAL
     | LIT_CADENA
     ;
expresion : exprSuma ;
exprSuma : e1:exprProducto (OP_SUMA exprProducto)* ;
exprProducto : e2:base (OP_PRODUCTO base)* ;
```

Podemos calcular varios conjuntos `SIGUIENTE` en diferentes lugares (que he marcado con etiquetas):

- `SIGUIENTE(exprSuma-la regla)`: Es `OP_SUMA` o fin de fichero.
- `SIGUIENTE(e1)`: Es `OP_SUMA` o fin de fichero.
- `SIGUIENTE(e2)`: Es `OP_PRODUCTO` o fin de fichero.

La estrategia basada en `SIGUIENTE` es la que utiliza ANTLR. Consiste en, una vez detectado el error, consumir tokens hasta que se encuentre alguno coincidente con el conjunto `SIGUIENTE` de la regla en la que se ha detectado. Los pasos a seguir son exactamente tres:

- Mostrar el error al usuario (con una llamada al método `reportError`)
- Consumir el token que ha provocado el error (llamando a `consume`)
- Consumir tokens de la entrada hasta que alguno sea válido – pertenezca al conjunto `SIGUIENTE(regla)`, donde `regla` es la regla en la que se ha producido el error. Todo ello se realiza con una simple llamada al método `consumeUntil`, pasándole como parámetro el

conjunto SIGUIENTE(regla).

A la estrategia basada en SIGUIENTE también se la conoce como "modo pánico" (*panic-mode*).

6.2.3: Conjuntos PRIMERO y SIGUIENTE

Durante el tratamiento de errores en las acciones es posible que necesitemos utilizar los conjuntos PRIMERO y SIGUIENTE de una regla. Para poder utilizarlos en la recuperación de errores, ANTLR proporciona dos abreviaturas:

- $FIRST(etiqueta) es BitSet que representa el conjunto PRIMERO(expresion), siendo expresion el token o referencia etiquetados con etiqueta. Sin etiqueta se refiere a la regla que lo contiene.
- $FOLLOW(regla) es el BitSet que representa el conjunto SIGUIENTE(regla). Sin etiqueta se refiere a la regla que lo contiene.

La clase BitSet se encuentra en antlr.collections.impl. Proporciona métodos para realizar operaciones comunes con conjuntos de bits, desde añadir un valor a hacer una operación booleana (or, and) con otro BitSet.

La estrategia basada en SIGUIENTE de ANTLR puede por tanto escribirse a mano, con un manejador de excepciones para toda la regla. Por ejemplo, el siguiente código:

```
tipo_pto_coma
   : tipo PTO_COMA
   ;
```

es equivalente a este otro:

```
tipo_pto_coma
: tipo PTO_COMA
;
exception catch [RecognitionException ex]
{
   // 1. Comunicar el error
   reportError(ex);
   // 2. Consumir el token problemático
   consume();
   // 3. Consumir tokens
   consumeUntil($FOLLOW(tipo_pto_coma));
   // consumeUntil($FOLLOW); también sería correcto
}
```

6.2.4: Estrategia basada en PRIMERO + SIGUIENTE

Es una mejora sobre la estrategia basada en SIGUIENTE. Al igual que en ella, se basa en ir elminando tokens que no sean adecuados, salvo que por el hecho de que además de probar con el conjunto SIGUIENTE de la regla, se utiliza el conjunto PRIMERO. Si el primer token que pertenece a uno de los grupos pertenece a SIGUIENTE se utilizar la estrategia habitual (fin de la regla). Por otro lado, si pertenece al conjunto PRIMERO, *se reinicia el reconocimiento de la regla*, es decir, la regla en la que se ha detectado el error vuelve a reconocerse desde el principio – invocando de nuevo el método que reconoce la regla.

Hay que tener en cuenta que el conjunto SIGUIENTE tiene prioridad absoluta sobre el conjunto PRIMERO, es decir, que cuando un token esté en los dos, si se encuentra, simplemente se saldrá de la regla.

La estrategia PRIMERO+SIGUIENTE no se utiliza de forma habitual, ya que en la mayoría de los casos los elementos de PRIMERO están contenidos en SIGUIENTE, así que finalmente no se gana mucho.

Un ejemplo de implementación:

```
asignacion: IDENT OP_ASIG^ expresion PUNTO_COMA! ;
exception catch [RecognitionException ex]
{
   reportError(ex);
   if($FIRST.member(LA(0)))
      asignacion();
   else
   {
      consume();
      Bitset auxiliar = $FOLLOW.or($FIRST);
      consumeUntil(auxiliar);
      if( ! $FOLLOW.member(LA(0)) ) asignacion();
   }
}
```

Los pasos que se siguen en la recuperación son los siguientes:

- Se imprime el mensaje de error
- Si el token que ha provocado el error pertenece a PRIMERO(regla), se reinicia la regla.
- Si no, se consume y se continúan consumiendo tokens hasta que se encuentre uno que pertenezca a PRIMERO(regla) o SIGUIENTE(regla). Si pertenece a SIGUIENTE, se considera que la entrada está sincronizada con la regla que siga a la actual, y se sale de la regla. En otro caso, pertenecerá a PRIMERO, luego se reinicia el reconocimiento de la regla.

La estrategia PRIMERO+SIGUIENTE es adecuada para reglas simples que se repitan muchísimo. En el ejemplo anterior sería adecuada si la regla asignacion sirviera para reconocer un fichero de configuración de un programa que solamente tuviera asignaciones.

6.2.5: Aprovechando el modo pánico

Hasta ahora hemos considerado el manejo de excepciones "a nivel de regla", es decir, cuando un error es encontrado mientras se está reconociendo una regla, ésta se da por perdida, eliminando símbolos hasta que se encuentra alguno que permita suponer que la regla se ha terminado (símbolos de sincronismo y conjunto SIGUIENTE). Hemos llamado a esta estrategia el "modo pánico".

En general el modo pánico es muy útil: las reglas "pequeñas", que reconocen unos pocos tokens como máximo de la entrada, se recuperan rápidamente. Por lo general, las reglas mas "grandes", que reconocen más tokens, están divididas en reglas más simples, en las que se efectúa la recuperación. Por lo tanto no suele haber problemas.

¡Pero hay que tener cuidado! Existen algunos casos en los que es recomendable otro tipo de recuperación. Por ejemplo, en clausuras realizadas sobre enumeraciones de tokens:

```
inicializaciones : (IDENT OP_IGUAL LIT_ENTERO PUNTO_COMA)+ ;
```

La regla anterior está pensada para poder reconocer un conjunto de 1 o más "inicializaciones". Una inicialización es un identificador seguido del operador igual y un entero, y acabando un punto y coma. El código java que se generará seguirá el siguiente esquema:

```
void inicializaciones()
{
   try
   {
      do{
         match(IDENT);
         match(OP_IGUAL);
         match(LIT_ENTERO);
         match(PUNTO_COMA);
      } while(true);
   } catch (RecognitionException Ex) {
      // Modo pánico
      ...
   }
}
```

La regla puede servir, por ejemplo, para leer un fichero de configuración. Imaginemos que se trata de un fichero extenso, de 10000 inicializaciones, y que hay un error en la segunda. Se lanzará un error y se empezarán a consumir tokens hasta que se encuentre uno perteneciente al conjunto `SIGUIENTE(inicializaciones)`. Las 9998 inicializaciones pendientes se consumirán en el modo pánico, y no se buscarán más errores.

Este tipo de situaciones es muy fácil de arreglar: basta con utilizar una regla adicional. Así:

```
inicializaciones : (inicializacion)+ ;
inicializacion : (IDENT OP_IGUAL LIT_ENTERO PUNTO_COMA)+ ;
```

Ahora se utilizarán dos bloques `try-catch`, uno en cada método de las reglas. Si se le pasa la misma entrada que en el caso anterior, el error de la segunda inicialización se recuperará dentro de la regla `inicializacion`, muchísimo antes.

En el caso anterior es posible e incluso más legible utilizar varias reglas para definir la gramática. Este método es el más recomendable para acelerar la recuperación del modo pánico.

Existe una alternativa, bastante más aparatosa, consistente en utilizar los manejadores de excepción internos de las reglas. Solamente es recomendable si la recuperación por defecto de ANTLR no es satisfactoria. Es decir, que es muy poco recomendable.

Aprovechar el modo pánico consiste, pues, en dividir las reglas “grandes” en reglas más “pequeñas” para que los errores se recuperen cuanto antes.

6.2.6: Tokens de sincronismo

La estrategia que vamos a utilizar en nuestro compilador es la de los tokens de sincronismo. Un poco más adelante definiremos propiamente qué son los tokens de sincronismo, pero antes vamos a comenzar con un ejemplo previo, que ilustrará cómo algunas situaciones pueden dejar “fuera de combate” al modo pánico.

Ejemplo previo: el modo pánico fracasando estrepitosamente

Imaginemos que estamos reconociendo la siguiente entrada:

```
clase A
{
   método m1(Entero pA)
   {
      pA ++ // ¡Error! ¡Falta un punto y coma!
   }
}
clase B
{
   atributo Entero aC;
   método m2()
   {
      aC++;
   }
}
```

En el código hay un error: falta un punto y coma. Siguiendo la traza del analizador[42], podemos ver que el error será detectado en la regla `expPostIncremento`. En `expPostIncremento` se estará esperando un elemento de su conjunto `SIGUIENTE`, pero en lugar de ello habrá obtenido una llave cerrada (`LLAVE_CE`, '}').

Supongamos que el conjunto `SIGUIENTE(expPostIncremento)` contiene el punto y coma (';'), el punto ('.') y el paréntesis cerrado (')'). Es decir,

`SIGUIENTE(expPostIncremento)={PUNTO_COMA, PUNTO, PARENT_CE}`[43].

Entonces, dado que `LLAVE_CE` no pertenece a `SIGUIENTE(expPostIncremento)`, el reconocedor simplemente ignorará el token, y empezará a consumir tokens hasta que encuentre uno que sí pertenezca a dicho conjunto. Y así comenzará una curiosa reacción en cadena.

La llave que se ha "tragado" el modo pánico era la llave de cierre del último método de una clase. Esta llave viene seguida de otra llave de cierre de la clase, que también será "tragada", así como la cabecera de la definición de la clase que venga detrás clase (`RES_CLASE IDENT` etc...), porque ninguno de esos tokens pertenece al conjunto `SIGUIENTE(expPostIncremento)`.

El modo pánico seguirá así consumiendo token tras token, hasta llegar al punto y coma que finaliza la declaración del atributo `aC` de la clase `B`. El analizador, aliviado, saldrá del modo pánico de `expPostIncremento`, considerando luego terminada la instrucción.

Pero los problemas no han acabado aún: el analizador todavía cree que se encuentra dentro de una lista de instrucciones, y buscará un token que pertenezca a `PRIMERO(instruccion)`. En lugar de ello encontrará el token "método" (`RES_METODO`), que no pertenece a dicho conjunto.Volverá a ponerse en marcha el modo pánico en `listaInstrucciones` y consumirá el token `RES_METODO`. Los dos tokens que siguen a `RES_METODO` son un identificador, `IDENT`, y un paréntesis abierto, `PARENT_AB`. Como el analizador está esperando una instrucción o la llave cerrada para acabar con el método, intentará reconocer una llamada a un método (y no una declaración de método). No lo conseguirá, porque `Entero` no pertenece al conjunto `PRIMERO (listaParámetros)`...

...el analizador continuará "equivocándose" dos veces más, antes de por fin poder sincronizar con la única instrucción del método `B:m2`, que se añadirá a la lista de instrucciones de `A:m1`. Si el análisis semántico está activado, se producirá un error más, porque el identificador "aC" no estará declarado.

42 Para seguir la traza será necesario compilar el analizador con la opción "-trace".

43 Este conjunto está, evidentemente, incompleto, pero bastará para presentar el problema. Lo seguro es que LLAVE_AB no pertenece a dicho conjunto.

Presentación de los tokens de sincronismo

El caso que acabamos de presentar es una consecuencia extrema de no usar símbolos de sincronismo.

La forma adecuada de recuperar el error es intuitiva: cuando se encontró la llave cerrada al principio del modo pánico, ésta indicaba el final de un grupo de instrucciones, lo que implica el final de la instrucción actual, que a su vez implica el final de la expresión actual. Lo que tendría que hacer el compilador sería, por lo tanto, cancelar la regla actual, volviendo a `listaInstrucciones`, donde reconocería la llave cerrada como fin de regla.

Vamos a expresar esto de una manera más formal. Podemos representar el momento de comienzo del error con una pila de llamadas. Esta pila simboliza los diferentes métodos que se han llamado recursivamente en el analizador recursivo descendente desde que empezó el análisis.

La regla `expPostIncremento` deberá "dejar pasar" el token `LLAVE_AB` a su "nivel superior", en nuestro caso `expCambioSigno`. Ésta a su vez la deberá "dejar pasar hacia arriba", y así sucesivamente hasta que una regla (`listaInstrucciones`) reconozca `LLAVE_AB` en la entrada.

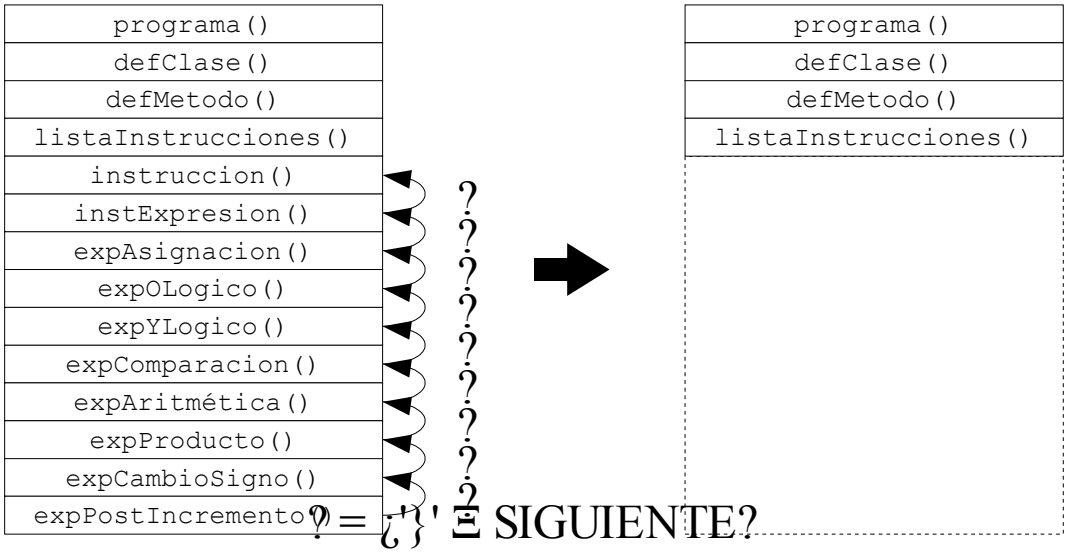

Ilustración 6.2 Pila de llamadas antes y después de recuperar un error

En la izquierda vemos el estado de la pila de llamadas en el momento en el que se encuentra la llave cerrada. En ese momento, el comportamiento deseado del analizador es el de "desapilar" llamadas hasta que se pueda reconocer la llave.

En general, una llave cerrada siempre podrá *siempre* utilizarse para "desapilar" métodos al encontrarse un error en la entrada. La llave cerrada es un *token de sincronismo.*

6.2.7: Implementación en ANTLR

Aunque pueden utilizarse diferentes grupos de símbolos de sincronismo en un mismo analizador, lo más usual es que solamente se utilice el mismo grupo para todas las reglas. Ésta será la estrategia que utilizaremos. Para implementarla debemos:

- Encontrar una manera de especificar el conjunto global de símbolos de sincronismo
- Encontrar el algoritmo de sincronización más adecuado.

El conjunto de tokens de sincronismo

Para representar el conjunto de símbolos de sincronismo de nuestro lenguaje recomiendo utilizar la clase `antlr.collections.impl.Bitset`. Esta clase ya está incluida automáticamente en el código java de nuestro analizador, luego lo único que hay que hacer es declarar un atributo estático final de la clase e iniciarlo con un método estático fina.

Supongamos que estamos definiendo un analizador sintáctico llamado `MiParser`. Supongamos que el conjunto de símbolos de sincronismo que hemos elegido sea `LLAVE_AB`, `LLAVE_CE`, `PUNTO_COMA` y `COMA`. Entonces, para crear el conjunto de símbolos de sincronismo habrá que escribir lo siguiente:

```
class MiParser extends Parser
options {...}
tokens {...}
{
   public static final BitSet SimbolosSincro = mk_SimbolosSincro();
   private static final BitSet mk_SimbolosSincro()
   {
      BitSet b = new BitSet();
      b.add( LLAVE_AB );
      b.add( LLAVE_CE );
      b.add( PUNTO_COMA );
      b.add( COMA);
      return b;
   }
}
... // reglas
```

Este conjunto es, sin duda, demasiado pequeño para LeLi. Presentaremos uno más adecuado en la siguiente sección. Entretanto, he aquí algunos consejos para elegirlo:

- Los símbolos de sincronismo suelen ser símbolos de puntuación, tanto "cerrados" (`LLAVE_CE`, `PARENT_CE`) como "abiertos" (`LLAVE_AB`, `PARENT_AB`). Los primeros porque significan el final de la estructura lingüística actual, y los segundos el principio de una nueva (y, por tanto, el final de la actual).
- De forma similar, pueden existir tokens que representen el principio de una nueva estructura (en LeLi son `RES_CLASE`, `RES_METODO`, etc...) que no se parezcan textualmente a símbolos de puntuación. Dichos tokens también deberían ser incluídos.
- Es mejor pecar por exceso que por defecto: es más conveniente tener demasiados tokens declarados como símbolos de sincronismo que demasiado pocos; los errores más frecuentes serán detectados de todas maneras.

El algoritmo de sincronización

El algoritmo de sincronización deberá lanzarse al capturar un token no esperado en la entrada, sustituyendo al manejador de excepciones por defecto. Consistirá en:

- No hacer nada si el token es un símbolo de sincronismo
- Si no es de sincronismo, se procederá como en el modo pánico normal, pero utilizando como conjunto de tokens válidos *la unión del conjunto SimbolosSincro con SIGUIENTE*:

Vamos a encapsular dicho algoritmo en el método sincronizar. Dicho método necesitará como parámetros la excepción que se lanzó (para hacer un `reportError` si es preciso) y el conjunto `SIGUIENTE` de la regla:

```
expPostIncremento :  expNegacion (OP_MASMAS^|OP_MENOSMENOS^)?
                  ;
                  exception catch[ RecognitionException e]
                  { sincronizar(e, $FOLLOW); }
```

Llamaremos *sincronizador* al manejador de excepciones que invoca el método `sincronizar`.

El método `sincronizar`, por su parte, tendrá el siguiente aspecto:

```
class MiParser extends Parser;
options {...}
tokens {...}
{
   ... // declaración de SimbolosSincro y mk_SimbolosSincro()

   public void sincronizar ( RecognitionException re, BitSet SIGUIENTE )
   throws RecognitionException, TokenStreamException
   {
      // Si es SS, "dejar pasar" (terminar la regla)
      if( ! simbolosSincro.member(LA(0)) )
      {
         // Si estamos guessing, lanzamos directamente
         if (inputState.guessing!=0) throw re;

         // Crear un nuevo bitset que contenga a los símbolos de
         // sincronismo y a SIGUIENTE utilizando la "or" lógica
         BitSet auxiliar = simbolosSincro.or(SIGUIENTE);

         // Mostrar el error y consumir, pero utilizando "auxiliar"
         reportError(re);
         consume();
         consumeUntil(auxiliar);
      }
   }
}
```

Problemas del algoritmo

Vamos a aclarar algo importante: el método que acabamos de describir no es la panacea. Entre otras razones, porque no puede sustituir al manejador de excepciones por defecto de ANTLR en todos los casos. Veamos por qué.

Empecemos revisando el ejemplo anterior. Teníamos el siguiente código:

```
clase A
{
   método m1(Entero pA)
   {
      pA ++ // ¡Error! ¡Falta un punto y coma!
   }
}
```

Ya hemos visto el comportamiento impecable del algoritmo de sincronización en este caso: el símbolo de sincronismo es "pasado hacia arriba" en la pila de llamadas. Estamos bastante abajo en la pila, y es muy probable que otra regla pueda "absorberlo" y se continúe el reconocimiento.

Al método solamente se le pueden echar en cara dos problemas:

- Se ha perdido el contenido completo de la instrucción (el lanzamiento de una excepción en la

regla provocó la cancelación de la construcción del AST)

- ¡No se ha emitido ningún mensaje de error!

Solucionaremos estos errores más adelante.

Existe, sin embargo, otro problema que debemos tratar. Podemos encontrarlo este otro código:

```
clase A
{
   método m1(Entero pA // Falta un paréntesis cerrado
   {
      pA ++ ;
   }

   método m2(...) {...}
   ...
}
```

En este caso la ausencia del paréntesis de terminación provocará que se entre en modo de sincronización[44] en la declaración de m1 (regla `decMetodo`). El algoritmo encontrará `LLAVE_AB`, que al ser símbolo de sincronismo será "pasado hacia arriba". La siguiente regla en la pila es `listaMiembros`, que no espera una llave abierta, así que por ser un token de sincronismo también lo pasará para arriba... ya se intuye lo que pasará. Las dos reglas que quedan, `decClase` y `programa`, también "pasarán hacia arriba" la llave.

Al acabarse el método programa se habrá acabado el análisis sintáctico, quedando el AST tal y como estuviera en el momento de producirse el primer error. Ni siquiera se analizarán el resto de métodos de la clase.

En resumen, debemos resolver tres problemas:

- Nuestro analizador aún es demasiado "perezoso" en la recuperación, pues no genera los mejores AST posibles.
- No se emiten mensajes de error cuando falta un token.
- En ocasiones hay "salidas prematuras" del análisis.

Como veremos, todos estos problemas tienen relación con el protocolo de capturas de excepciones de ANTLR. Vamos a mostrar dos técnicas que, aplicadas conjuntamente, nos permitirán mitigar los tres problemas de nuestro algoritmo de recuperación. Estas técnicas se llaman "trampas de excepciones" y "retrasaso de la detección de errores".

6.2.8: Trampas para excepciones

La técnica de las trampas para excepciones es una técnica de recuperación que puede utilizarse de forma independiente, aunque nosotros la utilizaremos en combinación con los tokens de sincronismo, para solucionar algunos de los problemas que hemos visto que posee.

La amenaza fantasma

Supongamos un analizador sintáctico con los símbolos de sincronismo habituales (paréntesis abierto y cerrado, punto y coma, punto, coma, llave abierta y llave cerrada).

Considérese la siguiente regla para invocar funciones, a la que hemos añadido un sincronizador:

44 Estamos suponiendo que se añade el manejador de excepciones con el método sincronizar a todas las reglas de la gramática.

```
llamada : IDENT PARENT_AB! listaExpresiones PARENT_CE! PUNTO_COMA!;
        exception catch [RecognitionException e]
        { sincronizar (e, $FOLLOW); }
```

Supongamos que el analizador que contiene esa regla se encuentra con una entrada así:

```
...
f ( x, y, z ;
...
```

El mecanismo de sincronización entrará en marcha, cuando detecte el punto y coma (estaría esperando una coma o un paréntesis cerrado). Se cancelará la construcción del AST (perdiéndose toda la información de la llamada) y se "pasará hacia arriba" el punto y coma, por ser un símbolo de sincronismo. Es decir, la regla adolece de "analizador perezoso" y "omisión de mensaje de error". Además, aunque improbable, si ninguna regla del analizador captura el punto y coma, habrá un problema de "salida prematura".

¡Todos estos problemas han sido provocados por un solo token que no existía! Tenemos que procurar que estos "tokens fantasma" no arruinen de esta manera nuestro reconocedor.

¡Más manejadores!

Hasta ahora solamente hemos colocado el algoritmo de recuperación en el manejador global de una regla. Recordemos que los manejadores de excepciones pueden, además, colocarse en dos lugares más:

- En una opción
- En un elemento con nombre[45]

La ventaja principal que tienen estos tipos de manejadores con respecto al global es que *no cancelan el reconocimiento de la regla ni la construcción del AST*. Solamente cancelan las partes afectadas. En otras palabras, si escribimos una llamada a un método así:

```
llamada
   : IDENT p1:PARENT_AB! listaExpresiones p2:PARENT_CE! pc:PUNTO_COMA!
   ;
   exception [p1] catch [RecognitionException e] // para p1
   { reportError(e);
   exception [p2] catch [RecognitionException e] // para p2
   { reportError(e);
   exception [pc] catch [RecognitionException e] // para pc
   { reportError(e); }
   exception catch [RecognitionException]  // sincronizador
   { sincronizar (e, $FOLLOW); }
```

Y se omite cualquiera de los dos paréntesis (¡o los dos!), se emitirá un mensaje de error, pero no se cancelará la construcción del AST. De la misma forma, si se omite el punto y coma, se emitirá el mensaje de error correspondiente, sin lanzarse errores.

El problema de esta solución es que es absolutamente engorrosa.

Presentación de la técnica de las trampas

La técnica que denominaremos como "trampa para excepciones" es una generalización del ejemplo anterior, y de la técnica de aprovechamiento del modo pánico que describimos anteriormente. Parte de la idea de que ciertos tokens son "obligatorios" en ciertos lugares de la

45 Consultar el principio del capítulo para más información sobre los manejadores de excepciones.

entrada (por ejemplo, detrás de la palabra reservada si siempre tiene que haber un paréntesis abierto). La omisión de dichos tokens es suficientemente importante como para emitir un mensaje de error, pero no lo suficiente como para cancelar el reconocimiento de la regla , porque son fácilmente recuperables. Llamaremos a estos tokens *tokens-trampa*.

En el subapartado anterior mostrábamos una manera de tratar los tokens trampa, utilizando un manejador de excepciones para cada aparición de cada token trampa en cada regla. Sin embargo ya hemos visto que este método es muy engorroso, y además oscurece el código (a veces hay más texto en los manejadores de excepciones que en la propia regla). Se impone pues encontrar un método mejor.

Dicho método existe; no hay más que aplicar el viejo teorema de la programación : "si algo se repite mucho, haz funciones". Considérense las siguientes reglas auxiliares:

```
parentAb : PARENT_AB! ;
  exception catch [ RecognitionException re ] { reportError(re); }
parentCe : PARENT_CE! ;
  exception catch [ RecognitionException re ] { reportError(re); }
puntoComa : PUNTO_COMA! ;
  exception catch [ RecognitionException re ] { reportError(re); }
```

Utilizándolos, la regla `llamada` quedaría así:

```
llamada : IDENT parentAb listaExpresiones parentCe puntoComa ;
        exception catch [ RecognitionException re ]
        { sincronizar (re, $FOLLOW); }
```

Ahora la regla ya es muy robusta; cuando hay un error, es capaz de recuperarse casi instantáneamente si se trata de una omisión de sus tokens-trampa. En caso contrario, recurre al sincronizador.

Podemos utilizar `parentAb`, `parentCe` y `puntoComa` para sustituir a los tokens `PARENT_AB!`, `PARENT_CE!` y `PUNTO_COMA!` en cualquier regla de nuestra gramática, sin disminuir un ápice la legibilidad, y obteniendo las ventajas del control de las excepciones.

Las reglas `parentAb`, `parentCe` y `puntoComa` las llamaremos "trampas para excepciones", por la forma que tienen de "capturar" las excepciones y utilizarlas de la manera adecuada, de la misma forma que la trampa de un cazador.

Para obtener un beneficio óptimo del uso de trampas, es necesario saber dónde colocarlas, es decir, qué tokens pueden ser tokens trampa. Unos cuantos consejos:

- Deben utilizarse tokens que sean "obligatorios". A veces no es fácil saber cuándo lo son. Por ejemplo, `PUNTO_COMA` es necesario al final de cada instrucción, luego puede sustituirse por la trampa para excepciones `puntoComa`. Normalmente sabremos que es obligatorio porque ya se ha recorrido parte de la regla en la que está -sabemos que estamos reconociendo una llamada, o una asignación. En otras ocasiones, sin embargo, `PUNTO_COMA` no es "obligatorio"- cuando sirve para codificar la instrucción nula no hay elementos previos que nos adviertan que deba haber un `PUNTO_COMA`, así que en dicha regla no debe utilizarse la trampa `puntoComa`[46].
- Los mejores candidatos para colocar trampas son los símbolos de sincronismo que "cierran", como `PARENT_CE`, `LLAVE_CE` o `PUNTO_COMA`.
- También pueden utilizarse símbolos de sincronismo que "empiecen" (`PARENT_AB`, `LLAVE_AB` ...). No obstante hay que tener cuidado con los tokens que se utilizan como raíz (que llevan detrás el operador de enraizamiento, '^') ya que si se sustituyen por una regla debe

46 En realidad sí se puede; pero no se ganará nada, y se perderá eficiencia en el analizador.

tenerse en cuenta que hay que enraizar el árbol en una acción.

- En general, los tokens de sincronismo son buenos candidatos para los tokens trampa, pero nótese que se trata de conceptos diferentes: los tokens de sincronismo permiten conocer el "estado" de un reconocedor al ser reconocidos mientras que los tokens trampa son tokens que "siempre deben estar", y cuya ausencia provoca un error.
- El método de reconocimiento de ANTLR puede hacer que algunas trampas para excepciones no funcionen correctamente, por reconocerse los errores antes de tiempo, de manera que no se activen (las trampas). Por ejemplo, las trampas para excepciones no funcionarán casi nunca en una gramática con k>1. Con k=1, también fallarán las trampas colocadas detrás de sub reglas opcionales (con el símbolo '?') o de una sub regla de elección de alternativas (con el símbolo '|'), con una de las alternativas vacía. En el siguiente apartado explicaré cómo neutralizar este comportamiento.

6.2.9: Retardo en el tratamiento de errores

El retardo del tratamiento de errores es la segunda técnica auxiliar que presentaremos para mejorar el uso de tokens de sincronismo con ANTLR. Al contrario que la técnica de las trampas para excepciones, se trata de una técnica "complementaria", que solamente puede utilizarse efectivamente añadida a las trampas para excepciones.

El problema de la "nada" y NoViableAltException

Con las trampas para excepciones hemos avanzado bastante. Nuestro primer ejemplo, una vez colocadas, compilará y mostrará el error correspondiente:

```
clase A
{
   método m1(Entero pA)
   {
      pA++ // ¡Error! Falta punto y coma
   }
}
```

No obstante aún es posible mejorar más nuestro mecanismo de resincronización; aún no es capaz de recuperarse de los errores tan bien como debería.

```
clase A
{
   método m1(Entero pA)
   {
      pA= pA + 1 // ¡Error! ¡Falta un punto y coma!
   }
}
```

Si lo compilamos, veremos que nada de lo que hemos hecho hasta ahora funciona con este ejemplo: la instrucción empieza a reconocerse correctamente, pero llegada a un punto se cancela, se entra en modo sincronización y se termina.

Compilando el analizador en modo de traceado[47] podremos observar en qué punto se cancela la operación: en la regla `expPostIncremento`. Parece que con las trampas para excepciones no ha sido bastante. Esta regla sigue resistiéndose.

Analicemos de nuevo el código de `expPostIncremento` para ver qué es lo que ocurre.

47 Con la opción -trace

`expPostIncremento` es una regla formada por una referencia a otra regla, `expNegacion`, seguida de una sub regla opcional (con el operador ?)[48]:

```
expPostIncremento :  expNegacion (OP_MASMAS^ | OP_MENOSMENOS^)? ;
```

Internamente ANTLR transforma las sub reglas opcionales en "alternativas que pueden evaluar a vacío". Es decir, internamente, ANTLR manejará `expPostIncremento` así:

```
expPostIncremento :  expNegacion (OP_MASMAS^ | OP_MENOSMENOS^ | /* nada */ ) ;
```

Ahora consideremos cómo implementa ANTLR las alternativas.

Cuando se llega a un punto en el que hay que reconocer una alternativa, con k=1, ANTLR utiliza un switch sobre el token actual para saber qué alternativa elegir. Así, cuando se genere el código para `expPostIncremento`, tenemos lo siguiente (ocultaremos algunas partes que no resultan interesantes para esta explicación):

```
public final void expPostIncremento() throws RecognitionException,
TokenStreamException
{
   ...
   try {      // manejador de errores
      expNegacion(); // reconoce la primera negación
      ...
      switch ( LA(1))  // switch sobre el token actual
      {
         case OP_MASMAS: // PostIncremento
         {
            ...
            match(OP_MASMAS);
            break;
         }
         case OP_MENOSMENOS: // PostDecremento
         {
            ...
            match(OP_MENOSMENOS);
            break;
         }
         /* nada */
         case OP_Y: case OP_O: case PUNTO_COMA: case COMA:
         case PARENT_CE: case OP_IGUAL: case OP_DISTINTO: case OP_ASIG:
         case OP_MENOR: case OP_MAYOR: case OP_MENOR_IGUAL:
         case OP_MAYOR_IGUAL: case OP_MAS: case OP_MENOS: case OP_PRODUCTO:
         case OP_DIVISION:
         {
            break;
         }
         default:
         {
            throw new NoViableAltException(LT(1), getFilename());
         }
      }
      ... // construir el AST
   }
   catch (RecognitionException ex) {
      ... // código de sincronización
```

48 Estamos obviando el manejador de excepciones. En este caso da igual que sea el modo pánico o un sincronizador; *el problema es que se entra en el manejador*, no cual se utiliza.

```
    }
...
}
```

Como puede verse, el código comienza reconociendo la expresión de negación (llamando a `expNegacion`). Detrás de él viene el `switch` para la sub regla opcional. Observemos sus casos.

El caso primero sirve claramente para reconocer el operador de post incremento, mientras que el segundo reconoce el post decremento. El tercer caso es el que sirve para reconocer la tercera alternativa, que es "nada". Esta regla se limita a comprobar que el siguiente token que llega al analizador pertenece a `SIGUIENTE(expNegacion)`, y terminar.

Por último, si el token actual no se corresponde con ninguno de los casos, en el caso default se lanza una excepción de alternativa no válida (`NoViableAltException`) sobre el token actual.

Esa es exactamente la excepción que se lanzó en nuestro ejemplo. El token que el analizador recibió fue token `LLAVE_CE`, que no pertenece a ninguno de los casos del switch. Se lanza la excepción, se entra en el código de sincronización, y se pierde el AST que se estaba construyendo.

La política que Terence Parr siguió cuando desarrolló ANTLR fue la de detectar los errores lo más pronto posible. En muchas ocasiones es la mejor solución, pero no en todas. Por ejemplo, el mecanismo de las trampas de excepciones se basa en reconocer los errores "en el momento en el que pueden producirse", y no antes.

Lo que quiero decir es que con la implementación por defecto de las sub reglas opcionales el error de que falta un punto y coma se reconoce demasiado pronto: nosotros pretendemos que el error sea controlado por la trampa `puntoComa`, pero llega `expPostIncremento` y "nos lo quita" antes de tiempo.

En general, si tenemos un sistema de recuperación de errores basado en "tratar los errores lo más tarde posible", éste chocará frontalmente con las reglas opcionales, en las que la política por defecto de ANTLR es "tratar los errores lo más pronto posible".

Existen varios caminos para desactivar este comportamiento por defecto. Nosotros vamos a centrarnos en capturar las excepciones.

Capturando las excepciones

¿Qué hacemos con cuando k tiene que ser >1? ¿O cuando deseamos tratar el error localmente (en la regla)? O, como en nuestro caso, ¿qué hacemos si hay un bug en ANTLR que impide desactivar las excepciones?

Solamente queda una opción: capturarlas. Solamente tenemos que buscar el manejador de excepciones adecuado.

Para capturar la `NoViableAltException` no podemos utilizar el manejador de excepciones de la regla, porque precisamente el objetivo de todo este apartado ha sido evitar llegar a dicho manejador (si utilizamos el manejador global de la regla perdemos el AST). Tenemos que capturar la excepción, pero de manera que la construcción del AST de la regla no se cancele. Así que deberemos utilizar un manejador de excepciones para un elemento con nombre o para una alternativa.

Especificar un manejador para cada una de las alternativas de la regla opcional de `expPostIncremento` (`OP_MASMAS`, `OP_MENOSMENOS` y nada) es posible, pero es muy engorroso. Por lo tanto la única alternativa que nos queda es utilizar los nombres. Es decir, queremos hacer algo así:

```
expPostIncremento :  expNegacion opPost:(OP_MASMAS^|OP_MENOSMENOS^)?
                  ;
exception [opPost] catch [RecognitionException e] { ... }
```

Donde "..." puede ser una instrucción nula (anulando efectivamente el lanzamiento de la excepción) o una llamada a `sincronizar`, de manera que solamente se ignore la excepción cuando se encuentre un símbolo de sincronismo.

Cuando compilemos el código veremos que ANTLR lo rechaza: ANTLR no admite nombres de sub reglas, al menos no en su versión 2.7.2.

Ante este problema, la única solución posible es utilizar una regla-subregla. Es decir, dividir la regla `expPostIncremento` en dos:

```
expPostIncremento :  expNegacion expPostIncremento2
                  ;
expPostIncremento2 : (OP_MASMAS^|OP_MENOSMENOS^)? ;
exception catch [RecognitionException e] { ... }
```

Pero aún seguirá quedando un problema: ¡la raíz!

Cuando se divide una regla en varias "reglas-subreglas", el problema es que es posible perder la raíz del AST. Este es exactamente nuestro caso. Al hacer que `OP_MASMAS` y `OP_MENOSMENOS` pertenezcan a un subárbol, estamos impidiendo al modo por defecto de construcción que los haga raíces. Si queremos que sean raíces vamos a tener que ayudar a ANTLR un poco:

```
expPostIncremento :  operando:expNegacion operador:expPostIncremento2
                     { ## = construyeExpUn(#operando,#operador); }
                  ;
```

Donde el método `construyeExpUn` será:

```
public AST construyeExpUn(AST operando, AST operador)
{
   if(null==operador) return operando;
   if(operando.getNextSibling()==operador)
   {
      operando.setNextSibling(null);
   }
   operador.setFirstChild(operando);
   return operador;
}
```

Y con esto conseguiremos desactivar la excepción producida en `expPostIncremento`. Pero el precio ha sido alto: hemos tenido que modificar el cuerpo de una regla, haciéndola menos legible, y añadir un método para reconstruir el AST si fuera necesario. ¡Debemos intentar evitar que nuestras gramáticas sean k>1 siempre que sea posible!

Aún no hemos terminado...

Sea cual sea el método que hayamos elegido para solucionar el problema de `NoViableAltException`, debe quedar claro que lo hemos solucionado en *una* regla. El siguiente ejemplo:

```
clase A
{
   método m1(Entero pA)
   {
      pA= pA + 1 // ¡Error! ¡Falta un punto y coma!
   }
}
```

no compilará aún de la mejor manera posible: ¡la instrucción sigue sin reconocerse!

Lo que ocurre es que aunque hayamos arreglado el problema de `NoVibleAltException` con `expPostIncremento`, no lo hemos hecho con el resto de expresiones. Si compilamos con la opción -trace nuestro analizador veremos que el problema se encuentra más arriba en la jerarquía de expresiones, concretamente en `expAsignacion`.

```
expAsignacion : expOLogico (OP_ASIG^ expOLogico)? ;
```

De nuevo hay una sub regla opcional que nos impedirá utilizar nuestra estrategia de sincronización.

Necesitaremos capturar la excepción, de nuevo dividiendo la regla en varias reglas-subreglas, y teniendo que realizar manualmente el enraizado:

```
{
   public AST construyeExpBin ( AST izquierda, AST derecha )
   {
      if(derecha != null)
      {
         if(izquierda.getNextSibling()==derecha)
         {
            izquierda.setNextSibling(null);
         }

         AST valor = derecha.getFirstChild();
         return #(derecha, izquierda, valor);
      }
      return izquierda;
   }
...
}
...

/** Asignaciones (nivel 9) */
expAsignacion : izq:expOLogico der:expAsignacion2
                { construyeExpBin(#izq, #der); }
              ;

protected expAsignacion2 : (OP_ASIG^ expOLogico)? ;
   exception catch [NoViableAltException nvae]
   { sincronizar(nvae, $FOLLOW); }
```

En este caso, al tratarse de una expresión binaria, hemos tenido que implementar un método diferente. Éste se parece mucho a `construyeExpUn`, con la excepción de que realiza algunas transformaciones previas con el AST.

Más adelante veremos cómo hemos empleado los tokens de sincronismo, trampas para excepciones y retardo en la recuperación en nuestro compilador de LeLi.

Sección 6.3: Implementación - Herencia de gramáticas

6.3.1: El problema

En este apartado responderemos a una pregunta muy simple :¿Cómo añadimos la gestión de errores al analizador sintáctico?

Si le preguntamos a nuestro proverbial programador primerizo, nos responderá más o menos "¡Pues modificando el código del analizador!". Este impulso modificador es muy comprensible. Si un código no realiza todas las funciones que se esperan de él, o no las realiza de la manera adecuada, lo lógico es modificarlo su hasta que lo haga.Y desde luego reescribir algunas de sus partes es la manera más inmediata para hacerlo.

Ahora bien, "modificar" no tiene por qué ser necesariamente "reescribir". Antes de reescribir el analizador, pensemos en los pros y los contras. Desde luego es la estrategia más rápida de implementar, por ser la más directa. Pero dado que lo que se van a añadir son acciones y manejadores de excepciones, se añadirá mucho código java. El analizador se hará dependiente del código java, además de ser cada vez más y más extenso.

¿Qué ocurrirá si, en el futuro, queremos realizar modificaciones en el analizador? Por ejemplo, podemos necesitar un analizador para C++. O un analizador con una gestión de errores radicalmente diferente.

Ya hemos visto los inconvenientes principales de reescribir un analizador para que haga algo que no sea reconocer. Pero si no se reescribe el analizador directamente ¿Qué otra opción hay?

ANTLR proporciona un mecanismo ideal para este caso: la herencia de gramáticas.

6.3.2: Presentando la herencia de gramáticas en ANTLR

Imaginemos un analizador léxico[49] que reconozca solamente identificadores (con caracteres ingleses):

```
class EnglishIdentLexer extends Lexer;
options {
charVocabulary = '\3'..'\129'; // Solamente ASCII básico
}

LETRA : ('a'..'z') | ('A'..'Z') | '_' ;
DIGITO : ('0'..'9');
IDENT : LETRA(LETRA|DIGITO)* ;
```

Con antlr es posible utilizar la definición de `EnglishIdentLexer` para definir un nuevo analizador léxico que lo extienda. Por ejemplo, para definir un analizador que además reconociera los enteros, bastaría con hacer lo siguiente:

```
/* se extiende EnglishIdentLexer, y no Lexer */
class EnglishIdentIntLexer extends EnglishIdentLexer;

ENTERO : (DIGITO)+;
```

Este analizador reconoce tanto los `IDENT`s como los `ENTERO`s (llamando a las reglas auxiliares `LETRA` y `DIGITO` cuando es necesario). Para implementarlo solamente ha sido necesario añadir la regla que hace falta, aprovechándose la definición previa.

49 Aunque para este ejemplo voy a utilizar un analizador léxico, la herencia de gramáticas se puede aplicar igualmente en analizadores sintácticos o semánticos.

Además, si hacemos algún cambio en la gramática padre (añadiendo una regla, o modificándola) los cambios se verán reflejados en las gramáticas derivadas de ella. Eso sí, habrá que recompilar.

6.3.3: Herencia ANTLR != Herencia java

Un primer vistazo puede hacernos pensar que para utilizar la herencia de gramáticas ANTLR se limita a utilizar el mecanismo de herencia de java. Es decir, que `EnglishIdentLexer.java` comienza así:

```
public class EnglishIdentLexer extends antlr.CharScanner
{
   ...
```

,y que `EnglishIdentIntLexer.java` comienza así:

```
public class EnglishIdentIntLexer extends EnglishIdentLexer
{
   ...
```

Y que en la subclase se añade el método `ENTERO` y eso es todo.

Pues bien, esto NO es cierto. Hay múltiples diferencias. La primera y más importante es que el analizador tendrá esta otra forma:

```
public class EnglishIdentIntLexer extends antlr.CharScanner
{
   ...
```

Por si no ha quedado claro en el título de este apartado, lo voy a escribir bien claro:

Herencia ANTLR != Herencia java

¿Por qué no se utiliza la herencia java? ¿Cómo se implementan las reglas de la supergramática en una gramática derivada?

Imaginemos que necesitamos un analizador que reconozca los identificadores y los números enteros, pero utilizando un juego de caracteres más grande – por ejemplo, el conjunto de caracteres del castellano. Si decidimos extender de `EnglisIdenIntLexer`, tendremos que hacer dos cosas:

- Cambiar la sección `options` para cambiar la opción `charVocabulary` a unicode occidental.
- Cambiar la definición de la regla `LETRA`.

```
/* extendemos a EnglishIdentIntLexer */
class universalLexer extends EnglishIdentIntLexer ;

/* Cambiamos el juego de caracteres válidos */
options
{ charVocabulary = '\3'..'\377'; }

/* Cambiamos LETRA */
LETRA : (a..z) | (A..Z)
      | 'á' | 'é' | 'í' | 'ó' | 'ú' | 'Á' | 'E' | 'Í' | 'Ó' | 'Ú'
      | 'ñ' | 'Ñ'
      | 'ü' | 'Ü'
      ;
```

Como puede verse, dado que `IDENT` utiliza internamente `LETRA`, no ha sido necesario ningún cambio adicional. Para modificar la sección `options` solamente ha hecho falta volver a escribirla.

¿Dónde está entonces el problema? ¿Por qué la herencia de ANTLR no puede basarse en la de java? Para poder responder, necesitamos investigar un poco más. Fijémonos en el método que implementa la regla `IDENT`.

La regla `IDENT` tiene siempre la misma forma : una `LETRA` seguida de un conjunto de `LETRA`s y `DIGITO`s.

¿Cómo comenzará el método? Testeará en la entrada el siguiente carácter para ver si es una `LETRA`... ¡Pero una `LETRA` es diferente en `EnglishIdentLexer` y en `UniversalLexer`!

La conclusión es que en `UniversalLexer` no solamente es necesario cambiar `LETRA`, también es necesario cambiar `IDENT`, porque su lookahead cambia si cambia el conjunto `PRIMERO(LETRA)`.

Herencia ANTLR != Herencia java, porque cambia el lookahead de algunas reglas de la gramática base.

Lo más sencillo ante esta problemática es simplemente copiar las reglas de la gramática base en la nueva gramática, y proceder como si se tratara de una gramática nueva.

Traducido del manual de ANTLR – fichero inheritance.html

Determinar qué reglas de la gramática base son afectadas [por el lookahead] no es sencillo, por lo que nuestra implementación simplemente hace una copia de la gramática base y genera un analizador completamente nuevo con las modificaciones apropiadas. Desde el punto de vista del programador, la compartición de código/gramáticas habrá ocurrido, mientras que desde el punto de vista de la implementación lo que se ha realizado es una copia de [las reglas de] la clase base.

*****

Si se crea un analizador llamado A haciéndolo subclase de otro analizador B, se crearán fichero intermedio (llamdado `extendedA.g`) que contendrá todas las reglas que se han añadido o modificado en A *y todas las reglas de B que han permanecido intactas.* Ese fichero será el que convierta en código java.

Bien. Ya hemos visto *una* de las razones por las que la herencia de gramáticas no es equivalente a la herencia de clases de java. Pero falta otra razón. Tiene que ver con los métodos y atributos que se añaden a la clase.

En ANTLR es posible añadir constructores, métodos y atributos al analizador, simplemente escribiendo su código entre llaves antes de la zona de reglas. Así:

```
class CountIdentsParser extends Parser;

{
   private int countIdents;
   CountIdentsParser(TokenStream stream)
   {
      super(stream);
      countIdents=0;
   }
}

regla : (IDENT {countIdents++;} )* ;
```

El analizador sintáctico de arriba sirve para reconocer una entrada formada por identificadores y

contarlos. He añadido un atributo privado (`countIdents`) y un constructor que lo inicia a 0[50].

Considérese ahora la clase siguiente:

```
class CountIdentsIntsParser extends CountIdentsParser;

{
   private int countInts;
   CountIdentsIntsParser(TokenStream stream)
   {
      super(stream);
      countInts=0;
   }
}

regla : (IDENT {countIdents++;} | ENTERO{countInts++;} )* ;
```

Hemos creado una sub gramática de `CountIdents`, a la que hemos añadido un atributo (`countInts`) para contar los números enteros además de los identificadores.

Si la herencia de ANTLR funcionase como en java todo iría bien. *Si funcionase como en java.*

La herencia de gramáticas tiene un comportamiento un tanto peculiar con respecto a las acciones de ámbito global (los atributos y métodos añadidos utilizando las llaves en la sección de gramática) de un analizador: la acción inicial del analizador es *sobreescrita*, y no *añadida*. Es decir, que el constructor de la clase `CountIdentsParser` se ha perdido, así como la definición del atributo `countIdents`. Por lo tanto la clase no compilará – se hace referencia a un elemento, `countIdents`, que no está definido.

Para obtener el resultado requerido sería necesario volver a incluir en la acción el atributo `countIdents`, e iniciarlo en el constructor.

```
class CountIdentsIntsParser extends CountIdentsParser;

{
   private int countIdents;
   private int countInts;
   CountIdentsIntsParser(TokenStream stream)
   {
      super(stream);
      countIdents=0;
      countInts=0;
   }
}

regla : (IDENT {countIdents++;} | ENTERO{countInts++;} )* ;
```

Claro está que llegados a éste punto ya no merece la pena utilizar la herencia de gramáticas, pero evidentemente el único valor de estas gramáticas es el documentativo. Además, ni siquiera compilarían : es necesario especificar un vocabulario en el que se definan los tokens `IDENT` y `ENTERO`.

6.3.4: Línea de comandos

Para poder compilar una gramática heredada de otra, es necesario especificar el fichero en el que

50 Un buen ejercicio sería utilizar el contador para limitar la entrada de caracteres – implementar un analizador que reconozca exactamente n enteros, siendo n una variable que debe suministrarse al crear el analizador. Pista: Sería necesario utilizar un predicado semántico en la regla y modificar la acción, posiblemente añadiendo un atributo y un parámetro al constructor.

se encuentra la segunda en la línea de comandos, con la opción -glib. Así, si en el fichero f1.g hay subgramáticas de otras presentes en el fichero f2.g, para compilarlas habrá que escribir:

```
c:\> java antlr.Tool -glib f2.g f1.g
```

Es posible añadir varios ficheros con la opción glib, separándolos con punto y coma. Por ejemplo, si f1.g "depente" de f2.g y f3.g:

```
c:\> java antlr.Tool -glib f2.g;f3.g f1.g
```

6.3.5: ¿Cómo se relaciona todo esto con la RE?

Muy sencillo: en lugar de reescribir el analizador sintáctico directamente, vamos a crear, utilizando la herencia de gramáticas, un analizador derivado de él, llamado `LeLiErrorRecoveryParser`, que estará definido en el fichero `LeLiErrorRecoveryParser.g`. La estructura de dicho fichero será la siguiente:

```
header{
   package leli;
}

class LeLiErrorRecoveryParser extends LeLiParser;
options {... <opciones modificadas> }
{
   ... <métodos, atributos y constructores en java>
}
... <reglas sobreescritas y nuevas reglas>
```

Será en dicho fichero en el que escribiremos todos los cambios que deseemos hacer a LeLiParser.

6.3.6: Importación de vocabulario

Al programar `LeLiParser` hemos declarado, en la sección tokens, una serie de tokens imaginarios que utilizábamos para múltiples propósitos, desde enraizar listas hasta representar el tipo vacío. Nuestro analizador debe ser capaz de reconocer y utilizar dichos tokens.

Por supuesto, `LeLiErrorRecoveryParser` tiene que ser capaz, además, de utilizar todos los tokens definidos en el nivel léxico, como `IDENT`, `RES_SI`, etc.

La herencia de gramáticas debería controlar por sí misma el problema de reconocer todos esos tokens, pero desgraciadamente no lo hace – recuérdese que mientras que las reglas son representadas con métodos, los tokens son representados como enteros en una interfaz java que hay que implementar.


La herencia es a nivel de *reglas*, no a nivel de tokens.

¿Cómo resolvemos el problema? Los que hayan estado atentos ya lo sabrán: el propio LeLiParser exporta todos los tokens que utiliza (incluyendo los del nivel léxico) con la opción `exportVocab`, lo que provoca la generación de un fichero de texto y un interfaz java.

Normalmente estos ficheros están destinados al analizador semántico, pero nada impide a nuestro analizador sintáctico utilizarlos también. Así que modificaremos las opciones para que tengan el siguiente aspecto:

```
class ErrorRecoveryParser extends LeLiParser
options
{
   importVocab = LeLiParserVocab;
}
...
```

Estamos suponiendo que no se van a declarar tokens nuevos en nuestro nuevo analizador, sino que se va a reutilizar el conjunto de tokens de `LeLiParser`. Si no fuera así, será necesario además exportar el nuevo conjunto de tokens utilizando `exportVocab`, para que el análisis semántico no se encuentre con tokens desconocidos.

Sección 6.4: Control de mensajes: La clase Logger

6.4.1: El problema

Ya hemos visto que el modo pánico de ANTLR consiste en, una vez capturada una excepción, pasársela al método `reportError`[51]:

```
public void regla r1()
{
   try{
   ... (cuerpo de la regla)
   }catch (RecognitionException e) {
      reportError(e);
      consume();
      consumeUntil(_tokenSetXX);
   }
}
```

`reportError` es un método implementado en la clase padre de nuestro analizador. Esta clase es antlr.Parser en el caso del analizador sintáctico, `antlr.CharScanner` en el caso del ananizador léxico y `antlr.TreeParser` en el caso del analizador semántico.

Dependiendo de qué tipo de analizador se esté utilizando, `reportError` admite un tipo de parámetro diferente: `RecognitionException` en el caso del analizador sintáctico o `TokenStreamException` en el caso del analizador léxico. Centrándonos en el analizador sintáctico, el código de `reportError` (implementada en `antlr.Parser`) es el siguiente:

```
    /** Parser error-reporting function can be overridden in subclass */
    public void reportError(RecognitionException ex) {
        System.err.println(ex);
    }
```

No es muy complejo: el método se limita a imprimir por pantalla la excepción. El comentario invita a modificar el método sobreescribiéndolo en nuestro analizador, y eso es lo que vamos a hacer. Para ello vamos a utilizar una clase especial, `Logger`, que describiré más adelante.

Ya hemos presentado la estrategia que vamos a seguir para *mostrar* los mensajes. Sin embargo no nos hemos planteado si los mensajes que ANTLR proporciona por defecto son adecuados.

Uno de los inconvenientes que observo en ANTLR es el idioma de los mensajes de error. Intentando realizar un compilador lo más "hispano" y amigable posible, no puedo dejar de pensar en modificar los mensajes de error para que en lugar de decir:

```
fichero:lin:col expecting IDENT, found PARENT_AB
```

digan:

```
fichero:lin:col se esperaba un identificador, se encontró un paréntesis abierto
('(')
```

Nótese que el mensaje de abajo se diferencia del de arriba no solamente en el idioma utilizado ("se esperaba"... ", se encontró"...) sino que también se han modificado los "nombres" de los tokens ("un identificador" en lugar de `IDENT`, "un paréntesis abierto" en lugar de `PARENT_AB`).

Vamos a ir viendo estos problemas de uno en uno.

51 ¡Atención! El código que muestro a continuación es para el *análisis sintáctico*. El modo pánico en el análisis léxico y semántico es, como es lógico, diferente, pues utiliza diferentes métodos y tipos. Sin embargo la funcionalidad es la misma.

6.4.2: La clase Logger del paquete antlraux

Volvamos por ejemplo al método `reportError` de la clase `antlr.Parser`. La implementación de este método es muy poco flexible; para empezar, no hay forma de cambiar el flujo por el que se emite el mensaje; simplemente se utiliza `System.err`. Además, no hay manera de modificar la forma en la que estos resultados se muestran en pantalla (se utiliza el método `toString()` de la clase `RecognitionException`). Por último, no hay manera de mantener un conteo de errores.

Para resolver todos estos problemas he implementado una clase, llamada `Logger`, que permite realizar todo lo anterior. Dado que estará dentro del paquete `antlraux.util`, su nombre completo será `antlraux.util.Logger`.

Si aún no se ha hecho, es muy aconsejable incluir el paquete `antlraux` en el `classpath` de ahora en adelante.

Su interfaz es la siguiente:

```
public class Logger
{
// Constructores
   public Logger( String name );
   public Logger( String name, OutputStream s );
   public Logger( String name, OutputStream s, String nls, String ts );
   public Logger( String name, OutputStream s, String nls, String ts,
FileLineFormatter flf );

// Añadir mensajes
   public void log ( String msg, int msgLevel );
   public void log ( String msg, int msgLevel, antlraux.util.LexInfo li);
   public void log ( String msg, int msgLevel, String fn,int line,int column );
   public void log ( String msg, int msgLevel, LexInfo li );
   public void print( String msg );

// Tabulación
   public void tabulate(); // Imprime "level" tabulaciones
   public void newLine();   // Imprime "newLineString" en el OutputStream
   public void incTabs();   // Aumenta el número de tabulaciones
   public void decTabs();   // Disminuye el número de tabulaciones
   public void setTabLevel();
   public void getTabLevel();

// Conteo de mensajes
   public int getLogCounter();
   public int resetLogCounter();

// Filtrado
   // nivel mínimo de los logs para que sean impresos
   public void setMinLogLevel(int minLogLevel);

}
```

Hay otros métodos (principalmente métodos "set" y "get"), pero éstos son los principales.

Esta clase ofrece mucha más flexibilidad en el tratamiento de los mensajes de error. Para empezar, el parámetro `type` del método `log` permite mostrar tanto mensajes de error como mensajes de aviso (*warnings*) o mensajes de información general. Además, es posible modificar la manera en la que la información se muestra, distribuyéndola en varios niveles utilizando las tabulaciones o modificando la forma en la que se muestran el nombre de fichero, la línea y la

columna donde se ha producido el error con el parámetro `flf` del tercer constructor. Por último es posible llevar la cuenta del número de errores que se han tenido con ayuda de los métodos `getLogCounter` y `resetLogCounter`.

Para poder utilizar `Logger` en `LeLiErrorRecoveryParser` necesitamos varias cosas:

- Hay que incluir (con `import`) la clase `antlraux.util.Logger`.
- `LeLiErrorRecoveryParser` debe contener una instancia de `Logger` como atributo.
- Hay que añadir un constructor al analizador que incluya un `Logger` como parámetro.
- Por último, hay que reescribir el nuevo método `reportError` para que use el `Logger`.

Para importar la clase `antlraux.util.Logger` en el analizador solamente hay que modificar la sección header en `LeLiErrorRecoveryParser.g`, anadiéndole la línea correspondiente:

```
header
{
   package leli;

   import antlraux.util.Logger;
}

class LeLiErrorRecoveryParser extends LeLiParser;
{
  ... código java (de momento vacío)
}
```

El resto de los cambios que vamos a efectuar los realizaremos sobre la sección de código java. Para empezar, añadir un atributo de tipo Logger y un constructor apropiado es tan sencillo como:

```
class LeLiErrorRecoveryParser extends LeLiParser;
{
   Logger logger = null;

   /* Nuevo constructor */
   LeLiErrorRecoveryParser( TokenStream lexer, Logger _logger )
   {
      this(lexer);
      logger = _logger;
   }
}
```

Y, finalmente, queda el método `reportError`. Por comodidad vamos a dividirlo en dos: el "normal", que recibirá una `RecognitionException` como parámetro, y el que utilizaremos como base, que recibirá el mensaje, nombre de fichero, línea y columna donde se produjo el error.

```
   /* reescribe antlr.Parser.reportError */
   public void reportError( RecognitionException e )
   {
      reportError(
         e.getMessage(), e.getFilename(), e.getLine(),e.getColumn() );
   }

   /* método auxiliar */
   public void reportError(
      String msg, String filename, int line, int column )
   {
      if( null==logger ) logger = new Logger( "error", System.err );

      logger.log( msg, 1, filename, line, column);
   }
```

En el segundo podemos ver que se realiza un test para ver si el `Logger` se ha creado o no (podría no haberlo hecho si el analizador no se hubiera creado con el constructor que hemos suministrado) y lo crea si es necesario.

Por supuesto, la clase `leli.Tool` (que se encargaba de coordinar todos los analizadores – ver final del capítulo anterior) también deberá ser modificada. Antes de crear nuestro analizador tendremos que crear un `Logger` para pasárselo al analizador en el constructor:

```
public class Tool
{
   ...
   Logger logger = new Logger(System.err);
   ErrorRecoveryParser parser = new ErrorRecoveryParser(lexer, logger);
   parser.programa();
   ...
}
```

Sección 6.5: Mejorando los mensajes de error

6.5.1: Introducción

En esta sección veremos cómo mejorar los mensajes de error emitidos por nuestro compilador de varias maneras:

- Cambiando el idioma de los mensajes de error
- Utilizando los alias de los tokens

6.5.2: Cambiando el idioma de los mensajes de error

Para averiguar cómo traducir los mensajes de error de ANTLR nos vamos a basar en una afirmación que ya he hecho anteriormente:

En ANTLR, todo error de reconocimiento conlleva el lanzamiento de una excepción.

Concretamente, conlleva el lanzamiento de una subclase de `ANTLRException`. En la documentación de ANTLR se presenta un esquema parecido al siguiente:

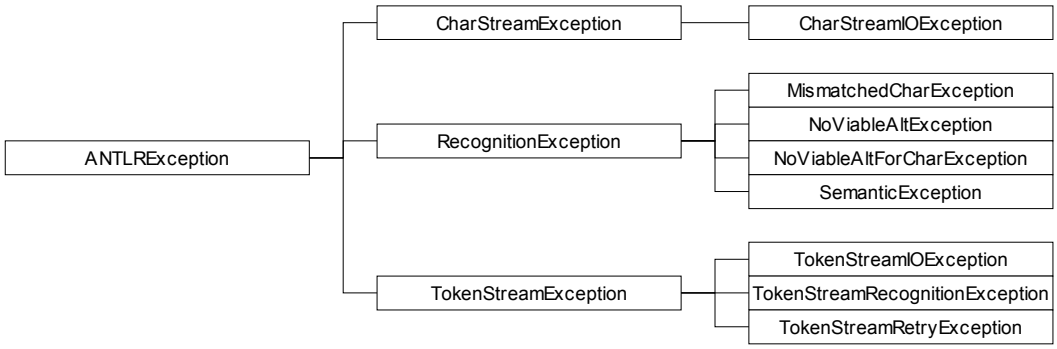

Ilustración 6.3 Relación jerárquica de las excepciones de ANTLR

Aunque es probable que la situación cambie en el futuro, de momento los mensajes de error de ANTLR están codificados "en duro" en las excepciones de antlr. Es decir, para poder efectuar la traducción habrá que modificar el código fuente de las excepciones y recompilar ANTLR.

Vamos a ir comentando para qué sirve cada una de estas excepciones y cómo hay que modificarlas.


Las definiciones de las excepciones han sido traducidas del manual de ANTLR.. Fichero err.html.

Excepción	Descripción y modificaciones
`ANTLRException`:	Es la raíz de la jerarquía de excepciones. No es necesario modificarla.
`CharStreamException`:	Es lanzada cuando algo malo ocurre en el flujo de caracteres. La mayor parte del tiempo se trata de un problema de entrada salida. No es necesario modificarla.

`CharStreamIOException`:	Ha habido un error de E/S en el flujo de caracteres de entrada. Los errores de E/S los dejaremos en inglés, porque en última instancia provienen de excepciones de E/S de java. No hay que modificarla.
`RecognitionException`:	Un problema genérico de reconocimiento en la entrada. Todas las reglas de los analizadores sintácticos pueden lanzarla. Hay que modificar el constructor por defecto, cambiando "parsing error" por "error de reconocimiento".
`MismatchedCharException`:	Es lanzada por `CharScanner.match()` cuando está espererando un carácter pero encuentra otro en el flujo de entrada. Buscar y reemplazar en todo el fichero: • "Mismatched char" por "Carácter inválido" • "expecting" por "se esperaba" • ", found" por ", se encontró" • "expecting anything but '" por "se esperaba cualquier cosa menos '" • "'; got it anyway" por "'; se aceptó de todos modos" • "expecting token" por "se esperaba un token" • " NOT" por "que NO estuviera" • " in range: " por " en el rango: " • " one of (" por " un carácter de los siguientes (" • "), found " por "), se encontró "
`MismatchedTokenException`:	Lanzada por `Parser.match()` cuando está esperando un token pero encuentra otro. Buscar y reemplazar: • "Mismatched Token: expecting any AST node" por "Token no válido: se esperaba cualquier nodo AST" • "Mismatched Token" por "Token no válido" • "<empty tree>" por "<árbol vacío>" • Mensajes equivalentes a los de MismatchedCharException pero para tokens.
`NoViableAltException`:	El analizador sintáctico encuentra un token que no comienza ninguna alternativa en la opción actual. Buscar y reemplazar: • "unexpected token: " por "token inesperado: " • "unexpected end of subtree" por "fin del subárbol inesperado" • "unexpected AST node: " por "nodo AST inesperado: "
`NoViableAltForCharException`:	El analizador léxico encuentra un carácter que no comienza ninguna alternativa en la opción actual. Buscar y reemplazar "unexpected char: " por "carácter inesperado: ".
`SemanticException`:	Excepción lanzada cuando un predicado semántico se incumple, o en una acción (mediante `throw`). No es necesario modificarla.

`TokenStreamException`:	Indica un error genérico en el flujo de tokens. No necesita cambios.
`TokenStreamIOException`:	Convierte una `IOException` en una `TokenStreamException`. No modificaremos los mensajes que vienen directamente de java.
`TokenStreamRecognitionException`:	Convierte una `RecognitionException` en `TokenStreamException` para que pueda pasarse en el flujo. No necesita modificaciones.
`TokenStreamRetryException`:	Se lanza cuando se ha abortado el reconocimiento de un token. No necesita cambios.

*****

Unas cuantas notas para acabar:

- ANTLR contiene algunas excepciones que no forman parte de la jerarquía basada en `antlr.ANTLRExcepcion`, como `antlr.FileCopyException`. No obstante no es posible traducirlas (provienen de errores de `java.io`).
- Un efecto secundario muy curioso de traducir las excepciones de ANTLR es que ahora el propio ANTLR las utiliza, así que a partir de ahora nos hablará en español la mayor parte del tiempo.
- Solamente gracias a que ANTLR es un software de código abierto (*open source*) hemos podido acceder a su código y modificar ciertos elementos para recompilar. Una licencia más restrictiva no nos lo habría permitido.

6.5.3: Alias de los tokens

El comportamiento predeterminado de ANTLR para mostrar los tokens en los mensajes de error es utilizar su nombre, es decir, la cadena que se obtiene al llamar al método `getText()` del token.

Normalmente `getText()` proporciona un texto que es adecuado para el usuario. Funciona especialmente bien con los tokens que tienen un texto predeterminado, como los las palabras reservadas o los operadores. Así, para la palabra reservada `RES_MIENTRAS` se muestra su texto correspondiente, es decir, "mientras". Un error provocado por la omisión de dicha palabra sería de esta forma:

```
fichero:linea:columna: Se esperaba "mientras"
```

En otros casos la función `getText()` no funciona tan bien. Por ejemplo, en el caso de los identificadores. Tal y como lo hemos construido, y según el modo de funcionamiento normal de ANTLR, cada token `IDENT` devuelve con `getText()` un texto diferente – concretamente el nombre del identificador. Es decir, que una vez reconocido "manolo" como un `IDENT`, entonces al invocar al método `getText()` sobre dicho ident obtendremos – obviamente – "manolo".

¿Dónde está, entonces, el problema?

El problema es que el `IDENT` solamente devuelve "manolo" *una vez que se ha reconocido "manolo" como un* `IDENT`. Pero este no es el caso de los errores de omisión.

En ciertas ocasiones se espera un `IDENT` en una expresión. Si dicho `IDENT` no se proporciona, debe lanzarse un mensaje de error. Se escribirá "Se esperaba " y a continación la cadena que se obtenga al llamar a `getText()` sobre un `IDENT`. *Pero esta vez el identificador no ha sido*

reconocido en la entrada. En un caso así, ¿qué cadena de texto devolverá `getText()`?

En un caso así ANTLR devuelve el texto por defecto de los tokens imaginarios, que viene a ser el texto que se ha utilizado para nombrarlos en el analizador. Por ejemplo, el texto por defecto del token `IDENT` es exactamente ése, "IDENT".

Así, el mensaje de error por omisión de un `IDENT` será de éste estilo:

```
fichero:linea:columna: Se esperaba IDENT
```

Un usuario avanzado sería capaz de deducir que "IDENT" significa "un identificador". Pero sigue sin ser un buen mensaje de error. ¡Los usuarios del compilador no deberían tener que conocer los nombres por defecto de los tokens imaginarios que utilizamos!

Lo mismo que ocurre con `IDENT` ocurre otros tokens imaginarios a nivel léxico, como `RES_ENTERO`, `RES_REAL` o `RES_CADENA`.

Lo que estamos buscando aquí es, por lo tanto, cambiar el texto por defecto de los tokens imaginarios.

Afortunadamente ANTLR proporciona un mecanismo que resuelve exactamente dicho problema: la opción local `paraphrase`. Esta opción permite cambiar el texto por defecto de los tokens en el analizador léxico. Así que hay que editar el fichero `LeLiLexer.g` y añadir a algunas de las reglas dicha opción.

La primera regla que cambiaremos será la de los identificadores. He señalado la línea que se ha añadido.

```
IDENT
options
{
   testLiterals=true; // Comprobar palabras reservadas
   paraphrase="un identificador";
}
:
   (LETRA|'_') (LETRA|DIGITO|'_')*
;
```

La siguiente regla que vamos a modificar es la que permite reconocer enteros y reales. Éstos eran reconocidos por la regla `LIT_NUMERO`:

```
LIT_NUMERO : (( DIGITO )+  '.' ) =>
                ( DIGITO )+ '.' ( DIGITO )* { $setType (LIT_REAL); }
           | ( DIGITO )+  { $setType (LIT_ENTERO); }
           ;
```

Mientras que tanto `LIT_NUMERO` como `LIT_REAL` se encuentran definidos como tokens imaginarios en la sección tokens del analizador léxico:

```
class LeLiLexer extends Lexer;
options {...}
tokens
{  ...
   LIT_ENTERO; LIT_REAL;
}
```

El mejor lugar para poder definir un alias para estos dos tokens sería la propia sección de tokens; algo así:

```
tokens
{
   ...
   LIT_ENTERO <paraphrase="un literal entero">;
   LIT_REAL <paraphrase="un literal real">;
}
```

Pero desgraciadamente ANTLR no tiene dicha capacidad implementada. Es decir, no se puede añadir un alias a un token imaginario.

La solución será, por tanto, definir dos reglas nuevas, `LIT_REAL` y `LIT_ENTERO`. Pero no va a ser tan fácil como parece. Si tuvimos que definirlas en una sola regla, LIT_NUMERO, fue por una única razón; problemas de lookahead. Si se quiere comprobar a qué me refiero, no hay más que comentar la regla `LIT_NUMERO`, borrar a `LIT_REAL` y `LIT_NUMERO` de la sección de tokens y escribir estas dos nuevas reglas:

```
LIT_REAL
options{ paraphrase="un literal real" }
   :  ( DIGITO )+ '.' ( DIGITO )*
   ;
LIT_ENTERO
options{ paraphrase="un literal entero" }
   :  ( DIGITO )+
   ;
```

ANTLR se negará a compilar este código, argumentando que las dos reglas son infinitamente iguales por la izquierda; si empieza a reconocer una serie de `DIGITO`s no sabrá si se trata de un `LIT_REAL` o un `LIT_ENTERO`. La única manera de salvar este escollo es reconocerlos en una sola regla utilizando un predicado sintáctico para distinguir entre ambos. Justamente como se hace en `LIT_NUMERO`.

Así que por un lado necesitamos tener a `LIT_ENTERO` y `LIT_REAL` declarados como reglas, pero por otro necesitamos reconocer los literales enteros y reales con la regla `LIT_NUMERO`. La solución por la que yo he optado es utilizar una pequeña "trampa", un *hack*, como le dicen en el extranjero, consistente en utilizar reglas "que provoquen un error" en el analizador.

Pensemos en los errores que se dan durante un análisis léxico. En el 99% de las ocasiones se trata de un carácter no válido. Si en un compilador de C intentamos llamar 'amig@' a una variable, obtendremos un error de carácter no válido lanzado por el analizador léxico del compilador, que espera una letra del abecedario[52], un dígito o el carácter de subrayado.

Los caracteres de la entrada se reconocen utilizando el método `antlr.CharScanner.match(char)` – entre otros-. Cuando el carácter que se encuentra no es el adecuado, se lanza una excepción del tipo `MismatchedCharException`.

El plan es el siguiente: elegimos dos caracteres al azar, de entre todo el juego de caracteres Unicode, que no estén permitidos en ANTLR, por ejemplo '@' y '#', y los reconocemos con las reglas `LIT_ENTERO` y `LIT_REAL` respectivamente, *pero inmediatamente después de reconocerlos lanzamos excepciones, simulando que match ha fallado.* Así:

52 Del abecedario ASCII, se entiende. Una 'ñ' tambien provocaría un error.

