



ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS



Fco. Javier Bohórquez Ogalla

10 de mayo de 2016



ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS



- Departamento: Ingeniería Informática.
- Director del proyecto: Iván Ruiz Rube
- Autor del proyecto: Fco. Javier Bohórquez Ogalla

Cádiz, 10 de mayo de 2016

Fdo: Fco. Javier Bohórquez Ogalla

Agradecimientos

Este proyecto no habría sido posible sin la ayuda y apoyo de mi novia Sandra Fernández Domínguez, su trabajo y esfuerzo han marcado este proyecto. Gracias por estar a mi lado durante todo el duro camino que ha supuesto terminar este trabajo.

Agradecer también a mis padres y mi hermano por el apoyo y ánimos que siempre me han dado.

El apoyo y criterio de mis compañeros de trabajo ha sido una pieza clave para finalizar este proyecto, en especial de Luis Manuel Vaca dado que sus directivas gráficas me han marcado un camino para un diseño cuidado.

Resumen

El proyecto OMI tiene como razón de ser servir como apoyo en el aprendizaje de los conceptos detrás de los sistemas intérpretes. Se vale de la teoría de autómatas y los lenguajes formales para construir una serie de herramientas que ayudan en el proceso de estudio de estos sistemas. El proyecto lo conforman una serie de herramientas y recursos enmarcados dentro del campo de los intérpretes y lenguajes formales.

OMI es un sistema intérprete del lenguaje de programación con el mismo nombre. El intérprete es de código abierto y modular. Define un lenguaje orientado a objetos, de tipado dinámico y con muchas otras características presentes en los lenguajes de programación modernos. Su objetivo es ilustrar el proceso de desarrollo y el funcionamiento interno que presenta un sistema intérprete para un lenguaje de programación.

El proyecto OMI presenta una serie de herramientas webs que permiten hacer uso del intérprete de forma online desde el navegador, detallándose los procesos sintácticos y semánticos llevados a cabo. Además permiten navegar por toda la documentación de desarrollo del proyecto. El proyecto pone a disposición de los desarrolladores la biblioteca de clases y funciones sobre las que se escribe. Esta puede ser utilizada para la construcción de módulos o ser incluida en otros proyectos en los que se desee hacer uso del intérprete.

OMI es un proyecto que pretende acercar el mundo de los intérpretes y los lenguajes formales, mediante el uso de herramientas gráficas e ilustrativas, y facilitando una documentación desarrollada y precisa. El proyecto consiste en el diseño de un lenguaje de programación y el desarrollo del intérprete asociado. El sistema software puede ser construido y ejecutado para que recibir código fuente y producir una salida correspondiente al proceso de interpretación, así otras herramientas, como las ofrecidas por el proyecto OMI, podrán usar esta capacidad para ilustrar el funcionamiento interno del intérprete e interactuar con él.

Palabras clave: Intérprete, lenguaje formal, autómata, didáctico, ilustrativo, gramática, analizador léxico, analizador sintáctico, ejecución semántica, biblioteca de desarrollo .

Índice general

I Prolegómeno	1
1. Introducción	3
1.1. Motivación	3
1.2. Alcance	4
1.2.1. Intérprete	4
1.2.2. Lenguaje	5
1.2.3. Módulos	6
1.2.4. Biblioteca de desarrollo	6
1.2.5. Sitio web	7
1.2.6. RunTree	7
1.3. Organización del documento	8
2. Estado del arte	9
2.1. Glosario de términos	9
2.1.1. Analizador léxico	9
2.1.2. Analizador sintáctico	9
2.1.3. Árbol sintáctico	9
2.1.4. Argumento	10

2.1.5.	Biblioteca de programación	10
2.1.6.	Biblioteca dinámica	10
2.1.7.	Biblioteca estática	10
2.1.8.	Clase	10
2.1.9.	Código fuente	11
2.1.10.	Comando	11
2.1.11.	Compilador	11
2.1.12.	Constante	11
2.1.13.	Dato	11
2.1.14.	Expresión	11
2.1.15.	Extensión	12
2.1.16.	Función	12
2.1.17.	Gramática	12
2.1.18.	Gramática libre de contexto	12
2.1.19.	Identificador	13
2.1.20.	Interprete	13
2.1.21.	Instrucción	13
2.1.22.	Lenguaje de programación	13
2.1.23.	Lenguaje de programación compilados	13
2.1.24.	Lenguaje de programación interpretados	14
2.1.25.	Lenguaje de programación de alto nivel	14
2.1.26.	Lenguaje de programación de bajo nivel	14
2.1.27.	Lenguaje de programación de tipado dinámico	14
2.1.28.	Lenguaje de programación de tipado estático	14

2.1.29. Módulo	14
2.1.30. Objeto	15
2.1.31. Operador	15
2.1.32. Paradigma de programación	15
2.1.33. Parámetro	15
2.1.34. Procedimiento	15
2.1.35. Programa	16
2.1.36. Programación declarativa	16
2.1.37. Programación imperativa	16
2.1.38. Script	16
2.1.39. Sentencia	16
2.1.40. Signatura	17
2.1.41. Tipo array	17
2.1.42. Tipo cadena de caracteres	17
2.1.43. Tipo expresión regular	17
2.1.44. Tipo lógico	17
2.1.45. Tipo numérico	18
2.1.46. Tipos de datos	18
2.1.47. Token	18
2.1.48. Variable	18
2.2. Análisis previo de mercado	18
2.2.1. Compilado o interpretado	19
2.2.2. Sistema de tipos	19
2.2.3. Tipos de datos	20

2.2.4.	Paradigmas de programación	21
2.2.5.	Operadores	28
2.2.6.	Funciones y clases de objetos	28
2.2.7.	Definición de lenguajes específicos de dominio	29
2.2.8.	Depuración	30
2.2.9.	Autodocumentación del proceso de interpretación	30
2.2.10.	Otras características funcionales	30
2.2.11.	Características no funcionales	31
3.	Planificación	35
3.1.	Metodología de desarrollo	36
3.2.	Planificación	37
3.2.1.	Diagramas de Gantt de las iteraciones generales	37
3.3.	Organización	39
3.4.	Costes	40
3.5.	Riesgos	41
3.5.1.	Riesgos tecnológicos	42
3.5.2.	Riesgos de requisitos	43
3.5.3.	Riesgos de soluciones	44
3.5.4.	Riesgos de costes, tiempos y recursos	46
3.6.	Aseguramiento de calidad	47
II	Desarrollo	49
4.	Requisitos del Sistema	53

4.1. Visión general	53
4.2. Situación actual	53
4.3. Necesidades	54
4.4. Objetivos	55
4.5. Requisitos funcionales	56
4.5.1. Intérprete	56
4.5.2. Ejecución	59
4.5.3. Tipos de datos	61
4.5.4. Sentencias	63
4.5.5. Definiciones	68
4.5.6. Asignaciones	78
4.5.7. Operadores aritméticos	79
4.5.8. Operadores lógicos	84
4.5.9. Operadores sobre cadenas	88
4.5.10. Operadores sobre array	91
4.5.11. Operadores sobre expresiones regulares	94
4.5.12. Operadores de conversión de tipo	95
4.5.13. Operadores de entrada/salida	96
4.5.14. Leer de la entrada estándar	96
4.5.15. Operadores informativos	96
4.5.16. Procesos	97
4.5.17. Ficheros	99
4.5.18. Fechas y tiempo	102
4.5.19. Errores	104

4.5.20. Extensiones	104
4.5.21. Biblioteca de desarrollo OMI	105
4.5.22. Web del proyecto OMI	106
4.5.23. RunTree	109
4.6. Requisitos no funcionales	112
4.6.1. Rendimiento	112
4.6.2. Usabilidad	112
4.6.3. Accesibilidad	113
4.6.4. Estabilidad	113
4.6.5. Mantenibilidad	113
4.6.6. Concurrencia	113
5. Análisis del Sistema	115
5.1. Modelo conceptual	115
5.1.1. Intérprete	116
5.1.2. Nodos ejecutables	117
5.1.3. Tipos de datos	118
5.1.4. Sentencias de control de flujo	119
5.1.5. Definiciones	121
5.1.6. Asignaciones	123
5.1.7. Operadores aritméticos	124
5.1.8. Operadores lógicos	126
5.1.9. Operadores sobre cadenas	128
5.1.10. Operadores sobre array	129
5.1.11. Operadores sobre expresiones regulares	130

5.1.12. Conversión de tipos	131
5.1.13. Operadores de acceso	131
5.1.14. Operadores condicionales	132
5.1.15. Operadores de entrada/salida	132
5.1.16. Operadores informativos	133
5.1.17. Procesos	134
5.1.18. Ficheros	135
5.1.19. Fechas	137
5.1.20. Errores	137
5.1.21. Extensiones	138
5.1.22. rTree	139
5.2. Modelo de casos de uso	140
5.2.1. Actores	140
5.2.2. Intérprete	141
5.2.3. runTree	146
5.3. Comportamiento del sistema	151
5.4. Diagramas de secuencia del sistema	152
5.4.1. Intérprete	152
5.4.2. runTree	153
5.5. Contratos de operaciones del sistema	155
5.5.1. Intérprete	156
5.5.2. runTree	162
5.6. Modelo de interfaz de usuario	167
5.6.1. Intérprete	167

5.6.2. runTree	167
5.6.3. Sitio web	168
6. Diseño del Sistema	171
6.1. Arquitectura del sistema	171
6.1.1. Arquitectura física	171
6.1.2. Arquitectura lógica	172
6.2. Diseño de la gramática	180
6.2.1. Programa	181
6.2.2. Sentencias	181
6.2.3. Expresiones	186
6.2.4. Identidades	196
6.3. Diseño de comunicaciones	199
6.3.1. Esquema JSON	199
6.4. Diseño de componentes	201
6.4.1. Interpretar código fuente	202
6.4.2. Sentencias	203
6.5. Diseño de la interfaz de usuario	205
6.5.1. Intérprete	205
6.5.2. runTree	205
6.5.3. Sitio web	206
7. Construcción del Sistema	209
7.1. Entorno de construcción	209
7.2. Ficheros de código fuente	210

7.3. Código fuente	215
7.3.1. Acceso a variable	215
7.3.2. Asignación	215
7.3.3. Operación suma	216
8. Pruebas del Sistema	217
8.1. Estrategia	217
8.2. Entorno de pruebas	217
8.2.1. Hardware	218
8.2.2. Software	218
8.3. Roles	218
8.4. Niveles de pruebas	218
8.4.1. Pruebas unitarias	218
8.4.2. Pruebas de integración	237
8.4.3. Pruebas funcionales del sistema	239
8.4.4. Pruebas no funcionales del sistema	240
III Epílogo	247
9. Manual de implantación y explotación	251
9.1. Introducción	251
9.2. Requisitos previos	251
9.3. Inventario de componentes	251
9.4. Procedimiento de instalación	251
9.5. Pruebas de implantación	253

9.6. Procedimientos de operación y nivel de servicio	253
10. Manual de usuario	255
10.1. Características	255
10.2. Requisitos previos	257
10.3. Uso del sistema	257
10.3.1. Obtener OMI	257
10.3.2. Instalación	257
10.3.3. Intérprete	261
10.3.4. Referencias del lenguaje	261
10.3.5. Funciones del lenguaje	318
10.3.6. Extensiones del lenguaje	341
10.3.7. Arquitectura cliente/servidor	343
10.3.8. Descripción del proceso de interpretación	343
10.3.9. runTree	343
11. Conclusiones	345
11.1. Objetivos alcanzados	345
11.2. Lecciones aprendidas	345
11.3. Trabajo futuro	346
12. Anexos	347
12.1. Calculadora	347
12.2. Cuestionarios	347
12.3. Tic-Tac-Toe	350
12.4. Fibonacci	354

12.5. N-body	354
------------------------	-----

Bibliografía	357
---------------------	------------

Información sobre Licencia	359
-----------------------------------	------------

Índice de figuras

3.1. Planificación general 01	37
3.2. Planificación general 02	38
3.3. Planificación general 03	38
3.4. Planificación general 04	38
3.5. Planificación general 05	39
3.6. Planificación general 06	39
3.7. Planificación general 07	39
5.1. Paquetes OMI	116
5.2. Clases intérprete	117
5.3. Clases nodos ejecutables	118
5.4. Clases tipos de datos	118
5.5. Paquetes sentencias de control de flujo	119
5.6. Clases sentencia	119
5.7. Clases sentencia vacía	119
5.8. Clases include	119
5.9. Clases exit	119
5.10. Clases goto	119
5.11. Clases if	119

5.12. Clases switch	120
5.13. Clases while	120
5.14. Clases do...while	120
5.15. Clases for	120
5.16. Clases foreach	120
5.17. Clases iloop	120
5.18. Clases continue	120
5.19. Clases break	120
5.20. Clases try	120
5.21. Clases throw	120
5.22. Clases witch	121
5.23. Paquetes definiciones	121
5.24. Clases sobre variables	121
5.25. Clases sobre funciones	122
5.26. Clases sobre clases	122
5.27. Clases sobre objetos	123
5.28. Clases sobre listas	123
5.29. Clases sobre pares clave/valor	123
5.30. Etiquetas	123
5.31. Definiciones globales	123
5.32. Clases sobre generadores	123
5.33. Paquetes asignaciones	123
5.34. Clases asignación	124
5.35. Clases asignación de referencia	124

5.36. Paquetes operadores aritméticos	124
5.37. Clases suma	124
5.38. Clases diferencia	124
5.39. Clases producto	124
5.40. Clases división	124
5.41. Clases potencia	125
5.42. Clases módulo	125
5.43. Clases tamaño	125
5.44. Clases incremento y asignación	125
5.45. Clases asignación e incremento	125
5.46. Clases decremento y asignación	125
5.47. Clases asignación y decremento	126
5.48. Paquetes operadores lógicos	126
5.49. Clases or	126
5.50. Clases and	126
5.51. Clases negación	126
5.52. Clases igual que	126
5.53. Clases distinto que	127
5.54. Clases menor que	127
5.55. Clases menor o igual que	127
5.56. Clases mayor que	127
5.57. Clases mayor o igual que	127
5.58. Clases idéntico a	127
5.59. Clases no idéntico a	127

5.60. Clases es nulo	127
5.61. Clases vacío	127
5.62. Paquetes operadores de cadenas	128
5.63. Clases concatenación	128
5.64. Clases explode	128
5.65. Clases implode	128
5.66. Clases sprintf	128
5.67. Clases buscar subcadena	128
5.68. Clases buscar y remplazar	128
5.69. Clases remplazar subcadena	128
5.70. Clases convertir a mayúsculas	128
5.71. Clases convertir a minúsculas	129
5.72. Paquetes operadores sobre array	129
5.73. Clases dividir en fragmentos	129
5.74. Clases reducir mediante función	129
5.75. Clases obtener último	129
5.76. Clases obtener primero	129
5.77. Clases insertar en posición	129
5.78. Clases eliminar posición	129
5.79. Clases insertar al inicio	130
5.80. Clases insertar al final	130
5.81. Clases eliminar del inicio	130
5.82. Clases eliminar del final	130
5.83. Paquetes operadores sobre expresiones regulares	130

5.84. Clases crear expresión regular	130
5.85. Clases match	130
5.86. Clases search	130
5.87. Paquetes conversión de tipos	131
5.88. Clases conversión a lógico	131
5.89. Clases conversión a entero	131
5.90. Clases conversión a flotante	131
5.91. Clases conversión a cadena	131
5.92. Paquetes operadores de acceso	131
5.93. Clases acceso a clave	131
5.94. Clases acceso a función	131
5.95. Clases acceso a variable de entorno	132
5.96. Paquetes Operadores condicionales	132
5.97. Clases ternario	132
5.98. Clases fusión de nulos	132
5.99. Paquetes operadores de entrada/salida	132
5.100 Clases salida estándar	133
5.101 Clases entrada estándar	133
5.102 Paquetes operadores informativos	133
5.103 Clases tipo de	133
5.104 Clases tamaño de	133
5.105 Clases información sobre	133
5.106 Paquete de procesos	134
5.107 Clases crear proceso	134

5.108Clases esperar finalización	134
5.109Clases obtener identificador	134
5.110Clases obtener identificador padre	134
5.111Clases ejecutar como proceso	134
5.112Clases salir de proceso	134
5.113Clases señal a proceso	135
5.114Clases manejador de señales	135
5.115Clases evaluar cadena	135
5.116Clases ejecutar comando del sistema	135
5.117Paquetes de ficheros	135
5.118Clases obtener un flujo a fichero	135
5.119escribir en flujo a fichero	135
5.120Clases leer de flujo a fichero	136
5.121Clases cambiar posición en fichero	136
5.122Clases obtener posición en flujo a fichero	136
5.123Clases cerrar flujo a fichero	136
5.124Clases leer fichero	136
5.125Clases escribir en fichero	136
5.126Clases escribir al final de fichero	136
5.127Paquetes de fechas	137
5.128Clases tiempo Unix	137
5.129Clases fecha y hora con formato	137
5.130Clases sleep	137
5.131Clases Errores	137

5.132	Paquete de extensiones	138
5.133	Clases plugins	138
5.134	Paquetes de Biblioteca GNU de internacionalización (gettext)	138
5.135	Clases gettext	139
5.136	Clases setlocale	139
5.137	Clases dgettext	139
5.138	Clases bindtextdomain	139
5.139	Clases textdomain	139
5.140	Clases rTree	140
5.141	Casos de usos intérprete	141
5.142	Casos de usos runTree	146
5.143	Secuencia interpretar entrada estándar	152
5.144	Secuencia interpretar línea	152
5.145	Secuencia interpretar fichero	152
5.146	Secuencia ver ayuda	152
5.147	Secuencia cargar extensión	153
5.148	Secuencia listar extensiones	153
5.149	Secuencia iniciar interpretación red	153
5.150	Secuencia obtener pasos interpretación red	153
5.151	Secuencia enviar código fuente	153
5.152	Secuencia siguiente paso	153
5.153	Secuencia siguiente sentencia	154
5.154	Secuencia activar ejecución automática	154
5.155	Secuencia desactivar ejecución automática	154

5.156 Secuencia limpiar salida	154
5.157 Secuencia ver información de nodo	154
5.158 Secuencia ver contenido tabla de símbolo	155
5.159 Secuencia guardar código fuente	155
5.160 Secuencia abrir código fuente	155
5.161 Contrato de operaciones intérprete	156
5.162 Contrato de operaciones runTree	162
5.163 Interfaz de usuario runTree	168
5.164 Interfaz de usuario web	170
6.1. Arquitectura lógica	172
6.2. Arquitectura analizador sintáctico	175
6.3. Arquitectura runTree	179
6.4. Niveles reglas de producción	180
6.5. Gramática programa	181
6.6. Gramática sentencias	181
6.7. Gramática sentencias de bloque	182
6.8. Gramática if	182
6.9. Gramática elif	182
6.10. Gramática while	183
6.11. Gramática dowhile	183
6.12. Gramática for	183
6.13. Gramática foreach	183
6.14. Gramática switch	184
6.15. Gramática cases	184

6.16. Gramática iloop	184
6.17. Gramática acceso iloop	184
6.18. Gramática trycatch	184
6.19. Gramática with	185
6.20. Gramática stmt	185
6.21. Gramática etiquetas	185
6.22. Gramática nombres de espacios	186
6.23. Gramática clases	186
6.24. Gramática métodos y atributos	186
6.25. Gramática expresiones	186
6.26. Gramática or lógico	187
6.27. Gramática and lógico	187
6.28. Gramática negación lógica	187
6.29. Gramática comparaciones	187
6.30. Gramática operadores aritméticos	188
6.31. Gramática producto y división	188
6.32. Gramática potencia y módulo	188
6.33. Gramática operadores cadenas de caracteres	188
6.34. Gramática flujo	189
6.35. Gramática llamadas	189
6.36. Gramática operadores condicionales	189
6.37. Gramática operador ternario	189
6.38. Gramática fusión de nulos	189
6.39. Gramática operadores unitarios	190

6.40. Gramática incrementos y decrementos	190
6.41. Gramática conversión de tipos	191
6.42. Gramática accesos	191
6.43. Gramática asignaciones	191
6.44. Gramática funciones	191
6.45. Gramática función lambda	192
6.46. Gramática función parcial	192
6.47. Gramática función de contexto	192
6.48. Gramática decoradores	192
6.49. Gramática decorador lambda	193
6.50. Gramática operadores clases y objetos	193
6.51. Gramática funciones del lenguaje	193
6.52. Gramática funciones aritméticas	193
6.53. Gramática funciones cadenas de caracteres	194
6.54. Gramática funciones array	194
6.55. Gramática funciones expresiones regulares	194
6.56. Gramática funciones fechas y tiempo	195
6.57. Gramática funciones acceso a entorno	195
6.58. Gramática fpes	195
6.59. Gramática file	195
6.60. Gramática funciones procesos	196
6.61. Gramática generadores	196
6.62. Gramática identidades	196
6.63. Gramática valores de parámetros	197

6.64. Gramática parámetros	197
6.65. Gramática pares	197
6.66. Gramática lista	197
6.67. Gramática diccionario	197
6.68. Gramática identificador	198
6.69. Gramática números	198
6.70. Gramática cadena de caracteres	198
6.71. Gramática array	198
6.72. Gramática expresión regular	199
6.73. Gramática comentarios	199
6.74. Interacción interpretar código fuente	202
6.75. Objetos sentencia condicional	203
6.76. Objetos operaciones aritméticas	203
6.77. Objetos asignaciones	204
6.78. Objetos asignaciones (2)	204
6.79. Objetos asignaciones (3)	204
6.80. Interfaz de usuario runTree	206
6.81. Interfaz de usuario home	206
6.82. Interfaz de usuario sobre OMI	207
6.83. Interfaz de usuario contacto	207
6.84. Interfaz de usuario índice de la documentación	207
6.85. Interfaz de usuario documento	207
6.86. Interfaz de usuario navegador de gramática	208
6.87. Interfaz de usuario navegador de clases	208

6.88. Interfaz de usuario navegador de ficheros	208
6.89. Interfaz de usuario descargas	208
7.1. Ficheros intérprete	211
7.2. Ficheros error	212
7.3. Ficheros plugin	212
7.4. Ficheros runTree	212
7.5. Ficheros expNode	212
7.6. Ficheros sTable	212
7.7. Ficheros symbols	212
7.8. Ficheros typeData	213
7.9. Ficheros stmtNode	213
7.10. Ficheros operatorBaseNode	213
7.11. Ficheros operatorLogicNode	213
7.12. Ficheros operatorArithNode	213
7.13. Ficheros	213
7.14. Ficheros operatorArrayNode	214
7.15. Ficheros operatorRegexpNode	214
7.16. Ficheros operatorDatteNode	214
7.17. Ficheros operatorFileNode	214
7.18. Ficheros operatorProcessNode	214
8.1. Espacio de memoria de nodos lógicos	243
8.2. Espacio de memoria de nodos aritméticos	244
8.3. Espacio de memoria de nodos de cadenas de caracteres	245

12.1. Diseño cuestionarios	348
--------------------------------------	-----

Índice de tablas

2.1. Lenguajes compilados e interpretados	19
2.2. Lenguajes según sistema de tipos	20
2.3. Lenguajes según tipo de datos soportados	21
2.4. Lenguajes imperativos	22
2.5. Lenguajes orientados a objetos	23
2.6. Lenguajes orientados a objetos basados en prototipos	23
2.7. Lenguajes dirigidos por eventos	24
2.8. Lenguajes basados en autómatas	24
2.9. Lenguajes orientados a aspectos	25
2.10. Lenguajes declarativos	25
2.11. Lenguajes funcionales	26
2.12. Lenguajes lógicos	26
2.13. Lenguajes dirigidos por restricciones	27
2.14. Lenguajes concurrentes	27
2.15. Lenguajes distribuidos	28
2.16. Lenguajes según operadores	28
2.17. Lenguajes según DSL	29
2.18. Lenguajes según depuración	30

2.19. Lenguajes autodocumentados	30
2.20. Lenguajes según otras características funcionales	31
2.21. Lenguajes según modularidad de recursos	31
2.22. Lenguajes según modularidad del propio lenguaje	32
2.23. Lenguajes según licencia	33
3.1. Salarios	40
3.2. Costes totales	41
3.3. Equipos	41
3.4. Riesgos tecnológicos	42
3.5. Riesgos de requisitos	44
3.6. Riesgos de soluciones	45
3.7. Riesgos de costes, tiempos y recursos	46
8.1. Tiempos Fibonacci OMI	241
8.2. Tiempos Fibonacci PHP	241
8.3. Tiempos Fibonacci Python	241
8.4. Tiempos N-body OMI	241
8.5. Tiempos N-body PHP	241
8.6. Tiempos N-body Python	242

Parte I

Prolegómeno

Capítulo 1

Introducción

El proyecto OMI ofrece un conjunto de herramientas y recursos que ayudan en el aprendizaje de la teoría de intérpretes y los lenguajes formales. OMI en si mismo es un intérprete desarrollado con la finalidad de servir como caso práctico, aplicando los conceptos estudiados en estos campos. Es un intérprete modular, en el sentido de que su funcionalidad y características puede ser extendida mediante módulos. OMI es código abierto y libre, por lo que puede ser utilizado, estudiado, modificado y/o redistribuido libremente.

El nombre OMI también hace referencia al lenguaje que es interpretado. Un lenguaje de programación de alto nivel, de tipado dinámico y de propósito general. Fundamentalmente imperativo orientado a objetos, pero que además presenta características propias de otros paradigmas como de la programación funcional.

El proyecto OMI pretende facilitar el estudio de los sistemas intérpretes mediante la documentación y la interactividad. Abarcando no solo el diseño y el desarrollo de uno de estos sistemas, si no el proceso de interpretación en si. De esta forma se presentan herramientas que permiten hacer uso del intérprete a la vez que ilustran su funcionamiento.

1.1. Motivación

La teoría de autómatas y los lenguajes formales son la base para la construcción de sistemas intérpretes. Muchos de los conceptos estudiados en estos campos sirven para construir otras ramas de estudio como la teoría de intérpretes y compiladores.

Normalmente, por limitaciones de tiempo, los cursos académicos relacionados con el estudio de estos campos están enfocados a los conceptos teóricos. Llevándose a cabo tan solo algunas prácticas que refuerzan el aprendizaje y ayudan a ver la aplicación real mediante casos sencillos y simplificados. Sería de gran ayuda en este proceso disponer de un caso práctico completo, que ilustre cómo se define y construye un lenguaje de programación actual, y que pueda ser con-

sultado cuándo, dónde y por quién lo deseé. OMI pretende ser una fuente de información que complemente el estudio práctico en estas áreas.

Aunque existen herramientas que dan soporte a la construcción de sistemas intérpretes, es difícil encontrar alguna que ayude a comprender cómo estos se construyen y funcionan. Los intérpretes modernos no están enfocados en ilustrar la tarea que llevan a cabo, tienen como propósito ejecutar programas de forma óptima y efectiva. Sería de utilidad disponer de una herramienta que documente los procesos sintácticos y semánticos llevados a cabo durante la interpretación de código fuente. OMI tiene como propósito generar información relativa al proceso de interpretación que muestre el funcionamiento del intérprete.

La comunidad académica no solo precisa de un objeto de estudio, si no también de un objeto que ayude a formar la comunidad. El proyecto OMI pretende ser una base para crear una comunidad centrada en el estudio de los lenguajes formales, en las que todos puedan contribuir a la vez de beneficiarse.

1.2. Alcance

El proyecto OMI abarca una serie de herramientas y recursos que ayudan a comprender cómo se construye y funciona un sistema intérprete.

1.2.1. Intérprete

El intérprete OMI es la parte fundamental del proyecto, las demás herramientas y recursos se valen o se construyen a partir de este. Es un sistema software capaz de analizar y ejecutar otros programas escritos en un lenguaje específico con el mismo nombre.

El intérprete puede procesar código fuente recibido mediante la entrada estándar. También es posible ejecutarlo como una terminal interactiva, mostrando un prompt e interpretando el código introducido. Por otro lado podrá ser configurado para obtener el código fuente mediante un puerto TCP, funcionando así como un software servidor. El software puede recibir una serie de parámetros que serán pasados como argumentos al programa. Además puede ser ejecutado de forma que produzca una salida en formato JSON relativa a acciones llevadas a cabo durante el proceso.

El intérprete puede ser extendido mediante módulos lo que permite ampliar las posibilidades del lenguaje subyacente. Por otro lado el software puede ser configurado desde el momento en el que es construido, de forma que puede prescindir y/o alterar algunas de sus características, adaptándose mejor al uso concreto que se le fuera a dar.

1.2.2. Lenguaje

OMI es un lenguaje multiparadigma de alto nivel y de propósito general, que presenta un tipado dinámico. Su sintaxis pretende ser sencilla y similar a los lenguajes de programación modernos. Contempla tipos de datos simples y compuestos, así como un conjunto de operaciones sobre estos. Los tipos de datos que pueden ser manipulados por el intérprete OMI comprenden:

- Lógicos
- Aritméticos
- Cadena de caracteres
- Array de elementos
- Expresiones regulares
- Funciones
- Referencias
- Objetos

El intérprete OMI soporta variedad de operadores que trabajan sobre los tipos de datos indicados. Estos son aquellos que se pueden encontrar de forma nativa en cualquier lenguaje de programación actual, a excepción de los operadores booleanos bit a bit. Esto incluye operadores tales como la suma, la potencia, la negación, la concatenación, el ternario, fusión de nulos... Se dispone además de una serie de operadores para la entrada/salida de datos.

También permite al usuario asignar valores a símbolos variables, que pueden ser utilizados posteriormente en otras expresiones y construcciones del lenguaje. Las variables pueden ser locales o globales.

El sistema intérprete es capaz de procesar estructuras de control tales como if, while, switch... Además presenta otras sentencias que operan sobre el flujo de ejecución, como por ejemplo la inclusión de ficheros, la rotura de bloques, la continuación de bucle o la parada de la ejecución. También permite la definición de etiquetas y el salto a las mismas.

En OMI es posible definir y realizar llamadas a funciones. Estas pueden hacer uso de parámetros por defecto, y estos pueden ser pasados por referencia o valor.

OMI es un lenguaje orientado a objetos. Permite la definición de clases y la instanciación de estas en objetos. Contempla algunas características propias de este paradigma:

- Definición de métodos y atributos
- Método constructor
- Visibilidad pública y privada

- Herencia simple
- Métodos y atributos estáticos
- Enlace estático en tiempo de ejecución
- Métodos mágicos
- Duck typing
- Tipos de datos básicos como clase de objetos

OMI además presenta características propias de la programación funcional:

- Funciones de primer orden
- Clausura de funciones
- Decoradores
- Pliegues
- Aplicación parcial
- Funciones anónimas
- Listas por compresión

OMI es un lenguaje con mecanismos que favorecen la reflexión y permiten llevar a cabo introspección de tipos.

El lenguaje ofrece una serie de recursos en forma de funciones destinados a la gestión de ficheros, procesos y fechas.

1.2.3. Módulos

OMI puede extender su funcionalidad y características mediante el uso de módulos. El proyecto OMI incluye el desarrollo de uno de estos módulos y la documentación de este proceso.

El módulo gettext añade funciones para la internacionalización de programas. Implementa las funciones de la biblioteca de GNU con el mismo nombre.

1.2.4. Biblioteca de desarrollo

La biblioteca de desarrollo OMI incluye todos los recursos de programación necesarios para construir el intérprete. Está escrita en C++, y se puede incluir en cualquier proyecto software escrito en este lenguaje. Esta biblioteca puede ser instalada de forma independiente al intérprete, y puede ser usada para el desarrollo de módulos o cualquier otro sistema software. Permite construir nodos semánticos a partir de los ya definidos e interpretar código fuente escrito en OMI.

1.2.5. Sitio web

El proyecto OMI incluye un sitio web que sirve como presentación del mismo, además de como medio de acceso a la documentación y el software desarrollado. Todas las páginas web pertenecientes al sitio contienen información relativa al proyecto y a las áreas que este ocupa.

Página de inicio: Describe e introduce brevemente el proyecto. Contiene enlaces a las demás secciones del sitio web. Además presenta un listado de noticias y enlaces de descargas a la última versión del intérprete.

Índice de documentación: Página que representa un índice de los documentos que conforman el proyecto.

Documentos: Páginas relativas a la documentación del proyecto en sí.