```
LIT_ENTERO
options { paraphrase = "un literal entero"; }
   : '@' {throw new MismatchedCharException(); }
   ;

LIT_REAL
options { paraphrase = "un literal real"; }
   : '#' {throw new MismatchedCharException(); }
   ;
```

A los literales cadena se les asigna un alias más facilmente. Solamente hay que utilizar `paraphrase`:

```
LIT_CADENA
options { paraphrase="una cadena"; }
:
   '"' !
   ( ~('"'|'\n'|'\r') )*
   '"' !
   ;
```

Por último vamos a poner alias a algunos símbolos de puntuación:

```
PUNTO_COMA
options { paraphrase="un punto y coma (';')"; }
   : ';' ;
COMA
options { paraphrase="una coma (',')"; }
   : ',' ;
LLAVE_AB
options { paraphrase="una llave abierta ('{')"; }
   : '{' ;
LLAVE_CE
options { paraphrase="una llave cerrada ('}')"; }
    : '}' ;
PUNTO
options { paraphrase="un punto ('.')"; }
   : '.' ;
PARENT_AB
options { paraphrase="un paréntesis abierto ('(')"; }
   : '(' ;
PARENT_CE
options { paraphrase="un paréntesis cerrado (')')"; }
   : ')' ;
BARRA_VERT
options { paraphrase="una barra vertical ('|')"; }
   : '&' ;
```

Una nota interesante sobre los alias: se guardan en el fichero de texto en el que se exportan los tokens (`LeLiLexerVocabTokenTypes.txt` en nuestro caso). De esta manera, cualquier analizador que importe el conjunto de tokens obtendrá también los alias, con lo que no hay que modificar los analizadores sintácticos o semánticos para que los utilicen; bastará con "recompilarlos".

Sección 6.6: Aplicación en el compilador de LeLi

6.6.1: Acotación del problema: el fichero de errores

La recuperación de errores es una de esas tareas que siempre están "en construcción". Es como la familia: se extiende y se extiende hasta donde uno esté dispuesto a tolerar.

Lo que quiero decir es que tenemos que ponernos límites. Nuestro analizador estará en la línea que une el analizador "completamente inútil", que cancela la construcción del AST global, y el "imposible analizador perfecto", que es capaz de interpretar cualquier entrada de datos del programador, y recuperarse de cualquier error, corrigiendo el programa de manera que funcione como el usuario quería, independientemente de lo que haya escrito. La cuestión es determinar en qué punto de dicha línea nos vamos a encontrar. Buscamos un equilibrio razonable entre prestaciones y facilidad de implementación.

Una de las maneras más sencillas de encontrar el punto que buscamos es utilizando un fichero de ejemplo, en el cual se encuentren presentes los errores de los que nos queremos recuperar. El fichero de ejemplo, además de mostrar los errores a corregir, es de por sí un excelente banco de pruebas.

Nuestro fichero de errores será el siguiente:

```
// Fichero err.leli

clase Inicio

     metodo inicio(
     {
        Sistema.Imprime("¡Hola mundo!"+nl)
     }

     método m (Entero a, b

        parametro.a = 1;
        Systema.imprime(a)
     }
}
```

El analizador deberá -al menos- ser resistente a los errores que mostramos en dicho fichero. Se trata sobre todo de omisiones de tokens de puntuación. Con que el analizador sea "resistente" a los errores queremos decir que reconozca el texto y muestre los errores por pantalla, pero sea capaz de construir el AST completo y correctamente.

En el estado actual el analizador no llegará muy lejos; no llegará ni siquiera a reconocer la clase, quedando un AST bastante pobre:

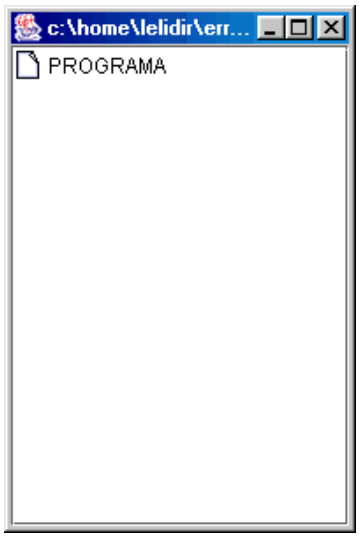

Ilustración 6.4 AST sin recuperación de errores

6.6.2: Aplicando los símbolos de sincronismo

La mayoría de los errores que encontramos en el fichero de ejemplo son omisiones de símbolos de puntuación – bien podríamos llamarlos de sincronismo. La situación en la que nos encontramos es ideal para probar la técnica que hemos aprendido en la sección anterior, es decir, la aplicación de símbolos de sincronismo, completada con trampas para excepciones y retraso en el tratamiento de errores.

Para implementar completamente la estrategia, vamos a seguir los siguientes pasos:

- Convertir la gramática en una k=1, con ayuda de los predicados sintácticos
- Añadir la infraestructura para manejar símbolos de sincronismo
- Colocar trampas de excepciones
- Retardar el tratamiento de errores en algunas reglas

Para poder entender completamente lo que vamos a discutir en este apartado, es muy importante haber comprendido la estrategia de recuperación que hemos explicado en la sección anterior. Haré contínuas referencias a ella a lo largo de este apartado. Los que no estén seguros de comprenderla, deberían repasarla.

Haciendo k=1

Ya hemos establecido que para que la técnica pueda ejecutarse idóneamente debemos utilizar una gramática con k=1. Así que vamos a modificar la opción correspondiente:

```
class LeLiErrorRecoveryParser extends LeLiParser;
options
{
   importVocab=LeLiParserVocab;
   k=1;
}
```

Si intentamos generar el analizador en el estado actual, ANTLR generará 4 mensajes de advertencia en 5 reglas: `declaracion`, `tipoRetorno`, `raizAcceso`, `subAcceso`, y

`alternativasSi`. Por cierto, los errores se detectarán en el fichero `extendedLeLiErrorRecoveryParser.g`, que ANTLR habrá generado en nuestro directorio de trabajo[53].

Habrá que modificar convenientemente las 5 reglas para conseguir una gramática con k=1. Por supuesto, tendremos que utilizar predicados sintácticos. No es muy difícil, se trata de detectar las alternativas que comienzan por el mismo token, y añadir un predicado sintáctico en la primera alternativa para eliminar la semejanza.

A continuación mostraré la nueva implementación de las reglas. Los predicados sintácticos aparecen señalados.

```
class LeLiErrorRecoveryParser extends Parser;
options {...} // importVocab, k=1

// ---------------------- HACER k=1 -----------------------------
declaracion ! // desactivar construcción por defecto
[AST r, AST t, boolean inicializacion] // parámetros
{
   AST raiz = astFactory.dupTree(r);     // copia del árbol
   raiz.addChild(astFactory.dupTree(t)); // copia del árbol
}
   : { inicializacion }? // pred. semántico
      (IDENT OP_ASIG) => i1:IDENT OP_ASIG valor:expresion
      {
         raiz.addChild(#i1);
         raiz.addChild(#valor);
         ## = raiz;
      }
   | { inicializacion }?
      (IDENT PARENT_AB) =>
      i2:IDENT parentAb le:listaExpresiones parentCe
      {
         raiz.addChild(#i2);
         raiz.addChild(#le);
         ## = raiz;
      }
   | i3:IDENT
      {
         raiz.addChild(#i3);
         ## = raiz;
      }
   ;

/** Regla auxiliar que codifica el tipo de retorno de un método */
protected tipoRetorno
   : (tipo IDENT) => tipo
   | /* nada */ {## = #[TIPO_VACIO,"TIPO_VACIO"]; }
     ;

/**
 * Raíz de los accesos que no son llamadas a un método de la
 * clase "actual"
 */
raizAcceso : ((IDENT|RES_CONSTRUCTOR) PARENT_AB)=>llamada
```

53 Esto no es en absoluto cierto. ANTLR no permite declarar un analizador sin al menos una regla de reconocimiento, por lo que extendedLeLiErrorRecoveryParser no se generará. Para ello hay que escribir una regla estúpida, como a: IDENT;

```
            | IDENT
            | literal
            | conversion
            | PARENT_AB! expresion parentCe
            ;

/** Regla que reconoce los accesos a atributos y métodos de un objeto. */
subAcceso
   : ((IDENT|RES_CONSTRUCTOR) PARENT_AB)=>llamada
   | IDENT
   | RES_SUPER
   ;

/**
 * Auxiliar (reconoce las alternativas de la instrucción "si"
 * sin warnings de ambigüedad)
 */
protected alternativasSi
   : (BARRA_VERT PARENT_AB)=>altSiNormal alternativasSi
   | altSiOtras
   | /* nada */
   ;
```

Como puede verse no es muy complicado. Lo único que habría que resaltar es el orden de los predicados cuando hay predicados sintácticos y semánticos en la misma alternativa. Como puede verse en la regla `declaracion`, primero se escriben los predicados semánticos y luego los sintácticos.

Infraestructura para los tokens de sincronismo

Lo primero que hará falta será incluir el método sincronizar y el conjunto simbolosSincro en el analizador. El conjunto de símbolos de sincronismo que -inicialmente- vamos a utilizar será

```
{LLAVE_AB, LLAVE_CE, PARENT_AB, PARENT_CE, COMA, PUNTO_COMA, PUNTO,
BARRA_VERT, RES_CLASE, RES_METODO, RES_ATRIBUTO, RES_DESDE, RES_HACER,
RES_MIENTRAS, RES_SI, RES_VOLVER}
```

Es decir, vamos a utilizar como tokens de sincronismo todos los “símbolos de puntuación” (separadores que no intervienen en la creación de expresiones) además de las palabras reservadas que sirven para reconocer el principio de una estructura recognitiva, como una instrucción o una declaración de atributo.

La implementación, una vez establecido el conjunto de tokens de sincronismo, es casi inmediata:

```
class LeLiErrorRecoveryParser extends LeLiParser;
options {...}
{
   /** Símbolos de sincronismo **/
   public static final BitSet simbolosSincro = mk_simbolosSincro();

   private static final BitSet mk_simbolosSincro()
   {
      BitSet b = new BitSet();
      b.add(LLAVE_AB);
      b.add(LLAVE_CE);
      b.add(PARENT_AB);
      b.add(PARENT_CE);
      b.add(COMA);
```

```
      b.add(PUNTO_COMA);
      b.add(PUNTO);
      b.add(BARRA_VERT);

      b.add(RES_CLASE);
      b.add(RES_METODO);
      b.add(RES_ATRIBUTO);
      b.add(RES_DESDE);
      b.add(RES_HACER);
      b.add(RES_MIENTRAS);
      b.add(RES_SI);
      b.add(RES_VOLVER);

      b.add(IDENT); // para método y parámetro
      return b;
   }

   /** Función de sincronización con símbolos de sincronismo **/
   public void sincronizar ( RecognitionException re,
                             BitSet SIGUIENTE )
   throws TokenStreamException, RecognitionException
   {
      ... // (Consultar implementación en la sección anterior)
   }
}
```

Exceptuando las reglas en las que necesitemos un tratamiento de erroes diferente (como en las trampas para excepciones), el método `sincronizar` será el que utilicemos como manejador de excepciones en todas las reglas que modifiquemos en la recuperación de errores. Las reglas que no toquemos seguirán utilizando el manejador que ANTLR proporciona por defecto.

6.6.3: Colocando las trampas para excepciones

Colocar trampas para excepciones no consiste más que en elegir una serie de tokens adecuados y sustituir sus referencias por referencias a una regla que los reconoce de una manera especial. Es decir, si `LLAVE_AB` es un token candidato a ser trampa, hay que sustituirlo por la regla `llaveAb` en toda la gramática (la utilidad de buscar-y-reemplazar de su editor de textos habitual le permitirá salir airoso de este aprieto).

Vamos a colocar trampas en los siguientes tokens:

```
{ LLAVE_AB,LLAVE_CE,PARENT_AB,PARENT_CE,PUNTO_COMA,PUNTO,COMA,BARRA_VERT }
```

Las trampas para excepciones que vamos a construir tienen todas la misma estrategia de recuperación, que hemos encapsulado dentro del método `errorFaltaToken`. Las trampas tendrán el siguiente aspecto:

```
/** Regla auxiliar para reconocer paréntesis necesarios **/
parentAb : PARENT_AB! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(PARENT_AB); }

/** Regla auxiliar para reconocer paréntesis necesarios **/
parentCe : PARENT_CE! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(PARENT_CE); }

/** Regla auxiliar para reconocer llaves necesarias **/
```

```
llaveAb : LLAVE_AB! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(LLAVE_AB); }

/** Regla auxiliar para reconocer llaves necesarias **/
llaveCe : LLAVE_CE! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(LLAVE_CE); }

/** Regla auxiliar para reconocer comas necesarias **/
coma : COMA! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(COMA); }

/** Regla auxiliar para reconocer barras verticales necesarias **/
barraVert : BARRA_VERT! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(BARRA_VERT); }

/** Regla auxiliar para reconocer puntos y comas necesarios **/
puntoComa : PUNTO_COMA! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(PUNTO_COMA); }

/** Regla auxiliar para reconocer puntos **/
punto : PUNTO! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(PUNTO); }
```

Vamos a ver la estructura del método `errorFaltaToken`, que como es habitual habrá que incluir en la sección de código java de `LeLiErrorRecoveryParser.g`:

```
public void errorFaltaToken( int token )
   throws TokenStreamException
   {
      Token t0 = LT(0);
      Token t1 = LT(1);

      if(t1.getType() != Token.EOF_TYPE)
      {
         String t1Text;
         if(t1.getType()==IDENT)
            t1Text = "el identificador '" + t1.getText() +"'";
         else
            t1Text = getTokenName(t1.getType());

         reportError( "Se esperaba " + getTokenName(token) +
                      ", se encontró " + t1Text,
                      this.getFilename(),
                      t1.getLine(),
                      t1.getColumn() - t1.getText().length() );
      }
      else
      {
         reportError ( "Se esperaba " + getTokenName(token) +
                       " cuando se alcanzó el final de fichero",
                       this.getFileName(),
                       t0.getLine(),
```

```
                    t0.getColumn() ) ;
        }
    }
```

El único parámetro del método es un entero que representa el tipo del token que falta (`IDENT`, `PUNTO_COMA`, etc). La mayor parte del código del método se dedica a emitir un mensaje de error lo más específico posible. Para crear dicho mensaje se sigue el siguiente algoritmo:

- Si se ha llegado al final del fichero, se escribe "se esperaba <token> cuando se alcanzó el final del fichero"
- Si no, hay un token en LT(1). El mensaje será "se esperaba <token>, pero se encontró un token <LT(1)>". Si el token en LT(1) es un identificador, se detallará un poco más, indicando que se trata de un identificador con el texto "xxx".

Llegados a este punto tenemos que incrustar las trampas en nuestras reglas.

- Bien, ahora solamente queda sobreescribir en `LeLiErrorRecoveryParser` todas las reglas de `LeLiParser` que usen uno de estos tokens para que utilicen una trampa para reglas - dice nuestro proverbial programador novato. Vamos a responderle.

- Esa solución no es adecuada. Dado que la mayoría de las reglas de `LeLiParser` utiliza algún token con trampa, vamos a tener que reescribir la mayoría de las reglas en `LeLiErrorRecoveryParser`.

- ¿Qué hacemos entonces? ¡No me dirás a estas alturas que vamos a modificar `LeLiParser` directamente! Con toda la lata que me has dado con utilizar la herencia de gramáticas para no modificar el fichero original...

- El propósito de utilizar la herencia no era ni mucho menos "mantener intacto LeLiParser.g", sino uno más profundo: facilitarnos la vida al programar, incrementando la flexibilidad sin perjudicar la mantenibilidad del sistema. Puede parecer que esto se puede conseguir sin tocar la gramática base, pero eso no es cierto siempre.

- A ver si lo he entendido: es como con la orientación a objetos; a veces al desarrollar una subclase nos damos cuenta de que la clase base necesita algún retoque.

- ¡Exactamente! Yo no lo habría explicado mejor.

- Bueno, entonces, ¿qué hacemos?

- Vamos a dejar la gramática base "preparada" para aceptar trampas para excepciones, aunque no las implemente. Para ello, definiremos las siguientes reglas *en LeLiParser.g*:

```
parentAb : PARENT_AB! ;
parentCe : PARENT_CE! ;
llaveAb : LLAVE_AB! ;
llaveCe : LLAVE_CE! ;
coma : COMA! ;
barraVert : BARRA_VERT! ;
puntoComa : PUNTO_COMA! ;
punto : PUNTO! ;
```

- ¡Oh!

- Además realizaremos las sustituciones (`PARENT_AB` por `parentAb`, etc) también en `LeLiParser.g` en lugar de en `LeLiErrorRecoveryParser.g`.

- ¡Claro! ¡Así, solamente hay que reescribir las reglas-trampa!

- Efectivamente. `LeLiParser` seguirá teniendo una gramática legible e independiente del

tratamiento de errores. Y el fichero `LeLiErrorRecoveryParser.g` será mucho más claro.

- Pero hay un inconveniente: ahora `LeLiParser` será más lento; se llaman a más métodos y hay más bloques `try/catch` para reconocer las mismas entradas que antes.

- La flexibilidad suele cobrarse un pequeño peaje de velocidad. Pero bien observado. Ahora, si no te importa, pasemos al siguiente punto.

6.6.4: Retardando el tratamiento de errores

Existen varias reglas en las que encontraremos el problema del lanzamiento indeseado de una `NoViableAltException`. Voy a capturar las excepciones en lugar de desactivarlas (prefiero enfrentarme a los errores en lugar de ingnorarlos; al menos programando).

En nuestra gramática estas reglas son las que pueden reconocer "nada", a excepción del cierre de Kleene. Por lo tanto estamos hablando de reglas o sub reglas que tengan alternativas (A|B|C) siendo alguna de ellas vacía (A|B|) o de reglas o sub reglas opcionales ((A)?).

Es decir, vamos a tener que capturar excepciones en `expAsignacion`, `expPostIncremento`, `clausulaExtiende` y `listaDecParams`. Comencemos con las dos primeras.

`expAsignacion` y `expPostIncremento` lanzan `NoViableAltException` indeseablemente en sub reglas. Vamos a tener que dividirlas en reglas-sub reglas para poder capturar la excepción.

Véase la sección anterior para una explicación más detallada sobre el retardo del tratamiento de errores, `expPostIncremento` y `expAsignacion`.

```
/** Asignaciones (nivel 9) */
expAsignacion : izq:expOLogico der:expAsignacion2
                { ## = construyeExpBin(#izq, #der); }
              ;

expAsignacion2 : ( OP_ASIG^ expOLogico )? ;
exception catch [NoViableAltException nvae]
{ sincronizar(nvae, $FOLLOW); }

/** Postincremento y postdecremento (nivel 2) */
expPostIncremento :  operando:expNegacion operador:expPostIncremento2
                     { ## = construyeExpUn(#operando, #operador); }
                  ;

expPostIncremento2 : ( OP_MASMAS^|OP_MENOSMENOS^ )?
                   ;
exception catch [NoViableAltException nvae]
{ sincronizar(nvae, $FOLLOW); }
```

Dado que las reglas-sub reglas contenían a la raíz de las reglas originales[54], nos hemos tenido que servir de los métodos `construyeExpUn` y `construyeExpBin`, que también habrá que especificar en la sección de código del analizador.

```
   public AST construyeExpUn(AST operando, AST operador)
   {
      if(null==operador) return operando;
      if(operando.getNextSibling()==operador)
      {
         operando.setNextSibling(null);
```

54 Si esto le suena a chino, consulte la sección anterior con detenimiento.

```
      }
      operador.setFirstChild(operando);
      return operador;
   }

   public AST construyeExpBin ( AST izquierda, AST derecha )
   {
      if(derecha != null)
      {
         // "desconectar" los nodos si es necesario
         if(izquierda.getNextSibling()==derecha)
         {
            izquierda.setNextSibling(null);
         }

         AST valor = derecha.getFirstChild();
         return #(derecha, izquierda, valor);
      }
      return izquierda;
   }
```

En `clausulaExtiende` simplemente utilizaremos un sincronizador.

```
clausulaExtiende : RES_EXTIENDE^ IDENT
                 | /*nada*/
                   { ## = #( #[RES_EXTIENDE,"extiende"],
                             #[IDENT, "Objeto"] ); }
                 ;
exception catch [NoViableAltException nvae]
{
   sincronizar(nvae, $FOLLOW);
   ## = #( #[RES_EXTIENDE,"extiende"], #[IDENT, "Objeto"] );
}
```

Nótese que incluso en el caso de que se lance una excepción seremos capaces de suministrar un AST para la cláusula extiende.

Hemos dejado lo mejor para el final. Recuperación de errores en una lista de declaraciones de parámetros.

La lista de declaraciones de parámetros es, seguramente, una de las estructuras de LeLi más complicadas de tratar en recuperación de errores. Lo que ocurre es que es una estructura realmente complicada: los parámetros han de estar separados por puntos y comas, pero cuando son del mismo tipo se separan por comas y solamente se necesita escribir el tipo una vez. Las dos formas de declaración (con comas y con puntos y comas) pueden coexistir en la misma de declaraciones. Finalmente, la propia lista puede ser completamente vacía, y ya sabemos que las opciones vacías no se prestan facilmente a la recuperación.

Se trata, en definitiva, de una estructura bastante compleja, que requiere un tratamiento de errores un poco más refinado.

Empecemos con lo obvio: un error en el reconocimiento de la lista de declaraciones de parámetros de un método no debe, bajo ningún concepto, anular el reconocimiento del método al que pertenece (provocando una excepción y cancelando decMetodo).

Por otro lado, vamos a intentar no “cancelar antes de tiempo” el análisis. Es decir, si encontramos un error en un parámetro, pero los demás están bien definidos, éstos deben ser correctamente analizados y su información incluida en el AST de la regla.

Para evitar que `listaDecParametros` pueda anular a `decMetodo` lo más sencillo es utilizar un sincronizador: colocamos el manejador de excepciones al final de la regla, y así aseguramos que ninguna excepción la abandone hacia el nivel superior de la pila de llamadas.

Queda por resolver el segundo problema: cuando ocurre un error que se recupera con un manejador de excepciones global, se cancela la construcción de AST, lo que en nuestro caso significa perder los ASTs ya generados. Para conservar la construcción del AST vamos a utilizar una estrategia de "salvaguarda": cada vez que declaremos una lista de parámetros del mismo tipo, modificaremos el AST de la regla (en lugar de hacerlo solamente al final).

El código para `listaDecParametros` será, por lo tanto, el siguiente:

```
listaDecParams
{ final AST raiz = #[RES_PARAMETRO,"parámetro"] ; }
   :
   { ## = #[LISTA_DEC_PARAMS,"parámetros"]; }
   ( l1:listaDeclaraciones[raiz, false]!
     { ##.addChild(#l1); }
     ( puntoComa ln:listaDeclaraciones[raiz, false]!
       { ##.addChild(#ln); }
     )*
   )? // opcional
     ;
     // Capturar NoViableAltException y resincronizar
   exception catch [NoViableAltException nvae]
   { sincronizar(nvae, $FOLLOW); }
```

Nótense los dos operadores de filtrado que he señalado en el código.

6.6.5: Errores frecuentes – acentuación

El problema

Ya hemos tratado el tipo más frecuente de error sintáctico que apareciá en el ejemplo. Me refiero al tipo "se esperaba el token x" o "se esperaba un token dentro del grupo (x,y,z)". Estos errores son provocados por excepciones. Sus mensajes tienen una parte fija ("se esperaba el token " o "se esperaba un token dentro del grupo ") y una parte variable, que se configura mediante parámetros de las excepciones ("x" y "x,y,z").

El problema es que en ocasiones los mensajes proporcionados por el compilador no son suficientes. Considérese el siguiente ejemplo:

```
clase Persona
{
   atributo Entero edad;
   metodo erroresLexicos(Entero edad)
   {
      atributo.edad = parametro.edad;
   }
```

Si se compila el ejemplo anterior, se obtendrán dos errores de reconocimiento parecidos a éstos:

```
Persola.leli:4:1: Se esperaba un token en el grupo (método, atributo)
Persona.leli:6:19: El identificador parametro no ha sido declarado
```

El segundo error es semántico y no aparecerá hasta que no hayamos implementado el nivel semántico en el capítulo siguiente, pero eso es lo de menos.

¿Qué es lo que ha ocurrido? En la línea 4 el programador ha escrito "metodo" en lugar de "método" y en la línea 6 ha escrito "parametro" en lugar de "parámetro". Es decir, en los dos casos se ha olvidado de incluir los acentos de las palabras reservadas.

¡Este es, sin duda, un error muy frecuente! Puede pasar un buen rato antes que el programador se dé cuenta de lo que ocurre. Los mensajes de error que se muestran no son adecuados.

Lo ideal sería que, antes incluso de llegar al nivel semántico, es decir en el analizador sintáctico, se detectaran estas omisiones de los acentos y se mostraran mensajes más descriptivos:

```
Persona.leli:4:1: Omisión de acento; escriba "método" en lugar de "metodo"
Persona.leli:4:1: Omisión de acento; escriba "parámetro" en lugar de
"parametro"
```

El método

Empezaremos con un método que encapsule toda la problemática relacionada con los mensajes de omisión de tilde. Llamaremos a dicho método `comprobarErrorTilde`.

```
public boolean comprobarErrorTilde ( Token t, AST ast,
                                     String textoIncorrecto, int tipoCorrecto,
                                     String textoCorrecto )
{
   if( t.getText().equals(textoIncorrecto) )
   {
      reportError( "Omisión de tilde. " +
                  "Escriba '" + textoCorrecto +
                   "' en lugar de '" + t.getText() + "'",
                   this.getFilename(),
                   t.getLine(),
                   t.getColumn() );
      ast.setType( tipoCorrecto );
      ast.setText( textoCorrecto );
      return false;
   }

   return true;
}
```

Este método toma como parámetros el token problemático, su AST, el texto sin acento, el tipo que debería tener el token y el texto que debería tener, y genera un mensaje adecuado, además de modificar el ast si es necesario. El método devuelve "true" si no había errores y "false" si ha tenido que cambiar algo.

Las estrategias

Tenemos que ingeniarnoslas para utilizar este método cuando se lea "parametro" o "metodo" en lugar de "parámetro" y "método".

La forma más sencilla de reconocer los errores de acentuación (y, en general, cualquier error de escritura de una palabra reservada del lenguaje) es reservar también las cadenas erróneas como si fueran palabras reservadas. Es decir, añadir dos tokens al analizador léxico, algo como `RES_PARAMETRO_ERROR` y `RES_METODO_ERROR`. Sin embargo esta forma de proceder impedirá al programador utilizar "parametro" y "metodo" como identificadores en su programa.

Aunque puede que no sea tan mala idea impedirle utilizar estos tokens, esta estrategia no es extensible. ¿Qué ocurre si muchas de las palabras reservadas tienen posibilidad de escribirse mal?

Utilizando esta estrategia se impide al programador utilizar identificadores que debería poder utilizar.

La segunda estrategia no necesita reservar los identificadores, pero es un poco más complicada de implementar. Para ello, lo que haremos será tratarlos como identificadores; identificadores que pueden sustituir a las palabras reservadas en determinadas ocasiones. Esta será la estrategia que utilizaremos.

Modificaciones en las reglas

Empecemos con `parámetro`, `RES_PARAMETRO` para los amigos. `RES_PARAMETRO` se utiliza en LeLi para solucionar el problema de diferenciación entre parámetros, variables y atributos de una clase cuando todos tienen el mismo nombre. Se utiliza anteponiéndola al identificador que vamos a utilizar con un punto. Se reconoce en la regla `raizAcceso`, que ya hemos redefinido en el analizador para hacer k=1.

`raizAcceso` es una regla-alternativa. Se da la feliz casualidad de que una de sus opciones es `IDENT`, de manera que lo único que vamos a tener que hacer es añadir una pequeña acción que emita un mensaje y modifique el árbol si el texto del identificador es "parametro". Todo eso ya se hace dentro de `comprobarErrorTilde`. Así, el código de `raizAcceso` quedará como sigue:

```
raizAcceso : ((IDENT|RES_CONSTRUCTOR) PARENT_AB)=>llamada
           | i:IDENT
             { comprobarErrorTilde( i, #i, "parametro",
                                    RES_PARAMETRO, "parámetro" ); }
           | literal
           | conversion
           | PARENT_AB! expresion parentCe
           ;
exception catch [NoViableAltException nvae]
{ sincronizar(nvae, $FOLLOW); }
```

El caso de "método" y "metodo" es un poco más complicado. Hay dos problemas: el token conflictivo debe usarse como raíz en un AST, y además no hay ninguna "alternativa para `IDENT`" ya hecha.

La palabra reservada `RES_METODO` solamente se utiliza para definir un método, en la regla `decMetodo`. Lo primero que he hecho ha sido sustituir en `decMetodo` la referencia directa a `RES_METODO` por una llamada a una regla auxiliar, de nombre `resMetodo`. Además he añadido una acción para "reenraizar" el AST.

```
decMetodo
   : r:resMetodo! (RES_ABSTRACTO)? tipoRetorno IDENT
     parentAb listaDecParams parentCe
     llaveAb listaInstrucciones llaveCe
     { ## = #(#r,##); }
   ;
```

Nótese que el AST que construye `resMetodo` solamente se incluye en el AST final en la acción, porque por defecto no se incluye (al estar sucecido del sufijo '!')[55].

La estructura de la regla que nos queda por examinar, `resMetodo`, será la siguiente:

55 Podemos hacer esto porque sabemos que #r no tiene hijos; si los tuviera y quisiéramos conservarlos, deberíamos escribir r.addChild(##); ##=#r;

```
resMetodo : RES_METODO
          | i:IDENT
            {
             if( comprobarErrorTilde( i, #i, "metodo",
                                      RES_METODO, "método") )
                {
                  throw new MismatchedTokenException( tokenNames,
                                                      i,
                                                      RES_METODO,
                                                      false,
                                                      getFilename() );
                }
            }
          ;
exception catch [NoViableAltException nvae]
{ sincronizar(nvae, $FOLLOW); }
```

En este caso vemos que se admiten dos alternativas: la palabra reservada `RES_METODO` o un `IDENT`. En este último caso hay un error. Queda por saber cuál es; en realidad es sencillo: si el token tiene como texto "metodo" es un error de acentuación, y en cualquier otro caso se trata de un token no válido.

Llamando a `comprobarErrorTilde` lanzaremos un error si y solo si el texto del token reconocido es "metodo". En cualquier otro caso, lanzaremos un error de token inesperado, idéntico al que se lanzaría si la regla `resMetodo` solamente fuera capaz de reconocer `RES_METODO`.

Vuelta a los tokens de sincronismo

La recuperación de errores ya es muy buena. Casi todos los casos están tratados. Voy a proponer aquí una pequeñísima mejora.

En el estado actual, ANTLR utiliza `RES_METODO` y `RES_PARAMETRO` como tokens de sincronismo. ¿No estaría bien utilizar "metodo" y "parametro" también como tokens de sincronismo?

¡Ah! Volvemos al problema que teníamos para tratar los errores de escritura de palabras reservadas. Lo más sencillo sería reconocer simplemente las dos cadenas de texto con dos tokens determinados (a los que hemos dado incluso nombre, `RES_METODO_ERROR` y `RES_PARAMETRO_ERROR`) y añadir dichos tokens a la lista de tokens de sincronismo, es decir, al BitSet estático `simbolosSincro`. El problema es, como ya hemos dicho, que de esta forma estamos impidiendo al programador utilizar "metodo" y "parametro" como nombres para sus métodos, parámetros, atributos, variables o clases.

Tendremos que hilar un poco más fino. Lo que vamos a hacer es modificar el algoritmo de sincronización, implementado en el método `sincronizar`, para que utilice *solamente algunos IDENTS* como símbolos de sincronismo, pero no todos.

Lo primero que vamos a hacer es añadir `IDENT` al conjunto de tokens de sincronismo:

```
public static final BitSet simbolosSincro = mk_simbolosSincro();

   private static final BitSet mk_simbolosSincro()
   {
      BitSet b = new BitSet();
      b.add(LLAVE_AB);
      ...

      b.add(IDENT);
      return b;
   }
```

Ahora vamos a añadir un atributo estático que almacene todos los identificadores que deseemos que participen en el sincronismo (en nuestro caso solamente dos). Vamos a utilizar una instancia de la clase `java.util.HashSet`.

```
   public static final HashSet identsSincro = mk_identsSincro();

   public static final HashSet mk_identsSincro()
   {
      HashSet hs = new HashSet();
      hs.add("metodo");
      hs.add("parametro");
      return hs;
   }
```

Para poder utilizar `java.util.HashSet`, vamos a necesitar importarlo en el header del analizador:

```
header {

package leli;

import antlraux.Logger;
import java.util.HashSet;
}
```

Finalmente hemos de modificar el método sincronizar para que utilice `identsSincro`:

```
   public void sincronizar ( RecognitionException re,
                             BitSet SIGUIENTE )
   throws RecognitionException, TokenStreamException
   {
      // Si es SS, "dejar pasar" (terminar la regla)
      if( ! simbolosSincro.member(LA(0)) )
      {
         // Si estamos guessing, lanzamos directamente
         if (inputState.guessing!=0) throw re;

         BitSet auxiliar = simbolosSincro.or(SIGUIENTE);

         // Mostrar el error y consumir, pero utilizando
         // auxiliar en lugar de SIGUIENTE
         // Y TENIENDO EN CUENTA IDENTIFICADORES ESPECIALES
         reportError(re);
         boolean bSincronizado = false;
         Token t = null;
         do {
```

```
            consume();
            consumeUntil(auxiliar);
            t = LT(0);
            if( t.getType()==IDENT &&                    // si es un IDENT y
                !identsSincro.contains(t.getText()) ) // NO es de
identsSincro
            { bSincronizado = false; }
            else
            { bSincronizado = true; }
          }while (!bSincronizado);
        }
    }
```

He resaltado los cambios más importantes. Ahora es posible que tanto `consume()` como `consumeUntil()` sean llamados más de una vez, por estar dentro de un bucle.

La condición de salida de dicho bucle es la siguiente:

- Se ha encontrado un token de sincronismo que no es un `IDENT`
- Se ha encontrado un `IDENT` perteneciente al conjunto identsSincro.

Cualquier token que no cumpla una de estas dos condiciones será "absorbido".

6.6.6: El resultado

Si lanzamos nuestro nuevo y flamante analizador a reconocer el fichero de ejemplo obtendremos unos resultados verdaderamente satisfactorios: comprobaremos que el analizador detecta correctamente los errores, recuperándose como es debido. La salida que deberíamos obtener sería parecida a la siguiente:

```
err.leli:5:11: Se esperaba una llave abierta ('{'), se encontró el
identificador 'metodo'
err.leli:5:17: Omisión de tilde. Escriba 'método' en lugar de 'metodo'
err.leli:6:16: Se esperaba un paréntesis cerrado (')'), se encontró una llave
abierta ('{')
err.leli:8:16: Se esperaba un punto y coma (';'), se encontró una llave cerrada
('}')
err.leli:12:16: Se esperaba un paréntesis cerrado (')'), se encontró el
identificador 'parametro'
err.leli:12:16: Se esperaba una llave abierta ('{'), se encontró el
identificador 'parametro'
err.leli:12:25: Omisión de tilde. Escriba 'parámetro' en lugar de 'parametro'
err.leli:14:16: Se esperaba un punto y coma (';'), se encontró una llave
cerrada ('}')
errores: 8
```

El AST también se habrá recuperado. No voy a mostrarlo completamente desplegado, porque o bien ocuparía más de una página o bien no podrían leerse todos los nombres; habrá que fiarse de mi palabra: los errores se han recuperado satisfactoriamente.

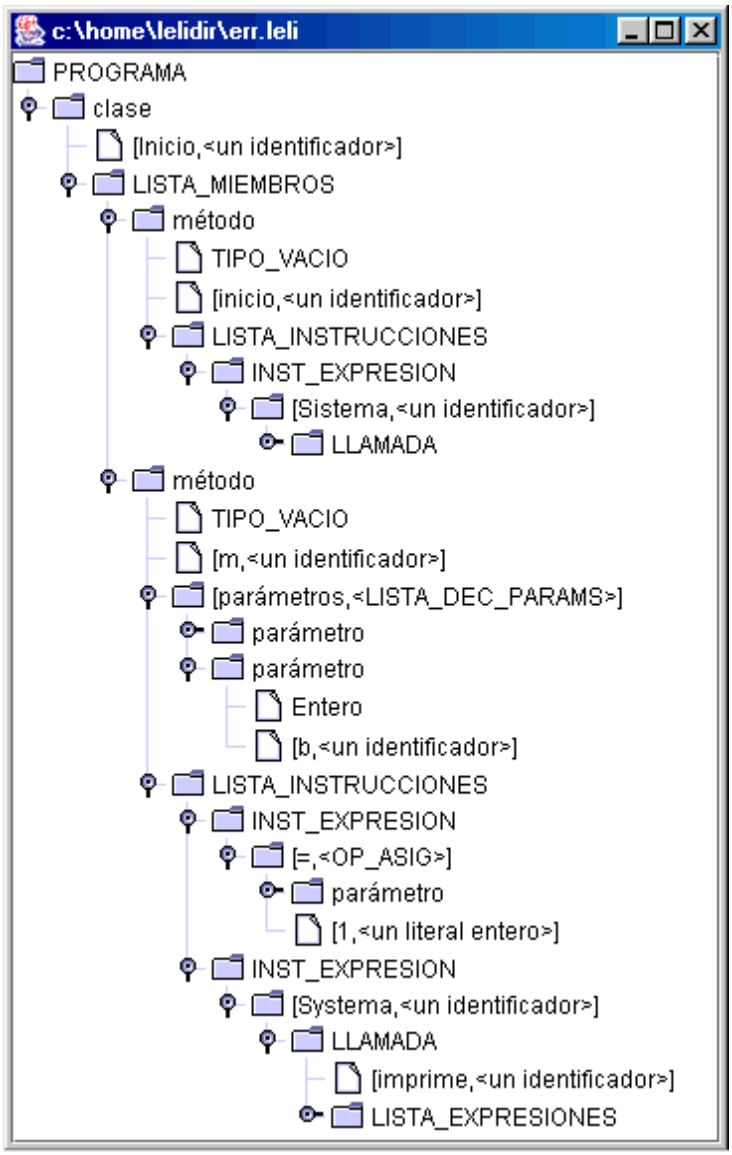

Ilustración 6.5 AST con la recuperación de errores funcionando

Sección 6.7: Código

6.7.1: Fichero LeLiLexer.g

Las modificaciones realizadas a este fichero en este capítulo han consistido mayormente en habilitar los alias para tokens, para generar mensajes de error más inteligibles.

```
header{
package leli;

/*-----------------------------------------------*\
 | Un intérprete para un Lenguaje Limitado(LeLi) |
 | _____________________________________________ |
 |                                               |
 |              ANALISIS LÉXICO                  |
 | _____________________________________________ |
 |                                               |
 |            Enrique J. Garcia Cota             |
\*-----------------------------------------------*/

}

/**
 *  El objeto que permite todo el analisis léxico.
 *  notas: comentar todo esto al final del analex.
 */
class LeLiLexer extends Lexer;

options{
   charVocabulary = '\3'..'\377';  // Unicodes usuales
   exportVocab=LeLiLexerVocab;     // Comentar al hacer el parser
   testLiterals=false;             // comprobar literales solamente cuando se
diga exp
   k=2;                            // lookahead
}

tokens
{
// Tipos basicos
   TIPO_ENTERO   = "Entero"   ;
   TIPO_REAL     = "Real"     ;
   TIPO_BOOLEANO = "Booleano" ;
   TIPO_CADENA   = "Cadena"   ;

// Literales booleanos
   LIT_CIERTO = "cierto" ; LIT_FALSO = "falso" ;

// Literales cadena
   LIT_NL = "nl"; LIT_TAB = "tab" ; LIT_COM = "com";

// Palabras reservadas
   RES_CLASE       = "clase"       ;
   RES_EXTIENDE    = "extiende"    ;
   RES_METODO      = "método"      ;
   RES_CONSTRUCTOR = "constructor" ;
   RES_ATRIBUTO    = "atributo"    ;
   RES_ABSTRACTO   = "abstracto"   ;
```

```
   RES_PARAMETRO   = "parámetro"   ;
   RES_CONVERTIR   = "convertir"   ;
   RES_MIENTRAS    = "mientras"    ;
   RES_HACER       = "hacer"       ;
   RES_DESDE       = "desde"       ;
   RES_SI          = "si"          ;
   RES_OTRAS       = "otras"       ;
   RES_SALIR       = "volver"      ;
   RES_ESUN        = "esUn"        ;
   RES_SUPER       = "super"       ;

// Operadores que empiezan con letras;
   OP_Y            = "y"  ;
   OP_O            = "o"  ;
   OP_NO           = "no" ;

// Los literales real y entero son "devueltos" en las
// acciones del token "privado" LIT_NUMERO
// LIT_REAL < paraphrase="un literal real" >;
// LIT_ENTERO < paraphrase="un literal entero" >;

}

{ // Comienza la zona de código
   protected Token makeToken(int type)
   {
      // Usamos la implementación de la superclase...
      Token result = super.makeToken(type);
      // ...y añadimos información del nombre de fichero
      result.setFilename(inputState.filename);
      // y devolvemos el token
      return result;
   }
}

/**
 * Los tres tipos de retorno de carro.
 */
protected NL :
   (
     ("\r\n") => "\r\n"  // MS-DOS
   | '\r'                // MACINTOSH
   | '\n'                // UNIX
   )
   { newline(); }
   ;

/**
 * Esta regla permite ignorar los blancos.
 */
protected BLANCO :
   ( ' '
   | '\t'
   | NL
   ) { $setType(Token.SKIP); } // La accion del blanco: ignorar
   ;
/**
 *  Letras españolas.
```

```
 */
protected LETRA
   : 'a'..'z'
   | 'A'..'Z'
   | 'ñ' | 'Ñ'
   | 'á' | 'é' | 'í' | 'ó' | 'ú'
   | 'Á' | 'É' | 'Í' | 'Ó' | 'Ú'
   | 'ü' | 'Ü'
   ;
/**
 * Dígitos usuales
 */
protected DIGITO : '0'..'9';

/**
 * Regla que permite reconocer los literales
 * (y palabras reservadas).
 */
IDENT
options
{
   testLiterals=true; // Comprobar palabras reservadas
   paraphrase="un identificador";
}
:
   (LETRA|'_') (LETRA|DIGITO|'_')*
;

// Separadores
PUNTO_COMA
options { paraphrase="un punto y coma (';')"; }
   : ';'
   ;
COMA
options { paraphrase="una coma (',')"; }
   : ','
   ;
LLAVE_AB
options { paraphrase="una llave abierta ('{')"; }
   : '{'
   ;
LLAVE_CE
options { paraphrase="una llave cerrada ('}')"; }
    : '}'
    ;
PUNTO
options { paraphrase="un punto ('.')"; }
   : '.'
   ;
PARENT_AB
options { paraphrase="un paréntesis abierto ('(')"; }
   : '('
   ;
PARENT_CE
options { paraphrase="un paréntesis cerrado (')')"; }
   : ')'
   ;
BARRA_VERT
```

```
options { paraphrase="una barra vertical ('|')"; }
   : '&'
   ;

//  operadores
OP_IGUAL       : "==" ;
OP_DISTINTO    : "!=" ;
OP_ASIG        : '='  ;
OP_MENOR       : '<'  ;
OP_MAYOR       : '>'  ;
OP_MENOR_IGUAL : "<=" ;
OP_MAYOR_IGUAL : ">=" ;
OP_MAS         : '+'  ;
OP_MENOS       : '-'  ;
OP_MASMAS      : "++" ;
OP_MENOSMENOS  : "--" ;
OP_PRODUCTO    : '*'  ;
OP_DIVISION    : '/'  ;


/**
 * Permite reconocer literales enteros y reales sin generar conflictos.
 */
LIT_NUMERO
   : ( ( DIGITO )+  '.' ) =>
        ( DIGITO )+ '.' ( DIGITO )*
        { $setType(LIT_REAL); }
    | ( DIGITO )+
      { $setType(LIT_ENTERO); }
    ;

LIT_ENTERO
options { paraphrase = "un literal entero"; }
   : '@' { throw new MismatchedCharException(); }
   ;

LIT_REAL
options { paraphrase = "un literal real"; }
   : '#' { throw new MismatchedCharException(); }
   ;

/**
 * Comentario de una sola línea
 */
protected COMENTARIO1
   :  "//" (~('\n'|'\r'))*
      { $setType(Token.SKIP); }
   ;

/** Comentario de varias líneas */
protected COMENTARIO2  :
   "/*"
   ( ('*' NL) => '*' NL
   | ('*' ~('/'|'\n'|'\r')) => '*' ~('/'|'\n'|'\r')
   | NL
   | ~( '\n' | '\r' | '*' )
   )*
   "*/"
```

```
   { $setType(Token.SKIP); }
   ;

/** Literales cadena */
LIT_CADENA
options { paraphrase="un literal cadena"; }
   : '"' !
     ( ~('"'|'\n'|'\r') )*
     '"' !
   ;
```

6.7.2: Fichero LeLiParser.g

El único cambio que se ha efectuado sobre este fichero ha sido prepararlo para aceptar fácilmente trampas para excepciones, transformando las referencias a tokens-trampa en referencias a reglas (`LLAVE_AB` se cambió por `llaveAb`, etc.)

Nótese que algunos cambios no se han efectuado, por ejemplo en las reglas `expNegacion` e `instNula`, por motivos de eficiencia.

Al final del fichero se han añadido las reglas que, al ser sobreescritas en `LeLiErrorRecoveryParser.g`, darán lugar a las trampas para excepciones.

```
header{

package leli;

/*-------------------------------------------------*\
 | Un intérprete para un Lenguaje Limitado(LeLi)  |
 | _____________________________________________  |
 |                                                |
 |              ANALISIS SINTÁCTICO               |
 | _____________________________________________  |
 |                                                |
 |            Enrique J. Garcia Cota              |
\*-------------------------------------------------*/

}

/**
 *  El objeto que permite todo el analisis sintactico.
 */
class LeLiParser extends Parser;

options
{
   k = 2;
   importVocab = LeLiLexerVocab;
   exportVocab = LeLiParserVocab;
   buildAST = true;
}

tokens
{
// Tokens imaginarios para enraizar listas
   PROGRAMA ;
   LISTA_MIEMBROS;
```

```
   LISTA_EXPRESIONES;
   LISTA_INSTRUCCIONES;
   LISTA_DEC_PARAMS;

// Tokens imaginarios que se utilizan cuando no hay raíces adecuadas
   OP_MENOS_UNARIO;
   INST_EXPRESION;
   INST_DEC_VAR;
   LLAMADA;
   ATRIBUTO;

// Otros
   TIPO_VACIO;
}


/**
 * Un programa esta formado por una lista de definiciones de clases.
 */
programa : (decClase)+
           {## = #( #[PROGRAMA, "PROGRAMA"] ,##);}
      ;

/** Definición de una clase. */
decClase : RES_CLASE^ IDENT clausulaExtiende
           llaveAb listaMiembros llaveCe
         ;

/** Cláusula "extiende" **/
clausulaExtiende : RES_EXTIENDE^ IDENT
                 | /*nada*/
                   { ## = #( #[RES_EXTIENDE,"extiende"],
                             #[IDENT, "Objeto"] ); }
                 ;

/** Auxiliar para definición de clase */
protected listaMiembros
   : (decMetodo|decConstructor|decAtributo)*
     { ## = #( #[LISTA_MIEMBROS, "LISTA_MIEMBROS"] ,##);}
   ;

/**
 * Declaración de los diferentes tipos que se pueden usar en LeLi
 */
tipo : TIPO_ENTERO   // tipo Entero
     | TIPO_REAL     // tipo Real
     | TIPO_BOOLEANO // tipo Booleano
     | TIPO_CADENA   // tipo Cadena
     | IDENT         // tipo no básico
     ;

declaracion ! // desactiva construcción por def. del AST
[AST r, AST t, boolean inicializacion] // parámetros
{
   AST raiz = astFactory.dupTree(r);     // copia del arbol
   raiz.addChild(astFactory.dupTree(t)); // copia del árbol
}
   : i1:IDENT
```

```
     {
        raiz.addChild(#i1);
        ## = raiz;
     }
   | { inicializacion }? // pred. semántico
     i2:IDENT OP_ASIG valor:expresion
     {
        raiz.addChild(#i2);
        raiz.addChild(#valor);
        ## = raiz;
     }
   | { inicializacion }?
     i3:IDENT parentAb li:listaExpresiones parentCe
     {
        raiz.addChild(#i3);
        raiz.addChild(#li);
        ## = raiz;
     }
   ;

listaDeclaraciones [AST raiz, boolean inicializacion]
   : t:tipo!
     declaracion[raiz,#t,inicializacion]
     (coma declaracion[raiz,#t,inicializacion])*
   ;


/**
 * Definición de un atributo (abstracto o no abstracto)
 */
decAtributo
{ boolean abstracto = false; }
   : raiz:RES_ATRIBUTO^
     ( a:RES_ABSTRACTO { abstracto=true; } )?
     listaDeclaraciones[#raiz, abstracto] puntoComa
   ;

/** Definición de un constructor */
decConstructor : RES_CONSTRUCTOR^ parentAb listaDecParams parentCe
                 llaveAb listaInstrucciones llaveCe
               ;

/** Definición de un método normal */
decMetodo
   : RES_METODO^ (RES_ABSTRACTO)? tipoRetorno IDENT
     parentAb listaDecParams parentCe
     llaveAb listaInstrucciones llaveCe
   ;

/** Regla auxiliar que codifica el tipo de retorno de un método */
protected tipoRetorno
   : tipo
   | /* nada */ {## = #[TIPO_VACIO,"TIPO_VACIO"]; }
   ;

/** Lista de parámetros. */
listaDecParams
{ final AST raiz = #[RES_PARAMETRO,"parámetro"] ;}
```

```
   :
   ( listaDeclaraciones[raiz, false]
     ( puntoComa listaDeclaraciones[raiz, false])*
   )? // opcional
   { ## = #(#[LISTA_DEC_PARAMS,"LISTA_DEC_PARAMS"], ##); }
    ;

/**
 * Regla que sirve para empezar a reconocer las expresiones de LeLi
 */
expresion: expAsignacion;

/** Asignaciones (nivel 9) */
expAsignacion : expOLogico (OP_ASIG^ expOLogico)? ;

/** O lógico (nivel 8) */
expOLogico : expYLogico (OP_O^ expYLogico)* ;

/** Y lógico (nivel 7) */
expYLogico : expComparacion (OP_Y^ expComparacion)* ;

/** Comparación (nivel 6) */
expComparacion
   : expAritmetica
   (
      ( OP_IGUAL^ | OP_DISTINTO^ | OP_MAYOR^ | OP_MENOR^
      | OP_MAYOR_IGUAL^ | OP_MENOR_IGUAL^
      )
      expAritmetica
   )*
  ;

/** Suma y resta aritmética (nivel 5) */
expAritmetica : expProducto ((OP_MAS^ | OP_MENOS^) expProducto)* ;

/** Producto y división (nivel 4) */
expProducto : expCambioSigno
              ((OP_PRODUCTO^ | OP_DIVISION^) expCambioSigno)*
            ;

/** Cambio de signo (nivel 3) */
expCambioSigno :
   ( OP_MENOS! expPostIncremento
     { ##=#(#[OP_MENOS_UNARIO, "OP_MENOS_UNARIO"], ##) ;}
   )
   | (OP_MAS!)? expPostIncremento
   ;

/** Postincremento y postdecremento (nivel 2) */
expPostIncremento :  expNegacion (OP_MASMAS^|OP_MENOSMENOS^)? ;

/** Negación y accesos (nivel 1) */
expNegacion : (OP_NO^)* expEsUn
            ;

/** EsUn + accesos (nivel 0) **/
expEsUn : acceso (RES_ESUN^ tipo)*
        ;
```

```
/**
 * Regla que permite reconocer los accesos de las expresiones de LeLi.
 * Los accesos son los valores que se utilizan en las expresiones:
 * literales, variables, llamadas a métodos, etc.
 */
acceso : r1:raizAcceso { ## = #(#[ACCESO], #r1); }
         ( punto sub1:subAcceso! { ##.addChild(#sub1); } )*
       | r2:raizAccesoConSubAccesos { ## = #(#[ACCESO], #r2); }
         ( punto sub2:subAcceso! { ##.addChild(#sub2); } )+
       ;

/**
 * Raíz de los accesos que no son llamadas a un método de la
 * clase "actual"
 */
raizAcceso : IDENT
           | literal
           | llamada
           | conversion
           | PARENT_AB! expresion parentCe
           ;
/**
 * Raíz de los accesos que no son llamadas a un método de la
 * clase "actual" y que obligatoriamente van sucedidos de un subacceso
 */
raizAccesoConSubAccesos
   : RES_PARAMETRO
   | RES_ATRIBUTO
   | RES_SUPER
   ;

/** Regla que reconoce los literales */
literal : LIT_ENTERO
        | LIT_REAL
        | LIT_CADENA
        | LIT_NL
        | LIT_TAB
        | LIT_COM
        | LIT_CIERTO
        | LIT_FALSO
        ;

/**
 * Esta regla se utiliza tanto para representar:
 * 1) Una llamada a un método del objeto actual
 * 2) Un subacceso en forma de llamada.
 * 3) Una llamada a un constructor del objeto actual
 * 4) Un subacceso en forma de constructor.
 */
llamada : IDENT parentAb listaExpresiones parentCe
          { ## = #(#[LLAMADA,"LLAMADA"],##); }
        | RES_CONSTRUCTOR^ parentAb listaExpresiones parentCe
        ;

/**
 *  Regla auxiliar que reconoce los parámetros de una llamada
 *  a un método y la inicialización del bucle "desde"
```

```
 */
protected listaExpresiones
   : ( expresion (coma expresion)* )?
     { ## = #(#[LISTA_EXPRESIONES, "LISTA_EXPRESIONES"],##); }
   ;

/** Conversión entre tipos */
conversion : RES_CONVERTIR^
             parentAb expresion coma tipo parentCe
           ;

/** Regla que reconoce los accesos a atributos y métodos de un objeto. */
subAcceso
   : llamada
   | IDENT
   | RES_SUPER
   ;

/** Una lista de 0 o más instrucciones */
listaInstrucciones
   : (instruccion)*
     {## = #( #[LISTA_INSTRUCCIONES,"LISTA_INSTRUCCIONES"], ##);}
   ;

/**
 * Las instrucciones. Pueden ser expresiones, instrucciones de control,
 * declaraciones de variables locales o la instrucción volver.
 */
instruccion : (tipo IDENT)=>instDecVar // declaración
            | instExpresion            // Instrucción - expresión
            | instMientras             // bucle mientras
            | instHacerMientras        // bucle hacer-mientras
            | instDesde                // bucle desde
            | instSi                   // Instrucción Si
            | instVolver               // Instrucción volver
            | instNula                 // Instrucción Nula
            ;

/** Instrucción nula */
instNula : PUNTO_COMA! ;

/** Instrucción volver */
instVolver : RES_VOLVER puntoComa ;

/** Instrucción-expresión */
instExpresion: expresion puntoComa
              {## = #( #[INST_EXPRESION,"INST_EXPRESION"], ##);}
             ;

/** Declaración de variables locales */
instDecVar
{ final AST raiz = #[INST_DEC_VAR,"variable"]; }
   : listaDeclaraciones[raiz,true] puntoComa ;

/** Bucle "mientras" */
instMientras : RES_MIENTRAS^ parentAb expresion parentCe
               llaveAb listaInstrucciones llaveCe
             ;
```

```
/** Bucle "hacer-mientras" */
instHacerMientras : RES_HACER^ llaveAb listaInstrucciones llaveCe
                    RES_MIENTRAS parentAb expresion parentCe
                    puntoComa
                  ;

/** Bucle "desde" */
instDesde : RES_DESDE^ parentAb listaExpresiones puntoComa
                                 listaExpresiones puntoComa
                                 listaExpresiones parentCe
            llaveAb listaInstrucciones llaveCe
          ;

/** Instruccion "si" muy parecida a la del lenguaje LEA. */
instSi :
   RES_SI^ parentAb expresion parentCe
   llaveAb listaInstrucciones llaveCe
   alternativasSi
   ;

/**
 * Auxiliar (reconoce las alternativas de la instrucción "si"
 * sin warnings de ambigüedad)
 */
protected alternativasSi
   : altSiNormal alternativasSi
   | altSiOtras
   | /* nada */
   ;

/** Auxiliar (alternativa normal de la instrucción "si") */
protected altSiNormal : BARRA_VERT^ parentAb expresion parentCe
                        llaveAb listaInstrucciones llaveCe
                      ;

/** Auxiliar (alternativa final "otras" de la instrucción "si") */
protected altSiOtras : barraVert RES_OTRAS^
                       llaveAb listaInstrucciones llaveCe
                     ;

// Reglas auxiliares que serán reescritas para el tratamiento de
// errores
parentAb : PARENT_AB! ;
parentCe : PARENT_CE! ;
llaveAb : LLAVE_AB! ;
llaveCe : LLAVE_CE! ;
coma : COMA! ;
barraVert : BARRA_VERT! ;
puntoComa : PUNTO_COMA! ;
punto : PUNTO! ;
```

6.7.3: Fichero LeLiErrorRecoveryParser.g

Éste es el fichero donde se implementan los principales mecanismos de recuperación sintáctica. Es el analizador más complicado que hemos realizado hasta ahora.

```
header{

package leli;

/*-----------------------------------------------*\
 | Un intérprete para un Lenguaje Limitado(LeLi) |
 | _____________________________________________ |
 |                                               |
 | ANALISIS SINTÁCTICO (RECUPERACIÓN DE ERRORES) |
 | _____________________________________________ |
 |                                               |
 |            Enrique J. Garcia Cota             |
\*-----------------------------------------------*/

import antlraux.Logger;

import java.util.HashSet;

}

/**
 *  El objeto que permite todo el analisis sintactico con
 *  recuperación de erroes.
 */
class LeLiErrorRecoveryParser extends LeLiParser;

options{
   k=1;
   importVocab=LeLiParserVocab;
}

{
   /** Gestor de errores del analizador **/
   public Logger logger = null;

   /** Símbolos de sincronismo **/
   public static final BitSet simbolosSincro = mk_simbolosSincro();

   /** Iniciar simbolosSincro **/
   protected static final BitSet mk_simbolosSincro()
   {
      BitSet b = new BitSet();
      b.add(LLAVE_AB);
      b.add(LLAVE_CE);
      b.add(PARENT_AB);
      b.add(PARENT_CE);
      b.add(COMA);
      b.add(PUNTO_COMA);
      b.add(BARRA_VERT);

      b.add(RES_CLASE);
      b.add(RES_METODO);
      b.add(RES_ATRIBUTO);
      b.add(RES_DESDE);
      b.add(RES_HACER);
      b.add(RES_MIENTRAS);
      b.add(RES_SI);
      b.add(RES_VOLVER);
```

```
   b.add(IDENT); // para método y parámetro
   return b;
}

/** Identificadores de sincronismo **/
public static final HashSet identsSincro = mk_identsSincro();

/** Iniciar identsSincro **/
public static final HashSet mk_identsSincro()
{
   HashSet hs = new HashSet();
   hs.add("metodo");
   hs.add("parametro");
   return hs;
}

/**
 * Método de sincronización con símbolos de sincronismo
 * y teniendo en cuenta algunos identificadores de sincronismo
 **/
public void sincronizar ( RecognitionException re,
                          BitSet SIGUIENTE )
throws RecognitionException, TokenStreamException
{
   // Si es SS, "dejar pasar" (terminar la regla)
   if( ! simbolosSincro.member(LA(0)) )
   {
      // Si estamos guessing, lanzamos directamente
      if (inputState.guessing!=0) throw re;

      // Crear un nuevo bitset que contenga a los
      // símbolos de sincronismo y a "siguientes"
      // utilizando la "or" lógica
      BitSet auxiliar = simbolosSincro.or(SIGUIENTE);

      // Mostrar el error y consumir, pero utilizando
      // auxiliar en lugar de SIGUIENTE
      reportError(re);
      boolean bSincronizado = false;
      Token t = null;
      do
      {
         consume();
         consumeUntil(auxiliar);
         bSincronizado = true;
         t = LT(0);
         // Excepciones con IDENT ("metodo" y "parametro"
         // son los únicos idents válidos para sincronizar)
         if( t.getType()==IDENT &&
             !identsSincro.contains(t.getText()) )
         { bSincronizado = false; }
         else
         { bSincronizado = true; }
      }
      while (!bSincronizado);
   }
}
```

```
/** Constructor añadiendo un ErrorManager **/
public LeLiErrorRecoveryParser(TokenStream lexer, Logger _logger)
{
   this(lexer);
   logger = _logger;
}

/** imprime un error **/
public void reportError( String msg,
                         String filename,
                         int line,
                         int column )
{
   if(null==logger)
   {
      logger = new Logger("error", System.err);
   }
   logger.log( msg, 1, filename, line, column);
}

/** Sobreescribir reportError **/
public void reportError(RecognitionException e)
{
   reportError( e.getMessage(),
                e.getFilename(),
                e.getLine(),
                e.getColumn() );
}

/** Error que se lanza cuando falta un token "necesario" **/
public void errorFaltaToken( int token )
throws TokenStreamException
{
   Token t0 = LT(0);
   Token t1 = LT(1);

   if(t1.getType() != Token.EOF_TYPE)
   {
      String t1Text;
      if(t1.getType()==IDENT)
         t1Text = "el identificador '" + t1.getText() +"'";
      else
         t1Text = getTokenName(t1.getType());

      reportError( "Se esperaba " + getTokenName(token) +
                   ", se encontró " + t1Text,
                   this.getFilename(),
                   t1.getLine(),
                   t1.getColumn() - t1.getText().length() );
   }
   else
   {
      reportError ( "Se esperaba " + getTokenName(token) +
                    " cuando se alcanzó el final de fichero",
                    this.getFilename(),
                    t0.getLine(),
                    t0.getColumn() ) ;
```

```
   }
}

/**
 * Error habitual de acentuación (tilde)
 * @returns true si t.getText().equals(textoIncorrecto),
 * false en cualquier otro caso
 **/
public boolean comprobarErrorTilde (
   Token t,
   AST ast,
   String textoIncorrecto,
   int tipoCorrecto,
   String textoCorrecto )
{
   if( t.getText().equals(textoIncorrecto) )
   {
      reportError( "Omisión de tilde. " +
                  "Escriba '" + textoCorrecto +
                   "' en lugar de '" + t.getText() + "'",
                   this.getFilename(),
                   t.getLine(),
                   t.getColumn() );
      ast.setType( tipoCorrecto );
      ast.setText( textoCorrecto );
      return false;
   }

   return true;
}

/** Re-construcción de una expresión unaria */
public AST construyeExpUn(AST operando, AST operador)
{
   if(null==operador) return operando;
   if(operando.getNextSibling()==operador)
   {
      operando.setNextSibling(null);
   }
   operador.setFirstChild(operando);
   return operador;
}

/** Re-construcción de una expresión binaria */
public AST construyeExpBin ( AST izquierda, AST derecha )
{
   if(derecha != null)
   {
      // "desconectar" los nodos si es necesario
      if(izquierda.getNextSibling()==derecha)
      {
         izquierda.setNextSibling(null);
      }

      AST valor = derecha.getFirstChild();
      return #(derecha, izquierda, valor);
   }
   return izquierda;
```

```
   }
}

// ---------------------- HACER k=1 ----------------------------

declaracion ! // desactivar construcción por defecto
[AST r, AST t, boolean inicializacion] // parámetros
{
   AST raiz = astFactory.dupTree(r);     // copia del árbol
   raiz.addChild(astFactory.dupTree(t)); // copia del árbol
}
   : { inicializacion }? // pred. semántico
      (IDENT OP_ASIG) => i1:IDENT OP_ASIG valor:expresion
     {
        raiz.addChild(#i1);
        raiz.addChild(#valor);
        ## = raiz;
     }
   | { inicializacion }?
      (IDENT PARENT_AB) =>
     i2:IDENT parentAb le:listaExpresiones parentCe
     {
        raiz.addChild(#i2);
        raiz.addChild(#le);
        ## = raiz;
     }
   | i3:IDENT
     {
        raiz.addChild(#i3);
        ## = raiz;
     }
   ;

/** Regla auxiliar que codifica el tipo de retorno de un método */
protected tipoRetorno
   : (tipo IDENT) => tipo
   | /* nada */ {## = #[TIPO_VACIO,"TIPO_VACIO"]; }
   ;

/* nota : raizAcceso es declarada más abajo */
// raizAcceso : ((IDENT|RES_CONSTRUCTOR) PARENT_AB)=>llamada
//            | IDENT
//            | literal
//            | conversion
//            | PARENT_AB! expresion parentCe
//            ;


/** Regla que reconoce los accesos a atributos y métodos de un objeto. */
subAcceso
   : ((IDENT|RES_CONSTRUCTOR) PARENT_AB)=>llamada
   | IDENT
   | RES_SUPER
   ;
exception catch [NoViableAltException nvae]
{ sincronizar(nvae, $FOLLOW); }

/**
```

```
 * Auxiliar (reconoce las alternativas de la instrucción "si"
 * sin warnings de ambigüedad)
 */
protected alternativasSi
   : (BARRA_VERT PARENT_AB)=>altSiNormal alternativasSi
   | altSiOtras
   | /* nada */
   ;
exception catch [NoViableAltException nvae]
{ sincronizar(nvae, $FOLLOW); }


// ------------------ TRAMPAS PARA EXCEPCIONES --------------------

/** Regla auxiliar para reconocer paréntesis necesarios **/
parentAb : PARENT_AB! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(PARENT_AB); }

/** Regla auxiliar para reconocer paréntesis necesarios **/
parentCe : PARENT_CE! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(PARENT_CE); }

/** Regla auxiliar para reconocer llaves necesarias **/
llaveAb : LLAVE_AB! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(LLAVE_AB); }

/** Regla auxiliar para reconocer llaves necesarias **/
llaveCe : LLAVE_CE! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(LLAVE_CE); }

/** Regla auxiliar para reconocer comas necesarias **/
coma : COMA! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(COMA); }

/** Regla auxiliar para reconocer barras verticales necesarias **/
barraVert : BARRA_VERT! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(BARRA_VERT); }

/** Regla auxiliar para reconocer puntos y comas necesarios **/
puntoComa : PUNTO_COMA! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(PUNTO_COMA); }

/** Regla auxiliar para reconocer puntos **/
punto : PUNTO! ;
exception catch [RecognitionException ex]
{ errorFaltaToken(PUNTO); }

// ------------ RETARDO DEL TRATAMIENTO DE ERRORES ----------

/** Asignaciones (nivel 9) */
expAsignacion : izq:expOLogico der:expAsignacion2
```

```
                { ## = construyeExpBin(#izq, #der); }
              ;

expAsignacion2 : ( OP_ASIG^ expOLogico )? ;
exception catch [NoViableAltException nvae]
{ sincronizar(nvae, $FOLLOW); }

/** Postincremento y postdecremento (nivel 2) */
expPostIncremento :  operando:expNegacion operador:expPostIncremento2
                     { ## = construyeExpUn(#operando, #operador); }
                  ;

expPostIncremento2 : ( OP_MASMAS^|OP_MENOSMENOS^ )?
                   ;
exception catch [NoViableAltException nvae]
{ sincronizar(nvae, $FOLLOW); }

/** Cláusula "extiende" **/
clausulaExtiende : RES_EXTIENDE^ IDENT
                 | /*nada*/
                   { ## = #( #[RES_EXTIENDE,"extiende"],
                             #[IDENT, "Objeto"] ); }
                 ;
exception catch [NoViableAltException nvae]
{
   sincronizar(nvae, $FOLLOW);
   ## = #( #[RES_EXTIENDE,"extiende"], #[IDENT, "Objeto"] );
}

/** Lista de parámetros. */
listaDecParams
{ final AST raiz = #[RES_PARAMETRO,"parámetro"] ; }
   :
   { ## = #[LISTA_DEC_PARAMS,"parámetros"]; }
   ( l1:listaDeclaraciones[raiz, false]!
     { ##.addChild(#l1); }
     ( puntoComa ln:listaDeclaraciones[raiz, false]!
       { ##.addChild(#ln); }
     )*
   )? // opcional
    ;
    // Capturar NoViableAltException y resincronizar
   exception catch [NoViableAltException nvae]
   { sincronizar(nvae, $FOLLOW); }

// ------------------- ERRORES DE ACENTUACIÓN ------------

/**
 * Raíz de los accesos que no son llamadas a un método de la
 * clase "actual" (recuperación del error de acentuación en
 * "parámetro")
 */
raizAcceso : ((IDENT|RES_CONSTRUCTOR) PARENT_AB)=>llamada
           | i:IDENT
             { comprobarErrorTilde( i, #i, "parametro",
                                    RES_PARAMETRO, "parámetro" ); }
           | literal
           | conversion
```

```
           | PARENT_AB! expresion parentCe
           ;
exception catch [NoViableAltException nvae]
{ sincronizar(nvae, $FOLLOW); }

/**
 * Definición de un método normal ( con recuperación del error
 * de acencuación de "método"
 **/
decMetodo
   : r:resMetodo! (RES_ABSTRACTO)? tipoRetorno IDENT
     parentAb listaDecParams parentCe
     llaveAb listaInstrucciones llaveCe
     { ## = #(#r,##); }
   ;

/** Regla que permite la recuperación del error de omisión de tilde */
resMetodo : RES_METODO
          | i:IDENT
            {
             if( comprobarErrorTilde( i, #i, "metodo",
                                      RES_METODO, "método") )
                {
                  throw new MismatchedTokenException( tokenNames,
                                                      i,
                                                      RES_METODO,
                                                      false,
                                                      getFilename() );
                }
            }
          ;
exception catch [NoViableAltException nvae]
{ sincronizar(nvae, $FOLLOW); }
```

Sección 6.8: Compilando y ejecutando el analizador

6.8.1: Compilación

Compilar el analizador sintáctico con recuperación de errores es muy sencillo: hay que hacer lo mismo que para compilar el analizador sintáctico normal, salvo por el hecho de que hay que incluir el fichero que contiene la "supergramática" (`LeLiParser.g`) en la línea de comandos, con la opción `-glib`. Es decir, que hay que compilarlo con la siguiente línea de comandos:

```
c:\leli\ java antlr.Tool -glib LeLiParser.g LeLiErrorRecoveryParser.g
```

Este proceso generará los ficheros habituales:

- `LeLiErrorRecoveryParser.java`
- `LeLiErrorRecoveryParserVocabTokenTypes.java`
- `LeLiErrorRecoveryParserVocabTokenTypes.txt`

y además un fichero con la gramática expandida, llamado `expandedLeLiErrorRecoveryParser.g`. Éste será el fichero que utilizará ANTLR para generar los tres anteriores.

Los números de línea errores que encuentre ANTLR a la hora de compilar un fichero *.g de una subgramática serán referencias al fichero de la gramática expandida, no del fichero original.

Los ficheros *.java se compilarán de la manera usual, utilizando el compilador de java, sin ningún añadido especial:

```
c:\leli\ javac *.java
```

6.8.2: Ejecución

En este apartado vamos a ver cómo lanzar el análisis con detección de errores. Concretamente, vamos a ver cómo modificar la clase `leli.Tool` para que lo utilice. Por lo tanto se hará mención a métodos y atributos de la clase que ya se explicaron en capítulos anteriores. Remítase a ellos si tiene dificultades para seguir esta sección.

Nuevos comandos

Una opción de la línea de comandos, `-erec`, permitirá activar o desactivar la recuperación de errores (utilizando una instancia de `LeLiParser` o `LeLiErrorRecoveryParser` para realizar el análisis).

Habrá que añadir un atributo con su método "setter":

```
public class Tool
{
   ...
   public boolean recuperacion = true;
   ...
   public void fijarRecuperacion(Boolean B)
   { recuperacion = B.booleanValue(); }
```

Después habrá que asociar el método al nuevo comando en `LeeLC`:

```
   public void leeLC(String args[])
   throws CommandLineParserException
   {
      ...
           clp.addCommand(this, "-erec", "fijarRecuperacion", "b",
              "Activa/desactiva la recuperación de errores sintácticos" );
      ...
```

Finalmente hay que modificar el método `trabaja` para que, en función del atributo `recuperacion`:

```
public void trabaja()
   {
      // También hay que crear el Logger!
      Logger logger = new Logger("error", System.err);
      try
      {
         // PASOS 2 y 3 permanecen sin modificaciones.
        ...
         // PASOS 4 y 5. Crear analizador sintáctico y pasarle
         // nombre fichero. El tipo de parser depende del atributo
         // recuperacion
         antlr.Parser parser = null;

         if(recuperacion)
            parser = new LeLiErrorRecoveryParser(lexer, logger);
         else
            parser = new LeLiParser(lexer);

         parser.setFilename(nombreFichero);
```

Nótese la creación y el uso de la clase `Logger` para manejar los mensajes de error.

Nuevo código de la clase Tool

La nueva clase `Tool` tendrá, por lo tanto, el siguiente aspecto:

```
package leli; // Tool también pertenece al paquete leli

import antlr.*;   // Incluir todo antlr
import java.io.*; // Para entrada-salida
import antlraux.util.LexInfoToken; // Nueva clase Token
import antlraux.util.Logger; // Logger

public class Tool{
   public static void main(String [] args)
   {
      try
      {
         FileInputStream fis = new FileInputStream(args[0]);
         LeLiLexer lexer = new LeLiLexer(fis);
         lexer.setFilename(args[0]);
         lexer.setTokenObjectClass("antlraux.util.LexInfoToken");

         Token tok = lexer.nextToken();
         while(tok.getType() != Token.EOF_TYPE)
         {
            System.out.println( tok.getFilename()+ ":" +
```

```
                                tok.getLine() +  ":" +
                                tok.getColumn() +  ": " +
                                tok.getText()+ ", " + tok.getType() );

            tok = lexer.nextToken();
          }

      }
      catch (FileNotFoundException fnfe)
      {
         System.err.println("No se encontró el fichero");
      }
      catch (TokenStreamException tse)
      {
         System.err.println("Error leyendo tokens");
      }
   }
}
```

Sección 6.9: Conclusión

El tratamiento de errores es una tarea un poco más desagradable que la construcción de analizadores sin gestión de errores. Al menos así me lo parece personalmente.

El código de los analizadores con gestión de errores, si no se tiene mucho cuidado, se convierte rápidamente en un gran galimatías, ininteligible a primera vista. Además, el código nativo (es decir, las acciones) hacen que el analizador se vuelva muy dependiente del lenguaje utilizado para generar.

Nosotros hemos solucionado algunos de estos problemas (la complejidad no tiene solución) con la herencia de gramáticas. Hemos conseguido aislar todo ese código "feo" y "poco mantenible" en una subgramática del analizador, dejando `LeLiParser` muy limpio, ofreciendo un esqueleto sobre el que poder implementar otros analizadores (por ejemplo, un compilador de LeLi en C++ y con recuperación de errores).

Esta importancia de la búsqueda la la limpieza en el código es uno de los principiales puntos que quería resaltar en este capítulo. Hasta ahora, el código de recuperación de errores se mezclaba sin orden ni concierto dentro del analizador, resultando un analizador muy poco deseable, al menos desde el punto de vista de la mantenibilidad.

El otro punto a resaltar en este capítulo es, por supuesto, la propia recuperación de errores. He propuesto algunos mecanismos y dado algunas ideas para la recuperación, resultando en un analizador bastante resistente[56].

Pero lo verdaderamente importante es comprender la esencia de la recuperación de errores. De momento, para poder implementar una buena recuperación de errores, sigue siendo muy importante conocer la manera en que el código del analizador se genera. El código de la recuperación ha sido embebido dentro del propio analizador. Una solución más deseable sería que al menos el método `sincronizar` fuera proporcionado por la librería antlraux; quizás lo haga en un futuro.

Por último hemos aprendido cómo se mejoran los mensajes de error de ANTLR, utilizando los alias de tokens y aprendiendo a traducir los mensajes a cualquier idioma (aunque nos hayamos decidido por el castellano). Esta "proeza", por supuesto, también puede lograrse con Bison y Flex: basta con buscar las cadenas con los mensajes de error en el código del analizador y recompilar (como quiera que se haga eso). Es la belleza del software de código abierto.

Hay un punto en el cual no nos hemos detenido: recuperación de errores en el análisis léxico. Nos hemos centrado muchísimo en el análisis sintáctico, como si fuera el único sitio donde se puede recuperar un error. Los analizadores léxicos de ANTLR tienen su propia infraestructura error/excepción/manejador para gestionar los errores en la entrada.

Sin embargo, el número de errores a nivel léxico es apabullantemente bajo en comparación con los que ocurren a nivel sintáctico. Además, ANTLR también proporciona un mecanismo por defecto para tratar dichos errores. Por estas dos razones he preferido dejar un poco de lado los errores léxicos y centrarme en los sintácticos. ¡Pero cualquier mente inquieta es libre de investigar!

En el capítulo siguiente abandonaremos completamente el nivel sintáctico para pasar al nivel semántico. Dejaremos de analizar flujos de datos para empezar a analizar datos estructurados en forma de ASTs.

56 ¡Aunque siempre es mejorable!

Capítulo 7: Análisis semántico de LeLi

"El verdadero significado de las cosas se encuentra al decir las mismas cosas con otras palabras."
Charles Chaplin

Capítulo 7: Análisis semántico de LeLi.....244

Sección 7.1: Introducción.....247
7.1.1: Errores semánticos estáticos..... 247
Errores de tipo.....247
Errores de lectura/escritura: R-value y L-value.....248
Errores de ubicación.....248
Errores de expresiones evaluables durante la compilación.....248
Errores de contexto.....249
7.1.2: Errores semánticos dinámicos..... 250
7.1.3: Añadiendo información léxica a los ASTs..... 250
El problema.....250
La clase antlraux.util.LexInfoAST.....250
Facilitándonos la labor en los iteradores de árboles: ASTLabelType.....251
Sección 7.2: Iterador simple de árboles..... 253
7.2.1: Estrategia de implementación del análisis semántico..... 253
7.2.2: Estructura del fichero de gramática..... 254
7.2.3: Regla raíz: Programa..... 254
7.2.4: Definición de clases..... 254
7.2.5: Definición de atributos..... 255
Definición de métodos normales, métodos abstractos y constructores.....255
7.2.6: Expresiones..... 256
7.2.7: Instrucciones..... 257
7.2.8: Código completo del iterador..... 258
7.2.9: Los errores de construcción del AST..... 262
Sección 7.3: El sistema Ámbito/Declaración/Tipo..... 263
7.3.1: Introducción..... 263
7.3.2: Presentación de el sistema Ámbito/Declaración/Tipo (ADT)..... 263
Evolución de la encapsulación en los lenguajes de programación.....263
Ámbitos.....264
Declaraciones.....265
Tipos.....266
ASTs especializados.....266
7.3.3: Implementación del sistema ADT: el paquete antlraux.context..... 266
La excepción antlraux.context.ContextException.....266
Un buen sistema de identificación : la dupla nombre-"etiqueta".....267
7.3.4: La clase antlraux.context.Scope..... 268
Identificación: nombre y etiqueta de los ámbitos.....268
Inserción de declaraciones.....268
Implementación implícita de la pila de ámbitos y búsquedas locales y no locales.....268
Organización de las declaraciones dentro de un ámbito.....269
El doble ordenamiento de los elementos.....270
7.3.5: La clase antlraux.context.Declaration..... 272
Elementos básicos: nombre y tipo.....272
Identificación: Etiqueta y nombre.....272
Inicialización.....272
Otros campos.....272
7.3.6: El sistema de tipos (antlraux.context.types.*)..... 273
Idenfificación: la interfaz antlraux.contex.types.Type.....273
Otras interfaces y clases interesantes del sistema de tipos.....273
Tipos "especiales".....274
El tipo de un método: antlraux.context.types.MethodType.....274

Compatibilidad de métodos....276
El tipo de un atributo: antlraux.context.types.AttributeType....277
El tipo de una clase: antlraux.context.types.ClassType....278
7.3.7: ASTs especializados....279
antlraux.context.asts.ScopeAST....279
antlraux.context.asts.TypeAST....280
antlraux.context.ExpressionAST....280
Notas sobre los árboles....280
7.3.8: Resumen....283
Sección 7.4: ADT y LeLi....284
7.4.1: Introducción....284
7.4.2: Las declaraciones en LeLi....284
7.4.3: Los ámbitos en LeLi....284
Algunas restricciones....284
La clase antlraux.context.Context....285
Utilización de Scope y Context....285
7.4.4: El sistema de tipos de LeLi....286
Tratamiento de errores semánticos y el Tipo Error....286
El paquete leli.types....287
Modelado de los tipos “normales” de LeLi....287
Gestión de los tipos predefinidos: LeLiTypeManager....288
Metaclases....289
Sección 7.5: Comprobación de tipos - Primera pasada....292
7.5.1: Comprobación de tipos en dos pasadas....292
Explicación detallada de los objetivos del analizador....293
Las fases del analizador....294
7.5.2: Definición del analizador....294
Cabecera....294
Zona de opciones....294
Zona de tokens....295
Zona de código nativo....295
7.5.3: Fase 1: Creación de ámbitos....296
Programa....296
Declaración de clases....296
Declaraciones de métodos....298
Declaraciones de constructores....299
Declaración de bucles....299
Instrucciones condicionales....300
Declaración de atributos....301
Declaraciones de parámetros....301
7.5.4: Fase 2: Preparación de las expresiones....302
7.5.5: Fase 3: Preparación de los tipos....303
7.5.6: Fase 4: Preparación del ámbito global....304
El fichero de tipos básicos....304
7.5.7: Fichero LeLiSymbolTreeParser.g....307
7.5.8: Notas finales sobre el analizador....316
Sección 7.6: Comprobación de tipos – segunda pasada....317
7.6.1: Introducción....317
Objetivos....317
Fases del análisis....317
7.6.2: Definición del analizador....317
Cabecera....317
Zona de opciones....318
Zona de tokens....318
Zona de código nativo....318
7.6.3: Fase 1 – Mantener el ámbito actual....319
Programa....320
Declaraciones de clases....320
Declaraciones de métodos y constructores....320
Bucles....321
Instrucciones condicionales....321
7.6.4: Fase 2: Adición de las variables locales a los ámbitos....322
7.6.5: Fase 3: Expresiones....322
errorExpresion....323
Expresión “esUn”....323

Expresiones unarias....325
Expresiones binarias....328
7.6.6: Fase 4: Accesos.... 331
Raíz de un acceso....332
Literales....335
Invocaciones....336
Conversiones de tipo....337
Los subaccesos....339
7.6.7: Fichero LeLiTypeCheckTreeParser.g.... 341
7.6.8: Notas finales sobre el analizador.... 358
Sección 7.7: Compilación y ejecución.... 359
7.7.1: Compilación.... 359
7.7.2: Ejecución.... 359
Cambios en la clase Tool....359
Código de la nueva clase Tool....359
Sección 7.8: Conclusión.... 364

Sección 7.1: Introducción

En el análisis léxico se detectaban los tokens. En el análisis sintáctico se agrupaban dichos tokens en reglas sintácticas, y se construía un primer AST.

El cometido del análisis semántico es doble:

- Por un lado debe detectar errores semánticos (incoherencias con respecto a las reglas semánticas del lenguaje).
- Por otro, debe enriquecer el AST con información semántica que será necesaria para la generación de código.

Los errores semánticos a los que nos estamos refiriendo aquí son los errores semánticos estáticos, que son aquellos detectados durante la compilación (en el análisis semántico). En oposición a los errores semánticos estáticos están los errores semánticos dinámicos, que se detectan durante la ejecución del programa ya compilado (y no forman parte del análisis semántico propiamente dicho).

7.1.1: Errores semánticos estáticos

Una vez pasado el nivel sintáctico, un programa aún puede tener errores detectables durante la compilación. Suelen tener lugar en las expresiones, pero no pueden ser detectados en el análisis sintáctico porque son sintácticamente correctos (pero incoherentes con algún aspecto semántico del lenguaje). Estos errores son los errores semánticos estáticos.

Los errores semánticos estáticos se dividen en diferentes grupos:

- Errores de tipo
- Errores de lectura/escritura de expresiones (o de L-value y R-value)
- Errores de restricción de ubicación
- Errores de expresiones evaluables en tiempo de compilación
- Errores de contexto

Errores de tipo

Son los errores semánticos estáticos más comunes. Aparecen cuando una o varias expresiones son incompatibles a nivel de un operador. Considérese el siguiente ejemplo:

```
clase Persona
{
   atributo Cadena nombre;
   atributo Entero edad;

   constructor(Entero E, Cadena C)
   {
      nombre = E;
      edad = C;
   }
}
```

Esta clase pasará sin errores el análisis sintáctico, pues las construcciones son coherentes a nivel sintáctico. Sin embargo, a nivel semántico hay errores. El programador ha debido de equivocarse al teclear[57] y ha asignado la edad al nombre y el nombre a la edad. Un buen compilador debería

57 Seguramente debido a una pésima política de nomenclatura. ¡Hay que elegir buenos nombres de variables!

emitir dos mensajes como éstos:

```
Persona.leli (8,10): El parámetro "E" de tipo "Entero" no puede ser
                      asignado al atributo "nombre" de tipo "Cadena"
Persona.leli (9,8): El parámetro "C" de tipo "Cadena" no puede ser
                    asignado al atributo "edad" de tipo "Entero"
```

Estos dos errores se han producido por incompatibilidad de los tipos Entero y Cadena con respecto a la asignación. Los errores por incompatibilidad de tipos son tan frecuentes que tienen nombre propio; se les llama simplemente *errores de tipo*.

Aunque los errores de tipo sean los más frecuentes entre los errores semánticos estáticos, no son los únicos.

Errores de lectura/escritura: R-value y L-value

Errores de lectura/escritura de expresiones. Éstos errores se dan en las asignaciones. En una asignación, al elemento a la izquierda del operador de asignación se le "copia" el valor de la expresión de la derecha. La parte derecha de la asignación debe ser "legible", y la parte izquierda debe ser "asignable".

Diremos que las expresiones que pueden ser "legibles" tienen *R-value* (R de *Right*- derecha, porque pueden estar a la derecha de una asignación), mientras que las que pueden ser "asignables" tienen *L-value* (porque pueden estar a la izquierda, *Left*).

Un ejemplo de expresiones sin R-value son las llamadas a métodos vacíos (métodos que no devuelven ningún valor). Por ejemplo:

```
método mA () // método vacío
{ ... }

método mB ()
{
   Entero a = mA(); // es erróneo porque se requiere un R-value
                    // (además, los tipos no coinciden)
}
```

El ejemplo típico de expresiones sin L-value son los literales. No se puede encontrar un literal en la parte izquierda de una expresión. ¿Qué sentido tendría?