Navegador de clases: Páginas relativas a la documentación de las clases incluidas en la biblioteca.

Navegador de ficheros: Páginas relativas a la documentación de los ficheros que conforman el código fuente de la biblioteca y el intérprete.

Navegador de gramática: Páginas relativas a la documentación gráfica de la gramática del lenguaje.

Descargas: Página que enlaza la descarga de las distintas versiones del software que conforma el proyecto, disponibles en varios formatos de instalación.

Sobre OMI: Página con información relativa a la motivación y circunstancias en las que se ha dado el proyecto. Además da detalles sobre los autores y los organismos implicados en el desarrollo del mismo.

Contacto: Página con información de contacto.

El sitio web también sirve como enlace a herramientas que permiten hacer uso del intérprete de forma online.

1.2.6. RunTree

Herramienta online que permite escribir código OMI e interpretarlo. La herramienta describe el árbol sintáctico resultado del análisis del código fuente, y lleva a cabo la ejecución semántica del mismo paso a paso. Además muestra información de todo el proceso, incluyendo su estado interno y la entrada/salida de datos. Esta aplicación representa un cliente del intérprete OMI cuando este es ejecutado como un servidor. Se comunica de forma distribuida con el intérprete y procesa los datos devueltos por el mismo para mostrarlos al usuario de una forma gráfica y ordenada.

1.3. Organización del documento

La documentación recogida en esta memoria presenta el siguiente contenido y organización:

Planificación: Se describe la gestión del proyecto. Aspectos tales como la metodología, organización, los costes, la planificación, los riesgos y el aseguramiento de la calidad.

Requisitos del sistema: Se describe las necesidades y la situación que han originado el desarrollo del proyecto. Además se detallan los objetivos y los requisitos que debe cumplir el sistema.

Análisis del sistema: Se lleva a cabo el modelado relativo al análisis del sistema. Haciendo uso de diagramas UML se describe el modelo conceptual de datos, el modelo de casos de uso, el modelo de comportamiento y la interfaz de usuario.

Diseño del sistema: Aborda la arquitectura lógica y física del sistema, la descripción de la gramática, el diseño de componentes y la interfaz de usuario.

Construcción del sistema: En esta sección se describe el entorno de construcción y la organización del código.

Pruebas del sistema: Se presenta el plan de pruebas seguido, especificando la estrategia seguida, el entorno de pruebas y los distintos niveles de pruebas realizadas.

Manual de implantación: Describe cómo se ha de llevar a cabo la implantación de los distintos sistemas software que conforman el proyecto para la puesta en producción.

Manual de usuario: Representa una guía de referencia para el correcto uso del software.

Conclusiones: Se detallan los objetivos alcanzados y las lecciones aprendidas en el desarrollo del proyecto.

Bibliografía: Lista y recapitula la bibliografía empleada en el desarrollo del proyecto.

Información sobre licencia: Se presenta y detalla información sobre la licencia de uso del software y la documentación.

Por otro lado el conjunto de software desarrollado estará disponible desde varios canales y formatos:

- CD
- Forja de software
- Apartado de descargas en la web

Capítulo 2

Estado del arte

2.1. Glosario de términos

2.1.1. Analizador léxico

Un analizador léxico o scanner es un componente software cuya implementación normalmente se corresponde con un autómata finito. Este se encarga de determinar si una determinada cadena pertenece o no al conjunto de cadenas que conforma el léxico del lenguaje. Esta pieza software recibe como entrada un contenido fuente escrito generalmente como cadenas de caracteres, aunque puede soportar otros tipos de codificaciones (audio, imágenes...), y devuelve un conjunto de tokens correspondientes.

2.1.2. Analizador sintáctico

Un analizador sintáctico o parser es un componente software cuya implementación normalmente se corresponde con un autómata de pila (si la gramática es libre de contexto). Este se encarga, ayudado generalmente de un analizador léxico, de tomar un contenido fuente escrito generalmente como cadena de caracteres (aunque puede soportar otros tipos de codificaciones), y construir una estructura más fácil de analizar y procesar (normalmente árboles). El resultado de llevar a cabo un análisis sintáctico normalmente es un árbol de derivación denominado árbol sintáctico.

2.1.3. Árbol sintáctico

Árbol de derivación producto del análisis sintáctico. La raíz y demás nodos no hojas se componen de símbolos no terminales, mientras que los nodos hojas se componen únicamente de símbolos terminales

2.1.4. Argumento

Los argumentos son parte de la llamada a una función o procedimiento. Un argumento representa el valor que es asociado a un determinado parámetro cuando es llamado. La definición puede extenderse a los valores que son pasados a un programa cuando este es ejecutado. Generalmente los argumentos son asociados a los parámetros de una forma posicional.

2.1.5. Biblioteca de programación

Una Biblioteca es un conjunto de funciones o componentes de programación agrupados y encapsulados, que se encuentran relacionados y que comparten características afines. Su principal función es la reutilización de funcionalidades entre programas. En los lenguajes compilados se pueden distinguir entre bibliotecas dinámicas o estáticas.

2.1.6. Biblioteca dinámica

Una Biblioteca de programación es dinámica si se encuentra ya compilada. El programa carga y hace uso de esta en tiempo de ejecución.

2.1.7. Biblioteca estática

Una Biblioteca de programación es estática si se encuentran en un lenguaje de alto nivel. Estas son añadidas y compiladas juntas al programa.

2.1.8. Clase

En el contexto de la programación, y más concretamente dentro de la programación orientada a objetos, una clase es una estructura que define un estado por medio de variables, denominadas atributos, y un conjunto de operaciones que encapsulan un determinado comportamiento, denominados métodos. Las clases representan una plantilla o modelo a partir de la cual se crean objetos o instancias pertenecientes a la misma.

2.1.9. Código fuente

El código fuente representa las instrucciones o sentencias que conforman el programa. Generalmente estas se presentarán en un formato de cadena de texto, aunque determinados sistemas pueden ser presentados en otros formatos como imágenes, audio ...

2.1.10. Comando

Instrucción u orden que el usuario proporciona a un sistema informático. El sistema indica al usuario que espera un comando por medio de una cadena de caracteres denominada prompt.

2.1.11. Compilador

Programa que traduce un programa escrito en un lenguaje de alto nivel a otro, generalmente de bajo nivel.

2.1.12. Constante

En el ámbito de la programación una constante representa un valor que no varía en el tiempo, por lo que una vez declarado solo podrá leerse y no modificarse. El valor constante puede estar asociado a un identificador o nombre que se usará para referenciarlo.

2.1.13. Dato

Representan la unidad mínima de información. Son valores que los programas manipulan para construir la solución al problema que pretenden resolver.

2.1.14. Expresión

Secuencia de símbolos que pertenecen a un lenguaje formal. Las expresiones deben cumplir una serie de reglas determinadas por el lenguaje en las que están escritas, de forma que admiten una determinada interpretación dentro del lenguaje y cuya evaluación la atribuirá de valor. En los lenguajes de programación una expresión suele ser una combinación de constantes, operadores, funciones y demás recursos del lenguaje.

2.1.15. Extensión

Sistema software que se relaciona con otro para extender o añadir funcionalidades de este. El grado de dependencia de las extensiones con la aplicación base es muy alto y en un solo sentido.

2.1.16. Función

En el contexto de la programación una función representa un conjunto de sentencias o instrucciones con un determinado propósito. Las funciones se ejecutan al ser llamadas desde otras funciones o procedimientos del programa, o incluso desde si misma (funciones recursivas). Las funciones pueden recibir datos desde el punto desde el que son llamadas por medio de los denominados parámetros de la función. Las funciones generalmente están diseñadas para devolver un determinado valor fruto de la ejecución del algoritmo que codifican o encierran. Las funciones son un elemento de suma importancia dentro de la programación funcional, pero son también muy utilizadas en otros paradigmas de programación, tanto declarativos como imperativos.

2.1.17. Gramática

Estructura que representa unas reglas de formación que define cadenas o frases que pertenecen a un determinado lenguaje natural o formal.

Una gramática es un cuádrupla $G = (VT, VN, S, P)$, donde:

VT : Conjunto de símbolos terminales.

VN : Conjunto de símbolos no terminales.

S : Símbolo inicial de la gramática, $S \in VN$.

P : Reglas de derivación.

Los tipos de gramáticas generalmente vienen determinadas por los tipos de reglas de derivación que las componen.

2.1.18. Gramática libre de contexto

Son gramáticas cuyas reglas de derivación no dependen de un contexto. Estas son de la forma $V \rightarrow w$ donde V es un símbolo no terminal y w es una cadena de terminales y no terminales.

Las gramáticas libres de contexto originan lenguajes libres de contexto que pueden ser implementados mediante autómatas de pilas.

2.1.19. Identificador

Elemento textual o símbolo que es parte del léxico de un lenguaje y que nombran entidades del mismo como variables, constantes, funciones...

2.1.20. Interprete

Programa que ejecuta directamente las sentencias escritas en un lenguaje de programación o de scripts, sin necesidad de compilar estas a un lenguaje de bajo nivel.

2.1.21. Instrucción

Secuencia de bits que el procesador es capaz de interpretar y ejecutar. Las instrucciones que un determinado procesador es capaz reconocer viene determinada por el conjunto de instrucciones del mismo, dado en el momento de su fabricación y según la arquitectura con la que fue diseñado. También son consideradas instrucciones las representaciones de estas en lenguajes de nemotécnicos como ensamblador.

2.1.22. Lenguaje de programación

Lenguaje formal generalmente usado para crear programas. Con un lenguaje de programación es posible expresar los algoritmos que determinan el comportamiento que debe llevar a cabo un determinado programa. Está formado por un conjunto de símbolos que representa el léxico y un conjunto de reglas sintácticas y semánticas.

2.1.23. Lenguaje de programación compilados

Se refiere a aquellos lenguajes de alto nivel tal que, para ser ejecutados, los programas codificados deben ser sometidos a un proceso de traducción a lenguajes de bajo nivel. Una vez sometido al proceso de compilación un programa puede ser ejecutado directamente por la computadora para la que fue compilada.

2.1.24. Lenguaje de programación interpretados

Se refiere a aquellos lenguajes de alto nivel tal que, para ser ejecutados, los programas codificados deben ser procesados por un intérprete que se encargará de obtener su representación a bajo nivel a la vez que lo ejecuta.

2.1.25. Lenguaje de programación de alto nivel

Permiten expresar algoritmos de una forma abstracta, adecuada a la capacidad cognitiva humana. Son más cercanos al lenguaje humano que al lenguaje máquina. Un programa codificado en un lenguaje de alto nivel necesita ser procesado y transformado al conjunto de instrucciones que la computadora puede ejecutar.

2.1.26. Lenguaje de programación de bajo nivel

Son lenguajes cercanos al hardware, condicionados a la computadora. Derivan del conjunto de instrucciones soportados por la máquina, y van desde la representación binaria de estas hasta el uso de nemotécnicos.

2.1.27. Lenguaje de programación de tipado dinámico

En los lenguajes de programación de tipado dinámico el tipo de los datos es determinado en tiempo de ejecución. Generalmente el tipo de dato es asociado al valor de una variable y no a la variable en si.

2.1.28. Lenguaje de programación de tipado estático

En los lenguajes de programación de tipado estático el tipo de los datos es determinado en tiempo de compilación. Generalmente el tipo de dato es asociado a la variable en el momento de su declaración.

2.1.29. Módulo

Parte de un software informático que lleva a cabo una función específica dentro del conjunto de tareas que esta realiza. Generalmente los módulos de un programa se encuentran organizados

jerárquicamente según el nivel de abstracción que presentan y el objetivo que cumplen.

2.1.30. Objeto

En el contexto de la programación, y más concretamente dentro de la programación orientada a objetos, un objeto es una estructura que encapsula un determinado estado (atributos) y a la cual se le puede aplicar una serie de operaciones (métodos). Generalmente estos son creados mediante la instanciación de una clase de objeto, que define una serie de objetos con un comportamiento y estados afines.

2.1.31. Operador

Símbolo matemático que indica que se debe llevar a cabo una operación determinada sobre un cierto número de operandos. El operador toma los elementos iniciales y los relaciona con otro elemento de un conjunto final que puede ser de la misma naturaleza o no.

2.1.32. Paradigma de programación

Representa un estilo fundamental de programación, sirve como una forma de construir estructuras y elementos de los programas. Las capacidades y estilos de muchos lenguajes de programación son definidos para soportar determinados paradigmas de programación. Algunos lenguajes son diseñados para seguir un único paradigma, mientras que otros persiguen el soporte para varios de estos.

2.1.33. Parámetro

Los parámetros son parte de la firma de una función o procedimiento. Un parámetro representa un valor que una función o procedimiento espera que sea transferido cuando son llamados. En algunos lenguajes de programación, determinados parámetros pueden presentar valores por defecto, que son los que se tomarán si no están presentes en la llamada.

2.1.34. Procedimiento

Representa un conjunto de actividades, eventos o tareas que son ejecutadas para llevar a cabo un determinado propósito. Estas son codificadas mediante sentencias, llamadas a funciones u otros

recursos del lenguaje.

2.1.35. Programa

Secuencia de instrucciones que al ser ejecutadas por una computadora se corresponderán con la realización de una tarea específica. Para que las instrucciones que conforman un programa puedan ser ejecutadas por una determinada computadora deben presentarse en un formato legible por esta, generalmente binario, y deben estar dentro del conjunto de instrucciones que soporta.

2.1.36. Programación declarativa

Los lenguajes declarativos utilizan construcciones matemáticas para describir el problema y así obtener la solución. En los lenguajes declarativos puros se cumple una transparencia referencial en todo el sistema por lo que se evitan efectos colaterales. Además no existen las asignaciones destructivas. Esto marca una diferencia con los lenguajes imperativos y es que las funciones declarativas no pueden depender o cambiar el estado del programa. Los lenguajes multiparadigma pueden ofrecer estructuras que garanticen estos principios.

2.1.37. Programación imperativa

El programa se ve como una entidad que presenta un estado y una serie de sentencias u operaciones que hacen que dicho estado cambie. Este tipo de programación es cercana a la máquina, ya que la implementación de la mayoría de computadores a nivel hardware es imperativa.

2.1.38. Script

Programa simple, representado por una lista de comandos que serán ejecutados por un determinado programa o motor de scripts. Normalmente son utilizados para automatizar procesos. Algunos entornos que pueden ser automatizados mediante scripts incluye determinadas aplicaciones software, páginas webs, los shells del sistema operativo o sistemas embebidos.

2.1.39. Sentencia

Unidad con valor semántico a partir de la cual se construye un lenguaje de alto nivel. Son para los lenguajes de alto nivel lo que las instrucciones los son para los de bajo nivel. Generalmente

se componen de expresiones u otras construcciones propias del lenguaje.

2.1.40. Signatura

La signatura de una función o método define el nombre o identificador del mismo, así como los parámetros de los que dispone. En algunos lenguajes puede incluir el tipo de dato de los parámetros y el tipo que es devuelto.

2.1.41. Tipo array

Un array o vector representa un conjunto ordenado de valores o elementos del mismo que se encuentran posicionados en memoria de forma contigua. En muchos lenguajes esta definición suele extenderse a listas de elementos que pueden tener diferente tipo.

2.1.42. Tipo cadena de caracteres

Una cadena de caracteres es una secuencia de caracteres, que comprende signos, símbolos, letras o números. En el ámbito de la programación se utiliza normalmente como un tipo de dato compuesto, representado mediante un array cuyos elementos son los caracteres que componen la cadena.

2.1.43. Tipo expresión regular

Cadena de caracteres que representa un lenguaje regular, normalmente conforman un patrón y son utilizadas para buscar o sustituir cadenas dentro de otras.

2.1.44. Tipo lógico

Un valor lógico, también denominados booleanos, representa un valor verdadero o falso. Un valor lógico es equivalente a un valor binario 0 ó 1.

2.1.45. Tipo numérico

Un valor numérico, también denominados aritméticos, representa un valor entero o real. Un valor entero comprende los números positivos, negativos y el cero. Un valor real posee una parte entera y otra decimal.

2.1.46. Tipos de datos

Según la naturaleza de los datos estos pueden quedar organizados en tipos. Un lenguaje de programación generalmente opera sobre unos tipos de datos predefinidos, aunque ofrecen la capacidad de definir tipos de datos más complejos a partir de estos.

2.1.47. Token

Elemento léxico con cierto valor para un determinado lenguaje de programación. Normalmente se corresponde con una cadena de caracteres que se puede corresponder con una palabra reservada, un identificador, un número... Un token puede contener un valor.

2.1.48. Variable

En el contexto de la programación una variable representa un espacio de almacenaje que contiene un valor (conocido o desconocido) que es asociado a un identificador. El valor guardado por una variable puede cambiar en el tiempo de ejecución del programa. El tipo de dato que puede almacenar una variable puede estar ligado a la variable, por lo que esta sólo podrá almacenar valores de un tipo determinado, o estar asociada al valor en sí por lo que la variable podrá almacenar valores de distintos tipos.

2.2. Análisis previo de mercado

En esta sección se incluye un análisis previo de mercado que sirva como comparativa sobre los lenguajes de programación presentes y sus principales características. El objetivo es determinar las carencias que estos presentan y que podrían ser cubiertas por el lenguaje de programación OMI, a la vez que se establecen las características de las que dispondrá el software. Para ello se tomará una muestra significativa y se estudiarán aquellas características consideradas de valor para el análisis.

Los lenguajes de programación tomados como muestra se corresponden con los más utilizados en

el año 2014 según Gartner (consultor tecnológico estadounidense). Esta lista es utilizada como referencia en el mundo tecnológico por medios especializados (http://blogs.gartner.com/mark_driver/2014/10/02/gartner-programming-language-index-for-2014/).

- C/C++
- Java
- PHP
- Python
- Ruby
- JavaScript & Node.js (JS)
- OMI

2.2.1. Compilado o interpretado

Un lenguaje de programación es compilado si el código fuente desarrollado en el mismo debe ser traducido a código máquina para su ejecución. En contra partida un lenguaje es interpretado si el código fuente es procesado y ejecutado directamente por una pieza software denominado intérprete.

Muchos lenguajes son traducidos a un lenguaje intermedio denominado Bytecode que será ejecutado por un intérprete.

Algunos lenguajes interpretados pueden ser compilados a código máquina mediante alguna herramienta externas generalmente no estándar, estos casos no serán considerados en el análisis. De igual forma algunos lenguajes que normalmente son compilados pero que disponen de herramientas software capaz de interpretarlo.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Compilado	X						
ByteCode		X					
Interpretado			X	X	X	X	X

Cuadro 2.1: Lenguajes compilados e interpretados

2.2.2. Sistema de tipos

El sistema de tipos hace referencia a cómo el lenguaje clasifica los valores y expresiones en tipos. Este puede ser estático o dinámico en función si el tipo se establece en tiempo de compilación o ejecución.

Algunos lenguajes de tipado estático presenta mecanismos para escribir código que no será comprobado estáticamente, aunque estas implementaciones presentan ciertas limitaciones y pueden llegar a ser inseguras en tiempo de ejecución produciendo resultados inesperados.

Por otro lado un lenguaje es fuertemente tipado si no permite usar un dato de un tipo concreto como si de otro se tratase, a menos que se realice una conversión explícita de tipos. Mientras que es débilmente tipado si no controla el tipo de las variables y demás datos que se utilizan. Normalmente un lenguaje de tipado dinámico es débilmente tipado.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Tipado estático	X	X					
Tipado dinámico			X	X	X	X	X
Tipado fuerte	X	X					
Tipado débil			X	X	X	X	X

Cuadro 2.2: Lenguajes según sistema de tipos

2.2.3. Tipos de datos

Los tipos de datos que se pueden describir y manejar en los lenguajes de programación suponen un factor diferenciador de importancia entre estos.

Muchos lenguajes presentan tipos de datos que han sido modelados e integrados atendiendo al paradigma orientado a objetos. Sin embargo muchos tipos de datos pueden ser definidos de una forma más ágil o descriptiva usando un léxico y gramática propios.

Existen tipos de datos como las pilas, colas o árboles que son simplificaciones u otras implementaciones de otros tipos.

La solución que dan algunos lenguajes para determinados tipos de datos (como los rangos) son en forma de otras construcciones del lenguajes como funciones u objetos.

Para profundizar en el tema se podría analizar la representación interna de los datos en cada lenguaje, para comparar atributos como la optimización de espacio o rangos permitidos.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Booleanos	X	X	X	X	X	X	X
Enteros	X	X	X	X	X	X	X
Flotantes	X	X	X	X	X	X	X
Cadenas de caracteres	X	X	X	X	X	X	X
Vectores	X	X	X	X	X	X	X
Vectores asociativos	X	X	X	X	X	(*)	X
Expresiones regulares		X	X	X	X	X	X
Nulo	X	X	X	X	X	X	X
Funciones	X	X	X	X	X	X	X
Objetos	X	X	X	X	X	X	X
Referencias	X		X				X
Símbolos o identidades					X		
Rangos			X	X	X	X	
Bloques de sentencias					X		
Tupla				X			
Tablas							X
Grafos							

Cuadro 2.3: Lenguajes según tipo de datos soportados

(*) JavaScript realmente no presenta vectores asociativos, no obstantes estos pueden ser representados mediante objetos.

2.2.4. Paradigmas de programación

Los lenguajes de programación presentan cierto grado de enfoque en un determinado paradigma. Los hay así que son enfocados totalmente a uno único, tratando la mayoría de conceptos según ese criterio. Mientras que otros son multiparadigma, presentando así características, recursos y conceptos de varios de ellos.

El grado de enfoque en un determinado paradigma de un lenguaje de programación se puede determinar por la forma de tratar o modelar los distintos conceptos, y por la cantidad de recursos y características que presente a la hora de afrontar problemas según marco que el paradigma supone.

Los paradigmas no son independientes entre si, es muy común que un determinado paradigma se base en otros para su definición y tome parte de sus características. Se consideran así los paradigmas más comunes o que supongan una base para otros más concretos.

Algunas de las características que serán analizadas pueden ser implementadas en determinados lenguajes mediante la concreción de otra más general o mediante el uso de otras más simples. Por otro lado existirán lenguajes que contemplen ciertos conceptos de una forma más directa, integrada en la gramática y más próxima a lo que persigue modelar.

Es común que algunos lenguajes implementen soluciones propias de un determinado paradigma pero en una forma y estilo propios de otro.

No formará parte del análisis aquellas características que son comunes en muchos lenguajes actuales: sentencias condicionales, sentencias iterativas, inclusión de ficheros, operadores con asignación, saltos a etiquetas ...

2.2.4.1. Imperativo

Describe la programación en términos de un flujo de instrucciones que operan sobre el estado del programa. Los lenguajes imperativos constan de sentencias que permiten controlar este flujo.

Existen paradigmas que toman el imperativo como base, por ejemplo la programación estructurada, por procedimientos o la modular.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Variables de ámbito global	X		X	X	X	X	X
Foreach		X	X	X	X	X	X
Iteración ágil(*)							X
Paso de parámetros por defecto	X		X	X	X		X
Keyword arguments				X	X		X
Número de parámetros arbitrarios	X	X	X	X	X	X	
Llamada sin respetar signatura						X	
Desempaquetado de parámetros				X			
Acceso a variables no locales				X			
Toda sentencia tiene un valor						X	

Cuadro 2.4: Lenguajes imperativos

(*) La iteración ágil representa una sentencia de control de flujo iterativa cuya expresión es mínima en cuanto codificación se refiere. La sentencia únicamente se compone de una expresión que representa una secuencia de elementos y un bloque de sentencias. La expresión de secuencia será valorada según el tipo de dato que encierra. Así si la expresión es un vector se tomará como secuencia los elementos en el mismo, mientras que si es un número se tomará la secuencia de números enteros desde 0 al mismo. El bloque de sentencias será ejecutado por cada elemento en la secuencia. En cada iteración el elemento correspondiente de la secuencia será asignado a un entidad propia del lenguaje denominada “elemento iterado” y la posición del mismo se asignará a otra entidad denominada “posición del elemento iterado”. Estas entidades permiten el uso de un índice numérico para obtener el valor dentro de sentencias de este tipo anidadas.

2.2.4.2. Orientado a objeto

Describe la programación en términos de objetos que se relacionan entre si. Los objetos son estructuras que encapsulan un estado y una funcionalidad.

La programación orientada a objetos se basa en la programación imperativa.

Los lenguajes de programación orientados a objetos normalmente utilizan clases para definir un conjunto de objetos que tendrán un comportamiento y estructura afín. Los objetos se consideran instancias o materializaciones de las clases.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Visibilidad	X	X	X		X		X
Definiciones estáticas	X	X	X	X	X		X
Polimorfismo	X	X	X	X	X	X	X
Duck typing			X	X	X	X	X
Herencia simple	X	X	X	X	X		X
Herencia múltiple	X			X			
Enlace estático dinámico			X				X
Interfaces		X	X				
Traits o Mixins			X	X	X		
Name mangling	X	X		X			
Métodos mágicos			X	X	X	X	X
Todo dato es un objeto				X	X	X	X

Cuadro 2.5: Lenguajes orientados a objetos

2.2.4.2.1 POO basada en prototipos

En los lenguajes basados en prototipos los objetos son creados directamente o mediante la copia de otros objetos. En estos lenguajes no se precisa del concepto de clase.

Los prototipos son objetos que serán clonados y de los cuales otros heredarán su comportamiento y propiedades.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Creación directa de objetos						X	X
Construcción de prototipos						X	
Herencia entre prototipos						X	

Cuadro 2.6: Lenguajes orientados a objetos basados en prototipos

2.2.4.3. Dirigida por eventos

En la programación dirigida por evento el flujo que sigue la ejecución de un programa viene determinado por los sucesos que ocurren en el sistema. Estos se pueden dar por acciones del usuario o por el propio programa.

Cabe decir que una programación dirigida por evento no tiene porqué darse en un entorno de concurrencia. Un gestor de eventos denominado "event loop" puede operar sobre una cola de estos

y ejecutarlos secuencialmente.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Detección de eventos						X	
Asociación de eventos						X	
Creación de eventos						X	
Lanzador de eventos						X	

Cuadro 2.7: Lenguajes dirigidos por eventos

2.2.4.4. Basada en autómatas

En la programación basada en autómatas el sistema se modela como una máquina de estados finita. Es un tipo de paradigma imperativo.

El concepto primario de este paradigma es el estado. Un estado captura un momento actual del sistema (t_0). Además a partir de un estado se deberá conocer todos los estados pasados ($t < t_0$), distinguiéndolos así de los futuros ($t > t_0$). A partir de un conjunto finito de estados y una serie de acciones de entradas se conforma un autómata.

Aunque con la mayoría de lenguajes imperativos se puede seguir este paradigma no es común que ofrezcan características enfocadas en el mismo. Un lenguaje que sí ofrece características propias de este paradigma es MetaQuotes.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Definición de estados							
Tabla de transiciones							
Construcción de autómatas							

Cuadro 2.8: Lenguajes basados en autómatas

2.2.4.5. Orientado a Aspectos

Describe la programación valiéndose del concepto de aspecto. Un aspecto es una funcionalidad que se presenta de forma transversal, ya sea en la totalidad del sistema o de una forma dispersa.

La programación orientada a aspectos persigue la correcta modularización del sistema, separando aquellas funcionalidades que son comunes a lo largo de toda la aplicación de aquellas que quedan encapsuladas en componentes.

Aunque es común ver la programación orientada a aspectos junto la orientada a objetos es posible aplicarla junto a otros paradigmas.

Aunque muchos de los lenguajes expuestos no soportan programación orientada a aspectos por si mismos, muchos de ellos constan de recursos externos que permiten la aplicación de este

paradigma. Estos recursos amplían el lenguaje añadiendo reglas gramaticales y estructurales. En la mayoría de casos el código fuente debe ser preprocesado para obtener un código intermedio en el lenguaje base. Estos recursos no son considerados en el análisis ya que no se puede decir que el lenguaje contemple esta característica por si mismo.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Aspectos							
Puntos de corte							
Consejos							
Decoradores				X			X

Cuadro 2.9: Lenguajes orientados a aspectos

2.2.4.6. Declarativo

Los lenguajes declarativos utilizan expresiones matemáticas para describir el problema y así obtener la solución.

En los lenguajes declarativos puros se cumple una transparencia referencial en todo el sistema por lo que se evitan efectos colaterales. Además no existen las asignaciones destructivas. Esto marca una diferencia con los lenguajes imperativos y es que las funciones declarativas no pueden depender o cambiar el estado del programa. Los lenguajes multiparadigma pueden ofrecer estructuras que garanticen estos principios. Un ejemplo de lenguaje declarativo es Haskell.

Otros paradigmas son concreciones del paradigma declarativo.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Transparencia referencial							
Asignaciones no destructivas							

Cuadro 2.10: Lenguajes declarativos

2.2.4.7. Funcional

Describe la programación en términos de funciones. Los lenguajes que atienden a estos paradigma generalmente son más cercanos al lenguaje matemático que al máquina. Es un paradigma declarativo.

Algunos lenguajes integran soluciones características de la programación funcional pero de una forma más próxima a la imperativa.

Cabe decir que algunas características funcionales pueden ser implementadas si el lenguaje dispone de otras características básicas, por ejemplo la currificación o la aplicación parcial pueden ser implementadas si se dispone de funciones de orden superior y de clausura. En estos casos no se considera que el lenguaje integre estas características de forma directa.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Función de orden superior	X	X	X	X	X	X	X
Clausura		X	X	X	X	X	X
Funciones Lambda	X	X	X	X	X	X	X
Pliegues			X	X	X	X	X
Mapeo de vectores			X	X	X	X	X
Filtro de vectores			X	X	X	X	X
Continuations			X	X	X	X	
Generadores			X	X	X	X	
Lista por comprensión				X			X
Curificación							
Aplicación parcial							X
Ajuste de patrones							
Guardas							
Composición de funciones							
Infixo para func. binarias							
Evaluación perezosa							
Mapeo y pliegue (x ==OR {1,2})							

Cuadro 2.11: Lenguajes funcionales

2.2.4.8. Lógico

Describe la programación en términos de reglas lógicas que serán sometidas a un motor de inferencias para resolver los problemas planteados. Es un paradigma declarativo.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Hechos							
Reglas							
Motor de inferencias							

Cuadro 2.12: Lenguajes lógicos

2.2.4.9. Programación por restricciones

La programación se describe en términos de restricciones que toda solución debe cumplir, luego el sistema se encarga de buscar la solución. La programación por restricciones se puede dar sobre distintos dominios de datos. Es un tipo de paradigma declarativo inicialmente derivado de la programación lógica. Algunos lenguajes populares de programación por restricciones son: B-Prolog, CHIP V5 o ECLiPSe.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Restricciones sobre booleanos							
Restricciones sobre enteros y racionales							
Restricciones sobre conjuntos finitos							

Cuadro 2.13: Lenguajes dirigidos por restricciones

2.2.4.10. Programación concurrente

La programación concurrente ofrece soluciones para expresar paralelismo en la ejecución de tareas, permitiendo resolver problemas que se dan cuando varios procesos comparten o compiten por recursos.

La programación concurrente está muy presente en sistemas distribuidos y en interfaces de usuario.

La programación concurrente estudia y dispone de una serie de técnicas para la sincronización y comunicación entre procesos y/o hilos. Estas técnicas pueden ser implementadas o añadidas en forma de recursos externos en todos los lenguajes analizados, pero se estudia la integración de estas con la gramática del lenguaje.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Procesos	X	X	X	X	X	X	X
Hilos	X	X	X	X	X		
Mutex							
Semáforos							
Monitores							
Bloques sincronizados			X				

Cuadro 2.14: Lenguajes concurrentes

2.2.4.11. Programación distribuida

En la programación distribuida el sistema software se despliega en varias máquinas que pueden no estar próximas físicamente.

La programación distribuida ofrece soluciones a problemas de comunicación y sincronización entre sistemas que no se ejecutan bajo un mismo entorno.

La programación distribuida se produce en un entorno de concurrencia por lo que ambos paradigmas se encuentran muy relacionados. Ofrece una serie de técnicas para la comunicación entre procesos distribuidos mediante el uso de un canal de comunicación denominado socket.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Sockets	X	X	X	X	X	X	
Paso de mensajes							
Llamada a Procedimiento Remoto							
Modelos de objetos distribuidos							

Cuadro 2.15: Lenguajes distribuidos

2.2.5. Operadores

Los lenguajes de programación proporcionan una serie de operadores sobre los tipos de datos que define: lógicos, aritméticos, acceso...