```
"Cadena1" = "Cadena2" ; // ??
```

Errores de ubicación

Son los errores que pueden producirse por una colocación incorrecta de alguna parte del lenguaje. LeLi carece de este tipo de errores, pero podemos encontrar algunos ejemplos en otros lenguajes. En el lenguaje C, por ejemplo, la instrucción `break` debe aparecer siempre dentro del cuerpo de un bucle o de un `switch`.

Errores de expresiones evaluables durante la compilación

Este error se da en las inicializaciones de atributos abstractos; un atributo abstracto no puede inicializarse a un valor que no pueda determinarse en tiempo de compilación:

```
clase Cualquiera
{
   atributo abstracto Cadena nombre = "Pedro"; // Correcto.
```

```
   atributo abstracto Entero valor = 1+2+nombre.longitud(); // Correcto.

   atributo Entero indice; // Atributo no abstracto

   atributo abstracto Entero indice2 = indice + 1; // Incorrecto.

}
```

La última declaración es incorrecta porque el valor de la variable `indice` no puede saberse en tiempo de ejecución.

En LeLi no hay tablas. En otros lenguajes donde sí se utilizan se buscan errores de expresiones constantes en el tamaño de dichas tablas, cuando no pueden inicializarse con un tamaño no constante.

Errores de contexto

El compilador debe tener en cuenta el contexto de las variables, de manera que no puedan referenciarse fuera de él. Éstos errores suelen producirse por identificadores mal escritos. En general, cualquier identificador puede producir un error de contexto.

```
1  método Booleano esCero(Entero a)
2  {
3     si(a==0)
4     {
5        Booleano result = cierto;
6     }
7     | otras
8     {
9        result = falso;
10    }
11    esCero = result;
12 }
```

Hay dos errores de contexto en las líneas 8 y 10; se hace referencia a la variable `result` cuando ésta ya se ha destruido (al cerrarse el cuerpo de la primera opción de la instrucción `si`). Por lo tanto deben emitirse dos errores.

Los errores de contexto son los únicos errores semánticos estáticos que pueden darse fuera de las expresiones: cualquier declaración de clase, variable, atributo o parámetro puede contener errores de contexto. Por ejemplo:

```
clase Persona {...}

clase Alumno extiende persona // error : persona está mal escrito
{
   atributo entero Edad; // error : la clase "entero" no existe (es "Entero")
   ...

   método hablar (persona P) // mismo error
   {
      persona M; // idem
   }
}
```

Los errores marcados aparecerán, claro está, si no se han definido las clases "`persona`" y "`entero`", con minúsculas.

7.1.2: Errores semánticos dinámicos

Son los errores que solamente pueden detectarse en tiempo de ejecución. En LeLi hay tres posibles errores semánticos en tiempo de ejecución:

- Desbordamiento numérico
- Que el método abstracto `Objeto.aserto` no se cumpla
- Que se haga una conversión de tipos (con `convertir`) descendente, y los tipos no sean adecuados.

7.1.3: Añadiendo información léxica a los ASTs

El problema

Cuando estudiamos en profundidad los ASTs en el tema 5 mencioné que la única información imprescindible que se guarda en los ASTs es "un texto" y "un tipo", accesibles respectivamente mediante los métodos `getText` y `getType` de la interfaz `antlr.collections.AST`. También hice hincapié en que aunque la clase `Token` disponga de métodos con nombres similares, no hay que confundirlas: aunque la mayoría de las instancias de Token será utilizada para iniciar un nodo AST (utilizando el método `AST.initialize(Token)`) no siempre será así; habrá tokens que no se utilizarán jamás para iniciar ASTs (típicamente los paréntesis y símbolos de separación) y habrá nodos AST cuyo tipo será imaginario, no siendo utilizado por token alguno (por ejemplo los ASTs que se utilizan para enraizar listas de tokens).

Los `Token`s, además, proporcionan información léxica (un nombre de fichero, una línea, una columna), la cual no es incluida en el interfaz `antlr.collections.AST.` , y no es implementada por `antlr.CommonAST`, el tipo que ANTLR utiliza por defecto para construir los árboles.

La información léxica (siempre que esté bien calculada) es un importante añadido a los mensajes de error; hasta tal punto de que su ausencia puede hacerlos inservibles. Sin embargo, al llegar al nivel semántico no podemos acceder a los tokens que la proporcionan. Para solucionarlo implementaremos una subclase de `antlr.CommonAST`, que sea capaz de guardar información léxica, y que sobreescriba el método initialize(Token) para copiar la información léxica; todos los ASTs que creemos serán instancias de dicha clase.

La clase antlraux.util.LexInfoAST

La nueva clase que vamos a utilizar se encontrará también en el paquete antlraux, dentro del

subpaquete `antlraux.util`. Al igual que a`ntlraux.util.LexInfoToken`, `LexInfoAST` implementará la interfaz `antlraux.util.LexInfo`[58].

Para que `LexInfoAST` funcione correctamente (muestre los nombres de fichero en los mensajes de error) será necesario que los tokens devueltos por el lexer contengan información sobre el nombre de fichero. Una de las opciones para lograr esto es utilizar `antlraux.util.LexInfoToken` como se sugirió en el capítulo 4.

De la misma manera que un analizador léxico puede generar Tokens homogéneos y heterogéneos diferentes de los tokens por defecto[59], los analizadores sintácticos permiten generar ASTs homogéneos y heterogéneos de clases diferentes a la ofrecida por defecto (`CommonAST`).

En nuestro caso, lo más sencillo será utilizar el método el método `setASTNodeType`[60] en el parser, ya sea el un parser con o sin capacidad de recuperación de errores sintácticos:

```
import antlraux.util.LexInfoAST;

...

Parser parser = new ... ;// crear el parser de la manera adecuada
parser.setFilename(nombreFichero);
parser.setASTNodeType("antlraux.util.LexInfoAST");
parser.programa(); // lanzar el reconocimiento
```

Dado que `LexInfoAST` implementa la interfaz AST, no será necesario modificar el código del parser para adaptarlo al nuevo tipo; basta con modificar un parámetro en tiempo de ejecución.

El método que se utiliza para iniciar los ASTs creados por un parser es por regla general `initialize(Token)`. Normalmente es conveniente que las clases de ASTs que sean generadas por un parser sobreescriban dicho método (como hace `LexInfoAST`).

De forma similar podemos cambiar el tipo de los ASTs generados por los iteradores de árboles que generen nuevos árboles. Por ejemplo:

```
import antlraux.util.LexInfoAST;

...

ast = parser.getAst(); // suponiendo que ast haya sido creado así

TreeParser treeParser = new ... ; // crear el treeParser
treeParser.setASTNodeType("antlraux.util.LexInfoAST");
treeParser.programa(ast);
```


Los ASTs creados por los iteradores de árboles son iniciados mediante el método `initialize(AST ast)`. Es conveniente que los ASTs que sean generados por un treeParser sobreescriban dicho método (como hace `LexInfoAST`).

Facilitándonos la labor en los iteradores de árboles: ASTLabelType

Frecuentemente tendremos que utilizar la información léxica añadida a nuestros ASTs en las acciones de los iteradores de árboles. Esta necesidad puede dar lugar a muchísimos *castings*, que

58 La interfaz antlraux.util.LexInfo se describe en la página 111.
59 Ver apartado "Tokens homogéneos y heterogéneos" en la página 110.
60 Ver sección 5.7.2 "ASTs heterogéneos", en la página 161.

harán el código bastante ilegible:

```
header { import antlraux.util.LexInfoAST; }

public class MiTreeParser extends TreeParser;
...
expresion: #(op:OP_MAS e1:expresion e2:expresion)
{
   LexInfoAST liop =(LexInfoAST)op;
   LexInfoAST lie1 =(LexInfoAST)e1;
   LexInfoAST lie2 =(LexInfoAST)e2;
  ... // Trabajar con liop, lie1 y lie2
}
```

Afortunadamente hay una manera de ahorrarse todos esos castings, consistente en utilizar la opción `ASTLabelType`, que realiza estos castings automáticamente:

```
header { import antlraux.util.LexInfoAST; }

public class MiTreeParser extends TreeParser;
options
{ ASTLabelType= "antlraux.util.LexInfoAST"; }
...

expresion : #(op:OP_MAS e1:expresion e2:expresion)
            {
                // Ahora op, e1 y e2 son de tipo LexInfoAST
            }
```

El único inconveniente de `ASTLabelType="claseAST"` solamente funcionará si todos los nodos del iterador de árboles son de la clase "claseAST" o de una de sus subclases (en otro caso se lanzará una `ClassCastException` antes o después).

Para nosotros esto no supondrá un problema pues todos los tipos de AST que utilizaremos pertenecerán al paquete `antlraux.context.asts`, y todos ellos son subclases de `LexInfoAST`.

Sección 7.2: Iterador simple de árboles

7.2.1: Estrategia de implementación del análisis semántico

Ya he comentado que el ciclo de análisis semántico se puede dividir en pequeños pasos diferenciados. Esta idea se entiende perfectamente con el siguiente dibujo (que ya he usado anteriormente):

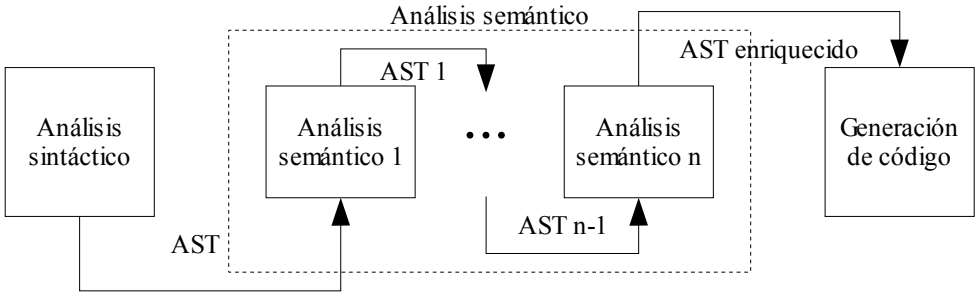

Ilustración 7.1 Análisis semántico dividido en subtareas

Hay dos maneras de implementar esta funcionalidad: las llamaremos forma "monolítica" y forma "en capas".

La forma monolítica consiste en ignorar totalmente la estructura del análisis semántico e implementar todo el análisis en una sola clase. Estaríamos implementando una clase que resolviera de una sola vez todo lo que hay dentro del rectángulo con líneas discontinuas de la figura. La principal ventaja de la estrategia monolítica es su velocidad: hay menos llamadas a métodos, pues hay menos análisis realizados sobre el AST. Su principal desventaja es que se generará un fichero muy grande, seguramente con partes de código repetidas, y con el cual será muy fácil cometer errores. En definitiva, será poco mantenible.

Por su parte, la aproximación en capas se basa en utilizar un analizador diferente para cada uno de los sub-análisis semánticos. Partiendo el análisis en varias partes potenciará la encapsulación, de manera que el conjunto será más mantenible[61]. Por otro lado, al basarse el sistema en lanzar varios análisis en lugar de 1, es muy probable que la velocidad sea menor que en el caso de la estrategia monolítica.

Dados nuestros fines didácticos, nos decantaremos indudablemente por la segunda estrategia, y dividiremos el análisis en varias capas. Téngase en mente que si lo que estuviéramos desarrollando fuera un producto comercial deberíamos considerar la estrategia monolítica, en aras de la eficiencia-ejecutar varias "pasadas" sobre el AST puede ser significativamente más lento que hacer solamente una.

Hay que resaltar también que hay ocasiones en las que simplemente no queda más opción que hacer múltiples pasadas; como veremos, esto será lo que ocurra a la hora de implementar el análisis de tipos de LeLi.

Para implementar los diversos analizadores semánticos que necesitaremos vamos a servirnos de nuevo de la herencia de gramáticas. Todos los sub-análisis actuarán sobre un AST más o menos parecido, de manera que, una vez especificado un esqueleto inicial, solamente tendremos que reescribir las reglas importantes de cada análisis.

Precisamente este esqueleto inicial será nuestro iterador simple de árboles, que llamaremos

61 Siempre y cuando las decisiones de diseño sean adecuadas, claro.

`LeLiTreeParser` y escribiremos en el fichero `LeLiTreeParser.g`. Los analizadores semánticos que implementaremos se organizarán en la siguiente jerarquía:

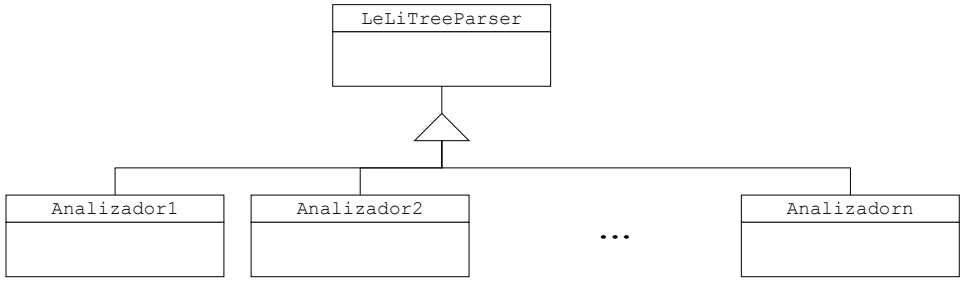

Ilustración 7.2 Esquema de clases para el análisis semántico

7.2.2: Estructura del fichero de gramática

El fichero `LeLiTreeParser.g` comenzará de la siguiente manera:

```
header {
   package leli;
}

class LeLiTreeParser extends TreeParser;

options
{
   k=1;
   buildAST = false; // Por defecto no construimos un AST nuevo
   importVocab=LeLiParserVocab;
}
```

`LeLiTreeParser` es un iterador de árboles, luego extenderá a `TreeParser`. En cuanto a las opciones, tendremos que:

- `k=1` .Esta opción es reduntante; por defecto k=1 en los analizadores semánticos.
- `buildAST=false`. Por defecto no construiremos un nuevo AST.
- `importVocab=LeLiParserVocab`. Importaremos el vocabulario que exportó en analizador sintáctico. Si hubiéramos tenido que añadir tokens en nuestro analizador con recuperación de errores tendríamos que importar su vocabulario.

7.2.3: Regla raíz: Programa

La regla raíz del analizador sigue siendo programa:

```
programa : #(PROGRAMA (decClase)+) ;
```

Un programa es un árbol con el token imaginario PROGRAMA como raíz y con 1 o más hijos que serán recorridos por la regla `decClase`.

7.2.4: Definición de clases

Una clase tendrá la siguiente forma:

```
decClase
   : #( RES_CLASE IDENT
```

```
        clausulaExtiende
        #(LISTA_MIEMBROS (decAtributo|decMetodo|decConstructor)*)
      )
  ;
```

El AST de una definición de clase tiene como raíz `RES_CLASE`. Su primer hijo es un `IDENT` (en el que se guarda el nombre de la clase) y su segundo hijo debe ser reconocido por la regla `clausulaExtiende`. El tercer hijo es un AST con raíz `LISTA_MIENBROS` y con de 0 a N hijos que son reconocidos por la regla `decAtributo`, `decMetodo` o `decConstructor`.

Hemos separado clausulaExtiende del resto de la definición de la clase para facilitar su reescritura en subgramáticas, por ejemplo para reconocer errores. La cláusula extiende siempre debe ser construida por el analizador sintáctico, teniendo la siguiente forma:

```
clausulaExtiende : #(RES_EXTIENDE IDENT);
```

7.2.5: Definición de atributos

Una definición de atributo puede ser de un atributo normal o abstracto. Gracias a que eliminamos el azúcar sintáctica, la regla para reconocer el AST de los atributos es muy sencilla:

```
decAtributo
  : #( RES_ATRIBUTO (RES_ABSTRACTO)? tipo IDENT (expresion)? )
  ;
```

No tenemos que preocuparnos por el hecho de que solamente los atributos abstractos pueden ser inicializados; esto ya se ha tenido en cuenta en la fase del análisis sintáctico.

`tipo` será así:

```
tipo : TIPO_ENTERO   // tipo Entero
     | TIPO_REAL     // tipo Real
     | TIPO_BOOLEANO // tipo Booleano
     | TIPO_CADENA   // tipo Cadena
     | IDENT         // tipo no básico
     | TIPO_VACIO  // tipo vacío
     ;
```

Nótese el cambio con respecto a la implementación de la regla tipo en el análisis sintáctico: en el análisis semántico, `TIPO_VACIO` forma parte de la regla `tipo`, de manera que para definir los tipos ya no hay dos reglas (`tipo` y `tipoRetorno`) sino una sola. Esto es posible porque ya se controló debidamente el uso del tipo vacío en el análisis sintáctico.

Esta "fusión" reducirá el número de reglas, haciendo el analizador más rápido y mantenible.

Definición de métodos normales, métodos abstractos y constructores

El AST de la definición de un método será reconocido por la regla `decMetodo`:

```
decMetodo
  : #( RES_METODO (RES_ABSTRACTO)? tipo
     IDENT
     listaDecParams
     listaInstrucciones
    )
  ;
```

Como puede verse, el nodo raíz tiene el tipo `RES_METODO`. El primer hijo sirve para indicar si el

método es abstracto o no (cuando no está). El tercer parámetro es el tipo del método, que como ya hemos dicho puede ser cualquier tipo, incluido `TIPO_VACIO`, y que ahora se reconoce con la regla `tipo`.

El siguiente hijo del AST es la lista de declaraciones de parámetros. Se reconoce con la siguiente regla:

```
listaDecParams : #(LISTA_DEC_PARAMS ( #(RES_PARAMETRO tipo IDENT) )* ) ;
```

Gracias de nuevo a la eliminación del azúcar sintáctica, reconocer las declaraciones de parámetros es muy simple: cada parámetro de la lista de parámetros tiene su propio AST, en el que se incluyen su nombre y su tipo.

El último hijo del AST se reconocerá con la regla listaInstrucciones, que tendrá la siguiente forma:

```
listaInstrucciones : #(LISTA_INSTRUCCIONES (instruccion)*) ;
```

Veremos más adelante la regla `instruccion`.

Por otro lado, la definición de un constructor tendrá la siguiente forma:

```
decConstructor : #( RES_CONSTRUCTOR listaDecParams listaInstrucciones) ;
```

7.2.6: Expresiones

A nivel semántico, las expresiones presentan menos dificultades que a nivel sintáctico: cuando el AST está bien construido, se utilizan diferentes tipos en la raíz de cada nodo, así que se prácticamente se puede recorre todo el AST de una expresión con una única regla:

```
expresion
   : #(OP_MAS           expresion expresion)
   | #(OP_MENOS         expresion expresion)
   | #(OP_ASIG          expresion expresion)
   | #(OP_O             expresion expresion)
   | #(OP_Y             expresion expresion)
   | #(OP_IGUAL         expresion expresion)
   | #(OP_DISTINTO      expresion expresion)
   | #(OP_MAYOR         expresion expresion)
   | #(OP_MENOR         expresion expresion)
   | #(OP_MAYOR_IGUAL   expresion expresion)
   | #(OP_MENOR_IGUAL   expresion expresion)
   | #(OP_PRODUCTO      expresion expresion)
   | #(OP_DIVISION      expresion expresion)
   | #(OP_MENOS_UNARIO expresion)
   | #(OP_MASMAS        expresion)
   | #(OP_MENOSMENOS    expresion)
   | #(OP_NO            expresion)
   | #(RES_ESUN         acceso tipo)
   | acceso
   ;
```

Solamente nos queda por definir `acceso`:

```
acceso : #(ACCESO raizAcceso (subAcceso)* );
```

Así que un acceso no es más que una raíz seguida de 0 o más `subAccesos`. No tenemos que preocuparnos de qué accesos tienen que ir seguidos obligatoriamente por al menos un acceso: ya nos encargamos de eso en la fase del análisis sintáctico. La raíz viene dada por las siguientes

reglas:

```
raizAcceso : IDENT
           | RES_PARAMETRO
           | RES_ATRIBUTO
           | RES_SUPER
           | literal
           | llamada
           | conversion
           | expresion // Expresiones entre paréntesis
           ;
```

Siendo `literal` la siguiente regla:

```
literal : LIT_ENTERO
        | LIT_REAL
        | LIT_CADENA
        | LIT_NL
        | LIT_TAB
        | LIT_COM
        | LIT_CIERTO
        | LIT_FALSO
        ;
```

, una `llamada` una invocación de un método o un constructor:

```
llamada : #(LLAMADA IDENT listaExpresiones )
        | #(RES_CONSTRUCTOR listaExpresiones )
        ;
listaExpresiones : #(LISTA_EXPRESIONES (expresion)* ) ;
```

y reconociéndose así una conversión de tipos:

```
conversion : #(RES_CONVERTIR expresion tipo) ;
```

Finalmente, la regla `subAcceso` tendrá esta forma:

```
subAcceso : llamada //Invocación de un método
          | IDENT   // Acceso a un atributo
          | RES_SUPER // Invocación de la clase padre
          ;
```

7.2.7: Instrucciones

Finalmente tenemos las instrucciones. Una instrucción puede ser de muchos tipos:

```
instruccion : instDecVar         // declaración
            | instExpresion      // Instrucción - expresión
            | instMientras       // bucle mientras
            | instHacerMientras  // bucle hacer-mientras
            | instDesde          // bucle desde
            | instSi             // Instrucción Si
            | instVolver         // Instrucción volver
            ;
```

Definir los diferentes tipos de instrucciones también es muy sencillo. La instrucción de declaración de variables tiene el tipo `INST_DEC_VAR`:

```
instDecVar : #(INST_DEC_VAR tipo IDENT (expresion)?) ;
```

La instrucción-expresión tiene `INST_EXPRESION`:

```
instExpresion : #(INST_EXPRESION expresion) ;
```

Los tres tipos de bucle se recorren también muy bien:

```
instMientras : #(RES_MIENTRAS expresion listaInstrucciones) ;

instHacerMientras : #(RES_HACER listaInstrucciones expresion) ;

instDesde : #(RES_DESDE listaExpresiones
                        listaExpresiones
                        listaExpresiones
                        listaInstrucciones )
          ;
```

La instrucción condicional es un poco más complicada, aunque no demasiado; un AST bien construido se puede recorrer fácilmente con un par de reglas:

```
instSi : #(RES_SI expresion listaInstrucciones (alternativasSi)*) ;

alternativasSi : #(BARRA_VERT expresion listaInstrucciones)
               | #(RES_OTRAS listaInstrucciones)
               ;
```

Por último, la más sencilla: la instrucción `volver`:

```
instVolver : RES_VOLVER ;
```

7.2.8: Código completo del iterador

El código completo del iterador es el siguiente:

```
header
{

package leli;

/*-------------------------------------------------*\
 | Un intérprete para un Lenguaje Limitado(LeLi) |
 | _____________________________________________ |
 |                                               |
 |              ANALISIS SEMÁNTICO 1             |
 | _____________________________________________ |
 |                                               |
 |            Enrique J. Garcia Cota             |
\*-------------------------------------------------*/

}

/**
 *  El objeto que permite recorrer los AST de LeLi.
 *  Hay que heredar de este objeto para hacer algo
 *  util (aparte de reconocer).
 */
class LeLiTreeParser extends TreeParser;

options
{
```

```
   importVocab=LeLiParserVocab;
   buildAST = false; // Por defecto no construimos un AST nuevo
}

/** Permite recorrer el nodo principal, llamado "programa" */
programa : #(PROGRAMA (decClase)+) ;

/**
 * Permite recorrer la definición de una clase. Una clase tiene:
 * - Un identificador
 * - Una OBLIGATORIA (se debe crear siempre en el parser) de "extiende"
 * - Una lista de atributos y métodos (miembros)
 */
decClase
   : #( RES_CLASE IDENT
        clausulaExtiende
          listaMiembros
      )
   ;

/** Lista de miembros de una clase **/
listaMiembros
   : #(LISTA_MIEMBROS (decAtributo|decMetodo|decConstructor)*)
   ;

/**
 * La palabra reservada "extiende" seguida de un identificador.
 * Debe sobreescribirse en las subgramáticas para añadir información
 * de contexto.
 **/
clausulaExtiende : #(RES_EXTIENDE IDENT);

/**
 * Esta regla permite recorrer los tipos.
 */
tipo : TIPO_ENTERO   // tipo Entero
     | TIPO_REAL     // tipo Real
     | TIPO_BOOLEANO // tipo Booleano
     | TIPO_CADENA   // tipo Cadena
     | TIPO_VACIO    // tipo vacío
     | IDENT         // tipo no básico
     ;
/**
 * Regla que recorre tanto los atributos abstractos como los
 * "normales"
 */
decAtributo
   : #( RES_ATRIBUTO (RES_ABSTRACTO)? tipo IDENT (expresion)? )
    ;

/**
 * Regla que recorre la definición de un método normal o abstracto
 */
decMetodo
   : #( RES_METODO (RES_ABSTRACTO)? tipo
      IDENT
      listaDecParams
      listaInstrucciones
```

```
      )
   ;

/**
 * Regla que recorre la definición de los constructores
 */
decConstructor
   : #( RES_CONSTRUCTOR listaDecParams listaInstrucciones )
   ;

/**
 * Declaración de los parámetros de un método
 */
listaDecParams : #(LISTA_DEC_PARAMS (decParametro)* ) ;

/** declaración de un parámetro **/
decParametro : #(RES_PARAMETRO tipo IDENT) ;

/**
 * Regla que permite reconocer todas las instrucciones
 */
expresion
   : #(OP_MAS          expresion expresion)
   | #(OP_MENOS        expresion expresion)
   | #(OP_ASIG         expresion expresion)
   | #(OP_O            expresion expresion)
   | #(OP_Y            expresion expresion)
   | #(OP_IGUAL        expresion expresion)
   | #(OP_DISTINTO     expresion expresion)
   | #(OP_MAYOR        expresion expresion)
   | #(OP_MENOR        expresion expresion)
   | #(OP_MAYOR_IGUAL  expresion expresion)
   | #(OP_MENOR_IGUAL  expresion expresion)
   | #(OP_PRODUCTO     expresion expresion)
   | #(OP_DIVISION     expresion expresion)
   | #(OP_MENOS_UNARIO expresion)
   | #(OP_MASMAS       expresion)
   | #(OP_MENOSMENOS   expresion)
   | #(OP_NO           expresion)
   | #(RES_ESUN        acceso tipo)
   | acceso
   ;

/** Regla que permite reconocer un acceso */
acceso : #(ACCESO raizAcceso (subAcceso)* );

/** Regla auxiliar que permite reconocer la raíz de un acceso */
raizAcceso : IDENT
           | RES_PARAMETRO
           | RES_ATRIBUTO
           | RES_SUPER
           | literal
           | llamada
           | conversion
           | expresion // expresiones con paréntesis
           ;

/** Regla auxiliar que permite reconocer los literales */
```

```
literal : LIT_ENTERO
        | LIT_REAL
        | LIT_CADENA
        | LIT_NL
        | LIT_TAB
        | LIT_COM
        | LIT_CIERTO
        | LIT_FALSO
        ;
/**
 * Permite recorrer una llamada a un método o un constructor
 */
llamada : #(LLAMADA IDENT listaExpresiones )
        | #(RES_CONSTRUCTOR listaExpresiones )
        ;

/**
 * Lista de expresiones. Sirve para representar parámetros pasados a
 * un método o un constructor
 */
listaExpresiones : #(LISTA_EXPRESIONES (expresion)* ) ;

/** Recorre una conversión de tipos */
conversion : #(RES_CONVERTIR expresion tipo) ;

/** Recorre un subacesso */
subAcceso : llamada
          | IDENT
          | RES_SUPER
          ;

/** Permite reconocer una lista de instrucciones */
listaInstrucciones : #(LISTA_INSTRUCCIONES (instruccion)*) ;

/** Regla que permite reconocer las instrucciones */
instruccion : instDecVar         // declaración
            | instExpresion      // Instrucción - expresión
            | instMientras       // bucle mientras
            | instHacerMientras  // bucle hacer-mientras
            | instDesde          // bucle desde
            | instSi             // Instrucción Si
            | instVolver         // Instrucción volver
            ;

/** Permite recorrer la declaración de una variable local */
instDecVar : #(INST_DEC_VAR tipo IDENT (expresion|listaExpresiones)? ) ;

/** Permite reconcer las "instrucciones-expresión" */
instExpresion : #(INST_EXPRESION expresion) ;

/** Permite reconocer los bucles "mientras" */
instMientras : #(RES_MIENTRAS expresion listaInstrucciones) ;

/** Permite recorrer los bucles "hacer-mientras" */
instHacerMientras : #(RES_HACER listaInstrucciones expresion) ;

/** Permite reconocer los bucles "desde" */
instDesde : #(RES_DESDE listaExpresiones
```

```
                        listaExpresiones
                        listaExpresiones
                        listaInstrucciones )
          ;

/**
 * Permite reconocer la instrucción Si.
 */
instSi : #(RES_SI expresion listaInstrucciones (alternativaSi)*) ;

/**
 * Regla auxiliar para reconocer las alternativas de la instrucción Si
 */
alternativaSi : #(BARRA_VERT expresion listaInstrucciones)
              | #(RES_OTRAS listaInstrucciones)
              ;

/** Reconoce la instrucción volver */
instVolver : RES_VOLVER ;
```

7.2.9: Los errores de construcción del AST

Recorrer un AST es un verdadero paseo comparado con analizar sintácticamente un flujo de tokens; esto es particularmente cierto si además el analizador sintáctico que ha construido el AST se ha esforzado adecuadamente en recuperarse de los errores.

No obstante, por mucho cuidado que hayamos tenido en la recuperación de errores sintácticos, es *imposible* que hayamos tenido en cuenta todos los errores que un programador pueda cometer (pues son virtualmente infinitos).

A pesar de todos nuestros esfuerzos en el capítulo 6, no está garantizado que los ASTs resultantes del análisis estén exentos de incoherencias: un error no detectado o no manejado convenientemente provocará la construcción de un AST defectuoso, que lanzará provocará el lanzamiento de errores en el análisis semántico. A fin de implementar un compilador realmente robusto, sería recomendable añadir recuperación de errores al iterador. No obstante, dado que ANTLR proporciona un mecanismo de recuperación de errores por defecto y que los errores de construcción del AST serán escasos, esto quedará como ejercicio para el lector.

Sección 7.3: El sistema Ámbito/Declaración/Tipo

7.3.1: Introducción

En todos los lenguajes de programación fuertemente tipados (definiremos más adelante qué es un lenguaje "fuertemente tipados") el nivel semántico tiene que hacer una serie de comprobaciones exhaustivas relativas a los tipos de las expresiones que se manejan en el código de entrada. Consideremos, por ejemplo, el siguiente código:

```
Entero a = 2;
Entero b = 1;
Booleano c = false;

a = a + b;
c = a > d;
```

Hay dos errores semánticos en el ejemplo. El primero se produce al declarar la variable booleana `c`, que se inicializa con el identificador "false" (un identificador desconocido en LeLi) en lugar de la cadena "falso" (que es la que se utiliza en LeLi para definir el valor "no cierto"). El segundo error está en la segunda asignación, pues se utiliza una variable llamada `d` que no ha sido declarada previamente.

En el estado actual, el compilador se limita a recorrer el árbol AST sin hacer nada; se limita a "aceptar" las construcciones que concuerdan sintácticamente con lo que él espera. Digamos que "asiente estúpidamente" a cualquier construcción que le proporcionemos, con tal de que sea sintácticamente correcta. Si pudiéramos ver lo que "piensa" el compilador, veríamos algo así.

```
...
Entero a=2;   <es un AST INST_DEC_VAR con Entero, el identificador a y 2. OK>
Entero b=1;   <es un AST INST_DEC_VAR con Entero, el identificador b y 1. OK>
Booleano c=false; <es un AST INST_DEC_VAR con Booleano, el identificador
                   c y el identificador "false". OK>

a=a+b;        <es un AST con los idents a y b y los oper. = y + . OK>
c=a>d;        <es un AST con los idents c, a y d y los oper. = y '>'. OK>
...
```

Este comportamiento es muy poco útil. Necesitamos hacer más inteligente a nuestro compilador, y que sea capaz de detectar los errores que hemos cometido, advirtiendo convenientemente al programador.

Ésto es lo que se persigue en el tratamiento de la información contextual, que nos disponemos a implementar. Pero para ello deberemos definir algunos conceptos previos, a lo que dedicaremos esta sección.

7.3.2: Presentación de el sistema Ámbito/Declaración/Tipo (ADT)

Evolución de la encapsulación en los lenguajes de programación

Los primeros lenguajes de programación se parecían mucho al lenguaje ensamblador. El programador debía conocer en profundidad todos los entresijos de la máquina, y saber qué bits se transmitirían entre qué componentes del procesador en cada instante. La primera directiva de control de flujo del programa fue `GOTO`, seguida un poco más tarde por `GOSUB`.

La segunda generación de lenguajes de programación introdujo una primera capa de abstracción.

Ya no era necesario sumar el registro AX con DX: se pasó a sumar el entero `cantidad` con el entero `incremento`. La orden `GOTO` se vio sustituida por las instrucciones de control condicionales y los bucles. Las subrutinas que se realizaban con `GOSUB` se fueron convirtiendo en funciones.

La tercera evolución de los lenguajes a llegado con la Programación Orientada a Objetos (POO). Los datos y funciones relacionados con un mismo ámbito se agruparon en objetos, convirtiéndose los datos en "atributos" y las funciones en "métodos" de dichos objetos.

Actualmente existe un cuarto nivel de abstracción, aunque en la mayoría de las aplicaciones actuales no se utiliza todavía. Estoy hablando de los Sistemas Distribuidos. Los Sistemas Distribuidos permiten al programador abstraerse de la red que sustenta su aplicación. Los objetos interaccionan entre ellos independientemente de la máquina en la que se encuentren, a través de la red. De esta forma, un Sistema Distribuido es más flexible que una aplicación monoestación.

Existen varios modelos para desarrollar Sistemas Distribuidos. Algunos ejemplos de tecnologías de Sistemas Distribuidos son CORBA, COM/DCOM de Microsoft™ y Enterprise JavaBeans de Sun™. La característica más interesante de COM/DCOM es que los atributos de un objeto son inaccesibles desde el exterior (si no es a través de métodos) y los métodos están agrupados en *interfaces*, que no son mas que grupos de métodos. Así, para poder ejecutar un método en un objeto con COM/DCOM es necesario obtener su interfaz, que es el "puente" que permite trabajar con el método de manera transparente con la red.

La evolución de los lenguajes de programación a lo largo del tiempo permite sacar muchas conclusiones. La que nos interesa aquí, sin embargo, es el incremento cada vez mayor de la *encapsulación*. Yo definiría la encapsulación como la "ocultación pertinente de los datos", es decir, que en cada unidad lógica del programa se "restringe" al programador el acceso a ciertos datos que no debería utilizar. Algunos programadores (en general inexpertos) ven la encapsulación como poco más que "manía incordiante" de algunos por hacer las cosas más "formales"[62]. Sin embargo, el tiempo y la experiencia han demostrado que la encapsulación es, simplemente, el camino a seguir. Con encapsulación se consiguen aplicaciones más flexibles y mantenibles, y se reducen los errores.

En los primeros programas hechos en ensamblador no había encapsulación; el programador podía acceder a cualquier dirección de la memoria de datos. Más adelante, en los lenguajes funcionales, solamente le estaba permitido acceder a las variables globales del programa y a las variables locales de la función que se está ejecutando. La POO añadía aún más encapsulación, introduciendo los atributos privados y protegidos (que son los que ponen tan nerviosos a los programadores inexpertos de los que hablábamos antes). Por último, en los Sistemas Distribuidos, la encapsulación es tan patente que incluso los métodos de un objeto no son directamente accesibles (es necesario obtener un interfaz antes).

Ámbitos

Tras esta pequeña disertación sobre encapsulación, vamos a presentar el mecanismo que se utiliza habitualmente para implementarla: los *ámbitos*.

Un ámbito puede entenderse como un conjunto de identificadores que pueden ser utilizados por el programador en un punto dado del programa. Los identificadores pueden servir para denominar tanto tipos de objetos (clases) como instancias de objetos (atributos, variables locales, parámetros) o llamadas a métodos.

A grandes rasgos, cada vez que el compilador encuentre un identificador en la entrada, deberá

62 A mí mismo me pasó.

consultar el sistema de ámbitos para obtener el tipo asociado a dicho identificador. Una vez obtenido dicho tipo, se pondrá en marcha el sistema de comprobación de tipos (ver más adelante).

Los ámbitos suelen "solaparse"; cada nuevo ámbito añade identificadores al ámbito que lo contiene; también puede enmascarar los identificadores previos con otros nuevos. Considérese el siguiente caso:

```
clase Solapamiento          // Comienza el ámbito de la clase "Solapamiento"
{
   atributo Cadena a;       // Declara el ATRIBUTO a
   método solapa(Entero a) // Comienza el ámbito del método "solapa"
   {
      Sistema.imprime(a);   // Imprime el valor del PARÁMETRO a
      Booleano a = falso;
      Sistema.imprime(a);   // Imprime el valor de la VARIABLE a "falso"
   }                        // Termina el ámbito del método
}                           // Termina el ámbito de la clase
```

El comportamiento general de los ámbitos se puede modelar con una "pila de tablas de nombres". Añadir elementos a la pila consiste, pues, en apilar y desapilar tablas de nombres.

Normalmente la pila no tendrá más de 3 niveles:

- El ámbito General, que incluye las clases `Sistema` y `Objeto`, los tipos básicos del lenguaje y los tipos del usuario
- El ámbito Clase, que añade los nombres de los atributos de la clase en los métodos de la clase.
- El ámbito Método, que añade los parámetros y la variable de retorno al cuerpo del método.
- Puede haber ámbitos adicionales para bucles, instrucciones condicionales, etc.

En general, cada vez que en LeLi se utiliza una llave abierta ('{') se abre un nuevo ámbito, que se cierra al llegar a la primera llave cerrada ('}'). Cuando se declara un objeto se inserta en el último ámbito abierto. Las dos excepciones a esta regla son el ámbito global, que no comienza ni termina con llaves, y los parámetros en los métodos, que a pesar de insertarse en el ámbito de su método preceden a la llave abierta.

En casi todos los lenguajes existe un ámbito por defecto, en el que se definen ciertos elementos básicos del lenguaje. En LeLi, el ámbito por defecto se llama "ámbito global", y contiene los tipos básicos (`Entero`, `Real`, `Cadena`, `Booleano`, `Objeto`) y la clase `Sistema`.

Los ámbitos se implementarán utilizando la clase `antlraux.context.Scope`, que estudaremos en breve.

Declaraciones

Las declaraciones son las unidades mínimas de información que guarda un ámbito. Cada vez que una clase, método, atributo, parámetro o variable se declare, deberemos introducir una declaración en el ámbito correspondiente.

Una declaración puede entenderse como un conjunto de datos. El más evidente de todos es el *nombre*: toda declaración tiene un nombre que la identifica y diferencia de las demás[63]. Otro dato relevante es el *tipo* de la declaración.

Para implementar las declaraciones utilizaremos la clase `antlraux.context.Declaration`, que

63 Como veremos en la siguiente sección, esto no es del todo cierto; hay casos en los que es necesario y deseable que varias declaraciones tengan el mismo nombre.

también estudiaremos más adelante.

Tipos

El tipo de una declaración es, como ya hemos dicho, su característica principal. Es una manera de "clasificar" o "catalogar" los objetos, de acuerdo con las propiedades y atributos que tienen, o los métodos que pueden invocar. Intuitivamente, podemos decir que una declaración es de tipo "entero", "real", etc.

Las declaraciones no son los únicos elementos con tipo de un lenguaje. También tienen tipo las expresiones (de forma que una expresión puede ser "entera", "real", etc). Los tipos de las expresiones se calculan de forma recursiva: los accesos simples ("nombres de objetos") cuyo tipo puede calcularse simplemente consultando la declaración adecuada en el ámbito actual. Después se siguen unas reglas determinadas: la suma de dos expresiones enteras es también entera, etc.

Al proceso de calcular los tipos de las expresiones de un programa durante la fase de compilación se le llama *cálculo de tipos*. Durante el cálculo de tipos se deben realizar muchas comprobaciones de operaciones inválidas (por ejemplo restar un entero a una cadena) por lo que en muchas ocasiones al cálculo de tipos se le llama *comprobación de tipos.*

A los lenguajes que realizan la comprobación de tipos durante la fase de compilación se les llama lenguajes *fuertemente tipados.* A los que la realizan en tiempo de ejecución, se les llama *débilmente tipados.*

Implementaremos nuestro sistema de tipos en torno a una serie de interfaces y clases localizadas en el subpaquete, `antlraux.context.types`. La interfaz más importante de este paquete será `antlraux.context.types.Type`.

ASTs especializados

En la página 161 ya estuvimos hablando de los ASTs heterogéneos. En la sección 7.1.3 presentamos el primer tipo de AST que vamos a utilizar en nuestro compilador, `LexInfoAST` – un AST con información léxica.

Durante el análisis semántico utilizaremos otras clases de ASTs para guardar información adicional – los veremos más adelante.

7.3.3: Implementación del sistema ADT: el paquete antlraux.context

Los conceptos básicos del sistema ADT estarán implementados utilizando clases del subpaquete de antlraux llamado `antlraux.context`. Otras clases, sin embargo, serán "dependientes del lenguaje LeLi", por lo que se incluirán en el paquete leli algún subpaquetes de éste.

En los siguientes apartados iremos viendo paulatinamente cómo implementaremos los ámbitos, declaraciones y tipos en nuestro compilador, y presentando las soluciones de carácter general que ofrece la librería antlraux. Pero antes de eso vamos a presentar la clase ContextException, que modeliza las excepciones debidas a errores detectados durante la etapa de comprobación de tipos.

La excepción antlraux.context.ContextException

Cada vez que se produzca un error durante el TIC se lanzará una excepción de tipo `antlraux.context.ContextException`.

La clase `antlraux.context.ContextException` es una subclase de `antlaux.util.LexInfoException`. Esta última es, a su vez, subclase de

`antlr.RecognitionException`. Por tanto, `ContextException` es una "sub-sub clase" de `RecognitionException`.

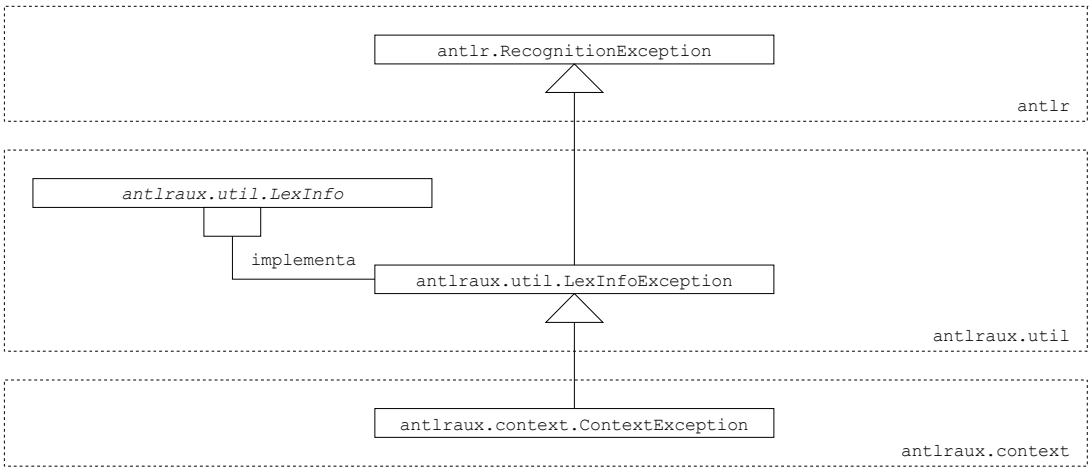

Ilustración 7.3 Jerarquía de antlraux.context.ContextException

Esta dependencia no es al azar: el código de recuperación de errores de los analizadores semánticos de ANTLR se basa en capturar las excepciones de tipo `RecognitionException`; lanzando excepciones de este tipo conseguiremos que los analizadores puedan manejar las excepciones debidas a errores de contexto, si éstas se producen.

`ContextException` implementa la interfaz `antlraux.util.LexInfo`, por lo que puede utilizarse para generar mensajes de error con información léxica.

Un buen sistema de identificación : la dupla nombre-"etiqueta"

Cuando Terence Parr diseñó la interfaz AST, decidió que todo nodo debería proporcionar dos informaciones: una cadena, "text" y un entero, "type".El primero contiene en la mayoría de los casos "el texto que formaba el token". El segundo sirve para diferenciar entre sí tokens implementados mediante la misma clase[64].

Este sistema de identificación es muy potente y flexible, hasta el punto de que lo usaremos para identificar los ámbitos, declaraciones y tipos de nuestro sistema.

No obstante, existe un pequeño problema con la nomenclatura utilizada: "type" es un nombre genérico que puede llevar a confusión; hemos de diferenciar entre "tipo de un AST" y "tipo de una declaración o expresión"; mientras que el primero no es más que un entero, que se establece en el nivel sintáctico, el segundo puede ser la instancia de una clase más o menos complicada, que se establece en el nivel semántico.

A fin de ahorrar confusiones futuras, he decidido utilizar un nombre diferente de "type" para designar a "los enteros que codifican un tipo dentro de una clase": utilizaré el término inglés*tag*, que viene a significar "etiqueta". Los ámbitos, declaraciones y tipos serán pues idenficables con la dupla `<tag,name>`. La etiqueta tendrá diversas funciones en cada uno: en los ámbitos codificará "el tipo de ámbitos" (de clase, de método, de condicional...); en las declaraciones, el "tipo de declaración" (clase, parámetro, variable...); en los tipos, el "tipo de tipo" (error, vacío, básico, definido por el usuario, método, constructor, clase, atributo, metaclase).

De esta manera, tanto los ámbitos como las declaraciones y los tipos poseerán *setters* y *getters*

64 O "tokens homogéneos". Fueron explicados en la sección 4.6.2, en la página 110.

para un atributo entero llamado *tag*, que tendrá una utilidad parecida al método *type* de los ASTs.

7.3.4: La clase antlraux.context.Scope

Esta clase (o una subclase de ella) será la que utilicemos para modelar los ámbitos.

Identificación: nombre y etiqueta de los ámbitos

Utilizaremos el sistema de identificación nombre-etiqueta que veíamos anteriormente. Así, toda instancia de la clase `antlraux.context.Scope` deberá poseer, al menos, un nombre (`name`) y una etiqueta (`tag`) que especifique su "tipo de ámbito". `name` es de tipo `String`, y `tag` es un entero. Los dos campos serán completamente dependientes del lenguaje que se esté modelando.

Puede utilizarse cualquier dupla <etiqueta, nombre> para identificar un ámbito; la clase no hace ninguna suposición con respecto a ellos.

La principal utilidad del nombre de un ámbito es de cara a los mensajes de error; es muy útil poder indicar el nombre de un ámbito en el caso de que se haya producido un error de inserción.

En cuanto a la etiqueta, su principal función es la de "chequeo de inserciones": un atributo de una clase no puede ser declarado dentro del ámbito de un método. La etiqueta también sirve para comprobar que estamos cerrando el ámbito adecuado; como veremos más adelante, la clase `LeLiContext` se sirve de la etiqueta para cerrar de la manera adecuada los diferentes tipos de ámbitos que puede encontrar.

Tanto `name` como `tag` tienen métodos get y set. Ambos pueden ser null, aunque es desaconsejable.

Inserción de declaraciones

Un único método permite insertar declaraciones en un ámbito: el método `insert`.

```
public void insert(Declaration d) throws ContextException
```

Este método insertará automáticamente la declaración d en el conjunto de declaraciones ámbito actual, cuya implementación se verá más adelante. La implementación actual no lanzará jamás una excepción; ésta se incluye en la definición del método para que las subclases puedan lanzarla si sobreescriben el método.

Implementación implícita de la pila de ámbitos y búsquedas locales y no locales

Los ámbitos suelen organizarse en "pilas", tal y como se vio en la sección anterior. Hay varias maneras de implementar este comportamiento; la opción por la que nos hemos decantado ha sido la de implementar la pila implícitamente en los ámbitos: en lugar de haber una clase "pila de ámbitos", se ha añadido un campo a los ámbitos que les permite saber qué ámbito es su "padre".

El campo que apunta al ámbito padre es de tipo `antlraux.context.Scope` y posee métodos get y set. Puede ser null, si el ámbito carece de ámbito padre (el ámbito es "global").

Un ámbito ofrecerá normalmente dos tipos de métodos de búsqueda: uno "local", para buscar en la tabla de símbolos propia del ámbito, y otro "no local", para ir ascendiendo por la jerarquía de ámbitos: primero se busca en el ámbito actual, luego en el padre, etc, deteniéndose la búsqueda al llegar al ámbito raíz.

Las referencias circulares no se detectan: si un ámbito se hace padre de uno de sus padres, invocar una búsqueda no local provocará una iteración infinita con el consiguiente

desbordamiento de pila.

Organización de las declaraciones dentro de un ámbito

Ya hemos comentado la idea básica sobre la que giran los ámbitos: un conjunto de declaraciones con nombre, presumiblemente organizadas en forma de tabla hash, siendo la clave hash el nombre de las declaraciones.

Lo más inmediato es utilizar la clase `Hashtable` proporcionada por la API de java.

Uno de los primeros problemas que encontraremos para utilizar dicha clase es que no permite la inserción de varios elementos con el mismo nombre. Esto puede ser permisible en lenguajes simples (en los que se establezca que un identificador sea utilizado para declarar un y solo un objeto de cada ámbito). Sin embargo en la definición de LeLi se estableció explícitamente que habría situaciones en las que varios objetos estarían identificados con el mismo nombre: concretamente en el caso de enmascaramiento podría haber un parámetro, un atributo y una variable (o más de una) identificados con el mismo nombre:

```
clase enmascaramiento
{
   atributo Cadena a;

   metodo enmascarar (Entero a, Entero b)
   {
      Real a = parámetro.a + b + 1.5;
   }
}
```

En el ejemplo anterior, el ámbito del método enmascarar contiene un parámetro de tipo `Entero` llamado `a` y una variable de tipo `Real` también llamada `a` (además, los dos están enmascarando un atributo `Cadena` que también se llama `a`, aunque en dicho caso no hay problema porque el atributo está declarado en el ámbito de la clase y no del método).

La conclusión de todo esto es que nuestra clase ámbito poseerá una tabla hash que en cada posición no tendrá una sola declaración, sino una lista de declaraciones. De esta forma, los métodos de búsqueda de declaraciones utilizando un nombre no devolverán una sola declaración, sino una lista de declaraciones.

Esta organización puede verse reflejada en la ilustración 7.4:

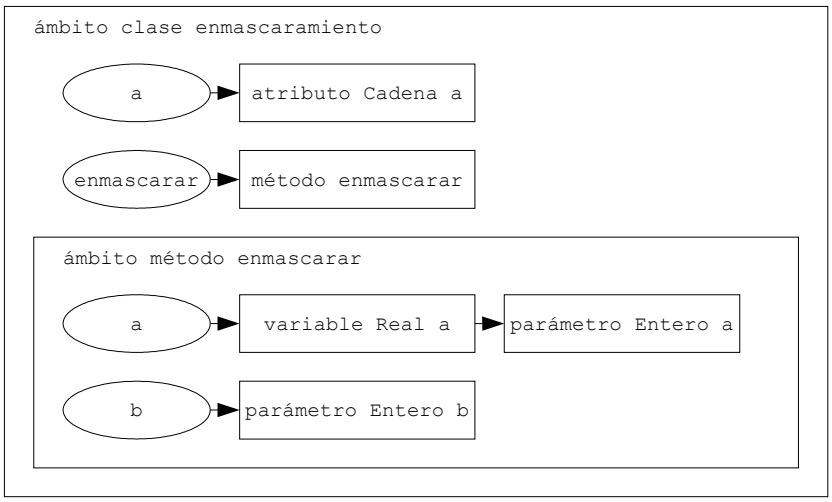

Ilustración 7.4 Ámbitos y enmascaramiento

Los métodos de búsqueda se verán afectados por esta organización, devolviendo listas de declaraciones en lugar de una sola declaración:

```
// Devuelve una lista de declaraciones, o null si no se encuentra ninguna
public LinkedList searchAllLocally(String name);

// Invoca searchLocally en toda la jerarquía de ámbitos; devuelve una lista con
// todas las declaraciones llamadas "name" o null si no se ha encontrado
// ninguna
public LinkedList searchAll(String name);
```

Por comodidad se han incluido otros dos, que devuelven directamente el primer elemento de la lista (que será el último insertado en el ámbito):

```
// Devuelve la primera declaración llamada "name",
// o null si no se encuentra ninguna
public Declaration searchFirstLocally(String name);

// Invoca searchFirstLocally en toda la jerarquía de ámbitos;
// devuelve la primera declaración encontrada en la jerarquía
// ninguna
public Declaration searchFirst(String name);
```

El doble ordenamiento de los elementos

En muchas ocasiones necesitaremos buscar "todas las declaraciones llamadas x" en un ámbito. Sin embargo, la búsqueda "por nombre" no será el único tipo de búsqueda que realizaremos; en otras ocasiones necesitaremos hacer una búsqueda "por etiqueta" de declaración; por ejemplo, necesitaremos obtener todos los métodos de una clase, o todos sus atributos.

Para permitir este tipo de búsquedas hemos organizado las declaraciones dentro de los ámbitos de manera que se puedan realizar los dos tipos de búsquedas (utilizando dos tablas hash diferentes, una que "ordena por nombre" y otra que "ordena por etiqueta").

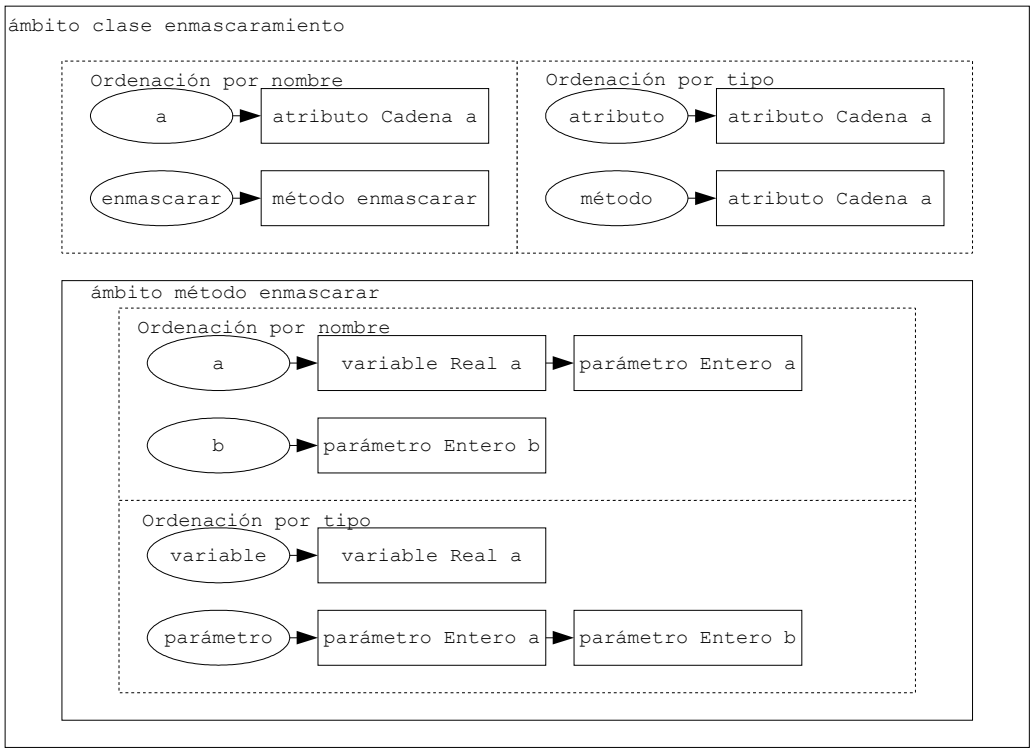

Ilustración 7.5 Doble ordenamiento de las declaraciones en los ámbitos

El principal problema de esta implementación es un incremento en la memoria consumida y un decremento en la velocidad de inserción.

```
// Devuelve una lista de declaraciones, o null si no se encuentra ninguna
public LinkedList searchAllLocally(int tag);

// Invoca searchLocally en toda la jerarquía de ámbitos; devuelve una lista con
// todas las declaraciones de etiqueta "tag" o null si no se ha encontrado
// ninguna
public LinkedList searchAll(int tag);

// Devuelve la primera declaración con etiqueta "tag",
// o null si no se encuentra ninguna
public Declaration searchFirstLocally(int tag);

// Invoca searchFirstLocally en toda la jerarquía de ámbitos;
// devuelve la primera declaración encontrada en la jerarquía
// ninguna
public Declaration searchFirst(int tag);
```

Además de estos métodos y los ya expuestos, existen otros que permiten realizar consultas en las que se combinan el tipo y el nombre del elemento buscado:

```
public LinkedList searchAllLocally(int tag, String name);
public LinkedList searchAll(int tag, String name);
public Declaration searchFirstLocally(int tag, String name);
public Declaration searchFirst(int tag, String name);
```

7.3.5: La clase antlraux.context.Declaration

Elementos básicos: nombre y tipo

En la sección anterior ya se mencionaron los elementos básicos que conforman una declaración: ésta debe poseer, al menos, un nombre (*name*) y un tipo (*type*). El primero es una simple cadena, mientras que el segundo es una instancia de la interfaz `antlraux.types.Type`, que veremos con más detenimiento más abajo.

Tanto el nombre como el tipo de una declaración tienen métodos get y set.

Identificación: Etiqueta y nombre

Las declaraciones poseen un campo "etiqueta" (*tag*) que sirve para indicar "el tipo de declaración". En oposición al "tipo del objeto declarado", anteriormente descrito, que indica si el objeto declarado es un "entero" o un "real", la etiqueta sirve para diferenciar si una declaración es "un parámetro", "una declaración de clase" o "un método".

La etiqueta es un atributo entero con métodos get y set.

En nuestro caso ha sido posible utilizar enteros ya definidos por los analizadores de nuestro compilador; aunque esto no es necesario en las etiquetas, es altamente recomendable, pues mantiene la homogeneidad del sistema.

La etiqueta es completada con el *nombre de la declaración*: una cadena que guarda un nombre. Por ejemplo, cuando se declara la clase Persona, su dupla `<etiqueta,nombre>` es `<RES_CLASE, "Persona">`.

Inicialización

Muchas declaraciones requieren guardar una expresión de inicialización. En LeLi, por ejemplo, los atributos abstractos y las variables pueden ser inicializadas con una expresión:

```
clase Persona
{
   atributo abstracto Entero mayoríaEdad = 18;
   método cualquiera()
   {
      Cadena mensaje = "Hola mundo";
   }
}
```

La implementación de Declaration prevé esta posibilidad y ofrece un atributo, llamado `initialValue`, para guardar dicha expresión. `initialValue` es de tipo AST, así que admite cualquier tipo de AST que se esté utilizando para representar las expresiones.

`initialValue` posee métodos get y set, y puede ser null.

Otros campos

La mayoría de las declaraciones serán creadas a partir de un nodo AST. Adicionalmente a los campos ya mostrados, las declaraciones ofrecen un campo llamado `ast` en el que se puede guardar una referencia a dicho AST, de manera que esté disponible en caso de necesidad. `ast` es de tipo AST y posee métodos get y set; puede ser null.

Finalmente, `Declaration` implementa la interfaz `antlraux.util.LexInfo`, así que ofrece los atributos `filename`, `line` y `column` habituales, con sus respectivos métodos get y set, así como

los métodos `copyLexInfo(LexInfo)` y `getLexInfoString`.

7.3.6: El sistema de tipos (antlraux.context.types.*)

La librería antlraux proporciona diversas clases e interfaces para trabajar con los tipos de un lenguaje; todas ellas se encuentran en el subpaquete `antlraux.context.types`.

Idenfificación: la interfaz antlraux.contex.types.Type

Todas las clases e interfaces relacionadas con el sistema de tipos en antlraux cumplirán la interfaz `antlraux.context.types.Type`. Para cumplir esta interfaz basta con implementar los métodos get y set de los identificadores habituales, es decir, un nombre (una cadena llamada `name`) y una etiqueta(un entero llamado `tag`). Es decir, que como ya habíamos advertido, la pareja identificadora <nombre,etiqueta> vuelve a utilizarse.

El sistema de tipos ofrece una clase, llamada `CommonType`, que lo implementa, utilizando una cadena (`String`) y un entero. Esta clase se ofrece por comodidad; los tipos pueden heredarse de `CommonType` o implementar ellos mismos la interfaz `Type`.

Otras interfaces y clases interesantes del sistema de tipos

Además de la interfaz `Type`, el sistema de tipos ofrece otras interfaces para modelar los tipos; todas ellas heredan de `Type`, y cada una tiene utilidades concretas:

- `antlraux.types.ModifiedType`: Permite añadir "modificadores" (como public, private o static) a un tipo. Para ello ofrece los métodos `addModifier(String)` y `boolean hasModifier(String)`. Ofrece otros para recorrer toda la lista de modificadores de un tipo, pero estos dos son los principales.
- `antlraux.types.HeritableType`: Sirve para modelar la relación tipo-supertipo comúnmente llamada "isA" (esUn). El único método de esta clase es el método `boolean isA (HeritableType)`.
- `antlraux.types.SimpleInheritanceType`: Hereda de `HeritableType`. Modela los tipos con herencia simple, es decir, aquellos tipos que solamente pueden tener una superclase, como en java. Sus métodos son `getSuperType()` y `setSuperType(SimpleInheritanceType)`.
- `antlraux.types.ScopedType`: Modela los tipos que tienen un ámbito (una instancia de antlraux.context.Scope) como los tipos de las clases y métodos (que expondremos en los siguentes subapartados). Los métodos de esta interfaz son `Scope getScope()` y `setScope (Scope)`.
- `antlraux.types.ClassMemberType`: Modela tipos especiales que se encuentran dentro de una clase. Más en el siguiente sub apartado.

Estas interfaces están pensadas para dotar de cierta homogeneidad al sistema de tipos; solamente he incluído en antlraux las que me han hecho falta para modelar el lenguaje LeLi; para ser verdaderamente genéricos habría que introducir muchas más interfaces, pero habrá que arreglarse con lo que hay.

Por comodidad he introducido algunas clases auxiliares, que implementan algunos de éstos interfaces:

- `antlraux.types.CommonType` implementa, como ya hemos dicho, la interfaz `Type`.
- `antlraux.types.DeclaredType` es una subclase de `CommonType` que implementa la interfaz `ModifiedType`, de manera que es un tipo que admite nombre, etiqueta y una lista de

modificadores (público, privado, etc).

Además de estas dos clases tendremos la clase `ClassType` y `MethodType`, que veremos en los siguientes apartados.

La variedad de tipos en los lenguajes de programación es muy amplia, y los tipos ofrecidos en el paquete `antlraux.context.types.*` son simplemente *ejemplos*. Frecuentemente no bastarán para modelar los tipos de un lenguaje, y será necesario crear nuevas clases.

Por último, presentar la excepción `antlraux.types.TypeException`. Esta subclase de `antlraux.ContextException` servirá para modelar los errores ocurridos en el sistema de tipos.

Tipos "especiales"

Los programadores solemos tener un concepto bastante intuitivo de lo que es un "tipo". Intuitivamente sabemos que los tipos están presentes en dos sitios:

- cada expresión (y cada una de sus subexpresiones) tiene un tipo.
- Se pueden "declarar" nuevos elementos en un ámbito; para eso están las instrucciones de declaración y las declaraciones de ámbitos. Por ejemplo:

```
método miMétodo(Entero a) // Declara el parámetro a
{
   Entero b = a + 1; // Declara la variable b
}
```

Estos tipos son los "usuales": se utilizan en las declaraciones que se realizan dentro del ámbito de un método, un bucle o una instrucción condicional. Es decir, "caracterizan" a variables y parámetros dentro un método. Son los tipos que se reconocen mediante la regla `tipo` de nuestro lenguaje (es decir, `Entero`, `Booleano`, `Objeto`, etc)

Esta concepción "intuitiva" de los tipos puede llegar a errores, porque confunde un "tipo" con "un tipo del lenguaje del compilador". La equivocación está provocada por el hecho de que los tipos de las declaraciones que se insertan en el ámbito de un método "coinciden" con los tipos del lenguaje LeLi.

El esquema ámbito/declaración/tipo no solamente funciona a nivel de métodos. En otras palabras,

Los parámetros y variables no son los únicos que provocan la inserción de declaraciones en un ámbito. Las *clases* también son declaradas e insertadas en un ámbito; *los métodos, constructores* y *atributos* de dichas clases también son declaraciones dentro de un ámbito y requieren un tipo.

Las declaraciones de clases, métodos, constructores y atributos tendrán, por tanto, sus propios "tipos", que poco tendrán que ver con los "tipos del lenguaje".

Vamos a analizar cada uno de estos diferentes "tipos especiales".

El tipo de un método: antlraux.context.types.MethodType

En este subapartado vamos a tratar de resolver algunos interrogantes que giran en torno a los métodos, a saber:

- ¿Cómo se integran los métodos y constructores de una clase con el sistema ámbito/declaración/tipo?
- ¿Cómo modelar el polimorfismo?

- ¿Cuáles son las restricciones de inserción de un método o constructor en el ámbito de una clase? Es decir, ¿cuándo se considerará que un método "no se puede insertar en el ámbito de una clase"?

Resolveremos aquí las dos primera cuestiones, y en el siguiente subapartado (compatibilidad de tipos) las tercera.

Los métodos y constructores de una clase deben introducirse en el ámbito de una clase, en forma de declaraciones, de forma parecida a los atributos. Por lo tanto, si somos capaces de especificar cómo serán la instancias de `Declaration` que modelarán las declaraciones de los métodos y constructores habremos resuelto el problema.

El nombre de las declaraciones será el nombre del método para los métodos, y la cadena "constructor" para los constructores. Es decir, si hay varios métodos con el mismo nombre (se usa el polimorfismo) serán insertados en la misma lista de declaraciones; de la misma forma, todos los constructores irán insertados en una lista llamada "constructor". Así es como el polimorfismo tiene cabida en el sistema ámbito/declaración/tipo.

La etiqueta de las declaraciones también será sencilla: `RES_METODO` para los métodos y `RES_CONSTRUCTOR` para los constructores.

El mayor escollo que encontraremos será modelar el tipo de las declaraciones. El tipo de la declaración debe servir para diferenciar los métodos con el mismo nombre, o los diferentes constructores, entre sí[65].

La manera estandarizada de diferenciar los métodos entre sí, cuando tienen el mismo nombre, se basa en la lista de parámetros: dos declaraciones de métodos son diferentes si no tienen el mismo número de parámetros, o si dichos parámetros no son todos del mismo tipo[66].

El tipo que utilizaremos para este menester estará modelado por la clase `antlraux.context.types.MethodType`. Es una subclase de DeclaratedType, luego admite nombre, etiqueta y una lista de modificadores.

`MethodType` implementa `ScopedType`, así que se puede obtener el ámbito del método representado por el tipo.

Otro atributo esencial de `MethodType` es `classType`, que es una referencia al tipo de la clase en la que se definió el método (ver la clase `ClassType` en el siguiente subapartado). `MethodType` tiene métodos get y set para este atributo.

Sin duda el campo más importante de `MethodType` es `params`; es una lista enlazada de tipos (objetos que cumplen la interfaz `Type`). Está protegida, pero se puede acceder en lectura utilizando el método `Enumeration params()`. Además están los métodos `addParam(Type)` y `clearParams()` para modificarla.

El último atributo de `MethodType` es el tipo de retorno, modelado mediante el atributo `returnType`, que puede ser cualquier objeto que cumpla la interfaz `Type`. Este atributo también puede ser accedido con métodos get y set.

`MethodType` cuenta con un constructor en el que se pueden especificar etiqueta, nombre, tipo de retorno, clase y ámbito del método:

65 Y no, no vale el tipo de retorno de cada método para eso. Los tipos de las declaraciones tienen que ser *diferenciadores*.

66 Cada dos tipos de parámetros, además de no ser iguales, tampoco pueden ser subclases el uno del otro; aunque eso depende del lenguaje.