En el análisis se tomarán aquellos operadores que supongan un factor diferenciador.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Potencia			X	X	X		X
División parte entera				X			
Unión de array			X	X	X		
Intersección de array					X		
Diferencia de array					X		
Fusión de Nulos			X				X
Ev. cortocircuito booleanos	X	X	X				
Ev. cortocircuito último valor				X	X	X	X
Ternario reducido			X				X
Comparación combinada			X		X		

Cuadro 2.16: Lenguajes según operadores

2.2.6. Funciones y clases de objetos

Los lenguajes generales ofrecen conjuntos de funciones y clases de objetos para afrontar problemas de muy distinta naturaleza. Estos pueden ser propios del lenguaje o ser añadidas mediante el uso de bibliotecas, módulos, extensiones...

Cada problema puede requerir soluciones propias. Los lenguajes de programación ofrecen gran variedad de recursos en forma de funciones o clases de objetos para dar solución a problemas comunes. No es objetivo de este análisis profundizar en este aspecto. No obstante cabe decir que existen determinadas categorías de recursos que pueden ser útiles en gran variedad de escenarios:

- Llamadas al sistema
- Fechas
- Ficheros

- Procesos
- i18n/l10n
- Bases de datos
- Matemáticas
- Sistema de ficheros
- Aleatoriedad
- Protocolos y formatos de internet

2.2.7. Definición de lenguajes específicos de dominio

Los lenguajes específicos de dominio son lenguajes enfocados a un problema o marco de problemas concretos. Existen lenguajes generales que permiten definir otros lenguajes destinados a un dominio específico.

Los DSL internos son lenguajes especificados a partir de la gramática y forma del lenguaje base sobre el que se escribe. La flexibilidad a la hora de escribir DSL internos depende en gran medida de la gramática y forma del lenguaje genérico. Existen lenguajes cuya gramática es muy flexible en este aspecto.

Los DSL externos son lenguajes que no están sujetos a la gramática y forma del lenguaje base. Debido a esto son más flexibles que los internos.

Aunque es posible escribir un DSL interno o externo en cada uno de los lenguajes referenciados, muchos de ellos no presentan características destinadas a tal fin y/o constan de ciertas limitaciones.

Es posible atribuirle a una gramática características como herencia, modularidad... Existen lenguajes especializados en escribir gramáticas como bison o ANTLR.

El léxico de muchos lenguajes está conformado por cadena de caracteres que serán reconocidos, no obstante un lenguaje puede utilizar un léxico de otra naturaleza como sonidos, secuencia de eventos, grupos de píxeles, componentes de un diagrama...

Los DSL generados pueden ser creados e interpretados en tiempo de ejecución o añadidos mediante módulos compilados al interprete base.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
DSL interno					X		
DSL externo							
Léxico flexible							

Cuadro 2.17: Lenguajes según DSL

2.2.8. Depuración

Algunos lenguajes presentan características para la depuración de los programas que se escriben sobre ellos. Muchos pueden ser depurados mediante herramientas u otros recursos externos.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Asertos			X	X	X	X	
Depurador integrado			X	X	X	X	

Cuadro 2.18: Lenguajes según depuración

2.2.9. Autodocumentación del proceso de interpretación

Un lenguaje puede tener la capacidad de autoexplicarse, por ejemplo puede dar información paso a paso sobre el proceso llevado a cabo para la interpretación, permitir seguir el flujo de ejecución o dibujar gráficas de rendimiento y espacio consumidos.

Para que el interprete/compilador presente autodocumentación debe ser capaz de dar información sobre todo el proceso. Además si integra mecanismo para generar DSL deberá documentar también los procesos aplicados a estos.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Análisis léxico							
Autómata léxico							
Análisis sintáctico							
Árbol sintáctico							X
Aplicación semántica							X
Árbol de análisis decorado							X
Estados de la tabla de símbolos							X
Tiempos de ejecución							
Espacio de memoria ocupado							X
Depurador integrado							X

Cuadro 2.19: Lenguajes autodocumentados

2.2.10. Otras características funcionales

En este punto se presentan funciones que cumplen algunos lenguajes y que no han sido categorizadas, pero que aportan valor al análisis.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Constantes	X	X	X		X	X	
Expresiones con sigilo			X				
Extensiones			X	X	X		X
Reflexión		X	X	X	X	X	
Introspección de tipos	X		X	X	X		
Incrustable en otros documentos			X				
Acceso a estructuras de bajo nivel	X						
Docstring integrado				X			
Interprete interactivo			X	X	X	X	X
Excepciones	X	X	X	X	X	X	X
Espacio de nombres	X	X	X	X	X	X	
IPC integrado							
ORM integrado							X
Patrones de diseño integrados							X
Recolección de Basura		X	X	X	X	X	X
Principio de ocultación	X	X	X				
Niveles de aviso y errores			X				

Cuadro 2.20: Lenguajes según otras características funcionales

2.2.11. Características no funcionales

2.2.11.1. Modularidad

Un interprete se puede presentar de forma modular según diferentes criterios:

2.2.11.1.1 Modularidad de recursos

Un lenguaje es modular en este sentido si tiene la capacidad de ampliar o configurar los recursos que brinda mediante módulos que encapsulan construcciones del lenguaje. Las construcciones (funciones y clases generalmente) encapsuladas en un módulo cumplen un propósito específico que puede ser útil en determinados problemas. Este tipo de módulos generalmente serán compilados de forma independiente y cargados dinámicamente, pero también pueden ser compilados junto el binario que representa el interprete.

La mayoría de lenguajes actuales son ampliable en recursos.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Modularización de recursos	X	X	X	X	X		X

Cuadro 2.21: Lenguajes según modularidad de recursos

2.2.11.1.2 Modularidad del lenguaje

El interprete puede tener una arquitectura interna modular, de forma que el lenguaje que representa puede ser extendido o personalizado tanto en léxico, como en gramática o en semántica. Estos módulos, que definen el lenguaje, pueden ser cargados en tiempo de ejecución o compilados junto al interprete.

Un lenguaje modular no es un lenguaje en si, sino un una base para construir otros lenguajes. A partir de un lenguaje de este tipo se pueden construir otros lenguajes, por ejemplo si se toma los módulos adecuados para construir el interprete se puede obtener un lenguaje puramente matemático.

Es necesario aclara que aunque el interprete tenga la capacidad de generar e interpretar lenguajes específicos de dominio, no quiere decir que brinde un lenguaje modular ya que las gramáticas generadas de esta forma se pueden ver como recursos del propio lenguaje. Un lenguaje capaz de generar DSL es modular en este sentido si el interprete que lo procesa puede compilarse sin esta característica.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Modularización del lenguaje							

Cuadro 2.22: Lenguajes según modularidad del propio lenguaje

2.2.11.2. Licencia

Todos los interpretes tomados en el análisis presentan licencias propias, pero es posible analizar las características de estas:

En este análisis se considerará únicamente la licencia de Node.js como JS dado que cada motor JavaScript dispone de su propia licencia.

Las licencias de C/C++ dependen del compilador tomado en el análisis, considerándose el compilador G++ de GNU.

Para Java se tomará la plataforma desarrollada por Sun Microsystems, aunque existen otras a las que se le aplican diferentes términos de licencia de uso.

	C/C++	Java	PHP	Python	Ruby	JS	OMI
Software libre	X		X	X	X	X	X
Compatible con DFSG	X			X	X	X	X
Aprobada por OSI	X		X	X	X	X	X
Compatible GPL	X			X	X	X	X
Copyleft	X				X		X
Utilizable junto con otras licencias	X			X	X	X	X

Cuadro 2.23: Lenguajes según licencia

2.2.11.3. Rendimiento

El rendimiento supone una característica muy valorada en los lenguajes de programación. Debido a la gran cantidad de algoritmos que pueden ser codificados y la diversidad de estos, se hace difícil un análisis exhaustivo del rendimiento. No obstante existen una serie de benchmarks muy popularizados y estandarizados. Incluso se pueden ver proyectos dedicados a someter a los lenguajes de programación a un conjunto de estos benchmark (The Computer Language Benchmarks Game: <http://benchmarksgame.alioth.debian.org/>).

2.2.11.4. Productividad

Los lenguajes de programación deben perseguir la productividad de sus usuarios, para ello deben presentar:

- Léxico estándar, uniforme y sencillo.
- Gramática clara y directa.
- Estructuras estándar y uniforme.

Aunque todos los lenguajes analizados presentan estas características en mayor o menor grado, no es objetivo de este análisis entrar en discusión al respecto.

2.2.11.5. Portabilidad

Una característica común en muchos interpretes y compiladores es que se presentan como software multiplataforma, siendo posible su uso en la mayoría de sistemas operativos actuales.

La portabilidad es una característica muy valorada, pero que no supone un factor diferenciador dado que la mayoría lo cumplen.

Capítulo 3

Planificación

En esta sección se trata todos los aspectos relativos a la gestión del proyecto. Se describe el proceso aplicado para el correcto desarrollo del proyecto, dentro de las limitaciones dadas por el alcance, tiempo, calidad y presupuesto.

La gestión del proyecto comprende la implantación de una metodología de desarrollo, viéndose esta como la implantación de una serie de pasos, técnicas, procedimientos y demás recursos que ayuden a desarrollar el producto software dentro de un marco de trabajo.

Por otro lado también se ha de tratar la planificación del proyecto, quedando este organizado en una serie de tareas y actividades derivadas de la metodología implantada.

En la ejecución de un proyecto es necesario la inversión de una serie de recursos, tales como herramientas, personal, equipos, herramientas... Se ha de determinar los recursos asignados a la ejecución del mismo, así como los roles de las personas asignadas y la relación entre estas.

El desarrollo de un proyecto software tiene unos costes derivados de los recursos asociados al mismo. Estos recursos serán tanto materiales como humanos, y tendrán un coste fijo asociado que se utilizará para el cálculo del coste total.

Otra de las tareas que se llevan a cabo durante la gestión de un proyecto software es el análisis de riesgos. Todo proyecto está sujeto a una probabilidad de que se den escenarios de riesgo en los que se pueda ver perjudicada la correcta realización del mismo. Se detectarán, listarán y analizarán los riesgos y sus consecuencias, así como la probabilidad de que estos sucedan y las prácticas que se llevarán a cabo para mitigar los efectos derivados.

Por último se expondrán las prácticas seguidas para asegurar la calidad del desarrollo y los productos obtenidos en cada paso de la metodología. Para ello se incluyen los estándares, prácticas y normas aplicados durante el desarrollo. Además se recogen los distintos tipos de revisiones, verificaciones y validaciones que se han llevado a cabo, así como los criterios para la aceptación o rechazo de cada producto y los procedimientos para implantar acciones correctoras o preventivas.

La planificación expuesta no recoge aspectos como la instalación, el mantenimiento o el soporte.

Esta se centra únicamente en el ciclo de desarrollo del proyecto y no en los procesos posteriores, los cuales, aunque también forman parte del ciclo de vida hábil del software, no se encuentran dentro de las etapas de desarrollo del mismo.

3.1. Metodología de desarrollo

Para la realización del proyecto se ha seguido una metodología iterativa e incremental. Más concretamente se ha tomado como base el proceso unificado de desarrollo de software, el cual sigue un enfoque dirigido por casos de uso y centrado en la arquitectura.

El ciclo de vida sigue un enfoque en espiral, dividido en cuatro etapas: determinar objetivos, análisis de riesgos, desarrollo y planificación.

Determinar objetivos :

- Se fijan los productos a obtener: requisitos, especificación, manuales ...
- Se fijan las restricciones a las que estará sujeta el proyecto
- Sólo en la primera iteración se lleva a cabo una planificación inicial en esta etapa.

Análisis de riesgos :

- Se estudia las posibles amenazas y eventos no deseados, así como los daños y consecuencias derivados de estos.
- Se evalúan las distintas alternativas que permitan minimizar los riesgos.

Desarrollo :

- Se lleva a cabo el desarrollo de lo fijado en las etapas anteriores.
- El desarrollo de cada iteración se divide en cuatro etapas: análisis, diseño, codificación y pruebas.

Planificación :

- Se analiza los productos obtenidos y el estado del proyecto.
- Se lleva a cabo una planificación de la siguiente iteración del ciclo de vida.

En un enfoque en espiral lo más común es que en la primera iteración se ofrezca un prototipo del producto a desarrollar, no obstante en el proyecto abordado no ha sido así. En lugar de ello se ha planteado una primera iteración que recoja el alcance del proyecto, así como los requisitos y análisis de los riesgos globales, además se realiza una planificación de las iteraciones que seguirán. Las demás iteraciones contemplan un subconjunto de estos requisitos, añadiéndose en cada una características al software, y afinando el análisis global llevado en la primera y siguientes iteraciones.

Para la realización de los productos obtenidos en cada paso de la metodología se ha utilizado el lenguaje de modelado UML.

3.2. Planificación

La planificación se divide en una serie de iteraciones. Todas las iteraciones con excepción de la primera tienen las mismas etapas en función de la metodología seguida.

Las etapas y subetapas en las que se divide cada iteración son:

- Objetivos
- Riesgos
- Desarrollo
 - Análisis
 - Diseño
 - Codificación
 - Pruebas
- Planificación

La planificación tiene como punto de partida el día 03/11/2014, día en la que se comenzó el desarrollo del proyecto. Se ha tomado una jornada laboral de 8 horas, y una semana hábil de 5 días.

En cada etapa de la planificación se hacen labores de documentación para que toda la información relativa al proyecto quede reflejada en la memoria del mismo.

3.2.1. Diagramas de Gantt de las iteraciones generales

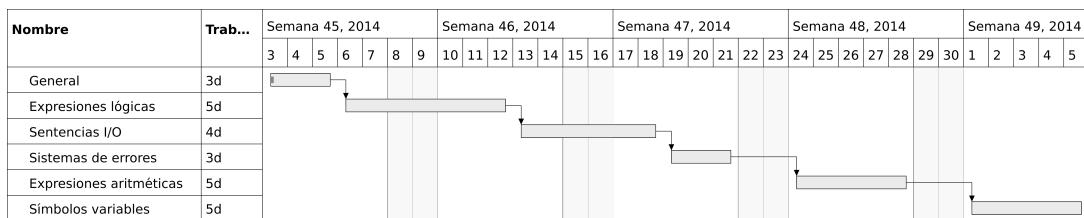


Figura 3.1: Planificación general 01

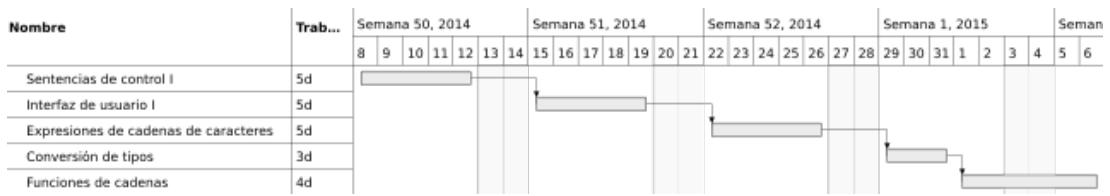


Figura 3.2: Planificación general 02

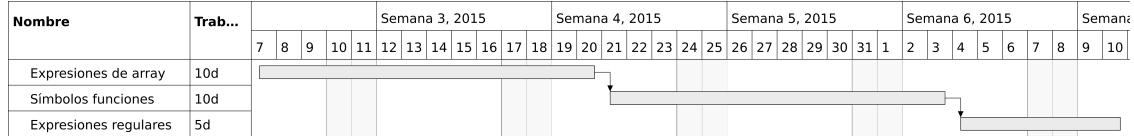


Figura 3.3: Planificación general 03

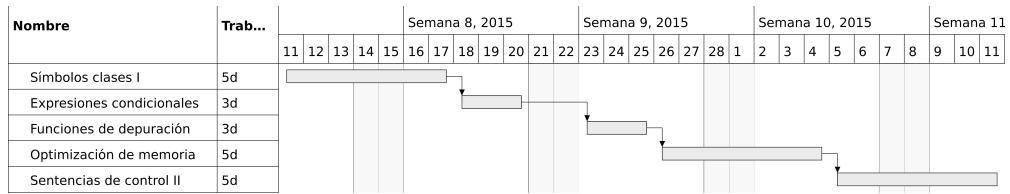


Figura 3.4: Planificación general 04

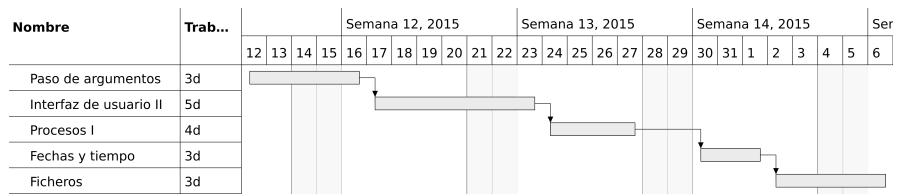


Figura 3.5: Planificación general 05

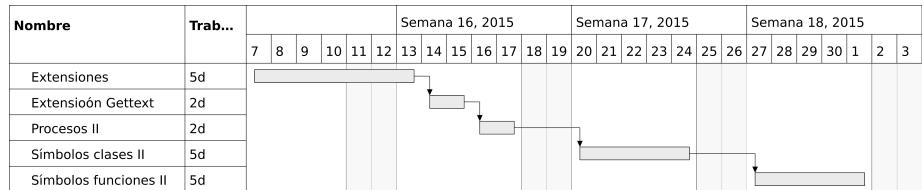


Figura 3.6: Planificación general 06

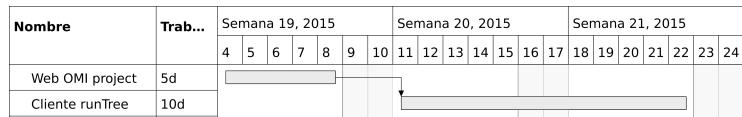


Figura 3.7: Planificación general 07

3.3. Organización

Para la realización del proyecto se ha utilizado los siguientes recursos humanos :

Fco. Bohórquez Ogalla: Director de proyecto, analista, diseñador de arquitectura, diseñador de sistemas, desarrollador, tester, control de calidad

Además se han utilizado los siguientes recursos materiales:

Equipo de trabajo 1: Computadora para las tareas de gestión, administración, desarrollo y documentación.

Las herramientas utilizadas son las siguientes:

Sistema operativo: GNU/Linux.

Entorno integrado de desarrollo: Geany.

Generador léxico: Flex.

Generador sintáctico: Bison.

Compilador: GCC.

Depurador: GDB.

Herramientas para la construcción automática: Autoconf, automake, make.

Desarrollo de diagramas: Dia, railroad diagram generator.

Control de versiones: Subversion.

Creación de documentación: Latex, Doxygen.

Planificación: Planner.

Bibliotecas de desarrollo: readline, boost.

Servidor HTTP: Apache.

Intérprete de Scripts de servidor: PHP.

Navegador web: Firefox, Chrome, Explorer.

Conversor latex a HTML: latex2html.

Editor gráfico: Gimp, Inkscape.

Comunicación: Servicio gratuito de correo electrónico.

3.4. Costes

A continuación se presenta el coste relativo a los recursos humanos invertidos en el desarrollo del proyecto. Para ello se ha tomado como referencia el documento BOE publicado el sábado 30 de noviembre de 2013 por el ministerio de empleo y seguridad social. En este documento se recoge la tabla salarial según el convenio colectivo de la empresa Trevenque Sistemas de Información SL.

Empleado	Grupo	Salario anual bruto	Salario mensual bruto
Fco. Bohórquez Ogalla	Programado senior	18.120,00€	1.510,00€

Cuadro 3.1: Salarios

El tiempo de desarrollo asciende a 8 meses a tiempo completo. El coste de los recursos humanos puede ser consultado en la tabla 3.2.

Para el desarrollo del proyecto se ha necesitado de un equipo con una potencia y prestaciones medias. El coste de los recursos materiales que suponen estas computadoras puede consultarse en la tabla 3.3.

Las herramientas utilizadas en el desarrollo del proyecto tiene licencia libre. El uso de estas herramientas no suponen coste adicional para el desarrollo del proyecto.

Fco. Bohórquez Ogalla	12.080,00€
Total	12.080,00€

Cuadro 3.2: Costes totales

Computador i5-4460/ 4GB/ 1TB	1x	409,00€
Total		409,00€

Cuadro 3.3: Equipos

3.5. Riesgos

En este punto se listan los riesgos identificados. Estos pueden originar un efecto negativo en el desarrollo del proyecto. Además se muestra la probabilidad de que estos se den y el impacto que tendrían. Una vez identificados los riesgos se definen los planes seguidos para reducir los efectos derivados estos o disminuir la probabilidad de que ocurran.

El análisis de riesgos se ha llevado a cabo en cada iteración del ciclo de desarrollo. Cada iteración a completado la información mostrada. Muchos de los riesgos indicados son comunes a todas las iteraciones realizadas.

Los riesgos analizados según el impacto que pueden ocasionar en el proyecto quedan divididos en:

Insignificantes: No merecen ser tenidos en cuenta.

Tolerables: Están dentro de un margen de aceptación, por tanto no comprometen el proyecto, ni el producto, ni la organización.

Graves: Comprometen gravemente el proyecto, el producto o la organización.

Críticos: Amenaza la supervivencia del proyecto, el producto o la organización.

Para dar claridad al análisis, la probabilidad de que se dé un determinado escenario de riesgo se presenta de forma relativa como sigue:

Muy baja: < 10 %.

Baja: del 10 al 25 %.

Moderada: del 25 al 50 %.

Alta: del 50 al 75 %.

Muy alta: > 75 %.

Los riesgos quedan organizados en distintas categorías para un mejor análisis.

3.5.1. Riesgos tecnológicos

Son los riesgos derivados del software y hardware necesarios para el desarrollo del proyecto.

Riesgo	Probabilidad	Impacto
Los recursos no están disponibles a tiempo	Baja	Tolerable
Fallos en el sistema operativo u otros software de sistema	Baja	Tolerable
Fallos en el hardware de desarrollo	Baja	Grave
Errores de configuración	Baja	Tolerable
Actualización del sistema no realizada correctamente	Baja	Tolerable
Integridad y privacidad de los datos	Baja	Grave
Manipulación deliberada de los programa	Moderada	Grave
Las herramientas de comunicación no se encuentran disponibles	Baja	Insignificante
El repositorio de código no se encuentra disponible	Baja	Tolerable

Cuadro 3.4: Riesgos tecnológicos

En el caso de que los recursos materiales, hardware de desarrollo, no estén disponibles a tiempo se puede comenzar con tareas de definición y análisis. Si la demora se hace demasiado extensa se cancelará el pedido del hardware y se realizará a otro proveedor. Esto puede ocasionar un retraso de días que puede ser mitigado dedicando este tiempo a tareas en las que no se precise de hardware.

Si se produce algún fallo en el software de sistema que da soporte al desarrollo este puede derivar en pérdidas de datos. La solución sería gestionar la incidencia o reinstalar el sistema. La perdida de datos se puede mitigar si se realizan copias de seguridad periódicas. Como medida preventiva se pueden guardar puntos de restauración del sistema para prevenir o mitigar la demora de tiempo que esto podría ocasionar.

Es posible que se produzcan fallos en el hardware sobre el cual se desarrolla el proyecto, esto puede ocasionar retrasos de entregas o la perdida de los datos. Para prevenir y mitigar este efecto negativo se puede realizar copias periódicas de los datos y mantener los equipos en un estado de funcionamiento óptimo, bien refrigerados y sin exposición a agentes externos.

Si se produce algún error en la configuración del sistema es posible que algunas características de este dejen de funcionar, en muchos casos esto puede llegar a ser perjudicial para el proyecto. Para prevenir y mitigar este efecto se deberán realizar copias de seguridad, además se deberá minimizar la cantidad de software instalado en los equipos. En el caso de producirse un fallo de configuración del sistema que afecte directamente a la ejecución del proyecto se deberá invertir tiempo para corregir las causas y administrar el sistema.

En algunos casos una actualización del sistema que da soporte a la ejecución del proyecto puede ocasionar fallos en el mismo o en la configuración. Para ello además de las soluciones y prácticas comentadas en los puntos anteriores, se puede minimizar el número de actualizaciones que no sean

críticas o que no sean indispensables para el desarrollo del proyecto. Como medida de actuación el sistema deberá ser restablecido mediante un administrador cualificado.

La integridad y privacidad de los datos se puede ver comprometida por algún uso indebido de los mismos o por agentes ajenos al proyecto. Para prevenir y mitigar los efectos de este escenario se realizarán copias de seguridad periódicas y se hará uso de un software de control de versiones. Además se protegerá el acceso a los datos mediante técnicas como claves de acceso, configuraciones de red pocas permisivas (por ejemplo mediante el uso de firewalls), configuraciones óptimas del sistema (en cuanto servicios y programas), cifrado, no permitir la conexión de dispositivos extraíbles, etc. Es muy importante en este punto controlar además los permisos de publicación, transferencia y portabilidad que tienen los empleados sobre los datos, quedando registrada y validada toda operación de esta naturaleza. Como medida de actuación ante un escenario de este tipo los datos quedarán en cuarentena y se llevará a cabo un análisis forense que determine los agentes implicados y los motivos.

Es posible que en algunos casos los programas que conforman el sistema sobre el que se desarrolla sean manipulado de forma consciente o inconsciente por una persona integrante del equipo o ajena al mismo. Para prevenir este escenario se acotará el uso del equipo al propósito que este tiene, no se permitirá la instalación de software adicional, se protegerá el acceso al sistema mediante claves robustas y se usará software antivirus. Como medida de actuación se bloqueará el acceso al proyecto a cualquier software que se tenga constancia de que se encuentre manipulado o infectado por alguna clase de malware, y el equipo implicado se podrá en cuarentena para evaluar el impacto de lo ocurrido.

Las herramientas de comunicación son un elemento clave en el desarrollo de un proyecto. Depender de terceros para dar soporte al proyecto en este aspecto puede llegar a ser perjudicial. Para prevenir pérdidas en el servicio se puede usar clientes estables, fiables y consolidados en el mercado. Para mitigar la perdida de información que supondría una caída del servicio se puede hacer copias de toda la información enviada por estos medios. Como medida de actuación, siempre y cuando sea necesario una comunicación, se utilizarán otros medios de comunicación como el teléfono.

El repositorio en el cual se aloja el código fuente puede sufrir caídas en el servicio. Para mitigar y prevenir este tipo de incidencias se alojará el sistema de control de versiones en alguna máquina de la infraestructura local al proyecto. Esto bloquea o dificulta el acceso remoto al código, lo cual tiene sus ventajas ya que disminuye el grado de exposición, pero hace que el acceso al código esté condicionado por un equipo local, siendo de vital importancia la disponibilidad y visibilidad de este.

3.5.2. Riesgos de requisitos

Son riesgos que surgen de los requisitos, ya sean debido a que estos han cambiado o que no se han recogido correctamente.

Si los requisitos no son enumerados y definidos de forma efectiva puede ocasionar que el sistema no haga lo que debe hacer. Para evitar esto se llevará a cabo múltiples reuniones con el cliente para la toma de requisitos, donde el analista tomará parte activa de estas y nunca llegará a

Riesgo	Probabilidad	Impacto
Especificación de requisitos insuficiente	Moderada	Grave
Captura de requisitos errónea	Moderada	Grave
Casos de uso complejos o mal redactados	Baja	Tolerable
No se han capturado todos los datos que definen o con los que trabaja el sistema	Baja	Tolerable
Añadir nuevas características sin tener en cuenta la arquitectura interna del sistema	Alta	Tolerable

Cuadro 3.5: Riesgos de requisitos

presuponer ningún aspecto que no quede bien acotado. En este punto el analista podrá optar por algunas de las técnicas conocidas para la toma de requisitos. Como plan de actuación se deberá poner en contacto con el cliente y pedir la aclaración o especificación de los puntos que no quedaron fijados.

De igual forma unos requisitos mal tomados puede tener como resultado un producto que no cumple con las expectativas. Para prevenir y evitar este escenario el analista debe ser muy conciso y minucioso en la toma de requisitos, haciendo que todo quede claro por ambas partes.

Aunque se realice una toma de requisitos completa, queda la posibilidad de que el analista no transmita correctamente qué va a hacer el sistema para dar solución a estos. Se deberá poner especial atención en que los casos de usos queden bien redactados, en un lenguaje simple y con la suficiente sencillez y completud.

Una de las principales actividades de un sistema informático es procesar datos, dado que el valor de la información viene atribuido por los datos que la forman. Si los datos sobre los que opera o definen el sistema no son determinados con exactitud el valor que este aporta disminuye. En la toma de requisitos se ha de poner especial atención en recoger los datos que construyen el modelo de datos de una forma completa y exacta.

Definir nuevos requisitos del sistema conociendo la estructura y características internas de este es una ventaja, no obstante no siempre es posible hacer que los nuevos requisitos se adapten a las posibilidades que el software brinda, ni el cliente tiene porque conocer las restricciones que determinadas decisiones de diseño han impuesto. Para prevenir y mitigar este efecto, y que el software sea ampliable en características de una forma abierta, se deben diseñar soluciones versátiles, flexibles al cambio y con las mínimas restricciones asociadas.

3.5.3. Riesgos de soluciones

Son riesgos asociados con el diseño de soluciones

El tomar como base una solución errónea puede ocasionar que el software no produzca los resultados esperados. Si los errores son detectados a tiempo el impacto puede ser bajo, pero si el error persiste en varias iteraciones del proceso de desarrollo puede ocasionar pérdidas cuantiosas. Para prevenir esta situación se deben pasar auditorías de calidad en todas las iteraciones, no solo en el sentido de que la solución se ha aplicado correctamente, sino también de que estas

Riesgo	Probabilidad	Impacto
Diseño de solución errónea o demasiada restrictiva	Moderada	Grave
Cambios versiones y características de las soluciones software utilizadas	Baja	Tolerable
No se encuentra un léxico adecuado	Baja	Tolerable
La gramática es confusa	Moderada	Tolerable
No se ha seguido el principio de reutilización de código	Baja	Tolerable
Código inteligible	Baja	Tolerable
Errores de seguridad	Moderada	Grave

Cuadro 3.6: Riesgos de soluciones

cumple unos criterios mínimos de optimización, seguridad, flexibilidad, etc. Cuando se detecta que una solución tomada no es correcta se deberá analizar el coste de las distintas alternativas para corregirlo y si fuera necesario aplicar la más óptima para el caso.

Muchas soluciones software utilizadas pueden cambiar de versión y con ella las características que ofrecen, pudiendo quedar algunas de ellas eliminadas. Esto puede ocasionar que el sistema desarrollado no cumpla con lo que antes sí hacía. Para evitar esta situación siempre se guardará una versión estable de las soluciones software tomadas. Antes de actualizar el software que da solución a algún aspecto del sistema se deberá de tener en cuenta el impacto que esta operación tendrá. En el caso de ser necesario actualizar se deberá adaptar el sistema para el uso de la nueva versión.

Para un lenguaje de programación el disponer de un léxico sencillo, fácil de recordar y común en otros lenguajes es un requisito necesario, si esto no se logra el resultado es un lenguaje que el mercado no estará dispuesto a usar. Para prevenir esto se ha de analizar y medir cada palabra que componga el léxico, sometiéndolo a evaluación por todo el equipo y por personas ajenos al mismo. Es posible tomar como referencias otros lenguajes disponibles en el mercado. Si se detecta o determina que una palabra del léxico no es adecuada se deberá de cambiar.

De igual forma que en el punto anterior, una gramática confusa puede originar un lenguaje poco usado, cuyo coste de aprendizaje sea elevado. Nuevamente para evitar este escenario se deberá analizar y evaluar la estructura de la gramática elegida. Si se detecta una gramática confusa se deberá analizar y proponer otras alternativas.

Una solución desarrollada que no sea reutilizable implica la múltiple realización del trabajo. Para mitigar y prevenir este efecto se deberá modularizar y acotar toda función que se pueda reutilizar. Las funciones, clases y demás recursos de programación deben tener un propósito concreto y bien definido. Si se detecta que una determinada parte del sistema se está reescribiendo por no seguir un principio de reutilización se deberá invertir tiempo en revertir esta situación.

Hacer que el sistema se conforme de partes de código que no sean fáciles de entender puede ir en contra del mantenimiento del proyecto y de la corrección del sistema. Para prevenir este escenario se deberá seguir unas reglas de estilo uniforme que construyan un código limpio y fácil de entender. Al detectar este tipo de código se ha de invertir tiempo en clarificar la sección afectada.

Al escribir programas complejos y extensos es muy común que se den fallos de seguridad que permitan explotar vulnerabilidades como podría ser el desbordamiento de buffer o algún tipo de inyección. Para evitar esto se podrá realizar auditorias de seguridad al código desarrollado. Si se detecta algún fallo de seguridad en la fase de desarrollo este debe ser notificado y corregido.

3.5.4. Riesgos de costes, tiempos y recursos

En esta sección se exponen los riesgos derivados de cálculos en cuanto a costes, recursos y tiempo.

Riesgo	Probabilidad	Impacto
Recursos necesarios no previstos	Baja	Tolerable
Planificación optimista	Moderada	Tolerable
La planificación de la siguiente iteración tarda más de lo esperado	Moderada	Tolerable
El presupuesto es insuficiente para afrontar el desarrollo	Baja	Crítica

Cuadro 3.7: Riesgos de costes, tiempos y recursos

Si el número de empleados contratados para el proyecto es insuficiente el tiempo necesario para el desarrollo del mismo puede incrementar considerablemente. Por otro lado contratar empleados conlleva un coste monetario. Es por tanto que se debe llegar a una configuración óptima. Para prevenir este escenario se ha de llevar a cabo una planificación realista llevada a cabo desde la experiencia y de una forma objetiva. Si la realidad difiere de la planificación realizada y se precisa de más empleados, habrá que asumir el coste de contratación además del tiempo de incorporación que variará en función del perfil.

Es posible que en la planificación inicial no se contemple la totalidad de los recursos materiales necesarios. Esto podría originar retrasos en las entregas y tiempos de espera. Para mitigar este efecto se puede tener un fondo reservado para posibles gastos adicionales no previsto, no obstante lo ideal es hacer una planificación exacta en cuanto a los recursos requeridos.

Es común que muchas planificaciones se hagan desde un punto de vista optimista, esto conlleva desviaciones entre lo calculado y la realidad, en temas como el tiempo, los recursos o los compromisos que se lleguen a materializar. Para evitar este escenario, se deberá llevar a cabo una planificación realista, con márgenes de actuación.

Retardar la planificación de la siguiente iteración puede llegar a ser un problema que bloquee el proceso de desarrollo, dado que no se han marcado las pautas para continuar. Para mitigar esto la planificación puede desarrollarse de forma paralela al recorrido de la iteración actual.

Es posible que el presupuesto para la realización del proyecto quede insuficiente, esto puede ocasionar el cese de la actividad del mismo. Como método de previsión se pueden estudiar varias alternativas de financiación.

3.6. Aseguramiento de calidad

Para asegurar una calidad aceptable del producto y los procesos llevados a cabo en el desarrollo se ha tomado como esquema el conjunto de normas ISO 9000. El sistema de gestión de la calidad está formado por procesos y es en si mismo un proceso en el que ingresan los requisitos del sistema, y se obtiene un producto que cumple los requisitos y satisface al cliente.

A continuación se describen los procesos y actividades seguidas para asegurar la calidad.

- Se implementa un ciclo de vida evolutivo en el que se examina y documenta cada decisión tomada, al final de cada iteración se comprueba la calidad del producto obtenido.
- Se realiza una planificación en la que se capturen las acciones a seguir para alcanzar los objetivos perseguidos.
- Los roles y responsabilidades del equipo quedan definido y atribuidos en cada iteración del ciclo.
- En cada iteración se examina las pruebas de calidad realizadas, la retroalimentación del cliente y la conformidad del producto.
- En cada iteración se contemplan las mejoras del producto en relación con los requisitos y necesidades del cliente.
- Todo el personal tiene los conocimientos y el entrenamiento adecuados para realizar la tarea que en la planificación se le ha sido asignada.
- Se utilizan entrevistas periódicas con el cliente para capturar los requisitos necesarios en cada iteración. Además de validar la corrección del producto obtenido
- Para la toma de requisitos se ha utilizado técnicas de objetivos medibles, de forma que los requisitos impuesto por el cliente se ven como objetivos generales. Estos son analizados repetidamente para obtener los requisitos críticos para el funcionamiento del sistema. En cada iteración se refinan los requisitos generales y se plantean nuevos, aplicándose la misma técnica para determinar los tomados en la iteración.
- Se presentan modelos de caso de uso que deben ser comprendidos y validados por el cliente.
- El diseño del producto obtenido en cada iteración queda documentado mediante modelos.
- Se realiza una revisión, verificación y validación de los diseños propuestos.
- La actividad de desarrollo del producto quedará bien documentada en el propio código.
- En cada iteración se definirán los casos de pruebas, estas serán llevadas a cabo para comprobar la corrección del producto.

A continuación los criterios para la aceptación o rechazo de los productos obtenidos en cada fase del desarrollo.

- Si se detecta algún requisito ambiguo o mal especificado este queda rechazado. En este caso se deberá concertar una entrevista con el cliente.
- Para la aceptación de los requisitos estos deben ser simples y sencillos, estar bien descritos y especificados de forma atómica.
- Si se detecta un caso de uso mal explicado o confuso este será rechazado.
- Los casos de uso para su aceptación deben estar completamente explicados, de una forma esquemática y fácil de entender por el cliente. Además debe quedar bien definidos los actores involucrados y las relaciones entre estos.
- Los modelos de datos serán aceptados si son claros, completos y se encuentran bien estructurados.
- Los diseños de soluciones serán aceptados si cumple los criterios de versatilidad, adaptación, optimización y claridad impuestos.
- Si en el diseño de la gramática se da alguna ambigüedad esta deberá ser redefinida.
- Si la elección de una palabra del léxico es confusa o demasiado compleja esta deberá ser redefinida.
- Si alguna clase no cumple el principio de única responsabilidad será rechazada.
- Si alguna clase no cumple el principio abierto/cerrado será rechazada.
- Si alguna clase no cumple el principio de sustitución de Liskov será rechazada.
- Si alguna clase no cumple el principio de segregación de la interfaz será rechazada.
- Si alguna clase no cumple el principio de inversión de dependencias será rechazada.
- Si algún módulo de código fuente desarrollado no cumple las reglas de estilo este será rechazado.
- Si algún módulo de código fuente no se encuentra debidamente documentado será rechazado.
- Si algún módulo de código fuente no sigue el principio de reutilización será rechazado.
- Si se detecta que no se han capturados todos los casos de prueba esenciales se produce un rechazo.
- Para la aceptación se deben completar satisfactoriamente todas las pruebas unitarias.

Parte II

Desarrollo

En esta parte se debe describir el desarrollo del proyecto siguiendo la metodología empleada. Sus capítulos no deben ser una descripción exhaustiva de todos los documentos, diagramas, código fuente y, en general, entregables generados, sino más bien una explicación resumida del desarrollo, estructurada según las etapas principales del proceso de ingeniería. Deben seleccionarse aquellos diagramas, fragmentos de código y secciones de los entregables que sean más significativos para dicha explicación. La totalidad de los entregables resultado del proyecto se ubicarán en los anexos y/o en el material en CD/DVD que acompañe al proyecto.

Capítulo 4

Requisitos del Sistema

4.1. Visión general

En esta sección se lleva a cabo un análisis de la situación actual en el estudio de la teoría de autómatas y los lenguajes formales. Se describirán las necesidades presentes en este campo, y que serán objetivo de las soluciones planteadas. A partir de estos objetivos se llevará a cabo una descripción de los requisitos que debe cumplir la solución tomada.

4.2. Situación actual

El estudio de los lenguajes formales es anterior a la concepción de las computadoras. Las matemáticas, la lógica y otras ciencias venían haciendo uso de los conceptos de los lenguajes formales para la solución de problemas.

Una computadora desde un punto de vista teórico es un autómata o máquina de estados, que es capaz de ejecutar una serie de instrucciones descritas en lenguaje máquina. Por tanto fueron los avances en la rama de la teoría de autómata y los lenguajes formales, entre otros campos, los que permitieron la concepción de las primeras computadoras. Actualmente la mayoría de las ingenierías de la información se estudian los conceptos teóricos detrás de las computadoras y los lenguajes de programación.

Lo más común es que un estudiante de informática comience sus estudios en estos campos con los autómatas: los tipos que existen, cómo se definen y para qué se utilizan. Llegando a comprender conceptos como el de estado, alfabeto, etc. Incluso estudiando definiciones de algunos tipos de lenguajes formales como los lenguajes regulares.

Posteriormente el alumno podría utilizar los conceptos aprendidos para estudiar distintos modelos de computación, los cuales son definidos formalmente y desde un punto de vista teórico. La máquina de turing y el cálculo lambda son piezas esenciales en este punto del aprendizaje.

Una vez que se es poseedor del conocimiento base en la teoría de autómatas y los lenguajes formales, el alumno puede aplicar estos conocimientos para estudiar las estructuras, mecanismos y demás conceptos detrás de los lenguajes de programación. En este punto se estudian los intérpretes y compiladores, y los conceptos básicos que hay detrás estos. Conceptos como el léxico, sintaxis y la semántica de los lenguajes de programación, las tablas de símbolos, o las distintos tipos de gramáticas.

Hasta aquí se habrá obtenido los conocimientos teóricos necesarios y el alumno podría comenzar a dar soluciones prácticas a problemas, aplicando así los conocimientos obtenidos. Así es común que se comience con el desarrollo de analizadores léxicos y sintácticos sencillos y concretos, para luego aplicarles una semántica. Ejercicios como el desarrollo de una calculadora suelen ser habituales. Además se estudian algunas de las herramientas que asisten al proceso del desarrollo de este tipo de tecnologías.

Después del proceso descrito al alumno se le ha brindado la oportunidad de profundizar en un campo con multitud de ramas, técnicas, metodologías y conceptos, que son fruto de años de estudio de expertos y apasionados. Podría profundizar en el proceso de compilación o traducción, en las distintas gramáticas, en técnicas de optimización o diseñar sus propias herramientas de traducción o interpretación de lenguajes formales.

Por otro lado, en la industria de la tecnología de la información se hace uso de lenguajes muy completos, con gran diversidad de mecanismos y bien consolidados, que son efecto de la evolución y las necesidades en el sector. En la mayoría de cursos académicos se estudian estas herramientas desde un punto de vista práctico y de uso.

La teoría de autómatas y lenguajes formales presenta la base para el estudio de los compiladores e intérpretes que son parte fundamental de la industrial actual. A pesar de ello no existen herramientas divulgativas, colaborativas e interactivas que, a partir de los conocimientos básicos, ayude a comprender cómo se desarrollan los distintos mecanismos y herramientas presentes en la tecnología actual.

Los lenguajes de programación han evolucionado mucho desde que comenzaron a alejarse del lenguaje máquina. El alumnado actual trabaja sobre los conceptos que le ayudan a entender esta evolución, pero no dispone de herramientas o medios para ver cómo estos conceptos son trasladados a un producto real y presente en el día a día de un programador actual.

4.3. Necesidades

Dada la situación actual se precisa de una herramienta que ayude a comprender cómo se implementa y construye un intérprete para un lenguaje de programación. Es condición necesaria que este proceso quede correctamente documentado. La herramienta elaborada deberá ser accesible por cualquier interesado en el tema, que desee profundizar en la práctica del desarrollo de intérpretes y lenguajes de programación.

Se partirá de los conceptos básicos de la teoría de autómatas y los lenguajes formales, así como de la teoría de compiladores e intérpretes. Se asume pues que el usuario dispone de este conocimiento.

4.4. Objetivos

Se llegará a construir un lenguaje de programación completo con características presente en la tecnología actual. Este proceso quedará correctamente documentado y se pondrá a disposición pública. Las características que serán contempladas son:

- Distintos tipos de datos simples y compuestos.
- Expresiones lógicas y aritméticas.
- Expresiones y funciones sobre cadenas.
- Expresiones y funciones sobre vectores de datos.
- Expresiones y funciones sobre expresiones regulares.
- Operadores de conversión de tipos.
- Mecanismos de entrada/salida.
- Creación de procesos.
- Manipulación de ficheros.
- Funciones de fecha y tiempo.
- Definición y uso de variables de tipado dinámico.
- Ámbito de variables.
- Sentencias de control de flujo condicionales e iterativas.
- Definición y uso de funciones y procedimientos.
- Mecanismos y técnicas de la programación funcional.
- Definición y uso de clases de objetos.
- Mecanismos y técnicas de la programación orientada a objeto.
- Integración de módulos que extienden el lenguaje.

El intérprete desarrollado podrá ser usado de una forma interactiva, permitiendo así la ejecución de instrucciones bajo demanda. Además deberá tener la capacidad de dar información estructurada de todo el proceso de interpretación. Otros sistemas podrán utilizar esta información para ilustrar este proceso de una forma flexible. El intérprete también se deberá poder ejecutar en modo servidor, recibiendo así cualquier cadena que hace de código fuente por un puerto TCP e interpretándola.

Toda la documentación generada a partir del proceso de desarrollo deberá ser estructurada y cumplimentada para formar una parte de una aplicación web que la haga accesible. La aplicación

web además permitirá el uso online del intérprete, y mostrará información relativa al proceso de interpretación.

El proyecto presentará una licencia de uso libre para que pueda ser usado abiertamente por la comunidad, a la vez que se nutrirá de las contribuciones de la misma.

4.5. Requisitos funcionales

4.5.1. Intérprete

4.5.1.1. Léxico, gramática y semántica

Número: 0001

Nombre: Léxico.

Categoría: Intérprete.

Descripción: El sistema debe fijar el léxico del lenguaje conformado por una conjunto de palabras y expresiones bien definidas y acotadas.

Número: 0002

Nombre: Gramática.

Categoría: Intérprete.

Descripción: El sistema debe definir una gramática que representará el lenguaje. La gramática debe ser libre de contexto, clara y uniforme en toda su extensión. Además debe estar libre de ambigüedades.

Número: 0003

Nombre: Interpretación semántica.

Categoría: Intérprete.

Descripción: Dado un contenido fuente el sistema debe analizarlo en función al léxico (análisis léxico) y la gramática (análisis sintáctico) del lenguaje y producir el resultado semántico asociado.

4.5.1.2. Contenido fuente

Número: 0004

Nombre: Comentarios.

Categoría: Intérprete.

Descripción: Se ha de contemplar un mecanismo para añadir comentarios al contenido fuente que serán ignorados durante la tarea de interpretación.

Los comentarios comprenderán desde un carácter “#”, o bien “//”, hasta fin de línea.

Por otro lado se ha de contemplar los comentarios de múltiples líneas, que deberán estar contenidos entre “/*” y “*/”.

Número: 0005

Nombre: Fuente desde línea de comandos.

Categoría: Intérprete.

Descripción: El intérprete debe ser capaz de obtener contenido fuente desde una línea de comandos.

Número: 0006

Nombre: Fuente desde entrada estándar.

Categoría: Intérprete.

Descripción: El intérprete debe ser capaz de obtener contenido fuente desde la entrada estándar del sistema.

Número: 0007

Nombre: Fuente desde fichero.

Categoría: Intérprete.

Descripción: El intérprete debe ser capaz de obtener contenido fuente desde un fichero.

Número: 0008

Nombre: Fuente desde puerto TCP.

Categoría: Intérprete.

Descripción: El intérprete debe poder ser ejecutado en modo servidor, recibiendo una cadena de caracteres por un puerto TCP, e interpretándola. Esto occasionará que el servidor devuelva por otro puerto una estructura de datos que representa el proceso de interpretación.

4.5.1.3. Salida

Número: 0009

Nombre: Salida por la salida estándar.

Categoría: Intérprete.

Descripción: El intérprete debe poder utilizar la salida estándar del sistema para imprimir el resultado de la interpretación según el código fuente. Para ello se deberá presentar sentencias que permitan escribir en esta.

Número: 0010

Nombre: Salida estructurada.

Categoría: Intérprete.

Descripción: El intérprete debe de tener la capacidad de producir una serie de datos estructurados que representen el proceso de interpretación llevado a cabo para un determinado código fuente. Estos datos podrán ser almacenados en un fichero o devueltos por un puerto TCP.

4.5.1.4. Entorno

Número: 0011

Nombre: Entorno de ejecución.

Categoría: Intérprete.

Descripción: El intérprete debe definir un entorno de ejecución en el que se controlen parámetros de entrada, variables de entornos del sistema operativo e información sobre el proceso como número de línea actual y los errores producidos.

4.5.1.5. Parámetros

Número: 0012

Nombre: Parámetros al programa.

Categoría: Entrada.

Descripción: Se debe facilitar un mecanismo para que el contenido fuente pueda recibir parámetros de entrada desde la invocación a su interpretación. Estos parámetros deberán ser copiados a símbolos variables accesibles desde el contenido fuente. Adicionalmente se tratará otro parámetro que se corresponderá con el número de parámetros dados.

4.5.2. Ejecución

4.5.2.1. Sentencias

Número: 0013

Nombre: Sentencia.

Categoría: Ejecución.

Descripción: Son las unidades interpretables más pequeña en las que se divide un contenido fuente. Las sentencias están sujetas a unas reglas sintácticas y encierran un significado semántico. El intérprete debe definir la gramática de cada sentencia y dotarlas de significado semántico. Toda sentencia debe finalizar con el carácter “;”, excluyendo las sentencias formadas por bloques de sentencias. Aunque carezca de sentido práctico, para evitar posibles errores de codificación y mantener coherencia en la sintaxis y la definición del lenguaje, se debe contemplar la sentencia vacía que sólo conste del carácter “;”.

Número: 0014

Nombre: Bloques de sentencias.

Categoría: Ejecución.

Descripción: Son un conjunto de sentencias que deberán ser interpretadas y ejecutadas secuencialmente. La disposición de sentencias en el bloque determinan el flujo de ejecución que se llevará a cabo cuando se intérprete el bloque. El contenido fuente en si mismo es un bloque de sentencias. Todo bloque de sentencias de más de una sentencia (con excepción del contenido fuente en si mismo) debe ir delimitado mediante llaves. Aunque no sea de uso común, para mantener coherencia en la sintaxis y la definición del lenguaje, se debe contemplar el bloque de sentencias vacío.

4.5.2.2. Expresiones

Número: 0015

Nombre: Expresiones.

Categoría: Ejecución.

Descripción: El intérprete debe ser capaz de evaluar expresiones. Estas son secuencias de datos, operadores, operandos, elementos de puntuación y/o palabras clave, que especifican una unidad computacional. Generalmente encierran un valor que se asocia a la expresión después de ser evaluada. Una sentencia puede estar formada por una o varias expresiones que deberán ser evaluadas o interpretadas para dotarla de significado. Una sentencia puede constar únicamente de una expresión en ese caso la sentencia es considerada la evaluación de dicha expresión.

La expresión más simple equivale a un único dato, en este caso el valor de la expresión será el del dato.

Número: 0016

Nombre: Expresiones de tipo definido.

Categoría: Ejecución.

Descripción: Son expresiones cuyo valor es de un tipo definido y fijo. El sistema debe interpretar estas expresiones para determinar el valor asociado a la mismas en un momento dado.

Nombre: Expresiones de tipo no definido.

Categoría: Ejecución.

Número: 0017

Descripción: Son expresiones cuyo valor no tiene un tipo definido ni fijo, sino que es durante la interpretación cuando se determina el tipo. El sistema debe interpretar estas expresiones para determinar, además del valor asociado a la mismas, el tipo de dato que guardan en un momento dado.

4.5.2.3. Datos

Nombre: Datos.

Categoría: Ejecución.

Número: 0018

Descripción: El intérprete deberá operar sobre datos. El contenido fuente definirá cómo se han de construir y/o acceder a los datos y las operaciones que se realizarán sobre ellos durante la ejecución.

Un dato será tratado en función de su tipo. El tipo de dato lo dota de una semántica, un significado. Así, todo dato deberá ser considerado un objeto, por lo que tendrán unas propiedades y funcionalidad ligadas al tipo como el que es tratado.

4.5.2.4. Operadores

Número: 0019

Nombre: Operadores.

Categoría: Operadores.

Descripción: Se ha de facilitar una serie de operadores que permitan manipular los datos.

Los operadores son en si mismo expresiones, por los que estos tendrán un valor asociado tras ejecutarse. Los operadores constarán de una serie operandos que intervendrán en la operación y que serán a su vez otras expresiones.

Los operadores se clasificarán en función del tipo de valor que tendrán tras la ejecución, los tipos de los operandos y/o la naturaleza del operador en si.

Un operador puede presentarse en forma de función, los operandos serán los parámetros de esta. La ejecución de la función conllevará la realización de la operación asociada al operador.

4.5.3. Tipos de datos

Número: 0020

Nombre: Tipos de datos.

Categoría: Tipos de datos.

Descripción: El sistema debe ser capaz de interpretar y operar sobre diferentes tipos de datos.

Las expresiones pueden tener un tipo de dato asociado que puede o no ser definido y fijo.

Los tipos de datos pueden ser simples o compuestos.

Un dato debe tratarse como diferente tipo en función del contexto en el que se utilice. Así un dato de un tipo concreto puede ser tratado como otro tipo de dato si fuese necesario. Un dato por si mismo siempre será considerado del tipo de dato con el que se creó, sin embargo cuando interviene en una operación es posible que se precise una conversión o equivalencia de tipos. Para ello debe tomar su valor como si de otro tipo se tratase. Si en la operación no es posible convertir el tipo en el tipo requerido se debe producir un error de tipos.

Se debe establecer un mecanismo de conversión de tipos. La relación de conversión de tipo debe ser transitiva, así si un dato de tipo lógico puede verse como un dato de tipo aritmético y un dato aritmético como un cadena de caracteres, entonces el dato de tipo lógico también puede verse como una cadena.

4.5.3.1. Nulo

Número: 0021

Nombre: Tipo de dato nulo.

Categoría: Tipo de dato.

Descripción: Se debe contemplar el tipo de dato nulo. Este tipo de dato tendrá un único valor posible. El valor nulo deberá representar un elemento no definido. Una expresión puede tomar el valor nulo cuando sea evaluada si se hace uso de elementos no definidos.

4.5.3.2. Lógico

Número: 0022

Nombre: Tipo de dato lógico.

Categoría: Tipo de dato.

Descripción: El sistema debe ser capaz de interpretar y operar sobre datos de tipo lógicos.

Este tipo de dato sólo contempla dos posibles valores: falso y verdadero. Este será el tipo de dato más simple. Un dato lógico puede ser tratado como un tipo de dato aritmético tomándose falso como el valor cero, y verdadero como el valor uno.

4.5.3.3. Aritmético

Número: 0023

Nombre: Tipo de dato aritmético.

Categoría: Tipo de dato.

Descripción: El sistema debe ser capaz de interpretar y operar sobre datos de tipo aritméticos.

Este tipo de dato contempla valores numéricos racionales que puedan ser representados mediante notación en coma flotante. Todo dato aritmético además tiene asociado un valor lógico cuando se utiliza como este tipo de dato, tal que cualquier número distinto de cero tiene valor verdadero y el cero tiene el valor falso. Además cuando un dato aritmético es tratado como una cadena de caracteres se tomará la cadena que representa al número. El intérprete debe tener la capacidad de configurarse con una representación interna distinta para los datos aritméticos.

4.5.3.4. Cadena de caracteres

Número: 0024

Nombre: Tipo de dato cadenas de caracteres

Categoría: Tipo de dato.

Descripción: El sistema debe ser capaz de interpretar y operar sobre datos de tipo cadena de caracteres. Este tipo de dato contempla cualquier sucesión de caracteres alfanuméricos, secuencias de escape, u otros signos o símbolos. Esta sucesión puede ser vacía. Una cadena de caracteres vendrá delimitada mediante comillas dobles o simples. Toda cadena de caracteres además tiene asociado un valor aritmético cuando se utiliza como este tipo de dato, tal que, si la cadena representa un número racional el valor será el del propio número, por otro lado, si la cadena no representa un número racional el valor aritmético de la misma será el número de caracteres que la conforman. No se ha de considerar el tipo de dato carácter simple, pudiéndose tratar este como una cadena de un solo elemento.

4.5.3.5. Expresiones regulares

Número: 0025

Nombre: Tipo de dato expresión regular.

Categoría: Tipo de dato.

Descripción: El sistema debe ser capaz de interpretar y operar sobre datos de tipo expresión regular. Una expresión regular consiste en una cadena de caracteres que representan un patrón. Las expresiones regulares tendrán una sintaxis PERL. Una expresión regular se delimita mediante caracteres acento grave (`). El tipo de dato expresión regular no debe ser tratado como otro tipo de dato.

4.5.3.6. Arrays

Número: 0026

Nombre: Tipo de dato array.

Categoría: Tipo de dato.

Descripción: El sistema debe ser capaz de interpretar y operar sobre datos de tipo array. Este tipo de dato contempla cualquier sucesión de elementos. Estos elementos pueden ser pares de expresiones clave/valor donde la clave servirá para referenciar el valor dentro de la sucesión. También pueden ser simples expresiones, por lo que se tomará automáticamente una clave numérica y secuencial según el orden del array y como valor el de la expresión. El significado semántico de las claves en un array puede ser numérico (array numérico) o cadenas de caracteres (array asociativo). Una definición de array se delimita mediante llaves y sus elementos se denotarán mediante un listado de expresiones o pares de estas. Un dato de tipo array solo puede ser tratado como un tipo de dato booleano, siendo falso si se encuentra vacío y verdadero en caso contrario.

4.5.4. Sentencias

4.5.4.1. Inclusión de ficheros

Número: 0027

Nombre: Inclusión de ficheros.

Categoría: Sentencias de control de flujo.

Descripción: Se ha de facilitar un mecanismo para incluir, en un punto de la ejecución, contenido fuente localizado en recurso externo. El recurso consistirá en un fichero con sentencias interpretables.

4.5.4.2. Saltar a etiqueta

Número: 0028

Nombre: Saltar a etiqueta.

Categoría: Sentencias de control de flujo.

Descripción: Se ha de facilitar un mecanismo para llevar el flujo de ejecución a la sentencia referenciada por una etiqueta.

4.5.4.3. Sentencia if

Número: 0029

Nombre: Sentencia if.

Categoría: Sentencias de control de flujo.

Descripción: Deben de existir una serie de sentencias condicionales que alteren el flujo de ejecución. Las sentencias if deberán estar construidas por bloques de sentencias y una serie de expresiones denominadas “condiciones”. La interpretación de una sentencia de este tipo debe consistir en la evaluación lógica de las “condiciones” para determinar el bloque de sentencias que se ejecutará. Las formas de la sentencia if que el intérprete debe aceptar son las siguientes:

- if (cond) stmts
- if (cond) stmts else stmts
- if (cond) stmts elif (cond) stmts ... else stmts

4.5.4.4. Sentencia switch

Número: 0030

Nombre: Sentencia switch.

Categoría: Sentencias de control de flujo.

Descripción: El intérprete debe ser capaz de interpretar sentencias del tipo switch case. Estas constan de una lista de bloques de sentencias precedidas de una expresión denominada “caso”. Dada una expresión base esta debe ser comparada mediante la operación de igualdad con cada uno de los casos, ejecutando el bloque correspondiente al “caso” cuya comparación sea positiva y todos los bloques siguientes. Se deberá poder especificar un bloque denominado “default” que no dispondrá de expresión “caso” y será ejecutado sin aplicar condición alguna.

4.5.4.5. Sentencia while

Número: 0031

Nombre: Sentencia while.

Categoría: Sentencias de control de flujo.

Descripción: El intérprete debe ser capaz de interpretar sentencias del tipo while. Esta es una sentencia de control iterativa que consta de una expresión denominada “condición” y un bloque de sentencias. El bloque de sentencias debe ser ejecutado mientras que “condición” permanezca verdadera.

4.5.4.6. Sentencia do...while

Número: 0032

Nombre: Sentencia do...while.

Categoría: Sentencias de control de flujo.

Descripción: El intérprete debe ser capaz de interpretar sentencias del tipo do while. Esta es una sentencia de control iterativa que consta de una expresión denominada “condición” y un bloque de sentencias. El bloque de sentencias debe ser ejecutado mientras que “condición” permanezca verdadera, llevándose a cabo la ejecución al menos una vez.

4.5.4.7. Sentencia for

Número: 0033

Nombre: Sentencia for.

Categoría: Sentencias de control de flujo.

Descripción: El intérprete debe ser capaz de interpretar sentencias del tipo for. Esta es una sentencia de control iterativa que consta de tres expresiones denominadas “inicialización”, “condición” y “paso”, además de un bloque de sentencias. Primero se ha de evaluar la expresión “inicialización”, luego el bloque de sentencias se ejecutará mientras “condición” se valore como verdadera. La expresión “paso” se deberá ejecutar al finalizar cada iteración.

4.5.4.8. Sentencia foreach

Número: 0034

Nombre: Sentencia foreach.

Categoría: Sentencia de control de flujo.

Descripción: Se han de interpretar sentencias del tipo foreach. Esta es una sentencia de control iterativa que consta un bloque de sentencias, de una expresión denominada “conjunto” y un identificador denominado “valor”. Se debe poder, aunque de forma opcional, especificar otro identificador que se denominará “clave”. El bloque de sentencias será ejecutado de forma iterativa en función el tipo de dato y el valor de “conjunto”. El “conjunto” será evaluado para determinar el número de iteraciones y el valor que se le asignará como variables a los identificadores en cada iteración. Dependiendo del tipo de la expresión “conjunto” la sentencia foreach deberá actuar como sigue:

Tipo lógico: El bloque de sentencias se ejecutará mientras “conjunto” sea verdadero. El identificador “valor” tomará el valor verdadero. En el caso en el que se especifique un identificador “clave” a este no se le asignará ningún valor.

Tipo aritmético: Si “conjunto” representa un número mayor que cero el bloque de sentencias se ejecutará tantas veces como el valor numérico que representa. En cada iteración “valor” se le asignará el número de la iteración comenzando por cero. Si se presenta un identificador “clave” a este no se le asignará ningún valor. Si el valor de “conjunto” es menor o igual a cero el bloque no deberá ejecutarse.

Tipo cadena de caracteres: Si “conjunto” es una cadena de caracteres que representa un número racional la ejecución deberá ser como si de un tipo aritmético se tratase. Si el “conjunto” es una cadena que no representa un número racional el bloque de sentencias se ejecutará por cada carácter en la cadena. En este último caso a “valor” se le asignará el carácter contemplado en cada iteración. Si se dio un identificador “clave” este no será asignado.

Tipo array: Si “conjunto” es un array, u otro tipo de dato derivado de este como un objeto, el bloque de sentencias se ejecuta por cada elemento en el mismo. Al identificador “valor” se le asignará el valor del elemento en el array correspondiente a la iteración. En el caso de que se facilite un identificador “clave” este deberá tomar la clave del elemento en el array.

Otros tipos: No se llevará a cabo ninguna operación.

4.5.4.9. Sentencia de iteración ágil

Número: 0035

Nombre: Sentencia de iteración ágil.

Categoría: Sentencia de control de flujo.

Descripción: Se ha de facilitar una sentencia de control que permita iterar un bloque de sentencias en función una expresión “conjunto” de forma ágil y sencilla. Para ello esta sentencia deberá operar igual que la sentencia foreach pero sin ser necesario, aunque posible, dar un identificador “valor” sobre el que se realizará la asignación. En lugar de ello la asignación que se produce en cada iteración se deberá realizar sobre un símbolo con identificador fijo y contenido variable denominado iterador. El iterador debe ser accesible desde el bloque de sentencias. Además se debe contemplar el acceso al iterador de varias sentencias de ciclo ágil cuando estas se presentan de forma anidada.

4.5.4.10. Sentencia with

Número: 0036

Nombre: Sentencia with.

Categoría: Sentencia de control de flujo.

Descripción: Se deberá facilitar un mecanismo que permita establecer una estructura compuesta como contexto. Así todo acceso que se realice, y cuya definición no exista, se deberá hacer sobre los elementos que de la estructura compuesta utilizada como contexto. Esta sentencia se deberá construir a partir del dato compuesto y un bloque de sentencias sobre el que se aplicará el contexto.

4.5.4.11. Finalizar bloque de sentencias

Número: 0037

Nombre: Finalizar bloque de sentencias.

Categoría: Sentencias de control de flujo.