```
public MethodType( int tag, String Name, Type returnType,
                   ClassType classType, Scope s )
```

Veamos un ejemplo. El método `imprime` de la clase `Sistema` (uno de ellos) tiene la siguiente declaración:

```
clase Sistema
{
   método abstracto imprime(Cadena mensaje) {...}
}
```

Si utilizáramos la clase `MethodType` para modelar el tipo de este método[67], deberíamos hacer lo siguiente:

```
// obtenemos el tipo de la clase Sistema
ClassType tipoSistema = ... ;

// obtenemos el ámbito del método imprime
MethodScope ms = ... ;

// Creamos el tipo de método
MethodType mt =
   new MethodType(RES_METODO, "imprime", obtenerTipoVacio(), tipoSistema, ms);

// Añadimos el modificador "abstracto" al tipo
   mt.addModifier("abstracto");

// Introducir mt en el ámbito de tipoSistema
   Declaration d = new Declaration(RES_METODO, "imprime", mt);
   ((ClassScope)tipoSistema.getScope()).insertarDecMetodo(d);
```

Compatibilidad de métodos

Por último, señalar una propiedad importante de los métodos: dos tipos de método pueden ser "compatibles". En antlraux, dos métodos son compatibles si:

- tienen el mismo nombre, o son los dos constructores y
- tienen el mismo número de parámetros y
- cada parámetro del primer tipo de método es compatible respecto al paso de parámetros con su equivalente en el otro tipo de métodos.

Esto quizás pueda entenderse mejor con código. `MethodType` ofrece el siguiente método:

```
public boolean compatibleWith(MethodType other)
```

Este método devuelve `true` si y sólo si:

- El método actual tiene el mismo nombre que `other`.
- Las listas de parámetros de los dos métodos son compatibles.

Para comprobar si las dos listas de parámetros son compatibles, se utiliza el método

```
public boolean compatibleParamsLists(MethodType other)
```

Este método devuelve `true` si y sólo si:

- Las listas de parámetros de ambos métodos tienen el mismo número de parámetros.

67 Más tarde veremos que lo que utilizaremos será una subclase de MethodType, pero por ahora nos basta con ella.

- Los parámetros son compatibles dos a dos.

Diremos que dos parámetros `myType` y `hisType` son compatibles en los métodos cuando la llamada al siguiente método devuelva `true`:

```
public boolean compatibleParams(Type myType, Type hisType)
{
      if( myType instanceof HeritableType &&
           hisType instanceof HeritableType )
      {
         if ( ((HeritableType)hisType).isA((HeritableType)myType) )
            return true;
      }
      else if( hisType.equals(myType)
           ||  myType.equals(hisType) )
         return true;

      return false;
}
```

Los tres métodos descritos (`compatibleWith`, `compatibleParamsList` y `compatibleParams`) son públicos, de manera que pueden ser utilizados por cualquier clase y por supuesto reescritos por subclases de `MethodType`.

Además de estos tres métodos, existe una versión ampliada de `compatibleWith`, que incluye tres booleanos para comparar la lista de modificadores, el tipo de retorno y el nombre. Su prototipo es el siguiente:

```
public boolean compatibleWith( MethodType other,
                               boolean compareName,
                               boolean compareReturnType,
                               boolean compareModifiers )
```

Una vez expuesta la compatibilidad de tipos de métodos, queda un interrogante por resolver: ¿para qué sirve?.

Es sencillo: la compatibilidad de parámetros permite resolver dos situaciones:

- Saber a qué método está llamando una llamada (utilizando la lista de parámetros de la llamada)
- Impedir que el mismo método sea introducido varias veces en una clase.

Y es que la regla para poder insertar un tipo de método en el ámbito de una clase es la siguiente:

Un método o constructor no debe poder insertarse en una clase si en dicha clase ya hay otro que es compatible con él.

La clase `leli.scopes.ClassScope`, de la que hemos hablado anteriormente, tiene en cuenta este hecho, lanzando una excepción cuando sea necesario.

MethodType se utilizará para modelar el tipo de los métodos “normales”, los abstractos y los constructores (cambiando etiqueta y tipo de retorno).

El tipo de un atributo: antlraux.context.types.AttributeType

Al igual que los métodos, los atributos también son “declarados”, así que necesitan de un tipo con el que “identificarse” dentro de su ámbito.

En un principio podría parecer que para este propósito nos basta con el tipo con el que es declarado el atributo: si es un atributo `Entero`, utilizamos el tipo `Entero`; si es `Cadena`, el tipo

`Cadena`, etc.

Sin embargo esto no es suficiente por una serie de razones:

- Dado un tipo de parámetro, debemos poder saber a qué clase corresponde
- Dado un tipo de parámetro, debemos poder saber si es un método abstracto o no
- También debemos saber con qué tipo fue declarado el atributo (`Entero`, `Cadena`, etc)

En otras palabras, necesitamos un tipo que implemente las interfaces `ClassMember` y `ModifiedType`. Además necesitará un atributo en el que guardar el tipo con el que se declaró, con sus métodos get y set.

Esta clase existe y se llama `antlraux.context.types.AttributeType` (nótese que es "Attribute", y no "Attribute**d**"). Al ser una subclase de `DeclaratedType`, implementa automáticamente las interfaces `Type` y `ModifiedType`. Además implementa la interfaz `ClassMember` (`setClassType` y `getClassType`).

Por último, `AttributeType` contiene un atributo de tipo `Type` llamado `type`, con métodos get y set, con el que modelar el tipo declarado del atributo.

`AttributeType` tiene un constructor en el que se definen todas sus propiedades excepto los modificadores, que se tienen que añadir utilizando el método `addModifier`. El prototipo del constructor de la clase es:

```
public AttributeType( int tag, String name, Type type, ClassType classType )
```

Veamos un ejemplo. Supongamos el siguiente atributo abstracto:

```
clase Persona
{
   atributo abstracto Entero mayoriaEdad = 18;
}
```

Cuando se reconozca ese código LeLi se ejecutará un código java equivalente al siguiente:

```
// Obtener el tipo de la clase Persona
ClassType tipoPersona = ... ;

// Crear el tipo del atributo
AttributeType at = new AttributeType( RES_ATRIBUTO, "mayoriaEdad",
                                      obtenerTipoEntero(), tipoPersona );
// Añadir modificador "abstracto"
at.addModifier("abstracto");

// Insertar el atributo en el ámbito de tipoPersona
Declaration d = new Declaration( RES_ATRIBUTO, "mayoriaEdad", at);
((ClassScope)tipoPersona.getScope()).insertarDecAtributo(d);
```

El tipo de una clase: antlraux.context.types.ClassType

De igual manera que los métodos pueden insertarse en el ámbito de una clase, las propias clases se insertan en el ámbito global. Esto lanza varios interrogantes:

- ¿Cómo relacionar las propias clases con el sistema ámbito/declaración/tipo?
- ¿Cómo modelar los métodos y atributos abstractos?
- ¿Cómo modelar la herencia?

Comencemos con la integración con ámbito/declaración/tipo.

Resolveremos el problema de la integración de una manera similar a la que hemos empleado en el caso de los métodos: utilizaremos una clase especial para modelar los tipos de las clases: `antlraux.context.types.ClassType`.

`ClassType` es una subclase de `DeclaredType`, por lo que implementa `ModifiedType`. Además implementa las interfaces `SimpleInheritanceType` y `ScopedType`. Al implementar el primero resolvemos el problema de modelar la herencia (recordemos que LeLi es un lenguaje en el que se utiliza la herencia simple). Al implementar el segundo conseguimos gestionar fácilmente los constructores, atributos y métodos, que se guardan en la instancia de `Scope` que la clase mantiene.

La clase `ClassScope` tiene un único constructor en el que se especifica la etiqueta, nombre, superclase y ámbito:

```
public ClassType( int tag, String typeName, ClassType superType, Scope s )
```

Además de los métodos de las diferentes interfaces que implementa, `ClassType` tiene tres métodos:

```
public Declaration searchCompatibleMethod( int methodTag, MethodType mt)
```

Sirve para "buscar" un método compatible con `mt` (se puede utilizar para buscar el método que se está invocando en una llamada)

```
public void addSuperClassMethods(int methodTag) throws ContextException
```

Dada una clase A con una superClase B, este método inserta en A los métodos de B "que deba insertar". Es decir, inserta en A los métodos de B que no sean compatibles con ninguno de los existentes en A. El método presupone que todas las declaraciones de métodos en los ámbitos de A y B tienen la misma etiqueta (`methodTag`)

```
public void addSuperClassAttributes(int attributesTag) throws ContextException
```

Similar al método anterior, pero utilizado para los atributos en lugar de para los métodos.

Los dos últimos métodos de `ClassType` ayudan a manejar la herencia.

7.3.7: ASTs especializados

En este apartado presentaremos otros ASTs proporcionados por antlraux que son de mucha ayuda durante el análisis sintáctico. Son tres: `ScopeAST`, `TypeAST` y `ExpressionAST`. Los tres forman el paquete `antlraux.context.asts.*`.

antlraux.context.asts.ScopeAST

La idea de este AST es muy sencilla: es una subclase de `antlr.LexInfoAST` (presentada en la sección 7.1.3) pero con las modificaciones necesarias para poder guardar una referencia a una instancia de `antlraux.context.Scope`. Es decir, es un AST que guarda información de un ámbito.

Esta clase es muy útil si, como en el caso de nuestro compilador, se desea realizar una comprobación de tipos en dos pasadas (ver apartado 7.5.1 en la página 292).

La implementación de `ScopeAST` es tan sencilla como parece: se heredan todos los métodos de `CommonAST`, y solamente queda añadir un atributo de tipo `Scope` (con sus respectivos métodos get y set). Además, se ha sobreescrito un métodos de inicialización (`initialize(AST)`) y se ha modificado el método `toString()` para que aparezca información del ámbito.

antlraux.context.asts.TypeAST

Este AST también representa un concepto muy sencillo: es un AST con una instancia de `Type` (una instancia de alguna clase que implemente la interfaz `antlraux.context.types.Type`).

`TypeAST` es muy parecida a `ScopeAST`: una subclase de `LexInfoAST`, a la que añade un atributo, aunque esta vez de tipo `Type` (con métodos get y set). También modifica `initialize(AST)` y `toString()`.

En la regla `tipo` de nuestros analizadores semánticos utilizaremos `TypeAST` para guardar el tipo de LeLi al que se refiere cada opción de la regla:

```
tipo
{ Type t = null; }
   : ( TIPO_ENTERO { t = obtenerTipoEntero(); }
     | TIPO_REAL   { t = obtenerTipoReal(); }
    ...
     )
   {
      TypeAST ast = new TypeAST(##);
      ast.setExpType(t);
      ## = ast;
   }
   ;
```

Así es más eficiente obtener el tipo de LeLi a partir de la regla `tipo`.

Nótese que en el código no se ha podido escribir ast.setType(t) – `setType` es un método de la interfaz AST que no podemos utilizar para los tipos del lenguaje (otra consecuencia de una nomenclatura demasiado genérica; debería llamarse "tag") así que hemos usado la nomenclatura `setExpType` y `getExpType`.

antlraux.context.ExpressionAST

Es el tipo de AST que deberemos utilizar en las expresiones del lenguaje. Es una subclase de `TypeAST`, así que automáticamente hereda de él la capacidad de guardar un tipo de LeLi (`setExpType` y `getExpType`). Además añade los siguientes atributos:

- `boolean RValue`: Si una expresión tiene Rvalue a false, no podrá ser utilizada en la parte derecha de una asignación (más información sobre Rvalue en la página 248).
- `boolean LValue`: Si una expresión no tiene LValue, no podrá ser utilizada en la parte *izquierda* de una asignación (más información sobre Lvalue en la página 248)
- `Object EValue`: Sirve para guardar un valor que pueda calcularse en tiempo de compilación (por ejemplo, inicialización de constantes). También permite guardar valores de expresiones en lenguajes interpretados.

Los tres atributos anteriores tienen, al igual que `expType`, métodos get y set correspondientes.

```
ExpressionAST también sobreescribe el método initialize(AST) y toString().
```

Notas sobre los árboles

Todos los tipos de AST que hemos presentado en este apartado tienen dos características importantes a destacar: inicializaciones voraces y un toString manejable.

Decimos que ScopeAST, TypeAST y ExpressionAST tienen "inicializaciones voraces" porque su

método initialize(AST) no se limita a copiar el tipo y texto del AST que le pasan: además copia su hijo y hermano, y características adicionales si el AST es de cierto tipo. Por ejemplo, el método initialize(AST) de `ExpressionAST` es el siguiente:

```
public void initialize(AST ast)
{
   // Copiar tipo, texto e información léxica
   super.initialize(ast);

   // Copiar hijo y hermano
   setFirstChild(ast.getFirstChild());
   setNextSibling(ast.getNextSibling());

   // Copiar datos adicionales si es un ExpressionAST
   if(ast instanceof ExpressionAST)
   {
      ExpressionAST aux = (ExpressionAST)ast;
      this.expType = aux.expType;
      this.RValue = aux.RValue;
      this.LValue = aux.LValue;
      this.EValue = aux.EValue;
   }
}
```

El constructor que toma un AST llama a `initialize(AST)` internamente, así que los constructores de las tres clases también son "voraces".

La razón de copiar el hijo y el hermano es que normalmente estos ASTs serán utilizados para "reemplazar" al ast generado por una regla , es decir, a "`##`". Ver como ejemplo el código de la página anterior.

Decimos que los tres ofrecen un método `toString` "manejable" porque utilizan la misma estrategia que `LexInfoAST` para presentar información adicional: por defecto no presentan información alguna, pero pueden presentarla activándola con el método estático `setVerboseStringConversion(boolean)`. Por defecto no se presentará mucha información; dado que `toString` es el método que se utiliza en las ventanas `ASTFrame`, podemos ver gráficamente la mejora. A continuación mostramos una `ASTFrame` mostrando un "hola mundo" de LeLi, con toda la información adicional desactivada:

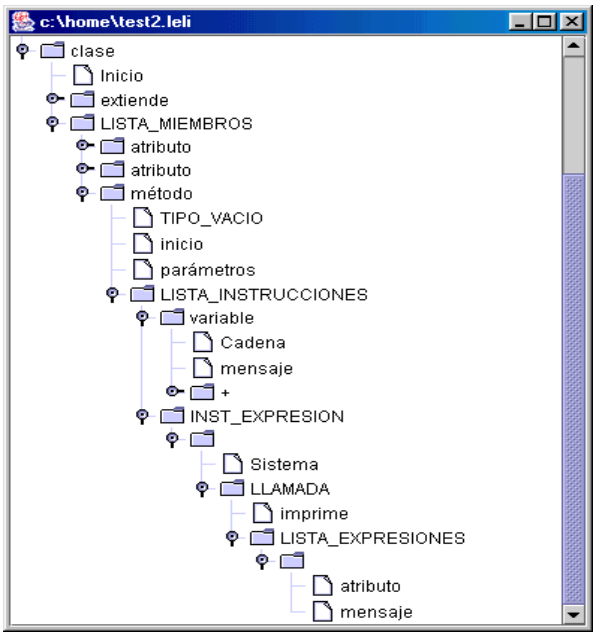

Ilustración 7.6 ASTs sin mostrar información adicional

Sin embargo obsérvese lo que ocurre si añadimos las siguientes líneas a `leli.Tool`:

```
BaseAST.setVerboseStringConversion(
   true, LeLiParser._tokenNames);
LexInfoAST.setVerboseStringConversion(true);
ScopeAST.setVerboseStringConversion(true);
TypeAST.setVerboseStringConversion(true);
ExpressionAST.setVerboseStringConversion(true);
```

Entonces obtendremos esta otra ventana:

Ilustración 7.7 ASTs con información adicional mostrada

Es posible desactivar la información de `LexInfoAST`, para no tener un exceso de información.

7.3.8: Resumen

En estas dos secciones hemos descrito el paquete antlraux.context y su subpaquete `antlraux.contex.types`. En concreto hemos definido los siguientes paquetes, clases e interfaces:

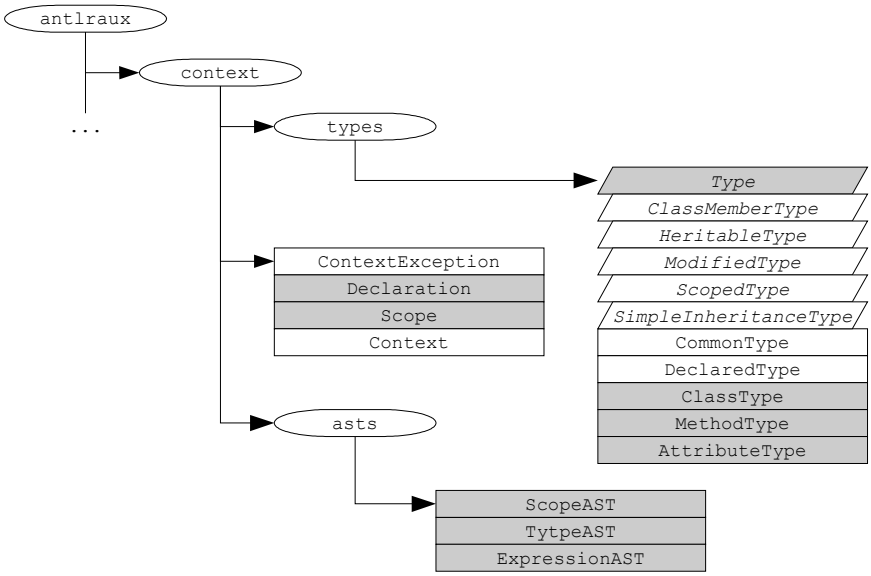

Ilustración 7.8 Esquema de paquetes de antlraux.context

Las clases e interfaces más importantes aparecen sombreadas: son `Scope`, `Declaration`, `Type`, `ClassType`, `MethodType`, `AttributeType` y los asts, `ScopeAST`, `ExpressionAST` y `TypeAST`.

Sección 7.4: ADT y LeLi

7.4.1: Introducción

En esta sección veremos cómo hemos utilizado las clases e interfaces proporcionadas por antlraux.context para implementar el sistema ADT en LeLi. Esta sección seguirá teniendo un estilo bastante "teórico", basado en definiciones; mis disculpas.

En la siguiente sección retomaremos el enfoque "tutorial".

7.4.2: Las declaraciones en LeLi

`Declaration` es una "clase-estructura": carece de métodos importantes, simplemente es una agrupación de datos. Su principal función es ser manejada por un ámbito (una instancia de la clase `antlraux.contex.Scope`).

`Declaration` posee un constructor "básico", que solamente rellena el nombre, el tipo y la etiqueta, dejando el resto de los campos con valores por defecto (null, -1). y varios "extendidos", que rellenan todos los atributos.

En LeLi `Declaration` se utilizará "tal cual": no se implementarán subclases de `Declaration`.

Las duplas identificadoras <etiqueta, nombre> de las declaraciones en LeLi serán las siguientes:

- `<TIPO_ENTERO, "Entero">` Para el tipo básico `Entero`.
- `<TIPO_REAL, "Real">` Para el tipo básico `Real`.
- `<TIPO_BOOLEANO, "Booleano">` Para el tipo básico `Booleano`.
- `<TIPO_CADENA, "Cadena">` Para el tipo básico `Cadena`.
- `<RES_CLASE, "nombreClase">` Para cualquier otra clase definida en LeLi (incluyendo las clases `Objeto` y `Sistema`)
- `<TIPO_VACIO, "vacío">` Para el tipo vacío.
- `<TIPO_ERROR, "error">` Para el tipo error.

7.4.3: Los ámbitos en LeLi

Algunas restricciones

La clase `Scope` permite modelar muchas situaciones que podemos encontrarnos al trabajar con ámbitos, porque es muy flexible: ignora completamente las etiquetas y nombres de las declaraciones y otros ámbitos. Una instancia de Scope acepta la inserción de cualquier declaración, y se le puede asignar cualquier otra instancia de Scope como padre, sin que se haga comprobación alguna. Esta actitud, necesaria para proveer de un esqueleto suficientemente flexible que se pueda utilizar en el mayor número de casos posible, puede resultar demasiado permisiva en algunas situaciones.

En LeLi, por ejemplo, hay ciertas situaciones que deben evitarse:

- El ámbito global no debe tener ningún ámbito padre y todas las declaraciones que se le inserten deben ser de etiqueta `RES_CLASE`. Los nombres de dichas declaraciones no deben coincidir.
- En el ámbito de una clase solamente pueden insertarse métodos, atributos y constructores, o lo que es lo mismo, declaraciones de etiqueta `RES_METODO`, `RES_CONSTRUCTOR` y `RES_ATRIBUTO`. Los ámbitos hijos del ámbito deben ser obligatoriamente ámbitos de métodos o constructores.

Además, el ámbito de una clase no debe aceptar la inserción de cualquier método o constructor; existen reglas concretas de aceptación, que veremos más adelante.

- El ámbito de un método o constructor admitirá la inserción de parámetros o variables. Solamente podrán hacerse hijos de ellos los ámbitos de bucles o de condicionales.
- Dentro de los ámbitos de bucles o de instrucciones condicionales solamente podrán instanciarse variables, y solamente podrán anidarse nuevos ámbitos de bucles o instrucciones.

La clase antlraux.context.Context

Esta clase existe para facilitar el trabajo con los ámbitos y para automatizar ciertas tareas de manejo de ámbitos (como mantener un "ámbito actual" y realizar comprobaciones al abrir y cerrar dicho ámbito). Con los métodos adecuados, una subclase de `Context` puede hacer de "fachada" entre el sistema ADT y los analizadores.

Una subclase de `Context`, `LeLiContext`, será utilizada abundantemente en el análisis semántico.

Utilización de Scope y Context

La estrategia elegida para implementar los ámbitos podría resumirse en una palabra: herencia. Se centra concretamente en ejes de acción:

- Para controlar las peculiaridades de cada tipo de ámbito utilizaremos diversas subclases de `antlraux.context.Scope`. Estas clases serán "dependientes del lenguaje LeLi", por lo que no estarán incluidas en `antlraux`, sino en `leli`. Concretamente dentro de un subpaquete llamado `leli.scopes`. Los nombres de estas clases serán **`GlobalScope`**, **`ClassScope`**, **`MethodScope`** e **`InternScope`**, y controlarán la inserción de declaraciones mediante métodos especializados; por ejemplo, `ClassScope` tiene un método para insertar declaraciones de métodos llamado `insertDecMetodo`, en el que se realizan todas las comprobaciones necesarias antes de insertarlo.
- Además, en el paquete `leli` encontraremos una subclase de `antlraux.context.Context`, llamada `leli.LeLiContext`. Esta clase tendrá varias funciones; por un lado será la encargada de mantener el "ámbito actual", es decir, el ámbito que esté activo en cada momento. Por otro lado hará de "fachada" entre los analizadores semánticos y todos los ámbitos (instancias de Scope o alguna de sus subclases) para simplificar el uso de los ámbitos; por ejemplo, contará con un método `insertDecMetodo` que "casteará" el ámbito actual a `ClassScope` e invocará el método `ClassScope.insertDecMetodo`. De esta manera los analizadores semánticos simplemente tendrán que hacer uso de la clase LeLiContext para manejar todos los aspectos de los ámbitos.

Téngase en cuenta que el segundo paso no es estrictamente necesario: es posible manejar directamente las diferentes instancias de `Scope` sin tener que utilizar una clase intermedia, pero el código sería más ilegible y menos mantenible de esa forma.

Las duplas <etiqueta, nombre> de los ámbitos en LeLi serán:

En el caso concreto de LeLi, la dupla <nombre, etiqueta> será:

- <"ámbito global", `PROGRAMA`> para el ámbito global (única instancia de `leli.scopes.GlobalScope`).
- <"clase nombreClase", `RES_CLASE`> para el ámbito de la clase llamada "nombreClase" (que es del tipo `leli.scopes.ClassScope`).

- <"método nombreMétodo", `RES_METODO`> para el ámtito del método llamado "nombreMétodo" (instancias de `leli.scopes.MethodScope`).
- <"constructor", `RES_CONSTRUCTOR`> para cualquier constructor (también se utilizará aquí la clase `leli.scopes.MethodScope`).
- <"condicional", `RES_SI`> para cualquier alternativa de una instrucción condicional, incluyendo la primera (leli.scopes.InternScope).
- <"bucle mientras", `RES_MIENTRAS`> para el cuerpo de un bucle `mientras` (`leli.scopes.InternScope`).
- <"bucle hacer-mientras", `RES_HACER`> para el cuerpo de un bucle `hacer-mientras` (`leli.scopes.InternScope`).
- <"bucle desde", `RES_DESDE`> para el cuerpo de un bucle `desde` (`leli.scopes.InternScope`).

7.4.4: El sistema de tipos de LeLi

Tratamiento de errores semánticos y el Tipo Error

Comencemos con una simple pregunda: ¿Qué ocurre cuando se detecta un error?

Consideremos el siguiente ejemplo:

```
método errorSemántico(Entero ent1)
{
   Sistema.imprime(en1);
}
```

En el ejemplo, presuntamente el programador ha querido iniciar la variable local `en2` con el valor del parámetro `ent1`. Sin embargo se ha equivocado al escribirlo (falta una t) de manera que se genera un error semántico, por utilizar en una expresión un identificador no declarado (`en1`).

Está claro que el compilador deberá anunciar que ha encontrado un error semántico claramente. Pero lo interesante es lo que debe hacer después. Una de las opciones sería emitir un mensaje de error y cancelar la creación del AST, pero acabamos de decir que hay que no podemos "parar": debemos construir un nuevo AST utilizando la información de contexto, *aunque existan errores*. Pero cuando se da un error semántico como el del ejemplo, ¿qué información semántica se puede utilizar? ¿Qué valor debemos darle a la expresión `en1`?

La pregunta no es qué *valor*. Eso no tiene importancia; el valor es utilizado en tiempo de ejecución, y como el programa ha tenido un error no se generará ningún ejecutable. Lo importante es qué *tipo*.

Para modelar situaciones excepcionales como la presente lo ideal es contar con un tipo especial, con características especiales. Me estoy refiriendo al tipo `TIPO_ERROR`.

`TIPO_ERROR` se deberá añadir a la sección de tokens del analizador como un token imaginario:

```
tokens
{

...
   TIPO_ERROR;
}
```

Este tipo será el asignado a las expresiones erróneas como la anterior. Así, el análisis semántico

discurrirá de la siguiente manera en el análisis anterior:

```
método errorSemántico2(Entero ent1)
<
   Recordar que se ha declarado el método vacío errorSemántico2
   No incluir variable de método - es vacío
   Recordar que se ha decarado el parámetro Entero ent1
>
{
   Entero ent2 = ent1;
   <
      Recordar que se ha declarado la variable ent2 Entera
      La expresión con la que se inicia es de tipo Entero
   >
   Sistema.imprime(en2);
   <
      Sistema se reconoce como una clase reservada.
      El método imprime forma parte de dicha clase.
      El identificador "en2" no es una expresión válida - no está declarado.
      Informar del error al usuario.
      El tipo de en2 será TIPO_ERROR.
      Continuar.
   >
}
```

Cada vez que en una expresión `TIPO_ERROR` se lanza un mensaje de error. Por lo tanto, los subsecuentes mensajes de error son reiterativos. Si un parámetro fue declarado de forma errónea, y es utilizado más adelante en una expresión, no tiene sentido anunciar que "la suma no puede realizarse entre el `Entero` y `TIPO_ERROR`", porque ya se ha avisado al usuario de un error previo. Los errores "reiterativos" como éstos se denominan "errores en cascada".

Para los errores en cascada vamos a dotar a `TIPO_ERROR` de algunas propiedades interesantes:

- `TIPO_ERROR` es compatible con cualquier otro tipo respecto a cualquier otro operador, incluyendo el paso como parámetro y el de asignación.
- Una expresión con `TIPO_ERROR` permite cualquier acceso a un atributo llamado de cualquier manera (`error.ident`). El tipo del acceso es también `TIPO_ERROR`.
- `TIPO_ERROR` permite cualquier invocación de cualquier método con cualquier tipo de parámetros (`error.método(p1,p2,...)`). El tipo de la expresión será también `TIPO_ERROR`.

El paquete leli.types

Todas las clases que serán utilizadas para modelar algún tipo en el compilador de LeLi (incluyendo el tipo error) estarán dentro del paquete `leli.types`. Todas ellas serán clases derivadas de clases del paquete `antlraux.context.types` o implementarán alguna interfaz de dicho paquete.

Modelado de los tipos "normales" de LeLi

Los tipos "normales" de LeLi son aquellos tipos que caracterizan a una variable, un parámetro o un atributo en LeLi. Esto incluye a los tipos básicos de LeLi (`Entero`, `Booleano`, etc) y a los tipos de LeLi definidos por el usuario. Los tipos normales serán modelados utilizando la clase **`leli.types.LeLiType`**. Ésta es a su vez una subclase de `antlraux.ClassType`.

`LeLiType` es básicamente lo mismo que `ClassType`, salvo por algunos métodos "envoltorio" (*wrapper*) y por su capacidad de utilizar metaclases (ver más adelante). También conoce el

vocabulario de los analizadores de LeLi, porque implementa `LeLiSymbolTreeParserVocabTokenTypes` (que veremos en la siguiente sección).

`LeLiType` ofrece dos constructores diferentes: el más simple presupone que la etiqueta del nuevo tipo a generar es `RES_CLASE`, mientras que el segundo permite especificar uno diferente.

```
public LeLiType( String name, LeLiType superType, Scope s,
                 LeLiMetaType metaClase )

public LeLiType( int tag, String name, LeLiType superType, Scope s,
                 LeLiMetaType metaClase )
```

Los dos permiten definir el nombre, superclase, ámbito y metaclase de las nuevas clases que se definan. Si no se especifica una etiqueta se asumirá la etiqueta `RES_CLASE`.

Gestión de los tipos predefinidos: LeLiTypeManager

La manera usual de obtener un tipo estará basada en buscar su declaración en el ámbito global, utilizando su nombre. Es decir, si queremos obtener el tipo representado por la cadena "Persona", tendremos que:

- Obtener el ámbito raíz.
- Buscar una declaración llamada "Persona"
- Obtener el tipo de la declaración (con `getType()`)

Para obtener "Persona" no tendremos más remedio que hacerlo de esta forma, porque el tipo Persona solamente se conocerá una vez comenzada la compilación. Sin embargo, hacer lo mismo con los tipos predefinidos del sistema sería altamente ineficiente: típicamente, tendremos que buscar "Entero", "Real" o "Cadena" un centenar de veces, para obtener la misma información.

En aras de la eficiencia se incluyó la clase `leli.types.LeLiTypeManager`, que se encarga de guardar los tipos básicos del lenguaje LeLi en forma de miembros estáticos. El código completo de `LeLiTypeManager` es el siguiente:

```
package leli.types;

import leli.LeLiSymbolTreeParserVocabTokenTypes;

public abstract class LeLiTypeManager
implements LeLiSymbolTreeParserVocabTokenTypes
{
   public static LeLiType tipoError =
      new LeLiType( TIPO_ERROR, "error", null, null,
                    new LeLiMetaType(TIPO_ERROR, "error", null) );

   public static LeLiType tipoVacio =
      new LeLiType( TIPO_VACIO, "vacío", tipoError, null,
                    new LeLiMetaType(TIPO_VACIO, "vacío", null) );

   public static LeLiType tipoObjeto =
      new LeLiType( RES_CLASE, "Objeto", tipoError, null,
                    new LeLiMetaType("Objeto") );

   public static LeLiType tipoClase =
      new LeLiType( METACLASE, "Clase", tipoObjeto, null, null);

   public static LeLiType tipoEntero =
```

```
      new LeLiType( TIPO_ENTERO, "Entero", tipoObjeto, null,
                    new LeLiMetaType("Entero"));

   public static LeLiType tipoReal =
      new LeLiType( TIPO_REAL, "Real", tipoEntero, null,
                    new LeLiMetaType("Real") );

   public static LeLiType tipoBooleano =
      new LeLiType( TIPO_BOOLEANO, "Booleano", tipoObjeto, null,
                    new LeLiMetaType("Booleano") );

   public static LeLiType tipoCadena =
      new LeLiType( TIPO_CADENA, "Cadena", tipoObjeto, null,
                    new LeLiMetaType("Cadena") );
}
```

El código comienza definiendo el tipo error y el tipo vacío. Después se definen el tipo `Objeto` y el tipo `Clase` (este último lo veremos en el siguiente sub apartado). Seguidamente se definen el resto de los tipos básicos de LeLi: `Entero`, `Real`, `Booleano` y `Cadena`. Nótese que `Objeto` se hace sub clase de error; esto hace más fácil reconocer llamadas a métodos que no han sido definidos correctamente (que tienen algún error en sus parámetros).

Una vez definida `LeLiTypeManager`, para trabajar con el tipo `Entero` solamente hay que escribir:

```
LeLiType t = LeLiTypeManager.tipoEntero;
```

Metaclases

Para resolver el problema de los accesos abstractos he incluído un nuevo tipo de clase llamado "metaclase".

Una de metaclase podría ser "la representación de una clase mediante una instancia". Aunque para nosotros será mucho más sencillo: una metaclase será, simplemente, un tipo especial de "tipo de clase" que solamente tiene atributos y métodos abstractos. Obviamente no se diferencia mucho de una clase normal y corriente.

Existen además otras diferencias, derivadas del tratamiento especial que requieren los miembros abstractos de una clase. Este tratamiento incluye que:

- Los métodos y atributos abstractos son "impermeables a la herencia": no pueden ser "sobreescritos", como los métodos "normales"
- Dentro del ámbito de un método abstracto solamente pueden utilizarse atributos abstractos.

Y sin duda la diferencia más importante:

- Los métodos abstractos no requieren instancias de una clase para ser ejecutados: basta con escribir el nombre de la clase, seguido de la llamada al método.

Por ejemplo, los métodos de impresión de la clase básica `Sistema` son abstractos; pueden invocarse escribiendo el nombre de la clase seguido del método:

```
Sistema.imprime("¡Hola mundo!"+nl);
```

Y ahora es cuando se plantea el problema: necesitamos un tipo nuevo para el acceso "Sistema", señalado en el código: Comparémoslos con el acceso señalado en este otro:

```
Persona mortadelo;
mortadelo.decir("¡Hola mundo!"+nl)
```

Las diferencias son clases:

- "Sistema" solamente puede utilizarse para acceder a métodos y atributos abstractos de la clase Sistema, mientras que "mortadelo" puede acceder a los métodos y atributos estáticos *y no estáticos* de la clase `Persona` o *de cualquier superclase de* `Persona`..
- "mortadelo" puede convertirse (con la construcción `convertir`) a cualquier superclase.
- "mortadelo" tiene constructores.
- Si la clase `Persona` tuviera algún miembro abstracto, hay que diferenciar los accesos "abstractos" y "no abstractos":

```
clase Persona
{
   atributo abstracto Entero mayoríaEdad = 18;
   atributo Entero edad;
}
...
Persona mortadelo;
si(mortadelo.edad >= Persona.mayoríaEdad)
{ Sistema.imprime("Mortadelo es mayor de edad"); }
| otras
{ Sistema.imprime("Mortadelo no es mayor de edad"); }
```

Está claro que los accesos "mortadelo" y "Persona" señalados en el código no pueden ser del mismo tipo; "Persona", por ejemplo, no puede acceder al atributo no abstracto "edad".

Diremos que mientras "mortadelo" es una instancia de la clase `Persona`, "Persona" es una referencia a la *metaclase* `Persona`.

El tipo-metaclase se implementa mediante `leli.types.LeLiMetaType`. `LeLiMetaType` es una subclase de `LeLiType`, y además uno de los atributos de `LeLiType`. En lenguaje UML podría representarse así:

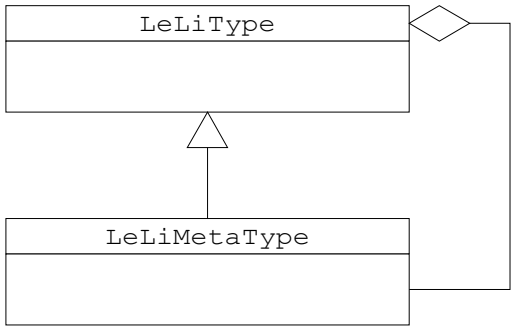

Ilustración 7.9 Relación entre LeLiType y LeLiMetaType

Esta estructura es un poco "extraña": la clase `LeLiType` tiene un atributo cuyo tipo es una subclase de `LeLiType` (clases recursivas). La estructura funciona porque que `LeLiType` no hace ningún uso de `LeLiMetaType`; solamente guarda una referencia.

No hay diferencias importantes de implementación entre `LeLiType` y `LeLiMetaType`; las diferencias son de tipo lógico:

- La etiqueta de `LeLiMetaType` es siempre `METACLASE`, mientras que la de `LeLiType` es

`RES_CLASE`, `TIPO_ENTERO`, `TIPO_REAL`, `TIPO_BOOLEANO`, `TIPO_CADENA`, `TIPO_ERROR` o `TIPO_VACIO`.

- `LeLiMetaType` posee un ámbito propio en el que solamente habrá declaraciones de métodos y atributos abstractos.
- `LeLiMetaType` no participa en la herencia: todas las metaclases se considerarán "hijas" de la metaclase `LeLiTypeManager.tipoClase`.

Sección 7.5: Comprobación de tipos - Primera pasada

7.5.1: Comprobación de tipos en dos pasadas

Hasta ahora todos los análisis que hemos realizado han requerido "un solo tratamiento" de los datos: el analizador léxico trataba los caracteres "una sola vez", es decir, cada carácter era tratado una sola vez y ya no se volvía a tener en cuenta. Algo parecido ocurría con el flujo de tokens: el analizador sintáctico lo recorría una sola vez, y construía el AST.

Para implementar la comprobación de tipos (CT de ahora en adelante) sin embargo, vamos a necesitar un análisis que recorra el AST *dos* veces (y por lo tanto deberemos implementar dos analizadores diferentes).

¿Por qué es necesario implementar dos analizadores diferentes? Tiene que ver con la manera en que hemos estructurado la información de las declaraciones (ámbitos, declaraciones, etc).

Considérese el siguiente fragmento de código:

El siguiente fragmento de código está basado en un fragmento de código de la lección V del tutorial de PDXScript de peroxide (**http://www.peroxide.dk**)

```
clare Recursiva
{
   método Entero a(Entero valor)
   {
      si (valor > 0)
      { a = b(valor - 1) ; }
      | otras
      { a = 0; }
   }

   método Entero b(Entero valor)
   {
      si (valor > 0)
      { b = a(valor - 1) ; }
      | otras
      { b = 0; }
   }
}
```

*****

En este ejemplo, el método `b` se invoca desde el método `a`, y al mismo tiempo `a` es invocado desde `b`. Recorriendo una sola vez el AST no podríamos "saber" qué significa `b(valor -1)` la primera vez que lo encontramos. Queda pues claro que es necesario realizar dos "pasadas":

- Una primera pasada que se encargue de "registrar" las declaraciones del lenguaje.
- Una segunda que utilice estas declaraciones para realizar las comprobaciones de tipos propiamente dichas[68].

Al analizador que realizará la primera pasada lo llamaremos `LeLiSymbolTreeParser`, y al segundo `LeLiTypeCheckTreeParser`. En esta sección estudiaremos el primero, y en la siguiente el segundo.

68 En C/C++ solamente se realiza una pasada, pero el programador está obligado a especificar los tipos de las funciones que va a utilizar por adelantado, utilizando los "prototipos de funciones" y "prototipos de clases".

Explicación detallada de los objetivos del analizador

La tarea principal del analizador será preparar el sistema ADT, creando todos los ámbitos e introduciendo las declaraciones que sean pertinentes en dichos ámbitos. Esto incluye:

- Crear todos los ámbitos pertinentes: el global, el de cada clase, método, instrucción condicional y bucle.
- Introducir las declaraciones de clases en el ámbito global
- Introducir las declaraciones de atributos, métodos y constructores en el ámbito de cada clase
- Introducir las declaraciones de parámetros en los métodos

Nótese que las declaraciones de variables locales no se incluirán en sus ámbitos en la primera pasada; esto ocurre porque las declaraciones de variables no deben estar "disponibles en todo su ámbito" como ocurre con los métodos; solamente deben poder utilizarse por las intrucciones que suceden a las declaraciones. Es decir, que mientras el siguiente código es válido:

```
método a(Entero e) // "a" "conoce" a "b"
{
   si( e>0 ) { b(e-1); }
}
método b(Entero e) // y simultáneamente "b" "conoce" a "a"
{
   a( e-1 );
}
```

Este otro no lo es:

```
método c()
{
   si (i>0) { b(i); } // "i" está indefinido en este punto
   Entero i=9;
}
```

De esta forma, las variables serán insertadas en los ámbitos durante la segunda pasada.

Simultáneamente, la primera pasada sirve para crear el AST heterogéneo que permitirá realizar la comprobación de tipos. Para ello deberá utilizar las clases definidas en el paquete `antlraux.context.asts.*`. Así, el analizador debe:

- Crear nodos del tipo `antlraux.context.asts.ScopeAST` para los nodos de definición de los elementos con ámbitos de LeLi, es decir las clases, los métodos, los métodos, las alternativas de las instrucciones condicionales y los cuerpos de los bucles. Es decir, deberá hacer que los nodos de tipo `RES_CLASE`, `RES_METODO`, `RES_CONSTRUCTOR`, `RES_MIENTRAS`, `RES_HACER`, `RES_DESDE`, `RES_SI`, `BARRA_VERT` y `RES_OTRAS`.
- Crear nodos del tipo `antlraux.context.asts.ExpressionAST` para todos los nodos de las expresiones.
- Crear nodos del tipo `antlraux.context.asts.TypeAST` para todas las alternativas de la regla `tipo`.

Por último, el primer analizador semántico debe preparar el ámbito global, insertando en él los tipos predefinidos del lenguaje (`Objeto`, `Entero`, `Cadena`, `Real`, `Booleano`, `Sistema`, `error` y `vacío`) de manera que estén disponibles para su utilización antes de empezar a leer el código del usuario. Como veremos esta tarea no es trivial.

Las fases del analizador

Dividiremos las reglas del analizador en cuatro grandes "grupos", a los que llamaremos "fases".

- En la primera fase se prepararán los ámbitos y se "engancharán" en los nodos AST que se crearán para ellos, es decir, las instancias de `antlraux.context.asts.ScopeAST`. En esta fase también se introducirán las declaraciones pertinentes dentro de los ámbitos.
- En la segunda fase se prepararán los ASTs de las expresiones (se crearán las instancias de la clase `antlraux.context.asts.ExpressionAST` para las expresiones).
- En la tercera fase se prepararán los ASTs de los tipos (instancias de `antlraux.context.asts.TypeAST`).
- La cuarta fase del analizador es la que se encarga de añadir todos los tipos predefinidos de LeLi al ámbito global.

7.5.2: Definición del analizador

Cabecera

`LeLiSymbolTreeParser` pertenecerá, como el resto de los analizadores que hemos implementado, al paquete leli, y como es habitual así lo especificaremos en la sección `header` junto con las importaciones necesarias, que serán muchas:

```
header{
   package leli;
   import ... ; // importaciones variadas
}
```

El se basará en el esqueleto proporcionado por el iterador simple de árboles, `LeLiTreeParser`, que describimos en la sección 7.2 "Iterador simple de árboles", utilizando la herencia de gramáticas una vez más.

```
class LeLiSymbolTreeParser extends LeLiTreeParser;
```

Zona de opciones

Las opciones serán las que siguen:

```
options
{
   importVocab=LeLiParserVocab;
   exportVocab=LeLiSymbolTreeParserVocab;
   buildAST = true;
   ASTLabelType = antlraux.util.LexInfoAST;
}
```

- `importVocab` y `exportVocab` sirven para importar y exportar el conjunto de símbolos. Nótese que no es necesario importar el conjunto de símbolos del analizador sintáctico con tolerancia a errores (`LeLiErrorRecoveryParserVocab`) porque en dicho analizador no se definió ningún nuevo símbolo.
- `buildAST` activa la construcción por defecto del AST – ¡este analizador, al contrario que LeLiTreeParser, sí construirá un nuevo AST!
- `ASTLabelType` está ahí por comodidad; como ya hemos explicado en la sección 7.1.3, "Añadiendo información léxica a los ASTs", todos los nodos ASTs que se crearán en nuestro

analizador sintáctico serán de la clase `LexInfoAST` o de una de sus subclases; así será más sencillo acceder a la información léxica de los ASTs (de otra forma tendríamos que utilizar infinidad de *castings* en multitud de sitios).

Zona de tokens

La sección de tokens tampoco presentará grandes sorpresas: se limitará a definir dos nuevos tipos de token, necesarios para implementar el tipo error y las metaclases de LeLi:

```
tokens
{
   TIPO_ERROR  = "error";
   METACLASE   = "metaclase" ;
}
```

Zona de código nativo

He aquí la zona de código nativo de este analizador:

```
{
   private LeLiContext contexto;
   private Logger logger;

   public LeLiSymbolTreeParser(Logger logger)
   throws RecognitionException
   {
      this();

      this.logger = logger;
      this.contexto = new LeLiContext(logger);

      setASTNodeClass("antlraux.util.LexInfoAST");
   }

   public LeLiSymbolTreeParser(Logger logger, LeLiContext contexto)
   throws RecognitionException
   {
      this();

      this.logger = logger;
      this.contexto = contexto;

      setASTNodeClass("antlraux.util.LexInfoAST");
   }

   public LeLiContext obtenerContexto()
   { return contexto; }

   public void reportError( String msg,
                            String filename,
                            int line,
                            int column )
   {
      if(null==logger)
      {
         logger = new Logger("error", System.err);
      }
```

```
      logger.log( msg, 1, filename, line, column);
   }

   public void reportError(RecognitionException e)
   {
      reportError( e.getMessage(), e.getFilename(),
                   e.getLine(), e.getColumn() );
      e.printStackTrace(System.out);
   }
}
```

La zona comienza con la definición de dos atributos del analizador: una instancia de Logger, una clase que ya hemos estudiado en capítulos anteriores, y que sirve para registrar los errores, y una instancia de LeLiContext.

`LeLiContext` es la "clase fachada" que permitirá a los analizadores sintácticos utilizar el sistema ámbito/declaración/tipo casi totalmente.

A continuación se declaran dos constructores muy sencillos, cuya razón de ser quedará clara en breve. Por último se sobreescribe el método `reportError(RecognitionException)` y se añade `reportError(String,String,int,int)`, de la misma manera que se hizo en el capítulo 6, en el analizador con recuperación de errores.

7.5.3: Fase 1: Creación de ámbitos

Programa

Como viene siendo habitual, comenzaremos con la regla programa del analizador. Un programa seguirá siendo una lista de declaraciones, aunque en esta ocasión con un pequeño añadido. Veamos:

```
programa
   : #( PROGRAMA
        instalarTiposBasicos
        (decClase)+ )
      { ## = new ScopeAST(##, contexto.getCurrentScope()); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

En esta ocasión se invoca una regla, llamada `instalarTiposBasicos`, antes de reconocer las declaraciones de las clases. `instalarTiposBasicos` es la fase 4 del analizador, y la estudiaremos más adelante.

El único aspecto que queda por comentar de la regla es la acción que se encuentra al final. En ella, el AST generado por la regla se sustituye por una instancia de `ScopeAST`. Téngase presente que `ScopeAST` utiliza un constructor "voraz" (que copia el hermano y el primer hijo del AST que se le pasa como parámetro) luego no es necesario hacer nada más para construirlo.

El resto de las reglas de la fase 1 acaban todas con una acción parecida; de ahora en adelante prescindiremos de comentarla.

Nótese también el manejador de excepciones, que se limita a imprimir el error, cancelando la recuperación.

Declaración de clases

La siguiente regla que vamos a estudiar es la que reconoce la declaración de las clases, es decir,

`decClase`. La regla tiene la siguiente forma:

```
decClase
{
   LeLiType claseActualTipo = null;
   ScopeAST ast = null;
}:
   #( RES_CLASE nombre:IDENT #(RES_EXTIENDE padre:IDENT)
   {
      ast = new ScopeAST(##, contexto.abrirAmbitoClase (#nombre));

      if(contexto.tiposBasicosLeidos == true)
      {
         LeLiType tipoPadre = contexto.obtenerTipo(#padre);

         claseActualTipo =
         new LeLiType( #nombre.getText(),
                       tipoPadre,
                       contexto.getCurrentScope(),
                       new LeLiMetaType(#nombre.getText()) );
      } else {
         claseActualTipo =
            LeLiTypeManager.obtenerTipoBasico(
               #nombre.getText(), #nombre );
         claseActualTipo.setScope( contexto.getCurrentScope() );
      }

      claseActualTipo.insertarAtributosSuperClase();
      contexto.insertarDecClase(##, #nombre, claseActualTipo);
   }
   listaMiembrosContexto[ claseActualTipo ] )
   {
      contexto.cerrarAmbitoClase();
      claseActualTipo.insertarMetodosSuperClase();
      ## = ast;
   }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

La regla comienza con la definición de dos variables locales. `claseActualTipo` sirve para guardar el tipo de la clase que se está reconociendo, mientras que `ast` sirve para guardar el AST que genera la regla (porque no se puede utilizar ## en varias acciones de una misma regla).

Después se reconocen los nodos iniciales del AST de declaración de las clases. Esto incluye la raíz (`RES_CLASE`), el nombre de la clase y la cláusula "extiende" con el nombre de la clase padre.

Tras reconocer los nodos iniciales comienza una acción en la que se inicializan las variables creadas al principio de la regla, es decir, `ast` y `claseActualTipo`. No hay mucho que decir sobre la variable `ast`: es simplemente una variable temporal para guardar el AST producido por la regla.

`claseActualTipo`, por su parte, sirve para guardar el tipo de la clase (la instancia de LeLiType que servirá para caracterizar la declaración de la clase en el ámbito global). La forma de iniciarla depende de si estamos reconociendo las clases "usuales" de un programa LeLi (definidas por un usuario) o los tipos por defecto del lenguaje. Volveremos a esto en la fase 4.

Las últimas dos líneas de la acción añaden los atributos de la superclase a la clase actual, y la insertan en el ámbito global, de manera que los métodos y constructores de la clase la pueden

utilizar.

Después de la acción se reconocen los atributos, constructores y métodos de la regla, con la regla `listaMiembrosContexto`. Esta regla es similar a `listaMiembros`, con la salvedad de que admite como parámetro el tipo de la clase que se está declarando:

```
listaMiembrosContexto [LeLiType claseActual]
   : #( LISTA_MIEMBROS ( decAtributoContexto[claseActual]
                       | decMetodoContexto[claseActual]
                       | decConstructorContexto[claseActual]
                       )*
      )
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

La última acción de la regla se encarga de "terminar" el ámbito de la clase. También inserta los métodos de la superclase en el ámbito de la clase, con el fin de acelerar búsquedas. Por último hace que `ast` sea el árbol generado por la regla (`##=ast`).

Declaraciones de métodos

Las declaraciones de métodos son reconocidas con la regla `decMetodoContexto`:

```
decMetodoContexto [LeLiType claseActual]
{
   ScopeAST ast = null;
   boolean abstracto = false;
   LeLiDecMetodoType metodoActual = null;
}
   : #( RES_METODO ( RES_ABSTRACTO {abstracto=true;} )?
        tr:tipo nombre:IDENT
        {
        contexto.abrirAmbitoMetodo(#nombre);

        LeLiType tret = (LeLiType)((TypeAST)#tr).getExpType();

        metodoActual =
           new LeLiDecMetodoType( #nombre.getText(),
                                  tret,
                                  claseActual,
                                  contexto.getCurrentScope() );
        if(abstracto) metodoActual.addModifier("abstracto");
        }
       listaDecParamsContexto[metodoActual]
       {
          ast = new ScopeAST( ##, contexto.getCurrentScope());
          contexto.insertarVariableMetodo(ast, #nombre, tret);
       }
       listaInstrucciones
      )
     {
       contexto.cerrarAmbitoMetodo();
       ## = ast;
       contexto.insertarDecMetodo(##, #nombre, metodoActual);
     }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

La regla comienza de forma parecida a `decClase`: tras declararse ciertas variables locales, se

reconoce la "cabecera" de la declaración. Esto incluye la raíz, `RES_METODO`; la palabra reservada `RES_ABSTRACTO` si procede; luego se reconoce el tipo del método (el tipo de la variable que devuelve – puede ser vacío); y después el nombre del método.

Después hay una acción en la que se crea el tipo de la declaración y el ámbito del método. Luego se reconocen los parámetros con la regla `listaDecParametrosContexto`. Una nueva acción inicializa la variable local ast e incluye la declaración de la variable del método en el ámbito del método, si éste no es vacío.

Tras todo este proceso ya es posible recorrer las instrucciones.

La acción al final de la regla termina el ámbito del método e inserta el AST producido por la regla en el ámbito de la clase correspondiente.

Declaraciones de constructores

La declaración de un constructor se reconoce con la regla `decConstructorContexto`:

```
decConstructorContexto[ LeLiType claseActual ]
{ LeLiDecConstructorType tipoConstructor = null; }
   : #( RES_CONSTRUCTOR
        {
           tipoConstructor =
             new LeLiDecConstructorType( claseActual,
                                         contexto.abrirAmbitoConstructor() );
        }
        listaDecParamsContexto [tipoConstructor]
        listaInstrucciones )
     {
        ## = new ScopeAST ( ##, contexto.cerrarAmbitoConstructor());
        contexto.insertarDecConstructor( ##, tipoConstructor );
     }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Como puede verse, las declaraciones de constructores se reconocen más sencillamente que las de métodos. No obstante la forma de reconocerlas es similar.

Tras la declaración de variables locales, se reconoce la cabecera de la declaración (constituida únicamente por la raíz del AST, `RES_CONSTRUCTOR`). Después hay una acción en la que se inicializa el tipo de la declaración. Luego se reconocen los parámetros y las instrucciones del constructor. Finalmente hay una acción que se encarga de cerrar el ámbito y registrar el constructor en el ámbito de la clase a la que pertenece.

Declaración de bucles

Implementar el comportamiento de los ámbitos de los bucles es muy sencillo, porque no se añaden declaraciones de bucles al ámbito del método al que pertenecen; solamente hay que crear y destruir el ámbito del bucle, y reconocer las instrucciones en su interior. Así, el bucle `mientras` se reconoce así:

```
instMientras
   : #( RES_MIENTRAS
        { contexto.abrirAmbitoMientras(); }
        expresion
        listaInstrucciones
      )
```

```
      { ## = new ScopeAST(##, contexto.cerrarAmbitoMientras()); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

El bucle `hacer-mientras` así:

```
instHacerMientras
   : #( RES_HACER
        { contexto.abrirAmbitoHacerMientras(); }
        listaInstrucciones
        expresion
      )
      { ## = new ScopeAST(##, contexto.cerrarAmbitoHacerMientras()); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Y el bucle `desde` de esta otra forma:

```
instDesde
   : #( RES_DESDE
        { contexto.abrirAmbitoDesde(); }
        listaExpresiones
        listaExpresiones
        listaExpresiones
        listaInstrucciones
      )
      { ## = new ScopeAST(##, contexto.cerrarAmbitoDesde()); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Instrucciones condicionales

Las instrucciones condicionales presentan la particularidad de que requieren un nuevo ámbito por cada *alternativa* de la instrucción. Dado que debemos "enganchar" los ámbitos en las raíces de los ASTs en los que se inician, deberemos añadir un ámbito en los nodos `RES_SI`, `BARRA_VERT` y `RES_OTRAS`, representando respectivamente el ámbito de la primera alternativa, de todas las alternativas intermedias y de la alternativa "otras" de una instrucción `si`. De esta forma, el código de la regla `instSi` será el siguiente:

```
instSi
{ Scope s = null; }
   : #( RES_SI
        { contexto.abrirAmbitoCondicional(); }
        expresion
        listaInstrucciones
        { s = contexto.cerrarAmbitoCondicional(); }
        (alternativaSi)*
      )
      { ## = new ScopeAST(##, s); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

El resto de los ámbitos se añaden en la regla `alternativasSi`:

```
alternativaSi
{ Scope s = null; }
   : ( #( BARRA_VERT
          { contexto.abrirAmbitoCondicional(); }
```

```
          expresion
          listaInstrucciones
          { s = contexto.cerrarAmbitoCondicional(); }
        )
      | #( RES_OTRAS
          { contexto.abrirAmbitoCondicional(); }
          listaInstrucciones
          { s = contexto.cerrarAmbitoCondicional(); }
        )
    )
    { ## = new ScopeAST(##, s); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Declaración de atributos

La declaración de un atributo no implica la creación de un ámbito; solamente la instroducción de una nueva declaración en un ámbito ya existente (el de la clase que posee el ámbito). Teniendo esto en cuenta, reconocer las declaraciones de atributos es relativamente sencillo. Las declaraciones de atributo se reconocen con la regla `decAtributoContexto`:

```
decAtributoContexto [LeLiType claseActual]
{ boolean abstracto = false; }
   : #( RES_ATRIBUTO (RES_ABSTRACTO {abstracto=true;})?
        t:tipo
        nombre:IDENT (v:expresion)? )
    {
        LeLiType tipo = (LeLiType)((TypeAST)#t).getExpType();
        AttributeType tipoAtributo =
        new AttributeType( RES_ATRIBUTO,
                           #nombre.getText(),
                           tipo,
                           claseActual );
        if(abstracto) tipoAtributo.addModifier("abstracto");
        contexto.insertarDecAtributo( ##, #nombre, tipoAtributo, #v);
    }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

La regla, a pesar de parecer aparatosa, es sencilla: la estructura completa de la declaración se reconoce, y después se crea el tipo de la declaración (el tipo es en este caso una instancia de `antlraux.context.types.AttributeType`). Si el atributo es abstracto, se añade el modificador "abstracto" al tipo. Finalmente, el método `insertarDecAtributo` de `contexto` se encarga de crear e introducir la declaración correspondiente.

Declaraciones de parámetros

La declaración de un parámetro tampoco implica la creación de un nuevo ámbito, solamente la inserción de una declaración. Las declaraciones de parámetros, además, no requieren de "tipos específicos" como los atributos o los métodos (pueden utilizarse los tipos básicos del sistema, "Entero", "Real", etc como tipos de las declaraciones[69]). La regla que reconoce las declaraciones de parámetros es `decParametroContexto`:

69 Esto ocurre porque los parámetros en LeLi no admiten modificadores (por ejemplo, para indicar paso por referencia o por valor). Si éste fuera el caso, sería necesario utilizar una clase de tipo específica para las declaraciones de parámetros.

```
decParametroContexto[LeLiMethodType decMetodo]
   : #(RES_PARAMETRO t:tipo nombre:IDENT)
     {
      LeLiType tipoParametro = (LeLiType)((TypeAST)#t).getExpType();
      decMetodo.addParam(tipoParametro);
        contexto.insertarDecParametro( ##, #nombre, tipoParametro );
     }
    ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Como puede verse, es muy sencilla.

`decParametroContexto` es utilizada por `listaDecParametrosContexto`:

```
listaDecParamsContexto [LeLiMethodType decMetodo]
   : #(LISTA_DEC_PARAMS (decParametroContexto[decMetodo])* )
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

7.5.4: Fase 2: Preparación de las expresiones

Las reglas de esta fase se limitan a "preparar" los ASTs que representan expresiones en LeLi, convirtiéndolos en instancias de `antlraux.context.asts.ExpressionAST`. Ningún valor de expresión (L-value, R-value, etc) es calculado en esta fase; simplemente se preparan los ASTs, de manera que no haya que crear ningún nuevo AST en la segunda pasada.

El código de la regla `expresion` será el siguiente:

```
expresion
   : ( #( OP_MAS expresion expresion )
     | #( OP_MENOS expresion expresion )
     | #( OP_ASIG expresion expresion )
     | #( OP_O expresion expresion )
     | #( OP_Y expresion expresion )
     | #( OP_IGUAL expresion expresion )
     | #( OP_DISTINTO expresion expresion )
     | #( OP_MAYOR expresion expresion )
     | #( OP_MENOR expresion expresion )
     | #( OP_MAYOR_IGUAL expresion expresion )
     | #( OP_MENOR_IGUAL expresion expresion )
     | #( OP_PRODUCTO expresion expresion )
     | #( OP_DIVISION expresion expresion )
     | #( OP_MENOS_UNARIO expresion)
     | #( OP_MASMAS      expresion)
     | #( OP_MENOSMENOS  expresion)
     | #( OP_NO expresion)
     | #( RES_ESUN acceso tipo)
     )
   { ## = new ExpressionAST(##); }
   | acceso
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Los accesos también son muy sencillos de reconocer en esta fase:

```
acceso : #(ACCESO raizAcceso (subAcceso)* )
         { ## = new ExpressionAST(##); }
       ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Hay que actuar de forma similar con `raizAcceso` y `subAcceso`:

```
raizAcceso : ( IDENT
             | RES_PARAMETRO
             | RES_ATRIBUTO
             | RES_SUPER
             )
             { ## = new ExpressionAST(##); }
           | literal
           | llamada
           | conversion
           | expresion
           ;
exception catch [RecognitionException ce] { reportError(ce); }

subAcceso : llamada
          | IDENT      { ## = new ExpressionAST(##); }
          | RES_SUPER  { ## = new ExpressionAST(##); }
          ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Finalmente, las reglas `literal`, `llamada` y `conversion` requieren de la misma operación:

```
literal : ( LIT_ENTERO
          | LIT_REAL
          | LIT_CADENA
          | LIT_NL
          | LIT_TAB
          | LIT_COM
          | LIT_CIERTO
          | LIT_FALSO
          )
          { ## = new ExpressionAST(##); }
        ;
exception catch [RecognitionException ce] { reportError(ce); }

llamada : ( #(LLAMADA IDENT listaExpresiones )
          | #(RES_CONSTRUCTOR listaExpresiones )
          )
          { ## = new ExpressionAST(##); }
        ;
exception catch [RecognitionException ce] { reportError(ce); }

conversion : #(RES_CONVERTIR expresion tipo)
             { ## = new ExpressionAST(##); }
           ;
exception catch [RecognitionException ce] { reportError(ce); }
```

7.5.5: Fase 3: Preparación de los tipos

La fase 3 del analizador está formada por una única regla, `tipo`. Los ASTs producidos por esta regla deben ser modificados para ser instancias de la clase `antlraux.context.asts.TypeAST`. Es decir, hay que realizar una conversión parecida a las de la fase 2.

En la preparación de tipos, sin embargo, las instancias de `TypeAST` se "rellenarán" con los datos pertinentes (en este caso el tipo), porque son necesarias en la zona 1 del analizador.

La regla `tipo` tendrá el siguiente aspecto:

```
tipo
{ Type t = null; }
   : ( TIPO_ENTERO   { t = LeLiTypeManager.tipoEntero;   }
     | TIPO_REAL     { t = LeLiTypeManager.tipoReal;     }
     | TIPO_BOOLEANO { t = LeLiTypeManager.tipoBooleano; }
     | TIPO_CADENA   { t = LeLiTypeManager.tipoCadena;   }
     | TIPO_VACIO    { t = LeLiTypeManager.tipoVacio;    }
     | i:IDENT       { t = contexto.obtenerTipo(i);      }
     )
   {
      ## = new TypeAST(##);
      ((TypeAST)##).setExpType(t);
   }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Nótese la simplicidad de la regla: cuando se reconocen tipos básicos de LeLi, se les asigna un AST cuyo tipo se obtiene del gestor de tipos predefinidos de LeLi (`LeLiTypeManager`). En el caso de los identificadores, se buscan el el contexto declaraciones de clases definidas por el usuario.

7.5.6: Fase 4: Preparación del ámbito global

La mayor parte del código de la fase 4 está contenido en la regla `instalarTiposBasicos`.

`instalarTiposBasicos` es invocada por la regla programa, al inicio de la Fase 1 del análisis:

```
programa: #(PROGRAMA instalarTiposBasicos (decClase)+) {...} ;
```

La regla `instalarTiposBasicos` será una regla que evaluará a "nada"; su cuerpo estará formado por una única acción:

```
instalarTiposBasicos: /* nada */
{
   << Cuerpo de la acción >>
}
```

Ahora solamente queda definir `<< Cuerpo de la acción >>`.

El fichero de tipos básicos

Hay varias maneras de definir los tipos por defecto de un lenguaje. Una de ellas consiste en crear los ASTs de sus declaraciones "a mano" y luego añadirlos como declaraciones:

```
AST decObjeto_Aserto =
   #( #[RES_METODO], #[RES_ABSTRACTO], #[TIPO_VACIO], #[IDENT, "aserto"],
      #( #[LISTA_PARAMS],
         #( #[RES_PARAMETRO], #[TIPO_BOOLEANO], #[IDENT, "condicion"]) ),
         #( #[RES_PARAMETRO], #[TIPO_CADENA], #[IDENT, "mensaje"] ) ) );

AST decObjeto =
   #( #[RES_CLASE], #[IDENT,"Objeto"], #(#[RES_EXTIENDE], #[IDENT,"Objeto"]),
      #(#[LISTA_MIEMBROS], decObjeto_aserto) );

... // Lo mismo con Cadena, Real, Booleano y Sistema
```

Pero este método es muy poco práctico: escribir los ASTs a mano es muy engorroso, lento y propenso a producir errores.

En lugar de definir los tipos predefinidos "a mano", utilizaremos un *fichero de definición*. Lo llamaremos `TiposBasicos.leli`, y será el siguiente:

```
clase ___Objeto
{
   método abstracto aserto(Booleano condición; Cadena mensaje)
   { }
}

clase ___Cadena
{
   método Entero longitud() { }
   método Cadena subCadena (Entero inicio, fin) { }
}

clase ___Entero {}

clase ___Real extiende ___Entero {}

clase ___Booleano {}

clase Sistema
{
   método abstracto imprime(Real r) { }
   método abstracto imprime(Booleano b) { }
   método abstracto imprime(Entero e) { }
   método abstracto imprime(Cadena c) { }
}
```

Así nos ahorramos la aparatosidad del AST generado a mano, aunque aparecen otros problemas:

- El reconocimiento `TiposBasicos.leli` es diferente del reconocimiento de un fichero LeLi normal y corriente; el análisis debe realizarse de formas diferentes. Para ello, la clase `LeLiContext` proporciona el atributo `tiposBasicosLeidos`, que es falso hasta que se lee el fichero.
- Será necesario reconocer completamente el fichero. Esto implica la instanciación de un nuevo analizador léxico, uno sintáctico y uno semántico. Afortunadamente podremos utilizar los ya existentes.
- Los tipos se utilizan antes de ser declarados. Por ejemplo, la clase `Cadena` se utiliza en un método de la clase `Objeto`. Para mitigar este problema será necesario introducir unos tipos "falsos" en el ámbito global, e ir sustituyéndolos por los verdaderos al ir reconociéndolos.

- No se pueden utilizar los nombres de los tipos por defecto (“Entero”, “Real”, ...) porque el analizador léxico los reconoce como palabras reservadas, y el analizador sintáctico espera un `IDENT`. Así que precedemos los nombres de los tipos básicos con dos caracteres de subrayado, que son eliminados durante el análisis.
- Nótese que no proporcionamos implementación alguna para los métodos de los tipos predefinidos; consúltese el siguiente capítulo para más información.

El código de la acción de `instalarTiposBasicos` es como sigue. Comienza instalando los tipos “falsos”, como decíamos anteriormente:

```
instalarTiposBasicos :
{
   contexto.instalarTiposBasicos(); // Instala los tipos “falsos”
```

Después creando un flujo para el fichero `TiposBasicos.leli`:

```
   String nombreFichero = "TiposBasicos.leli";
   FileInputStream is = null;
   try
   {
      is = new FileInputStream(nombreFichero);
   }catch (FileNotFoundException fnfe){
      throw new RecognitionException(
         "El fichero '"+nombreFichero+"' no se encontró");
   }
```

Luego creando un analizador léxico:

```
   LeLiLexer lexer = new LeLiLexer(is);
   lexer.setTokenObjectClass("antlraux.util.LexInfoToken");
   lexer.setFilename(nombreFichero);
```

Y uno sintáctico:

```
   // Crea el analizador sintáctico
   LeLiErrorRecoveryParser parser =
      new LeLiErrorRecoveryParser(lexer,logger);
   parser.setASTNodeClass("antlraux.util.LexInfoAST");
   parser.setFilename(nombreFichero);

   try{
      parser.programa();
   }catch (TokenStreamException tse){
      throw new RecognitionException (
         "Excepción TokenStreamException encontrada "+
         "mientras se leía el fichero TiposBasicos.leli");
   }
```

Obtenemos el AST generado por el analizador sintáctico:

```
   AST ast = parser.getAST();
```

Llegados a este punto debemos crear un analizador semántico de primera pasada *dentro de una acción del analizador semántico de primera pasada*. Por eso el analizador tiene dos constructores: uno para el que “lee” el fichero `TiposBasicos.leli` y otro para el que lee los ficheros normales de LeLi.

```
   // Crea el analizador semántico
```

```
   LeLiSymbolTreeParser tp =
      new LeLiSymbolTreeParser(logger, contexto);
```

Una vez creado el analizador semántico, hay que lanzar el análisis. Normalmente se invocaría el método `tp.programa()`. Ahora bien, dado que programa invoca la regla `insertarTiposBasicos`, tendríamos una recursión infinita si lo hiciéramos. En lugar de ello vamos a utilizar una regla especial, `programaBasico`:

```
   tp.programaBasico(ast);
```

`programaBasico` es como `programa`, pero sin `insertarTiposBasicos`:

```
programaBasico
   : #( PROGRAMA (decClase)+ )
      { ## = new ScopeAST(##, contexto.getCurrentScope()); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

`insertarTiposBasicos` y `programaBasico` son las dos únicas reglas de la fase 4 del analizador.

Pero volvamos a `instertarTiposBasicos`. Una vez que se ha invocado `programaBasico`, los tipos básicos están insertados en el ámbito global. Solamente queda realizar unos pequeños ajustes.

Para empezar, el AST de la lista de clases generada se devuelve. Esto es un arreglo meramente visual: al devolver éste AST podremos ver los árboles de las definiciones de los tipos predefinidos en el AST del programa:

```
   // Devolver el primer hijo del programa, que lleva "atados"
   // sus "hermanos" (siblings)
   ## = (LexInfoAST)(tp.getAST().getFirstChild());
```

Finalmente hay unas cuantas acciones adicionales: se activan las comprobaciones y se marca el contexto, para indicar que los tipos básicos ya han sido leídos.

```
   contexto.activarComprobaciones();

   contexto.tiposBasicosLeidos = true;
}
;
```

La fase 4 es la más “dispersa” del analizador. Para no tener que utilizar dos analizadores diferentes, en algunas reglas del analizador y algunos métodos de `LeLiContext` se consulta la variable `tiposBasicosLeidos` para ver qué es lo que hay que hacer. Una implementación un poco más “limpia”, pero más complicada de comprender, sería utilizar la herencia para modificar el comportamiento de las reglas y métodos que lo necesitasen.

7.5.7: Fichero LeLiSymbolTreeParser.g

A continuación mostramos el listado completo del fichero:

```
header
{

package leli;

/*-------------------------------------------------*\
 | Un intérprete para un Lenguaje Limitado(LeLi) |
```

```
  | ____________________________________________ |
  |                                              |
  | ANALISIS SEMÁNTICO -- Instalación de símbolos |
  | ____________________________________________ |
  |                                              |
  |            Enrique J. Garcia Cota            |
\*---------------------------------------------*/

import antlraux.context.ContextException;
import antlraux.context.Scope;
import antlraux.context.asts.*;
import antlraux.context.types.Type;
import antlraux.context.types.AttributeType;

import leli.types.*;

import antlraux.util.LexInfoAST;
import antlraux.util.Logger;

import antlr.TokenStreamException;

import java.io.FileInputStream;
import java.io.FileNotFoundException;

}

class LeLiSymbolTreeParser extends LeLiTreeParser;

options
{
   importVocab=LeLiParserVocab;
   exportVocab=LeLiSymbolTreeParserVocab;
   buildAST = true;
   ASTLabelType = antlraux.util.LexInfoAST;
}

tokens
{
   TIPO_ERROR  = "error";
   METACLASE   = "metaclase" ;
}

{
   /**
    * La clase {link LeLiContext} es una fachada que facilita la manipulación
    * contextual
    **/
   private LeLiContext contexto;

   /** El logger de la clase **/
   private Logger logger;

   /** Constructor habitual **/
   public LeLiSymbolTreeParser(Logger logger)
   throws RecognitionException
   {
      this();
```

```
      this.logger = logger;
      this.contexto = new LeLiContext(logger);

      setASTNodeClass("antlraux.util.LexInfoAST");
   }

   /** Constructor privado (utilizado para leer TiposBasicos.leli **/
   public LeLiSymbolTreeParser(Logger logger, LeLiContext contexto)
   throws RecognitionException
   {
      this();

      this.logger = logger;
      this.contexto = contexto;

      setASTNodeClass("antlraux.util.LexInfoAST");
   }

   public LeLiContext obtenerContexto()
   { return contexto; }

   public void reportError( String msg,
                            String filename,
                            int line,
                            int column )
   {
      if(null==logger)
      {
         logger = new Logger("error", System.err);
      }
      logger.log( msg, 1, filename, line, column);
   }

   public void reportError(RecognitionException e)
   {
      reportError( e.getMessage(), e.getFilename(),
                   e.getLine(), e.getColumn() );
      e.printStackTrace(System.out);
   }
}

// ------------------- FASE 1: Creación de ámbitos -----------------
//
// La fase 1 consiste en:
//  * Crear la jerarquía de ámbitos
//  * Insertar las declaraciones de parámetros, variables, atributos, clases,
//    métodos y constructores en sus respectivos ámbitos
//  * Asociar el ámbito adecuado a la raíz de los ASTs de objetos con ámbito
//    (clases, métodos, constructores, bucles y alternativas condicionales)
//


/** Regla que sirve para recorrer el AST de definición de un programa LeLi **/
programa
   : #( PROGRAMA
        instalarTiposBasicos
        (decClase)+ )
      { ## = new ScopeAST(##, contexto.getCurrentScope()); }
```

```
   ;
exception catch [RecognitionException ce] { reportError(ce); }

/** Declaración de una clase **/
decClase
{
   LeLiType claseActualTipo = null;
   ScopeAST ast = null;
}:
   #( RES_CLASE nombre:IDENT #(RES_EXTIENDE padre:IDENT)
   {
      ast = new ScopeAST(##, contexto.abrirAmbitoClase (#nombre));

      if(contexto.tiposBasicosLeidos == true)
      {
         LeLiType tipoPadre = contexto.obtenerTipo(#padre);

         claseActualTipo =
         new LeLiType( #nombre.getText(),
                       tipoPadre,
                       contexto.getCurrentScope(),
                       new LeLiMetaType(#nombre.getText()) );
      } else {
         claseActualTipo =
            LeLiTypeManager.obtenerTipoBasico(
               #nombre.getText(), #nombre );
         claseActualTipo.setScope( contexto.getCurrentScope() );
      }

      claseActualTipo.insertarAtributosSuperClase();
      contexto.insertarDecClase(##, #nombre, claseActualTipo);
   }
   listaMiembrosContexto[ claseActualTipo ] )
   {
      contexto.cerrarAmbitoClase();
      claseActualTipo.insertarMetodosSuperClase();
      ## = ast;
   }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

/** Lista de miembros de una clase **/
listaMiembrosContexto [LeLiType claseActual]
   : #( LISTA_MIEMBROS ( decAtributoContexto[claseActual]
                       | decMetodoContexto[claseActual]
                       | decConstructorContexto[claseActual]
                       )*
      )
   ;
exception catch [RecognitionException ce] { reportError(ce); }

/** Declaración de un método **/
decMetodoContexto [LeLiType claseActual]
{
   ScopeAST ast = null;
   boolean abstracto = false;
   LeLiDecMetodoType metodoActual = null;
}
```

```
   : #( RES_METODO ( RES_ABSTRACTO {abstracto=true;} )?
        tr:tipo nombre:IDENT
        {
        contexto.abrirAmbitoMetodo(#nombre);

        LeLiType tret = (LeLiType)((TypeAST)#tr).getExpType();

        metodoActual =
           new LeLiDecMetodoType( #nombre.getText(),
                                  tret,
                                  claseActual,
                                  contexto.getCurrentScope() );
        if(abstracto) metodoActual.addModifier("abstracto");
        }
       listaDecParamsContexto[metodoActual]
       {
          ast = new ScopeAST( ##, contexto.getCurrentScope());
          contexto.insertarVariableMetodo(ast, #nombre, tret);
       }
       listaInstrucciones
      )
     {
       contexto.cerrarAmbitoMetodo();
       ## = ast;
       contexto.insertarDecMetodo(##, #nombre, metodoActual);
     }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

/** Declaración de un atributo **/
decConstructorContexto[ LeLiType claseActual ]
{ LeLiDecConstructorType tipoConstructor = null; }
   : #( RES_CONSTRUCTOR
        {
           tipoConstructor =
             new LeLiDecConstructorType( claseActual,
                                         contexto.abrirAmbitoConstructor() );
        }
        listaDecParamsContexto [tipoConstructor]
        listaInstrucciones )
     {
        ## = new ScopeAST ( ##, contexto.cerrarAmbitoConstructor());
        contexto.insertarDecConstructor( ##, tipoConstructor );
     }
   ;
exception catch [RecognitionException ce] { reportError(ce); }


/** Bucle mientras **/
instMientras
   : #( RES_MIENTRAS
        { contexto.abrirAmbitoMientras(); }
        expresion
        listaInstrucciones
      )
     { ## = new ScopeAST(##, contexto.cerrarAmbitoMientras()); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

```
/** Bucle hacer-mientras **/
instHacerMientras
   : #( RES_HACER
        { contexto.abrirAmbitoHacerMientras(); }
        listaInstrucciones
        expresion
      )
     { ## = new ScopeAST(##, contexto.cerrarAmbitoHacerMientras()); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

/** Bucle desde **/
instDesde
   : #( RES_DESDE
        { contexto.abrirAmbitoDesde(); }
        listaExpresiones
        listaExpresiones
        listaExpresiones
        listaInstrucciones
      )
     { ## = new ScopeAST(##, contexto.cerrarAmbitoDesde()); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

/** Instrucción condicional **/
instSi
{ Scope s = null; }
   : #( RES_SI
        { contexto.abrirAmbitoCondicional(); }
        expresion
        listaInstrucciones
        { s = contexto.cerrarAmbitoCondicional(); }
        (alternativaSi)*
      )
     { ## = new ScopeAST(##, s); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

/** Alternativas de la instrucción condicional **/
alternativaSi
{ Scope s = null; }
   : ( #( BARRA_VERT
          { contexto.abrirAmbitoCondicional(); }
          expresion
          listaInstrucciones
          { s = contexto.cerrarAmbitoCondicional(); }
        )
       | #( RES_OTRAS
            { contexto.abrirAmbitoCondicional(); }
            listaInstrucciones
            { s = contexto.cerrarAmbitoCondicional(); }
        )
     )
     { ## = new ScopeAST(##, s); }
    ;
exception catch [RecognitionException ce] { reportError(ce); }
```

```
/** Declaración de un atributo **/
decAtributoContexto [LeLiType claseActual]
{ boolean abstracto = false; }
   : #( RES_ATRIBUTO (RES_ABSTRACTO {abstracto=true;})?
        t:tipo
        nombre:IDENT (v:expresion)? )
     {
        LeLiType tipo = (LeLiType)((TypeAST)#t).getExpType();
        AttributeType tipoAtributo =
        new AttributeType( RES_ATRIBUTO,
                           #nombre.getText(),
                           tipo,
                           claseActual );
        if(abstracto) tipoAtributo.addModifier("abstracto");
        contexto.insertarDecAtributo( ##, #nombre, tipoAtributo, #v);
     }
    ;
exception catch [RecognitionException ce] { reportError(ce); }

listaDecParamsContexto [LeLiMethodType decMetodo]
   : #(LISTA_DEC_PARAMS (decParametroContexto[decMetodo])* )
   ;
exception catch [RecognitionException ce] { reportError(ce); }

/** Declaración de un parámetro **/
decParametroContexto[LeLiMethodType decMetodo]
   : #(RES_PARAMETRO t:tipo nombre:IDENT)
     {
      LeLiType tipoParametro = (LeLiType)((TypeAST)#t).getExpType();
      decMetodo.addParam(tipoParametro);
        contexto.insertarDecParametro( ##, #nombre, tipoParametro );
     }
    ;
exception catch [RecognitionException ce] { reportError(ce); }

// -------------- FASE 2: Preparación de las expresiones ----------
//
// La fase 2 consiste en transformar los AST de las expresiones
// en instancias de antlraux.context.asts.ExpressionAST. Aunque se
// crean los nodos, están "vacíos"; se rellenarán en la segunda pasada
//

expresion
   : ( #( OP_MAS expresion expresion )
     | #( OP_MENOS expresion expresion )
     | #( OP_ASIG expresion expresion )
     | #( OP_O expresion expresion )
     | #( OP_Y expresion expresion )
     | #( OP_IGUAL expresion expresion )
     | #( OP_DISTINTO expresion expresion )
     | #( OP_MAYOR expresion expresion )
     | #( OP_MENOR expresion expresion )
     | #( OP_MAYOR_IGUAL expresion expresion )
     | #( OP_MENOR_IGUAL expresion expresion )
     | #( OP_PRODUCTO expresion expresion )
     | #( OP_DIVISION expresion expresion )
     | #( OP_MENOS_UNARIO expresion)
     | #( OP_MASMAS      expresion)
```

```
     | #( OP_MENOSMENOS  expresion)
     | #( OP_NO expresion)
     | #( RES_ESUN acceso tipo)
     )
   { ## = new ExpressionAST(##); }
   | acceso
   ;
exception catch [RecognitionException ce] { reportError(ce); }

acceso : #(ACCESO raizAcceso (subAcceso)* )
         { ## = new ExpressionAST(##); }
       ;
exception catch [RecognitionException ce] { reportError(ce); }

raizAcceso : ( IDENT
             | RES_PARAMETRO
             | RES_ATRIBUTO
             | RES_SUPER
             )
             { ## = new ExpressionAST(##); }
           | literal
           | llamada
           | conversion
           | expresion
           ;
exception catch [RecognitionException ce] { reportError(ce); }

literal : ( LIT_ENTERO
          | LIT_REAL
          | LIT_CADENA
          | LIT_NL
          | LIT_TAB
          | LIT_COM
          | LIT_CIERTO
          | LIT_FALSO
          )
          { ## = new ExpressionAST(##); }
        ;
exception catch [RecognitionException ce] { reportError(ce); }

llamada : ( #(LLAMADA IDENT listaExpresiones )
          | #(RES_CONSTRUCTOR listaExpresiones )
          )
          { ## = new ExpressionAST(##); }
        ;
exception catch [RecognitionException ce] { reportError(ce); }

conversion : #(RES_CONVERTIR expresion tipo)
             { ## = new ExpressionAST(##); }
           ;
exception catch [RecognitionException ce] { reportError(ce); }

subAcceso : llamada
          | IDENT      { ## = new ExpressionAST(##); }
          | RES_SUPER  { ## = new ExpressionAST(##); }
          ;
exception catch [RecognitionException ce] { reportError(ce); }
```

```
// -------------- FASE 3: Preparación de los tipos ----------
//
// En esta fase se transforman los nodos reconocidos por la regla
// "tipo" y se convierten en instancias de antlraux.context.asts.TypeAST.
// Los nodos se inicializan con el tipo que representan
//

tipo
{ Type t = null; }
   : ( TIPO_ENTERO   { t = LeLiTypeManager.tipoEntero;   }
     | TIPO_REAL     { t = LeLiTypeManager.tipoReal;     }
     | TIPO_BOOLEANO { t = LeLiTypeManager.tipoBooleano; }
     | TIPO_CADENA   { t = LeLiTypeManager.tipoCadena;   }
     | TIPO_VACIO    { t = LeLiTypeManager.tipoVacio;    }
     | i:IDENT       { t = contexto.obtenerTipo(i);      }
     )
   {
      ## = new TypeAST(##);
      ((TypeAST)##).setExpType(t);
   }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

// ------------ Fase 4: Instalación de tipos básicos ---------------
//
// Las reglas de esta fase sirven para insertar los tipos predefinidos
// del lenguaje (también llamados "tipos básicos") en el ámbito global
// antes de comenzar a reconocer un nuevo reconocimiento.
//

programaBasico
   : #( PROGRAMA (decClase)+ )
      { ## = new ScopeAST(##, contexto.getCurrentScope()); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

instalarTiposBasicos : /* nada */
{
   contexto.instalarTiposBasicos();

   String nombreFichero = "TiposBasicos.leli";
   FileInputStream is = null;
   try
   {
      is = new FileInputStream(nombreFichero);
   }catch (FileNotFoundException fnfe){
      throw new RecognitionException(
         "El fichero '"+nombreFichero+"' no se encontró");
   }

   LeLiLexer lexer = new LeLiLexer(is);
   lexer.setTokenObjectClass("antlraux.util.LexInfoToken");
   lexer.setFilename(nombreFichero);

   LeLiErrorRecoveryParser parser =
      new LeLiErrorRecoveryParser(lexer,logger);
   parser.setASTNodeClass("antlraux.util.LexInfoAST");
```

```
   parser.setFilename(nombreFichero);

   try{
      parser.programa();
   }catch (TokenStreamException tse){
      throw new RecognitionException (
         "Excepción TokenStreamException encontrada "+
         "mientras se leía el fichero TiposBasicos.leli");
   }

   AST ast = parser.getAST();

   LeLiSymbolTreeParser tp =
      new LeLiSymbolTreeParser(logger, contexto);

   tp.programaBasico(ast);

   // Devolver el primer hijo del programa, que lleva "atados"
   // sus "hermanos" (siblings)
   ## = (LexInfoAST)(tp.getAST().getFirstChild());

   contexto.activarComprobaciones();

   contexto.tiposBasicosLeidos = true;
}
;
```

7.5.8: Notas finales sobre el analizador

Mucha de la funcionalidad de este analizador se realiza entre bastidores; la clase `LeLiContext` ejerce de "fachada" entre el analizador y la jerarquía de ámbitos de LeLi, de manera que en el analizador no es necesario, por ejemplo, saber que el tipo del ámbito global es `leli.scopes.GlobalScope` o que el de los métodos es `leli.scopes.MethodScope`, que es una subclase del de los bucles e instrucciones codicionales, `leli.scopes.InternScope`. Simplemente se utilizan métodos como `abrirAmbitoBucle` o `insertarDecParametro`, y `LeLiContext` se encarga de realizar todos los castings y las comprobaciones necesarias. En caso de que alguna de estas comprobaciones no sea correcta, se lanza una excepción del tipo `antlraux.context.ContextException`, que al ser una subclase `antlr.RecognitionException` se transmite adecuadamente al analizador.

Otra razón para utilizar `LeLiContext` como una clase independiente en lugar de meter todo el código en acciones del analizador es que de esta forma `LeLiContext` puede ser utilizado tanto en la primera pasada como en la segunda.

Cambiando de tema, hay que reseñar que no hemos justificado la necesidad de tener que "guardar" el ámbito actual en nodos especiales del árbol (las instancias de `ScopeAST`). No se utilizan en ninguna parte del analizador ¿por qué no limitarse a simplemente construir la jerarquía de ámbitos?

Es evidente – ¡porque hace falta durante la segunda pasada! Lo veremos más detenidamente en la fase 1 del próximo analizador.

Sección 7.6: Comprobación de tipos – segunda pasada

7.6.1: Introducción

En la primera pasada del analizador hemos construido la jerarquía de ámbitos de nuestro programa. En los ámbitos de dicha jerarquía hemos incluido las declaraciones de todas las clases, atributos, métodos, constructores y parámetros pertinentes, de manera que están disponibles en la segunda pasada. Así, las referencias circulares o recursivas son completamente posibles.

Objetivos

La tarea más importante de la segunda pasada será completar los datos de los `ExpressionAST` que `LeLiSymbolTreeParser` se encargó de crear. Es decir, habrá que calcular el L-value, R-value, E-value, y el tipo de todas y cada una de las expresiones del programa.

Dos tareas secundarias también son necesarias: por un lado es necesario mantener constantemente el "ámbito actual" – cuando se "entra" en el cuerpo de una clase se debe cambiar el ámbito actual al ámbito de dicha clase, y si luego se entra en un método de dicha clase volver a cambiarlo, hasta que se termine. La otra tarea consiste en insertar las declaraciones de las *variables* en los ámbitos. Recordemos que las variables no pueden ser utilizadas en referencias circulares (no pueden ser utilizadas antes de ser declaradas). Para evitarlo, no se insertan durante la primera pasada, sino durante la segunda.

Fases del análisis

De nuevo por "fase" entenderemos "un conjunto de reglas que sirven para realizar una tarea". Hemos dividido el análisis de la segunda pasada en cuatro fases:

- En la fase 1 se mantiene el "ámbito actual".
- En la fase 2 se insertan las declaraciones de variables en el ámbito actual.
- En la fase 3 se calculan las propiedades (L-value, R-value, etc) de las expresiones de LeLi, excepto de los accesos.
- En la fase 4 calculamos las propiedades de las expresiones de los accesos.

7.6.2: Definición del analizador

Cabecera

En la cabecera del fichero `LeLiTypeCheckTreeParser.g` encontramos las opciones habituales: primero la de pertenencia al paquete `leli`, seguida de las importaciones.

```
header
{

package leli;

import ... ; // Importaciones varias

}
```

Luego viene la declaración de la clase del analizador. `LeLiTypeCheckTreeParser` es un iterador de árboles del lenguaje LeLi, y por lo tanto es una subclase de `LeLiTreeParser`:

```
class LeLiTypeCheckTreeParser extends LeLiTreeParser;
```

Zona de opciones

Las opciones del analizador son las que se muestran a continuación:

```
options
{
   importVocab=LeLiSymbolTreeParserVocab;
   exportVocab=LeLiTypeCheckTreeParserVocab;
   buildAST = false;
   ASTLabelType = antlraux.util.LexInfoAST;
}
```

Tras las opciones de importación y exportación de vocabulario, vemos que `buildAST` está desactivado; no es necesario construir ningún nuevo AST porque en la primera pasada ya se construyeron todos los nodos necesarios, aunque los nodos del tipo `ExpressionAST` no tengan sus propiedades iniciadas al valor correcto.

La última opción, como el lector ya debería saber, automatiza los "castings" de los nodos, haciendo que todos se vean como instancias de `LexInfoAST` en las acciones.

Zona de tokens

No se define ningún token nuevo en este analizador, así que la zona de tokens es inexistente.

Zona de código nativo

La zona de código nativo comienza con las habituales declaraciones de atributos del analizador. En este caso son tres:

```
   public Logger logger=null;
   private LeLiContext contexto;
   private LeLiType claseActual = null;
```

El primero de ellos es el logger del analizador. `contexto` sirve para guardar una referencia al `contexto` resultante de la primera pasada (`LeLiSymbolTreeParser.contexto`). Por último, `claseActual` es una variable que permitirá al analizador conocer la clase actual. Esto es necesario para implementar los accesos que involucran utilizar la palabra reservada `super`.

Tras los atributos hay declaraciones de atributos estáticos que representan dos tipos "especiales". Son los tipos que se utilizarán para caracterizar los accesos que comienzan con la palabras reservadas "parámetro" o "atributo". Se diferenciarán del resto de los parámetros por sus etiquetas, que son `RES_PARAMETRO` y `RES_ATRIBUTO` respectivamente.

```
   public static final LeLiType tipoParametroAux =
      new LeLiType(RES_PARAMETRO,"parámetro", null, null, null);

   public static final LeLiType tipoAtributoAux =
      new LeLiType(RES_ATRIBUTO,"atributo", null, null, null);
```

Estos dos tipos no han sido incluidos en `LeLiTypeManager` como el resto de los tipos predefinidos de LeLi porque solamente se utilizan en la segunda pasada.

Tras las declaraciones de atributos tenemos la el constructor de la clase:

```
LeLiTypeCheckTreeParser( Logger logger,
                         LeLiContext contexto )
{
   this();
   this.logger = logger;
   this.contexto = contexto;
}
```

Y tras el constructor hay varias formas de `reportError`:

```
public void reportError( String msg,
                         String filename,
                         int line,
                         int column )
{
   if(null==logger)
   {
      logger = new Logger("error", System.err);
   }
   logger.log( msg, 1, filename, line, column);
}

public void reportError(RecognitionException e)
{
   reportError( e.getMessage(), e.getFilename(),
                e.getLine(), e.getColumn() );
}

public void reportError( String msg,
                         LexInfo info )
{
   reportError( msg, info.getFilename(),
                info.getLine(), info.getColumn()) ;
}
```

Los dos primeros son los habituales. El tercero es nuevo: se proporciona un mensaje y un LexInfo, y se genera el log de error a partir de él.

7.6.3: Fase 1 – Mantener el ámbito actual

Durante la primera pasada nos hemos tomado la molestia de "insertar" ciertas referencias a ámbitos en ciertos nodos del tipo `ScopeAST`. ¿Por qué?

La razón es sencilla: necesitamos saber cuándo cambiar de ámbito durante la segunda fase. Es cierto que podríamos realizarlo de otra forma: cada vez que llegáramos a la declaración de una clase nueva, tendríamos que buscar en el ámbito global la declaración de la clase, obtener de ella el ámbito, y hacerlo el "ámbito actual". Habría que realizar operaciones parecidas para cada método, constructor, bucle o alternativa condicional. Esto se convertiría en una cantidad importante de líneas de código, más de las necesarias para realizarlo de esta otra manera (si excluimos el código de la clase `ScopeAST`, que es amablemente proporcionada por `antlraux`). Además, el código sería más ineficiente sin `ScopeAST`s.

Para ir cambiando el ámbito actual utilizaremos dos métodos específicos de la fase 1 que añadiremos a la zona de código nativo:

```
   public void cambiaAmbitoActual(LexInfoAST ast)
   {
      Scope nuevoAmbito = ((ScopeAST)ast).getScope();
      contexto.setCurrentScope(nuevoAmbito);
   }

   public void cierraAmbitoActual()
   { contexto.toFather(); }
```

Estos dos métodos se encargan de obtener el ámbito a partir de una instancia de `ScopeAST` y de "terminarlo". Cuando un ámbito se termina deja de ser el actual, pasando a serlo su padre.

¡Empecemos!

Programa

El primer ámbito que controlaremos será el ámbito inicial, al leer la raíz del AST:

```
programa
   : {  cambiaAmbitoActual( #programa ); }
     #( PROGRAMA (decClase)+ )
    ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Normalmente las acciones de esta fase comenzarán con una acción en la que se invoque a `cambiaAmbitoActual`, y finalizarán con otra en la que se invoque a `cierraAmbitoActual`. En este caso, no obstane, no se cierra el ámbito, pues cerrar el ámbito global dejaría a `LeLiContext` en una situación peligrosa (`currentScope==null`).

Nótese que capturamos el error para que el logger del analizador pueda dar cuenta de él. Los manejadores de excepciones de esta fase del analizador son todos iguales, así que no hablaremos más de ellos.

Declaraciones de clases

Las declaraciones de clases se reconocen fácilmente:

```
decClase
   : { cambiaAmbitoActual( #decClase ); }
     #( RES_CLASE nombre:IDENT
        { claseActual = contexto.obtenerTipo(nombre); }
        clausulaExtiende
        listaMiembros )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Lo único digno de ser resaltado en esta regla es que se actualiza el atributo `claseActual`.

Declaraciones de métodos y constructores

Las declaraciones de métodos y constructores son sencillas; simplemente se reconocen los elementos que las forman, abriendo y cerrando los ámbitos convenientemente.

```
decMetodo
   : { cambiaAmbitoActual( #decMetodo ); }
     #( RES_METODO ( RES_ABSTRACTO )? tipo
        IDENT listaDecParams listaInstrucciones
      )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

decConstructor
   : { cambiaAmbitoActual( #decConstructor ); }
     #( RES_CONSTRUCTOR listaDecParams listaInstrucciones )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Bucles

Y los bucles también son muy sencillos. Simplemente hay que reconocerlos, abriendo y cerrando ámbitos...

```
instMientras
   : { cambiaAmbitoActual( #instMientras ); }
    #( RES_MIENTRAS expresion listaInstrucciones )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

instHacerMientras
   : { cambiaAmbitoActual( #instHacerMientras ); }
     #( RES_HACER listaInstrucciones expresion )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

instDesde
   : { cambiaAmbitoActual( #instDesde ); }
     #( RES_DESDE
        listaExpresiones listaExpresiones listaExpresiones
         listaInstrucciones )
      { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Instrucciones condicionales

Las reglas que reconocen las instrucciones condicionales son tan sencillas como las anteriores:

```
instSi
   : { cambiaAmbitoActual( #instSi ); }
     #( RES_SI expresion listaInstrucciones
        { cierraAmbitoActual(); }
        (alternativaSi)* )
   ;
exception catch [RecognitionException ce] { reportError(ce); }

alternativaSi
```

```
   : { cambiaAmbitoActual ( #alternativaSi ); }
     ( #(BARRA_VERT expresion listaInstrucciones)
     | #(RES_OTRAS listaInstrucciones)
     )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

7.6.4: Fase 2: Adición de las variables locales a los ámbitos

Se trata de una fase muy corta; solamente cuenta con una regla, `instDecVar`:

```
instDecVar
   : #(INST_DEC_VAR t:tipo nombre:IDENT (valor:expresion|le:listaExpresiones)?)
     {
        LeLiType type = (LeLiType) ((TypeAST)t).getExpType();
        contexto.insertarDecVariable( #instDecVar, nombre, type, valor, le);
     }
   ;
exception catch [RecognitionException ce] { reportError(ce); }
```

Esta regla es un calco de la regla `decParametro` de la primera pasada. Las variables de LeLi pueden ser caracterizadas directamente con su tipo (no necesitan un tipo "especial" como los atributos o los métodos).

En la siguiente fase es donde las cosas se complican.

7.6.5: Fase 3: Expresiones

Llamaremos *características de las expresiones* al tipo, L-value, R-value y E-value. En esta fase del analizador calcularemos las características de las expresiones de cada programa. Por lo tanto vamos a estar tratanto con los ASTs que se reconocen con la regla expresion.

El modo de actuación será el siguiente. Para cada expresión o conjunto de expresiones:

- Las expresiones de esta fase están formadas por un operador y uno o más operandos. El primer paso es comprobar que pueden leerse los operandos que deben poder leerse y pueden escribirse los que deben poder escribirse. Es decir, hay que comprobar que los L-values y R-values de los operandos son correctos.
- Después calcularemos los valores de L-value y R-value para la expresión que estemos recorriendo.
- Después calcularemos el tipo. Deberemos tener en cuenta incompatibilidades de tipos. En caso de error irreparable, haremos que el tipo de la expresión sea `TIPO_ERROR`.
- Finalmente calcularemos, si es posible, el E-value de la expresión.

Todas las expresiones de LeLi se reconocen con la regla expresion, así que por ella debemos empezar. Para simplificar un poco las cosas en el nivel la hemos dividido en cuatro sub reglas:

```
expresion : expresionBinaria
          | expresionUnaria
          | expresionEsUn
          | acceso
          ;
```

Las veremos todas en este apartado excepto `acceso` y sus reglas asociadas, que son objeto de estudio en el siguiente.

errorExpresion

Durante las fases 2 y tres, frecuentemente tenemos que hacer comprobaciones que no siempre son satisfechas por el código del usuario de LeLi. Por ejemplo, puede intentar restar un entero a una cadena, o intentar leer una expresión sin L-value. En tal caso se lanzará una excepción, que deberá ser capturada por el manejador de excepciones adecuado. El mensaje de error será mostrado por pantalla de la forma usual, pero además será necesario modificar las características del nodo donde se haya producido el error, para asegurarse de que no se producen "cascadas de errores". Para ello debemos hacer que las características de la expresión donde se ha producido el error tomen valores "seguros". Los valores seguros son cierto para el L-value y el R-value, el tipo error para el tipo y null para el E-value de la expresión. De esta forma, podemos utilizar el método `errorExpresion` (que debe ser añadido en la zona de código nativo del iterador) para automatizar el proceso en el caso de que se produzca un error:

```
public void errorExpresion( RecognitionException re,
                            ExpressionAST ast )
{
   errorExpresion( re, ast, true, true,
                   LeLiTypeManager.tipoError,null);
}
```

Este método se limita a invocar una versión "extendida", que permite especificar los valores seguros de la expresión en lugar de los que se utilizan por defecto:

```
public void errorExpresion( RecognitionException re, ExpressionAST ast,
                            boolean Rvalue, boolean LValue,
                            LeLiType tipo, Object EValue )
{
   reportError(re);
   ast.setExpType(tipo);
   ast.setRValue(RValue);
   ast.setLValue(LValue);
   ast.setEValue(EValue);
}
```

Ambas formas de errorExpresion pueden ser utilizadas en los manejadores de excepciones de las reglas de esta fase y la siguiente. Quizás se verá más claro cuando las veamos.

Expresión "esUn"

Esta expresión es la que utiliza la palabra reservada "esUn". La sintaxis es un nodo de tipo `RES_ESUN` en la raíz, una expresión en el primer hijo y un tipo en el segundo:

```
expresionEsUn   : #(RES_ESUN e:expresion t:tipo)
```

Una vez reconocida la estructura, debemos realizar nuestras operaciones con una acción. Comenzaremos haciendo el *casting* a `ExpressionAST` de la regla:

```
{
   ExpressionAST ast = (ExpressionAST)#expresionEsUn;
```

Lo primero que debemos hacer es comprobar que la expresión "e" tiene R-value (se puede "leer"). Si no es el caso, debemos lanzar una excepción:

```
ExpressionAST exp = (ExpressionAST)e;
if(!exp.getRValue())
{
   throw new ContextException(
      "Se necesita una expresión con R-value",
      ast );
}
```

Toda expresión `esUn` devuelve un booleano, independientemente de su contenido. Además, el booleano es de "solo lectura"; es dedir, que tiene R-value pero no L-value. Si traducimos todo esto literalmente tendremos:

```
ast.setExpType(LeLiTypeManager.tipoBooleano);
ast.setLValue(false);
ast.setRValue(true);
```

Lo que nos deja únicamente el E-value por determinar. El E-value de una expresión es el "valor" de una expresión. Por ejemplo, el E-value de `1 esUn Entero` es el booleano `cierto`.

El problema de calcular el E-Value de la expresión es que hay que tener en cuenta varias reglas. Así, sea `tipoExpresion` el tipo de la expresión de la izquierda del `esUn` y sea `tipoDer` el tipo a su derecha. Entonces:

- Si el `tipoExpresion` es el tipo error (ocurrió un error mientras se resolvía la expresión) entonces el valor de la expresión `esUn` es cierta (porque el tipo error es por defecto "compatible con todo").
- Si `tipoDer` es erróneo (el usuario ha escrito el nombre de una clase inexistente) entonces también es cierto.
- Si `tipoDer` es `Objeto`, entonces la expresión es siempre cierta.
- Si `tipoExpresion` es una subclase de `tipoDer`, entonces la expresión es siempre cierta.
- Si `tipoExpresion` no es una subclase de `tipoDer` y a su vez `tipoDer` no es una subclase de `tipoExpresion`, entonces la expresión `esUn` es **falsa**.
- En cualquier otro caso (`tipoDer` es una subclase de `tipoExpresion`) el resultado de la evaluación no puede conocerse en tiempo de compilación, y debe calcularse en tiempo de ejecución.

Traducimos todas estas reglas a código java, y obtenemos lo siguiente:

```
LeLiType tipoExpresion = (LeLiType)((ExpressionAST)e).getExpType();
LeLiType tipoDer = (LeLiType)((TypeAST)t).getExpType();
if( tipoExpresion.getTag()==TIPO_ERROR
||  ( tipoDer.getTag()==TIPO_ERROR ||
        tipoDer.getName().equals("Objeto") )
||  tipoExpresion.isA(tipoDer) )
{
   ast.setEValue(new Boolean(true));
}
else if ( !tipoExpresion.isA(tipoDer) &&
          !tipoDer.isA(tipoExpresion) )
{
   ast.setEValue(new Boolean(false));
} // en otro caso E-value se queda como esté (a null)
}
;
```

Para finalizar, utilizaremos un manejador de excepciones que pondrá valores "seguros" en la expresión además de imprimir el mensaje de error correspondiente. Para ello invocaremos el método `errorExpresion` que habíamos definido en la zona de código nativo del analizador:

```
exception catch [RecognitionException re]
{
   errorExpresion( re, (ExpressionAST)#expresionEsUn,
                   false, true, LeLiTypeManager.tipoBooleano, null );
}
```

En este caso no utilizamos los valores seguros por defecto del lenguaje, sino que utilizamos unos personalizados para la regla. De ahora en adelante prescindiremos de comentar los manejadores de excepciones de las reglas de las fases 3 y 4; todos son muy similares.

Expresiones unarias

Las expresiones unarias de LeLi son las que tienen un operador como raíz y un único operando como hijo. Esto incluye las operaciones de cambio de signo, post incremento, post decremento y negación lógica:

```
expresionUnaria
   :
   ( #(OP_MENOS_UNARIO expresion)
   | #(OP_MASMAS       expresion)
   | #(OP_MENOSMENOS   expresion)
   | #(OP_NO           expresion)
   )
   {
      // acción
   }
```

Nótese que hemos agrupado los cuatro patrones árbol en una sola sub regla (delimitada por los paréntesis señalados en el código) porque la acción será la misma para todos.

Comenzaremos con el L-value y R-value. Se da la casualidad de que todas las expresiones unarias devuelven valores de "solo lectura" – con R-value pero sin L-value. Así que solamente hay que darles los valores adecuados:

```
      // Hacemos el casting
      ExpressionAST raiz = (ExpressionAST) #expresionUnaria;

      raiz.setLValue(false);
      raiz.setRValue(true);
```

Ahora vienen las comprobaciones. Es necesario que el operando tenga R-value para que la expresión sea válida, así que en caso contrario se lanza una excepción:

```
ExpressionAST operandoAST = (ExpressionAST)(raiz.getFirstChild());
      if(!operandoAST.getRValue())
         throw new ContextException(
           "Se necesita una expresión con rvalue", operando );
```

En el caso de las operaciones de post incremento y post decremento es necesario, además, que el operando tenga L-value (sea "modificable"):

```
if( ( raiz.getType()==OP_MASMAS ||
      raiz.getType()==OP_MENOSMENOS) &&
      !operandoAST.getLValue() )
   throw new ContextException(
      "Se necesita una expresión con lvalue", operandoAST );
```

Hemos terminado con L-value y R-value, así que pasamos al tipo de la expresión. Lo primero que hemos de hacer es obtener el tipo del operando. Dado que solamente algunos tipos básicos de LeLi pueden intervenir en las expresiones unarias (`Entero`, `Real` y `Booleano`) y que los tipos básicos de LeLi pueden diferenciarse directamente con su etiqueta, vamos a utilizar un entero para los tipos, y los diferenciaremos con sus etiquetas:

```
int tipoOperando = operandoAST.getExpType().getTag();
Type t = null; // aquí guardaremos el tipo de la expresión
```

Lo primero que comprobaremos es que el tipo del operando no sea erróneo; de ser así, también lo será el de la expresión, y habremos acabado de calcular su tipo:

```
if(tipoOperando==TIPO_ERROR)
   t = LeLiTypeManager.tipoError;
```

Si el operando no es erróneo, entonces el tipo de la expresión puede ser calculado con las siguientes reglas:

- En el caso del cambio de signo, el post incremento y el post decremento, el tipo de la expresión coincide con el del operando, siempre y cuando éste sea numérico (Entero o Real). En otro caso se produce un error.
- En el caso de la negación, la expresión unaria es booleana si el operando es booleano, y errónea en otro caso.
- En cualquier otro caso la situación es indefinida, y por lo tanto errónea.

Las reglas anteriores se traducen al siguiente código (incluimos la comprobación de `tipoOperando==TIPO_ERROR` por claridad)

```
if(tipoOperando==TIPO_ERROR)
   t = LeLiTypeManager.tipoError;
else
{
   switch(raiz.getType())
   {
   case OP_MENOS_UNARIO: case OP_MASMAS:
   case OP_MENOSMENOS:

      if(isNumeric(tipoOperando))
         t = operandoAST.getExpType();
      else t = null;
   break;
   case OP_NO:
      if(tipoOperando==TIPO_BOOLEANO)
         t = LeLiTypeManager.tipoBooleano;
      else t = null;
   break;

   default :
      t = null;
   break;
   }
```

```
      }

      if(null==t)
      {
         throw new ContextException(
            "El tipo '"+ operandoAST.getExpType() +
            "' no es compatible con el operador unario'" +
            raiz.getText() + "'", operandoAST);
      }

      raiz.setExpType(t);
```

Obsérvese el método `isNumeric`. Su implementación no podría ser más simple:

```
   public boolean isNumeric(int id)
   { return (id==TIPO_ENTERO || id==TIPO_REAL); }
```

Hay que insertarlo en la zona de código nativo.

Ahora solamente nos queda calcular el E-value de la expresión. Las reglas para calcularlo son:

- Si el operando no tiene E-value, entonces la expresión unaria completa no tiene E-Value.
- En las expresiones de negación, si el E-value del operando es un booleano, entonces el E-value es la negación de dicho booleano.
- En el post incremento, el operando debe tener un E-value entero o flotante. Entonces el E-value del post incremento será el E-value del operando + 1.
- El post decremento es idéntico, pero restando en vez de sumando.
- El cambio de signo solamente puede aplicarse a tipos numéricos, y su E-value es el E-value del operando (ya sea real o entero) cambiado de signo.
- En todos los demás casos el E-value no puede calcularse, y será null.

Lo que en código viene a ser lo siguiente:

```
if(null==o)
        raiz.setEValue(null);
     else
     {
        switch(raiz.getType())
        {
        case OP_NO:
           if(o instanceof Boolean)
              raiz.setEValue(new Boolean(!((Boolean)o).booleanValue()));
        break;
        case OP_MENOS_UNARIO:
           if(o instanceof Integer)
              raiz.setEValue(new Integer(-((Integer)o).intValue()) );
           else if (o instanceof Float)
              raiz.setEValue(new Float(-((Float)o).floatValue()) );
        break;
        case OP_MASMAS:
           if(o instanceof Integer)
              raiz.setEValue(new Integer( ((Integer)o).intValue()+1 ) );
           else if (o instanceof Float)
              raiz.setEValue(new Float( ((Float)o).floatValue()+1) );
        break;
        case OP_MENOSMENOS:
```

```
            if(o instanceof Integer)
              raiz.setEValue(new Integer( ((Integer)o).intValue()-1 ) );
            else if (o instanceof Float)
              raiz.setEValue(new Float( ((Float)o).floatValue()-1) );

          default: raiz.setEValue(null); break;
          }
      }
```

Aquí se acaba la acción y la regla. Como siempre debe ir sucedida del manejador de excepciones correspondiente:

```
exception catch [RecognitionException re]
{
   errorExpresion( re, (ExpressionAST)#expresionUnaria,
                   false, true, LeLiTypeManager.tipoError, null );
}
```

Expresiones binarias

Las características de las expresiones binarias no son más difíciles de calcular que el resto. El único problema que presentan es el número de posibilidades que presentan: hay 13 operadores binarios en LeLi. Además, algunos de ellos, como la suma, son trementamente versátiles y se utilizan en un montón de situaciones parecidas, pero diferentes. Todos estos casos hay que tenerlos en cuenta, y ello produce un código bastante "aparatoso".

Los 13 operadores binarios son la suma, resta, producto, división, disyunción ('o'), conjunción ('y'), igualdad, desigualdad, mayor, mayor o igual, menor, menor o igual y asignación.

```
expresionBinaria
   :
   (  #(OP_MAS           expresion expresion)
   |  #(OP_MENOS         expresion expresion)
   |  #(OP_O             expresion expresion)
   |  #(OP_Y             expresion expresion)
   |  #(OP_IGUAL         expresion expresion)
   |  #(OP_DISTINTO      expresion expresion)
   |  #(OP_MAYOR         expresion expresion)
   |  #(OP_MENOR         expresion expresion)
   |  #(OP_MAYOR_IGUAL   expresion expresion)
   |  #(OP_MENOR_IGUAL   expresion expresion)
   |  #(OP_PRODUCTO      expresion expresion)
   |  #(OP_DIVISION      expresion expresion)
   |  #(OP_ASIG          expresion expresion)
   )
   {
      // acciones aquí
   }
   ;
```

Como veremos, las acciones a realizar no son pocas.

Comencemos por comprobar el L-value y R-value de los operandos que lo precisen.

Solamente la asignación precisa de L-value en uno de sus operandos (el de la izquierda). En este caso concreto resulta más cómodo hacer el test justo debajo de la regla, en lugar de dentro de la acción global. Así:

```
expresionBinaria
   :
   (  #(OP_MAS          expresion expresion)
   ...
   |  #(OP_ASIG         eizq:expresion expresion)
      {
         if( !((ExpressionAST)eizq).getLValue() )
            throw new ContextException(
               "La parte izquierda de la asignación no tiene l-value",
               #expresionBinaria );
      }
   )
```

Ya en la acción, las operaciones relacionadas con el L-value y el R-value son muy sencillas; el siguiente código debería ser comprensible para el lector:

```
ExpressionAST raiz = (ExpressionAST) #expresionBinaria;
// Calculamos r-value y l-value
raiz.setLValue(false); // L value siempre falso
raiz.setRValue(true);  // R siempre cierto
```

Lo siguiente que vamos a hacer será comprobar el R-value de los operadores, que debe ser cierto para los dos en todos los casos. Para ello deberemos obtener los ASTs de dichos operadores. Nótese que se utilizan los métodos de navegación de los ASTs (`getFirstChild` y `getNextSibling`) para obtenerlos, en lugar de utilizar etiquetas. En este caso era más eficiente y sencillo (ANTLR no permite utilizar la misma etiqueta varias veces en la misma regla. O bueno, sí lo permite, pero el código generado no compila).

```
// Obtenemos los dos operandos
ExpressionAST ei = (ExpressionAST)(raiz.getFirstChild());
ExpressionAST ed = (ExpressionAST)(ei.getNextSibling());

if(!ei.getRValue())
   throw new ContextException(
      "Se necesita una expresión con r-value", ei );

if(!ed.getRValue())
   throw new ContextException(
      "Se necesita una expresión con r-value", ed );
```

Ahora hay que calcular el tipo de la expresión. Las reglas son las siguientes:

- Si el tipo de cualquiera de los dos operandose es erróneo, entonces el tipo de la expresión es erróneo.
- La suma permite sumar `Entero/Entero` (resultado: `Entero`), `Entero/Real` (`Real`), `Real/Entero`(`Real`), `Real/Real`(`Real`), y `Cadena/`cualquier tipo básico (resultado: `Cadena`).
- La resta y el producto y la divisón: `Entero/Entero` (resultado: `Entero`), `Entero/Real` (`Real`), `Real/Entero`(`Real`), `Real/Real`(`Real`)
- La asignación siempre tiene un tipo vacío.
- La conjunción y la disyunción son de tipo booleano, siempre y cuando se apliquen sobre operandos booleanos.
- La comparación y la igualdad funcionan entre tipos iguales y tipos numéricos, y son expresiones booleanas.
- Mayor, menor, mayor o igual y menor o igual actúan solamente sobre tipos numéricos (no se

pueden comparar cadenas) y en caso de que los dos operandos sean numéricos la expresión es de tipo booleano.

- En cualquier otro caso el tipo es erróneo, y hay que lanzar un error.

El código necesario para implementar este comportamiento es similar al que veíamos en las expresiones unarias:

```
if(ti==TIPO_ERROR || td==TIPO_ERROR)
      t = LeLiTypeManager.tipoError;
   else
   {
      switch(raiz.getType())
      {
      case OP_MAS:
         if(ti==TIPO_CADENA || td==TIPO_CADENA)
            t = LeLiTypeManager.tipoCadena;
         else if(ti==TIPO_ENTERO && td==TIPO_ENTERO)
            t = LeLiTypeManager.tipoEntero;
         else if(isNumeric(ti) && isNumeric(td))
            t = LeLiTypeManager.tipoReal;
      break;
      case OP_MENOS: case OP_PRODUCTO: case OP_DIVISION:
         if(ti==TIPO_ENTERO && td==TIPO_ENTERO)
            t = LeLiTypeManager.tipoEntero;
         else if(isNumeric(ti) && isNumeric(td))
            t = LeLiTypeManager.tipoReal;
      break;
      case OP_ASIG:
         t = LeLiTypeManager.tipoVacio;
      break;
      case OP_Y: case OP_O:
         if(ti==TIPO_BOOLEANO && td==TIPO_BOOLEANO)
            t = LeLiTypeManager.tipoBooleano;
      break;
      case OP_IGUAL: case OP_DISTINTO:
         if( ti==TIPO_BOOLEANO && td==TIPO_BOOLEANO ||
             ti==TIPO_CADENA   && td==TIPO_CADENA ||
             isNumeric(ti) && isNumeric(td) )
            t = LeLiTypeManager.tipoBooleano;
      break;
      case OP_MAYOR: case OP_MENOR:
      case OP_MAYOR_IGUAL: case OP_MENOR_IGUAL:
         if( isNumeric(ti) && isNumeric(td) ||
             TIPO_CADENA==ti && TIPO_CADENA==td )
         t = LeLiTypeManager.tipoBooleano;
      break;
      }
   }

   if(null==t)
   {
      throw new ContextException( "El tipo '"+
         ei.getExpType() + "' no es compatible con '" +
         ed.getExpType() + "' con respecto al operador '"+
         raiz.getText() + "'",
         raiz);
   }
```

```
    raiz.setExpType(t);
```

Por último queda calcular el E-value de las expresiones binarias. Ésta ha sido una de las tareas más laboriosas a las que he tenido que enfrentarme; se trata de un gigantesco bucle switch con literalmente 42 comprobaciones "if". No es difícil de comprender, es simplemente muy laborioso. Me abstendré de explicar una a una todas las reglas, y también de presentar aquí el código. El lector que lo desee podrá encontrarlo al final del apartado.

Una vez calculado el E-value de la expresión solamente quedará capturar las excepciones. En este caso vamos a utilizar los valores seguros por defecto:

```
  }
  ; // fin de la regla
exception catch [RecognitionException re]
{ errorExpresion(re, (ExpressionAST)#expresionBinaria); }
```

7.6.6: Fase 4: Accesos

Los accesos son un tipo especial de expresión, que no implica la utilización de ningún operador. Son las partículas básicas a partir de las cuales se construyen las expresiones (todas las hojas del AST de una expresión son accesos). Al formar parte de las expresiones comparten con ellas sus características (tipo, L-value, R-value, E-value). En este apartado calcularemos esas otras características.

Un acceso está formado por un primer elemento, llamado la "raíz del acceso", seguida por cero o más "sub-accesos". Tanto la raíz como los sub subaccesos tienen características de expresiones. Cuando no hay subaccesos, las características del acceso son las de su raíz. En otro caso, las características del acceso son las del último subacceso de la lista.

Sintácticamente, la raíz aparece al principio, y cada sub acceso se anida utilizando un punto:

```
raiz.subAcceso1.subAcceso2.subAcceso3
```

El AST que representa un acceso utiliza el token imaginario `ACCESO` como raíz. Su primer hijo es el AST de la raíz del acceso, y si hay sub accesos éstos son hijos adicionales. Una ilustración puede ayudar a aclarar este galimatías:

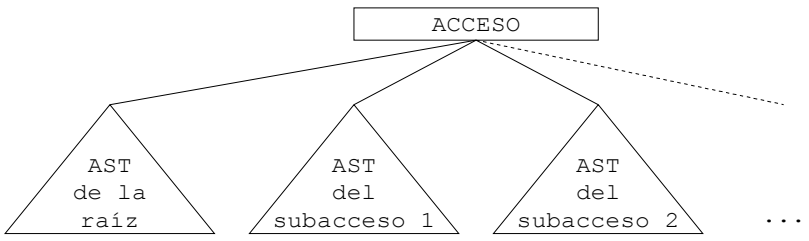

Ilustración 7.10 AST de un acceso

Cada sub acceso (excepto probablemente el último) tiene un ámbito asociado en el cual debe encontrarse el siguiente acceso de la lista. Por ejemplo, supongamos la clase `Persona` que tiene un atributo `Nombre` de tipo cadena que a su vez tiene un método `Entero` longitud. El siguiente acceso será válido entonces:

```
Persona cid("Rodrigo", "Díaz de Vivar");
Sistema.imprime(cid.Nombre.longitud(); // Imprime "7"
```

De lo anterior se deduce que *cada punto de la lista de subaccesos implica un cambio de ámbito*. Para poder trabajar convenientemente con cada subacceso necesitamos por lo tanto conocer al menos el ámbito del subacceso anterior en la lista de sub accesos. En otras palabras, cada sub acceso "depende" del acceso anterior en la lista – y deberemos pasárselo como parámetro a la regla que lo reconozca[70].

En nuestro iterador básico de ASTs reconocíamos un acceso conjugando las reglas `raizAcceso` y `subAcceso`. En la situación actual, dado que debemos pasarle un parámetro a subacceso, utilizaremos una regla propia llamada `subAccesoContexto`. Mantendremos una referencia al AST más actual, y sus características serán copiadas al acceso al finalizar el reconocimiento.

Con todo esto en mente, la regla acceso quedará como sigue:

```
acceso
{ ExpressionAST arbolActual = null; }
   : #( ACCESO r:raizAcceso
        { arbolActual = (ExpressionAST)r; }
        ( s:subAccesoContexto[arbolActual]
          { arbolActual = (ExpressionAST)s; }
        )*
      )
   {
      // Copiar características del último subacceso
      ExpressionAST raiz = (ExpressionAST)#acceso;
      raiz.setRValue(arbolActual.getRValue());
      raiz.setLValue(arbolActual.getLValue());
      raiz.setEValue(arbolActual.getEValue());
      raiz.setExpType(arbolActual.getExpType());
   }
   ;
exception catch [RecognitionException re]
{ errorExpresion(re, (ExpressionAST)#acceso); }
```

Ahora llegó el momento de reconocer las raíces de un acceso.

Raíz de un acceso

Las raíces de un acceso se reconocen con la regla `raizAcceso`. Ésta comienza con la declaración de una variable auxiliar que nos ahorrará tener que realizar tediosos *castings* en las acciones de la regla:

```
raizAcceso
{ ExpressionAST ast = (ExpressionAST)#raizAcceso; }
   :
```

Después hay una serie de alternativas. La raíz de un acceso puede ser:

- Un identificador
- La palabra reservada `super`
- La palabra reservada `atributo`
- La palabra reservada `parámetro`
- Un literal

70 Podríamos pasar solamente el ámbito, pero es más cómodo pasar directamente el AST.

- Una invocación de un método de la clase actual (una llamada)
- Una conversión de tipos
- Cualquier expresión entre paréntesis

Comencemos por la primera: un identificador. Los identificadores pueden leerse y escribirse por defecto, así que tienen R-value y L-value.

```
i:IDENT
{
   ast.setRValue(true);
   ast.setLValue(true);
```

Ya tenemos R-value y L-value. Lo siguiente que hay que hacer es calcular el tipo.

Para ello debemos empezar por obtener una declaración llamada "i":

```
   Declaration d = contexto.obtenerDeclaracion(i);
```

Una vez llegados a éste punto, pueden darse tres situaciones:

- Que `d` sea una declaración de una variable o un parámetro (su etiqueta es `INST_DEC_VAR` o `RES_PARAMETRO`). En tal caso obtener el tipo es trivial (`d.getType()`).
- Que sea una declaración del atributo de una clase (la etiqueta es `RES_ATRIBUTO`). Entonces será un poco más complicado, porque recordemos que las declaraciones de atributos tienen un tipo "especial"(`antlraux.context.types.AttributeType`) al que hay que "extraerle" el tipo "real".
- Por último, puede que el identificador sea el nombre de una clase (puede ser, por ejemplo, "Objeto", "Sistema"). Los accesos con raíces de este tipo son accesos a miembros abstractos de las clases (ej. `Sistema.imprime("Hola")`). La etiqueta de la declaración en tal caso es o bien `RES_CLASE` o bien una de las etiquetas de los tipos básicos (`TIPO_ENTERO`, `TIPO_REAL`, `TIPO_BOOLEANO`, `TIPO_CADENA`). Para obtener el tipo de la raíz será necesario utilizar las metaclases.
- Si no se da ninguno de los casos anteriores, el tipo es erróneo.

En código:

```
   switch(d.getTag())
   {
      case INST_DEC_VAR: case RES_PARAMETRO:
         ast.setExpType(d.getType());
      break;
      case RES_ATRIBUTO: // acceso a un atributo
         // Obtener el "tipo atributo" (AttributeType)
         // Recordar que AttributeType es similar a MethodType
         AttributeType tipoAtributo = (AttributeType)(d.getType());
         // Y de ahí el verdadero tipo (Entero, Cadena, etc)
         ast.setExpType(tipoAtributo.getType());
      break;
      case RES_CLASE: // acceso a un miembro abstracto
      case TIPO_ENTERO: case TIPO_REAL:
      case TIPO_BOOLEANO: case TIPO_CADENA:
         LeLiType clase = (LeLiType)(d.getType());
         LeLiMetaType metaClase = clase.getMetaType();
         ast.setExpType(metaClase);
      break;
      default: // incluye TIPO_ERROR
```

```
            ast.setExpType(LeLiTypeManager.tipoError);
         break;
      }
```

Por último queda calcular el E-value. Pues bien, no hace falta. En LeLi no hay constantes, de manera que ningún atributo, ya sea abstracto o no abstracto, puede tener un valor precalculado en tiempo de compilación.

Sin embargo, si hubiera alguna manera de especificar constantes en LeLi, (por ejemplo, si los atributos abstractos fueran constantes) entonces se podría obtener el E-value de los identificadores a partir de la expresión de inicialización de la declaración[71].

La clase `Declaration` incluye un campo de tipo AST llamado `initialValue`. Sirve para guardar expresiones de inicialización en declaraciones, por ejemplo ésta:

```
int a = 2+3;
```

Si la expresión de inicialización existe y tiene E-value, entonces éste será el E-value del acceso:

```
      // Cálculo del E-Value
      ExpressionAST iv = (ExpressionAST)d.getInitialValue();
      if(iv!=null)
         ast.setEValue(iv.getEValue());
```

Estas líneas aparecerán comentadas en el iterador.

Bueno, y con eso se terminan los identificadores. Afortunadamente el resto de las raíces de acceso son mucho más sencillas.

Tomemos, por ejemplo, las raíces con palabras reservadas, es decir, `parámetro`, `atributo` y `super`. Se pueden reconocer con la siguiente alternativa:

```
   | ( RES_PARAMETRO { ast.setExpType(tipoParametroAux); }
     | RES_ATRIBUTO  { ast.setExpType(tipoAtributoAux);  }
     | RES_SUPER     { ast.setExpType(claseActual.getSuperType()); }
     )
     {
        ast.setLValue(false);
        ast.setRValue(true);
        ast.setEValue(null);
     }
```

Las tres palabras reservadas tienen idénticas características como expresión, variando el tipo. Son de “solo lectura”, así que tienen R-value pero no L-value. En ningún caso pueden tener E-value.

En cuanto al tipo, el más sencillo es el de `super`; se obtiene la superclase de la clase actual. Los tipos de `parámetro` y `atributo` son los tipos auxiliares que definimos como atributos estáticos del analizador en la zona de código nativo.

El resto de las alternativas de raizAcceso tienen sus propias reglas, así que simplemente hay que invocarlas:

```
   | literal
   | llamadaContexto [claseActual]
   | conversion
   | expresion
   ;
exception catch [RecognitionException re]
```

71 Aunque seguramente habría que modificar un poco la forma de calcular el L-value.

```
{ errorExpresion(re, ast); }
```

Literales

Los literales son muy fáciles de tratar. Para empezar, todos ellos son de "solo lectura":

```
literal
{
   ExpressionAST ast = (ExpressionAST)#literal;
   // Antes de hacer nada, ya hemos colocado L-value y R-value
   ast.setRValue(true);
   ast.setLValue(false);
}
```

Calcular el tipo y E-value de los diferentes literales es muy sencillo:

```
   : LIT_ENTERO
     {
        ast.setEValue(new Integer(ast.getText()));
        ast.setExpType(LeLiTypeManager.tipoEntero);

     }
   | LIT_REAL
     {
        ast.setEValue(new Float(ast.getText()));
        ast.setExpType(LeLiTypeManager.tipoReal);
     }
   | LIT_CADENA
     {
        ast.setEValue(ast.getText());
        ast.setExpType(LeLiTypeManager.tipoCadena);
     }
   | LIT_NL
     {
        ast.setEValue("\n");
        ast.setExpType(LeLiTypeManager.tipoCadena);
     }
   | LIT_TAB
     {
        ast.setEValue("\t");
        ast.setExpType(LeLiTypeManager.tipoCadena);
     }
   | LIT_COM
     {
        ast.setEValue("\"");
        ast.setExpType(LeLiTypeManager.tipoCadena);
     }
   | LIT_CIERTO
     {
        ast.setEValue(new Boolean(true));
        ast.setExpType(LeLiTypeManager.tipoBooleano);
     }
   | LIT_FALSO
     {
        ast.setEValue(new Boolean(false));
        ast.setExpType(LeLiTypeManager.tipoBooleano);
     }
   ;
```

```
exception catch [RecognitionException re]
{ errorExpresion(re, (ExpressionAST)#ast); }
```

Invocaciones

Las invocaciones (o "llamadas") en este analizador son reconocidas con la regla `llamadaContexto` en lugar de llamada. Como las llamadas pueden ser un subacceso, no queda más remedio que "pasarles" un parámetro con la "clase invocadora".

La regla comienza con la declaración de algunas variables locales a la regla.

```
llamadaContexto [ LeLiType llamador ]
{
   ExpressionAST ast = (ExpressionAST)#llamadaContexto;
   Declaration d = null;
   LeLiMethodType lmt = null;
```

Después se realiza una comprobación de seguridad: las palabras reservadas `parámetro` y `atributo` no son invocadores válidos (no se puede escribir `parametro.m()`).

```
   // Comprobar si se está usando la palabra reservada
   // "parámetro" o "atributo" seguidas de una llamada a
   // un método o constructor
   if( llamador.getTag() == RES_PARAMETRO ||
       llamador.getTag() == RES_ATRIBUTO )
   {
      throw new ContextException(
        "Las palabras reservadas 'parámetro' y 'atributo' no pueden "+
        "ir seguidas de una llamada a un método o un constructor",
        ast );
   }
```

La regla `llamadaContexto` admite tanto llamadas a métodos como a constructores. Al reconocerlos, inicializamos las variables locales `d` y `lmt`, definidas al principio de la presente regla:

```
   ( #( LLAMADA nombre:IDENT
        { lmt = new LeLiLlamadaType(nombre.getText(), llamador); }
        listaExpresionesContexto [lmt]
        { d = llamador.buscarMetodoCompatible(lmt); }
      )
   | #( RES_CONSTRUCTOR
          { lmt = new LeLiLlamadaConstructorType(llamador); }
          listaExpresionesContexto[lmt]
          { d = llamador.buscarConstructorCompatible(lmt); }
      )
   )
   {
      // acción
   }
```

Nótese que las dos alternativas están entre parántesis, formando una única sub regla. De esta forma, podemos añadir una acción común a las dos alternativas, que trabaje con `d` y `lmt`.

Hay que obtener las características del subacceso, dependiendo del valor de `d` y `lmt`. Así,

- Si el llamador es erróneo, entonces se asumen valores seguros (tipo error, L y R value, E-value null).

- Si `d` es null, entonces no se ha encontrado un método o constructor compatible. Lanzar un error.
- En otro caso se trata de una invocación normal y corriente. Las invocaciones no tienen L-value ni E-value. Su tipo es el tipo que devuelvan los métodos que invocan, y su R-value depende de si éste es vacío o no.

O dicho de otra forma:

```
      if( llamador.getTag()==TIPO_ERROR )
      {
         ast.setExpType(LeLiTypeManager.tipoError);
         ast.setRValue(true);
         ast.setLValue(true);
      }
      else if( null==d )
      {
         throw new ContextException(
            "La clase '" + llamador.getName() +
            "' no contiene un método o constructor compatible con " +
            lmt.toString(), ast );
      }
      else
      {
         ast.setLValue(false);
         ast.setEValue(null);

         LeLiMethodType decMetodo= (LeLiMethodType)d.getType();

         ast.setExpType(decMetodo.getReturnType());
         if( ast.getExpType().getTag()==TIPO_VACIO )
            ast.setRValue(false);
         else
            ast.setRValue(true);
      }
     }
     ;
exception catch [RecognitionException re]
{ errorExpresion(re, ast); }
```

Una nota adicional: el código de la regla `listaExpresionesContexto` es el siguiente:

```
listaExpresionesContexto [LeLiMethodType lmt]
   : #( LISTA_EXPRESIONES
        ( e:expresion
          { lmt.addParam(((ExpressionAST)e).getExpType()); }
        )* )
   ;
exception catch [RecognitionException re] { reportError(re); }
```

Conversiones de tipo

Las conversiones se realizan mediante la palabra reservada `convertir`. Se trata de convertir una expresión a un tipo dado. Las conversiones se reconocen utilizando la regla `conversion`. La regla comienza con la declaración de una variable para facilitar los *castings* y la inicialización del L-value y el R-value.

```
conversion
{
   ExpressionAST ast = (ExpressionAST)#conversion;
   ast.setLValue(false);
   ast.setRValue(true);
}
```

El AST reconocido por esta regla tiene `RES_CONVERTIR` como raíz, y dos hijos: una expresión y un tipo.

```
   : #(RES_CONVERTIR e:expresion t:tipo)
```

Hay que calcular el tipo y E-value de la expresión, cuando sea posible. Vamos a ver:

- Cuando `t` es el tipo "Objeto", el resultado de la conversión siempre existirá y será de ese mismo tipo.
- Cuando `t` sea erróneo, el tipo de la expresión será erróneo.
- Si la expresión es de un tipo que es subclase de `t`, entonces la conversión siempre se podrá realizar, siendo `t` el tipo de la conversión.
- Deben contemplarse las conversiones entre tipos. Existen conversiones entre `Cadena` y los demás tipos básicos de LeLi, y entre `Entero` y `Real`. El resto de las conversiones no son posibles o no se pueden realizar en tiempo de compilación.
- Si el tipo de la expresión no es una subclase de `t` ni viceversa, entonces no hay conversión posible. Lanzar un error.
- En otro caso, hay que decidir en tiempo de ejecución.

El código que implementa estas reglas es el siguiente:

```
   {
      ExpressionAST ee = (ExpressionAST)e;
      LeLiType te = (LeLiType)ee.getExpType();
      LeLiType tt = (LeLiType)((TypeAST)t).getExpType();

      if( tt.getTag()==TIPO_ERROR ||
          tt.getName().equals("Objeto") )
      {
         ast.setExpType(tt);
      }
      else if( te.isA(tt) )
      {
         ast.setExpType(tt);
      }
      // Convertir tipos básicos y el resto
      switch( tt.getTag() )
      {
      case TIPO_CADENA: // Convertir a cadena
         switch(te.getTag())
         {
         case TIPO_ENTERO: case TIPO_REAL:
            ast.setExpType(tt);
            if(null==ee.getEValue())
               ast.setEValue(null);
            else
               ast.setEValue(ee.getEValue().toString());
         break;
```

```
        case TIPO_BOOLEANO:
           ast.setExpType(tt);
           if(null==ee.getEValue())
              ast.setEValue(null);
           else
           {
              boolean b = ((Boolean)ee.getEValue()).booleanValue();
              ast.setEValue(b?"cierto":"falso");
           }
        // Comprobar el resto en tiempo de ejecución
        // (ej. conversión Objeto->Cadena)
        }
      break;
      case TIPO_REAL: // Convertir a Real
        switch(te.getTag())
        {
           case TIPO_ENTERO:
              ast.setExpType(tt);
              if(null==ee.getEValue())
                 ast.setEValue(null);
              else
              {
                 int i = ((Integer)ee.getEValue()).intValue();
                 ast.setEValue(new Float(i));
              }
           break;
           // Comprobar el resto en tiempo de ejecución
           // (ej. conversión Objeto->Real)
        }
      break;
      default:
        if( !te.isA(tt) && !tt.isA(te))
           throw new ContextException(
              "La expresión no es convertible al tipo '"+tt+"'",
              e );
        ast.setExpType(tt);
        ast.setEValue(null);
        // Comprobar el resto en tiempo de ejecución
        // (ej. conversión Objeto->Tipo def. usuario)
      }
   }
   ;
exception catch [RecognitionException re]
{ errorExpresion(re, (ExpressionAST)ast); }
```

Los subaccesos

La última regla que nos queda por analizar es `subAccesoContexto`. La regla comienza fuerte, inicializando casi todos las características de expresión al inicio, amén de declarando algunas variables locales que serán muy útiles.

```
subAccesoContexto [ExpressionAST anteriorAST]
{
   ExpressionAST ast = (ExpressionAST)#subAccesoContexto;
   ast.setExpType(LeLiTypeManager.tipoError);
   ast.setRValue(true);
   ast.setLValue(true);
```

```
   ast.setEValue(null);
   LeLiType tipoAnterior = (LeLiType)(anteriorAST.getExpType());
}
```

Como puede verse, en LeLi en principio todo sub acceso tiene R-value y L-value, y carece de E-value[72]. Así que lo único que hay que calcular es el tipo.

Los subaccesos en LeLi solamente pueden ser de tres tipos: identificadores, llamadas y la palabra reservada `super`.

El trabajo de reconocer las llamadas ya lo tenemos hecho: basta con utilizar `llamadaContexto`:

```
   : llamadaContexto [tipoAnterior]
```

El subacceso con super también es muy fácil de implementar. Basta con obtener la superclase del subacceso anterior:

```
   | RES_SUPER
      {
         ast.setExpType(contexto.obtenerSuperTipo(tipoAnterior, ast));
      }
```

Por último quedan los identificadores. Como veremos no será tan fácil.

El modo de actuar dependerá del tipo anterior.

- Si el tipo anterior es el tipo auxiliar `tipoParametroAux` (su etiqueta es `TIPO_PARAMETRO`) entonces hay que hacer una búsqueda específica de parámetros en el ámbito actual.
- Algo similar ocurre con el tipo auxiliar `tipoAtributoAux` (su etiqueta es `RES_ATRIBUTO`). Solamente hay que tener en cuenta que las declaraciones de los atributos tienen tipos "especiales", a los que hay que "extraerles" el tipo real.
- Si no se da ninguno de los dos casos anteriores, estamos ante un subacceso normal y corriente a un atributo de una clase. Hay que buscar el atributo en el ámbito de la clase, y "extraer" el tipo como en el caso anterior.

Y por fin el código:

```
   | nombre:IDENT
      {
       Declaration d = null;
       switch(tipoAnterior.getTag())
       {
          case RES_PARAMETRO:
             d = contexto.obtenerObjeto(RES_PARAMETRO, nombre);
             ast.setExpType(d.getType());
          break;

          case RES_ATRIBUTO:
          {
             d = contexto.obtenerObjeto(RES_ATRIBUTO, nombre);
             AttributeType tipoAtributo = (AttributeType)d.getType();
             ast.setExpType(tipoAtributo.getType());
          }
          break;

          default:
          {
```

72 Como ya hemos mencionado, esto podría ser diferente de poder declararse "constantes" en LeLi.