Descripción: Se ha de disponer de un mecanismo para indicar que el flujo debe salir de una sentencia de control. Se ha de contemplar las sentencias anidadas.

4.5.4.12. Finalizar iteración

Número: 0038

Nombre: Finalizar iteración.

Categoría: Sentencias de control de flujo.

Descripción: El sistema debe facilitar algún recurso que permita finalizar la iteración actual de una sentencia de control en ejecución y comenzar con la siguiente. Este mecanismo debe contemplar la posibilidad de salir de varias sentencias de control anidadas.

4.5.4.13. Finalizar ejecución

Número: 0039

Nombre: Finalizar ejecución.

Categoría: Sentencias de control de flujo.

Descripción: Se ha de disponer de un mecanismo para que el sistema finalice de forma inmediata de interpretar el contenido fuente.

4.5.4.14. Capturar excepción

Número: 0040

Nombre: Capturar excepción.

Categoría: Sentencias de control de flujo.

Descripción: Se ha de disponer de un mecanismo para definir un bloque de sentencias en el que puede suceder una excepción, a la vez que permita definir un bloque de sentencias en el que esta será tratada. La excepción podrá devolver un valor que debe ser capturado. No se ha de contemplar bloques en los que se trate cada excepción según su tipo, en lugar de ello se usará un único bloque de sentencias y el tipo deberá ser evaluado dentro de este.

4.5.4.15. Lanzar excepción

Número: 0041

Nombre: Lanzar excepción.

Categoría: Sentencias de control de flujo.

Descripción: Se ha de disponer de un mecanismo para lanzar excepciones. Cuando una excepción es lanzada esta podrá ser capturada por un bloque de sentencias destinado para ello. Si la excepción no es capturada se producirá un error. Cuando una excepción es lanzada se deberá de dar un valor que podrá ser utilizado en el bloque que la captura.

4.5.5. Definiciones

Número: 0042

Nombre: Identificadores.

Categoría: Definiciones.

Descripción: El intérprete debe facilitar mecanismos para que el usuario defina e identifique expresiones, datos, bloques de sentencias, y otras construcciones y elementos del lenguaje. Se precisa una manera unívoca de nombrar estos elementos. Un identificador válido debe estar formado por una secuencia de caracteres alfanuméricos de al menos un carácter, donde el primer carácter a de ser una letra.

Número: 0043

Nombre: Tabla de símbolos.

Categoría: Definiciones.

Descripción: El intérprete debe ser capaz de gestionar tablas de símbolos. Los símbolos hacen referencias a valores, funciones y otras expresiones del lenguajes. Para acceder a estos símbolos se debe utilizar un identificador. Se hace necesario el acceso y uso de los símbolos según el contexto de ejecución, determinado por el ámbito y el tipo símbolo, para ello deben poder coexistir diferentes tablas de símbolos globales. Para evitar conflictos en el uso de identificadores algunos conceptos deben disponer de su propia tabla de símbolos.

4.5.5.1. Variables

Número: 0044

Nombre: Variables.

Categoría: Definiciones.

Descripción: El intérprete debe ser capaz gestionar una serie de símbolos denominados variables. Estos relacionan un identificador con un valor que puede variar durante el proceso de ejecución. El tipo de una variable dependerá del tipo del valor al que referencia (tipado dinámico), este podría ser de cualquiera de los tipos de datos soportados. La tabla de símbolos de variables debe adaptarse al contexto de ejecución.

Número: 0045

Nombre: Variables globales.

Categoría: Definiciones.

Descripción: Aunque la tabla de símbolos de variables es dependiente del contexto de ejecución se ha de facilitar algún mecanismo para que un dato esté disponible independientemente del contexto en el que se acceda.

4.5.5.2. Funciones

4.5.5.2.1 Definición de función

Número: 0046

Nombre: Definición de función

Categoría: Definiciones.

Descripción: Se necesita de un mecanismo que permita definir y nombrar bloques de sentencias. Estos bloques podrán recibir unos valores de entrada y producir una salida. Las sentencias en el bloque podrán operar sobre los parámetros de entrada, representados por unos símbolos variables que tomarán distintos valores en cada ejecución. Tras interpretarse el bloque de sentencias se podrá tomar un valor considerado de salida.

La definición de una función representará en si misma un dato, por lo que podrán formar parte de operaciones y otras expresiones. Una función se define mediante un bloque de sentencias, una lista de identificadores que nominan a los parámetros de entrada y un identificador que le da nombre a la propia función, aunque este último no debe ser necesario (funciones anónimas).

4.5.5.2.2 Llamada a función

Número: 0047

Nombre: Llamada a función

Categoría: Definiciones.

Descripción: Dada una función, se debe disponer de un mecanismo que permita la ejecución del bloque de sentencias que la forma, mediante el uso de unos valores concretos como parámetros de entrada, y con la posibilidad de tomar el valor de salida.

Una llamada a función se deberá componer de un identificador relativo a su definición, y una lista de expresiones que determinarán los valores de los parámetros. La llamada deberá ser en sí misma una expresión que tomará como valor la salida de la función tras la ejecución.

Los valores de los parámetros se corresponderán con los parámetros de la definición de la función de forma posicional.

4.5.5.2.3 Valor de retorno

Número: 0048

Nombre: Valor de retorno

Categoría: Definiciones.

Descripción: Se necesita de un mecanismo en forma de sentencia que, dada una función, determine el valor del salida que se tomará en la llamada a la misma. La sentencia return se compondrá de una expresión correspondiente al valor salida. Al ser interpretada la ejecución de la función deberá finalizar, y tomará el valor de la expresión dada.

4.5.5.2.4 Valores de parámetros por defecto

Número: 0049

Nombre: Valores de parámetros por defecto.

Categoría: Definiciones.

Descripción: Dada la definición de una función, debe existir un mecanismo para que los parámetros de esta puedan tener valores por defecto. Estos valores serán asignados a los parámetros cuando los valores no sean dados en una llamada a función.

Como la correspondencia entre parámetros en una llamada a función se hace de forma posicional, los valores por defecto deberán ser especificados desde el final de la lista de parámetros hasta el inicio.

4.5.5.2.5 Parámetros por valor

Número: 0050

Nombre: Parámetros por valor.

Categoría: Definiciones.

Descripción: Cuando una función es ejecutada todos los símbolos variables que se definan y utilicen deben tratarse de forma local al bloque de sentencias de la función. De esta forma los símbolos variables definidos fuera de la función no serán accesibles desde el cuerpo de la misma y viceversa. Cuando se realice una llamada a función los valores de los parámetros deben ser copiados a los símbolos variables correspondientes.

4.5.5.2.6 Parámetros por referencia

Número: 0051

Nombre: Parámetros por referencia.

Categoría: Definiciones.

Descripción: Se necesita de un mecanismo que permita que los parámetros de una función referencien valores definidos fuera del cuerpo de la misma. De esta forma se podrá acceder y/o modificar datos externos a la función.

Cuando en una llamada a función se especifiquen expresiones que sean símbolos variables como algunos de sus parámetros, si estos se definieron en la función como parámetros por referencia, el valor del símbolo en la llamada será referenciado por el símbolo correspondiente de la función.

4.5.5.2.7 Función lambda

Número: 0052

Nombre: Función lambda.

Categoría: Definiciones.

Descripción: Se debe dar la posibilidad de crear funciones anónimas. Estas funciones carecerán de nombre y normalmente se utilizarán en la asignación de variables, como parámetros de otras funciones o como valor de retorno. Las funciones lambda deberán ser en sí misma una expresión que toma como valor el dato correspondiente a la función.

4.5.5.2.8 Expresiones parametrizadas

Número: 0053

Nombre: Función lambda simple.

Categoría: Definiciones.

Descripción: Se debe de facilitar un mecanismo para crear funciones simples, que solo consten de una lista de parámetros y de una única expresión que será devuelta y que constituirá el cuerpo de la función.

4.5.5.2.9 Referencia a función

Número: 0054

Nombre: Referencia a función.

Categoría: Definiciones.

Descripción: Se debe facilitar un mecanismo para referenciar funciones ya creadas, de forma que puedan ser asignadas a variables, pasadas como parámetros o devueltas como valor de retorno.

4.5.5.2.10 Funciones de orden superior

Número: 0055

Nombre: Funciones de orden superior.

Categoría: Definiciones.

Descripción: Se debe poder definir funciones de orden superior, estas pueden recibir otras funciones como parámetros o tomar como valor de retorno otra función.

4.5.5.2.11 Funciones clausura

Número: 0056

Nombre: Funciones clausura.

Categoría: Definiciones.

Descripción: Se debe poder definir funciones dentro del contexto de otras. La función de clausura puede tener variables que dependan del entorno en el que se ha definido la función. De esta forma cuando la función es llamada podrá acceder al valor de la variable en el contexto en el que se definió.

4.5.5.2.12 Aplicación parcial

Número: 0057

Nombre: Aplicación parcial.

Categoría: Definiciones.

Descripción: Se debe facilitar un mecanismo que permita, a partir de una función, obtener otra equivalente donde se ha dado valor a un subconjunto de los parámetros.

4.5.5.2.13 Decoradores

Número: 0058

Nombre: Decoradores.

Categoría: Definiciones.

Descripción: Se debe facilitar un mecanismo para definir decoradores. Un decorador será un tipo especial de función. Al igual que una función se define mediante un identificador que lo nomina, una lista de parámetros y un bloque de sentencias.

A diferencia de las funciones ordinarias, la llamada a un decorador deberá tener como parámetro una función que será decorada, como resultado se deberá obtener una función que tendrá las siguientes características:

- La lista de parámetros que admite será la misma que la lista con la que se definió el decorador
- El bloque de sentencias será el del decorador pero haciendo uso de la función que ha sido decorada

Se debe facilitar un mecanismo para referenciar la función que se va a decorar dentro del decorador. Para ello se utilizará la función de contexto.

4.5.5.2.14 Función de contexto

Número: 0059

Nombre: Función de contexto.

Categoría: Definiciones.

Descripción: Se debe facilitar un mecanismo para acceder a la función de contexto. Esta será una función cuyo valor dependerá del contexto en el que se ejecute:

- En el primer nivel de ejecución la función de contexto no estará definida.
- En el cuerpo de una función será la propia función.
- En el cuerpo de un decorador será la función que se decorará.

4.5.5.3. Clases de objetos

4.5.5.3.1 Clase de objeto

Número: 0060

Nombre: Clase de objeto.

Categoría: Definiciones.

Descripción: El lenguaje debe contemplar el paradigma de la programación orientada a objetos.

Una clase se ha de ver como una definición estática e inmutable que será utilizada para la creación de objetos.

Las clases definirán tipos de objetos que tendrán métodos y atributos comunes. Una clase se construye mediante un identificador que le da nombre y un bloque de sentencias que contendrá una serie de funciones (métodos) y símbolos variables (atributos).

Las características de la programación orientada a objetos que se deberán contemplar son:

Abstracción: Un objeto por si mismo representará una entidad abstracta que podrá tener cierta funcionalidad asociada, disponer de atributos que establezcan su estado interno o comunicarse con otros objetos.

Encapsulamiento: Un objeto podrá contener todos los elementos correspondiente a su definición, estado y funcionalidad.

Principio de ocultación: Un objeto podrá tener atributos y/o métodos privados, de forma que sólo sean accesibles desde el propio objeto.

Polimorfismo: Se debe permitir a objetos de distinto tipo se le pueda enviar mensajes sintácticamente iguales, de forma que se pueda llamar un método de objeto sin tener que conocer su tipo.

Herencia: Se debe contemplar la herencia simple entre clases de forma que una clase se pueda definir mediante otra.

4.5.5.3.2 Objeto

Número: 0061

Nombre: Objeto.

Categoría: Definiciones.

Descripción: El sistema debe ser capaz de interpretar y operar sobre objetos. Los objetos serán vistos como estructuras de datos funcionales, formado tanto por datos de diferentes tipos (atributos), como por funciones (métodos).

Se podrá crear objetos a partir de una clase ya definida. Las clases de objetos pueden definir un método constructor que será utilizado cuando se cree un objeto a partir de la misma.

Dentro del bloque de sentencias que conforma un método es posible hacer referencia a los demás valores del objeto mediante la expresión especial “this”.

Las clases de objetos podrán definir un método constructor que será llamado cuando el objeto sea instanciado.

Un objeto podrá disponer de un método que será llamado cuando se precise su conversión a un tipo de dato cadena de caracteres. También podrán disponer de métodos que serán llamados cuando se acceda a un atributo o método no existente.

4.5.5.3.3 Elementos privados

Número: 0062

Nombre: Elementos privados.

Categoría: Clases de Objetos.

Descripción: Se debe facilitar un mecanismo para definir atributos y métodos de una clase de objetos como privados. Estos elementos solo serán accesibles desde métodos del propio objeto. Se deberá contemplar el acceso a estos elementos sobre objetos del mismo tipo dentro de métodos de la clase.

4.5.5.3.4 Elementos estáticos

Número: 0063

Nombre: Elementos estáticos.

Categoría: Clases de Objetos.

Descripción: Se debe facilitar un mecanismo para definir atributos y métodos de una clase de objetos pertenecientes a la propia clase. Estos elementos no serán trasladados a los objetos instanciados.

4.5.5.3.5 Herencia de clases

Número: 0064

Nombre: Herencia de clases.

Categoría: Clases de Objetos.

Descripción: Se debe de disponer de un mecanismo que permita establecer una relación de herencia entre unas clases dadas. Así será posible la definición de nuevas clases partiendo de otras. La clase derivará de otra extendiendo su funcionalidad y definición.

En la definición de una clase se debe de disponer de un mecanismo que permita especificar la clase que se extenderá. La nueva clase tendrá todos los atributos y métodos de la extendida y añadirá los suyos propios, pudiendo sobrescribirse los ya existentes.

4.5.5.3.6 Instanciación de clases

Número: 0065

Nombre: Instanciación de clases.

Categoría: Clases de Objetos.

Descripción: Dada una clase, se debe de disponer de un mecanismo que permita crear objetos a partir de la misma. Para construir un objeto a partir de la instancia de una clase se deben llevar las funciones y variables definidas en el cuerpo de la clase a métodos y atributos del objeto.

Una clase puede definir un método constructor que deberá ser llamado sobre el objeto recién creado cuando la clase es instanciada.

La instancia se deberá realizar mediante un operador que, a partir de un identificador correspondiente a la clase y una lista de expresiones correspondientes a los parámetros del método constructor, tome como valor el objeto recién creado.

4.5.5.3.7 Acceso al objeto en ejecución

Número: 0066

Nombre: Acceso al objeto en ejecución.

Categoría: Clases de Objetos.

Descripción: Se debe de disponer de un mecanismo que permita acceder a los atributos y métodos de un objeto desde la ejecución de un método del mismo. Este mecanismo, correspondiente a una expresión, deberá tomar como valor el objeto en ejecución.

4.5.5.3.8 Acceso al objeto en ejecución como clase padre

Número: 0067

Nombre: Acceso al objeto en ejecución como clase padre.

Categoría: Clases de Objetos.

Descripción: Se debe de disponer de un mecanismo que permita acceder a los atributos y métodos de la clase padre de un objeto desde la ejecución de un método del mismo. Este mecanismo, correspondiente a una expresión, deberá tomar como valor el objeto en ejecución, pero tomando como métodos y atributos los de la clase padre de la cual deriva.

4.5.5.3.9 Enlace estático en tiempo de ejecución

Número: 0068

Nombre: Enlace estático en tiempo de ejecución.

Categoría: Clases de Objetos.

Descripción: Se debe de disponer de un mecanismo que permita acceder a los atributos y métodos estáticos de una clase hija desde un método estático de la clase padre.

4.5.5.3.10 Obtener clase

Número: 0069

Nombre: Obtener clase

Categoría: Clases de Objetos.

Descripción: Se debe de disponer de un mecanismo para, a partir de un objeto, obtener la clase a la que pertenece como una cadena correspondiente al nombre.

4.5.5.4. Listas

Nombre: Listado.

Categoría: Definiciones.

Número: 0070

Descripción: Se ha de facilitar un mecanismo que permita agrupar expresiones para darles un significado operacional común. Este deberá consistir en una serie de expresiones separadas por comas.

4.5.5.5. Pares clave/valor

Nombre: Pares clave/valor.

Categoría: Definiciones.

Número: 0071

Descripción: Se ha de facilitar un mecanismo que permita relacionar un par de expresiones para darles un significado estructural. Este deberá consistir en el par de expresiones separadas por el carácter ":".

4.5.5.6. Etiqueta

Número: 0072

Nombre: Etiqueta.

Categoría: Definiciones.

Descripción: El sistema debe ser capaz de interpretar y operar sobre etiquetas. Una etiqueta es una referencia a una sentencia concreta dentro del contenido fuente. Las etiquetas dependen quedarán definidas dentro de un contexto determinado por el bloque de sentencias en el que se encuentren.

4.5.5.7. Listas por comprensión

Número: 0073

Nombre: Listas por comprensión.

Categoría: Definiciones.

Descripción: Se necesita de un mecanismo que sea una expresión por si mismo y que permita generar arrays desde una sentencia iterativa. Este mecanismo se formará mediante una expresión seguida da una sentencia for. La expresión será ejecutada tras iteración del bucle y será asignada como último elemento de un array. Al final de la ejecución la expresión tomará el valor del array generado.

4.5.5.8. Expresiones reflexivas

Número: 0074

Nombre: Expresiones reflexivas.

Categoría: Definiciones.

Descripción: Se ha de disponer de un mecanismo de reflexión, que permita utilizar expresiones del lenguaje para definir otras expresiones como identificadores.

4.5.6. Asignaciones

4.5.6.1. Asignación

Número: 0075

Nombre: Asignación.

Categoría: Asignaciones.

Descripción: Se hace necesario la gestión de los símbolos variables creados durante la ejecución, lo que implica la asignación de valores a las variables que serán definidas y utilizadas por el contenido fuente dado por el usuario. El valor que es asignado a una variable puede ser cualquier tipo de dato contemplado, incluso funciones o objetos. El valor asignado puede ser determinado a partir de cualquier expresión que tenga un valor asociado después de su

ejecución. La operación de asignación debe ser en sí misma una expresión que toma como valor tras su ejecución el valor asignado.

4.5.6.2. Asignación de referencia

Número: 0076

Nombre: Asignación de referencia.

Categoría: Asignaciones.

Descripción: Se debe facilitar un mecanismo para que dos símbolos variables distintos refieran al mismo valor. Para ello se ha de facilitar un mecanismo para obtener la referencia de un símbolo variable, de forma que esta pueda ser usada en una asignación.

4.5.7. Operadores aritméticos

4.5.7.1. Suma

Número: 0077

Nombre: Suma.

Categoría: Operadores aritméticos.

Descripción: Se debe contemplar la expresión correspondiente a la operación aritmética “suma”.

Para realizar esta operación se deberá tomar el valor aritmético de cada operando. Tras realizarse la operación, el valor de la expresión deberá ser el resultado aritmético de la misma.

4.5.7.2. Diferencia

Número: 0078

Nombre: Diferencia.

Categoría: Operadores aritméticos.

Descripción: Se debe contemplar la expresión correspondiente a la operación aritmética “resta”.

Para realizar esta operación se deberá tomar el valor aritmético de cada operando. Tras realizarse la operación, el valor de la expresión deberá ser el resultado aritmético de la misma.

4.5.7.3. Producto

Número: 0079

Nombre: Producto.

Categoría: Operadores aritméticos.

Descripción: Se debe contemplar la expresión correspondiente a la operación aritmética “producto”. Para realizar esta operación se deberá tomar el valor aritmético de cada operando. Tras realizarse la operación, el valor de la expresión deberá ser el resultado aritmético de la misma.

4.5.7.4. División

Número: 0080

Nombre: División.

Categoría: Operadores aritméticos.

Descripción: Se debe contemplar la expresión correspondiente a la operación aritmética “división”. Para realizar esta operación se deberá tomar el valor aritmético de cada operando. El segundo operando debe ser distinto de 0. Si el segundo operando tiene valor aritmético 0 se deberá mostrar un error que informe del caso. Tras realizarse la operación, el valor de la expresión deberá ser el resultado aritmético de la misma.

4.5.7.5. Potencia

Número: 0081

Nombre: Potencia.

Categoría: Operadores aritméticos.

Descripción: Se debe contemplar la expresión correspondiente a la operación aritmética “potencia”. Para realizar esta operación se deberá tomar el valor aritmético de cada operando. Tras realizarse la operación, el valor de la expresión deberá ser el resultado aritmético de la misma.

4.5.7.6. Módulo

Número: 0082

Nombre: Módulo.

Categoría: Operadores aritméticos.

Descripción: Se debe contemplar la expresión correspondiente a la operación aritmética “módulo”. Para realizar esta operación se deberá tomar el valor aritmético de cada operando. El segundo operando debe ser distinto de 0. Si el segundo operando tiene valor aritmético 0 se deberá mostrar un error que informe del caso. Tras realizarse la operación, el valor de la expresión deberá ser el resultado aritmético de la misma.

4.5.7.7. Tamaño

Número: 0083

Nombre: Tamaño.

Categoría: Operadores aritméticos.

Descripción: Se precisa de algún mecanismo que dado un dato calcule el tamaño de este. Este operador calculará el tamaño dependiendo del tipo de dato del operando. Tras ejecutarse la expresión el valor que tome será de tipo aritmético.

Lógico: Si es verdadero el tamaño es uno, si es falso será cero.

Aritmético: Tomará el número de dígitos decimales.

Cadena: El tamaño será el número de caracteres de la cadena.

Array: Para el tipo de dato array u otros derivados se el tamaño será el número de elementos contenidos en el mismo.

Otro tipo de dato: Se deberá dar un error de tipos.

4.5.7.8. Incremento y asignación

Número: 0084

Nombre: Incremento y asignación.

Categoría: Operadores aritméticos.

Descripción: Dado un identificador o expresión que referencia a una dato variable, el valor de esta se debe poder incrementar y reasignar. Para ello se tomará el valor aritmético de la variable, se incrementará en una unidad y se reasignará a la misma. El valor de la expresión será el de la variable incrementada.

4.5.7.9. Asignación e incremento

Número: 0085

Nombre: Asignación e incremento.

Categoría: Operadores aritméticos.

Descripción: Dado un identificador o expresión que referencia a una dato variable, el valor de esta se debe poder incrementar y reasignar. Para ello se tomará el valor aritmético de la variable, se incrementará en una unidad y se reasignará a la misma. El valor de la expresión será el de la variable antes de ser incrementada.

4.5.7.10. Decremento y asignación

Número: 0086

Nombre: Decremento y asignación.

Categoría: Operadores aritméticos.

Descripción: Dado un identificador o expresión que referencia a una dato variable, el valor de esta se debe poder decrementar y reasignar. Para ello se tomará el valor aritmético de la variable, se decrementará en una unidad y se reasignará a la misma. El valor de la expresión será el de la variable decrementada.

4.5.7.11. Asignación y decremento

Número: 0087

Nombre: Asignación y decremento.

Categoría: Operadores aritméticos.

Descripción: Dado un identificador o expresión que referencia a una dato variable, el valor de esta se debe poder decrementar y reasignar. Para ello se tomará el valor aritmético de la variable, se decrementará en una unidad y se reasignará a la misma. El valor de la expresión será el de la variable antes de ser decrementada.

4.5.7.12. Suma y asignación

Número: 0088

Nombre: Suma y asignación.

Categoría: Operadores aritméticos.

Descripción: Dado un identificador o expresión que referencia a una dato variable, al valor de esta se ha de poder sumar otra expresión y reasignarle el resultado. Para ello se tomará el valor aritmético de la variable, se le sumará el valor aritmético de la expresión y se reasignará a la variable el resultado. El valor de la expresión será el resultado de la suma aritmética.

4.5.7.13. Diferencia y asignación

Número: 0089

Nombre: Diferencia y asignación.

Categoría: Operadores aritméticos.

Descripción: Dado un identificador o expresión que referencia a una dato variable, al valor de esta se ha de poder restar otra expresión y reasignarle el resultado. Para ello se tomará el valor aritmético de la variable, se le restará el valor aritmético de la expresión y se reasignará a la variable el resultado. El valor de la expresión será el resultado de la resta aritmética.

4.5.7.14. Producto y asignación

Número: 0090

Nombre: Producto y asignación.

Categoría: Operadores aritméticos.

Descripción: Dado un identificador o expresión que referencia a una dato variable, al valor de esta se ha de poder multiplicar otra expresión y reasignarle el resultado. Para ello se tomará el valor aritmético de la variable, se calculará el producto y se reasignará a la variable el resultado. El valor de la expresión será el resultado del producto aritmético.

4.5.7.15. División y asignación

Número: 0091

Nombre: División y asignación.

Categoría: Operadores aritméticos.

Descripción: Dado un identificador o expresión que referencia a una dato variable, al valor de esta se ha de poder dividir por otra expresión y reasignarle el resultado. Para ello se tomará el valor aritmético de la variable, se realizará la división y se reasignará a la variable el resultado. La expresión no puede tener un valor aritmético de cero. El valor de la expresión será el resultado de la división aritmética.

4.5.7.16. Potencia y asignación

Número: 0092

Nombre: Potencia y asignación.

Categoría: Operadores aritméticos.

Descripción: Dado un identificador o expresión que referencia a una dato variable, al valor de esta se ha de poder elevar a otra expresión y reasignarle el resultado. Para ello se tomará el valor aritmético de la variable, se elevará y se reasignará a la variable el resultado. El valor de la expresión será el resultado de la potencia.

4.5.7.17. Módulo y asignación

Número: 0093

Nombre: Módulo y asignación.

Categoría: Operadores aritméticos.

Descripción: Dado un identificador o expresión que referencia a una dato variable, al valor de esta se ha de poder dividir por otra expresión y reasignarle el resto originado. Para ello se tomará el valor aritmético de la variable, se realizará la división por el valor aritmético de la expresión y se reasignará a la variable el resto obtenido. La expresión no puede tener un valor aritmético de cero. El valor de la expresión será el resultado de la operación módulo.

4.5.8. Operadores lógicos

4.5.8.1. AND lógico

Número: 0094

Nombre: AND lógico.

Categoría: Operadores lógicos.

Descripción: Se debe contemplar la expresión correspondiente a la operación lógica AND. Para ello se deberá tomar el valor lógico de cada uno de los operandos. La evaluación de la operación lógica AND debe ser de cortocircuito, tomándose el valor del último elemento evaluado. Así, aunque esta expresión se corresponde con un operador lógico, el valor de la misma será el del último elemento evaluado.

4.5.8.2. OR lógico

Número: 0095

Nombre: OR lógico.

Categoría: Operadores lógicos.

Descripción: Se debe contemplar la expresión correspondiente a la operación lógica OR. Para ello se deberá tomar el valor lógico de cada uno de los operandos. La evaluación de la operación lógica OR debe ser de cortocircuito, tomándose el valor del último elemento evaluado. Así, aunque esta expresión se corresponde con un operador lógico, el valor de la misma será el del último elemento evaluado.

4.5.8.3. NOT lógico

Número: 0096

Nombre: NOT lógico.

Categoría: Operadores lógicos.

Descripción: Se debe contemplar la expresión correspondiente a la operación lógica NOT. Para ello se deberá tomar el valor lógico de su único operando y negarlo. La expresión deberá tomar un valor de tipo booleano tras realizarse la operación.

4.5.8.4. Vacío

Número: 0097

Nombre: Vacío.

Categoría: Operadores lógicos.

Descripción: Se necesita de un operador que determine si un dato se considera vacío. Este operador tendrá un único operando y funcionará igual que el operador lógico NOT. El valor que tomará la expresión será lógico.

4.5.8.5. Es nulo

Número: 0098

Nombre: Es nulo.

Categoría: Operadores lógicos.

Descripción: Se necesita de un operador que determine si un dato o expresión contiene el valor nulo.

4.5.8.6. Menor que

Número: 0099

Nombre: Menor que.

Categoría: Operadores de comparación.

Descripción: Se debe contemplar la expresión correspondiente a la operación lógica “menor que”. Para ello se deberá tomar el valor aritmético de cada operando. La expresión deberá tomar un valor de tipo booleano tras realizarse la operación.

4.5.8.7. Menor o igual que

Número: 0100

Nombre: Menor o igual que.

Categoría: Operadores de comparación.

Descripción: Se debe contemplar la expresión correspondiente a la operación lógica “menor o igual que”. Para ello se deberá tomar el valor aritmético de cada operando. La expresión deberá tomar un valor de tipo booleano tras realizarse la operación.

4.5.8.8. Mayor que

Número: 0101

Nombre: Mayor que.

Categoría: Operadores de comparación.

Descripción: Se debe contemplar la expresión correspondiente a la operación lógica “mayor que”. Para ello se deberá tomar el valor aritmético de cada operando. La expresión deberá tomar un valor de tipo booleano tras realizarse la operación.

4.5.8.9. Mayor o igual que

Número: 0102

Nombre: Mayor o igual que.

Categoría: Operadores de comparación.

Descripción: Se debe contemplar la expresión correspondiente a la operación lógica “mayor o igual que”. Para ello se deberá tomar el valor aritmético de cada operando. La expresión deberá tomar un valor de tipo booleano tras realizarse la operación.

4.5.8.10. Igual que

Número: 0103

Nombre: Igual que.

Categoría: Operadores de comparación.

Descripción: Se debe contemplar la expresión correspondiente a la operación lógica “igual que”.

La operación de igualdad debe ser independiente de los tipos de datos de los operandos, aplicándose en función del tipo de dato más completo que compartan. Por ejemplo si se compara un dato cadena que no representa un número racional con uno aritmético, como el tipo de dato común a ambos es el booleano, ambos tomarán su valor lógico para la comparación. Si ambos datos son tipos compuestos, se ha de comprobar mediante la operación de igualdad todos los elementos simples que lo componen por pares y de forma posicional. Como valor de la expresión se toma el valor booleano de la operación.

4.5.8.11. Idéntico que

Número: 0104

Nombre: Idéntico que.

Categoría: Operadores de comparación.

Descripción: Se debe contemplar la expresión correspondiente a la operación lógica “idéntico que”. Esta operación se refiere a una operación lógica de igualdad pero contemplando además que los datos tengan el mismo tipo. Como valor de la expresión se debe tomar el valor booleano resultado de aplicar la operación.

4.5.8.12. Distinto que

Número: 0105

Nombre: Distinto que.

Categoría: Operadores de comparación.

Descripción: Se debe contemplar la expresión correspondiente a la operación lógica “distinto que”. Esta operación debe ser independiente de los tipos de datos de los operandos, aplicándose en función del tipo de dato más completo que compartan. Por ejemplo si se compara un dato cadena que no representa un número racional con uno aritmético, como el valor más completo que ambos pueden tomar es el booleano, tomarán su valor lógico para la comparación. Si ambos datos son tipos compuestos, se ha de comprobar mediante la operación de igualdad todos los elementos simples que lo componen por pares y de forma posicional. Como valor de la expresión se toma el valor booleano de la operación.

4.5.8.13. No idéntico que

Número: 0106

Nombre: No idéntico que.

Categoría: Operadores de comparación.

Descripción: Se debe contemplar la expresión correspondiente a la operación lógica “no idéntico que”. Esta operación se corresponde con la operación inversa de la operación “idéntico que”. Como valor de la expresión se debe tomar el valor booleano resultado de aplicar la operación.

4.5.9. Operadores sobre cadenas

4.5.9.1. Concatenación

Número: 0107

Nombre: Operador concatenación.

Categoría: Operadores sobre cadena de caracteres.

Descripción: La expresión que simboliza una operación de concatenación precisa de dos operandos que serán tratados como cadena de caracteres. El valor que tomará la expresión será la cadena resultante de concatenar ambas.

4.5.9.2. explode

Número: 0108

Nombre: Función explode.

Categoría: Operadores sobre cadena de caracteres.

Descripción: La función explode deberá tomar dos cadenas de caracteres como operandos denominadas “texto” y “separador”. El valor será el array resultante de separar la cadena “texto” en diferentes cadenas en función la cadena “separador”.

4.5.9.3. implode

Número: 0109

Nombre: Función implode.

Categoría: Operadores sobre cadena de caracteres.

Descripción: Representa la operación inversa a explode. La función implode deberá tomar como argumentos un array denominado “listado” y una cadena denominada “separador”. El valor de la expresión será una cadena resultado de concatenar cada uno de los elementos de “listado” separados por la cadena “separador”.