```
              d = contexto.obtenerAtributo( tipoAnterior, nombre );
              AttributeType tipoAtributo = (AttributeType)d.getType();
              ast.setExpType(tipoAtributo.getType());
           }
        break;
        }
        ExpressionAST valorAST = (ExpressionAST)d.getInitialValue();
        if(valorAST!=null)
        ast.setEValue(valorAST.getEValue());
      }
    ;
exception catch [RecognitionException re]
{ errorExpresion(re, (ExpressionAST)#subAccesoContexto); }
```

¡Y ésa era la última regla!

7.6.7: Fichero LeLiTypeCheckTreeParser.g

A continuación se lista el fichero completo:

```
header
{

package leli;

/*-------------------------------------------------*\
 | Un intérprete para un Lenguaje Limitado(LeLi)  |
 | _____________________________________________  |
 |                                                |
 |     ANALISIS SEMÁNTICO -- chequeo de tipos     |
 | _____________________________________________  |
 |                                                |
 |             Enrique J. Garcia Cota             |
\*-------------------------------------------------*/

import antlraux.context.Context;
import antlraux.context.Scope;
import antlraux.context.Declaration;
import antlraux.context.ContextException;
import antlraux.context.asts.*;
import antlraux.context.types.Type;
import antlraux.context.types.AttributeType;
import antlraux.util.LexInfo;
import antlraux.util.LexInfoAST;
import antlraux.util.Logger;
import leli.types.*;

}

class LeLiTypeCheckTreeParser extends LeLiTreeParser;
options
{
   importVocab=LeLiSymbolTreeParserVocab;
   exportVocab=LeLiTypeCheckTreeParserVocab;
   buildAST = false;
   ASTLabelType = antlraux.util.LexInfoAST;
}
```

```
{
   public Logger logger=null;
   private LeLiContext contexto;
   private LeLiType claseActual = null;

   // se usa en los accesos, cuando aparece la palabra reservada
   // "parámetro" (ej parámetro.a)
   public static final LeLiType tipoParametroAux =
      new LeLiType(RES_PARAMETRO,"parámetro", null, null, null);

   // Idem para atributo (ej atributo.p1)
   public static final LeLiType tipoAtributoAux =
      new LeLiType(RES_ATRIBUTO,"atributo", null, null, null);

   LeLiTypeCheckTreeParser( Logger logger,
                            LeLiContext contexto )
   {
      this();
      this.logger = logger;
      this.contexto = contexto;
   }

   public void reportError( String msg,
                            String filename,
                            int line,
                            int column )
   {
      if(null==logger)
      {
         logger = new Logger("error", System.err);
      }
      logger.log( msg, 1, filename, line, column);
   }

   public void reportError(RecognitionException e)
   {
      reportError( e.getMessage(), e.getFilename(),
                   e.getLine(), e.getColumn() );
   }

   public void reportError( String msg,
                            LexInfo info )
   {
      reportError( msg, info.getFilename(),
                   info.getLine(), info.getColumn()) ;
   }

// ------ Métodos de la fase 1

   public void cambiaAmbitoActual(LexInfoAST ast)
   {
      Scope nuevoAmbito = ((ScopeAST)ast).getScope();
      contexto.setCurrentScope(nuevoAmbito);
   }

   public void cierraAmbitoActual()
   { contexto.toFather(); }
```

```
// ------ Métodos de la fase 3

   public boolean isNumeric(int id)
   { return (id==TIPO_ENTERO || id==TIPO_REAL); }


   public void errorExpresion( RecognitionException re,
                               ExpressionAST ast )
   {
      errorExpresion( re, ast, true, true,
                      LeLiTypeManager.tipoError,null);
   }

   public void errorExpresion( RecognitionException re,
                               ExpressionAST ast,
                               boolean LValue,
                               boolean RValue,
                               LeLiType tipo,
                               Object EValue )
   {
      reportError(re);
      ast.setExpType(tipo);
      ast.setRValue(RValue);
      ast.setLValue(LValue);
      ast.setEValue(EValue);
   }
}


//-------------- FASE 1: Controlar el ámbito actual -------------

programa
   : {  cambiaAmbitoActual( #programa ); }
     #( PROGRAMA (decClase)+ )
   ;
exception catch [RecognitionException ce] { reportError(ce); }

decClase
   : { cambiaAmbitoActual( #decClase ); }
     #( RES_CLASE nombre:IDENT
        { claseActual = contexto.obtenerTipo(nombre); }
        clausulaExtiende
        listaMiembros )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

decMetodo
   : { cambiaAmbitoActual( #decMetodo ); }
     #( RES_METODO ( RES_ABSTRACTO )? tipo
        IDENT listaDecParams listaInstrucciones
      )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

decConstructor
   : { cambiaAmbitoActual( #decConstructor ); }
```

```
     #( RES_CONSTRUCTOR listaDecParams listaInstrucciones )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

instMientras
   : { cambiaAmbitoActual( #instMientras ); }
    #( RES_MIENTRAS expresion listaInstrucciones )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

instHacerMientras
   : { cambiaAmbitoActual( #instHacerMientras ); }
     #( RES_HACER listaInstrucciones expresion )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

instDesde
   : { cambiaAmbitoActual( #instDesde ); }
     #( RES_DESDE
        listaExpresiones listaExpresiones listaExpresiones
        listaInstrucciones )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

instSi
   : { cambiaAmbitoActual( #instSi ); }
     #( RES_SI expresion listaInstrucciones
        { cierraAmbitoActual(); }
        (alternativaSi)* )
   ;
exception catch [RecognitionException ce] { reportError(ce); }

alternativaSi
   : { cambiaAmbitoActual ( #alternativaSi ); }
     ( #(BARRA_VERT expresion listaInstrucciones)
     | #(RES_OTRAS listaInstrucciones)
     )
     { cierraAmbitoActual(); }
   ;
exception catch [RecognitionException ce] { reportError(ce); }

//------ FASE 2: Adición de las variables locales a los ámbitos --------
//
// Esta fase consiste simplemente en añadir las declaraciones de las
// variables locales a los ámbitos. Se hace aquí en lugar de en el
// primer paso porque no deseamos que las variables declaradas
// "más tarde" conozcan las que están declaradas "antes". Además,
// preparamos las expresiones de los objetos declarados.
//

instDecVar
   : #(INST_DEC_VAR t:tipo nombre:IDENT (valor:expresion|le:listaExpresiones)?)
     {
        LeLiType type = (LeLiType) ((TypeAST)t).getExpType();
```

```
        contexto.insertarDecVariable( #instDecVar, nombre, type, valor, le);
      }
    ;
exception catch [RecognitionException ce] { reportError(ce); }


//---------------  FASE 3: Expresiones ----------------
//
// Toda expresión de LeLi es de algún tipo. La mayoría de las
// expresiones tienen un tipo básico (Cadena, Booleano...).
//
// En esta fase calcularemos los tipos de cada expresión, así
// como otros valores importantes de las expresiones como son
// el L-value y el R-value.
//

expresion : expresionBinaria
          | expresionUnaria
          | expresionEsUn
          | acceso
          ;

expresionEsUn
   : #(RES_ESUN e:expresion t:tipo)
   {
      ExpressionAST ast = (ExpressionAST)#expresionEsUn;
      ExpressionAST exp = (ExpressionAST)e;
      if(!exp.getRValue())
      {
         throw new ContextException(
            "Se necesita una expresión con R-value",
            ast );
      }

      ast.setExpType(LeLiTypeManager.tipoBooleano);
      ast.setLValue(false);
      ast.setRValue(true);

      LeLiType tipoExpresion = (LeLiType)exp.getExpType();
      LeLiType tipoDer = (LeLiType)((TypeAST)t).getExpType();
      if( tipoExpresion.getTag()==TIPO_ERROR
      ||  ( tipoDer.getTag()==TIPO_ERROR ||
               tipoDer.getName().equals("Objeto") )
      ||  tipoExpresion.isA(tipoDer) )
      {
         ast.setEValue(new Boolean(true));
      }
      else if ( !tipoExpresion.isA(tipoDer) &&
                !tipoDer.isA(tipoExpresion) )
      {
         ast.setEValue(new Boolean(false));
      }
   }
   ;
exception catch [RecognitionException re]
{
   errorExpresion( re, (ExpressionAST)#expresionEsUn,
                   false, true, LeLiTypeManager.tipoBooleano, null );
```

```
}

expresionUnaria
   :
   ( #(OP_MENOS_UNARIO expresion)
   | #(OP_MASMAS       expresion)
   | #(OP_MENOSMENOS   expresion)
   | #(OP_NO           expresion)
   )
   {
      ExpressionAST raiz = (ExpressionAST) #expresionUnaria;
      // Calculamos r-value y l-value
      raiz.setLValue(false);
      raiz.setRValue(true);

      ExpressionAST operandoAST = (ExpressionAST)(raiz.getFirstChild());
      if(!operandoAST.getRValue())
         throw new ContextException(
            "Se necesita una expresión con rvalue", operandoAST );
      if( ( raiz.getType()==OP_MASMAS ||
            raiz.getType()==OP_MENOSMENOS) &&
          !operandoAST.getLValue() )
          throw new ContextException(
           "Se necesita una expresión con lvalue", operandoAST );

      // Calculamos tipo de la expresión
      int tipoOperando = operandoAST.getExpType().getTag();
      Type t = null;

      if(tipoOperando==TIPO_ERROR)
         t = LeLiTypeManager.tipoError;
      else
      {
         switch(raiz.getType())
         {
         case OP_MENOS_UNARIO: case OP_MASMAS:
         case OP_MENOSMENOS:

            if(isNumeric(tipoOperando))
               t = operandoAST.getExpType();
            else t = null;
         break;
         case OP_NO:
            if(tipoOperando==TIPO_BOOLEANO)
               t = LeLiTypeManager.tipoBooleano;
            else t = null;
         break;

         default :
            t = null;
         break;
         }
      }

      if(null==t)
      {
         throw new ContextException(
            "El tipo '"+ operandoAST.getExpType() +
```

```
            "' no es compatible con el operador unario'" +
            raiz.getText() + "'", operandoAST);
      }

      raiz.setExpType(t);

      // Calculamos el valor de la expresión unaria, si se puede
      Object o = operandoAST.getEValue();
      if(null==o)
         raiz.setEValue(null);
      else
      {
         switch(raiz.getType())
         {
         case OP_NO:
            if(o instanceof Boolean)
               raiz.setEValue(new Boolean(!((Boolean)o).booleanValue()));
         break;
         case OP_MENOS_UNARIO:
            if(o instanceof Integer)
               raiz.setEValue(new Integer(-((Integer)o).intValue()) );
            else if (o instanceof Float)
               raiz.setEValue(new Float(-((Float)o).floatValue()) );
         break;
         case OP_MASMAS:
            if(o instanceof Integer)
               raiz.setEValue(new Integer( ((Integer)o).intValue()+1 ) );
            else if (o instanceof Float)
               raiz.setEValue(new Float( ((Float)o).floatValue()+1) );
         break;
         case OP_MENOSMENOS:
            if(o instanceof Integer)
               raiz.setEValue(new Integer( ((Integer)o).intValue()-1 ) );
            else if (o instanceof Float)
               raiz.setEValue(new Float( ((Float)o).floatValue()-1) );

         default: raiz.setEValue(null); break;
         }
      }
   }
   ;
exception catch [RecognitionException re]
{
   errorExpresion( re, (ExpressionAST)#expresionUnaria,
                   false, true, LeLiTypeManager.tipoError, null );
}

expresionBinaria
   :
   (  #(OP_MAS          expresion expresion)
   |  #(OP_MENOS        expresion expresion)
   |  #(OP_O            expresion expresion)
   |  #(OP_Y            expresion expresion)
   |  #(OP_IGUAL        expresion expresion)
   |  #(OP_DISTINTO     expresion expresion)
   |  #(OP_MAYOR        expresion expresion)
   |  #(OP_MENOR        expresion expresion)
   |  #(OP_MAYOR_IGUAL  expresion expresion)
```

```
  |  #(OP_MENOR_IGUAL  expresion expresion)
  |  #(OP_PRODUCTO     expresion expresion)
  |  #(OP_DIVISION     expresion expresion)
  |  #(OP_ASIG         eizq:expresion expresion)
     {
        if( !((ExpressionAST)eizq).getLValue() )
           throw new ContextException(
              "La parte izquierda de la asignación no tiene l-value",
              #expresionBinaria );
     }
  )
  {
     ExpressionAST raiz = (ExpressionAST) #expresionBinaria;
     // Calculamos r-value y l-value
     raiz.setLValue(false);
     raiz.setRValue(true);

     ExpressionAST ei = (ExpressionAST)(raiz.getFirstChild());
     ExpressionAST ed = (ExpressionAST)(ei.getNextSibling());

     if(!ei.getRValue())
        throw new ContextException(
           "Se necesita una expresión con r-value", ei );

     if(!ed.getRValue())
        throw new ContextException(
           "Se necesita una expresión con r-value", ed );

     int ti = ei.getExpType().getTag();
     int td = ed.getExpType().getTag();
     Type t = null;

     // Calculamos el tipo de la expresión binaria
     if(ti==TIPO_ERROR || td==TIPO_ERROR)
        t = LeLiTypeManager.tipoError;
     else
     {
        switch(raiz.getType())
        {
        case OP_MAS:
           if(ti==TIPO_CADENA || td==TIPO_CADENA)
              t = LeLiTypeManager.tipoCadena;
           else if(ti==TIPO_ENTERO && td==TIPO_ENTERO)
              t = LeLiTypeManager.tipoEntero;
           else if(isNumeric(ti) && isNumeric(td))
              t = LeLiTypeManager.tipoReal;
        break;
        case OP_MENOS: case OP_PRODUCTO: case OP_DIVISION:
           if(ti==TIPO_ENTERO && td==TIPO_ENTERO)
              t = LeLiTypeManager.tipoEntero;
           else if(isNumeric(ti) && isNumeric(td))
              t = LeLiTypeManager.tipoReal;
        break;
        case OP_ASIG:
           t = LeLiTypeManager.tipoVacio;
        break;
        case OP_Y: case OP_O:
           if(ti==TIPO_BOOLEANO && td==TIPO_BOOLEANO)
```

```
            t = LeLiTypeManager.tipoBooleano;
      break;
      case OP_IGUAL: case OP_DISTINTO:
         if( ti==TIPO_BOOLEANO && td==TIPO_BOOLEANO ||
             ti==TIPO_CADENA   && td==TIPO_CADENA ||
             isNumeric(ti) && isNumeric(td) )
            t = LeLiTypeManager.tipoBooleano;
      break;
      case OP_MAYOR: case OP_MENOR:
      case OP_MAYOR_IGUAL: case OP_MENOR_IGUAL:
         if( isNumeric(ti) && isNumeric(td) ||
             TIPO_CADENA==ti && TIPO_CADENA==td )
         t = LeLiTypeManager.tipoBooleano;
      break;
      }
   }

   if(null==t)
   {
      throw new ContextException( "El tipo '"+
         ei.getExpType() + "' no es compatible con '" +
         ed.getExpType() + "' con respecto al operador '"+
         raiz.getText() + "'",
         raiz);
   }

   raiz.setExpType(t);

   // Calculamos el valor de la expresión (si se puede)
   Object izq = ei.getEValue();
   Object der = ed.getEValue();

   if( null==izq || null==der )
      raiz.setEValue(null);
   else
   {
      String si = null, sd = null;
      Integer ii = null, id = null;
      Float fi = null, fd = null;
      Boolean bi = null, bd = null;

      if     (izq instanceof String)  si = (String)izq;
      else if(izq instanceof Integer) ii = (Integer)izq;
      else if(izq instanceof Float)   fi = (Float)izq;
      else if(izq instanceof Boolean) bi = (Boolean)izq;

      if     (der instanceof String)  sd = (String)der;
      else if(der instanceof Integer) id = (Integer)der;
      else if(der instanceof Float)   fd = (Float)der;
      else if(der instanceof Boolean) bd = (Boolean)der;

      switch(raiz.getType())
      {
      case OP_MAS:
         if(si!=null && (sd!=null || id!=null || fd!=null) )
            raiz.setEValue( si + der.toString() );
         if(si!=null && (bd!=null))
            raiz.setEValue( si + (bd.booleanValue()?"cierto":"falso") );
```

```
            if(sd!=null && (ii!=null || fi!=null) )
              raiz.setEValue( izq.toString() + sd );
            if(sd!=null && (bi!=null))
              raiz.setEValue( (bi.booleanValue()?"cierto":"falso") + sd );
            if(ii!=null && id!=null)
              raiz.setEValue( new Integer(ii.intValue()+id.intValue()) );
            if(fi!=null && fd!=null)
              raiz.setEValue( new Float(fi.floatValue()+fd.floatValue() ));
            if(fi!=null && id!=null)
              raiz.setEValue( new Float(fi.floatValue()+id.intValue()) );
            if(ii!=null && fd!=null)
              raiz.setEValue( new Float(ii.intValue()+fd.floatValue()) );
          break;
          case OP_MENOS:
            if(ii!=null && id!=null)
              raiz.setEValue( new Integer(ii.intValue()-id.intValue()) );
            if(fi!=null && fd!=null)
              raiz.setEValue( new Float(fi.floatValue()-fd.floatValue()) );
            if(fi!=null && id!=null)
              raiz.setEValue( new Float(fi.floatValue()-id.intValue()) );
            if(ii!=null && fd!=null)
              raiz.setEValue( new Float(ii.intValue()-fd.floatValue()) );
          break;
          case OP_PRODUCTO:
            if(ii!=null && id!=null)
              raiz.setEValue( new Integer(ii.intValue()*id.intValue()) );
            if(fi!=null && fd!=null)
              raiz.setEValue( new Float(fi.floatValue()*fd.floatValue()) );
            if(fi!=null && id!=null)
              raiz.setEValue( new Float(fi.floatValue()*id.intValue()) );
            if(ii!=null && fd!=null)
              raiz.setEValue( new Float(ii.intValue()*fd.floatValue()) );
          break;
          case OP_DIVISION:
            if(ii!=null && id!=null)
              raiz.setEValue( new Integer(ii.intValue()/id.intValue()) );
            if(fi!=null && fd!=null)
              raiz.setEValue( new Float(fi.floatValue()/fd.floatValue()) );
            if(fi!=null && id!=null)
              raiz.setEValue( new Float(fi.floatValue()/id.intValue()) );
            if(ii!=null && fd!=null)
              raiz.setEValue( new Float(ii.intValue()/fd.floatValue()) );
          break;
          case OP_O:
            if(bi!=null && bd!=null)
              raiz.setEValue( new Boolean(bi.booleanValue() || bd.booleanValue()
) );
          break;
          case OP_Y:
            if(bi!=null && bd!=null)
              raiz.setEValue( new Boolean(bi.booleanValue() && bd.booleanValue()
) );
          break;
          case OP_IGUAL:
            raiz.setEValue( new Boolean(izq.equals(der)) );
          break;
          case OP_DISTINTO:
            raiz.setEValue( new Boolean(!izq.equals(der)) );
```

```
        break;
        case OP_MAYOR:
           if(si!=null && sd!=null)
              raiz.setEValue( new Boolean(si.compareTo(sd)>0) );
           if(ii!=null && id!=null)
              raiz.setEValue( new Boolean(ii.compareTo(id)>0) );
           if(fi!=null && fd!=null)
              raiz.setEValue( new Boolean(fi.compareTo(fd)>0) );
           if(fi!=null && id!=null)
              raiz.setEValue( new Boolean(fi.compareTo(id)>0) );
           if(ii!=null && fd!=null)
              raiz.setEValue( new Boolean(ii.compareTo(fd)>0) );
        break;
        case OP_MENOR:
           if(si!=null && sd!=null)
              raiz.setEValue( new Boolean(si.compareTo(sd)<0) );
           if(ii!=null && id!=null)
              raiz.setEValue( new Boolean(ii.compareTo(id)<0) );
           if(fi!=null && fd!=null)
              raiz.setEValue( new Boolean(fi.compareTo(fd)<0) );
           if(fi!=null && id!=null)
              raiz.setEValue( new Boolean(fi.compareTo(id)<0) );
           if(ii!=null && fd!=null)
              raiz.setEValue( new Boolean(ii.compareTo(fd)<0) );
        break;
        case OP_MAYOR_IGUAL:
           if(si!=null && sd!=null)
              raiz.setEValue( new Boolean(si.compareTo(sd)>=0) );
           if(ii!=null && id!=null)
              raiz.setEValue( new Boolean(ii.compareTo(id)>=0) );
           if(fi!=null && fd!=null)
              raiz.setEValue( new Boolean(fi.compareTo(fd)>=0) );
           if(fi!=null && id!=null)
              raiz.setEValue( new Boolean(fi.compareTo(id)>=0) );
           if(ii!=null && fd!=null)
              raiz.setEValue( new Boolean(ii.compareTo(fd)>=0) );
        break;
        case OP_MENOR_IGUAL:
           if(si!=null && sd!=null)
              raiz.setEValue( new Boolean(si.compareTo(sd)<=0) );
           if(ii!=null && id!=null)
              raiz.setEValue( new Boolean(ii.compareTo(id)<=0) );
           if(fi!=null && fd!=null)
              raiz.setEValue( new Boolean(fi.compareTo(fd)<=0) );
           if(fi!=null && id!=null)
              raiz.setEValue( new Boolean(fi.compareTo(id)<=0) );
           if(ii!=null && fd!=null)
              raiz.setEValue( new Boolean(ii.compareTo(fd)<=0) );
        break;
        default :
           raiz.setEValue(null);
        break;
        }
     }
  }
  ;
exception catch [RecognitionException re]
{ errorExpresion(re, (ExpressionAST)#expresionBinaria); }
```

```
//-------------- FASE 4: Manejo de tipos en acceso --------------
//
// Esta es la fase más complicada. Aquí se tratan accesos como
//
//   Persona pedro(19,"Pedro");
//   Cadena ro = pedro.obtenerNombre().subcadena(4,5);
//
// El problema está en que en cada subacceso
// (cada punto "." de "pedro.obtenerNombre().subcadena(4,5))
// hay que cambiar de juego de subaccesos válidos (hay que
// cambiar de ámbito).
//

acceso
{ ExpressionAST arbolActual = null; }
   : #( ACCESO r:raizAcceso
        { arbolActual = (ExpressionAST)r; }
        ( s:subAccesoContexto[arbolActual]
          { arbolActual = (ExpressionAST)s; }
        )*
      )
   {
      ExpressionAST raiz = (ExpressionAST)#acceso;
      raiz.setRValue(arbolActual.getRValue());
      raiz.setLValue(arbolActual.getLValue());
      raiz.setEValue(arbolActual.getEValue());
      raiz.setExpType(arbolActual.getExpType());
   }
   ;
exception catch [RecognitionException re]
{ errorExpresion(re, (ExpressionAST)#acceso); }

raizAcceso
{ ExpressionAST ast = (ExpressionAST)#raizAcceso; }
   :
   i:IDENT
   {
      ast.setRValue(true);
      ast.setLValue(true);

      Declaration d = contexto.obtenerDeclaracion(i);

      // Cálculo del tipo
      switch(d.getTag())
      {
         case INST_DEC_VAR: case RES_PARAMETRO:
            ast.setExpType(d.getType());
         break;
         case RES_ATRIBUTO: // acceso a un atributo
            // Obtener el "tipo atributo" (AttributeType)
            // Recordar que AttributeType es similar a MethodType
            AttributeType tipoAtributo = (AttributeType)(d.getType());
            // Y de ahí el verdadero tipo (Entero, Cadena, etc)
            ast.setExpType(tipoAtributo.getType());
         break;
         case RES_CLASE: // acceso a un miembro abstracto
         case TIPO_ENTERO: case TIPO_REAL:
```

```
        case TIPO_BOOLEANO: case TIPO_CADENA:
           LeLiType clase = (LeLiType)(d.getType());
           LeLiMetaType metaClase = clase.getMetaType();
           ast.setExpType(metaClase);
        break;
        default: // incluye TIPO_ERROR
           ast.setExpType(LeLiTypeManager.tipoError);
        break;
      }

      // Cálculo del E-Value
      // ExpressionAST iv = (ExpressionAST)d.getInitialValue();
      // if(iv!=null)
      //   ast.setEValue(iv.getEValue());
      }
    | ( RES_PARAMETRO { ast.setExpType(tipoParametroAux); }
      | RES_ATRIBUTO  { ast.setExpType(tipoAtributoAux);  }
      | RES_SUPER     { ast.setExpType(claseActual.getSuperType()); }
      )
      {
         ast.setLValue(false);
         ast.setRValue(true);
         ast.setEValue(null);
      }
    | literal
    | llamadaContexto [claseActual]
    | conversion
    | expresion
    ;
exception catch [RecognitionException re]
{ errorExpresion(re, ast); }

literal
{
   ExpressionAST ast = (ExpressionAST)#literal;
   ast.setRValue(true);
   ast.setLValue(false);
}
   : LIT_ENTERO
     {
        ast.setEValue(new Integer(ast.getText()));
        ast.setExpType(LeLiTypeManager.tipoEntero);

     }
   | LIT_REAL
     {
        ast.setEValue(new Float(ast.getText()));
        ast.setExpType(LeLiTypeManager.tipoReal);
     }
   | LIT_CADENA
     {
        ast.setEValue(ast.getText());
        ast.setExpType(LeLiTypeManager.tipoCadena);
     }
   | LIT_NL
     {
        ast.setEValue("\n");
        ast.setExpType(LeLiTypeManager.tipoCadena);
```

```
      }
    | LIT_TAB
      {
         ast.setEValue("\t");
         ast.setExpType(LeLiTypeManager.tipoCadena);
      }
    | LIT_COM
      {
         ast.setEValue("\"");
         ast.setExpType(LeLiTypeManager.tipoCadena);
      }
    | LIT_CIERTO
      {
         ast.setEValue(new Boolean(true));
         ast.setExpType(LeLiTypeManager.tipoBooleano);
      }
    | LIT_FALSO
      {
         ast.setEValue(new Boolean(false));
         ast.setExpType(LeLiTypeManager.tipoBooleano);
      }
    ;
exception catch [RecognitionException re]
{ errorExpresion(re, (ExpressionAST)#ast); }

llamadaContexto [ LeLiType llamador ]
{
   ExpressionAST ast = (ExpressionAST)#llamadaContexto;
   Declaration d = null;
   LeLiMethodType lmt = null;

   // Comprobar si se está usando la palabra reservada
   // "parámetro" o "atributo" seguidas de una llamada a
   // un método o constructor
   if( llamador.getTag() == RES_PARAMETRO ||
       llamador.getTag() == RES_ATRIBUTO )
   {
      throw new ContextException(
        "Las palabras reservadas 'parámetro' y 'atributo' no pueden "+
        "ir seguidas de una llamada a un método o un constructor",
        ast );
   }
}
   :
   ( #( LLAMADA nombre:IDENT
        { lmt = new LeLiLlamadaType(nombre.getText(), llamador); }
        listaExpresionesContexto [lmt]
        { d = llamador.buscarMetodoCompatible(lmt); }
      )
   | #( RES_CONSTRUCTOR
          { lmt = new LeLiLlamadaConstructorType(llamador); }
          listaExpresionesContexto[lmt]
          { d = llamador.buscarConstructorCompatible(lmt); }
        )
    )
    {
     if( llamador.getTag()==TIPO_ERROR )
     {
```

```
        ast.setExpType(LeLiTypeManager.tipoError);
        ast.setRValue(true);
        ast.setLValue(true);
      }
      else if( null==d )
      {
        throw new ContextException(
          "La clase '" + llamador.getName() +
          "' no contiene un método o constructor compatible con " +
          lmt.toString(), ast );
      }
      else
      {
        ast.setLValue(false);
        ast.setEValue(null);

        LeLiMethodType decMetodo= (LeLiMethodType)d.getType();

        ast.setExpType(decMetodo.getReturnType());
        if( ast.getExpType().getTag()==TIPO_VACIO )
          ast.setRValue(false);
        else
          ast.setRValue(true);
      }
     }
     ;
exception catch [RecognitionException re]
{ errorExpresion(re, ast); }

listaExpresionesContexto [LeLiMethodType lmt]
   : #( LISTA_EXPRESIONES
        ( e:expresion
          { lmt.addParam(((ExpressionAST)e).getExpType()); }
        )* )
   ;
exception catch [RecognitionException re] { reportError(re); }

conversion
{
   ExpressionAST ast = (ExpressionAST)#conversion;
   ast.setLValue(false);
   ast.setRValue(true);
}
   :
   #(RES_CONVERTIR e:expresion t:tipo)
   {
      ExpressionAST ee = (ExpressionAST)e;
      LeLiType te = (LeLiType)ee.getExpType();
      LeLiType tt = (LeLiType)((TypeAST)t).getExpType();

      if( tt.getTag()==TIPO_ERROR ||
          tt.getName().equals("Objeto") )
      {
        ast.setExpType(tt);
      }
      else if( te.isA(tt) )
      {
        ast.setExpType(tt);
```

```
      }
      // Convertir tipos básicos y el resto
      switch( tt.getTag() )
      {
      case TIPO_CADENA: // Convertir a cadena
         switch(te.getTag())
         {
         case TIPO_ENTERO: case TIPO_REAL:
            ast.setExpType(tt);
            if(null==ee.getEValue())
               ast.setEValue(null);
            else
               ast.setEValue(ee.getEValue().toString());
         break;
         case TIPO_BOOLEANO:
            ast.setExpType(tt);
            if(null==ee.getEValue())
               ast.setEValue(null);
            else
            {
               boolean b = ((Boolean)ee.getEValue()).booleanValue();
               ast.setEValue(b?"cierto":"falso");
            }
         // Comprobar el resto en tiempo de ejecución
         // (ej. conversión Objeto->Cadena)
         }
      break;
      case TIPO_REAL: // Convertir a Real
         switch(te.getTag())
         {
            case TIPO_ENTERO:
               ast.setExpType(tt);
               if(null==ee.getEValue())
                  ast.setEValue(null);
               else
               {
                  int i = ((Integer)ee.getEValue()).intValue();
                  ast.setEValue(new Float(i));
               }
            break;
            // Comprobar el resto en tiempo de ejecución
            // (ej. conversión Objeto->Real)
         }
      break;
      default:
         if( !te.isA(tt) && !tt.isA(te))
            throw new ContextException(
               "La expresión no es convertible al tipo '"+tt+"'",
               e );
         ast.setExpType(tt);
         ast.setEValue(null);
         // Comprobar el resto en tiempo de ejecución
         // (ej. conversión Objeto->Tipo def. usuario)
      }
   }
   ;
exception catch [RecognitionException re]
{ errorExpresion(re, (ExpressionAST)ast); }
```

```
subAccesoContexto [ExpressionAST anteriorAST]
{
   ExpressionAST ast = (ExpressionAST)#subAccesoContexto;
   ast.setExpType(LeLiTypeManager.tipoError);
   ast.setRValue(true);
   ast.setLValue(true);
   ast.setEValue(null);
   LeLiType tipoAnterior = (LeLiType)(anteriorAST.getExpType());
}
   : llamadaContexto [tipoAnterior]
   | RES_SUPER
      {
         ast.setExpType(contexto.obtenerSuperTipo(tipoAnterior, ast));
      }
   | nombre:IDENT
      {
        Declaration d = null;
        switch(tipoAnterior.getTag())
        {
           // Comprobar si se está usando la palabra reservada
           // "parámetro" o "atributo" como raíz del acceso
           case RES_PARAMETRO:
              d = contexto.obtenerObjeto(RES_PARAMETRO, nombre);
              ast.setExpType(d.getType());
           break;

           case RES_ATRIBUTO:
           {
              d = contexto.obtenerObjeto(RES_ATRIBUTO, nombre);
              // La clase que representa el tipo de los atributos
              // es "AttributeType". Hay que obtener el tipo
              // "real" (Entero, Cadena) del atributo,
              // haciendo getType().
              AttributeType tipoAtributo = (AttributeType)d.getType();
              ast.setExpType(tipoAtributo.getType());
           }
           break;

           // Atributos "normales" - también hay que "sacar" el atributo
           default:
           {
              d = contexto.obtenerAtributo( tipoAnterior, nombre );
              AttributeType tipoAtributo = (AttributeType)d.getType();
              ast.setExpType(tipoAtributo.getType());
           }
        break;
        }
        ExpressionAST valorAST = (ExpressionAST)d.getInitialValue();
        if(valorAST!=null)
        ast.setEValue(valorAST.getEValue());
      }
    ;
exception catch [RecognitionException re]
{ errorExpresion(re, (ExpressionAST)#subAccesoContexto); }
```

7.6.8: Notas finales sobre el analizador

La primera nota sobre el analizador es parecida a la que se hacía sobre `LeLiSymbolTreeParser`: depende profundamente de `LeLiContext`, donde se realizan buen número de acciones.

Otra nota importante es que con este analizador damos por terminado el análisis semántico, pero *no está realmente terminado*. Algunas de las cosas que no hemos realizado son:

- Comprobar que las expresiones de las instrucciones condicionales son siempre de tipo booleano (es muy fácil de realizar)
- Ídem con las condiciones de salida de los bucles
- En los constructores de las clases todos los atributos deberían ser inicializados. Ésto debería comprobarse.
- Hace falta testear más algunas zonas del código. Seguro que algunos *bugs* se han escapado a pesar de mis esfuerzos.
- No hay recuperación de errores en ninguno de los dos iteradores; el compilador no es capaz de recuperarse de errores que generen un AST incompleto.

Nota número tres: A la hora de calcular el tipo y E-value de las expresiones (especialmente las expresiones binarias y unarias) he optado por agrupar todas las alternativas en un solo bloque y utilizar "bloques switch masivos", en lugar de realizar acciones independientes en cada alternativa. Resulta que de la primera forma el código ocupa menos (las variables solamente se declaran una vez, y en algunos casos se pueden agrupar varias opciones del `switch` en un solo `case`).

Por último, téngase presente que en la primera pasada se incluyeron los ASTs de las declaraciones de los tipos básicos en el AST generado. Esto tiene la interesante consecuencia de que *la segunda pasada actúa también sobre los ASTs de los tipos predefinidos*.

Sección 7.7: Compilación y ejecución

7.7.1: Compilación

Compilar los analizadores es sencillo si tenemos en cuenta cómo lo hemos hecho en ocasiones anteriores.

Compilar el iterador simple es tan sencillo como escribir:

```
C:\> java antlr.Tool LeLiParser.g
```

Para compilar `LeLiSymbolTreeParser.g` necesitaremos que `LeLiParser.g` esté accesible en el `path`. Luego hay que escribir:

```
C:\> java antlr.Tool -glib LeLiParser.g LeLiSymbolTreeParser.g
```

`LeLiTypeCheckTreeParser` puede compilarse de forma similar.

```
C:\> java antlr.Tool -glib LeLiParser.g LeLiTypeCheckTreeParser.g
```

`LeLiSymbolTreeParser.g` debe compilarse antes de compilar `LeLiTypeCheckTreeParser.g`, pues el segundo importa el vocabulario del primero.

Luego hay que compilar los ficheros *.java generados:

```
C:\> javac *.java
```

7.7.2: Ejecución

Cambios en la clase Tool

Aunque tenga poco interés para el usuario medio de un compilador, a mí me ha resultado muy útil poder lanzar mis iteradores independientemente con un comando especial, sin tener que recompilar. Para poder hacerlo, he añadido a `Tool` un nuevo comando que permite especificar con un número entero el "nivel" de análisis semántico que se desee producir (0 para una "iteración pasiva sobre el AST", 1 para aplicarle únicamente `LeLiSymbolTreeParser` y 2 para aplicarle también `LeLiTypeCheckTreeParser`.

Se han añadido algunas líneas para activar la nomenclatura "verbosa" de los ASTs, de manera que muestren información adicional al ser imprimidos por pantalla. La configuración actual es la de mostrar la información adicional de todos los tipos de AST excepto `LexInfoAST`. Este comportamiento solamente puede cambiarse recompilando.

Código de la nueva clase Tool

A continuación se lista el código de la nueva clase Tool. Las zonas que han cambiado aparecen señaladas.

```
package leli; // Tool forma parte del paquete LeLi

import java.io.*;

import antlr.collections.AST;
import antlr.collections.impl.*;
import antlr.debug.misc.*;
import antlr.*;
```

```
import antlraux.clparse.*;
import antlraux.context.*;
import antlraux.context.asts.*;
import antlraux.util.*;
import antlraux.util.LexInfoAST;
import antlraux.util.Logger;


public class Tool
{
   public String mensajeAyuda="";
   public boolean mostrarVentana=false;
   public int nivelSemantico=1;
   public String nombreFichero="";
   public FileInputStream fis=null;
   public boolean recuperacion=true;
   public boolean imprimeArbol=false;

   public void fijarMostrarVentana(Boolean B)
   { mostrarVentana=B.booleanValue(); }

   public void fijarNivelSemantico(Integer I)
   { nivelSemantico=I.intValue(); }

   public void fijarNombreFichero(String s)
   { nombreFichero=s;   }

   public void fijarRecuperacion(Boolean B)
   { recuperacion = B.booleanValue(); }

   public void fijarImprimeArbol(Boolean B)
   { imprimeArbol = B.booleanValue(); }

   public Tool ()
   { }

   public void leeLC(String args[])
   throws CommandLineParserException
   {
      CommandLineParser clp =
         new CommandLineParser("leli.Tool", args);
      clp.addCommand(this, "-ventana", "fijarMostrarVentana", "b",
                     "Enseña o no la ventana del AST (defecto no)");
      clp.addCommand(this, "-nsem", "fijarNivelSemantico", "i",
                     "Nivel semántico: 0 nada, 1 solo recorrer (defecto),"+
                     "2 chequeo de tipos 3 total");
      clp.addCommand(this, "-f", "fijarNombreFichero", "s",
                     "Fija el nombre del fichero a reconocer");
      clp.addCommand(this, "-erec", "fijarRecuperacion", "b",
                    "Activa/desactiva la recuperación de errores
sintácticos" );
      clp.addCommand(this, "-imprime", "fijarImprimeArbol", "b",
                     "Imprime el AST en la salida estándar");
      mensajeAyuda = clp.getUsageMessage(true);
      clp.parseWhilePossible();
      clp.executeWhilePossible();
```

```
    if( nombreFichero==null ||
        nombreFichero.equals("") )
    {
       throw new CommandLineParserException(
          "Se necesita un nombre de fichero");
    }

    try
    {
       fis = new FileInputStream(nombreFichero);
    }catch (FileNotFoundException fnfe){
         throw new
       CommandLineParserException(
          "El fichero '"+nombreFichero+"' no se pudo abrir");
    }
  }

  public void imprimeAyuda()
  {
    System.err.println(mensajeAyuda);
  }


  public void trabaja()
  {
    Logger logger = new Logger("error", System.err);
    try
    {
       System.out.println("Reconociendo el fichero '"+nombreFichero+"'");

       // PASOS 2 y 3. Crear analizador léxico y pasarle nombreFichero
       // y nueva clase Token
       LeLiLexer lexer = new LeLiLexer(fis);
       lexer.setFilename(nombreFichero);
       lexer.setTokenObjectClass("antlraux.util.LexInfoToken");

       // PASOS 4 y 5. Crear analizador sintáctico y pasarle
       // nombre fichero

       antlr.Parser parser = null;

       if(recuperacion)
          parser = new LeLiErrorRecoveryParser(lexer, logger);
       else
          parser = new LeLiParser(lexer);

       parser.setFilename(nombreFichero);
       parser.setASTNodeClass("antlraux.util.LexInfoAST");

       // PASO 6. Comenzar el análisis
       if(recuperacion)
          ((LeLiErrorRecoveryParser)parser).programa();
       else
          ((LeLiParser)parser).programa();


       int erroresSint = logger.getLogCounter();
       if(erroresSint>0)
```

```
      {
         System.err.println("errores sintácticos: "+ erroresSint);
      }

      // PASO 7. Obtener el AST
      AST ast = parser.getAST();

      // Aux: verboseStringConversion
      BaseAST.setVerboseStringConversion(
         true,
         LeLiSymbolTreeParser._tokenNames);
      ScopeAST.setVerboseStringConversion(true);
      LexInfoAST.setVerboseStringConversion(false);
      ExpressionAST.setVerboseStringConversion(true);
      TypeAST.setVerboseStringConversion(true);
      DeclarationAST.setVerboseStringConversion(true);

      // PASO 8. Recorrer el AST
      TreeParser treeParser = null;
      if(nivelSemantico==0)
      {
         treeParser = new LeLiTreeParser();

         ((LeLiTreeParser)treeParser).programa(ast);
      }
      if(nivelSemantico>=1)
      {
         treeParser =
            new LeLiSymbolTreeParser(logger);

         ((LeLiSymbolTreeParser)treeParser).programa(ast);
         ast = treeParser.getAST();
      }
      if(nivelSemantico>=2)
      {
         LeLiContext contexto =
            ((LeLiSymbolTreeParser)treeParser).obtenerContexto();

         treeParser =
            new LeLiTypeCheckTreeParser(logger, contexto);

         ((LeLiTypeCheckTreeParser)treeParser).programa(ast);
      }

      // PASO 9. Mostrar el AST por pantalla
      if(imprimeArbol==true)
      {
         System.out.println(ast.toStringList());
      }

      // PASO 10. Representación en ventana del AST
      if(mostrarVentana)
      {
         ASTFrame frame = new ASTFrame(nombreFichero, ast);
         frame.setVisible(true);
      }
   }
   catch (TokenStreamException tse)
```

```
      {
         tse.printStackTrace(System.err);
      }
      catch (RecognitionException re)
      {
         re.printStackTrace(System.err);
      }
   }

   public static void main(String args[])
   {
      Tool t = new Tool();

      try{
         t.leeLC(args);
         t.trabaja();
      }catch(CommandLineParserException clpe){
         System.err.println(clpe.getMessage());
         System.err.println(t.mensajeAyuda);
      }
   }
}
```

Sección 7.8: Conclusión

El análisis semántico es el capítulo más largo del documento. Por un lado hay teoría en abundancia, y por el otro se describen en profundidad *tres* analizadores, cuando lo normal es describir uno por capítulo. Además, se describe el funcionamiento del paquete más voluminoso de antlraux.

Podríamos habernos extendido mucho más; al análisis semántico le pasa un poco como a la gestión de errores – se extiende hasta donde uno esté dispuesto a tolerar. Ya mencioné en las notas de `LeLiTypeCheckTreeParser` que faltan algunos filos por pulir.

Pero el objetivo de este documento no es cubrir el desarrollo de un compilador ideal. Es mucho más didáctico: enseñar a hacer un compilador, utilizando el desarrollo de uno como ejemplo. Y no veo cómo añadir más niveles al análisis puede contribuir a alcanzar este objetivo.

He aquí el AST de un "Hola mundo" con errores sintácticos (de los que el compilador se recuperó sin problemas). Nótese que aparecen los tipos predefinidos del sistema además de la clase `Inicio`:

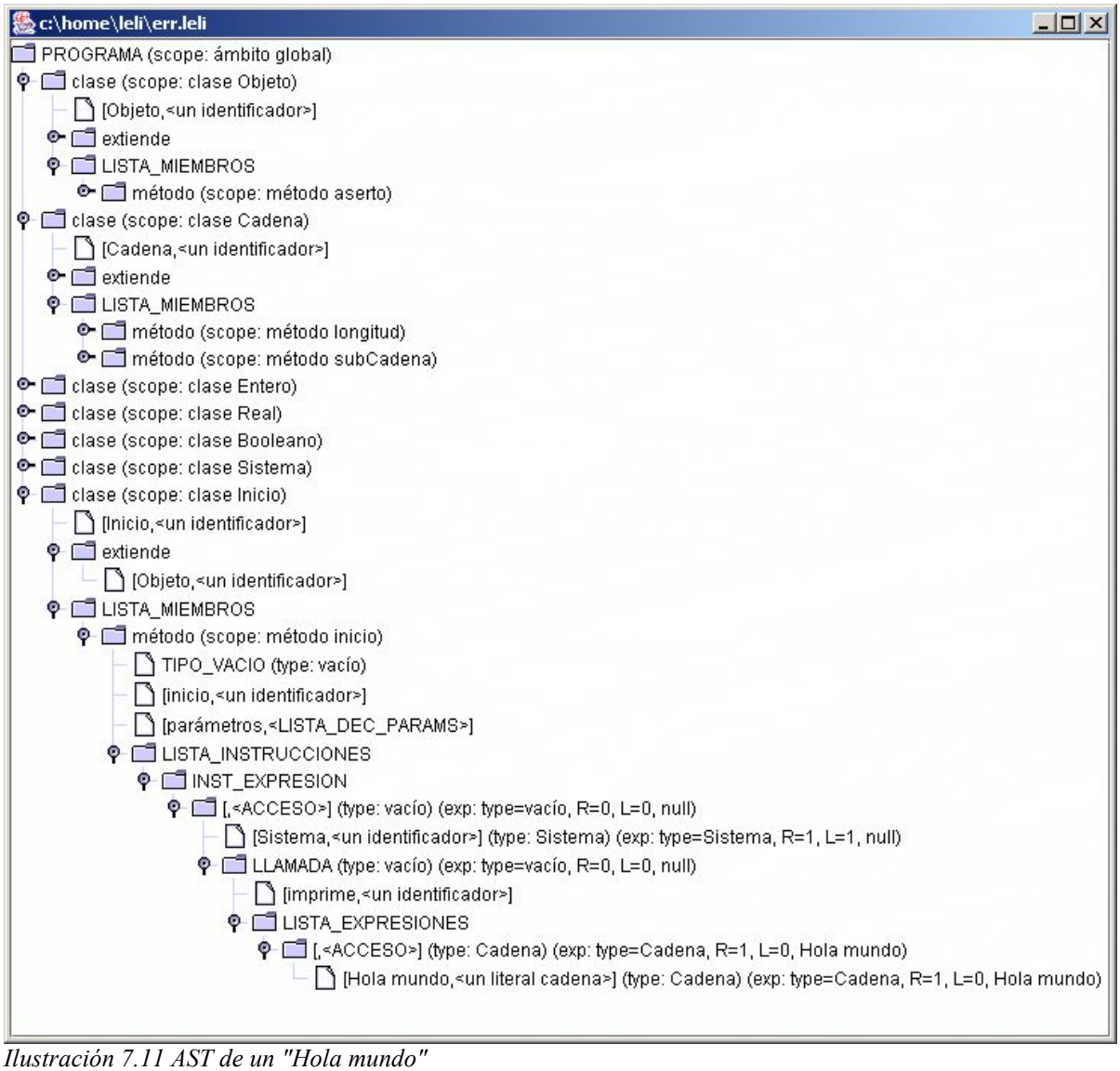

Ilustración 7.11 AST de un "Hola mundo"

Capítulo 8: Generación de código

"Si torturas suficiente a los datos, acabarán confesando"
Ronald Coase

Capítulo 8: Generación de código.....365

Sección 8.1: Introducción.....367
8.1.1: No generaremos código..... 367
8.1.2: Estructura del capítulo..... 367
Sección 8.2: Las Máquinas.....368
8.2.1: Definiciones..... 368
8.2.2: Componentes principales..... 368
La UAL.....368
La memoria.....369
Los registros.....369
Los buses.....369
Otros componentes.....370
8.2.3: Juego de órdenes..... 370
Las órdenes.....370
Órdenes de movimiento y modos de direccionamiento.....370
Órdenes de cálculo aritmético.....371
Órdenes de salto incondicional.....372
Saltos condicionales.....372
Otras 373
El tamaño importa.....373
8.2.4: Las otras máquinas..... 374
Sección 8.3: Gestión de la memoria.....376
8.3.1: Definición..... 376
8.3.2: División clásica de la memoria..... 376
La zona de código.....376
Zona de datos estáticos.....377
La pila.....377
El montículo.....377
8.3.3: Acerca de la pila..... 378
Orígenes de la pila.....378
Invocando funciones.....379
El hardware de la pila.....380
Invocando funciones en C.....380
Invocando métodos.....383
El valor de retorno de una función o un método.....385
8.3.4: Acerca del motículo..... 386
Motivación del montículo.....386
Implementación.....387
Liberación explícita de memoria: Modelo de C/C++.....388
Liberación implícita de memoria: Recolección de basura.....390
Hardware del montículo.....391
El montículo y LeLi.....391
8.3.5: Acerca del offset..... 391
Problema.....391
La jerarquía de declaraciones.....391
Tamaño de un tipo.....392
Disposición de los objetos en la memoria.....392
El desplazamiento.....393
Cálculo de las direcciones de las instancias.....393
Sección 8.4: Código intermedio.....395
8.4.1: Introducción..... 395

8.4.2: Proposiciones ... 395
8.4.3: Objetivos y ventajas del código intermedio ... 396
Sección 8.5: Generación de instrucciones y expresiones ... 397
8.5.1: Introducción ... 397
8.5.2: Instrucciones condicionales ... 397
Caso simple: un solo bloque condicional ... 397
Caso 2: varios bloques condicionales ... 397
Caso 3: Instrucción condicional completa ... 398
Gestión de las etiquetas ... 398
8.5.3: Bucles ... 398
Bucle mientras ... 398
Bucle hacer-mientras ... 399
Bucle desde ... 399
8.5.4: Instrucciones/expresiones ... 399
Expresiones aritmético-lógicas: registros y valores temporales ... 399
Asignaciones ... 400
Accesos a atributos (NO a métodos) ... 400
Accesos a métodos (invocaciones) ... 401
Sección 8.6: Optimización de código ... 402
8.6.1: Introducción ... 402
8.6.2: Optimizaciones independientes de la máquina objetivo ... 402
Eliminación de subexpresiones comunes ... 402
Eliminación de código inactivo ... 402
Transformaciones algebraicas ... 403
Optimizaciones relacionadas con constantes e invariantes ... 403
Optimizaciones de reordenamiento de cálculo ... 403
8.6.3: Optimizaciones dependientes de la máquina objetivo ... 404
Asignación de registros ... 404
Transformaciones de reordenamiento ... 404
Desenrollado de bucles ... 404
Uso de hardware específico ... 405
8.6.4: Portabilidad vs. eficiencia ... 406
Código abierto ... 406
Librerías del sistema ... 406
Compilación JIT ... 407
Sección 8.7: Miscelánea ... 408
8.7.1: Introducción ... 408
8.7.2: Mapeado de tipos ... 408
Mapeado de flotantes sobre enteros ... 408
Mapeado de caracteres y booleanos ... 409
Mapeado de cadenas de caracteres y tablas ... 409
Mapeado de tipos definidos por el usuario ... 409
8.7.3: Librerías y tipos básicos del sistema ... 410
Opción 1: Código embebido ... 410
Opción 2: Condiciones especiales controladas ... 411
Opción 3: Soporte nativo ... 411
Sección 8.8: Conclusión ... 412

Sección 8.1: Introducción

La generación de código es la última fase del proceso de compilación:

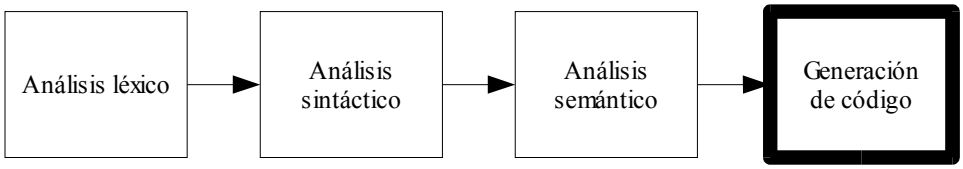

Ilustración 8.1 La última fase del proceso

La generación de código toma como entrada una representación abstracta del código fuente y produce como resultado un código equivalente.

8.1.1: No generaremos código

A pesar de ser una parte muy importante de un compilador, no implementaremos ningún generador de código para el nuestro. La principal razón de esta decisión es que un generador de código no ilustra ninguna capacidad de ANTLR que no hayamos visto ya; la única – si se usa alguna – es la de recorrer ASTs, cosa que ya hemos hecho varias veces en la fase del análisis semántico. He decidido concentrarme en el análisis semántico para ilustrar el funcionamiento de ANTLR, y he dejado de lado la implementación del generador de código.

A lo poco ilustrativo del caso se han unido estas otras razones:

- Mi incapacidad por decidirme sobre la máquina objetivo del compilador
- El hecho de que en ningún caso estamos implementando un compilador comercial: como ya mencioné en la introducción de este documento, LeLi adolece de varias carencias que lo hacen inviable para este propósito (como la incapacidad para utilizar punteros o tablas).
- En muchos casos la generación de código no es necesaria: la mayoría de los lectores de este documento se contentarán con realizar las acciones pertinentes tras haber construído un AST adecuado, como hicimos en el capítulo 2 con microcalc.

Dicho esto, me gustaría hacer una aclaración: aunque no implementemos generador alguno, sí que indicaremos cómo se debe implementar cada aspecto del generador.

8.1.2: Estructura del capítulo

Todo generador de código tiene una "máquina objetivo"; es decir, la máquina sobre la cual se ejecutará el código generado. Muchos de los algoritmos de optimización de código se basan en un profundo conocimiento de la máquina objetivo. En la siguiente sección analizaremos la estructura que tienen las computadoras actuales.

La sección 8.3 estará dedicada al tema de la gestión de memoria durante la generación de código; definiremos conceptos como offset, pila y montículo.

En la seccion 8.4 nos detendremos a analizar el código intermedio.

Posteriormente, en la sección , analizaremos cómo tratar los "casos especiales" que aparecen en el lenguaje (¿cómo tratar los tipos básicos del lenguaje? ¿la información de tipos?).

En otra sección, la 8.6, trataremos el tema de la optimización.

Por último habrá una sección dedicada a las conclusiones y sugerencias.

Sección 8.2: Las Máquinas

8.2.1: Definiciones

Normalmente el código generado durante la generación de código tiene como finalidad ser "ejecutado" en algún dispositivo. A este dispositivo lo llamaremos la "máquina objetivo".

Por "máquina" entenderemos "cualquier mecanismo capaz de realizar las acciones codificadas en un código generado por un compilador". Al proceso de "realizar las acciones codificadas en un código generado" se le suele llamar *ejecutar* un código.

Es una definición bastante pobre; no se define qué es un "mecanismo" ni qué son "acciones" ni "código". Tomando la definición al pie de la letra, un coche que se detuviera ante un semáforo en rojo y esperara a que se pusiera en verde para continuar entraría dentro de esta definición. Al fin y al cabo, se trata de un "mecanismo" (un coche) que ejecuta una "acción" (esperar la verde para continuar) codificada en un "código" (el de los semáforos).

Para restringir un poco esta definición, añadiremos que "el código" es "el resultado de un proceso satisfactorio de compilación de un código fuente utilizando un compilador", y las "acciones" son "los algoritmos contenidos en dicho código fuente".

Nos abstendremos por el momento de definir estrictamente qué entendemos por "mecanismo". Retomaremos la cuestión en el último apartado de la sección.

8.2.2: Componentes principales

Como ya indicamos en la sección 1.2.1 "Conceptos básicos: software y hardware" en un procesador hay 3 elementos básicos: una unidad aritmético-lógica, o UAL, un conjunto de registros, y un módulo de memoria, unidos todos mediante buses de información:

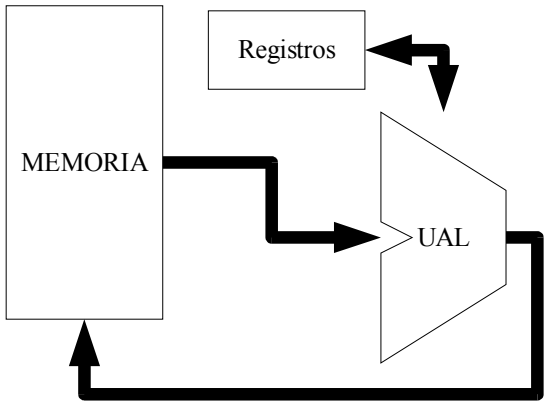

Ilustración 8.2 Esquema de una CPU

La UAL

Es el conjunto de mecanismos de la computadora cuya función es realizar los cálculos del programa. Por "cálculos" entendemos cualquier operación aritmética (suma, resta, multiplicación, división) o lógica (comparación) entre elementos provinentes tanto de la memoria como de los registros.

La memoria

Por "memoria" entenderemos cualquier tipo de dispositivo de almacenamiento de datos que haya en el ordenador. Esto incluye memoria RAM, caché del procesador, discos duros y flexibles, CD-ROMS etc. No se incluyen los registros en este conjunto.

Es en la memoria donde se almacenan los datos necesarios para poder ejecutar programas; esto incluye los propios programas (el *software* del que hablábamos en el primer capítulo).

Los registros

Los registros son pequeñas unidades de memoria de muy baja capacidad, pero que pueden ser accedidos más rápidamente que la memoria. Existen dos tipos de registro: los de uso general y los específicos.

Los registros de uso general son aquellos que pueden ser utilizados por los programas para almacenar datos que intervienen en los cálculos. El número de registros de uso general varía de un modelo de microprocesador a otro, y también cambia su tamaño y los nombres que reciben.

Inicialmente había 4 registros de uso general en las máquinas 80x86 (desarrolladas por Intel y AMD en la actualidad): A, B, C y D, y tenían 8 bits de capacidad. Más tarde "se extendieron", llegando a tener 16 bits, y llamándose EA, EB, EC, ED. A, B, C y D podían seguir siendo referenciados por el código máquina como los 8 bits menos significativos de sus hermanos mayores. En la siguiente versión de la arquitectura se volvieron a extender los registros, llegando a los 32 bits, y llamándose EAX, EBX, ECX, EDX. Tanto los registros de 8 como los de 16 bits siguien siendo utilizables como la parte menos significativa de estos registros.

Los registros específicos también pueden ser utilizados por los programas, pero no de la misma forma que los registros de uso general; en ellos no puede insertarse cualquier dato que intervenga en un cálculo; hay que insertar datos concretos que modifican el comportamiento de la máquina.

Algunos ejemplos de registros específicos son:

- El contador del programa: es el registro en el que se guarda la siguiente línea de código máquina que se ejecutará. Modificándolo se consigue cambiar el flujo del programa.
- El registro de comparaciones y banderas: es el registro en el que se guardan los resultados de algunas operaciones; en uno de sus bits se guarda el resultado de la última comparación booleana (1 o 0 para cierto o falso) y en otros bits hay diversas "banderas" (*flags*) dedicadas a indicar si ha habido un *overflow* o si la memoria accedida es "escribible".

El contador del programa en las máquinas compatibles con la arquitectura 80x86 se llama EIP (*Extended Instruction Pointer*). Los resultados de las comparaciones se guardan en el registro A (EAX).

Los buses

La única utilidad de los buses de datos es la de transportar información. Son elementos "pasivos", en el sentido de que no modifican la estructura lógica de la CPU; son solamente un transporte, un soporte físico, como el plástico que soporta el silicio.

Ahora bien, desde el punto de vista práctico, los buses son importantes porque pueden constituir cuellos de botella del sistema: puede que los componentes generen información demasiado deprisa de lo que los buses sean capaces de manejar, lo que puede provocar errores y disminución de la eficiencia. Si hay buses de diferentes velocidades en la máquina, esto supondrá interesantes problemas de optimización del código.

Otros componentes

Un computador que solamente disponga de los 4 componentes enunciados previamente será inútil. Los ordenadores necesitan de otros elementos que, aun no siendo "esenciales" desde el punto de vista lógico, en la práctica convierten un simple amasijo de circuitos en una herramienta muy potente. Algunos de estos elementos son:

- La circuitería de control: Es la parte hardware del procesador que se encarga de "controlar" el procesador, produciendo señales de control a los diferentes componentes a partir de los códigos binarios de la memoria.
- Dispositivos de salida: Impresoras, monitores, altavoces, etc, junto con las tarjetas y puertos que los conectan al procesador.
- Dispositivos de entrada: Como un teclado, un micrófono o una tarjeta de red.
- Alimentación: Sistema de distribución de energía eléctrica.
- Refrigeración: Para determinadas zonas que alcanzan altas temperaturas.
- Sistema de arranque: Es el conjunto de elementos eléctricos y lógicos que hace posible el arranque del ordenador (interruptor de encendido, circuitería y ROM de la BIOS)

8.2.3: Juego de órdenes

Las órdenes

Los programas que ejecuta una máquina son secuencias de órdenes grabadas en una zona de la memoria. Físicamente son secuencias de ceros y unos; lógicamente codifican una serie de acciones a realizar. Otros nombres que se les dan a las órdenes son "instrucciones" o "proposiciones".

Es muy infrecuente escribir directamente el código binario de un programa en un libro. En lugar de ello se utiliza un pseudo lenguaje que es una traducción directa de las órdenes contenidas en la memoria, pero que es más inteligible para los humanos. Es el lenguaje ensamblador[73]. Los elementos principales del ensamblador son los mnemónicos, que identifican qué es lo que hacen las diferentes órdenes.

Las órdenes pueden clasificarse en diversos grupos según su funcionalidad:

- órdenes de movimiento de datos
- órdenes de cálculos aritméticos
- órdenes de salto incondicional
- ordenes de salto condicional
- otras

Las iremos viéndolas una a una en los siguientes subapartados.

Órdenes de movimiento y modos de direccionamiento

Tienen la forma

```
MOV fuente, destino
```

Donde

73 Ya hablamos del ensamblador en la sección 1.2.2 y posteriores.

- `fuente` es una dirección de memoria, un registro o un valor.
- `destino` es una dirección de memoria o un registro.

Los valores normalmente se especifican simplemente con un número. Los registros se referencian con su nombre. Así, la orden "insertar el valor 1 en el registro A" tendrá la forma:

```
MOV 1, A
```

Las direcciones de memoria son un poco más complicadas de referenciar, ya que pueden hay varios "modos de referencia". Se les llama *modos de direccionamiento*. La sintaxis concreta para cada modo varía según el ensamblador que se utilice; nosotros podremos identificar un acceso a memoria gracias a los corchetes ([]).

El modo más sencillo es el modo *absoluto*, consistente en utilizar directamente un número. Así:

```
MOV 1, [1024]
```

Escribe el valor 1 en la posición de memoria 1024. Normalmente las posiciones de memoria son bytes: la 1024 es 1 byte, la 1025 es el siguiente etc.

Otro modo de direccionamiento es el modo *registro*. Esta vez la dirección de memoria es la que se encuentra en el registro. Por ejemplo:

```
MOV 255, A  # guarda el valor 255 en el registro A
MOV 13, [A] # guarda el valor 13 en la dirección contenida en A
```

Estas dos instrucciones guardan el valor 13 en la dirección de memoria 255, que es direccionada a través del registro A. Nótese la sintaxis utilizada para definir los comentarios.

Un modo derivado del modo registro es el modo *registro indizado* o *registro con desplazamiento*. Este modo incluye una constante que debe añadirse al valor del registro para calcular la dirección de memoria. Por ejemplo:

```
MOV 255, A
MOV 0, 6[A] # la dirección modificada es 2050+6 = 2056
```

El siguiente modo de direccionamiento es un poco más complicado. Se llama modo *indirecto*. Este modo utiliza un registro para apuntar a una dirección de memoria *que a su vez apunta a otra dirección de memoria*. Si estuviéramos hablando del lenguaje C diríamos que en el registro se guarda la dirección de memoria donde está guardado un puntero.

Para denotar el modo indirecto utilizaremos el asterisco (*), de la misma forma que lo utilizamos en C. Un ejemplo de uso del modo indirecto es el siguiente:

```
MOV 150 , [252] # guarda el valor 150 en la línea 252
MOV 252 , A     # guarda el valor 252 en el registro A
MOV *[A], B     # guarda el valor 150 en el registro B
```

Por último, está el modo *indizado indirecto*, o *indirecto con desplazamiento*. Es similar al anterior pero añadiendo una constante:

```
MOV 150 , [252] # guarda el valor 150 en la línea 252
MOV 250 , A     # guarda el valor 250 en el registro A
MOV *2[A], B    # guarda el valor 150 en el registro B
```

Órdenes de cálculo aritmético

Las operaciones aritméticas tienen una forma parecida a la de movimiento:

```
op fuente, destino
```

Donde

- `fuente` es una dirección de memoria, un registro o un valor.
- `destino` es una dirección de memoria o un registro.
- `op` es un mnemónico del siguiente conjunto: {ADD, SUB, MUL, DIV}. Los mnemónicos representan respectivamente la suma, resta, multiplicación y división.

```
ADD A, B # suma el contenido de los registros A y B y lo guarda en B
SUB 32,[120] # resta 32 al contenido de la línea 120
MUL *[B], [340] # multiplica el contenido de *[B] y [340] y lo guarda
                # en la línea 340
DIV 2, B # divide el contenido del registro B por dos
```

Hay algunos casos especiales en los cuales las operaciones aritméticas no devuelven los resultados que deberían. Concretamente:

- ADD y MUL pueden provocar *overflow*, es decir, que pueden producir un resultado demasiado grande para ser guardado en fuente.
- DIV puede provocar un error de división por cero.

Normalmente el comportamiento del computador ante estas situaciones puede "programarse". Aquí supondremos que simplemente se limita a "apuntar" en un "registro de banderas" (ver más adelante) si ha ocurrido algún problema.

Órdenes de salto incondicional

Para poder utilizar las instrucciones de salto incondicional exigen que al menos una de las órdenes del programa tenga una *etiqueta*. Una etiqueta es una cadena de texto que identifica a orden. Se sitúa delante de ella, separada con los dos puntos (:). No puede haber dos etiquetas iguales en un mismo programa:

```
etiqueta1: MOV 0, A # Esta orden tiene la etiqueta "etiqueta1"
```

La instrucción de salto condicional tiene la siguiente forma:

```
JMP etiqueta
```

Donde `etiqueta` es una etiqueta válida del programa.

Una instrucción de salto condicional modifica el flujo natural del programa (ejecutar una orden detrás de otra) permitiendo "saltar" a una orden cualquiera del programa. Una vez ejecutada la orden, el flujo del programa continúa con la siguiente orden.

```
ZeroA: MOV 0,A
JMP ZeroA
```

El código anterior es un "bucle infinito"que nunca deja de poner a cero el registro A.

Saltos condicionales

Los saltos condicionales se realizan si se cumple cierta condición lógica, que normalmente implica una comparación numérica.

La orden JE (*Jump if Equal*) sirve para ver si dos valores son iguales:

```
JE A,B,etiqueta
MOV 0,B
etiqueta: MOV 0,A
```

En el código anterior, si A es igual a B, A se hace 0. En otro caso, tanto A como B se hacen 0.

JNE (*Jump if Not Equal*) es la instrucción contraria a JE: salta si y solo si los valores son *distintos*. De esta manera, la versión del código anterior con JNE sería así:

```
JNE A,B,etiqueta
MOV 0,A
etiqueta: MOV 0,B
```

(Nótese que la segunda y tercera orden se han intercambiado).

JG y JGE sirven para ver si un valor es "mayor" o "mayor o igual" que otro, saltando si es así. Sus mnemónicos significan respectivamente *Jump if Greater* y *Jump if Greater or Equal*.

Por ejemplo:

```
JG  A, 0, et1 # salta a et1 si A>0
JGE A, 0, et1 # ídem, pero salta si A>=0
```

Similares a JG y JGE son las órdenes JL y JLE (*Jump if Lesser* y *Jump if Lesser or Equal*), que saltan si un vaor es "menor" o "menor o igual" que otro dado:

```
JL  A, 10, et1 # salta a et1 si A<10
JLE A, 10, et1 # ídem, pero salta si A <=10
```

Otras

Existen muchísimas otras instrucciones en los procesadores actuales; enunciaremos solamente algunas.

Están por ejemplo las instrucciones de cálculos sobre valores en coma flotante, que utilizan registros especiales. Normalmente se escriben precediendo de una "f"a sus equivalentes enteros. Así, FMOV sirve para mover valores flotantes, FADD para sumarlos, etc.

También están las instrucciones de procesamiento en paralelo, como el juego de instrucciones MMX que AMD e Intel incluyen en sus procesadores desde hace bastantes años. Aunque estas órdenes tienen muchas aplicaciones, son utilizadas mayoritariamente para el procesamiento gráfico.

El tamaño importa

Hay un detalle que quizás ha pasado un poco velado a lo largo de este apartado: el tamaño (en bits) de los operadores de cada orden.

Hemos mencionado que los registros de una máquina x86 pueden ser accedidos de diferentes maneras; por ejemplo, el registro de 32 bits EAX puede accederse como "A" (y se obtiene el byte menos significativo) o como EA (16 bits), además de como EAX.

En otras palabras: las máginas x86 pueden manejar datos de diferentes tamaños.

Es necesario por lo tanto especificar una política de manejo de datos de diferentes tamaños. Dicha política debe dar respuesta a preguntas tales como "¿qué pasa cuando se suman un número de 16 bits y uno de 8? ¿Está permitido que el destino tenga 8 bits en dicho caso?" o "¿Cómo se inserta un registro de 32 bits en la zona de memoria 1024? ¿Solamente se pueden insertar en las zonas de memoria múltiplos de 4 o se pueden insertar en cualquier?".

8.2.4: Las *otras* máquinas

Hasta ahora hemos nos hemos referido a "la máquina" como una arquitectura profundamente estandarizada: memoria, cálculo, registros, buses. Da la impresión de que aunque la cantidad, capacidad y velocidad de dichos componentes varíe, las computadoras son esos cuatro componentes unidos de una u otra forma. Y por lo tanto su juego de instrucciones es un calco del hardware que hay por debajo.

Pues bien, esto no es cierto; desde hace tiempo se conocen otros modos de ejecutar código. Nótese la definición de "máquina" de dábamos al principio de la sección: "una máquina es cualquier mecanismo que sea capaz de ejecutar las acciones codificadas en un código". En ningún momento se hace referencia a "unidades de memoria" o "registros" en dicha definición. Así, es posible implementar una máquina sin registros, que utilice directamente la memoria, como la máquina virtual que el equipo Peroxide (**http://www.peroxide.dk**) utiliza para el lenguaje PDXScript[74]. Esta máquina utiliza solamente la pila, de manera que para sumar dos números primero hay que apilarlos. Después, una orden especial se encarga de desapilar los dos números de la cima de la pila, sumarlos y guardar el resultado. Así, para sumar 1 y 2 hay que generar lo siguiente:

```
PUSH 1 # Estado de la pila: (1)
PUSH 2 # Estado de la pila: (1,2)
ADD    # Estado de la pila: (3)
```

De hecho, si examinamos nuestra definición de máquina, veremos que no se hace referencia a *ningún componente hardware.* Solamente se menciona la palabra "mecanismo". Es cierto que la acepción habitual de la palabra es "ingenio mecánico hecho de una o varias piezas móviles". Sin embargo no es el único: un programa software es también un mecanismo.

De hecho, una máquinas muy famosa y usada en el mundo no tiene un solo circuito, tuerca o soldadura: es un programa software, y se llama la Máquina Virtual de Java (o *Java Virtual Machine,* JVM). Es "virtual" precisamente porque no existe físicamente: es un programa de ordenador. Eso sí, necesita de un sistema operativo sobre el que ejecutarse[75].

Existen versiones de la JVM para muchos sistemas operativos; también hay una versión de la JVM que se ejecuta sobre Microsoft Windows. Es el ejecutable JAVA.EXE que se utiliza para ejecutar las aplicaciones java (y el propio ANTLR).

Es fácil ver dónde están las "acciones" y "el código" de JVM. Las órdenes para la JVM están "codificadas" dentro de los ficheros *.class, que los "lee" y realiza las "acciones" codificadas en ellos.

Una ventaja que tiene la JVM sobre las máquinas tradicionales es que, al carecer de un soporte físico real, el juego de órdenes de las que dispone no está limitado por dicho soporte. Así, el lenguaje de los ficheros *.class es una especie de "lenguaje java en binario"; por ejemplo, admite nativamente *cualquier tipo declarado en java*, no solamente "enteros" y "flotantes" como hacen las mejores máquinas clásicas actualmente[76]. Es decir, que los tipos no "se pierten" al generar el código.

La principal desventaja de la JVM es la eficiencia: la máquina virtual "ocupa" parte del tiempo de proceso, de manera que los programas java son casi siempre algo más lentos que sus equivalentes

74 La JVM tampoco utiliza registros; solamente la pila.

75 Actualmente hay diversos proyectos que hacen que esto ya no sea tan necesario. Por ejemplo, LEO es un intento de hacer un sistema operativo en Java (de manera que los binarios del sistema sean los propios ficheros *.class). Hay otras alternativas para soportar Java a nivel de hardware.

76 El tipo boolean es emulado con un tipo entero valiendo 0 o 1, según la documentación de la jdk 1.1.4.

compilados para la máquina "real" sobre la que se ejecutan.

Otro ejemplo de máquina virtual es la plataforma .NET de Microsoft. El lenguaje en este caso se llama CLR (*Common Languaje Runtime*) y tiene la ventaja de que puede generarse desde compiladores para varios lenguajes. Su principal defecto es que por el momento solamente hay versiones de .NET para algunas versiones de Windows[77].

.NET soluciona el problema de la eficiencia de Java gracias a la compilación JIT (*Just In Time*) que transforma los ficheros CLR en ficheros ejecutables directamente sobre windows.

Ante esta perspectiva es muy difícil definir por completo qué es una máquina. Nos quedaremos con la mejor definición que le hemos dado: una máquina es "cualquier mecanismo capaz de ejecutar los algoritmos de un programa fuente a partir del producto de una compilación satisfactoria sobre dicho programa".

Que los valientes intenten definir "mecanismo".

77 Defecto que el proyecto MONO, de la empresa Ximian, podría solucionar en un futuro cercano.

Sección 8.3: Gestión de la memoria

8.3.1: Definición

Los programas que generemos deberán guardar sus datos en la memoria de la máquina objetivo. Por tanto, gestionar dicha memoria es una tarea capital.

Implementar la gestión de la memoria variará de dificultad dependiendo de la máquina objetivo. Así, en las máquinas virtuales de última generación, como la JVM, la gestión es sencilla: cuenta con órdenes tales como `new`, que permiten aislar al generador de código de los detalles de la gestión.

Si en cambio estamos generando código para una máquina más "clásica", encontraremos algunas dificultades. En esta sección sugeriremos cómo superar algunos de estos problemas.

8.3.2: División clásica de la memoria

Existe una manera estandarizada de dividir la memoria en zonas, dependiendo de la función que realize cada zona. Las zonas son código, datos estáticos, pila y montículo, además de una zona vacía "tierra de nadie".

La siguiente ilustración aparece en "Compiladores. Principios, técnicas y herramientas" (ed. Pearson) de Aho, Sethi y Ullman, en la sección 7.2 "Organización de la memoria"

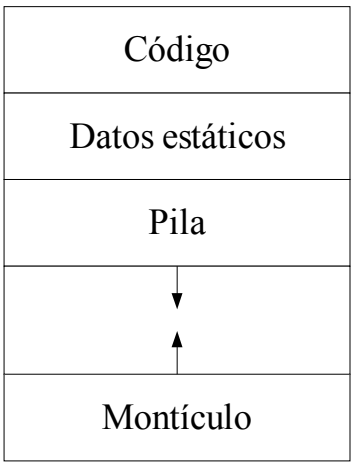

Ilustración 8.3 División clásica de la memoria

*****

La zona de código

Es la zona donde se guardan las secuencias de órdenes que constituyen el programa a ejecutar. Por lo general se trata de una zona "tranquila", en el sentido de que sus datos no suelen ser modificados durante toda la ejecución del programa[78].

Ya hemos mencionado cómo se codifican las órdenes en la memoria de una máquina clásica. Normalmente los bits más significativos codifican el tipo de la instrucción (MOV, ADD, JMP...), y los siguientes son utilizados para guardar los parámetros.

78 La excepción son los programas con código automodificado. Bryon Hapgood tiene un artículo muy bueno sobre esto en "*Game Programming Gems 2*" (ed. Charles River Media), sección 1.14.

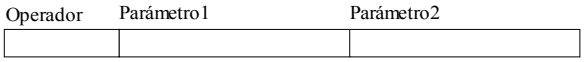

Ilustración 8.4 Típica línea de código

El esquema de la figura anterior sirve para palabras de cualquier tamaño; simplemente indica que los primeros bits representan el tipo de orden y el resto los parámetros.

Es importante tener en cuenta que incluso en una misma máquina el número de bits dedicado a codificar el operador es variable; por ejemplo, en el Motorola 68000 un salto ocupaba unos pocos bits, y el resto era utilizado para la línea a la que saltar. Otras órdenes utilizaban más bits para codificarse, y por lo tanto tenían menos espacio para codificar los parámetros.

Por supuesto, lo mismo que ocurre con los operadores ocurre con los parámetros: no todos ocupan lo mismo.

En ocasiones los parámetros de una orden ocupan tanto espacio que la orden ocupa varias líneas de memoria:

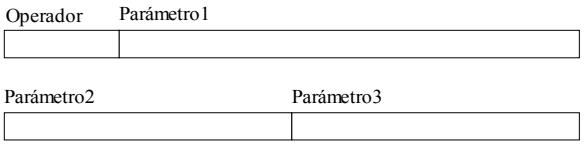

Ilustración 8.5 Orden en varias líneas de memoria

Por supuesto esto no es deseable en términos de eficiencia; pero en algunas ocasiones simplemente no hay otra solución.

Zona de datos estáticos

Esta zona, como su propio nombre indica, tampoco cambiará mucho durante el curso del programa. Es la zona dedicada a guardar datos "inmutables pero necesarios" para la ejecución del programa. Algunos ejemplos de información que se puede guardar en esta zona:

- Constantes del programa (si no han sido sustituidas en el análisis semántico)
- Información sobre el sistema de tipos: clases, subclases, etc, necesaria para poder realizar *conversiones de tipo hacia abajo* (convertir una clase a una subclase) de forma segura.
- Tipos y funciones básicos del sistema.
- Cadenas de texto que aparecen se utilicen en el código (como el "Hola mundo").
- ...

La pila

La pila es una de las zonas "variables" de la memoria. Es allí donde debemos alojar los objetos que se instancien durante el programa; los objetos se crean y destruyen continuamente, y por lo tanto la pila está continuamente cambiando.

Estudiaremos el funcionamiento de la pila con más detenimiento en el siguiente apartado.

El montículo

Es la otra gran zona "variable" de la memoria. En principio tiene la misma utilidad que la pila: alojar las instancias de los objetos durante la ejecución de un programa. Más información en el

siguiente apartado.

8.3.3: Acerca de la pila

Orígenes de la pila

En un principio los lenguajes de programación muy simples y de muy bajo nivel (muy cercanos a la máquina objeto). Eran tam simples que solamente había una función por programa; principalmente se utilizanban en el ámbito universitario para realizar cálculos matemáticos. Se parecían a esto:

```
5   dim a, b, c, aux, x1, x2 as Float
10  input a
20  input b
30  input c
40  if a=0 then goto 200
60  let aux = sqrt(b*b - 4*a*c)
70  let x1 = (-b + aux)/2*a
80  let x2 = (-b - aux)/2*a
90  print "solutions: "; x1 ; "and" ; x2
100 exit
200 print "No solution"
210 exit
```

Este código en pseudo-BASIC[79] calcula la solución de una ecuación de segundo grado. No es muy complicado de manejar a nivel de memoria: todas las variables debían ser declaradas al principio del programa, y eran simplemente alocadas en la zona de datos estáticos.

El problema surgió con los primeros paradigmas funcionales, o "subrutinas":

```
5   dim a, b, c, aux, x1, x2 as Float
10  input a
20  input b
30  input c
40  gosub 100
50  print "solutions: "; x1 ; "and" ; x2
60  exit
100 rem ---- comienza la subrutina de cálculo ------
110 if a=0 then goto 150
120 let aux = sqrt(b*b - 4*a*c)
130 let x1 = (-b + aux)/2*a
140 let x2 = (-b - aux)/2*a
150 return
200 print "No solution"
210 exit
```

Las primeras subrutinas eran meros pedazos de código a los que no se les pasaba parámetro alguno; accedían a variables globales por su nombre. Dentro de sus limitaciones, las subrutinas tenían su utilidad: comenzaron a permitir la reutilización de código.

La instrucción `gosub` era una evolución de `goto`(una instrucción de salto incondicional normal). Cuando se utilizaba la instrucción `gosub xxx`, el flujo del programa cambiaba haciendo xxx la siguiente línea a ejecutar de igual manera que lo haría con `goto xxx`, *con la salvedad de que al llegar a la instrucción return el flujo del programa debía recobrarse donde se dejó*(la línea siguiente a a quella que invocó el `gosub`). Estaba claro que era necesario "guardar" la instrucción

79 Nótense los números de línea; las primeras versiones de BASIC no eran suficientemente potentes para manejar etiquetas. Se numeraban de 10 en 10 por si hacía falta insertar código intermedio más adelante.

que ejecutaba la subrutina en alguna parte.

Pero podía complicarse más: dado que las subrutinas podían "anidarse" (invocarse las unas a las otras) el programa podía presentar n `gosub` anidados – luego tendría que recordar n números de línea diferentes.

Entonces es cuando nace la pila del programa: la pila inicial era una pequeña sección de memoria en la que el procesador "apilaba" el contador del programa en cada `gosub`, y lo "desapilaba" en cada `return`. Su nombre viene de ese comportamiento tan típicamente FILO[80].

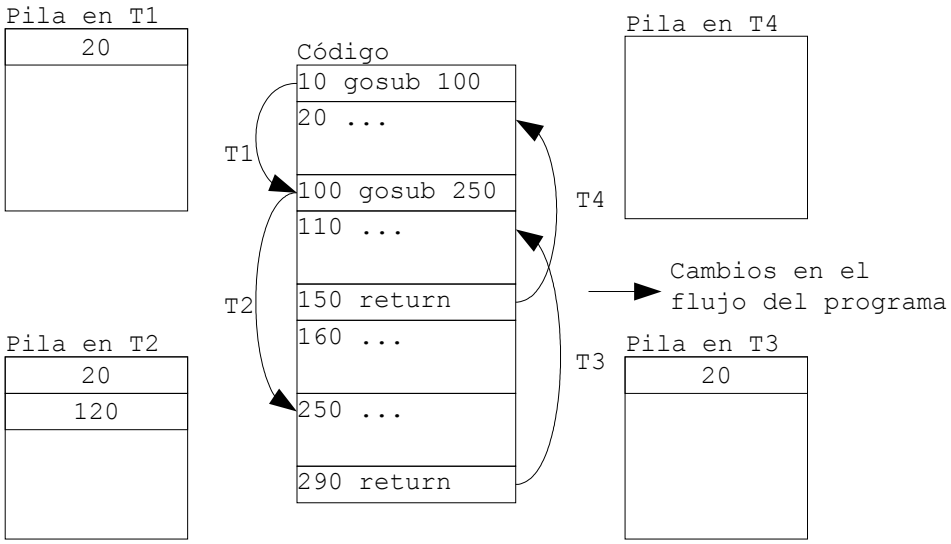

Ilustración 8.6 Comportamiento de la pila con gosub

Téngase presente que a pesar del uso de la pila, ésta sigue siendo un pedazo de memoria, y por lo tanto sus elementos accesible aleatoriamente – para acceder al tercer valor de la pila no hace falta desapilar los dos primeros. En esto se diferencia del tipo abstracto de datos "Pila", que se enseña a los estudiantes de los primeros cursos de informática.

Invocando funciones

El siguiente paso en la evolución de los lenguajes fueron las "verdaderas funciones". Éstas aceptaban parámetros, y además permitían la declaración de variables locales. Las funciones también pueden "anidarse", como las subrutinas. También hay "funciones recursivas", es decir, funciones que se llaman a sí mismas. Además, los lenguajes ya son de más alto nivel: la máquina objeto se va haciendo más invisible. Ahí tenemos al lenguaje C:

```
long factorial(long n)
{
   if(n<=1) return 1;
   else return(n*factorial(n-1));
}

int main (int argc, char*[] argv )
{
   int n;
   long l = 0;
   sscanf(argv[1], "%d", &n);
```

80 *"First In, Last Out"* o "Lo primero que entra es lo último que sale". Una pila es la estructura de datos FILO por excelencia.

```
    l = factorial(n);
    printf("El factorial de %d es %l", n, l );
}
```

El problema, especialmente con las funciones recursivas, es que hay que hacer más cosas a la hora de invocar una función que las que había que hacer al invocar una subrutina:

- Hay que pasarle parámetros, en caso de que los tenga; los parámetros de una función se "apilan".
- Hay que guardar las variables locales de la función. Para eso también se utilizará la pila.
- Hay que guardar valores temporales que se utilicen en las expresiones. Se reservará espacio para ellos en la pila.
- Hay que guardar en la pila el estado de los registros de la máquina: al poder invocarse una función desde varios sitios, la única manera de utilizarlos de forma segura en una función es "apilándolos y desapilándolos".
- Por supuesto, también hay que guardar la dirección de retorno.

El hardware de la pila

En los inicios del paradigma funcional, la tarea de gestión de la pila era más complicada de lo deseable: los programas debían implementar sus propias pilas a partir de la memoria "en bruto", realizando las apilaciones y desapilaciones con ayuda de algún registro de la máquina (que apuntaba siempre a la cima de la pila).

Este comportamiento llegó a ser tan habitual – y tedioso – que los fabricantes de hardware decidieron implementar capacidades de manejo de la pila a nivel de hardware, a saber:

- División automatizada de la memoria, incluyendo la creación de una pila.
- Un registro especializado en apuntar a la cima de la pila, y otro dedicado a apuntar a su "base temporal" (allí donde empiezan los datos interesantes para el método actual).
- Órdenes especiales de manejo de la pila, principalmente PUSH, y POP, y otras más secundarias, como CALL, LEAVE y RET (ver más abajo).

El registro que apunta a la cima de la pila en las máquinas x86 se llama ESP (*Extended Stack Pointer*). El registro de base de la pila en las máquinas x86 es el EBP (*Extended Base Pointer*).

A partir de la aparición del paradigma funcional, la pila gana relevancia con respecto a la zona de datos estáticos: se vuelve la protagonista absoluta de la generación de código. Nótese que incluso el "hilo principal del programa" (la función `main`) hace uso de la pila (para recibir los parámetros de la línea de comandos y para declarar las variables locales `n` y `l`). De hecho la zona de datos estáticos puede implementarse con la pila.

Invocando funciones en C

El proceso que se muestra a continuación fue obtenido del documento "C function call conventions", UMBC CMSC 313, disponible gratuitamente para usos no lucrativos en **http://www.csee.umbc.edu/~chang/cs313.f02/stack.shtml** *Mis agradecimientos al profesor Richard Chang.*

Supongamos que estamos ejecutando el código generado del cuerpo de una función f1 y desde dicha función se invoca otra función f2. El estado inicial de la pila y los registros será el que se muestra en la siguiente figura:

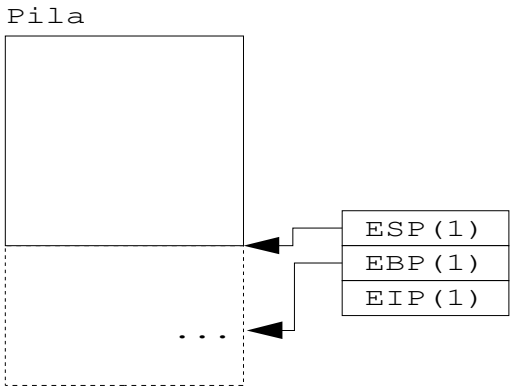

Ilustración 8.7 Estado inicial de la máquina

Los pasos para invocar una función f2 a partir de una función f1 en el lenguaje C son aproximadamente los siguientes:

- Apilamos el estado de la máquina en la pila, es decir, los registros[81].
- Apilamos los parámetros de la función.
- Apilamos el valor del contador del programa (EIP) en la pila, de manera que la dirección de retorno está en la cima de la pila, y luego lo cambiamos. En lugar de hacerlo con dos órdenes, (PUSH + GOTO), lo que haremos será utilizar la orden `CALL f2`, que realiza las dos acciones.

En este momento el estado de la máquina es el siguiente:

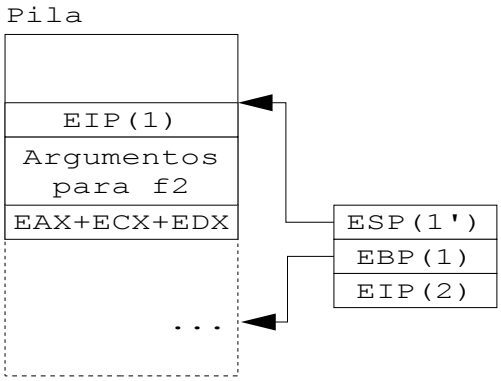

Ilustración 8.8 Va a empezar a ejecutarse f2

Llegados a este punto el flujo de código pasa a ejecutar el código de f2. Ésta debe realizar unos cuantos "preparativos" antes de poder empezar a utilizar la pila.

- Lo primero que debe hacer es preparar su propia pila. EBP está apuntando a un lugar en la pila de f1, y éste valor debe conservarse. Así que el valor de EBP se apila.
- Después se transfiere el valor de ESP a EBP. Esto permite referenciar los argumentos con un offset a partir de EBP y libera ESP para hacer otras cosas. Así, la mayoría de las funciones de C comienzan con las órdenes:

81 Una optimización muy común es solamente apilar los registros que estén siendo usados por el método.

```
PUSH EBP
MOV ESP, EBP
```

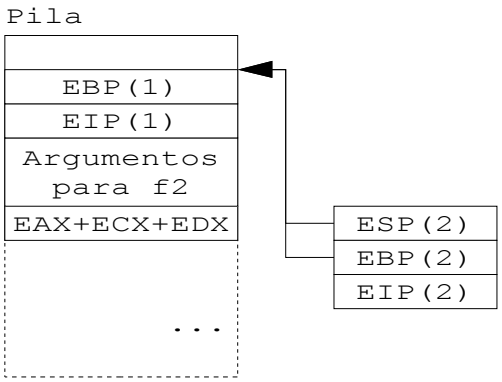

Ilustración 8.9 Primeros preparativos de f2

En el siguiente paso, f2 debe alojar espacio en la pila para sus variables locales y valores temporales de las expresiones que utilice. Suponiendo que en total sumen 20 bytes, entonces habrá que restar a ESP 20:

```
SUB 20, ESP
```

Ahora tanto las variables locales como los valores temporales pueden ser referenciados a partir de EBP.

Finalmente, f2 debe preservar los valores de otros registros de la máquina (EBX, ESI y EDI) en caso de que los utilice, apilándolos. La pila quedará como se muestra en la figura 8.10.

Llegados a éste punto, el código de f2 puede ejecutarse. Esto puede implicar apilar y desapilar valores en la pila, cambiando ESP. Sin embargo EBP permanece fijo. Esto es conveniente porque significa que siempre podemos referirnos al primer argumento de la función como EBP+8.

La ejecución de la función f2 podría involucrar la invocación de otras funciones, e incluso invocaciones recursivas de f2. Sin embargo, mientras EBP sea restaurado al volver de dichas invocaciones, los parámetros y variables de la función siguen pudiendo referenciarse como desplazamientos sobre EBP.

Cuando el código de f2 se termina, debe realizar una pequeña “limpieza”:

- Lo primero que debe hacer f2 es guardar el valor de retorno en el registro EAX. Las funciones que necesitan de más de 4 bytes para un valor de retorno se "transforman" en una función con un parámetro extra de tipo "puntero" y sin valor de retorno.
- Después, si f2 ha modificado los valores de EBX, ESI o EDI, deberá restaurarlos con los valores contenidos en la pila.
- Por último, no necesitamos las variables locales o temporales, luego restauramos EBP y ESP a los valores que tenían en f1. Es decir, ejecutamos:

```
MOV EBP, ESP
POP EBP
```

A partir del i386 hay dos órdenes equivalentes:

```
LEAVE
RET
```

Llegados a este punto, el aspecto de la pila es el mismo que mostrábamos en la ilustración 8.8.

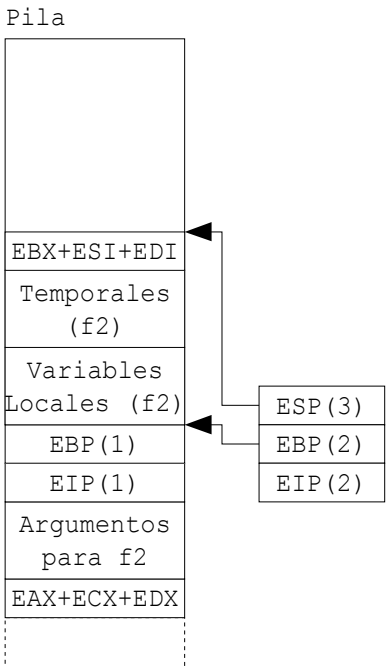

Ilustración 8.10 Listo para ejecutar f2

En este momento el control vuelve a f1, que a su vez deberá realizar algunas acciones de restauración. Concretamente deberá "desapilar" los parámetros que le pasó a f2 de la pila. Para ello, deberá añadirle a ESP el número de bytes ocupados por dichos parámetros. Por ejemplo, si los parámetros de f2 ocupan 12 bytes, entonces la instrucción que viene después de `CALL f2` en f1 es:

```
ADD 12, ESP
```

Después f1 debería guardar el valor de retorno de f2 (contenido en EAX o en algún lugar de la pila) en la dirección apropiada, seguramente haciendo un MOV.

Finalmente, f1 puede desapilar los valores de EAX, ECX y EDX de la pila, dejando la pila y los registros tal y como estaban antes de empezar todo el proceso de invocación (es decir, como en la figura 8.7).

*****

Invocando *métodos*

Por último llegó la orientación a objetos. Esta nueva evolución trajo consigo nuevos cambios en la distribución de la memoria, y no solamente en la pila: las zonas de *código y datos estáticos* también sufrieron cambios de otra manera.

La herencia y el polimorfismo hacen que la tarea de conocer qué método estamos invocando en cada caso no sea trivial. Imagínese el lector el siguiente escenario:

```
clase A
{
   método m() { Sistema.imprime("A:m"); }
}

clase B extiende A
{
```

```
   método m() { Sistema.imprime("B:m"); }
}

clase Inicio
{
   método abstracto inicio()
   {
      B b;
      convertir(A,b).m();
   }
}
```

En este caso, a pesar de que la variable `b` es temporalmente convertida al tipo `A`, la salida del programa es "`B:m`", ya que verdaderamente se trata de una instancia de `B`.

El secreto de esta implementación está en las tablas virtuales, o *vtables*, que son tablas en las que el compilador apunta las direcciones donde empieza la implementación de los métodos "reales" de cada tipo del lenguaje. Así, supongamos que el método A:m se empieza a generar en la línea 1111 de la zona de código 2222. Las *vtables* de A y B serán diferentes:

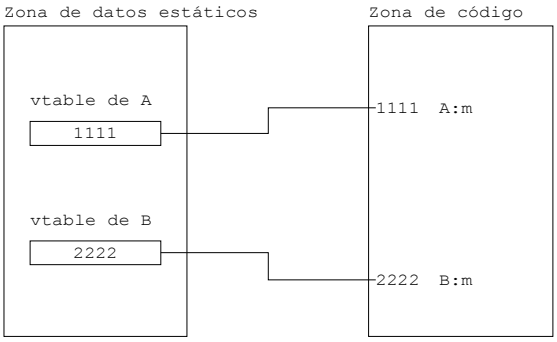

Ilustración 8.11 Las vtables

Esto hace que para invocar un método sea necesario conocer la vtable del objeto invocador. De manera que las *vtables* de todas las variables locales y parámetros del código deben incluirse en la pila.

Pero nos olvidamos de un elemento importante: los atributos.

Cuando una instancia con atributos se declara, sus atributos son guardados en la pila, uno tras otro, como si se tratara de variables locales independientes.

Es muy frecuente modificar los atributos de una clase dentro de sus métodos:

```
clase Persona
{
   atributo Cadena nombre;

   constructor (Cadena n)
   { cambiaNombre(n); }

   método cambiaNombre(Cadena n)
   { nombre = n; }
}

clase Inicio
{
   método inicio()
```

```
   {
      Persona J("Juan"), P("Pedro"), M("María");
      P.cambiaNombre("Paco");
   }
}
```

En el código anterior hay diversas instancias de la clase `Persona`, por lo que en la pila habrá una serie de "atributos nombre" apilados.

El método `cambiaNombre` es invocado 4 veces (3 en los constructores y 1 explícitamente) desde diferentes instancias de `Persona`, así que debe cambiar cadenas que están en diferentes direcciones de la pila.

¿Cómo implementar este comportamiento?

Es evidente: necesitamos "pasar" un parámetro más al método `cambiaNombre`, a partir del cual el método pueda decidir dónde está la cadena de texto a cambiar. En este caso, vamos a pasarle la dirección en la que empieza la instancia de Persona. En otras palabras, vamos a comportarnos como si el método `cambiaNombre` tuviera un parámetro extra:

```
método cambiaNombre(Persona p, Cadena n)
{
   p.nombre = n;
}
```

Por otro lado cuando invoquemos el método `cambiaNombre`:

```
P.cambiaNombre("Paco");
```

Se añadirá automáticamente el objeto llamador a la pila. Será como si hubiéramos escrito lo siguiente:

```
cambiaNombre(P, "Paco");
```

Todo esto deberá realizarse de forma transparente para el programador, que se limitará a escribir e invocar `cambiaNombre` normalmente.

Así que tendremos la instancia del objeto llamador en la primera posición de la pila cuando invoquemos métodos de clases. Este comportamiento es muy común en los lenguajes orientados a objetos que generan código para máquinas clásicas.

¿Cómo se conecta todo esto con las *vtables*? Muy sencillo: toda instancia de una clase debe "guardar un puntero" a la zona de memoria de datos estáticos en la que se describe su tipo (y donde está disponible la *vtable* de la clase). Por ejemplo, si la descripción de la clase

Persona comenzase en la línea 3333 de la zona de datos estáticos, las variables J, P y M en la pila tendrían el siguiente aspecto:

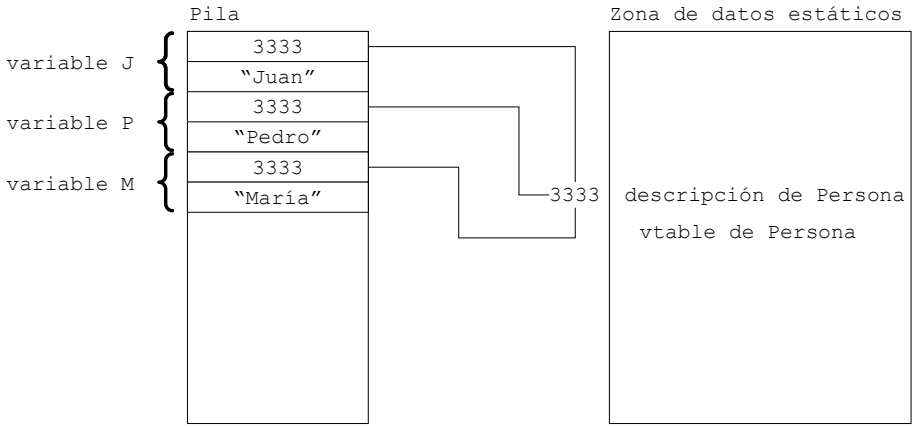

Ilustración 8.12 Instancias de una clase apuntando a su descriptor

Nótese que no hacemos que las instancias apunten directamente a la vtable, sino a una entidad que llamamos "descripción de la clase". Esta es una pequeña sección de memoria en la que se incluyen otras informaciones además de la *vtable*, como la descripción de la superclase.

El valor de retorno de una función o un método

Hasta ahora hemos evitado hablar de un aspecto importante sobre funciones y métodos: ¿cómo se manejan los valores de retorno con la pila?

Existen varias alternativas; para datos pequeños, es usual utilizar algún registro de la máquina, en aras de la eficiencia (dado que los registros son apilados y desapilados en las invocaciones, esto no supone peligro alguno).

Para datos más grandes podemos servirnos de la pila: puede utilizarse una sección de la pila tan grande como sea necesario para alojar el valor de retorno.

Más sobre el valor de retorno en la siguiente sección.

8.3.4: Acerca del motículo

Motivación del montículo

Hay un detalle importante que ya hemos mencionado sobre la pila, pero volveremos a hacerlo aquí a modo de recordatorio:


Los datos de utilizados en la pila son tremendamente perecederos.

Toda variable local de una función que sea alojada en la pila tiene sus días contados; cuando la función se termine, ¡pop! será desapilada.

Este comportamiento es deseable para la mayor parte de las variables. Sin embargo, hay otras ocasiones en las que se desea que las variables "sobrevivan" a la función que las creó. Por ejemplo:

- datos compartidos entre varios hilos de procesamiento
- cadenas de texto
- ...

En aplicaciones de tamaño medio o grande, es frecuente que haya varios hilos de procesamiento trabajando en paralelo sobre datos compartidos. Normalmente dichos datos son "creados" por

una función, y deben permanecer accesibles por todos los hilos mientras sean necesarios.

Más flagrante aún es el caso del manejo de cadenas de texto. Manejar cadenas de texto con la pila es muy engorroso; prueba de ello es que los estudiantes de C con frecuencia escriben programas como el siguiente:

```
#include <stdio.h>

char * obtenerNombre()
{
   char nombre[12];
   printf("Introduzca un nombre: ");
   scanf("%s", nombre);
   return nombre;
}

void main(void)
{
   char * nombre = obtenerNombre();
   printf("El nombre introducido es %s\n", nombre);
}
```

Este programa se limita a pedir la introducción de un nombre por la entrada estándard y luego lo muestra en pantalla. A pesar de lo simple que es, esconde un error importante, fruto del desconocimiento de la pila.

Cuando la función `obtenerNombre` es invocada, lo primero que hace es reservar un espacio de 12 caracteres *en la pila*. El lenguaje C implementa las cadenas de caracteres como tablas de caracteres, y las tablas con punteros, de tal suerte que la variable `nombre` es un puntero que apunta al inicio de dicho espacio.

La función `scanf` utiliza el puntero para "rellenar" el espacio dependiendo de lo que introduzca el usuario (suponiendo que no se introduzcan más de 11 caracteres por el teclado).

Finalmente, se copia el puntero en el registro EAX para devolverlo (en C el valor de retorno se guarda en EAX).

Pero justo antes de que se termine la ejecución de obtenerNombre, el espacio de la pila ocupado por sus variables locales se libera (se marca como "libre"), de tal manera que cualquier otra parte del código puede sobreescribir dicha parte "apilando y desapilando". Esto incluye los 12 caracteres del nombre.

Cuando el flujo del código vuelve a la función main, se guarda el valor de EAX en un nuevo puntero de la pila, que también hemos llamado nombre (nótese que se copia el *puntero*, no *los 12 caracteres que empiezan donde indica el puntero*).

Finalmente se invoca la función `printf`. El comportamiento del programa en este punto se vuelve dependiente del compilador; depende de la implementación concreta de `printf` que se emplee: el programa puede funcionar como se espera (mostrando el mensaje correctamente) si `printf` utiliza "poco" la pila. Lo más probable, sin embargo, es que el mensaje aparezca corrupto, porque los 12 caracteres del nombre se hayan sobreescrito. Aún en el caso de que printf funcione correctamente habrá un *bug* escondido: al añadir nuevas instrucciones (llamadas) al método `main` el nombre se corromperá.

Es necesario una zona de memoria en la que podamos guardar las variables "que sobrevivan a la función que las creó". De momento no entraremos en detalles sobre cuándo deben ser destruidas estas variables.

Esto es el montículo.

Implementación

El montículo es una zona de memoria como otra cualquiera. En la división de memoria que hicimos en la figura 8.3 (página 376) representábamos la pila y el montículo, compartiendo la memoria: mientras la pila crece "hacia abajo" el montículo lo hace "hacia arriba", de manera que se optimiza el espacio. Esta es solamente una de las opciones; en otras ocasiones la pila y el montículo son completamente disjuntos.

Dado el uso que queremos darle al montículo, éste no puede implementarse como una estructura FILO, como la pila.

El montículo se organiza en "pedazos" que llamaremos *segmentos*. En algunos casos estos segmentos son de tamaño fijo, de forma que cuando se tiene que alojar un dato en la memoria, bien ocupa parcialmente un segmento (si el dato es más pequeño que el segmento) o bien ocupa varios segmentos consecutivos (si ocurre al contrario).

Sin embargo, lo normal es que el montículo esté formado por segmentos de tamaño variable: inicialmente hay un solo segmento que ocupa todo el montículo, y conforme van llegando peticiones de alojamiento se va "particionando", generando segmentos más pequeños. Nos centraremos en esta alternativa, y dejaremos de lado los montículos de tamaño de segmento fijo.

Cuando llegan las primeras peticiones el comportamiento del montículo se parece al de la pila: da la impresión de que los segmentos se "apilan". Por ejemplo, en la siguiente figura mostramos el estado de un montículo tras realizarse 5 peticiones de alojamiento de datos de tamaño variable:

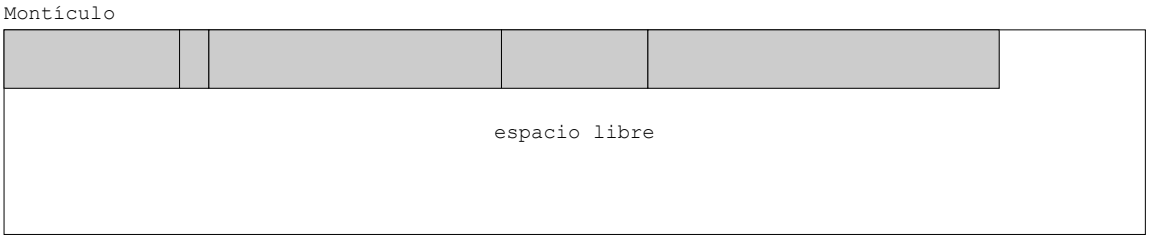

Ilustración 8.13 Montículo con 5 elementos alojados

Sin embargo, como ya hemos dicho, un montículo no es una estructura FILO. Puede que el segundo y cuarto dato dejen de ser necesarios antes que los demás, así que se declararán como "libres", dando lugar a lo que se denomina "fragmentación de la memoria":

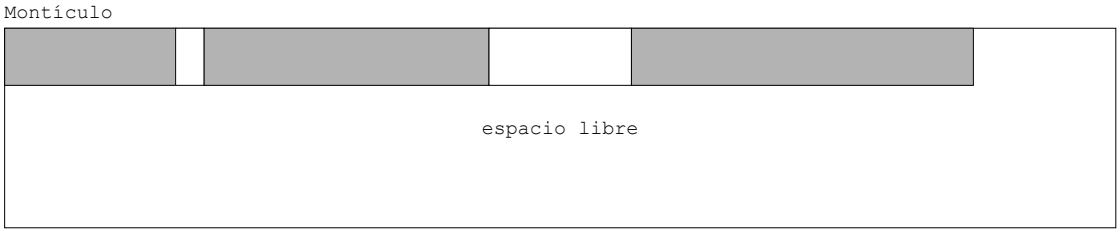

Ilustración 8.14 Montículo con segmentos libres y ocupados

La fragmentación supone un problema porque no se puede alojar en el montículo un bloque de datos mayor que el mayor de los segmentos libres, aunque en el montículo haya espacio para ello.

De todo lo anterior pueden deducirse dos consecuencias principales:

- Que es necesario llevar un control de los segmentos que están libres y de los que están

ocupados.

- Que puede ser necesario "compactar" los datos.

Controlar los segmentos libres y ocupados es sencillo: generalmente se utiliza una lista enlazada cuyos elementos apuntan al inicio y final de los segmentos libres, que suelen ser minoría.

La compactación es un poco más compleja; dado que nuestro lenguaje puede contener "punteros a memoria", la compactación haría que los punteros no apuntaran a las direcciones adecadas. Esto puede solucionarse utilizando una capa intermedia en las direcciones (en lugar de utilizar un "puntero a la dirección 2345 de la memoria" tendremos un "puntero al tercer segmento del montículo").

Liberación explícita de memoria: Modelo de C/C++

Ya sabemos cuando se "aloja" la memoria del montículo – cuando se demanda explícitamente.

Lo que no hemos visto todavía es cuándo *se libera* dicha memoria.

En realidad es sencillo: un objeto ha de ser borrado del montículo *cuando ya no haga falta*. ¿Y cuándo deja de hacer falta un objeto?

La respuesta de C y C++ es "cuando lo diga el programador". En C y C++ el programador está encargado de controlar el ciclo de vida de los objetos que crea en el montículo, y de ordenar su destrucción cuando ya no son necesarios.

Las herramientas que permiten la gestión del montículo en C son las funciones `malloc` y `free`.

```
char nombre [12] ; /* reserva 12 caracteres en la PILA */

/* Los caracteres son liberados al terminar el método donde se
   declararon */

// reserva 12 caracteres en el montículo
char * nombre = (char*)malloc(12*sizeof(char));

// Los caracteres no son liberados hasta que se invoca "free"()
free(nombre);
```

Nótese que `malloc` se limita a reservar un número determinado de bytes en el montículo, de manera que hay que calcular manualmente el tamaño a reservar, y hacer un *casting* al tipo de datos que estemos reservando. En el caso concreto del ejemplo hay código reiterativo; `12*sizeof(char)=12`, y además el casting no es necesario (`malloc` devuelve un `char*`). Pero lo he añadido por corrección; es casi obligatorio invocar `malloc` de esta forma.

Además de la necesidad de conversión y de cálculo manual del tamaño, la memoria devuelta por malloc no está "inicializada" de forma alguna; normalmente contiene basura, de manera que frecuentemente es necesario inicializarla a valores correctos antes de empezar a utilizarla.

Otro detalle importante es la liberación; cuando un objeto x contiene punteros a otros objetos del montículo, y éstos son únicos (no hay más punteros que apunten a estos datos), entonces si hacemos `free(x)` nos quedaremos con "basura" en el montículo: habrá segmentos marcados como "ocupados" que nunca se podrán liberar (porque hemos perdido los punteros para poder hacer `free`).

Nótese también que `malloc` y `free` no son prestaciones del lenguaje: son funciones de una librería estandar del lenguaje.

Todos estos inconvenientes se resolvieron en C++ con las palabras reservadas `new` y `delete`, y

su relación con los constructores y destructores de clases.

```
class Banana
{
   public:
      Banana();  // Constructor
      ~Banana(); // Destructor
}
...
Banana b; // Crea una instancia de Banana en la PILA
          // inicializándola con el constructor
// La instancia será destruída (llamando al destructor) al terminar el
// método actual
...
// Crea una instancia de Banana en el MONTÍCULO, y la inicializa
Banana b = new Banana();
...
// b no se destruye hasta que en algún método se invoca delete
delete b;
// delete invoca al destructor antes de desalojar el objeto de memoria
```

Con `new` y `delete` se resuelven casi todos los problemas de `malloc` y `free`:

- No es necesario hacer *castings*.
- La memoria queda automáticamente inicializada al llamar al constructor (si éste está bien escrito, por supuesto).
- La "liberación de otros objetos con punteros" puede ser resuelta al llamar al constructor (de nuevo si éste está bien escrito).
- Al ser una prestación del lenguaje, el tamaño a reservar se calcula automáticamente, incluso con las tablas (puede hacerse `new Banana [10]`).

Sin embargo hay un asunto que no se resuelve: el programador. Si el programador se olvida de invocar `delete` una vez por cada vez que hizo un `new`, o si no implementa los destructores adecuadamente, corremos el riesgo de tener basura en nuestro montículo.

Liberación implícita de memoria: Recolección de basura

Aunque la estrategia de C++ resuelve muchos de los problemas del montículo, sigue teniendo deficiencias, pues depende de la pericia del programador para determinar qué segmentos del montículo ya no son necesarios y por lo tanto pueden ser liberados.

Java resuelve el problema de la liberación de memoria del montículo de una forma diferente: en lugar de dejar el asunto en manos del programador, es el propio programa el que decide qué fragmentos de memoria ya no son necesarios y pueden eliminarse de manera segura. El criterio que la máquina virtual de Java sigue para averiguar si un objeto ya no está siendo usado por el sistema es el *conteo de referencias*.

Para empezar, en Java todos los objetos *son alojados en el montículo*. La pila también se utiliza, pero solamente para guardar punteros a las variables del montículo.

La JVM caracteriza cada segmento del montículo con un entero. Este entero, inicialmente a 1, es el *contador de referencias del objeto*. Cuando se añade un nuevo puntero al segmento en la pila se aumenta dicho contador. Cuando la referencia deja de ser válida (se desapila) se le resta una unidad al contador.

Cuando un segmento tiene su contador de referencias a 0 entonces ya no es accesible desde el

código. Se ha convertido en "basura", y ya no es necesario para la ejecución del programa, por lo que el segmento puede ser liberado[82].

Los segmentos no se liberan en cuanto su contador de referencias se hace 0, por motivos de eficiencia. En lugar de ello, cada cierto tiempo se activa el *recolector de basura.* El recolector de basura es un hilo que se encarga de recorrer el montículo liberando los segmentos "basura".

Esta estrategia soluciona los problemas de C++; en el momento en que la liberación del montículo ya no está en manos del programador, la degeneración del montículo debida a errores humanos es mucho más pequeña.

Sin embargo existe un pequeño inconveniente: el recolector de basuras consume tiempo de procesamiento. Esto provoca inquietudes en algunos programadores, que no desean sacrificar un porcentaje de la CPU ejecutando una tarea que podrían implementar ellos mismos con destructores, como en C++.

Otros programadores, por el contrario, defienden que la estrategia de Java es más eficiente que la de C++: mantienen que java "realiza todas las liberaciones de memoria de una vez, recorriendo una sola vez el montículo cada cierto tiempo", mientras que C++ "hace pequeñas liberaciones constantemente, obligando a recorrer el montículo muchas veces".

La conveniencia o no de utilizar un recolector de basura en un lenguaje es uno de esos puntos que dividen a los programadores de manera irreconciliable, como ocurre con las tabulaciones o la posición de la llave en java y C++.

Hardware del montículo

Por norma general no hay un software específico encargado de gestionar el montículo, al contrario de lo que ocurre con la pila. El principal motivo es que hay muchas estrategias de implementación, y cada una es adecuada para un uso.

Los sistemas operativos ofrecen métodos de alojamiento de memoria, que pueden ser utilizados por los programadores para crear montículos. Sin embargo, la liberación de la memoria es responsabilidad del programa, no del sistema operativo.

El montículo y LeLi

Los lenguajes necesitan montículos cuando sus nombres pueden apuntar a más de una dirección de memoria, es decir, cuanto tienen "punteros". LeLi, por su parte, carece de punteros o algo que se le parezca, por lo que creemos que podría implementarse con una pila, excepto para manejar las cadenas de texto.

8.3.5: Acerca del offset

Problema

La disposición de los objetos en la memoria, tanto en la pila como en el montículo, es en forma de "bloques", o "trozos" de memoria, en la que se guarda información sobre cada instancia. Estos bloques comienzan en una cierta dirección, y deben "relacionarse" con los accesos en el código: Cuando el programador escriba P.nombre el lenguaje tiene que saber que tiene que generar un acceso para la dirección 0xFFC1A.

¿Cómo se relaciona un nombre con su dirección?

82 Esta es una descripción muy simple del algoritmo. Hay casos excepcionales que deben ser tenidos en cuenta para implementar un algoritmo de recogida de basuras. Consúltese **http://java.sun.com** para más detalles sobre la recolección de basuras.

Las palabras clave aquí hay dos palabras clave. La primera es *declaración*.

La jerarquía de declaraciones

Con el sistema de ámbitos conseguimos relacionar cada acceso del código con una declaración. Si añadiéramos una dirección a las declaraciones, podríamos obtener las direcciones de cada acceso a través de las declaraciones.

En un lenguaje medianamente estructurado, las declaraciones se organizan con una cierta jerarquía. Por ejemplo: los métodos se declaran dentro de clases, las variables dentro de métodos. Casi toda declaración tiene una "declaración padre".

Supongamos que tenemos una dirección de memoria en la pila (por ejemplo 1111) a partir de la cual vamos a guardar una instancia de la clase `Persona`, con la siguiente implementación:

```
clase Persona
{
   atributo Cadena Nombre;
   atributo Cadena Apellidos;
   atributo Entero Edad;
}
```

Entonces se nos planteará el problema de averiguar cuándo espacio reservar en la memoria[83] para cada instancia de Persona. Lo que nos lleva a otra palabra clave: el *tamaño* de un tipo.

Tamaño de un tipo

Se nos ha planteado el problema de averiguar el tamaño del tipo Persona. Por tamaño de un tipo entenderemos la cantidad de bytes que ocupa cada instancia de dicho tipo en memoria.

En el caso de LeLi el tamaño de cada tipo es muy sencillo de calcular. Para empezar, los tipos básicos tienen un tamaño predefinido:

- 1+1 bytes para los booleanos
- 1+4 bytes para las cadenas
- 1+4 bytes para los enteros
- 1+4 bytes para los flotantes

(Las cadenas usan solamente 4 bytes porque "apuntan" a una zona del montículo donde está la verdadera información)

El primer byte de cada tipo está reservado para la información de tipo: es un identificador de tipo; de ahí se deduce que solamente puede haber 256 tipos diferentes en un programa LeLi.

Los métodos y constructores tienen siempre un tamaño de 0 bytes.

El tamaño de las clases es de 1 bytes + la suma de los tamaños de todos sus atributos. Así, el tamaño de la clase `Persona` es 1+(1+4)+(1+4)+(1+4) = 16 bytes.

Disposición de los objetos en la memoria

Ya sabemos que el tipo `Persona` ocupa 16 bytes en memoria. Pero aún no sabemos cómo se organizan esos bytes.

Es muy sencillo. El primer byte de memoria es un identificador de tipo. Este identificador permite realizar operaciones relacionadas con el tipo del objeto. Todas las instancias comienzan con este

83 Nótese que no utilizamos los términos "pila" o "montículo" en este apartado; los razonamientos que vamos a presentar son válidos en las dos zonas de memoria.

byte identificador.

Después, los atributos del objeto deben disponerse en cierto orden. A falta de un orden mejor, nosotros seguiremos el orden de declaración: los primeros atributos declarados serán los primeros atributos en la memoria. Así, la memoria ocupada por una instancia de `Persona` tendrá el siguiente aspecto:

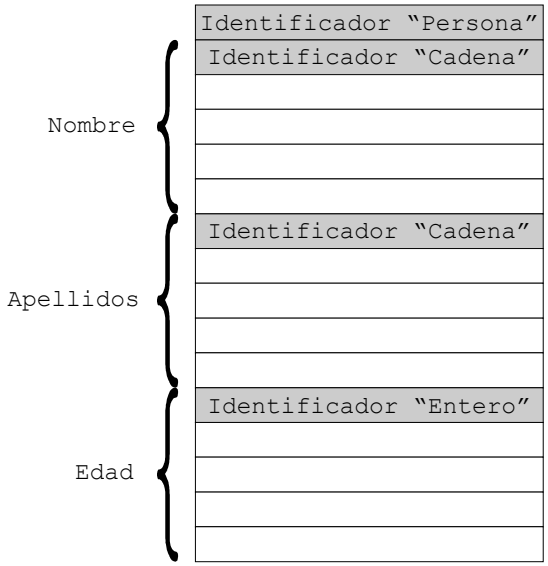

Ilustración 8.15 Disposición en memoria de Persona

El desplazamiento

De la disposición en memoria de los objetos surge el concepto de desplazamiento. El desplazamiento de un atributo A de una clase C es el número de bytes de distancia entre el inicio de una instancia de C hasta que empieza la memoria dedicada a A. Por ejemplo, el desplazamiento de la edad en `Persona` es de 9 bytes.

Lo que es aplicable a los atributos es aplicable al resto de los objetos que ocupan memoria en el lenguaje: los parámetros tienen un offset con respecto al inicio de la pila de su método. Las variables son un poco diferentes, porque pueden hacerse declaraciones dentro de un bucle de manera que los offsets no puedan calcularse completamente en tiempo de compilación.

Cálculo de las direcciones de las instancias

Las direcciones de cada elemento del lenguaje pueden ser calculadas en tiempo de compilación utilizando el desplazamiento. Así:

- Primero se crea la zona de datos estáticos, donde se guardan las informaciones de tipo y las constantes.
- Por otro lado se va generando la zona de código.
- El espacio reservado a la pila aparece después. Cada vez que se va a invocar un método, se apila un nuevo nivel (guardando registros y parámetros en la pila) y se modifican los registros de pila.
- Cada vez que se termina la ejecución de un método, se recupera el nivel de pila anterior.
- Las variables locales que se van declarando se apilan en la pila.

- Cada vez que se declara un nuevo objeto, se calcula su desplazamiento y se guarda en la declaración
- Las direcciones de los accesos se resuelven utilizando los desplazamientos.

Las instancias de `Declaration` necesitarán un nuevo campo, llamado `offset`, para guardar el desplazamiento de cada objeto que se declara.

Sección 8.4: Código intermedio

8.4.1: Introducción

En el "libro del dragón" ("Compiladores: Principios, técnicas y herramientas", de Aho, Sethi y Ullman) hay un capítulo entero dedicado a la generación de código intermedio.

Esta sección está enteramente basada en el capítulo 8 de dicho libro.

El código intermedio es una forma más de representar la información intermedia de un programa.

El código propuesto por Aho, Sethi y Ullman es una secuencia de instrucciones (llamadas "proposiciones" en el libro) de la siguiente forma:

```
x := y op Z
```

Donde x, y y z son nombres o variables temporales, op es un operador, como una suma aritmética o una comparación. Observese que las expresiones complejas no se permiten; deben ser descompuestas en series de proposiciones más simples.

Al código intermedio propuesto también se le llama "código de tres direcciones", porque siempre utiliza tres datos (x,y,z).

8.4.2: Proposiciones

Además de la forma general de una proposición, existen otras.

```
x := op y
```

Sirve para modelar los operadores unarios.

```
x := y
```

Para realizar copias.

```
goto etiqueta
```

Sirve para realizar saltos incondicionales.

```
if x opcomp y goto etiqueta
```

Sirve para realizar saltos condicionales. `opcomp` solamente puede ser un operador de comparación (<, >, =, etc.)

Las llamadas a procedimientos se modelan utiliznado las proposiciones `param` y `call`. De esta forma, una llamada a un procedimiento consta de cero o más proposiciones `param` seguidas de `call`:

```
param x1
param x2
param x3
call f, 3
```

Sirve para invocar el método `f(x1,x2,x3)`.

```
x := y[i];
```

Esta proposición es de las llamadas proposiciones con índices. Lo que se le asigna a x no es el valor de y, sino el que se encuentra a "i" posiciones de memoria de y. i debe ser un objeto de

datos, una constante o una variable temporal.

```
x[i] := y;
```

Es similar a la proposición anterior, pero a la inversa: la dirección que se modifica es el destino, y no la fuente.

```
x := &y; x := *Y; *x := y;
```

Son las tres formas disponibles para trabajar con punteros.

8.4.3: Objetivos y ventajas del código intermedio

El código intermedio es un paso intermedio entre el AST enriquecido y el código generado:

Ilustración 8.16 Lugar del código intermedio

Utilizar un código intermedio presenta varias ventajas. Es suficientemente abstracto para ser independiente de la máquina destino, pero suficientemente concreto para poder ser sometido a ciertas optimizaciones (las independientes de la máquina).

También puede distribuirse en lugar del código objetivo, y convertirse en código máquina en la propia máquina objetivo, aplicándose las optimizaciones dependientes de la máquina (en lugar de tener que limitarse al mínimo común múltiplo). Algo parecido es lo que puede hacerse con la compilación JIT de la plataforma .NET de Microsoft.

Generar código intermedio también facilita la reusabilidad: una vez desarrollemos un compilador/optimizador para código intermedio, podemos utilizarlo en otros proyectos que generen código intermedio compatible, aumentando la productividad.

Para más información sobre el código intermedio, sírvase el lector leer el capítulo 8 de “Compiladores: Principios, técnicas y herramientas”, de Aho, Sethi y Ullman.

Sección 8.5: Generación de instrucciones y expresiones

8.5.1: Introducción

En esta sección veremos cómo generar código para cada tipo de instrucción o expresión que podamos encontrarnos. Comenzaremos con las instrucciones de control (instrucción condicional y bucles) y después nos concentraremos en las instrucciones-expresiones.

8.5.2: Instrucciones condicionales

Caso simple: un solo bloque condicional

El caso de un bloque condicional simple se traduce como un salto condicional sobre una condición. Así, el código generado para el siguiente bloque:

```
si(<condicion 1>)
{
   <bloque 1>
}
```

Será el siguiente:

```
<código necesario para evaluar condicion1>
JNE condicion1, 1, condicionFalsa1
<código del bloque 1>
condicionFalsa1: # instrucción vacía
```

Caso 2: varios bloques condicionales

En este caso tendremos una instrucción condicional como la que sigue:

```
si(<condicion 1>)
{ <bloque 1> }
|(<condicion 2>)
{ <bloque 2> }
...
|(<condicion n>)
{ <bloque n> }
```

Entonces el código resultante será como muchos casos simples, colocados uno detrás de otro, con la particularidad de que hay que saltar al final del bloque en cuanto una de las condiciones se cumple:

```
<código necesario para evaluar condicion1>
JNE condicion1, 1, bloque2
<código del bloque 1>
JMP finalCondicional
bloque2: <código necesario para evaluar condicion2>
JNE condicion2, 1, bloque3
<código del bloque 2>
JMP finalCondicional
...
bloqueN: <código necesario para evaluar condicionN>
JNE condicionN, 1, finalCondicional
<código del bloque N>
finalCondicional: ... ;
```

Caso 3: Instrucción condicional completa

La instrucción condicional en este caso finaliza con una cláusula "otras":

```
si(<condicion 1>)
{ <bloque 1> }
|(<condicion 2>)
{ <bloque 2> }
...
|(<condicion n>)
{ <bloque n> }
|otras
{ <bloque otras> }
```

El código del bloque "otras" debe ejecutarse si y sólo si ninguno de los bloques anteriores se ha cumplido. A efectos prácticos, el código generado será equivalente a éste otro:

```
si(<condicion 1>)
{ <bloque 1> }
|(<condicion 2>)
{ <bloque 2> }
...
|(<condicion n>)
{ <bloque n> }
|(cierto)
{ <bloque otras> }
```

Así que podemos ver que éste caso es sencillamente una versión del caso anterior con la última condición siempre cierta.

Gestión de las etiquetas

Nótese que, sobre todo en el último caso, la instrucción condicional puede llegar a utilizar un número elevado de etiquetas.

Es muy común disponer de un objeto especializado, que llamaremos "gestor de etiquetas", que permita obtener nuevas etiquetas libres. El gestor puede ser algo tan sencillo como un objeto con un atributo entero que va incrementándose cada vez, de manera que devuelve "et01", "et02", "et03", etc en cada petición de etiquetas libres.

Actualmente antlraux no proporciona un gestor de etiquetas, pero podría hacerlo en un futuro.

8.5.3: Bucles

Bucle mientras

Es el más sencillo. Tiene la forma:

```
mientras (condición)
{
   bloque de instrucciones
}
```

El código para generarlo también es muy sencillo:

```
inicioBucle: <código para condición>
JE 0, <condición>, finBucle # si la condición es falsa, saltar al final
<código para bloque de instrucciones>
JMP inicioBucle
finBucle: ...
```

Bucle hacer-mientras

El bucle hacer-mientras tiene la siguiente forma:

```
hacer
{
   bloque de instrucciones
} mientras (condición);
```

El código necesario para generarlo utiliza únicamente una etiqueta:

```
inicioBucle:
<código para bloque de instrucciones>
<código para condición>
JE 1, <condición>, inicioBucle # si la condición es cierta, saltar al inicio
...
```

Bucle desde

El bucle desde tiene la siguiente forma:

```
desde (exp1; condicion; exp2)
{
   bloque de instrucciones
}
```

Y el código para generarlo será el siguiente:

```
<código para exp1, que normalmente es una asignación>
inicioBucle: <código para condición>
JE 0, <condición>, finBucle # salta al final si condición es falsa
<código para bloque de instrucciones>
<código para exp2, que normalmente es un post-incremento>
JMP inicioBucle
finBucle: ...
```

8.5.4: Instrucciones/expresiones

Expresiones aritmético-lógicas: registros y valores temporales

En una máquina sin registros, traducir una expresión a código es muy simple: solamente hay que hacer uso de la pila, apilando y desapilando valores. Cuando se introducen los registros, sin embargo, todo se complica.

Trabajar con registros es mucho más rápido que trabajar direcciones de memoria pura y dura. Al mismo tiempo, el número de registros de una máquina clásica es limitado; con mucha frecuencia algunos valores simplemente no pueden ser asignados a registros, porque "no quedan registros libres". Los valores que deben ser manejados y no pueden ser contenidos en un registro se guardarán en la pila, en una zona de valores temporales.

Para un método o función dado, es posible precalcular en tiempo de compilación el número de

bytes que se necesitarán para guardar valores temporales, así que con frecuencia se aloja el espacio para los valores temporales nada más iniciar el código de dicho método.

El problema de elegir qué datos guardar en registros y cuales guardar en la zona de datos temporales es NP-completo, así que no queda más remedio que recurrir a las heurísticas.

Lo más usual es disponer de un "gestor de registros" durante la generación de código. Este objeto se encarga de gestionar los registros; tiene en cuenta cuáles están ocupados y libres, y proporciona al generador de código "identificadores de registro", que son objetos que encapsulan a un registro libre, si lo hay, o a una zona de datos temporales.

La forma de trabajar con un gestor de registros es:

- El generador de código precisa de un registro para almacenar un valor, así que pide al gestor de registros un registro libre.
- El gestor de registros comprueba si hay algún registro libre con las características deseadas. Si es así, devuelve su identificador de registro, y lo marca como ocupado. En caso contrario, devuelve un identificador de registro que apunta a una zona de memoria.
- Cuando el valor del registro ya no es necesario, es responsabilidad del generador de código comunicárselo al gestor de registros, para que éste pueda volver a utilizarlo más adelante.
- En la implementación del gestor de registros deben tenerse en cuenta factores que liberan o reservan registros automáticamente, como las invocaciones a métodos. Lo usual es que se cree un gestor de registros para cada método.

Un gestor de registros puede ser útil incluso en máquinas sin registros (porque se puede encargar de reservar y liberar espacios temporales en la pila). Aunque en la actualidad antlraux no proporciona ningún gestor de registros, es posible que incluya uno en el futuro.

Una vez obtenido el identificador de registro de los operandos, generar las expresiones aritmético-lógicas es trivial: hay que utilizar ADD para las sumas, SUB para las restas, etc.

Asignaciones

Las asignaciones suelen poder transformarse en una orden MOV (si no tenemos en cuenta posibles optimizaciones). Así, para la instrucción de asignación `exp1 = exp2`, y sobreentendiendo que `exp1` tiene L-value y `exp2` tiene R-value, y los tipos son compatibles respecto a la asignación, el código generado será:

```
<código para obtener el identificador de registros de la dirección exp1>
<código para obtener el identificador de registros del valor de exp2>
MOV <valor-exp2>, <dirección-exp1>
```

Accesos a atributos (NO a métodos)

Decir "acceso a atributo" es lo mismo que decir "dirección de memoria". Al final, cada acceso a un atributo se convierte en una dirección de memoria accedida en el código.

Normalmente un acceso tendrá la siguiente forma:

```
raiz.atributo1.atributo2. ... .atributoN
```

Lo que deberemos hacer es comenzar por la izquierda, la raíz, y luego ir calculando la dirección final basándonos en los conocimientos que ya tenemos sobre la disposición de las instancias de clases en la memoria.

Normalmente la dirección de la raíz la habremos precalculado y guardado en su declaración (en la

instancia de Declaration que modela la declaración de "raíz"). El objeto atributo1 comenzará, digamos 7 byte más tarde (tras el identificador de tipo de raíz), y el siguiente 3 bytes más tarde que el inicio de atributo2... tanto "7" como "3" son los *offsets* que también deberemos de haber calculado.

Como puede verse, calcular las direcciones de un acceso a un atributo es tan sencillo como comenzar con una dirección en la raíz y luego ir sumando *offsets*.

Accesos a métodos (invocaciones)

En la sección 8.3 (Gestión de la memoria) hemos visto cómo hay que preparar la pila al invocar un método. Básicamente debemos:

- apilar BSP, y luego hacer BSP=SP
- apilar PC y al menos los registros usados en el método en la pila,.
- apilar también la dirección donde comienza el objeto "invocador" del método
- apilar los parámetros

Es decir, el código a generar es:

```
<código necesario para obtener la dirección del objeto invocador>
<código necesario para obtener la dirección donde comienza el nuevo método>
<código necesario para obtener la dirección de memoria del primer parámetro>
<código necesario para obtener la dirección de memoria del segundo parámetro>
...
PUSH <dirección del objeto invocador>
PUSH <dirección del primer parám
etro>
PUSH <dirección del segundo parámetro>
...
PUSH ... # registros utilizados por el método
CALL <dirección del nuevo método>
SUB ESP, <tamaño de los parámetros>
```

Sección 8.6: Optimización de código

8.6.1: Introducción

Si se ignoran las cuestiones relativas a la optimización, la generación de código debería ser un proceso muy sencillo: basta con recorrer el AST de entrada y "traducir" cada parte del código; se establece una única manera de traducir cada tipo de instrucción, expresión y declaración del lenguaje, y se aplica a cada nodo del AST (consúltese la sección para más información sobre los bloques básicos).

Sin embargo un proceso tan simple generará un código poco eficiente, "de poca calidad". En la actualidad es poco frecuente que esto se tolere: se necesitan programas de "buena calidad", que funcionen bien y rápido.

Nótese que utilizamos el término "de buena calidad" y no "óptimo": El problema de la optimización de código es NP-completo[84], así que encontrar el programa óptimo para suele ser imposible. No obstante se espera que el compilador realice al menos realice una serie de optimizaciones básicas.

Las optimizaciones son transformaciones que se realizan sobre un código para aumentar su rapidez de ejecución y disminuir su ocupación de memoria sin alterar los resultados que se obtienen con él.

Hay dos grandes grupos de optimizaciones: las que dependen de la máquina objetivo y las que no. Las primeras se basan en aprovechar las características del hardware sobre el que se va a ejecutar el código, mientras que las segundas se basan en heurísticas aplicadas al propio código. Empezaremos con estas últimas.

Téngase presente que solamente haremos una introducción a las optimizaciones en este capítulo; hacer un estudio exhaustivo de todas las técnicas de optimización queda fuera del ámbito de este libro; podría escribirse un libro completo solamente sobre ese asunto.

8.6.2: Optimizaciones independientes de la máquina objetivo

Las optimizaciones independientes de la máquina objetivo son:

- Eliminación de subexpresiones comunes
- Eliminación de código inactivo
- Transformaciones algebraicas
- Optimizaciones relacionadas con constantes e invariantes
- Optimizaciones de reordenamiento de cálculos

Eliminación de subexpresiones comunes

Esta transformación analiza el código existente buscando subexpresiones que calculen el mismo valor. Por ejemplo, en la ecuación `y = (x+1)*(x+1)` no tiene sentido calcular x+1 dos veces.

84 NP Completo = De gran complejidad computacional (más que polinómica). Aunque no está formalmente demostrado, se sospecha que los problemas NP-completos son computacionalmente irresolubles incluso para muestras pequeñas.

Eliminación de código inactivo

Supongase que la variable "y" del ejemplo anterior no vuelve a utilizarse en el resto del código. Entonces no es necesario calcular su valor; puede ignorarse la expresión por completo. Precisamente esto es lo que hace este tipo de transformación: elimina los cálculos innecesarios.

Transformaciones algebraicas

Las transformaciones algebraicas más útiles son aquellas que intercambian expresiones o sub expresiones "caras" por otras equivalentes pero más "baratas". Algunos ejemplos de estas transformaciones son:

- cambiar `x*1` por `x`
- cambiar `x+0` por `x`
- cambiar `1+2*3` por `7`
- cambiar `1>2` por `falso`
- cambiar `"¡Hola "+"mundo!"` por `"¡Hola mundo!"`
- etc.

Esta optimización puede implementarse simplemente sustituyendo una expresión por su E-value, si éste es no nulo.

Optimizaciones relacionadas con constantes e invariantes

Estas optimizaciones se parecen mucho a las algebraicas. Son optimizaciones que se basan en la invariabilidad de una cierta expresión o un cierto cálculo, ya sea en todo el programa o en una parte de él.

Por ejemplo existe la *optimización de las invariantes de un bucle.* Esta optimización se aplica sobre el cuerpo de un bucle, y se basa en buscar expresiones que se recalculen en cada iteración del bucle, pero cuyo valor no cambie. Cuando una expresión no cambia de valor dentro de un bucle se llama *invariante del bucle.* La optimización consiste por cambiar el cálculo de la invariante por el valor que mantiene dentro del bucle. Nótese que esta optimización solamente afecta a las instrucciones del cuerpo del bucle; una instrucción externa a éste no sufriría alteración alguna.

Optimizaciones de reordenamiento de cálculo

Existen ocasiones en las que un determinado cálculo puede realizarse más rápidamente dependiendo del orden de evaluación de las subexpresiones. Un ejemplo muy conocido de este caso es el problema de la multiplicación de matrices.

Supóngase un lenguaje de manipulación matemática que contemple en sus expresiones la posibilidad de multiplicar matrices de números de diferentes dimensiones. Es decir, que el lenguaje admite órdenes como la siguiente:

```
M = A * B * C ;
```

Siendo M, A, B y C matrices de números de diferentes dimensiones (A es de mxn, B es de nxp y C es de pxq, con lo que M es de mxq).

Pues bien, dependiendo de los valores de m,n,p y q, resultará más eficiente calcular la expresión como (A*B)*C o como A*(B*C), pues el número de operaciones será menor. Así que siempre

que los índices m,n,p y q sean fijos y conocidos en tiempo de compilación, será posible optimizar el cálculo de la matriz M para que se realize el menor número de operaciones posible.

8.6.3: Optimizaciones dependientes de la máquina objetivo

El hardware actual está en constante evolución; cada innovación en cada procesador hace posible una nueva optimización del código generado para dicho procesador. Así, es imposible enumerar todas las posibles mejoras que pueden realizarse para cada procesador – cada pieza de hardware ofrece un sinfín de posibles optimizaciones. En esta sección solamente enumeraremos algunas optimizaciones pero no todas.

- Asignación de registros
- Transformaciones de reordenamiento
- Desenrollado de bucles
- Uso de hardware específico

Asignación de registros

Los registros de una máquina suelen ser una buena herramienta de optimización de código, dado que son mucho más rápidos que la memoria.

Lo ideal sería que todos los valores de un programa pudieran ser manejados con registros, y nunca con accesos a memoria, pero esto no es posible porque el número de registros de una máquina suele ser muy limitado – algunos valores deberán ser guardados en memoria.

La optimización de asignación de registros consiste en buscar en cada momento qué datos guardar en los registros y qué datos guardar en memoria. El objetivo de esta optimización es minimizar el número de accesos a la memoria distribuyendo sabiamente los registros.

Transformaciones de reordenamiento

En las máquinas primitivas todas las órdenes se ejecutaban en un ciclo de reloj de la máquina, de forma secuencial (primero la primera, luego la segunda, etc). En los procesadores actuales, sin embargo, es frecuente que varias órdenes se ejecuten en paralelo – las órdenes se dividen en "fases", al igual que la arquitectura del procesador, de tal forma que si hay n fases entonces puede haber n órdenes ejecutándose simultáneamente en el analizador (estando cada una en una fase). El proceso de ejecución se parece en este caso a una "cadena de montaje" en la que van entrando las instrucciones.

Uno de los principales problemas de esta arquitectura son los bloqueos; ciertas instrucciones pueden tardar más que otras en realizar una fase, de manera que "bloquean la cadena de montaje". Las transformaciones de reordenamiento cambian el orden de las órdenes de los programas para minimizar los ciclos de bloqueo.

Otro problema son las cancelaciones. En ciertas ocasiones una instrucción (típicamente un salto condicional) "cancela" el resto de las instrucciones de la cadena de montaje, lo que puede provocar la pérdida de muchos ciclos de procesamiento. Este problema suele resolverse con hardware adicional (resolución temprana de las comparaciones, cacheo de las dos ramas del salto, etc.) aunque el reordenamiento también puede aliviar la situación.

Los procesadores actuales son capaces de reordenar automáticamente las instrucciones en tiempo de ejecución, así que esta optimización ha caído un poco en desuso.

Desenrollado de bucles

La técnica de desenrollado de bucles es muy popular en las optimizaciones. Tal y como su nombre indica, la técnica de desenrollado de bucles consiste en "repetir" el cuerpo de un bucle más de una vez, cambiando los valores a modificar.

El desenrollado de bucles consta de dos fases. En la primera se "replican" n veces las líneas del cuerpo del bucle, modificando los accesos que se realizan. "n" es el nivel de desenrollado del bucle.

La segunda fase es la de reordenamiento: las nuevas líneas se reordenan para minimizar los ciclos de bloqueo.

En la siguiente figura se muestra un ejemplo de desenrollado triple de un bucle:

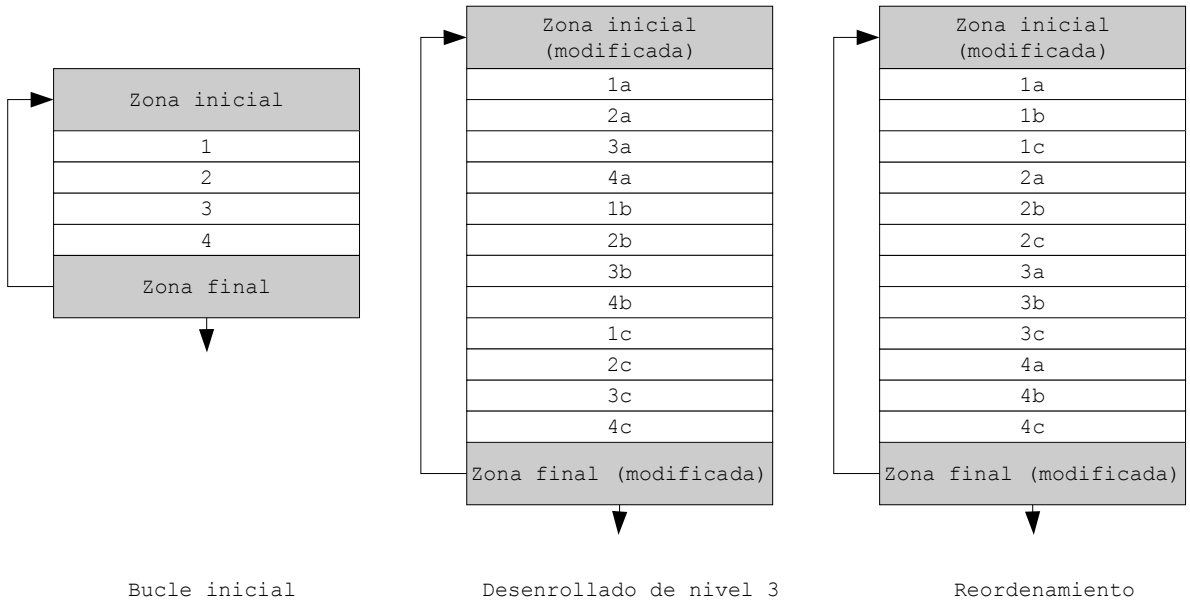

Ilustración 8.17 Desenrollado de un bucle

En la figura se muestra el conjunto de bloques básicos de un bucle (izquierda), un desenrollado de nivel tres del cuerpo del bucle (centro) y un reordenamiento de las instrucciones del cuerpo.

El desenrollado de bucles no es una ciencia exacta. El nivel de desenrollado y las posibles reordenaciones que pueden realizarse dependen enormemente del cuerpo del bucle.

Uso de hardware específico

En ciertas ocasiones los fabricantes de hardware incluyen componentes con una función muy específica, en oposición a la naturaleza "genérica" del sistema clásico (UAL+memoria+registros). Hay multitud de ejemplos:

- Componentes de tratamiento vectorial, como el conjunto de registros MMX de la arquitectura x86
- La tecnología 3DNow! de AMD
- Los coprocesadores matemáticos
- Las tarjetas gráficas

Además, los procesadores actuales tienden a alejarse cada vez más del paradigma RISC (que tiene un juego de órdenes muy reducido) para ir añadiendo nuevas órdenes en cada generación de procesadores. Así, antiguamente la única manera de generar código para:

```
x = x+1;
```

era generar (suponiendo que x esté en la dirección 210 de memoria)

```
MOV A, [210]
ADD A, 1
MOV [210], A
```

Sin embargo con la aparición de la instrucción INC es posible realizar:

```
INC [210]
```

Ahorrándose ciclos de ejecución y el uso de un registro.

Es posible incrementar enormemente la eficiencia de ciertos programas utilizando componentes de hardware específicos o instrucciones como INC.

Por último, el propio sistema operativo también puede ofrecer servicios que aumentan la eficiencia, como librerías estándar.

8.6.4: Portabilidad vs. eficiencia

Mientras que las optimizaciones independientes de la máquina siempre son deseables, las dependientes de la máquina no siempre pueden aplicarse.

En muchas ocasiones no se conoce de antemano la máquina en la que se va a ejecutar el código. Por ejemplo, en la actualidad muchas aplicaciones comerciales deben poder ejecutarse en el binomio "Wintel", esto es, una máquina con un procesador "Intel" y un sistema operativo "Windows".

Ahora bien, existen varios procesadores Intel, y existen varios sistemas operativos Windows. Es inviable distribuir una aplicación compilada para cada pareja {procesador, sistema operativo}, así que lo que hacen los fabricantes de software es sencillo: distribuyen software con las optimizaciones que funcionen en *todas* las combinaciones posibles; se utiliza el "mínimo común múltiplo" de las optimizaciones.

Un programa compilado con el mínimo común múltiplo de las optimizaciones será probablemente más ineficiente que el mismo programa compilado específicamente para una máquina.

Tres soluciones se proponen para solucionar esto.

Código abierto

La primera de ellas es proporcionar el código fuente de la aplicación junto con los binarios ejecutables. Si un cliente desea optimizar la aplicación, no tiene más que recompilarla activando las optimizaciones específicas de su máquina.

Sin embargo la estrategia del código abierto no es siempre posible de implementar: muchas empresas son reacias a proporcionar el código fuente de sus aplicaciones.

Librerías del sistema

Otra forma de aumentar la eficiencia consiste en utilizar librerías externas al programa en sí, que lo "aislen" de la máquina objetivo. Al actualizarse estas librerías mejorará la eficiencia del

programa automáticamente.

Algunos ejemplos de estas librerías son:

- Los drivers de los diferentes componentes hardware.
- La librería gráfica OpenGL.
- La librería de desarrollo de videojuetos DirectX, de Microsoft.

Compilación JIT

La estrategia de la compilación JIT también se basa en utilizar un paso intermedio, aunque en este caso el propio código generado sea el paso intermedio. Por ejemplo, el compilador de la plataforma .NET de Microsoft no genera ficheros binarios que se ejecuten directamente sobre windows (en código máquina para windows) sino que genera un código intermedio llamado CLR (*Common Language Runtime*). Este código puede ser interpretado por una máquina virtual, de forma similar a lo que se hace con java.

Sin embargo, la plataforma .NET también proporciona un compilador JIT (*Just In Time*) que "compila" el CLR para generar código máquina de windows, adaptado a la versión del sistema operativo y del procesador que el usuario esté utilizando. Y así los programas son siempre eficientes (pues aprovechan todas las optimizaciones dependientes de la máquina) y portables.

La plataforma .NET sería la perfecta forma de crear programas eficientes y portables si no fuera porque la portabilidad que ofrece es limitada: solamente se contemplan algunos sistemas operativos de Microsoft. Una verdadera pena que limita mucho el uso de la plataforma.

Sección 8.7: Miscelánea

8.7.1: Introducción

En esta sección observaremos cómo tratar diversas situaciones diversas que pueden darse a la hora de generar código para una determinada máquina objetivo, y que no encajan en ninguna otra sección del capítulo.

Nos centraremos en dos áreas: el mapeado de tipos y la implementación de librerías y tipos predefinidos del sistema.

8.7.2: Mapeado de tipos

Generalmente los lenguajes que compilaremos serán mucho más ricos que las máquinas para las que generaremos código; las diferencias entre la máquina y el lenguaje pueden ser muchas.

Por lo general la diferencia más importante es la ausencia de soporte nativo en la máquina para todos los tipos del lenguaje. Muchas máquinas clásicas tenían un único tipo de datos, un entero, con el que debían "emularse" el resto de los tipos del lenguaje.

El proceso de "emular" los tipos de los que una máquina objetivo carece se denomina *mapeado de tipos*, y a dicho proceso estará dedicado este apartado.

Mapeado de flotantes sobre enteros

En este sub apartado veremos cómo trabajar con flotantes sobre una máquina que en principio no está preparada para trabajar con ellos. El formato de números flotantes que vamos a utilizar es el estándar de 32 bits del IEEE:

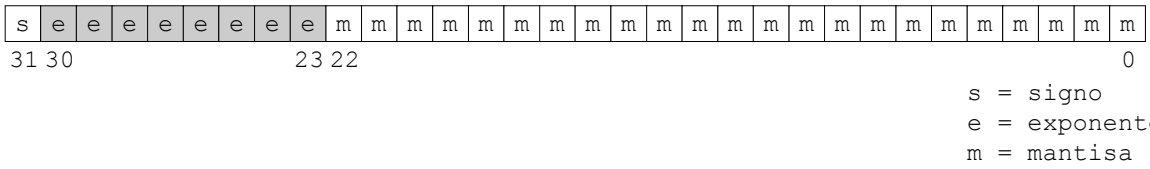

Ilustración 8.18 El formato de número flotante de 32 bits del IEEE

El exponente es almacenado como un número positivo "reducido", de manera que para obtener el verdadero exponente es necesario restarle 27. La mantisa se guarda de forma normalizada, con un "1" implícito delante de la fracción de 23 bits. Así se maximiza la precisión.

La siguiente fórmula permite calcular el valor "real" de un número guardado en formato flotante:

$$n = s * 1.m * 2^{(e-127)}$$

Algunos valores "especiales" son representados utilizando el exponente. Cuando e = 255, m codifica condiciones especiales como NaN (*Not a Number*, "no un número"), que es el resultado de dividir cualquier flotante por 0. El exponente e = 0 se utiliza en números no normalizados – números tan pequeños que el rango del exponente supera los 8 bits.

Hay un formato de 64 bits que utiliza el mismo formato básico, pero con 11 bits para el exponente y 52 para la mantisa.

¿Qué hacer cuando una máquina no soporta por defecto este tipo? Es decir, cuando no incluye un juego de órdenes para manejar flotantes. Lo usual es que, al menos, soporte enteros de 32 bits, y operaciones de manejo de bits. En dichas condiciones nosotros mismos deberemos codificar

funciones en el lenguaje ensamblador de la máquina para "convertir" de entero a flotante y de flotante a entero, así como para sumar, restar, multiplicar y dividir flotantes, e invocar dichas funciones (utilizando la pila, apilando y desapilando los parámetros de la forma habitual).

Por motivos de espacio no podemos detallar extensamente cómo implementar estas funciones; para alguien que ha programado un compilador hasta el punto de generar código no debería ser muy difícil.

Mapeado de caracteres y booleanos

En el caso de que la máquina objetivo no ofrezca órdenes para el manejo directo de caracteres o booleanos, la solución es inmediata: mapearlos sobre el tipo entero más pequeño que pueda manejar la máquina (muchas máquinas pueden manejar enteros de varios tamaños).

Si el mapeado se hace cuidadosamente, no será necesario implementar ninguna función "extra", como ocurría con los flotantes.

Recuérdese también que en algunos casos los caracteres que se necesitan son unicode, y por lo tanto ocupan 16 bits, en lugar de los 8 usuales.

Mapeado de cadenas de caracteres y tablas

Por lo general las cadenas y tablas pueden manejarse de manera similar. Y hay muchas maneras de hacerlo; la más conocida – aunque no por ello la más adecuada en todos los casos – es la del lenguaje C.

En C no hay diferencia alguna entre una cadena y una tabla de caracteres. Son simplemente una serie de caracteres consecutivos en memoria, cuyo inicio está apuntado por un puntero. A efectos prácticos, la única diferencia entre una tabla y una cadena de caracteres es que esta última tiene que terminar obligatoriamente con el carácter de terminación, '\0'.

El carácter de terminación es una de las formas de determinar el tamaño de una cadena; aunque no es especialmente rápido, es económico. Presenta dos problemas: lentitud (hay que recorrer toda la cadena para averiguar su longitud).

Otra opción consiste en utilizar los primeros bytes de la cadena (usualmente los 4 primeros) para codificar el tamaño de dicha cadena, y así no depender del carácter de terminación.

En cuanto a la zona de almacenamiento, hay dos opciones: la pila o el montículo. La pila puede utilizarse satisfactoriamente para almacenar cadenas o tablas de tamaño máximo conocido (que no puedan exceder un tamaño determinado, de acuerdo a una cláusula en su creación). Muchos lenguajes solamente admiten cadenas y tablas de tamaño máximo conocido, declarando dicho tamaño al declararlas:

```
Nombre: Cadena [20]; // Nombre tiene un tamaño máximo de 20
Usuarios: Tabla [20..65] de Usuarios ; // Usuarios tiene 45 usuarios,
                                       // numerados del 20 al 65
```

La pila, sin embargo, resulta muy difícil de utilizar si las cadenas o tablas tienen un tamaño dinámico, como viene ocurriendo desde hace algún tiempo en la mayoría de los lenguajes de programación. En dichos casos es mejor utilizar el montículo para almacenar los datos. Es muy útil almacenar un puntero al comienzo de dichos datos en la pila.

Mapeado de tipos definidos por el usuario

Aún hoy es poco fecuente que las máquinas objetivo ofrezcan órdenes de manejo de tipos

definidos por el usuario[85].

Los tipos que un usuario define siempre pueden representarse como "conjuntos de datos" (o "registros de datos"), de manera que los datos que representan pueden disponerse consecutivamente en la memoria. En el caso de los tipos "clase" estos conjuntos de datos son los atributos. Así, el tipo `Persona` puede representarse como dos cadenas (nombre y apellidos) y un entero (edad) dispuestos consecutivamente en memoria.

El mapeado de tipos definidos por el usuario es más sencillo de implementar, por ejemplo, que el mapeado de flotantes.

8.7.3: Librerías y tipos básicos del sistema

Los tipos definidos por el usuario pueden definirse en base a otros tipos definidos por el usuario, pero en última instancia estos tipos estarán definidos en torno a tipos básicos del lenguaje: enteros, cadenas, reales, etc. Las librerías que el usuario pueda crear, al final, se basarán también en dichos tipos básicos, y en otras librerías básicas ofrecidas por el compilador.

Los tipos básicos y librerías del sistema suelen ser "especiales" en ciertos aspectos; admiten más operaciones de las que admiten los tipos generados por el usuario. Estas capacidades especiales tienen que ver con la forma en que su código se genera.

Tomemos por ejemplo la clase `Sistema` del lenguaje LeLi. Dicha clase possee varios métodos que, al ser invocados con los parámetros adecuados, imprimirán mensajes en la pantalla. ¿Cómo conseguiremos ésto?

Está claro que para hacerlo hemos de conseguir que los métodos de impresión de la clase `Sistema` se traduzcan a código ensamblador de una manera especial; una que implica la invocación de órdenes especiales de impresión de caracteres en la pantalla. En otras palabras, el cuerpo generado de los métodos de la clase `Sistema` debe contener órdenes "especiales", que permitan a la máquina objetivo imprimir mensajes por pantalla.

Hay tres maneras de lograr ésto:

- Embebiendo código generado dentro del código fuente de dichas clases.
- Haciendo que el compilador controle de una forma especial dichas clases durante la generación de código.
- Soportando nativamente dicha clase en la máquina objetivo.

Opción 1: Código embebido

Ésta aproximación consiste en dotar al lenguaje del compilador con capacidad para incluir código de la máquina objetivo directamente en sus métodos. El mecanismo se parecería mucho al bloque __asm que utiliza el lenguaje C para incluir código ensamblador[86]:

85 Excepción hecha de la JVM y de la plataforma .NET.
86 Los bloques _asm son una extensión de C++ no estándard de Microsoft.