4.5.9.4. sprintf

Número: 0110

Nombre: Función de formato.

Categoría: Operadores sobre cadena de caracteres.

Descripción: Se hace necesario un mecanismo que permita generar cadenas formateadas. Este deberá consistir en una cadena de caracteres denominada “formato” y un listado de expresiones. La cadena formato contendrá una serie de directivas de formato. Estas directivas serán sustituidas por el valor correspondiente, según posición, de la lista de expresiones. Cuando se realiza cada sustitución el valor es formateado según la directiva.

Las directivas de formato tienen el siguiente forma:

$$\%[\text{operador}][\text{precisión}][\text{formato}]$$

Los posibles operadores serán los siguientes:

+: Fuerza la impresión del símbolo + cuando se formatean números positivos.

^ : Convierte el caracteres a mayúsculas cuando se formatean cadenas de texto.

#: Añade el carácter 0x cuando se formatean números hexadecimales y el carácter 0 cuando se formatean octales.

La precisión se refiere al número de decimales que se imprimirán en el caso de formatear números o el número de caracteres en el caso de formatear cadenas.

El carácter de formato indica que tipo de formato se le dará al valor:

i|d: Número entero.

u: Sin signo.

f: Coma flotante.

%: Carácter %.

e: Notación científica.

o: Octal.

x: Hexadecimal.

s|c: Cadena de texto.

4.5.9.5. Buscar subcadena

Número: 0111

Nombre: Función de búsqueda de subcadena.

Categoría: Operadores sobre cadena de caracteres.

Descripción: Esta es una función básica en el tratamiento de cadenas. Opera sobre dos argumentos que serán tratados como cadenas de caracteres, uno denominado “texto” y otro “subcadena”. La función toma como valor un dato aritmético relativo a la posición de la primera ocurrencia de “subcadena” dentro de “texto”. Si no se encuentra ningún resultado se tomará el valor nulo. Adicionalmente se puede dar otro operando denominado “offset” que simbolice la posición dentro de “texto” a partir de la cual se comenzará a buscar.

4.5.9.6. Buscar y remplazar

Número: 0112

Nombre: Función de remplazo de subcadena.

Categoría: Operadores sobre cadena de caracteres.

Descripción: Se necesita de un mecanismo para buscar y remplazar subcadenas dentro de otra. Este debe buscar las ocurrencias de una subcadena “búsqueda” en una cadena “texto”, sustituyéndolas por una cadena de “reemplazo”. Esta función debe admitir el número máximo de sustituciones que se llevarán a cabo. Tras la ejecución debe tomar como valor la cadena resultante de sustituir en la cadena principal las ocurrencias de la subcadenas por la cadena de reemplazo.

Adicionalmente la subcadena de “búsqueda” puede ser una expresión regular, en cuyo caso se buscará subcadenas que pertenezcan al conjunto de las palabras definido por la expresión regular.

Si se utiliza una expresión regular como patrón de búsqueda deberá ser posible utilizar en la cadena de reemplazo parte de la subcadena que concuerda con la expresión regular. Para ello se ha de formar la expresión regular mediante subexpresiones delimitadas por “()”. En la cadena de reemplazo se debe poder hacer referencia, de forma posicional, a las subcadenas correspondientes a cada una de las subexpresiones.

4.5.9.7. Remplazar subcadena

Número: 0113

Nombre: Función de remplazo de subcadena mediante posiciones.

Categoría: Operadores sobre cadena de caracteres.

Descripción: Se necesita de un mecanismo para buscar y remplazar subcadenas dentro de otra. Este debe sustituir en una cadena “texto” la subcadena comprendida entre dos posiciones dadas por expresiones numéricas, sustituyendo la subcadena correspondiente por una cadena de “reemplazo”. Tras la ejecución debe tomar como valor la cadena resultante de sustituir, en la cadena principal, la subcadena correspondiente a las pociones dadas por la cadena de reemplazo.

4.5.9.8. Convertir a mayúsculas

Número: 0114

Nombre: Función conversión a mayúsculas.

Categoría: Operadores sobre cadena de caracteres.

Descripción: Dada una cadena de caracteres se necesita de un mecanismo que convierta todos los caracteres alfabéticos que la conforman en mayúsculas. El valor que se tomará será la cadena resultante de la operación.

4.5.9.9. Convertir a minúsculas

Número: 0115

Nombre: Función conversión a minúsculas.

Categoría: Operadores sobre cadena de caracteres.

Descripción: Dada una cadena de caracteres se necesita de un mecanismo que convierta todos los caracteres alfabéticos que la conforman en minúsculas. El valor que se tomará será la cadena resultante de la operación.

4.5.10. Operadores sobre array

4.5.10.1. Dividir en fragmentos

Número: 0116

Nombre: Función dividir array en fragmentos.

Categoría: Operadores sobre array.

Descripción: Se precisa de un mecanismo que, dado un array y un valor aritmético. Se divida el array en varios subarray de tantos elementos como indique el valor (a excepción del último). La expresión tomará como valor un array que contiene cada uno de los subarray.

4.5.10.2. Reducir mediante función

Número: 0117

Nombre: Función reducir array.

Categoría: Operadores sobre array.

Descripción: Se necesita de un mecanismo que dado un array y una función reduzca el array a un solo valor. La función de reducción deberá recibir como parámetro dos valores correspondiente al valor acumulado y al nuevo valor. La función de reducción se ejecutará por cada elemento del array (excepto para el primero) tomando el valor acumulado y el nuevo valor, y devolviendo el próximo valor acumulado. Como valor este operador tomará el valor de la reducción.

4.5.10.3. Obtener último

Número: 0118

Nombre: Función obtener último elemento de array.

Categoría: Operadores sobre array.

Descripción: Se precisa de un mecanismo que, dado un array se obtenga el último elemento de este, o el valor nulo si este está vacío.

4.5.10.4. Obtener primero

Número: 0119

Nombre: Función obtener primer elemento de array.

Categoría: Operadores sobre array.

Descripción: Se precisa de un mecanismo que, dado un array se obtenga el primer elemento de este, o el valor nulo si este está vacío.

4.5.10.5. Insertar en posición

Número: 0120

Nombre: Función para insertar un elemento en una posición de array.

Categoría: Operadores sobre array.

Descripción: Se precisa de un mecanismo que, dado un array, un elemento y una expresión aritmética que hace de posición, se inserte el elemento en dicha posición del array. Si la posición se encuentra fuera de rango se dará un error de acceso.

4.5.10.6. Eliminar posición

Número: 0121

Nombre: Función eliminar elemento de array de una posición.

Categoría: Operadores sobre array.

Descripción: Se precisa de un mecanismo que, dado un array y una expresión aritmética que hace de posición, se elimine el elemento que ocupa dicha posición dentro del array. Si la posición se encuentra fuera de rango se dará un error de acceso.

4.5.10.7. Insertar al inicio

Número: 0122

Nombre: Función insertar al inicio de un array.

Categoría: Operadores sobre array.

Descripción: Se precisa de un mecanismo que, dado un array y un elemento de un tipo arbitrario, se inserte el elemento al comienzo del array.

4.5.10.8. Insertar al final

Número: 0123

Nombre: Función insertar al final de un array.

Categoría: Operadores sobre array.

Descripción: Se precisa de un mecanismo que, dado un array y un elemento de un tipo arbitrario, se inserte el elemento al final del array.

4.5.10.9. Eliminar del inicio

Número: 0124

Nombre: Función eliminar del comienzo de un array.

Categoría: Operadores sobre array.

Descripción: Se precisa de un mecanismo que, dado un array, se elimine el primer elemento del mismo. Como valor se tomará el elemento eliminado del array, o el valor nulo si este se encontraba vacío.

4.5.10.10. Eliminar del final

Número: 0125

Nombre: Función eliminar del final de un array.

Categoría: Operadores sobre array.

Descripción: Se precisa de un mecanismo que, dado un array, se elimine el último elemento del mismo. Como valor se tomará el elemento eliminado del array, o el valor nulo si este se encontraba vacío.

4.5.11. Operadores sobre expresiones regulares

4.5.11.1. Crear expresión regular

Número: 0126

Nombre: Operador creador de expresión regular.

Categoría: Operadores sobre expresiones regulares.

Descripción: Dada una cadena de caracteres se necesita de un operador que convierta esta en una expresión regular. El valor del operador será la expresión regular.

4.5.11.2. Comprobar expresión regular

Número: 0127

Nombre: Función comprobación de expresión regular.

Categoría: Operadores sobre expresiones regulares.

Descripción: Se precisa un operador que, dada una expresión regular y una cadena de caracteres, compruebe si esta pertenece al lenguaje definido por la expresión regular. Se deberá tomar como valor un dato de tipo lógico resultado de la operación.

Para que el resultado sea positivo la cadena debe pertenecer al conjunto de palabras delimitadas por la expresión regular. Si tan solo existe correspondencia parcial el resultado será negativo.

4.5.11.3. Buscar mediante expresión regular

Número: 0128

Nombre: Función de búsqueda estructurada.

Categoría: Operadores sobre expresiones regulares.

Descripción: Se precisa de un mecanismo que lleve a cabo una búsqueda estructurada, es decir, obtener una estructura de datos array condicionada por una expresión regular denominada “patrón de búsqueda” y una cadena de caracteres “texto” sobre la que se comprueba.

En la ejecución de este operador se deberá buscar en “texto” subcadenas que pertenezcan al conjunto definido por la expresión regular, originando un array con cada una de las coincidencias, que será el valor que tome la operación.

Adicionalmente la expresión regular podría estar formada por subexpresiones delimitadas por “()”. En dicho caso se buscará en “texto” subcadenas que pertenezcan al conjunto delimitado por la expresión regular. Por cada subcadena encontrada se creará un array con las correspondencias de cada subexpresión. Cada uno de los arrays resultantes se deberán guardar en otro que será el valor que tome la operación.

Si se utiliza una expresión regular formada por subexpresiones los arrays correspondientes a cada subcadena deberán tener índices numéricos. Sin embargo debe darse la posibilidad de especificar una lista ordenada de cadenas claves para crear un array asociativo.

Además se ha de contemplar la búsqueda estructurada sobre un array de cadenas “texto”.

4.5.12. Operadores de conversión de tipo

4.5.12.1. Conversión a numérico

Número: 0129

Nombre: Conversión a numérico.

Categoría: Operadores de conversión de tipo.

Descripción: Se ha de facilitar un operador que permita convertir un dato a tipo aritmético.

Esta conversión se deberá realizar en función al tipo de dato origen y de la forma descrita en el requisito en el que se hace referencia al mismo. El valor que tomará la operación deberá ser el dato tras la conversión de tipos. La conversión se podrá realizar a un número entero o flotante.

4.5.12.2. Conversión a lógico

Número: 0130

Nombre: Conversión a lógico.

Categoría: Operadores de conversión de tipo.

Descripción: Se ha de facilitar un operador que permita convertir un dato a tipo lógico.

Esta conversión se deberá realizar en función al tipo de dato origen y de la forma descrita en el requisito en el que se hace referencia al mismo. El valor que tomará la operación deberá ser el dato tras la conversión de tipos.

4.5.12.3. Conversión a cadena de caracteres

Número: 0131

Nombre: Conversión a cadena de caracteres.

Categoría: Operadores de conversión de tipo.

Descripción: Se ha de facilitar un operador que permita convertir un dato a tipo cadena de caracteres. Esta conversión se deberá realizar en función al tipo de dato origen y de la forma descrita en el requisito en el que se hace referencia al mismo. El valor que tomará la operación deberá ser el dato tras la conversión de tipos.

4.5.13. Operadores de entrada/salida

4.5.13.1. Escribir en la salida estándar

Número: 0132

Nombre: Escribir en la salida estándar.

Categoría: Operadores de entrada/salida.

Descripción: Se debe disponer de una serie de mecanismos que permitan llevar a cabo la salida de datos. Esta permitirán que el contenido fuente pueda mostrar en la salida estándar los datos sobre los que opera. Estos datos deben tener una representación gráfica y ser imprimibles.

4.5.14. Leer de la entrada estándar

Número: 0133

Nombre: Leer de la entrada estándar.

Categoría: Operadores de entrada/salida.

Descripción: El intérprete debe implementar algún recurso que permita que el contenido fuente del usuario obtenga datos de la entrada estándar para operar. Este mecanismo deberá dar la posibilidad de mostrar un prompt. Además debe dar la opción de que la entrada finalice al introducir un salto de línea o al generarse una señal EOF (end-of-file).

4.5.15. Operadores informativos

4.5.15.1. Tipo de

Número: 0134

Nombre: Obtener tipo de dato.

Categoría: Operadores informativos.

Descripción: Debido al tipado dinámico se precisa de un mecanismo para conocer el tipo de dato relacionado con una variable. Este deberá mostrar en la salida estándar el tipo de la variable asociada a un identificador dado.

4.5.15.2. Tamaño de

Número: 0135

Nombre: Obtener tamaño de dato.

Categoría: Operadores informativos.

Descripción: Se precisa de un mecanismo para obtener el tamaño en memoria que ocupa el valor referenciado por una variable.

4.5.15.3. Información sobre

Número: 0136

Nombre: Obtener información interna de un dato.

Categoría: Operadores informativos.

Descripción: Se precisa de un mecanismo para obtener información estructural de un dato, cómo este es representado por el intérprete y el tamaño que ocupa.

4.5.16. Procesos

4.5.16.1. Ejecutar comando

Número: 0137

Nombre: Ejecutar comando.

Categoría: Procesos.

Descripción: Se debe facilitar un operador que ejecute un comando dado. Para ello se deberá usar el interprete de comandos definido por el entorno del sistema operativo. Este operador contemplará un único operando correspondiente al comando a ejecutar. Como valor se deberá tomar la cadena de caracteres correspondiente a la salida del comando.

4.5.16.2. Evaluar cadena

Número: 0138

Nombre: Evaluar cadena.

Categoría: Procesos.

Descripción: Deberá existir un operador que utilice el interprete para procesar una cadena de caracteres escrita en el léxico y con la sintaxis del propio lenguaje. Este operador tomará como valor una cadena de caracteres relativa a la salida generada por la interpretación.

4.5.16.3. Bifurcación de proceso

Número: 0139

Nombre: Bifurcación de proceso.

Categoría: Procesos.

Descripción: Se necesita de un mecanismo que bifurque el flujo de ejecución mediante la creación de un proceso hijo. El intérprete deberá crear un proceso clonado de si mismo, cuya ejecución proseguirá en el mismo punto. El operador de bifurcación tomará valor aritmético cero en el proceso hijo, mientras que en el padre tomará el valor aritmético correspondiente al identificador del proceso hijo.

4.5.16.4. Espera entre procesos

Número: 0140

Nombre: Espera entre procesos.

Categoría: Procesos.

Descripción: Se necesita de un mecanismo que permita hacer que la ejecución de un proceso padre espere a que todos o algunos de sus hijos finalicen su ejecución. Este podrá operar sobre un dato aritmético que referenciará al identificador de proceso del hijo que se ha de esperar. El valor que tomará consistirá en el código correspondiente a la señal de salida producida por el último proceso finalizado.

4.5.16.5. Obtener identificador de proceso

Número: 0141

Nombre: Obtener identificador de proceso.

Categoría: Procesos.

Descripción: Todo proceso tiene un identificador único en el sistema sobre el que se ejecuta. Se deberá disponer de un operador que tome el valor del identificador de proceso correspondiente al intérprete.

4.5.16.6. Obtener identificador de proceso padre

Número: 0142

Nombre: Obtener identificador de proceso padre.

Categoría: Procesos.

Descripción: Debe existir algún mecanismo que permita obtener el identificador del proceso padre cuando el interprete se ejecuta como proceso hijo de otro.

4.5.16.7. Señales entre procesos

Número: 0143

Nombre: Señales entre procesos.

Categoría: Procesos.

Descripción: Debe existir algún mecanismo que permita mandar señales entre procesos. Estas señales se corresponderán con señales UNIX. Para mandar una señal a un proceso se deberá dar un identificador de proceso y un entero correspondiente a la señal a enviar.

4.5.16.8. Manejador de señales a procesos

Número: 0144

Nombre: Manejador de señales a procesos.

Categoría: Procesos.

Descripción: Debe existir algún mecanismo que permita especificar una función que será ejecutada cuando el proceso reciba una determinada señal.

4.5.16.9. Llamar a función como proceso

Número: 0145

Nombre: Llamar a función como proceso.

Categoría: Operadores sobre procesos.

Descripción: Se precisa de operador que mediante una función y un listado de parámetros realice una llamada a la misma mediante la creación de un proceso hijo.

4.5.17. Ficheros

4.5.17.1. Fichero

Número: 0146

Nombre: Fichero.

Categoría: Operadores sobre ficheros.

Descripción: El intérprete debe ser capaz de manipular ficheros, para ello se precisa de un tipo de dato que simbolice un puntero a un fichero del sistema de ficheros. Este tipo de dato no debe ser convertido a ningún otro tipo de dato ni viceversa. Además solo será tratado por algunos operadores dedicados.

No se tendrán en cuenta los ficheros binarios.

4.5.17.2. Abrir ficheros

Número: 0147

Nombre: Abrir ficheros.

Categoría: Operadores sobre ficheros.

Descripción: Es necesario un mecanismo que permita abrir ficheros para su manipulación. Este operará sobre una cadena de caracteres que simbolice la ruta al fichero y otra que determine el modo en el que será abierto. Se tomará como valor un dato de tipo puntero a fichero. Los posibles modos serán:

r: Lectura.

r+: Lectura y/o escritura.

w: Escritura truncando el contenido del fichero.

w+: Lectura y/o escritura truncando el contenido del fichero.

a: Escritura posicionando el puntero al final del fichero.

a+: Lectura y/o escritura posicionando el puntero al final del fichero.

Todos los modos a excepción de sólo lectura deberán crear el fichero si este no existe.

4.5.17.3. Cerrar ficheros

Número: 0148

Nombre: Cerrar ficheros.

Categoría: Operadores sobre ficheros.

Descripción: Es necesario un mecanismo que permita cerrar ficheros abiertos a partir de un puntero al mismo. Se deberá finalizar cualquier flujo de datos abierto y el fichero quedará cerrado. Como valor se deberá tomar un dato de tipo lógico que determine si la operación se ha realizado correctamente.

4.5.17.4. Escribir en fichero

Número: 0149

Nombre: Escribir en fichero.

Categoría: Operadores sobre ficheros.

Descripción: Se hace necesario un operador que, dado un dato de tipo puntero a fichero, pueda escribir datos en la posición referenciada por el mismo. Así este operador trabaja sobre dos operandos, un puntero a fichero y una cadena de caracteres que simbolizará el contenido a escribir. Como valor el operador toma el número de bytes que fueron escritos.

4.5.17.5. Leer de fichero

Número: 0150

Nombre: Leer de fichero.

Categoría: Operadores sobre ficheros.

Descripción: Se hace necesario un operador que, dado un dato de tipo puntero a fichero, lea desde la posición referenciada por el mismo hasta un carácter de nueva línea, o bien un número de caracteres dado. Así el operador deberá tomar como valor una cadena de caracteres que represente el contenido leído.

4.5.17.6. Cambiar posición de puntero a fichero

Número: 0151

Nombre: Cambiar posición de puntero a fichero.

Categoría: Operadores sobre ficheros.

Descripción: Una operación básica sobre punteros a ficheros es desplazar este dentro del contenido del mismo. Para ello se precisa de un operador que, dado un puntero a fichero, cambie la posición de este dentro del propio fichero. Así la nueva posición deberá ser una expresión numérica que represente un offset relativo al principio del fichero, el final o la posición actual del puntero. La expresión correspondiente al operador deberá tomar un valor booleano que determine si el cambio de posición se ha realizado correctamente.

4.5.17.7. Obtener la posición actual de puntero a fichero

Número: 0152

Nombre: Obtener la posición actual de puntero a fichero.

Categoría: Operadores sobre ficheros.

Descripción: Se necesita de un operador que dado un puntero a fichero tome el valor aritmético que represente la posición de este dentro del mismo.

4.5.17.8. Obtener contenido de un fichero

Número: 0153

Nombre: Obtener contenido de un fichero.

Categoría: Operadores sobre ficheros.

Descripción: Se precisa de un operador que simplifique la tarea de obtener el contenido completo de un fichero, sin que sea necesario disponer de un puntero al mismo. Para ello se deberá facilitar una cadena de caracteres que simbolice la ruta completa del fichero. El operador tomará como valor una cadena de caracteres que contenga todo el contenido del fichero. En el caso de que el fichero no exista se deberá tomar como valor la cadena vacía.

4.5.17.9. Cadena como contenido de un fichero.

Número: 0154

Nombre: Cadena como contenido de un fichero.

Categoría: Operadores sobre ficheros.

Descripción: Se precisa de un operador que simplifique la tarea de escribir una cadena de caracteres en un fichero, sin que sea necesario disponer de un puntero al mismo. Si el fichero existe su contenido deberá ser truncado, si no existe será creado. Este operador tendrá como operandos dos cadenas de caracteres que se correspondan con la ruta del fichero y la cadena a escribir. Como valor se tomará la cadena escrita.

4.5.17.10. Añadir cadena al contenido de un fichero

Número: 0155

Nombre: Añadir cadena al contenido de un fichero.

Categoría: Operadores sobre ficheros.

Descripción: Se precisa de un operador que simplifique la tarea de añadir una cadena de caracteres al final de un fichero, sin que sea necesario disponer de un puntero al mismo. Si el fichero no existe será creado. Este operador tendrá como operandos dos cadenas de caracteres que se correspondan con la ruta del fichero y la cadena a escribir. Como valor se tomará la cadena escrita.

4.5.18. Fechas y tiempo

4.5.18.1. Obtener fecha

Número: 0156

Nombre: Obtener fecha.

Categoría: Fechas y tiempo.

Descripción: Es necesario disponer de un operador que dé formato a la fecha/hora local. Este operador deberá tener como operando una expresión cadena de caracteres que contenga una serie de directivas de formato. El valor que tomará la operación será una cadena de caracteres que represente la fecha/hora en el formato dado. Las directivas de formato serán las siguientes:

%d: Día del mes con dos dígitos.

%j: Día del mes sin ceros iniciales.

%l: Día de la semana de forma alfabética completa.

%D: Día de la semana de forma alfabética y con tres letras.

%w: Día de la semana de forma numérica (0-domingo,6-sábado).

%z: Día del año de forma numérica.

%F: Mes de forma alfabética.

%m: Mes de forma numérica con dos dígitos.

%n: Mes de forma numérica sin ceros iniciales.

%M: Mes de forma alfabética con tres letras.

%Y: Año con cuatro dígitos.

%y: Año con dos dígitos.

%a: Periodo del día (am/pm) en minúsculas.

%A: Periodo del día (am/pm) en mayúsculas.

%g: Hora en formato 12h sin ceros iniciales.

%G: Hora en formato 24h sin ceros iniciales.

%h: Hora en formato 12h con dos dígitos.

%H: Hora en formato 24h con dos dígitos.

%i: Minutos con dos dígitos.

%U: Segundos desde la Época Unix (1 de Enero del 1970 00:00:00 GMT).

%%: Carácter %.

4.5.18.2. Tiempo Unix

Número: 0157

Nombre: Operador time.

Categoría: Fechas y tiempo.

Descripción: Se precisa de un operador que calcule el número de segundos desde la Época Unix (1 de Enero del 1970 00:00:00 GMT). Este operador no tendrá operandos y tomará el valor aritmético correspondiente.

4.5.18.3. Parar ejecución

Número: 0158

Nombre: Parar ejecución.

Categoría: Fechas y tiempo.

Descripción: Se deberá de proporcionar un mecanismo que permita suspender o bloquear la ejecución durante un tiempo dado. Constará de una expresión que represente el valor aritmético del tiempo en segundos.

4.5.19. Errores

Número: 0159

Nombre: Información de errores.

Categoría: Error.

Descripción: Si se produce un error el intérprete debe informar del tipo y causa del error, así como del contexto en el que se ha producido (nombre y línea de fichero).

4.5.20. Extensiones

4.5.20.1. Extensión

Número: 0160

Nombre: Extensión.

Categoría: Extensiones.

Descripción: La funcionalidad y características del intérprete deben ser extensible mediante módulos dinámicos. Estos módulos añadirán sentencias, operadores y demás elementos propios de un lenguaje de programación.

Para que un extensión pueda ser utilizada se deberá cargar.

4.5.20.2. Carga de extensiones mediante fichero de configuración

Número: 0161

Nombre: Carga de extensiones mediante fichero de configuración.

Categoría: Extensiones.

Descripción: Se debe facilitar un mecanismo que permita especificar un listado de extensiones que serán cargados al ejecutarse el intérprete. Estos módulos serán especificados en un fichero de texto plano separados mediante saltos de línea. Toda ejecución del intérprete conllevará la carga de las extensiones especificadas.

4.5.20.3. Carga de extensiones mediante sentencia

Número: 0162

Nombre: Carga de extensiones mediante sentencia.

Categoría: Extensiones.

Descripción: Se debe facilitar un mecanismo que permita cargar una extensión en tiempo de ejecución. Para ello se deberá facilitar la ruta del módulo correspondiente a la extensión como una cadena de caracteres. Tras la carga de la extensión las características de esta serán añadidas al intérprete.

4.5.20.4. Biblioteca GNU de internacionalización (gettext)

Número: 0163

Nombre: Extensión gettext.

Categoría: Extensión gettext.

Descripción: Se deberá facilitar a modo de extensión la funcionalidad y características de la biblioteca GNU de internacionalización (i18n), gettext.

4.5.21. Biblioteca de desarrollo OMI

4.5.21.1. Recursos del intérprete

Número: 0164

Nombre: Recursos del intérprete.

Categoría: Biblioteca OMI.

Descripción: Todos los recursos de desarrollo sobre los que se construye el intérprete tales como clases, funciones, etc. Deberán ser construidos para que puedan ser utilizados desde otros desarrollos, estableciéndose así un marco de trabajo para otros proyectos.

4.5.21.2. Interpretar desde otros proyectos

Número: 0165

Nombre: Interpretar desde otros proyectos.

Categoría: Biblioteca OMI.

Descripción: Se debe facilitar un mecanismo para que otros proyectos software puedan interpretar código OMI de forma interna. Esto se hará mediante la llamada a una función de la biblioteca OMI.

4.5.22. Web del proyecto OMI

4.5.22.1. Sitio web

Número: 0166

Nombre: Sitio web.

Categoría: Web OMI.

Descripción: Se ha de disponer de una web que sirva de plataforma para acceder al proyecto y todos los recursos que este brinda. Esta web tendrá una estructura estática.

4.5.22.2. Home del sitio web

Número: 0167

Nombre: Home del sitio web.

Categoría: Web OMI.

Descripción: La web OMI debe dispone de una página de entrada en la que se recoge:

- Una descripción resumida del proyecto.
- Enlaces a las secciones principales.
- Un listado de noticias que será mantenida manualmente por el administrador
- Enlaces a las descargas de la última versión de los recursos.

4.5.22.3. Sobre OMI

Número: 0168

Nombre: Sobre OMI.

Categoría: Web OMI.

Descripción: Deberá recoger información sobre el proyecto, el alcance del mismo, la autoría y los organismos implicados.

4.5.22.4. Contacto

Número: 0169

Nombre: Contacto.

Categoría: Web OMI.

Descripción: deberá listar diferentes medios de contacto.

4.5.22.5. Índice de documentación

Número: 0170

Nombre: Índice de documentación.

Categoría: Web OMI.

Descripción: Se ha de dispone de una página en la que se enlace de forma ordenada las distintas secciones de la documentación, así como las distintas herramientas para navegar por esta.

4.5.22.6. Página de documentación

Número: 0171

Nombre: Índice de documentación.

Categoría: Web OMI.

Descripción: Cada documento relativo a la presente memoria deberá estar disponible desde la web del proyecto.

4.5.22.7. Características

Número: 0172

Nombre: Características.

Categoría: Web OMI.

Descripción: Se ha de disponer de una página en la que se liste las características del intérprete.

4.5.22.8. Navegador de gramática

Número: 0173

Nombre: Gramática.

Categoría: Web OMI.

Descripción: Se ha de disponer de una página que sirva para navegar por las reglas gramaticales que definen el intérprete.

4.5.22.9. Navegador de clases

Número: 0174

Nombre: Clases.

Categoría: Web OMI.

Descripción: Se ha de disponer de una página que sirva para navegar por las clases de programación sobre las que se construye el intérprete.

4.5.22.10. Navegador de archivos

Número: 0175

Nombre: Archivos .

Categoría: Web OMI.

Descripción: Se ha de disponer de una página que sirva para navegar por los archivos de código fuente que sirven para construir el intérprete.

4.5.22.11. Wiki

Número: 0176

Nombre: Wiki .

Categoría: Web OMI.

Descripción: Se ha de disponer de una wiki para el proyecto que recoja aspectos relativos al proyecto y las materias que este estudia.

4.5.23. RunTree

4.5.23.1. Intérprete online

Número: 0177

Nombre: Intérprete online.

Categoría: RunTree.

Descripción: Se ha de disponer de una herramienta que permita hacer uso del intérprete desde el navegador. Esta herramienta tomará un código fuente escrito en el lenguaje de programación y lo enviará a un intérprete OMI para su procesamiento. Luego mostrará, paso a paso, información relativa al proceso de interpretación.

4.5.23.2. Escribir código fuente

Número: 0178

Nombre: Escribir código fuente.

Categoría: RunTree.

Descripción: La herramienta deberá de disponer de un campo de formulario en el que se pueda escribir el código fuente que será enviado para interpretar. Este campo deberá mostrar un resaltado de sintaxis.

4.5.23.3. Árbol sintáctico

Número: 0179

Nombre: Árbol sintáctico.

Categoría: RunTree.

Descripción: La herramienta deberá de mostrar el árbol sintáctico resultado de la interpretación y permitir la navegabilidad por este.

4.5.23.4. Tabla de símbolos

Número: 0180

Nombre: Tabla de símbolos.

Categoría: RunTree.

Descripción: La herramienta deberá de mostrar el estado de las tablas de símbolos en cada paso del proceso.

4.5.23.5. Salida de datos

Número: 0181

Nombre: Salida.

Categoría: RunTree.

Descripción: La herramienta deberá mostrar la salida generada por cada paso del proceso.

4.5.23.6. Entrada de datos

Número: 0182

Nombre: Entrada de datos.

Categoría: RunTree.

Descripción: La herramienta deberá solicitar al usuario la entrada de datos cuando el proceso de interpretación lo requiera.

4.5.23.7. Limpieza de salida

Número: 0183

Nombre: Limpieza de salida.

Categoría: RunTree.

Descripción: La herramienta deberá dar la posibilidad de limpiar la salida generada por el proceso de interpretación.

4.5.23.8. Consola de información

Número: 0184

Nombre: Consola de información.

Categoría: RunTree.

Descripción: La herramienta deberá disponer de una sección en la que se muestre textualmente la operación llevada a cabo.

4.5.23.9. Ejecución paso a paso

Número: 0185

Nombre: Paso a paso.

Categoría: RunTree.

Descripción: La herramienta deberá de disponer de acciones que permitan avanzar la ejecución un paso.

4.5.23.10. Ejecución sentencia a sentencia

Número: 0186

Nombre: Sentencia a sentencia.

Categoría: RunTree.

Descripción: La herramienta deberá de disponer de acciones que permitan avanzar la ejecución una sentencia.

4.5.23.11. Ejecución automática

Número: 0187

Nombre: Ejecución automática.

Categoría: RunTree.

Descripción: La herramienta deberá de disponer de la capacidad de ejecutar paso a paso el programa pero de una forma automática.

4.5.23.12. Guardar código fuente

Número: 0188

Nombre: Guardar código fuente.

Categoría: RunTree.

Descripción: La herramienta deberá de disponer de la capacidad de guardar el código fuente escrito en un fichero local.

4.5.23.13. Abrir código fuente

Número: 0189

Nombre: Abrir código fuente.

Categoría: RunTree.

Descripción: La herramienta deberá de disponer de la capacidad de leer el código fuente escrito en un fichero local para enviarlo a interpretar.

4.6. Requisitos no funcionales

4.6.1. Rendimiento

Es condición necesaria que el sistema software desarrollado presente un rendimiento óptimo. Para ello se medirá el rendimiento desde dos aspectos: tiempo y espacio

4.6.1.1. Tiempo

A pesar de que el objetivo del intérprete no es servir para la producción de software, al ser un sistema que será utilizado para el desarrollo de otras aplicaciones, el rendimiento en cuanto al tiempo debe ser aceptable en comparación con las herramientas existentes en el mercado.