```
void main (void)
{
   int a = 0;
   __asm{
      add a, 1
   }
}
```

Podríamos hacer lo mismo; añadiendo un tipo de instrucción especial para insertar código de la máquina objeto directamente dentro de nuestro código fuente.

Este mecanismo tiene el inconveniente de la dependencia; si la máquina objetivo cambia, el código puede quedar inservible.

Además, el código embebido no es optimizable por el compilador.

Opción 2: Condiciones especiales controladas

Otra opción consiste en que el generador de código esté "vigilando" los métodos que declaran en el código fuente. Al generar el código para los métodos especiales, como `Sistema.imprime` el generador cancela la forma usual de generarlos; ignora las instrucciones del cuerpo del método y genera el código que sea necesario. De la misma forma el generador está "pendiente" de las invocaciones a métodos especiales, interviniendo cuando es necesario para generar las invocaciones adecuadamente.

Esta solución adolece del mismo problema que la anterior: dependencia de la máquina objetivo. En este caso es el propio código del generador el que es dependiente de la máquina objetivo, y que será inservible si se cambia.

Opción 3: Soporte nativo

Por último tenemos la posibilidad de que la propia máquina objetivo se encargue de manejar adecuadamente las clases con métodos especiales. Esta solución solamente es aplicable en máquinas objetivo que estén profundamente relacionadas con el lenguaje, al estilo de la JVM o de la máquina virtual de la plataforma .NET.

La JVM, por ejemplo, es capaz de tratar la reflexión del lenguaje java[87] casi automáticamente.

87 La reflexión es el mecanismo que java ofrece para manejar metaclases

Sección 8.8: Conclusión

Como ya se mencionó en la introducción, la generación de código no ilustra ninguna de las caracteríscas que ya hemos presentado en el capítulo anterior (recorrido de ASTs), por lo que hemos decidido no malgastar espacio, energía y tiempo en implementar esta etapa del compilador.

Por otra parte, un documento sobre compiladores no puede estar completo sin una sección sobre generación de código, por lo que finalmente me he decidido por un tono introductorio, sin llegar a presentar una implementación para el lenguaje LeLi. Como ya he dicho otros factores (falta de tiempo y de una máquina objetivo satisfactoria) también han influido en mi decisión.

Dicho esto, revisemos lo qué es lo que hemos estudiado en este capítulo.

Comenzamos con un estudio de las posibles máquinas objetivos y de la gestión de memoria. Después hemos introducido el concepto de código intermedio.

En la siguiente sección hemos estudiado (no muy profundamente) cómo generar código para las expresiones y declaraciones de un lenguaje, y después hemos presentado algunas estrategias de optimización.

Finalmemte en la última sección, “miscelánea”, veíamos aspectos varios de la generación de código.

Innumerables son los temas que no hemos cubierto. La optimización de código, por sí sola, basta para escribir un libro, y nosotros la hemos despachado en cinco páginas. Tampoco hemos implementado el nivel de generación de código de nuestro compilador, que se ha quedado inacabado.

Este capítulo ha sido menos práctico de lo que debería, en mi opinión. Mis disculpas a los lectores que se hayan aburrido. Es muy difícil poder presentar las partes más “jugosas” de un tema y al mismo tiempo mantener un tono introductorio.

El siguiente capítulo, “Conclusiones” es el último de este documento. En él dejaremos de lado el lenguaje LeLi, que ha sido el hilo central de los últimos cuatro capítulos, y nos centraremos en ANTLR. Presentaremos las conclusiones a las que hemos llegado con respecto a la herramienta.

Capítulo 9: Conclusiones

"Saltar rápidamente a conclusiones rara vez conduce a felices aterrizajes."
S. Siporin

Capítulo 9: Conclusiones.....413

Sección 9.1: Introducción.....414
Sección 9.2: ANTLR y el análisis léxico.....415
9.2.1: La primera impresión..... 415
9.2.2: Usabilidad..... 415
9.2.3: Eficiencia..... 415
9.2.4: Prestaciones..... 416
9.2.5: Carencias..... 416
9.2.6: Conclusión..... 417
Sección 9.3: ANTLR y el análisis sintáctico.....418
9.3.1: La primera impresión..... 418
9.3.2: Usabilidad..... 418
9.3.3: Eficiencia..... 418
9.3.4: Prestaciones..... 418
9.3.5: Carencias..... 419
9.3.6: Conclusión..... 419
Sección 9.4: ANTLR y el análisis semántico.....420
9.4.1: La primera impresión..... 420
9.4.2: Usabilidad..... 420
9.4.3: Eficiencia..... 420
9.4.4: Prestaciones..... 420
9.4.5: Carencias..... 421
9.4.6: Conclusión..... 422
Sección 9.5: Conclusiones finales.....423
9.5.1: Sobre ANTLR..... 423
9.5.2: Sobre el manejo de los ASTs..... 423
9.5.3: Sobre la eficiencia..... 423
9.5.4: Sobre el futuro de ANTLR..... 425
9.5.5: ¿Merece la pena ANTLR?..... 425
9.5.6: Final..... 426

Sección 9.1: Introducción

En los últimos 5 capítulos nos hemos dedicado a comentar los detalles de implementación del compilador de LeLi, a modo de ejemplo de uso de ANTLR.

En este capítulo abandonaremos definitivamente el lenguaje LeLi y nos dedicaremos a enunciar las conclusiones a las que hemos llegado, en lo referente a ANTLR.

He estructurado este capítulo según la estructura básica de un compilador o intérprete. Comenzaremos analizando cómo se comporta ANTLR en el nivel léxico, y seguiremos con los niveles sintáctico y semántico. Por último habrá una serie de conclusiones finales.

¡Bueno, comencemos!

Sección 9.2: ANTLR y el análisis léxico

9.2.1: La primera impresión

Es mala.

Como ya hemos visto, ANTLR utiliza el algoritmo pred-SLL(k) hasta sus últimas consecuencias, incluyento en el análisis léxico. Esto hace que haya que tener en cuenta los "caracteres de lookahead" si se plantean problemas de ambigüedad.

La primera impresión es mala porque desde siempre hemos utilizado expresiones regulares para realizar el análisis léxico.

9.2.2: Usabilidad

El asunto se hace patente si intentamos reconocer enteros y reales a la vez. Simplemente no podremos utilizar reglas independientes; deberemos utilizar la misma regla, y diferenciar cada tipo de número con un predicado sintáctico, tal y como veíamos en el análisis sintáctico de LeLi.

El algoritmo LL(k) favorece (levemente) que aparezcan algunas ambigüedades. En el caso de LeLi apareció una.

La primera vez que uno se enfrenta a estas dificultades es un tanto desesperante ("¿¡por qué no se habrán limitado a usar expresiones regulares y se habrán dejado de tonterías!?"). Pero las dificultades suelen solventarse muy rápido en la mayor parte de los casos.

9.2.3: Eficiencia

Ciertamente el código no es tan eficiente como debería. Simplemente hay demasiadas llamadas a métodos. Por ejemplo, hay demasiadas invocaciones del método LA(1), que podrían optimizarse utilizando una variable temporal. Cada vez que un carácter es consumido, se invoca el método `match(caracter)`, así que éste se invoca en muchas ocasiones.

ANTLR debería de proporcionar algún mecanismo para hacer las sub reglas *inline*, de manera que se ahorren algunas invocaciones de métodos.

Otra posible optimización es la eliminación de comprobaciones repetitivas. Por ejemplo, al reconocer la regla `IDENT`, se genera un bloque `switch` muy grande antes de reconocer cada letra:

```
switch ( LA(1)) {
case 'A':  case 'B':  case 'C':  case 'D':
case 'E': etc, etc..
{
   mLETRA(false);
   break;
}
```

Estas comprobaciones vuelven a repetirse al reconocer cada letra:

```
switch ( LA(1)) {
case 'a':  case 'b':  case 'c':  case 'd':
 ... etc
{
   matchRange('a','z');
   break;
}
```

Estas comprobaciones "extra" deberían poder evitarse, seguramente con el método de *inlineación* que comentamos más arriba.

La implementación de los predicados sintácticos también podría mejorarse: en ocasiones pueden hasta doblar el tiempo de ejecución de una regla.

9.2.4: Prestaciones

A pesar de los pequeños problemas de usabilidad y la merma de eficiencia, el método de generación de analizadores léxicos de ANTLR ofrece importantes ventajas:

- Pueden incluirse acciones en cualquier parte de cualquier regla de una forma muy sencilla. Con expresiones regulares no sería tan sencillo.
- La herencia de gramáticas se puede utilizar también con los analizadores léxicos
- El código generado se entiende perfectamente.
- Para simplificar el reconocimiento es posible aumentar k, sin que se pierda la eficiencia.
- Hay un recuperador de errores por defecto que puede ser modificado si se desea.
- Las reglas permiten el paso de parámetros y la devolución de valores.

Muchas de estas prestaciones son posibles gracias a que cada regla del analizador léxico está implementada en un método independiente del analizador. Por lo tanto la optimización de "inlineación" de la que hablábamos antes solamente podría utilizarse en reglas en las que no se aprovechasen todas estas capacidades.

Existen algunas prestaciones específicas para los analizadores léxicos:

- ANTLR ofrece además servicios adicionales para manejar palabras reservadas, con lo que se alivia mucho el reconocimiento de identificadores.
- ANTLR permite intercambiar analizadores léxicos con ayuda del objeto `LexerSharedInputState`.
- La sensibilidad a mayúsculas puede desactivarse muy fácilmente con una opción del analizador. Para tratamientos más finos (por ejemplo, permitir mayúsculas y minúsculas solamente en algunos tokens) basta con reescribir el método `CharScanner::testLiteralsTable (String, int)`.
- **La clausura positiva y el cierre de Kleene permiten escribir la mayoría de las reglas como si fueran expresiones regulares**
- Además, en las reglas léxicas pueden especificarse rangos de caracteres.
- Si por algún motivo no nos gusta la implementación de ANTLR, podemos suministrar nuestro propio `TokenStream` para leer los tokens.
- Compatibilidad con Unicode.

9.2.5: Carencias

La principal carencia que detectamos en los analizadores léxicos es el funcionamiento de la opción `paraphrase`: solamente puede especificarse para tokens definidos en reglas léxicas (y no en los tokens imaginarios definidos en la zona de 'tokens' del analizador). Ya hemos visto que esta situación puede remediarse utilizando reglas "ficticias" en el analizador, pero no deja de ser un "truco"; debería de haber una forma más estandarizada de hacerlo.

También mejorarse el tratamiento de mayúsculas/minúsculas. Debería poder activarse y

desactivarse la sensibilidad a mayúsculas en cada regla o sub regla (esto puede hacerse sobreescribiendo `CharScanner::testLiteralsTable(String, iny)`, pero debería poder realizarse automáticamente).

Por último, resaltar que jamás se invoca el método `Token::setFilename` durante el análisis léxico, de manera que para conseguir tokens con nombres de fichero no basta con implementar una versión propia de `Token`; también hay que sobreescribir el método `ChasScanner::makeToken` en nuestro analizador léxico.

Con todo estas carencias son poco importantes. Además, todas pueden solucionarse de alguna forma.

9.2.6: Conclusión

El instinto me indicaba que donde más puntos perdería el sistema sería en la usabilidad. Pero finalmente no resultó muy traumático. Además, implementar el análisis léxico ayuda a comprender el paradigma pred-LL(k).

Así que finalmente es en la eficiencia donde se pierden más puntos.

En las prestaciones del sistema no tiene rival. Ofrece más de lo que podría ofrecer una implementación basada en expresiones regulares (a costa de perder eficiencia).

Sección 9.3: ANTLR y el análisis sintáctico

9.3.1: La primera impresión

Es muy buena.

El uso de la clausura positiva y el cierre de Kleene permiten especificar muy fácilmente las listas de elementos. Comprender los entresijos de pred-LL(k) es más sencillo que comprender los conflictos shift-reduce de los analizadores LR (como Bison). Añadir predicados sintácticos es fácil, una vez que se conocen un par de reglas. Además, no hay que hacerlo casi nunca (5 reglas en la gramática de LeLi).

Uno tiene que acostumbrarse a que en LL "solamente se reconoce una regla cada vez, y no varias simultáneamente" como ocurre en LR. Si no se cumple una determinada regla, entonces hay un error.

La recuperación de errores es un poco más complicada, porque para implementarla eficazmente es necesario conocer exactamente cómo se genera el código de ANTLR.

A primera vista la construcción de ASTs es sencilla: los operadores ^ y ! facilitan la tarea.

9.3.2: Usabilidad

Como ya hemos dicho, es bastante cómodo utilizar – una vez que se ha comprendido el paradigma pred-LL(k), con el que el programador toma contacto desde el análisis léxico.

Cuando comenzamos a construir ASTs un poco más complicados, sin embargo, comenzaremos a tener dificultades. Construir ASTs heterogéneos es una tarea laboriosa y que puede dar lugar a muchos errores antes de ser realizada correctamente. Además está poco documentada.

9.3.3: Eficiencia

Los analizadores sintácticos adolecen de problemas de eficiencia similares a los léxicos: demasiadas invocaciones a métodos.

De nuevo sería aconsejable poder hacer *inline* algunos métodos del analizador, para aumentar la velocidad.

La implementación de los predicados sintácticos también podría mejorarse.

9.3.4: Prestaciones

Para el nivel sintáctico ANTLR ofrece las mismas prestaciones que para el nivel léxico (exceptuando las prestaciones reservadas al nivel léxico):

- Pueden incluirse acciones en cualquier parte de cualquier regla de una forma muy sencilla.
- La herencia de gramáticas se puede utilizar también con los analizadores sintácticos
- El código generado se entiende perfectamente.
- Para simplificar el reconocimiento es posible aumentar k, sin que se pierda la eficiencia.
- Hay un recuperador de errores por defecto que puede ser modificado si se desea.
- Las reglas permiten el paso de parámetros y la devolución de valores.
- El AST puede mostrarse gráficamente en una ventana SWING.

La recuperación de errores merece una mención especial, porque es bastante fácil de

implementar.

9.3.5: Carencias

Además de los inconvenientes mencionados en el apartado "eficiencia", el modo de manejo de los ASTs que utiliza ANTLR es poco práctico.

Para empezar, se echan en falta más métodos para crear árboles en las acciones. Por ejemplo, no hay una forma clara y estandarizada de realizar una duplicación de un AST (hay que recurrir a invocar directamente un método de la fábrica de ASTs que contiene el analizador).

A pesar de que mejoró con respecto a la versión anterior, el trabajo con ASTs heterogéneos sigue siendo complicado, por varios motivos:

- Cada nueva clase de AST debe ser implementada por el programador. ANTLR podría automatizar la tarea de crear las nuevas clases.
- Hay mucha discusión sobre si es acertado que las instrucciones de modificación y creación de ASTs se incluyan en las acciones. Mucha gente opina que deberían estar más separadas; algo así como tener "acciones de modificación del AST" y "acciones normales", separadas. Las acciones de modificación del AST tendrían un lenguaje totalmente independiente del lenguaje de generación del analizador (java, C++, C#).
- La "especial" disposición en memoria de los ASTs (en la que cada nodo tiene dos "punteros" a su "hijo-abajo" y a su "hermano-derecha") hacen que de vez en cuando se produzcan resultados insospechados al trabajar con ASTs (por ejemplo, podemos olvidarnos de eliminar los hermanos de un nodo al añadirlo como hijo a otro)

En definitiva, las principales carencias que encontramos se dan en el manejo de los ASTs.

Solamente noté una carencia en el recorrido de ASTs: la versión 2.7.2 de ANTLR no permite nombrar sub reglas. Es decir, no permite hacer algo así:

```
persona : IDENT apellidos:(IDENT IDENT) ;
```

La principal aplicación de las sub reglas con nombre es la recuperación de errores. En el momento en el que podemos nombrar una sub regla podemos especificar un manejador de excepciones para dicha regla:

```
persona : IDENT apellidos:(IDENT IDENT) ;
exception [apellidos] catch [RecognitionException re]
{ ... /* tratar el error */ }
```

En el estado actual no queda más remedio que dividir la regla en dos:

```
persona : IDENT apellidos;
apellidos : IDENT IDENT ;
exception catch [RecognitionException re]
{ ... /* tratar el error */ }
```

En ocasiones dividir las reglas no es deseable (se pierde eficiencia, y trabajar con el analizador se vuelve más complicado al aumentar el número de reglas).

9.3.6: Conclusión

En general implementar analizadores sintácticos con ANTLR es un placer. Sin embargo, si necesitamos manejar ASTs "complejos" (heterogéneos), nos dejará un regusto amargo (especialmente si generamos código C++).

Sección 9.4: ANTLR y el análisis semántico

9.4.1: La primera impresión

Es un poco desconcertante.

Acostumbrados a tratar flujos de caracteres o de tokens, de repente estamos analizando estructuras arbóreas. Y aún en ese caso seguimos utilizando un analizador pred-LL(k).

El aspecto del patrón árbol es extraño – no parece tan sencillo de tratar como los flujos de tokens.

Y sin embargo, al poco tiempo de estar usando el patrón árbol comenzamos a estar muy cómodos. Incluso escribir enunciados sintácticos resulta sencillo.

9.4.2: Usabilidad

Recorrer los ASTs es bastante sencillo; un iterador sin acciones puede implementarse rápidamente. Lo más usual es que cada tipo de AST tenga un nodo característico como raíz, así que la gramática será LL(1), y no harán falta predicados sintácticos.

Sin embargo, el análisis semántico no consiste simplemente en recorrer ASTs. También hay que transformarlos, enriquecerlos. Y aunque ANTLR sea bastante usable recorriendo ASTs, no lo es tanto modificándolos.

En la sección anterior mencionábamos que la forma de controlar la construcción de los ASTs es más complicada de lo necesario. Pues bien, estas complicaciones se agravan en la fase de análisis semántico. Véase el apartado de “carencias” para un informe más detallado sobre esta cuestión.

9.4.3: Eficiencia

De nuevo encontramos los mismos problemas de eficiencia que en el resto de los niveles: simplemente demasiados métodos son invocados, y en ciertas ocasiones las “comprobaciones de caracteres” se suceden. Aunque de nuevo gracias a esta pequeña pérdida de eficiencia podemos disfrutar de muy buenas prestaciones.

A los problemas de eficiencia ya mencionados hay que añadirles que ANTLR anima al programador a crear un nuevo AST en cada análisis semántico, y no a modificar el ya existente. La creación de cada nuevo AST requiere bastante tiempo.

9.4.4: Prestaciones

De nuevo la estrategia pred-LL(k) proporciona a ANTLR múltiples capacidades para el análisis:

- Pueden incluirse acciones en cualquier parte de cualquier regla de una forma muy sencilla.
- La herencia de gramáticas se puede utilizar también con los analizadores semánticos
- El código generado se entiende bien (aunque se vuelve borroso si se activa la generación automática del AST)
- Para simplificar el reconocimiento es posible aumentar k, sin que se pierda la eficiencia.
- Hay un recuperador de errores por defecto que puede ser modificado si se desea.
- Las reglas permiten el paso de parámetros y la devolución de valores.
- Puede mostrarse gráficamente el AST en una ventana SWING.

9.4.5: Carencias

Hay mucho que mejorar en el manejo de los ASTs.

Al trabajar los ASTs, sigue habiendo "efectos colaterales": si un AST "A" se hace hijo de otro AST "B", entonces automáticamente todos los hermanos de A también pasarán a ser hijos de B. Este comportamiento puede ser desconcertante.

El "pseudo-lenguaje" de transformación y manejo de ASTs en las acciones es insuficiente:

- Faltan órdenes, por ejemplo para duplicar o buscar ASTs.
- No se contempla la creación automatizada de ASTs heterogéneos
- Si en una regla se está creando un AST, es complicado cambiar la cabeza de dicho AST[88].

Pero sin duda lo que más impacta en la usabilidad es la imposibilidad de acceder a un hijo concreto del AST, de manera que hay que "iterar" por los hijos hasta llegar al nodo deseado. Por ejemplo, supongamos la siguiente regla de un analizador semántico:

```
arbol: #(ARBOL IDENT (TRATAMIENTO IDENT)? IDENT )
```

Supongamos que debemos tratar *en una acción* un AST que satisfaga la regla arbol. Normalmente podremos reconocer el AST con la propia regla y acceder a los nodos utilizando etiquetas:

```
arbol: #(ARBOL nombre:IDENT (TRATAMIENTO tratamiento:IDENT)? apellido:IDENT )
{
   String trat;
   if(tratamiento==null) trat = "";
   else trat = tratamiento.getText()+ " ";

   System.out.println ( trat + nombre.getText() + apellido.getText() );
}
```

Ahora bien, supongamos que queremos trabajar con el AST *fuera del analizador*, es decir, en una clase de java normal y corriente.

Entonces la cosa se complica; debemos "iterar manualmente por los hijos" para obtener los diferentes datos que necesitamos:

```
public void trataArbol(AST ast)
{
   AST nombre = ast.getFirstChild();
   AST hermano = nombre.getNextSibling();
   String trat="";
   AST apellido = hermano;
   if(hermano.getType()==TRATAMIENTO)
   {
      trat = hermano.getText() + " ";
      apellido = hermano.getNextSibling();
   }

   System.out.println ( trat + nombre.getText() + apellido.getText() );
}
```

La estrategia de "crear un nuevo AST" en lugar de "modificar el AST existente" no me parece acertada; debería haber más servicios de modificación del AST actual, y primar éstos por encima de los de creación de uno nuevo.

88 Porque se desconoce el padre del AST; es necesario pasarlo como parámetro o guardarlo como atributo en el analizador.

Tampoco se ayuda mucho al programador en lo referente a ASTs heterogéneos; su creación debería automatizarse.

9.4.6: Conclusión

Todos los problemas encontrados en el nivel sintáctico de ANTLR (dificultad de manejo, falta de eficiencia) provienen del mismo sitio: la peculiar arquitectura de los ASTs. La estructura "un nodo = un primer hijo y un hermano", es, en mi opinión "demasiado flexible".

En la siguiente sección analizaremos cómo podrían solucionarse.

Sección 9.5: Conclusiones finales

9.5.1: Sobre ANTLR

Al comenzar a trabajar con ANTLR pensaba que las principales dificultades que tendría serían derivadas de implementar un analizador léxico con un autómata recursivo descendente. Más tarde constaté que la mayoría de las dificultades que pueden presentarse durante el análisis léxico se resolvían bastante bien con las "ayudas" que ANTLR proporciona en el análisis léxico: reglas EBNF, tratamiento de literales, tratamiento de mayúsculas y minúsculas, rangos de caracteres... Incluso superaba a flex en algunos aspectos, como la compatibilidad con Unicode y la posibilidad de generar código para varios lenguajes.

El análisis sintáctico, por su parte, resultó muy cómodo: habiendo comprendido el fundamento de pred-LL(k) durante el análisis léxico, resultó muy sencillo adaptarse a los flujos de tokens.

La recuperación de errores, por su parte, fue un poco más complicada de comprender. Cuando uno ha trabajado con Bison, adaptarse a que "solamente se puede reconocer una regla cada vez" es complicado. Finalmente dominé la recuperación de errores con un enfoque práctico: observando los cambios que se producían en el código generado, a la sazón bastante inteligible.

Aprender a *crear* el AST fue aproximadamente tan complicado como implementar la recuperación de errores; después hubo que *implementar* la creación del AST, con lo que podemos concluir que la mayor parte del tiempo de desarrollo con ANTLR se emplea en la creación y manejo del AST.

9.5.2: Sobre el manejo de los ASTs

El manejo de los ASTs es sin duda la tarea pendiente para Terence Parr y comañía. Da la impresión de que en un primer momento no valoraró suficientemente la necesidad de "crear y transformar" los ASTs, en oposición a "recorrerlos". Así, para implementar el nivel semántico, ANTLR se basa en *iteradores de árboles* (subclases de `TreeParser`) pero se echan en falta *creadores de árboles* (¿`TreeCreator`?) y *modificadores de árboles* (¿`TreeModifier`?).

Las únicas ayudas que ANTLR proporciona al programador son los "patrones de creación y modificación" (#[], ##, etc) que resultan insuficientes. Además, al estar mezclados con las acciones, tienden a disminuir la portabilidad de los analizadores (porque suelen aparecer mezclados con instrucciones de código nativo).

Por último, poder acceder automáticamente a los hijos de cada AST utilizando su nombre es una necesidad básica que ANTLR no cumple. La estructura (nodo-hijo-hermano), aunque muy flexible, no la permite.

9.5.3: Sobre la eficiencia

Una vez establecido que el principal problema de ANTLR es el pobre manejo de los ASTs, analicemos el segundo problema: la eficiencia.

Uno de los principales argumentos en contra de los analizadores recursivos descendentes es que "solamente son útiles porque son muy fáciles de implementar, y el código es más inteligible". Su eficiencia, comparada con los analizadores basados en tablas, es como mucho mediocre.

Los analizadores recursivos descendentes no son la única opción a la hora de implementar la estrategia LL de análisis. Existen otras soluciones basadas en tablas, en las que no hay que pagar el "sobrepeso" de las invocaciones de métodos.

¿Por qué, entonces, se han utilizado los analizadores recursivos descententes? ¿Simplemente porque son más fáciles de implementar? ¿Porque el código es más inteligible?

Rotundamente no.

ANTLR utiliza analizadores recursivos descendentes para tratar de una forma estandarizada los *errores*.

En los lenguajes avanzados, como java o C# (y en menor medida C++) cuando hay un error el comportamiento estandarizado es lanzar una excepción. Para manejar la excepción se utilizan cláusulas try-catch. Si no las hay, se sale del método en el que se produce la excepción.

Se presentaban varias opciones para manejar los errores en ANTLR:

- No utilizar excepciones. Utilizar un método de manejo de errores "propio", como hace bison.
- Utilizar las excepciones, pero no utilizar autómatas recursivos descendentes.
- Usar la combinación autómata recursivo descendente+excepciones (ésta fue la opción elegida)

La primera opción (no utilizar excepciones) puede redundar en un analizador más rápido. Sin embargo se pierde mucha potencia. Con las excepciones, el usuario puede controlar con mucha exactitud cómo y cuándo gestionar los errores. Además, el usuario de ANTLR puede incluir nuevos errores (haciendo subclases de las excepciones de ANTLR) y lanzarlas desde cualquier clase; los analizadores de ANTLR podrán trabajar con estos nuevos errores fácilmente.

No utilizar autómatas recursivos descendentes plantea la siguiente pregunta: ¿dónde colocar las cláusula `try/catch` para recuperar las excepciones? Una opción sería colocar una cláusula try-catch en cada elemento de una regla (al hacer un *match* de cada carácter, token o nodo AST). Sin embargo esto sería tremendamente ineficiente, y podría no ser lo más adecuado en todos los casos (se intentaría recuperar la entrada demasiado pronto). También podría ponerse un único bloque try/catch alrededor de todo el analizador; pero entonces con toda seguridad la recuperación sería infructuosa.

Es más adecuado tener un bloque try/catch por regla. Esto ofrece una granularidad "buena", que no captura los errores "demasiado pronto ni demasiado tarde", y con una eficiencia adecuada. Pero llegados a este punto es tan eficiente tener "todas las reglas en un solo método, con un try/catch por regla" como tener "una regla en cada método, con un try/catch por método"- manejar los bloques try/catch se haría tan complicado que compensa utilizar métodos separados.

Además, con la solución actual es muy sencillo y eficiente pasar parámetros a una regla y recuperar valores que la regla arroje.

Dicho esto, habrá ocasiones en las que el usuario no precise de un profundo control de las excepciones, y no desee pasar o recuperar valores de una regla. En estos casos el usuario debería poder especificar que no desea un método para dichas reglas.

Por ejemplo, en el analizador léxico de LeLi las reglas `LETRA` y `DIGITO` son reglas auxiliares, escritas simplemente para mejorar la legibilidad del analizador. De cara a la implementación, sin embargo, no aportan nada; sería deseable que ANTLR dispusiera de una palabra reservada como `inline` que hiciera que el siguiente código:

```
inline LETRA: (A..Z)|(a..z);
inline DIGITO: (0..9);
IDENT: (LETRA|DIGITO|'_')(LETRA|DIGITO|'_')* ;
```

Se transformase en éste otro antes de generar el analizador:

```
IDENT: (((A..Z)|(a..z))|(0..9)|'_')(((A..Z)|(a..z))|(0..9)|'_')* ;
```

Que es más eficiente y equivalente en términos de entradas reconocidas y errores recuperables.

9.5.4: Sobre el futuro de ANTLR

En la página web de antlr (**http://www.antlr.org**), recientemente remodelada, hay varios artículos sobre las discusiones del equipo desarrollador de ANTLR sobre el futuro de la herramienta.

Uno de ellos es el resultado de una discusión sobre el futuro de ANTLR que tuvo lugar entre Loring Craimer, Monty Zukowski y Terence Parr entre los días 12 y 14 de Julio de 2003 (el enlace es **http://www.antlr.org/workshop/cabal-2003**). En esta discusión se decidieron algunas capacidade que la nueva versión de ANTLR (3.0) tendrá.

Para empezar, unas acciones especiales servirán para modificar el AST. Dichas acciones irán dentro de los símbolos #{} en lugar de en las acciones (que van delimitadas con llaves, {}).

Este código está copiado de las notas ofrecidas como resultado del workshop de julio 2003 (ver enlace más arriba)

```
#{ acción de árbol }
#{ #(a b c) } // Construcción del árbol
#[...] // Construcción de un nodo
#{ {pred1}? #(a b c) // aplicar solamente si pred1 es cierto
   | {pred2}? #(b a c)
   | #(c b a)
}
```

*****

Para más detalles consúltese el enlace mostrado más arriba.

Incluyendo estas nuevas acciones en los analizadores sintácticos y semánticos se refuerza el aspecto “de creación y modificación” de los ASTs.

Parece que los ASTs seguirán sin poder referenciar a los hijos de un nodo con su nombre fuera de los analizadores.

En cuanto a la eficiencia, en el documento aparecen varias referencias a poder hacer reglas `inline`.

El desarrollo de ANTLR suele ser bastante lento; es probable que la versión 3.0 aparezca en los próximos dos años.

9.5.5: ¿Merece la pena ANTLR?

¡Sí!

A pesar de las carencias, el balance total es muy positivo. El código que genera, aunque no es el más rápido, es muy robusto y comprensible. Es una solución más avanzada que el binomio bison+flex (permite crear y recorrer ASTs).

Quizás queda un poco atrás en el apartado de la eficiencia. Conforme aumente la velocidad de los procesadores esta diferencia se irá haciendo cada vez más pequeña, pero en algunos casos en los que se necesite procesar rápidamente una cantidad alta de datos podría resultar insuficiente.

En el ámbito académico, sin duda ANTLR es muy adecuado: su sencillez facilita el aprendizaje.

9.5.6: Final

Y con esto hemos llegado al final del documento. Espero que el lector encuentre tanto placer leyéndolo como lo hizo el autor escribiéndolo. Si ayuda al menos a una persona a comprender mejor los compiladores me daré por satisfecho.

¡Ah! ANTLR significa *ANother Tool for Language Recognition* (Otra Herramienta para el Reconocimiento de Lenguajes).

Gracias por su atención.

Apéndice A: Referencias

Sobre ANTLR

- En la página oficial de ANTLR, **http://www.antlr.org**, hay abundante documentación.
- Para una información más directa, se puede utilizar la lista de correo de ANTLR. Es un servicio ofrecido por Yahoo groups: **http://www.yahoogroups.com/groups/antlr-interest**.
- Además hay un FAQ de ANTLR: **http://www.jguru.com/faq/ANTLR**
- Y un foro: **http://www.jguru.com/forums/home.jsp?topic=ANTLR**

Sobre ANTLRAUX

- La página oficial de antlraux: **http://antlraux.sourceforge.net**
- Eventualmente puede haber información adicional en mi página personal: **http://imaginatica.us.es/~enrique**

Sobre compiladores

- "El libro del dragón". **Compiladores. Principios, Técnicas y Herramientas**. A.V. Aho, R. Sethi, J.D. Ullman.
- El grupo de noticias *comp.compilers*: **http://compilers.iecc.com**
- C.N. Fischer, R.J. Leblanc. **Crafting a Compiler with C**, Benjamin/Cummings Publishing Company, Inc. Redwood City, California, 1991.
- A.I. Holub, **Compiler Design in C**, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1990.
- B.W. Kernighan, P.J. Plauger, **Software Tools in Pascal**, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.
- B. Meyer, Introduction to the **Theory of Programming Languages**, Prentice-Hall International Series in Computer
- R. Sethi, **Lenguajes de programación. Conceptos y constructores**, Addison-Wesley Iberoamericana, Wilmington, Delaware, 1992.
- J.P. Tremblay, P.G. Sorenson, **The Theory and Practice of Compiler Writing**, McGraw-Hill International Editions, 1985.
- D.A. Watt, **Programming Language Processors**, Prentice-Hall International Series in Computer Science, 1993.

Otros

- La página oficial de java: **http://java.sun.com**
- La plataforma eclipse de desarrollo java: **http://www.eclipse.org**
- El modo ANTLR para la plataforma eclipse: **http://antlreclipse.sourceforge.net**
- El editor Emacs: **http://www.gnu.org/software/emacs/emacs.html**.
- El editor Xemacs: **http://www.xemacs.org**.
- El modo ANTLR para Emacs y Xemacs: **http://antlr-mode.sourceforge.net**.

Apéndice B: Glosario

A.

- **ADT, Sistema**: *Sistema Ámbito/Declaración/Tipo*. Es un conjunto de conceptos y reglas para modelar el comportamiento contextual de los lenguajes de programación.
- **Algoritmo**: una lista de instrucciones donde se especifica una sucesión de operaciones necesarias para resolver cualquier problema de un tipo dado.
- **Ámbito**: Es una abstracción que sirve para representar y organizar la jerarquía de declaraciones de un lenguaje de programación. Normalmente los ámbitos se organizan "anidadamente" (un ámbito está dentro de otro, y así sucesivamente hasta llegar al ámbito raíz). Los ámbitos permiten gestionar ciertos aspectos semánticos de los lenguajes de programación, como la vida de las variables o el enmascaramiento de nombres.
- **Análisis léxico**: Proceso de análisis según el cual un conjunto (flujo) de caracteres, generalmente suministrados por medio de un fichero de texto, se transforma en un flujo de tokens. La herramienta que realiza un análisis léxico es el analizador léxico. Por ejemplo, cuando un analizador encuentra una serie de caracteres como ésta:

```
int 2;
función foo() {}
```

  El analizador la transforma en algo como lo siguiente:

```
RES_INT NUMERO PUNTO_COMA
RES_FUNCION IDENT PARENT_IZQ PARENT_CER LLAVE_AB LLAVE_CER
```

  Donde cada una de las palabras en mayúsculas escritas es un token. Nótese que un analizador léxico es capaz de "ver" los saltos de línea y los espacios, pero por lo general éstos se "ignoran".
- **Análisis sintáctico**: Proceso de análisis según el cual un conjunto (flujo) de tokens se agrupa siguiendo una serie de reglas gramaticales. Normalmente el flujo de tokens analizados proviene de un análisis léxico previo. Las funciones del análisis sintáctico son 2: encontrar errores en el flujo suministrado y, si no los hay, generar un Árbol de Sintaxis Abstracta (AST). La herramienta que realiza el análisis sintáctico es el analizador sintáctico. Una analogía usual que se hace con el lenguaje natural es que el análisis léxico "transforma las palabras en frases".
- **Análisis semántico**: Proceso de análisis según el cual un Árbol de Sintaxis Abstracta se transforma en otra cosa, pudiendo ser esta última otro Árbol de Sintaxis Abstracta.
- **Analizador recursivo descendente**: los analizadores recursivos descendentes son un conjunto de métodos mutuamente recursivos (que se llaman unos a otros) que normalmente utilizan un solo símbolo de *lookahead* (una sola variable) para tomar las decisiones de análisis.
- **Analizer**: Término inglés utilizado para refererirse a los analizadores sintácticos.
- **Árbol de Sintaxis Abstracta**: Es una forma de representación de la información contenida en un fichero de código. Generalmente es el producto de un análisis sintáctico o semántico.
- **ASCII**: Conjunto de 256 caracteres al que se limitaban los primeros lenguajes de programación. Incluye principalmente los caracteres utilizados por la lengua inglesa y algunos símbolos especiales.
- **AST**: *Abstract Syntax Tree*, Árbol de Sintaxis Abstracta.

- **AST heterogéneo**: AST cuyos nodos no son todos de la misma clase.
- **AST homogéneo**: AST con todos los nodos de la misma clase.
- **AST degenerado**: Es un AST que no tiene un solo nodo como raíz. Puede considerarse que no tiene raíz, o que tiene varias (excepto cuando es un AST vacío).
- **AST vacío**: AST sin nodos.
- **Atributo**: Las clases son tipos compuestos, es decir, contienen varios "datos". Estas "datos" son los atributos de las clases.

B.

- **Basura**: Es un segmento ocupado del montículo que no es referenciado por ningún objeto de la memoria; el segmento debe ser liberado.

C.

- **Cierre de Kleene**: Uno de los modificadores del lenguaje EBNF. Representa "cero o más elementos" y se representa con el asterisco ("*").
- **Clase**: Es la unidad básica de información en un lenguaje orientado a objetos. Una clase consta de métodos y atributos. Las clases son definiciones de tipos de datos, y como tales pueden ser instanciadas en variables.
- **CLASSPATH**: Variable del sistema que indica a la JVM dónde buscar los ficheros *.class necesarios para la ejecución de programas java.
- **Clausura Positiva**: Es uno de los modificadores del lenguaje EBNF, que sirve para representar "uno o más elementos". Se representa con el signo "+".
- **Código máquina**: Código comprensible por una máquina en formato binario (formados por ceros y unos). El código máquina es incomprensible para las personas (excepto para unas cuantas raras excepciones). Los compiladores se encargan de transformar ficheros en un lenguaje de programación, comprensible por los humanos, en código máquina, comprensible por las máquinas. Los ficheros en código máquina pueden ser de tres tipos: ficheros ejecutables (los más usuales), como son los archivos *.EXE de Windows, librerías (ficheros en código máquina que contienen funciones que los ficheros ejecutables utilizan) y ficheros para máquinas virtuales.
- **Código intermedio**: Es una forma de representación intermedia entre el análisis semántico y la generación de código. Facilita las optimizaciones y su reutilización de un compilador a otro. También hace al compilador independiente de la máquina objetivo.
- **Compilación cruzada**: Es un tipo de compilación según la cual una máquina X compila un código fuente produciendo un código máquina para otra máquina diferente Y. A Y se le llama "máquina objetivo".
- **Compilador**: Programa capaz de generar ficheros en código máquina y librerías a partir de ficheros de texto escritos en un lenguaje de programación compilado.
- **Constructor**: Método especial de una clase que sirve para iniciar los atributos de sus instancias antes de poder usarlas.

D.

- **Declaración**: Una declaración es, básicamente, una asociación entre un nombre y un tipo. Las

declaraciones se organizan en ámbitos. Así, cuando declaramos la variable de tipo Entero llamada “Manolo” en el método “M”, estamos insertando en el ámbito del método “M” una declaración que relaciona el nombre “Manolo” y el tipo “Entero”.

- **Declaration**: Véase declaración.
- **Desplazamiento**: Distancia entre el comienzo de un dato en memoria y su “referencia”.
- **DFA**: *Deterministic Finite Authomata*, o Autómata Finito Determinista. Es el algoritmo que se utiliza para realiza el análisis léxico en muchas herramientas, como flex. Está basado en un autómata formado por estados y transiciones, únicamente.

E.

- **EBNF**: *Enhanced BNF*. Es una versión extendida de BNF, que permite utilizar cierres de Kleene y clausuras positivas, entre otras.
- **Enlazador**: Es un software encargado de combinar varios ficheros binarios (usualmente librerías) para generar otro fichero binario, ya sea una librería o un ejecutable.
- **E-Value**: Atributo que guarda el valor de una expresión. En ciertas ocasiones se utiliza para pre-calcular el valor de ciertas expresiones en tiempo de compilación, aunque esto no es siempre posible. Las expresiones sin E-Value tienen el atributo E-Value a null. Las que sí lo tienen pueden tener una cadena, entero, etc. Por ejemplo, el E-Value de la expresión 1+2+3 es el entero “6”, mientras que la expresión a+b no puede ser calculada en tiempo de compilación, por lo que tiene un E-Value de null.

F.

- **FILO**: *First In, Last Out*. Categoría de objeto “contenedor” (que sirve para guardar otros objetos) en la que el primer objeto que se inserta será el último en salir. Un ejemplo de objeto FILO es la Pila (no confundir con la zona de memoria).
- **FIFO**: *First In, First Out*. Categoría de objeto “contenedor” (que sirve para guardar otros objetos) en la que el primer objeto que se inserta será el primero en salir. Un ejemplo de objeto FIFO es la Cola.

G.

- **GNU**: *GNU is Not* Unix. Es un acrónimo recursivo. Esta agrupación de personas es una de las principales impulsoras del software libre *(free software)*. Para más información visite **http://www.gnu.org**.
- **Gramática**: Conjunto de reglas que definen un lenguaje.

H.

- **Herencia de gramáticas**: Es un mecanismo de ANTLR que permite aprovechar las reglas de una gramática en otra, sin tener que volver a escribirlas. Mejora la encapsulación.
- **Hola mundo**: Es el primer programa que suele escribirse cuando se está aprendiendo a programar en un nuevo lenguaje. Se limita a imprimir en la pantalla del ordenador un mensaje cualquiera, que suele ser “Hola mundo”.

I.

- **IDE**: *Integrated Developping Environment*. Es una herramienta que permite actuar con una

herramienta de desarrollo (por lo general, un intérprete) de manera "gráfica", es decir, con menús desplegables, botones, etc. Las IDEs evitan a los programadores el tener que utilizar diferentes programas para desarrollar software, así como el uso de consolas de comandos.

- **Interfaz (java)**: Es una de las prestaciones del lenguaje java, que consiste en un conjunto de atributos y métodos. La diferencia entre un objeto y una interfaz es que los métodos de esta última no tienen cuerpo, y como ningún objeto puede tener éste tipo de métodos las interfaces no son instanciables. Para que lo sean es necesario que un objeto las *implemente*.
- **Intérprete**: Es una herramienta que permite tratar una información codificada en un determinado lenguaje, "comprendiéndola", para obtener un resultado. Un ejemplo de intérprete es un compilador.
- **Instanciación de un tipo**: Mecanismo mediante el cual se crea una variable de dicho tipo.
- **Instrucción**: Unidad mínima de especificación de procesos en un lenguaje de programación orientado a objetos. Las instrucciones forman parte del cuerpo de los métodos. Los tipos de instrucciones diferentes son las asignaciones y las instrucciones de control. Las instrucciones de control cambian el orden de ejecución del resto de las instrucciones, que normalmente es secuencial.

J.

- **Java**: El término java se usa para describir la plataforma de desarrollo emblemática de Sun Microsystems, consistente en una máquina virtual y un conjunto de librerías, y para denominar el lenguaje de programación que se usa en dicha plataforma.
- **JVM**: *Java Virtual Machine*. La máquina virtual de java.

K.

- k: Es la letra que se utiliza para denominar el *lookahead* en un compilador.

L.

- **Lenguaje**: Es un conjunto de símbolos y reglas que sirve para representar información. Los lenguajes que tratamos en este documento son lenguajes de programación, es decir, lenguajes que sirven para definir algoritmos.
- **Lenguaje compilado**: Es un lenguaje de programación cuyo fin es ser compilado por un compilador para obtener uno o varios ficheros en código máquina.
- **Lexer**: Abreviatura de ***Lex****ical Analiz****er***, o analizador léxico.
- **Linker**: Véase enlazador.
- **Lookahead**: Dado un algoritmo de análisis aplicado sobre un flujo de símbolos, el lookahead de dicho algoritmo es el número de símbolos del flujo que dicho algoritmo "tiene en cuenta" antes de aplicar una regla al flujo. Una idea intuitiva del lookahead sería la "longitud de visión" del algoritmo: mientras más símbolos "ve", más lookahead tiene.
- **LR(k)**: Es un algoritmo de análisis de datos "de abajo a arriba", con un lookahead k. Más información en el primer capítulo.
- **LL(k)**: Es un algoritmo de análisis de datos "de arriba a abajo", con lookahead k. Más información en el primer capítulo.
- **L-value**: Atributo booleano de cualquier expresión de un lenguaje. Una expresión con L-value

puede ser utilizada en la parte izquierda de una asignación (puede asignársele un nuevo valor). Un ejemplo de expresión con L-value sería un acceso simple a una variable no constante: `a=2;` una expresión sin L-value sería un acceso a un literal: `"Hola"="Adiós";` no es válido en la mayoría de los lenguajes.

M.

- **Máquina**: Para nosotros, una máquina es cualquier mecanismo que permita ejecutar las acciones codificadas en el código generado por un compilador.
- **Máquina clásica**: Máquina formada por memoria, procesador, buses y (muy frecuentemente) registros.
- **Máquina objetivo**: Es la máquina para la cual se compila un código fuente. En la compilación cruzada, una máquina compila un código para que sea ejecutado por una tercera máquina, que se denomina "objetivo".
- **Máquina virtual**: Es un programa que funciona en otra máquina (que a su vez puede ser virtual) "simulando" una máquina distinta. Es un "emulador", y también es la especificación de la máquina emulada por ese emulador. Un ejemplo de máquina virtual es la JVM.
- **Método**: Los métodos son las partes de las clases en las que están codificadas las acciones que éstas pueden realizar. Un método consta de una cabecera, en la que se definen sus parámetros y los valores que puede devolver, y un cuerpo, en el que las acciones a realizar están codificadas en forma de instrucciones.
- **Método no vacío**: Es aquel método que devuelve un valor.
- **Método vacío**: No devuelve ningún valor.
- **Metaclase**: Las metaclases son una forma de representar y manipular los tipos de un lenguaje dentro del propio código del lenguaje. Dependiendo de las prestaciones del lenguaje, mediante el uso de metaclases se puede determinar si una instancia satisface un interfaz determinado, o si es una subclase de otra clase dada.
- **Montículo**: El montículo es una zona de la memoria de la máquina objetivo dedicada a contener las variables que no pueden ser gestionadas por la pila, es decir, las variables que deben "sobrevivir" a las funciones/métodos en las que fueron declaradas. El montículo se va dividiendo en "segmentos" conforme nuevas peticiones de reserva de memoria van llegando.

O.

- **Objeto**: El concepto de "objeto" es el pilar básico de un paradigma de la programación, la "programación orientada a objetos". Básicamente un objeto es un conjunto de informaciones (atributos) que presenta al exterior unos servicios en forma de interfaz (métodos). Esta manera de organizar la información en grupos con funcionalidades asociadas incrementa la *encapsulación* con respecto a otros paradigmas como la programación funcional. Los métodos de los objetos pueden compartirse y combinarse haciendo uso de la *herencia* y el *polimorfismo*.
- **Offset**: Véase desplazamiento.

P.

- **Parser**: Término inglés utilizado para referirse a los analizadores sintácticos.
- **PATH**: Variable del sistema que indica en qué directorios pueden buscarse los programas

accesibles directamente con su nombre (por ejemplo dir o format).

- **Pila**: Importante zona de memoria de la máquina objetivo encargada de diversos menesteres, tales como el alojamiento de variables locales y parámetros para funciones/métodos. También sirve para guardar y restaurar el estado de los registros antes y después de cada invocación de una función o método.
- **Pred-LL(k)**: Es el algoritmo de análisis que utiliza ANTLR, aunque siendo purista habría que llamarlo pred-SLL(k). Enriquece el algoritmo SLL(k) con predicados sintácticos y semánticos.
- **Predicado semántico**: Predicado que decide la validez de una regla de una gramática de acuerdo a informaciones de carácter semántico.
- **Predicado sintáctico**: Predicado que permite resolver incoherencias entre varias alternativas de una regla de una gramática entre las que hay conflictos por falta de lookahead. Permite incrementar el lookahead virtualmente hasta el infinito (permite incluso reconocer gramáticas no-LL(k) para ningún k)
- **`PRIMERO(regla)`**: Sea M el conjunto de todas muestras que satisfacen `regla`. En tal caso, se define `PRIMERO(regla)` como el conjunto de todos los símbolos iniciales de M.

R.

- **Recuperación de errores**: Dícese de los mecanismos y algoritmos que aplica un compilador para continuar analizando una entrada tras encontrar un error en dicha entrada.
- **Recolector de basura**: Es un proceso o hilo que se encarga de revisar periódicamente el montículo revisando la basura que encuentre (véase Basura).
- **R-value**: Atributo booleano que tienen todas las expresiones. Una expresión con R-value puede ser utilizada en la parte derecha de una asignación (porque devuelve un valor). Un ejemplo de expresión con R-value es un literal: `a=1;` Una expresión sin R-value es más dificil de encontrar, pero hay algunas. Por ejemplo, en LeLi, las llamadas a métodos vacíos (que no devuelven un valor) no tienen R-value: `a=Sistema.imprime("Hola")`;

S.

- **Scanner**: Analizador léxico (abreviatura del inglés).
- **Scope**: Véase ámbito.
- **Semántica estática**: Es el conjunto de restricciones y reglas semánticas que pueden comprobarse durante la etapa de compilación.
- **SLL(k)**: *Strong LL(k)*. Es un algoritmo descrito por primera vez por Terence Parr en su tesis. SLL(k) es una variante del algoritmo de análisis LL(k). Su principal interés radica en que los analizadores SLL(k) no crecen exponencialmente en tiempo y espacio al crecer k (lo hacen linealmente). La "pega" es que SLL(k) es ligeramente menos potente que LL(k).
- **Software de código abierto**: Software cuyo código fuente debe ser distribuido junto a los programas ejecutables. No confundir con software libre: mientras que el software libre se basa en un planteamiento filosófico (cuya finalidad es que el software sea accesible para todo el mundo, libre de cargos o patentes) el software de código abierto se basa en un planteamiento técnico (si proporcionamos el código fuente con los binarios, el desarrollo del producto será más sencillo y eficiente).
- **Software libre**: Movimiento filosófico que defiende el derecho de la humanidad a acceder al software de forma total y libre de cargos. Este "derecho total de acceso" incluye la posibilidad

de leer y modificar el código fuente de los programas. El único derecho que no tienen los usuarios es hacer "no libre" el software libre. La organización mundial más importante en el terreno del software libre es la *Free Software Foundation* (FSF).

- **Software propietario**: Al contrario que en el software libre, en el software propietario los usuarios solamente disponen de los ficheros en código máquina (también llamados "ficheros binarios") de los programas. Usualmente el usuario no tiene ningún derecho sobre el software, salvo el de ejecución. En la actualidad la mayoría de las empresas de informática producen y utilizan software propietario, aunque esta situación cambia gradualmente. La compañía desarrolladora de software propietario más conocida es Microsoft.

T.

- **Tipo**: Un tipo es una "categoría" o una "clase" de "dato" de un lenguaje de programación. Los datos de un lenguaje de programación son las declaraciones y las expresiones. Los tipos son el pilar básico del análisis semántico, porque muchas de las reglas de la semántica de un lenguaje suelen hacer referencia a ellos. Durante la fase de análisis semántico el compilador debe comprobar que la entrada suministrada satisface las reglas semánticas de tipos del lenguaje.
- **Token**: Símbolo. Es la partícula más pequeña de información que maneja un analizador sintáctico. Son los equivalentes de las "palabras" del lenguaje natural. La cualidad esencial de un token es su tipo (que en ANTLR se representa con un entero)
- **Token imaginario**: Token que jamás aparecerá en las muestras, pero se introduce en la gramática por comodidad.
- **Token de sincronismo**: Token que al ser encontrado en una entrada garantiza que el analizador está en un estado reconocible del análisis. Así, es posible realizar una recuperación de errores (situándose el analizador en un estado válido del análisis) si durante la fase de recuperación se encuentra un token de sincronismo en la entrada.
- **Token-trampa**: Token que puede ser sustituido por una trampa para excepciones.
- **Trampa para excepciones**: Es una regla especialmente diseñada para "soportar" la omisión de un token que se esperaba en una regla, anunciando el error pero recuperándose de él.
- **Type**: Véase tipo.

U.

- **Unicode**: El estándar unicode define un juego de caracteres (letras, símbolos) universal para todos los lenguajes actuales escritos del hombre, en oposición al vetusto ASCII, que incluye un pequeño subconjunto.

V.

- **Variable**: La unidad mínima de información en un lenguaje de programación cualquiera. En los lenguajes llamados fuertemente tipados, una variable tiene además un tipo fijo. Las acciones que se pueden realizar con una variable dependen de su tipo, sea éste fijo o variable.
- **Variable de método**: Es la variable que se crea por defecto para guardar el valor que guarda dicho método.
- **Vocabulario**: Conjunto de símbolos terminales que definen un lenguaje.

Apéndice C: Cuestiones técnicas

Apéndice C: Cuestiones técnicas....435

Sección C.1: Instalación de ANTLR sobre Windows.... 436

Sección C.2: Internacionalización.... 438

C.2.1: El problema.... 438

C.2.2: Defensa de la internacionalización.... 438

C.2.3: Editando los ficheros.... 438

C.2.4: Recompilando.... 439

Sección C.3: Interacción de ANTLR con otros programas.... 441

C.3.1: GNU Emacs/Xemacs.... 441

C.3.2: La plataforma Eclipse.... 442

C.3.3: Otros programas.... 443

Sección C.1: Instalación de ANTLR sobre Windows

En la página oficial de ANTLR (**http://www.antlr.org**) hay un manual de instrucciones bastante intuitivo, que indica paso a paso qué es lo que hay que hacer para instalar ANTLR. No obstante voy a presentar un resumen en castellano.

Para instalar ANTLR sobre Windows:

1. Si no lo ha hecho ya, instale algún compilador de java, por ejemplo jsdk. Recomiendo situarlo en un directorio de fácil escritura, como `c:\jsdk`. *Lo verdaderamente importante es que no haya espacios en el nombre.*
2. Si no lo ha hecho ya, edite el fichero c:\autoexec.bat (o ejecute la aplicación msconfig y utilice la pestaña de autoexec.bat) y compruebe que la variable `PATH` contiene el directorio actual ('.') y el directorio `c:\jsdk\bin` (si `c:\jsdk` fue el directorio elegido para instalar jsdk). Configure adecuadamente la variable `CLASSPATH`. Esto le permitirá compilar fácilmente desde la línea de comandos. El código de su fichero `AUTOEXEC.BAT` debería de parecerse a lo siguiente:

```
mode con codepage prepare=((850) C:\WINDOWS\COMMAND\ega.cpi)
mode con codepage select=850
keyb sp,,C:\WINDOWS\COMMAND\keyboard.sys

SET PATH="c:\jsdk\bin;."

SET CLASSPATH=".";
SET CLASSPATH=%CLASSPATH%;"C:\jsdk\jre\lib\rt.jar"
SET CLASSPATH=%CLASSPATH%;"C:\jsdk\lib\dt.jar"
SET CLASSPATH=%CLASSPATH%;"C:\jsdk\lib\tools.jar"
SET CLASSPATH=%CLASSPATH%;"C:\jsdk\jre\lib\ext\dnsns.jar"
SET CLASSPATH=%CLASSPATH%;"C:\jsdk\jre\lib\ext\ldapsec.jar"
SET CLASSPATH=%CLASSPATH%;"C:\jsdk\jre\lib\ext\localedata.jar"
SET CLASSPATH=%CLASSPATH%;"C:\jsdk\jre\lib\ext\sunjce_provider.jar"
```

3. Descargar la versión que se desee (recomiendo la más reciente) de ANTLR desde **http://www.antlr.org**
4. ANTLR vendrá comprimido en un fichero zip o jar. Descomprima dicho fichero (los ficheros jar son ficheros zip, puede cambiarles la terminación tranquilamente) en algún directorio fácilmente escribible, como `c:\antlr272`[89].
5. Añada el directorio de antlr a la variable CLASSPATH. Por ejemplo, en el fichero autoexec.bat expuesto un poco más arriba, debería añadir algo como esto:

```
SET CLASSPATH=%CLASSPATH%;C:\antlr272
```

6. Reinicie el sistema.

Tras ejecutar todos estos pasos, para compilar un fichero de definición de gramática desde la línea de comandos (pongamos, por ejemplo, `grammar.g`) solamente tendrá que escribir lo siguiente:

89 Yo en realidad utilizo c:\programs\jsdk y c:\home\antlr. Los directorios que indico no son más que ejemplos.

```
c:\>Mis Documentos\java antlr.Tool grammar.g
```

Al descomprimir el fichero comprimido en el directorio antlr272, veremos que dentro del fichero zip o jar hay un directorio llamado antlr. No debemos incluirlo en el classpath.

Eventualmente es posible que al descomprimir también encontremos un fichero jar, llamado por ejemplo `antlr.jar`. Este fichero es un jar pequeño (alrededor de 80 KB) que permite *ejecutar* un compilador java desarrollado con antlr, pero no *compilarlo*. Para más detalles sobre la compilación de ANTLR, consúltese la siguiente sección.

Sección C.2: Internacionalización

C.2.1: El problema

El problema de la internacionalización despareció en la versión 2.7.2 de ANTLR (el juego de caracteres válidos se extendió hasta el 377). No obstante estas indicaciones siguen siendo válidas para versiones anteriores (2.7.2rc2 y previas)

Uno de los primeros problemas (y más fácilmente resolubles) que encontré con ANTLR es la incompatibilidad que ANTLR tiene por defecto con los caracteres no anglosajones (no ASCII).

El problema apareció cuando traté de compilar un fichero de prueba como el siguiente:

```
header {
/* un análisis léxico muy estúpido */
}

class MyLexer extends Lexer;
IDENT : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Al compilarlo con ANTLR obtuve el siguiente mensaje de error:

```
C:\>java antlr.Tool foo.g
ANTLR Parser Generator Version 2.7.2rc2 (20021130-1)1989-2002 jGuru.com
error: Token stream error reading grammar(s):
foo.g:2:8: expecting '*', found 'n'
TokenStreamException: expecting '*', found 'n'
```

Me dí cuenta de que a ANTLR no le gustan los acentos (la línea que da error es la 2, es decir, el comentario, que tiene tres letras acentuadas). Al eliminar dichas letras la gramática funcionaba perfectamente.

C.2.2: Defensa de la internacionalización

Podría discutirse su utilidad: forzando el uso de un conjunto de caracteres tan limitado se está casi obligando a escribir las gramáticas (¡y los comentarios asociados!) en inglés. De esta forma será más probable que alguien de España comprenda una gramática escrita por alguien de Korea del Norte (siempre y cuando los dos hablen inglés). En teoría.

Y digo en teoría porque además de ésto puede ocurrir dos casos más:

1. Que el escritor de la gramática no tenga un nivel de inglés suficiente, siendo su nomenclatura y comentarios ambiguos.
2. Que el escritor escriba la gramática y los comentarios en su lengua materna, sin escribir los caracteres conflictivos. Ésta es sin duda la alternativa más probable.

Además, puede darse el caso de que, como yo, alguien necesite utilizar palabras que contengan caracteres no ASCII (método, parámetro).

En fin, al grano. Vamos a recompilar ANTLR para hacerlo compatible con el castellano.

C.2.3: Editando los ficheros

El rango de caracteres válidos que se utiliza en ANTLR es el ASCII puro, esto es, desde al carácter unicode número 3 hasta el 176. Vamos a cambiar éste rango para llevarlo desde el 3 hasta el 377.

ANTLR es parcialmente auto generado, esto es, algunas clases han sido generadas utilizando uno o varios ficheros *.g con la versión anterior de ANTLR. En la distribución 2.7.2rc2 hay 6 ficheros que deben ser modificados:

- `c:\antlr272RC2\antlr\preprocessor\preproc.g`
- `c:\antlr272RC2\antlr\actions\cpp\action.g`
- `c:\antlr272RC2\antlr\actions\java\action.g`
- `c:\antlr272RC2\antlr\actions\csharp\action.g`
- `c:\antlr272RC2\antlr\antlr.g`
- `c:\antlr272RC2\antlr\tokdef.g`

Algunas de las versiones de antlr que se encuentran en el sitio www.antlr.org son **incompletas**: faltan multitud de ficheros, y entre ellos antlr.g. Si no encuentra algún fichero en su distribución, descargue una diferente.

Lo que tenemos que hacer es simple: buscar la cadena '\3' en los ficheros *.g del directorio `c:\antlr272RC2\antlr` y sus subdirectorios, cambiando el valor de la opción. Es decir, cuando en uno de dichos ficheros encontremos una línea así (`preproc.g` y los tres `action.g`):

```
charVocabulary = '\3'..'\176';
```

Deberemos editarla y cambiarla por algo así:

```
charVocabulary = '\3'..'\377';
```

Y en los ficheros en los que nos encontremos algo como ésto (`antlr.g` y `tokdef.g`):

```
protected
VOCAB
   :  '\3'..'\176'
   ;
```

Vamos cambiar el rango de la siguiente manera:

```
protected
VOCAB
   :  '\3'..'\377'
   ;
```

La regla de oro: ¡siempre haga copias de seguridad antes de recompilar!

C.2.4: Recompilando ANTLR

Nota especial: la forma de recompilar ANTLR se ha modificado entre las versiones 2.7.2rc2 y la 2.7.2. En esta última ANTLR incorpora una herramienta de auto-compilación (paquete antlr.build). De todas maneras la forma de recompilar explicada en este apartado sigue siendo válida para versiones anteriores.

Una vez modificados los ficheros de generación, es necesario “compilarlos” con antlr para que las nuevas clases java se generen.

Estoy suponiendo que ha instalado adecuadamente java y ANTLR para poder ejecutarlos fácilmente desde la línea de comandos. Revise el principio de este apéndice donde indico claramente cómo hacerlo.

Para ello, sitúese en el directorio de cada fichero modificado y “pásele” antlr. Por ejemplo, para

hacerlo con el primer fichero escriba lo siguiente en la consola:

```
c:\antlr272RC2\antlr\preprocessor\> java antlr.Tool preproc.g
```

Ignore los mensajes de advertencia (en las acciones son muchos). Debe repetir el mismo procedimiento para cada fichero *.g que haya modificado. Lo más sencillo es utilizar el siguiente fichero de ejecución por lotes (debe ejecutarse desde el directorio c:\antlr272RC2\ ):

```
cd antlr
java antlr.Tool antlr.g
java antlr.Tool tokdef.g
cd preprocessor
java antlr.Tool preproc.g
cd ..\actions\cpp
java antlr.Tool action.g
cd ..\csharp
java antlr.Tool action.g
cd ..\java
java antlr.Tool action.g
cd ..\..\..
```

Atención: este fichero no va incluido en la distribución de antlr; debe ser codificado a mano y guardado en el directorio de instalación de antlr (c:\antlr272RC2)

Una vez compilados los ficheros *.g habrá que compilar los ficheros *.java generados. Dependiendo de qué versión de ANTLR estemos utilizando, ésto deberá realizarse de diferentes maneras:

En versiones antiguas de ANTLR (2.7.2rc2 y anteriores) se suministran varios ficheros de procesamiento por lotes (ficheros *.bat) para compilar bajo MS-DOS. `build.bat` es el archivo que permite recompilar toda la

```
c:\antlr272RC2\> build.bat
```

Dependiendo de la máquina que utilice, el proceso puede durar más o menos tiempo. Por la pantalla aparecerán mensajes informativos.

Al final se obtendrá un mensaje de error, por no estar presente el directorio “parseview”. Parseview era un complemento de la antigua implementación de ANTLR que permitía hacer un debug visual del reconocimiento de los textos. Actualmente su desarrollo se ha detenido y ya no es compatible con ANTLR, por lo que no se distribuye con él. Por lo tanto, si el mensaje de error nos inquieta podemos la última línea de build.bat, es decir, ésta:

```
javac -deprecation -verbose  -classpath ..\ parseview\*.javad
```

Una vez modificado el fichero `build.bat` y lanzado (y si todo ha ido bien) obtendremos una versión internacional de ANTLR.

Pero eso ocurre en versiones antiguas de ANTLR. En las nuevas (2.7.2 y superiores) se ha implementado un paquete de “auto-compilación” llamado `antlr.build`. Así, para recompilar ANTLR habrá que escribir lo siguiente:

```
c:\> java antlr.build.Tool build
```

Si todo ha ido bien, la compilación debería realizarse sin contratiempos.

Para más información sobre recompilación de ANTLR, consúltese la documentación que acompaña a su distribución de ANTLR.

Sección C.3: Interacción de ANTLR con otros programas

Algunos programadores encontramos engorroso trabajar con aplicaciones en modo consola de texto en los sistemas operativos de Microsoft[90]. Por lo tanto buscamos otras opciones, como integrar dichas herramientas con IDEs y editores de texto. En éste subapéndice voy a mostrar cómo hacer a ANTLR interactuar con algunos de dichos programas.

C.3.1: GNU Emacs/Xemacs

Presentación

GNU Emacs es un editor de ficheros de texto muy conocido en el entorno Linux. Es un programa desarrollado en Lisp y distribuido libremente por la plataforma GNU, de manera que puede descargarse gratuitamente desde su página oficial (**http://www.gnu.org/software/emacs/emacs.html**).

Los detractores de GNU Emacs le echan en cara su lentitud (por estar programado en Lisp) y su poca innovación (pues su aspecto y funcionalidad apenas han cambiado desde su creación). La "lentitud" se ha solucionado aumentando la velocidad de los procesadores actuales; la innovación se solucionó con la versión "en ventanas", llamada Xemacs (**http://www.xemacs.org**). Xemacs incluye una organización de menús más intuitiva y un entorno realmente gráfico, en oposición a GNU Emacs, que continúa estando en modo texto.

Tanto GNU Emacs como Xemacs tienen una forma de personalización similar, por lo que en adelante utilizaré el término "Emacs" para referirme a cualquiera de los dos, diferenciando entre ellos cuando sea necesario.

Mucha gente desconoce que aunque Emacs sea un editor usado sobre todo sobre Linux, existen versiones para windows.

Funcionalidades añadidas

Como ya he dicho, Emacs es un programa desarrollado en Lisp. El método para ampliarlo es simple: se utilizan ficheros escritos en lenguaje Lisp, que se sitúan en cierto directorio, se compilan y a partir de dicho momento las funcionalidades de dicho fichero (siempre que no haya habido errores de compilación) estarán disponibles.

El fichero que nos servirá para utilizar Emacs en conjunción con ANTLR se llama antlr-mode.el, y puede encontrarse en **http://antlr-mode.sourceforge.net**[91]. Este fichero forma parte de la distribución estándard de Emacs en las últimas versiones del editor (a partir de la 21.1 con GNU Emacs y con cualquier versión de Xemacs).

El modo antlr añade las siguientes funciones al modo normal de funcionamiento de Emacs:

- Identación automática. Emacs realizará automáticamente la inserción de caracteres de tabulación y espacios de acuerdo con el estilo de codificación de ANTLR: solamente hay que presionar *tab*. La identación se realiza automáticamente si se inserta cualquier símbolo de sintaxis de ANTLR (es decir, alguno de éstos :.;|(){}). Use el menú para identar todas las líneas de la región señalada.
- *Syntax Highlighting*. Cuando se activa, este modo resalta los símbolos de gramática y el

90 Probablemente porque la consola de texto ofrecida por esta compañía y su forma de configuración (editando autoexec.bat) son las peores del mercado; han permanecido virtualmente intactas desde el MS-DOS.

91 A partir de la versión 2.7.2 de ANTLR puede encontrarse en el directorio "extras" de la distribución.

código de las acciones.

- Navegación por el código. Encuentre la definición de una regla, token o clase de gramática simplemente utilizando el menú Index, un comando de teclado o eligiendo una entrada en el *frame* speedbar.
- Navegación rápida. Muévase a la regla siguiente o anterior o al principio o final de la definición de la regla actual pulsando un par de teclas.
- Visibilidad de acciones. Controle la visibilidad de las acciones menos importantes.
- Ayuda para las opciones. Inserte o cambie una de las opciones de ANTLR utilizando el menú o un comando de teclado. Este modo conoce las opciones soportadas por ANTLR, sus valores permitidos y la posición correcta donde insertar la opción (y añade options{} si es necesario).
- Invocación de la herramienta. Es posible invocar directamente ANTLR desde Emacs y encontrar la posición del error eligiendo el error con el ratón o un comando de teclado.
- Generación de makefile. Enseñar las dependencias de makefile para todas las gramáticas y ficheros generados en el directorio actual.

Instalación de antlr-mode

En caso de no tener por defecto instalado el fichero antlr-mode, o de que se quiera instalar una nueva versión de dicho fichero, es posible instalarlo manualmente. Aunque en la página oficial de antlr-mode se expone claramente cómo hacerlo, voy a transcribir el proceso de instalación en castellano:

1. Copiar el fichero antlr-mode.el dentro del *load path* de Emacs[92].
2. Compile dicho fichero (para ello use el menú **Build->Byte-compile this file**).
3. Insertar las siguientes líneas en el código en el fichero de configuración de Emacs:

```
(autoload 'antlr-mode "antlr-mode" nil t)
(setq auto-mode-alist (cons '("\\.g\\'" . antlr-mode) auto-mode-alist))
(add-hook 'speedbar-load-hook  ; would be too late in antlr-mode.el
          (lambda () (speedbar-add-supported-extension ".g")))
```

4. Para personalizar Emacs, utilice el submenú **Major Mode->ANTLR->Customize Antlr**.

C.3.2: La plataforma Eclipse

Presentación

La plataforma Eclipse (sitio oficial: **http://www.eclipse.org**) es la respuesta que Sun Microsistems ha dado a Microsoft Visual Studio. Aunque ambos entornos son visualmente parecidos, son completamente diferentes tanto a nivel programático como administrativo.

Eclipse es open source, así que puede descargarse gratuitamente desde su página oficial.

Eclipse es un programa basado en ventantas y programado en java. Yo pensé sería intolerablemente lento trabajar con él, pero me equivoqué completamente; Eclipse ha desarrollado un conjunto de librerías gráficas (diferentes de SWING y AWT) muy rápidas. ¡Ya era hora!

La plataforma Eclipse está disponible para varios sistemas operativos, entre ellos Microsoft Windows.

92 Normalmente el load-path de Emacs es algo parecido a c:\Directorio-de-instalación-de-Emacs\site-lisp

Funcionalidades añadidas

Eclipse es una plataforma altamente personalizable, mediante paquetes java que se denominan *plugins*. Estos paquetes se copian directamente en el subdirectorio plugins del directorio de instalación.

Esta política de plugins ha proporcionado a eclipse una flexibilidad enorme (muchísimo mayor que la de Visual Studio, que no se hizo con la personalización en mente). Aunque inicialmente Eclipse estuviera enfocada al desarrollo en Java (el modo java es un plugin más), hayan aparecido plugins para desarrollo en otros lenguajes, como C/C++.

Uno de estos nuevos plugins es el plugin para antlr, que puede encontrarse en **http://antlreclipse.sourceforge.net/**. Es un producto aún joven, con muchos aspectos por desarrollar (he encontrado algunos bugs). La velocidad de actualización no es tan rápida como sería deseable.

El plugin añade las siguientes funcionalidades a Eclipse:

- Nuevo tipo de proyecto: desarrollo ANTLR.
- Marcadores para errores en los ficheros de gramáticas.
- *Syntax highlighting*.
- Vista en esquema (*outline*) del fichero de gramática.
- Distinción entre ficheros generados automáticamente por ANTLR y ficheros java escritos por otros métodos.
- Encontrar definiciones de reglas (puede hacerse pulsando F3).

Instalación

La instalación del paquete es sencilla: tras descargar el *plugin* (en realidad son tres paquetes diferentes) de internet, han de descomprimirse en el directorio *plugins* de la plataforma eclipse. A partir de entonces, cuando se reinicie Eclipse el *plugin* estará activo.

Una vez activado el *plugin*, será necesario configurar manualmente cierta propiedad que tiene el compilador para evitar que los ficheros de gramáticas se copien en el directorio de salida de los ficheros compilados. Para ello, en la ventana de Preferencias, añadir la cadena "*.g" en Java/Compiler/Pestaña Other/Caja Filtered resources.

Un problema importante del *plugin* de antlr para Eclipse es que incluye su propia versión de ANTLR en uno de sus paquetes. Si se desea utilizar una versión diferente de ANTLR (o internacionalizarlo) será necesario importar el plugin org.antlr en forma de proyecto Eclipse y modificar los ficheros que sea necesario, recompilando, etc.

C.3.3: Otros

Es posible hacer que casi cualquier otro editor o IDE con capacidad para compilar ficheros se integre con ANTLR, de la misma forma que se haría con bison y c++. La técnica más evidente consiste en configurar los ficheros de definición de gramática para ser compilados de una forma "personalizada" (llamando a la línea de comandos; esto ocurre incluso con Visual Studio, por eso es incomprensible que sigan usando la horrorosa consola de MS DOS).

En el caso de que no se puedan crear dependencias entre ficheros, pueden hacerse dos proyectos diferentes, uno para generar los ficheros a partir de las gramáticas, y otro para compilar en java, y

finalmente compilar *en batch* primero uno y luego el otro. Esto es lo que estoy haciendo yo con el editor que utilizo actualmente, jcreator.

Muchos editores (entre ellos jcreator) permiten especificar en ficheros de texto muy fáciles de entender cómo se colorea la gramática de los lenguajes que soportan. De esta forma he conseguido que jcreator coloree adecuadamente mis ficheros de gramáticas.

En cuanto al resto de prestaciones (navegación, visibilidad) no creo que haya ninguna manera de simularlas con programas que no estén desde el principio pensados para ello, o que no puedan extenderse con plugins. De todas formas éstas son las prestaciones que menos útiles me parecen.

La versión 2.7.2 de ANTLR incorpora el módulo de interacción con el editor jedit y con GNU Emacs/Xemacs. Se encuentran en el directorio “extras”.

Apéndice D: Contenido del CD-ROM

Directorio “software”

- El programa de instalación de la **plataforma JSDK 1.4.1** para Windows.
- La **documentación de la API de JSDK 1.4.1** comprimida en formato ZIP.
- La última versión disponible de la **plataforma Eclipse** para Windows.
- La última versión disponible de **Xemacs** para Windows.
- La última versión disponible de **OpenOffice.org** para Windows.
- La última versión disponible de **Adobe Acrobat Reader** para Windows.
- La última versión disponible de **WinRAR**.

Directorio “antlr”

- La versión de **ANTLR** que se ha utilizado en este documento (2.7.2)
- La última versión del **modo antlr para Eclipse** disponible en **http://antlreclipse.sourceforge.net**
- La última versión del **modo antlr para Emacs/Xemacs** disponible en **http://antlr-mode.sourceforge.net**

Directorio “antlraux”

- La última versión de la **librería antlraux** (binarios + código fuente).
- Documentación en formato **javadoc** de dicha librería.

Directorio “leli”

- El código fuente (ficheros *.java y *.g) que hemos creado para interpretar el lenguaje LeLi.
- Los ejecutables (ficheros *.class) del intérprete.
- El fichero `tiposBasicos.leli`
- Algunos ficheros *.leli de ejemplo

Directorio “doc”:

- Este documento en formato OpenOffice.org 1.1 Writer (*.sxw)
- Este documento en formato PDF (*.pdf)