4.6.1.2. Espacio

El sistema debe ser óptimo en cuanto el espacio en memoria que ocupan sus estructuras. Debe hacer un uso correcto de la memoria. Si las estructuras que conforman el intérprete ocupan demasiado espacio, los datos de los que hagan uso el usuario en sus programas también lo harán.

4.6.2. Usabilidad

4.6.2.1. Reglas léxicas y sintácticas

El sistema deberá interpretar un lenguaje con reglas léxicas y sintácticas claras, que sean fáciles de comprender y asimilar. La estructura del lenguaje en cuanto a estos dos aspectos debe ser similar a los lenguajes de programación actuales.

4.6.2.2. Interfaz

La interfaz de entrada salida del interprete debe ser clara y presentar las opciones de una forma adecuada. Además la interfaz web debe ser amigable a la forma de operación del usuario que la utilice.

4.6.3. Accesibilidad

El sistema desarrollado debe ser accesible desde cualquier computadora con acceso a internet. Para ello se utilizará un navegador web. Por otro lado el interprete podrá ser instalado de forma local para un uso individual.

4.6.4. Estabilidad

Se requiere que el sistema desarrollado presente un umbral de fallos bajo. Debe adaptarse a los casos excepcionales, y en caso de error informar adecuadamente de los motivos y causas del mismo.

4.6.5. Mantenibilidad

El sistema desarrollado debe ser mantenible en el tiempo. Para ello debe estar correctamente documentado, modularizado y estructurado.

4.6.6. Concurrencia

El sistema deberá ser accesible por varios usuarios en el mismo marco de tiempo. Es por ello que deberá ser capaz de funcionar en un entorno de concurrencia.

Capítulo 5

Análisis del Sistema

Esta sección cubre el análisis del sistema de información a desarrollar, haciendo uso del lenguaje de modelado UML.

5.1. Modelo conceptual

El presente capítulo representa un análisis de los datos que construyen el sistema OMI y cómo estos se relacionan. Se describe el modelo conceptual de datos del sistema mediante diagramas de clases. Las clases son organizadas en paquetes para facilitar la modularidad del sistema y su entendimiento.

El proceso de interpretar consiste en tomar código fuente, procesarlo y ejecutar su significado semántico. Por tanto el modelo de datos estará constituido por entidades que guardan un significado concreto y preciso dentro del lenguaje. Estos elementos, que representan la unidad semántica mínima, son denominados nodos ejecutables, debido a que cuando son ejecutados producen el resultado semántico asociado. Muchos nodos ejecutables por si solos no presentan un resultado semántico completo, por lo que precisan de otros nodos.

El diagrama general de paquetes describe los paquetes que componen el sistema según el carácter funcional de las entidades que contienen. Un paquete podrá contener clases u otros paquetes.

El paquete “interpreter” describe las entidades que procesan el contenido fuente según el léxico y la gramática del lenguaje OMI. El objetivo es generar el árbol de nodos ejecutables correspondiente al programa. Al procesarse este árbol se aplicará la semántica que encierran las líneas de código del contenido fuente, produciéndose de esta forma la ejecución del programa.

El paquete “runNode” describe el nodo ejecutable y aquellos tipos de nodos derivados de este, que son abstractos y que serán extendidos por tipos más específicos.

El paquete “typeData” describe los nodos correspondientes a los tipos de datos básicos que pueden ser manipulados por el sistema.

El paquete “error” describe el sistema de errores y los nodos ejecutables que permiten su control.

El paquete “extensions” describe el sistema de extensiones del interprete, el cual permite extender la funcionalidades del lenguaje de una forma dinámica. Además contiene dos el modelado de dos extensiones concretas.

Los paquetes siguientes categorizan y agrupan nodos ejecutables según la funcionalidad que encierran y el tipo de dato sobre el que operan.

El último paquete “rtree” describe el modelo de datos correspondiente al sistema software cliente. Una aplicación web que hace uso del interprete de forma online y representa el estado de este.

5.1.1. Intérprete

El sistema OMI se corresponde con un interprete que opera sobre un contenido fuente escrito en el lenguaje con el mismo nombre. El interprete se compone de un analizador sintáctico que encierra la gramática del lenguaje, esta es descrita a partir de una serie de tokens.

El analizador sintáctico se vale de un analizador léxico que validará y obtendrá los tokens (bajo petición) desde el código fuente. El analizador léxico debe controlar el fichero que contiene el código fuente, así como la línea y posición que se encuentra procesando en el mismo.

Los tokens obtenidos se definen por un identificador y la línea del código fuente en la que se generó, además pueden tener asociado un valor que puede ser numérico o cadena. Serán utilizados por el analizador sintáctico para determinar las reglas gramaticales que se deben aplicar y construir el árbol sintáctico correspondiente.

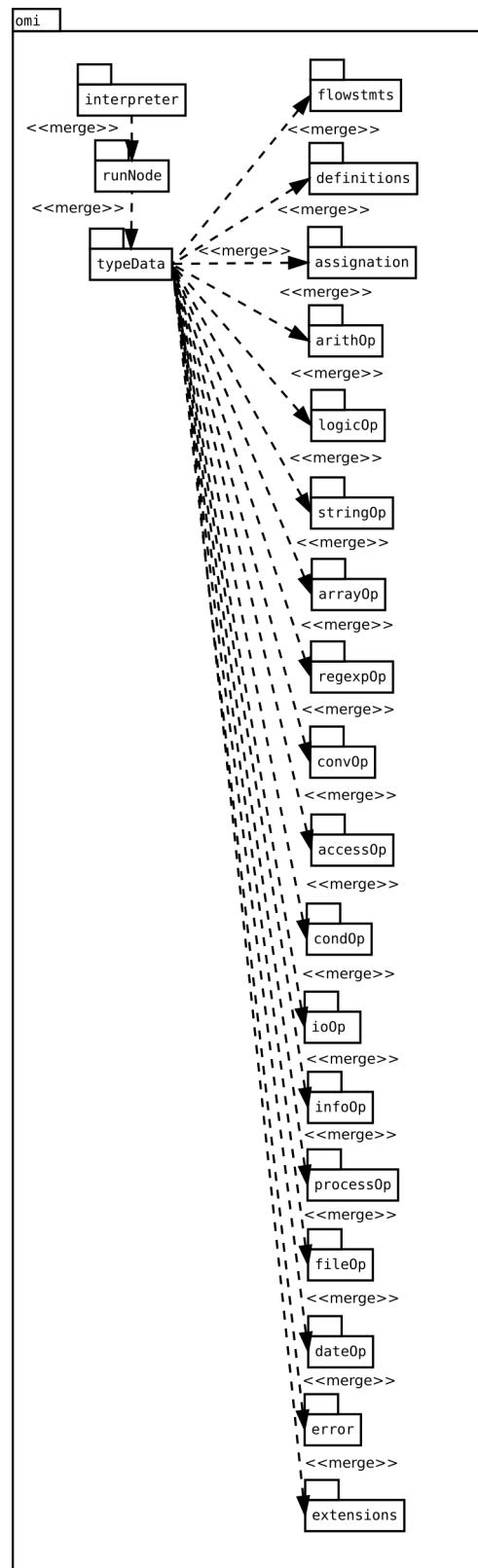


Figura 5.1: Paquetes OMI

Este árbol está formado por nodos denominados ejecutables, dado que al ser procesados en profundidad se llevará a cabo la ejecución del programa. Los nodos ejecutables dan significado semántico a cada una de las sentencias que componen el contenido fuente.

El interprete se compone además de un contexto denominado principal, que será sobre el que opere de forma predeterminada. Un contexto está formado por una serie de tablas de símbolos que serán manipuladas por ciertos nodos ejecutables cuando sean procesados. Estas tablas guardan referencias a nodos ejecutables correspondientes a símbolos variables, funciones y clases de objetos que son definidos en el código fuente. Existen determinados nodos que al ser ejecutados pueden cambiar el contexto en uso.

El interprete es ejecutado con una serie de argumentos que alteran su funcionamiento.

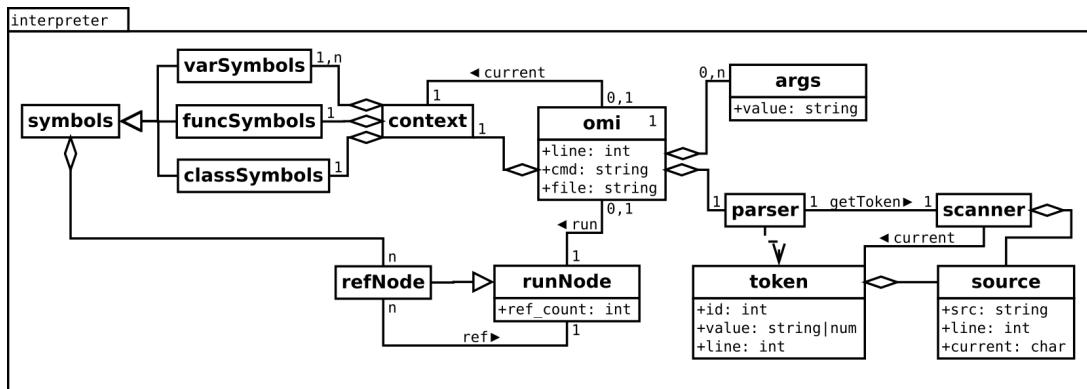


Figura 5.2: Clases intérprete

5.1.2. Nodos ejecutables

Se definen un nodo ejecutable para cada aspecto o funcionalidad que contemple el lenguaje. Cada sentencia se corresponde con un nodo ejecutable, que a su vez puede estar compuestos de otros nodos. Cada nodo ejecutable guarda el número de nodos que lo referencian para que se pueda hacer un uso óptimo del mismo.

Las expresiones son nodos ejecutables que tomarán un valor tras ser procesados. Generalmente forman parte de otros nodos correspondientes a sentencias u otras expresiones. El valor que toman pueden ser de un tipo determinado y conocido (numérico, lógico, etc), o de tipo indeterminado o no conocido hasta que el nodo es procesado.

Las expresiones de tipo determinado son extendidas por cada tipo de dato soportado por el lenguaje. Además pueden ser consideradas tipos de objetos y estar así asociadas a una clase. De esta forma toda expresión puede disponer de métodos y atributos según el tipo de dato que guarde.

Las expresiones de tipo indeterminado se componen de una referencia al nodo que guarda el valor tras la ejecución, este podrá ser una expresión de tipo determinado.

Las expresiones son nodos imprimibles lo que significa que tienen una representación gráfica asociada que puede ser volcada en la salida estándar.

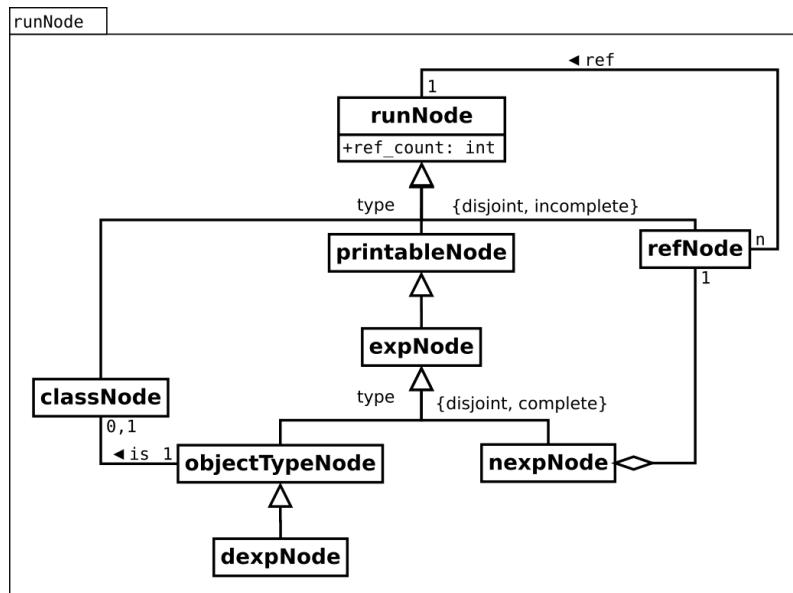


Figura 5.3: Clases nodos ejecutables

5.1.3. Tipos de datos

Este paquete contiene los nodos que representan expresiones con tipos de datos definidos. Se describe cada tipo de dato como un nodo con un valor asociado (en algunos casos el tipo puede comprender un único valor).

Muchos nodos son especializaciones de tipos de datos, correspondiéndose con expresiones que guardan un valor del tipo de dato al que extienden. Así por ejemplo los nodos de operaciones aritméticas generalmente extenderán al nodo del mismo tipo de dato.

Algunos nodos de tipos datos son concretados por nodos que representan un valor constante de dicho tipo de dato.

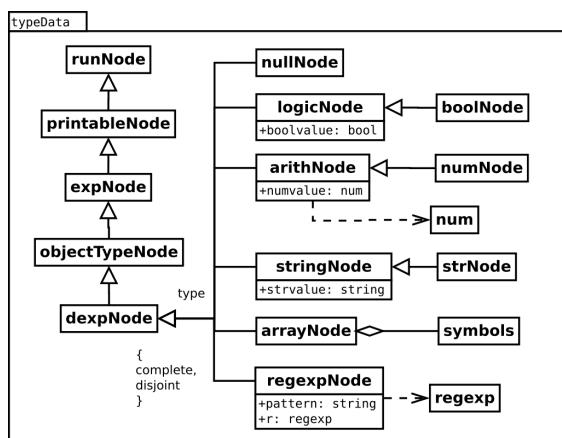


Figura 5.4: Clases tipos de datos

5.1.4. Sentencias de control de flujo

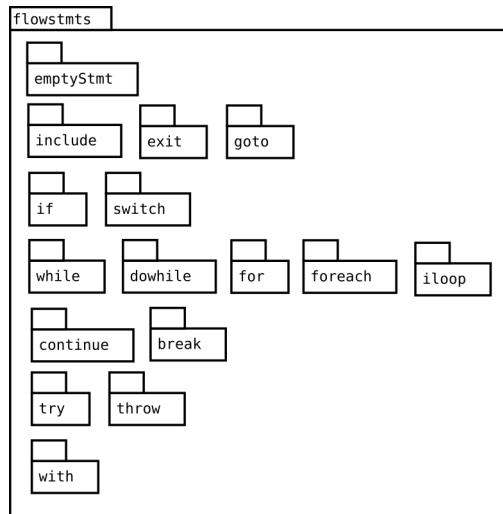


Figura 5.5: Paquetes sentencias de control de flujo

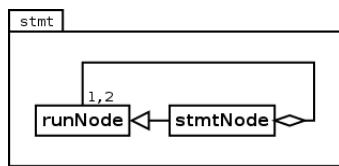


Figura 5.6: Clases sentencia

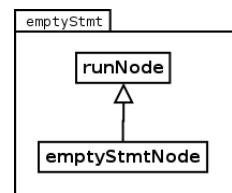


Figura 5.7: Clases sentencia vacía

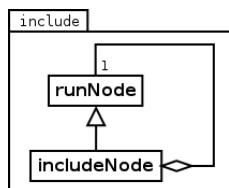


Figura 5.8: Clases include

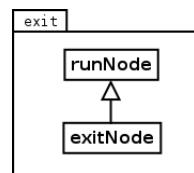


Figura 5.9: Clases exit

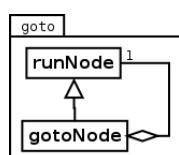


Figura 5.10: Clases goto

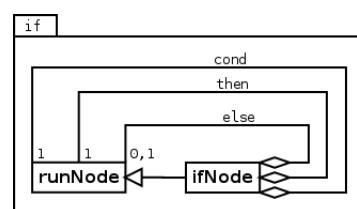


Figura 5.11: Clases if

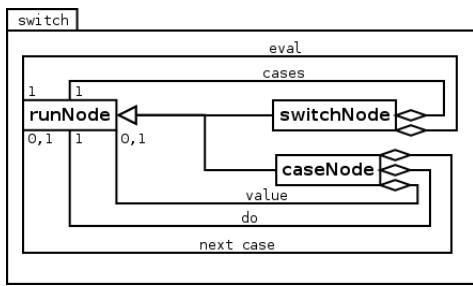


Figura 5.12: Clases switch

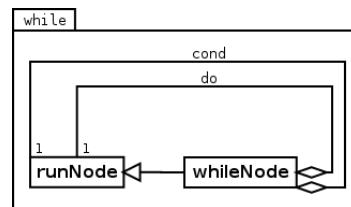


Figura 5.13: Clases while

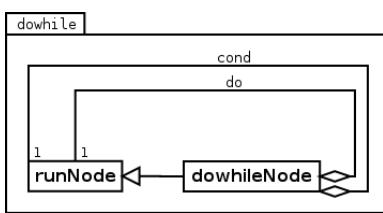


Figura 5.14: Clases do...while

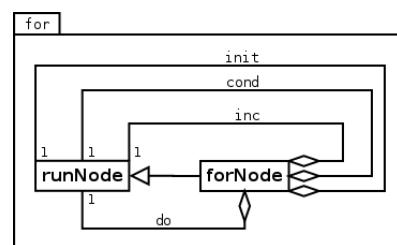


Figura 5.15: Clases for

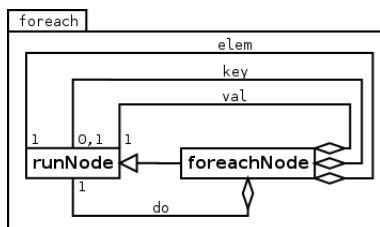


Figura 5.16: Clases foreach

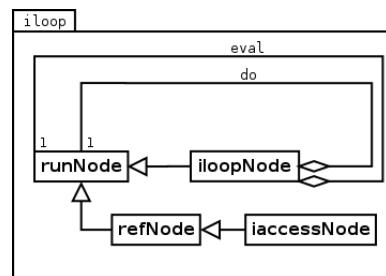


Figura 5.17: Clases iloop

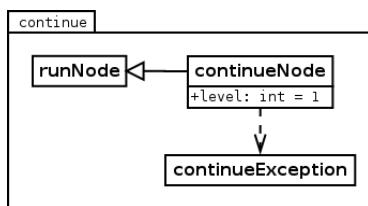


Figura 5.18: Clases continue

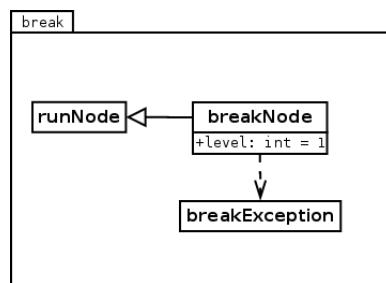


Figura 5.19: Clases break

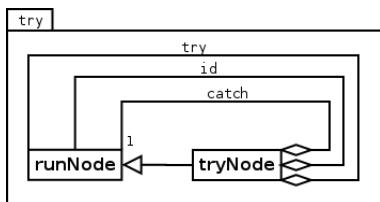


Figura 5.20: Clases try

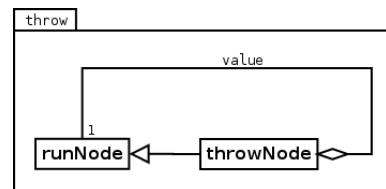


Figura 5.21: Clases throw

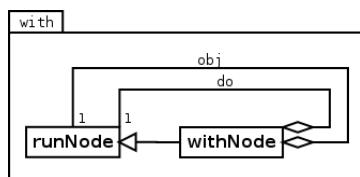


Figura 5.22: Clases witch

5.1.5. Definiciones

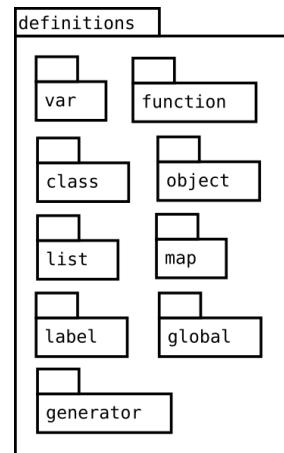


Figura 5.23: Paquetes definiciones

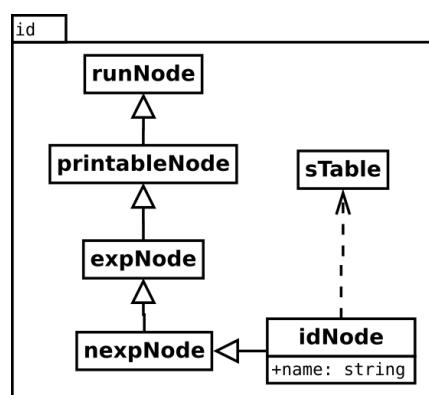


Figura 5.24: Clases sobre variables

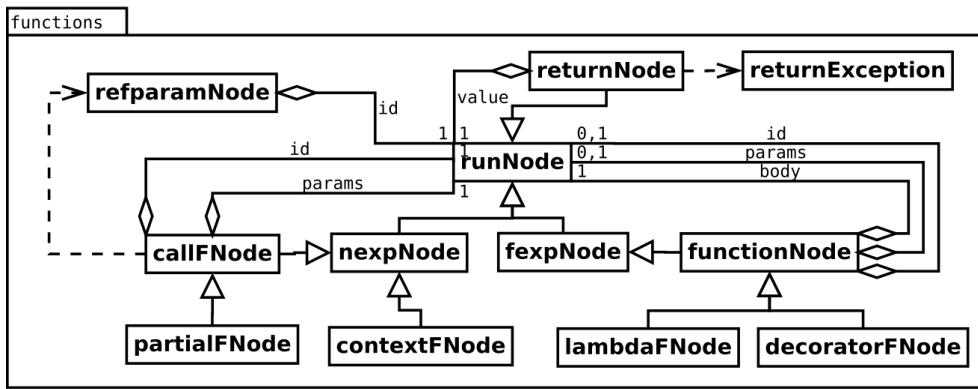


Figura 5.25: Clases sobre funciones

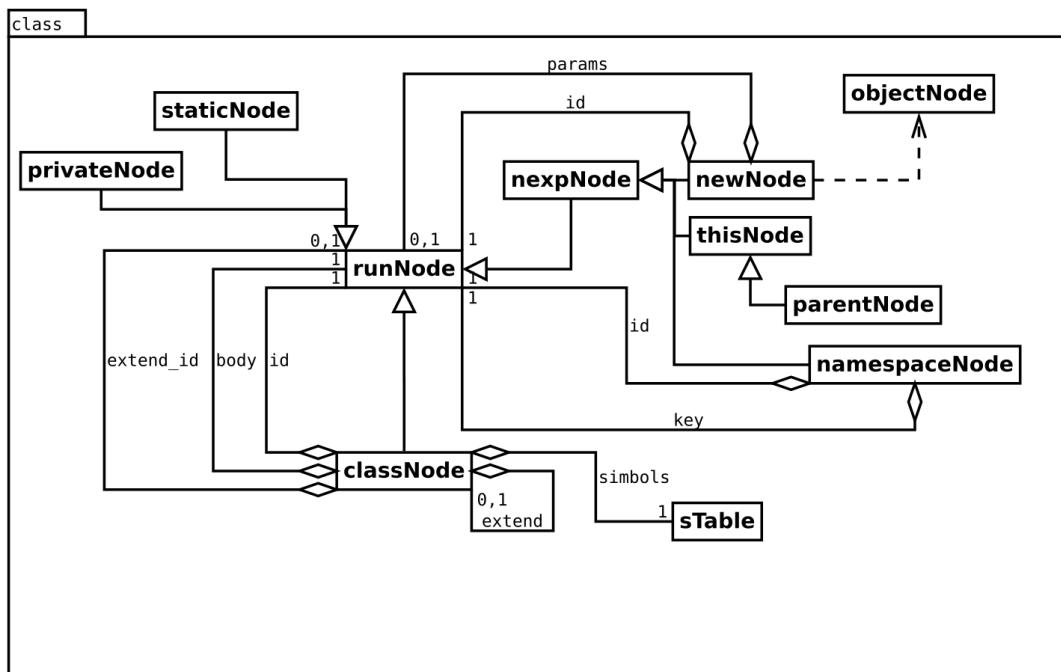


Figura 5.26: Clases sobre clases

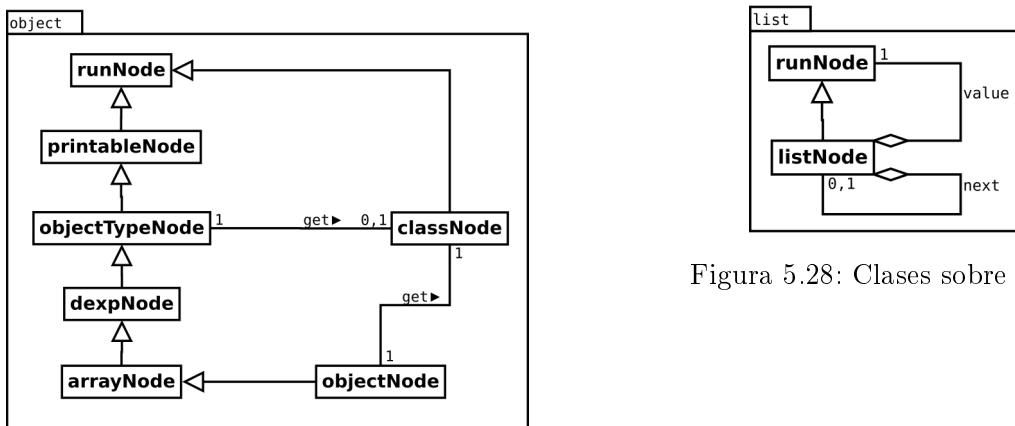


Figura 5.27: Clases sobre objetos

Figura 5.28: Clases sobre listas

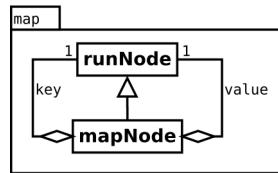


Figura 5.29: Clases sobre pares clave/valor

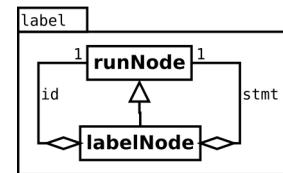


Figura 5.30: Etiquetas

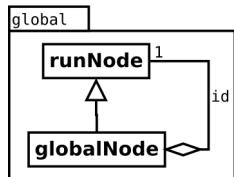


Figura 5.31: Definiciones globales

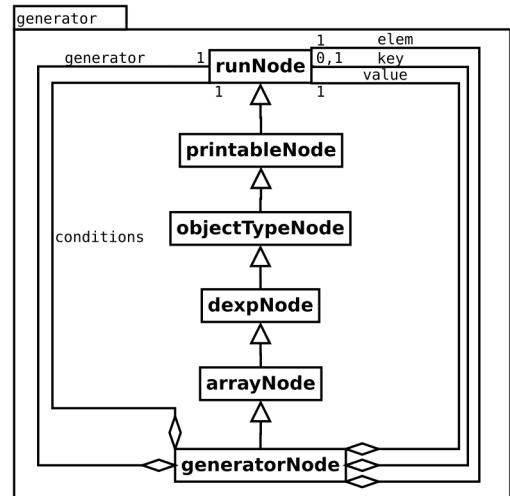


Figura 5.32: Clases sobre generadores

5.1.6. Asignaciones

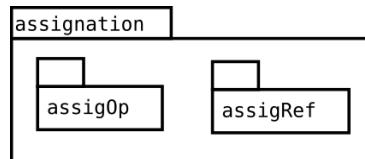


Figura 5.33: Paquetes asignaciones

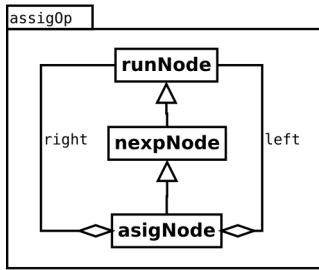


Figura 5.34: Clases asignación

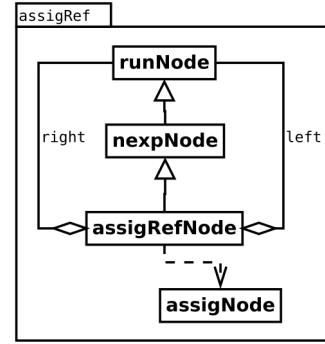


Figura 5.35: Clases asignación de referencia

5.1.7. Operadores aritméticos

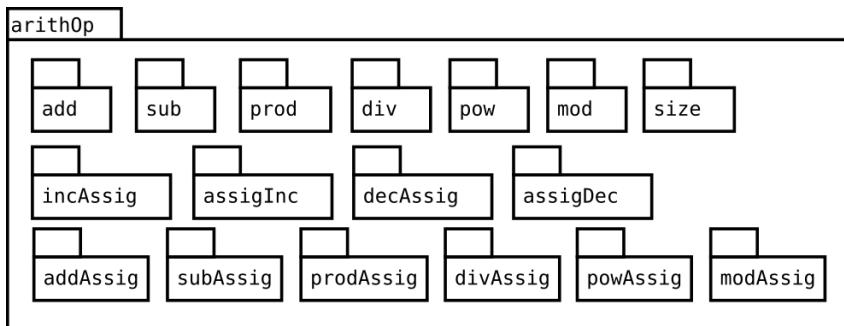


Figura 5.36: Paquetes operadores aritméticos

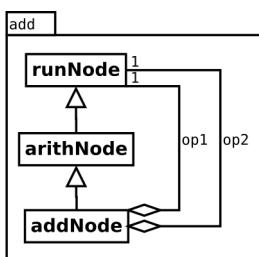


Figura 5.37: Clases suma

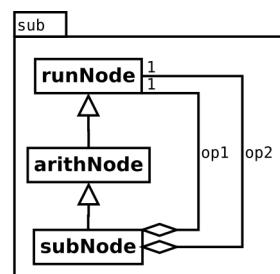


Figura 5.38: Clases diferencia

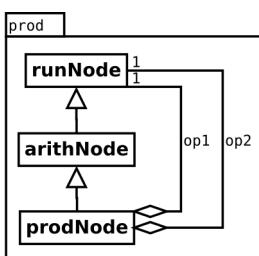


Figura 5.39: Clases producto

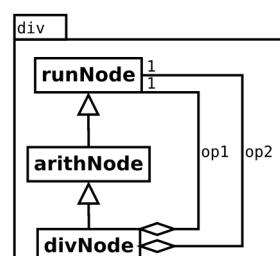


Figura 5.40: Clases división

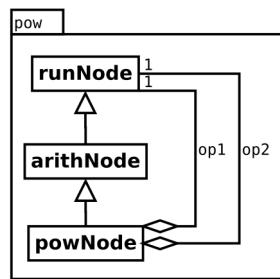


Figura 5.41: Clases potencia

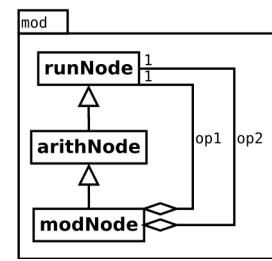


Figura 5.42: Clases módulo

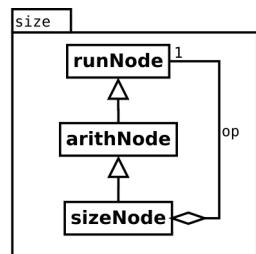


Figura 5.43: Clases tamaño

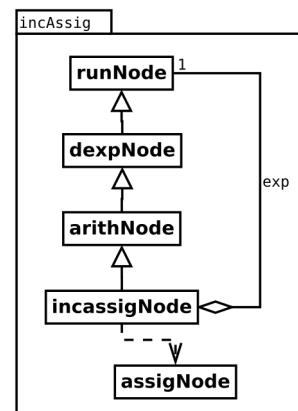


Figura 5.44: Clases incremento y asignación

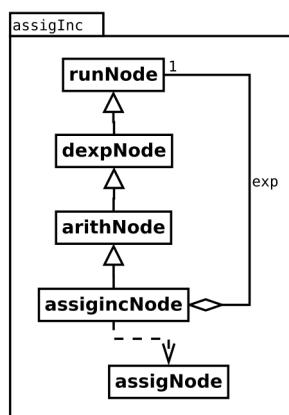


Figura 5.45: Clases asignación e incremento

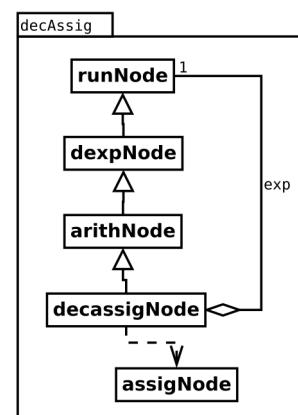


Figura 5.46: Clases decremento y asignación

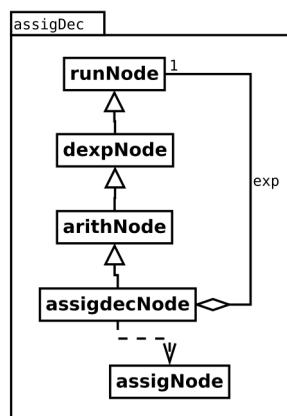


Figura 5.47: Clases asignación y decremento

5.1.8. Operadores lógicos

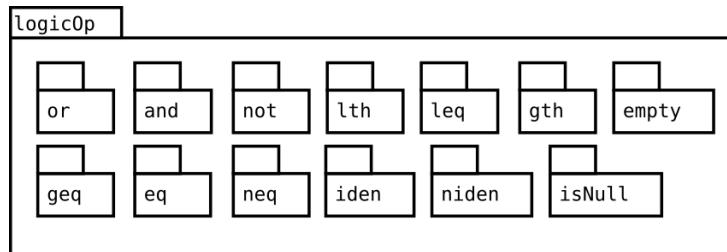


Figura 5.48: Paquetes operadores lógicos

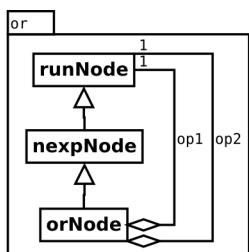


Figura 5.49: Clases or

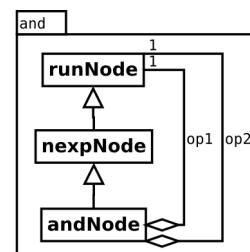


Figura 5.50: Clases and

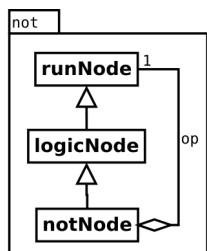


Figura 5.51: Clases negación

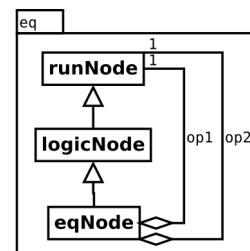


Figura 5.52: Clases igual que

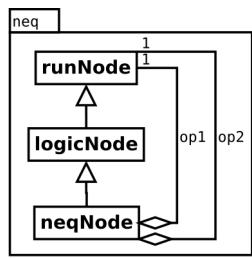


Figura 5.53: Clases distinto que

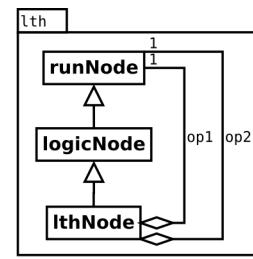


Figura 5.54: Clases menor que

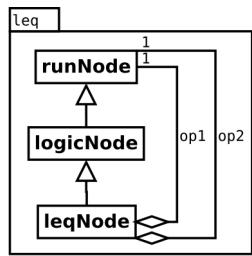


Figura 5.55: Clases menor o igual que

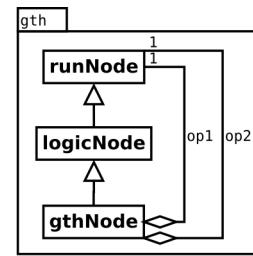


Figura 5.56: Clases mayor que

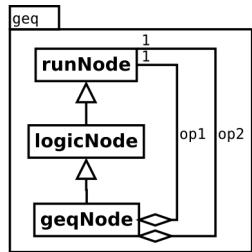


Figura 5.57: Clases mayor o igual que

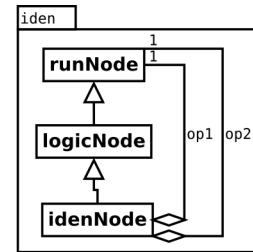


Figura 5.58: Clases idéntico a

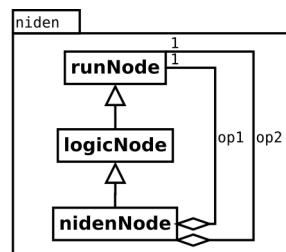


Figura 5.59: Clases no idéntico a

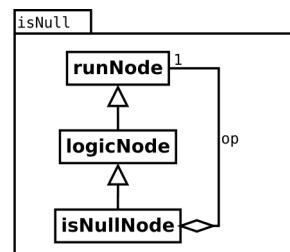


Figura 5.60: Clases es nulo

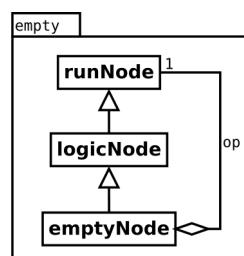


Figura 5.61: Clases vacío

5.1.9. Operadores sobre cadenas

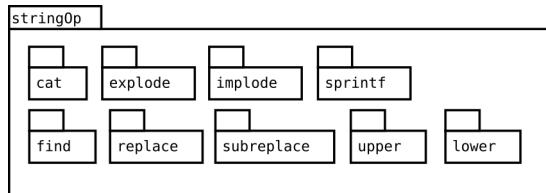


Figura 5.62: Paquetes operadores de cadenas

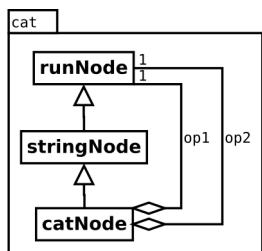


Figura 5.63: Clases concatenación

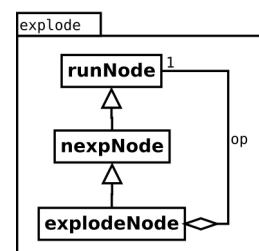


Figura 5.64: Clases explode

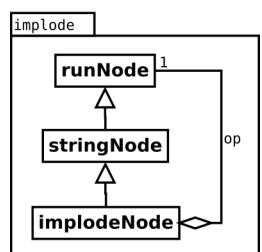


Figura 5.65: Clases implode

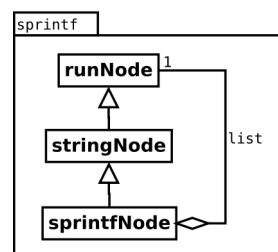


Figura 5.66: Clases sprintf

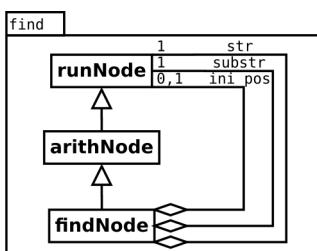


Figura 5.67: Clases buscar subcadena

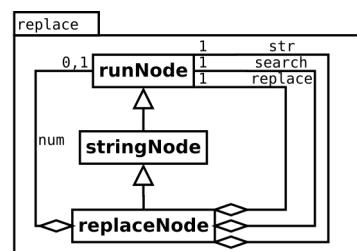


Figura 5.68: Clases buscar y remplazar

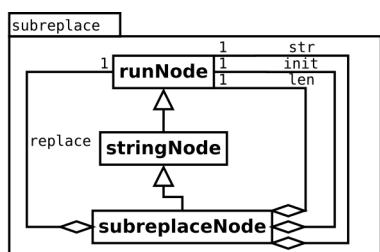


Figura 5.69: Clases remplazar subcadena

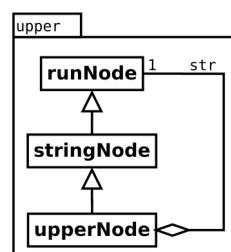


Figura 5.70: Clases convertir a mayúsculas

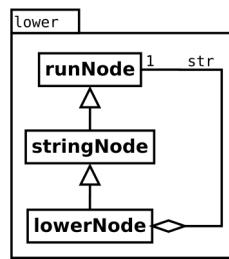


Figura 5.71: Clases convertir a minúsculas

5.1.10. Operadores sobre array

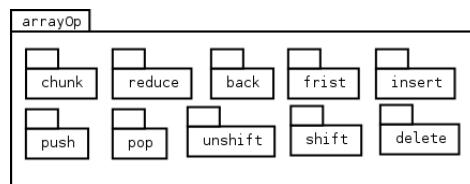


Figura 5.72: Paquetes operadores sobre array

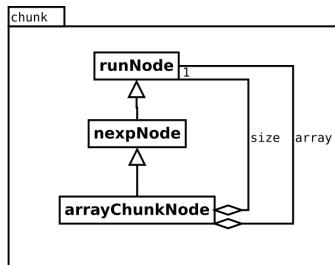


Figura 5.73: Clases dividir en fragmentos

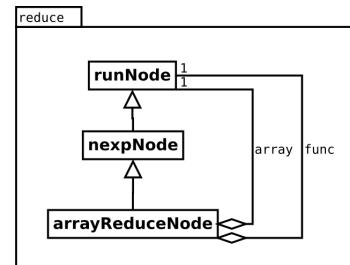


Figura 5.74: Clases reducir mediante función

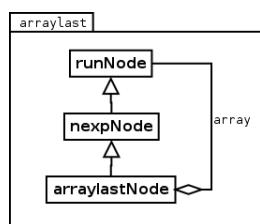


Figura 5.75: Clases obtener último

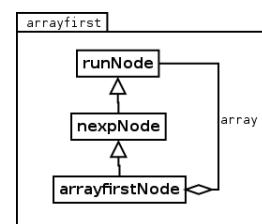


Figura 5.76: Clases obtener primero

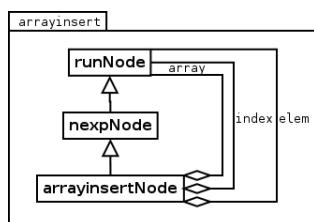


Figura 5.77: Clases insertar en posición

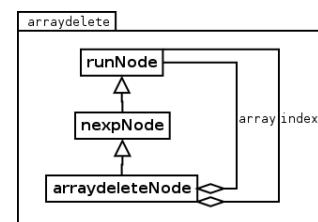


Figura 5.78: Clases eliminar posición

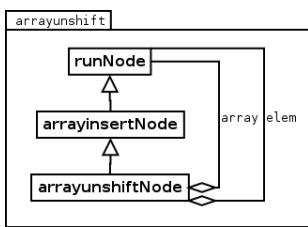


Figura 5.79: Clases insertar al inicio

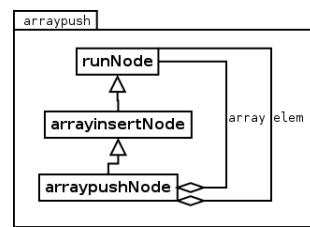


Figura 5.80: Clases insertar al final

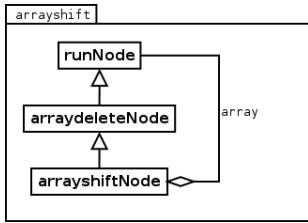


Figura 5.81: Clases eliminar del inicio

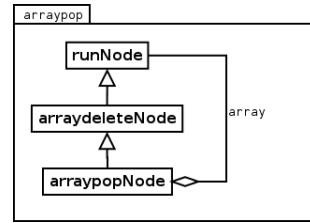


Figura 5.82: Clases eliminar del final

5.1.11. Operadores sobre expresiones regulares

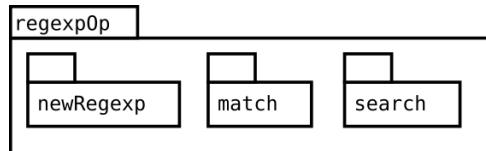


Figura 5.83: Paquetes operadores sobre expresiones regulares

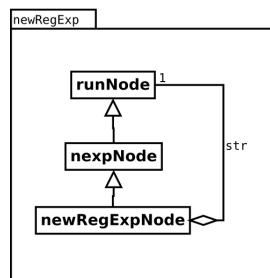


Figura 5.84: Clases crear expresión regular

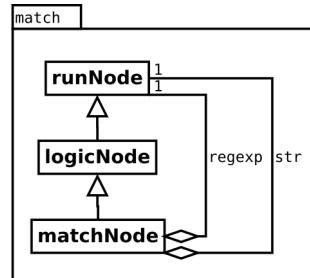


Figura 5.85: Clases match

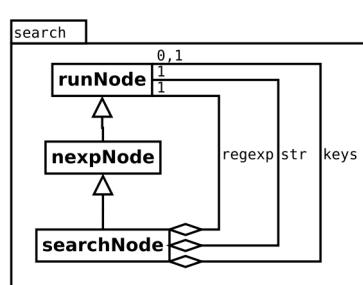


Figura 5.86: Clases search

5.1.12. Conversión de tipos

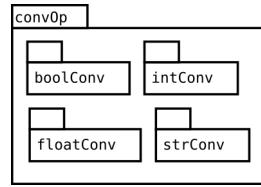


Figura 5.87: Paquetes conversión de tipos

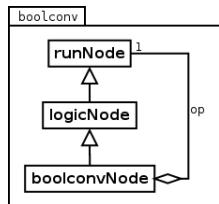


Figura 5.88: Clases conversión a lógico

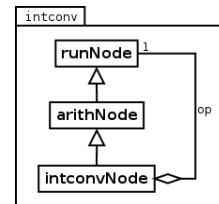


Figura 5.89: Clases conversión a entero

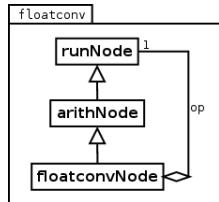


Figura 5.90: Clases conversión a flotante

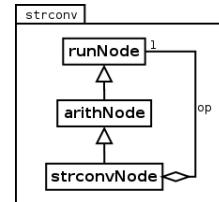


Figura 5.91: Clases conversión a cadena

5.1.13. Operadores de acceso

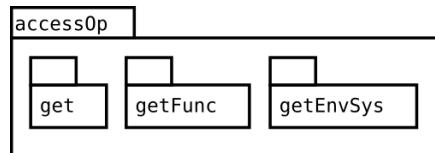


Figura 5.92: Paquetes operadores de acceso

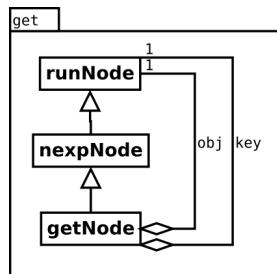


Figura 5.93: Clases acceso a clave

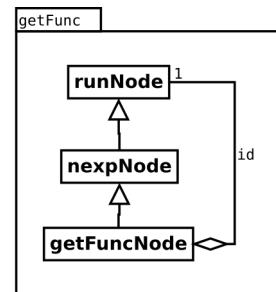


Figura 5.94: Clases acceso a función

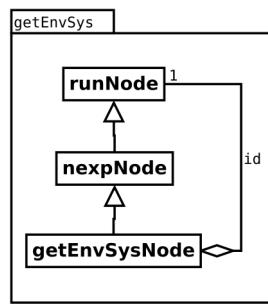


Figura 5.95: Clases acceso a variable de entorno

5.1.14. Operadores condicionales

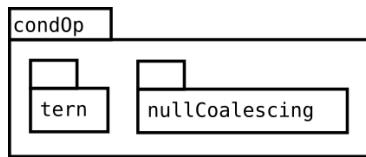


Figura 5.96: Paquetes Operadores condicionales

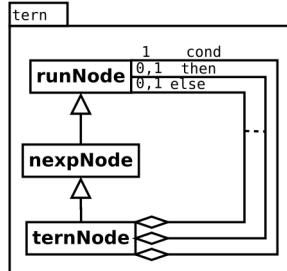


Figura 5.97: Clases ternario

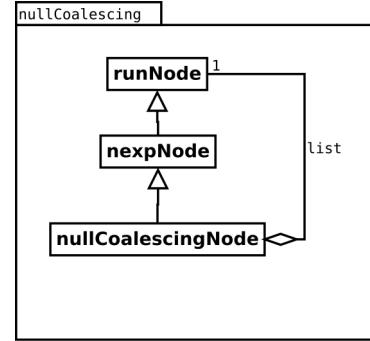


Figura 5.98: Clases fusión de nulos

5.1.15. Operadores de entrada/salida

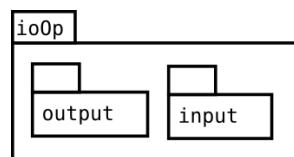


Figura 5.99: Paquetes operadores de entrada/salida

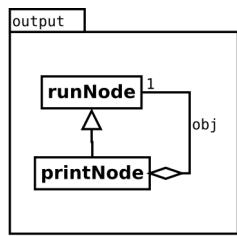


Figura 5.100: Clases salida estándar

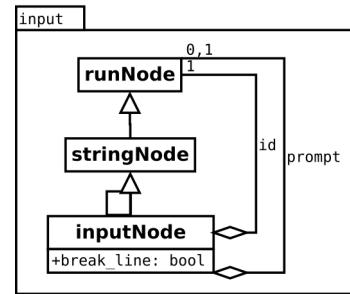


Figura 5.101: Clases entrada estándar

5.1.16. Operadores informativos

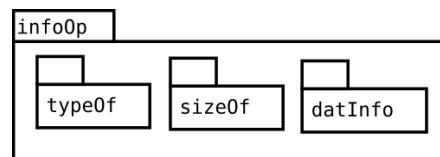


Figura 5.102: Paquetes operadores informativos

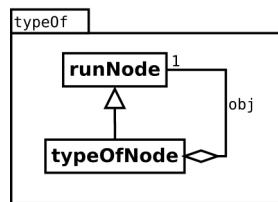


Figura 5.103: Clases tipo de

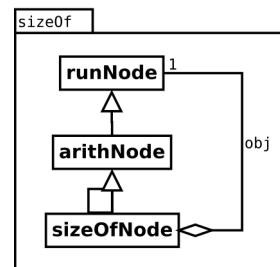


Figura 5.104: Clases tamaño de

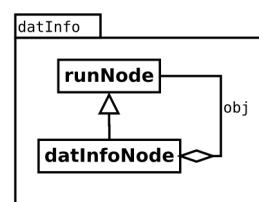


Figura 5.105: Clases información sobre

5.1.17. Procesos

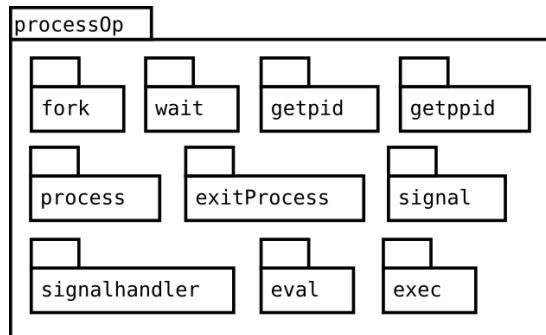


Figura 5.106: Paquete de procesos

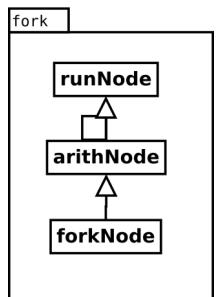


Figura 5.107: Clases crear proceso

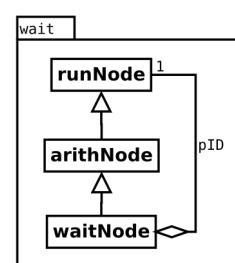


Figura 5.108: Clases esperar finalización

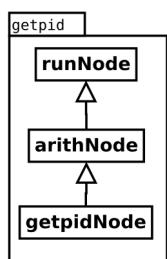


Figura 5.109: Clases obtener identificador

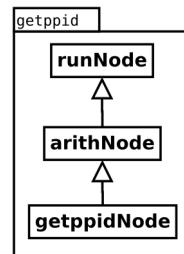


Figura 5.110: Clases obtener identificador padre

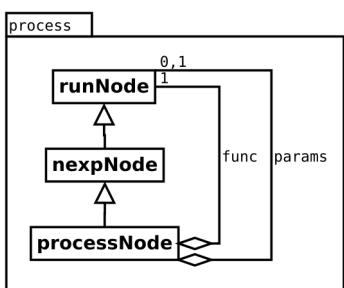


Figura 5.111: Clases ejecutar como proceso

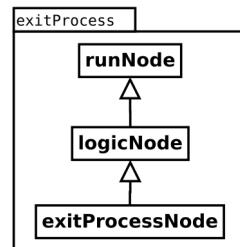


Figura 5.112: Clases salir de proceso

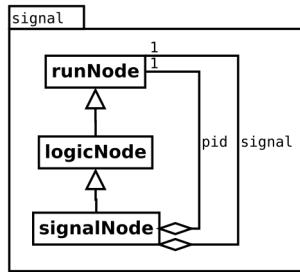


Figura 5.113: Clases señal a proceso

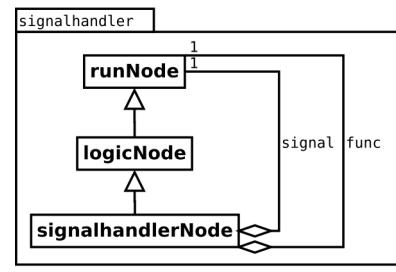


Figura 5.114: Clases manejador de señales

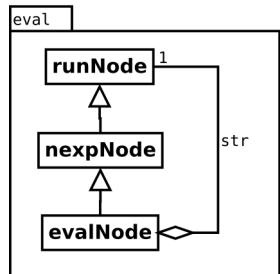


Figura 5.115: Clases evaluar cadena

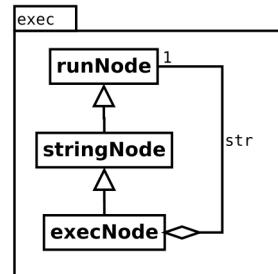


Figura 5.116: Clases ejecutar comando del sistema

5.1.18. Ficheros

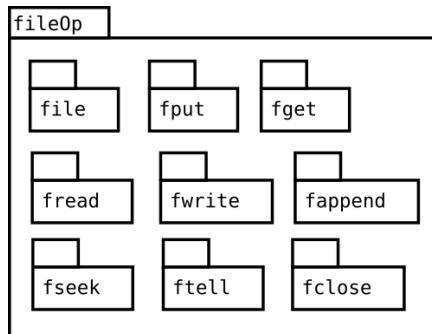


Figura 5.117: Paquetes de ficheros

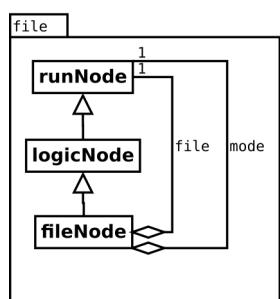


Figura 5.118: Clases obtener un flujo a fichero

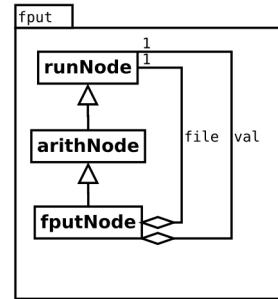


Figura 5.119: escribir en flujo a fichero

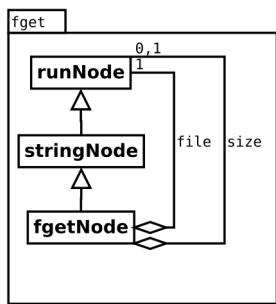


Figura 5.120: Clases leer de flujo a fichero

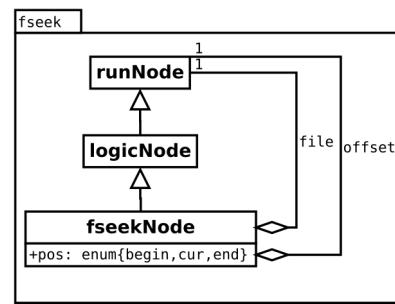


Figura 5.121: Clases cambiar posición en fichero

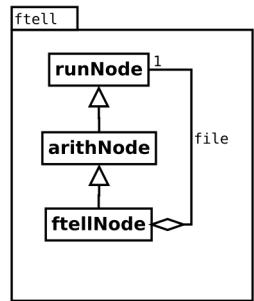


Figura 5.122: Clases obtener posición en flujo a fichero

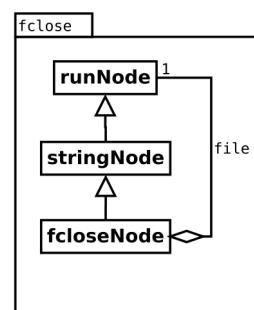


Figura 5.123: Clases cerrar flujo a fichero

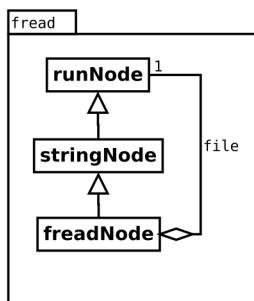


Figura 5.124: Clases leer fichero

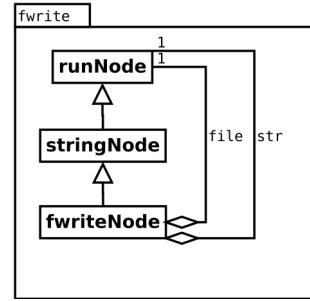


Figura 5.125: Clases escribir en fichero

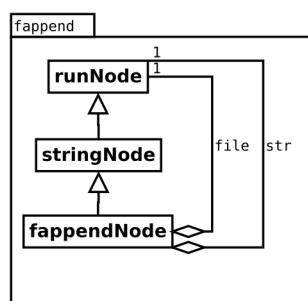


Figura 5.126: Clases escribir al final de fichero

5.1.19. Fechas

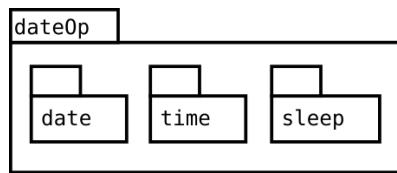


Figura 5.127: Paquetes de fechas

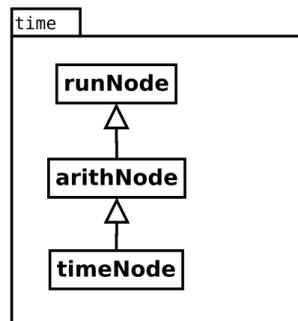


Figura 5.128: Clases tiempo Unix

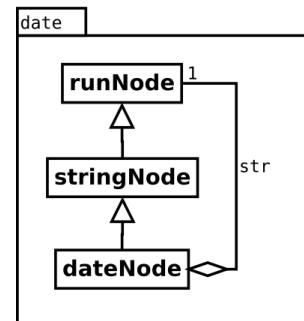


Figura 5.129: Clases fecha y hora con formato

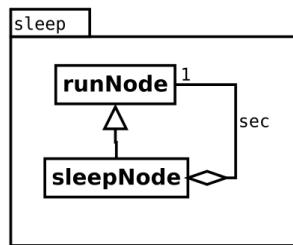


Figura 5.130: Clases sleep

5.1.20. Errores

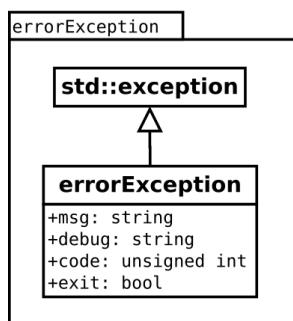


Figura 5.131: Clases Errores

5.1.21. Extensiones

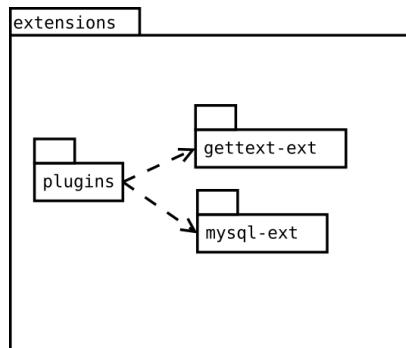


Figura 5.132: Paquete de extensiones

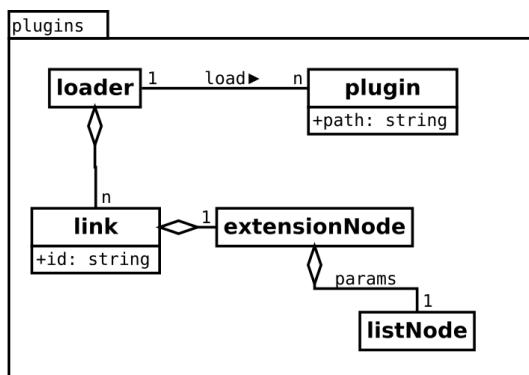


Figura 5.133: Clases plugins

5.1.21.1. Biblioteca GNU de internacionalización (gettext)

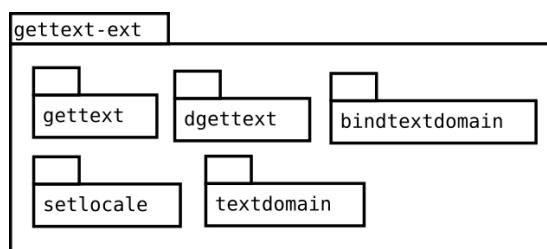


Figura 5.134: Paquetes de Biblioteca GNU de internacionalización (gettext)

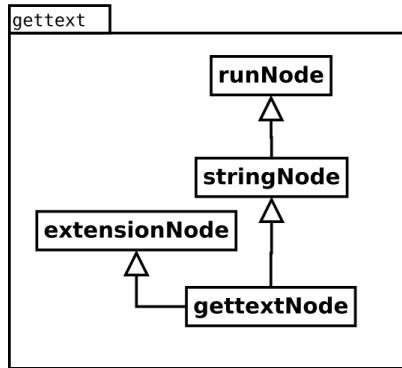


Figura 5.135: Clases gettext

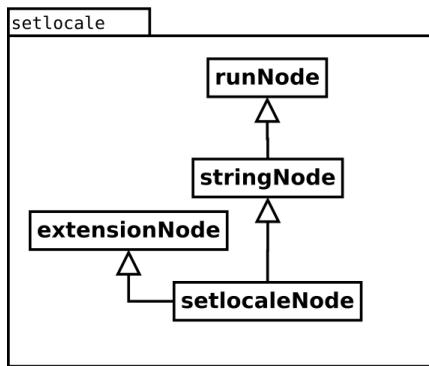


Figura 5.136: Clases setlocale

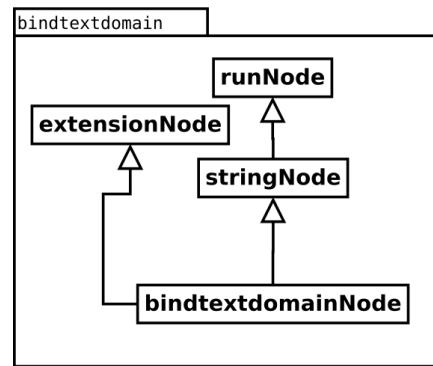


Figura 5.138: Clases bindtextdomain

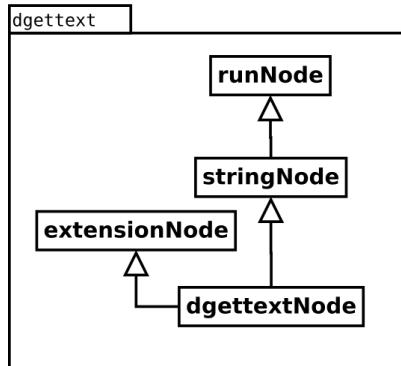


Figura 5.137: Clases dgettext

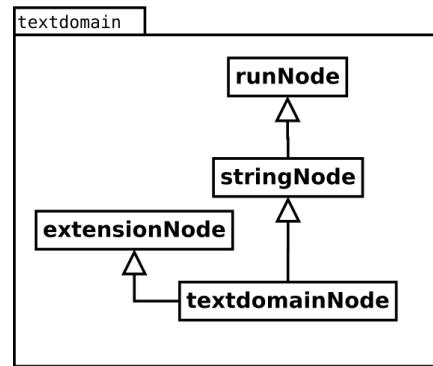


Figura 5.139: Clases textdomain

5.1.22. rTree

El intérprete OMI tiene la capacidad de generar una salida relativa a su estado y funcionamiento. Para completar el proyecto se precisa de una herramienta capaz de interpretar y representar este estado interno de forma gráfica y textual.

El modelo de datos del cliente OMI se define de forma similar al intérprete. La principal diferencia es que en el intérprete este modelo de datos se usa para procesar y ejecutar el código fuente,

mientras que en el cliente se usa para representar gráficamente el proceso llevado a cabo. Es por ello que el modelo de datos del cliente es más abstracto.

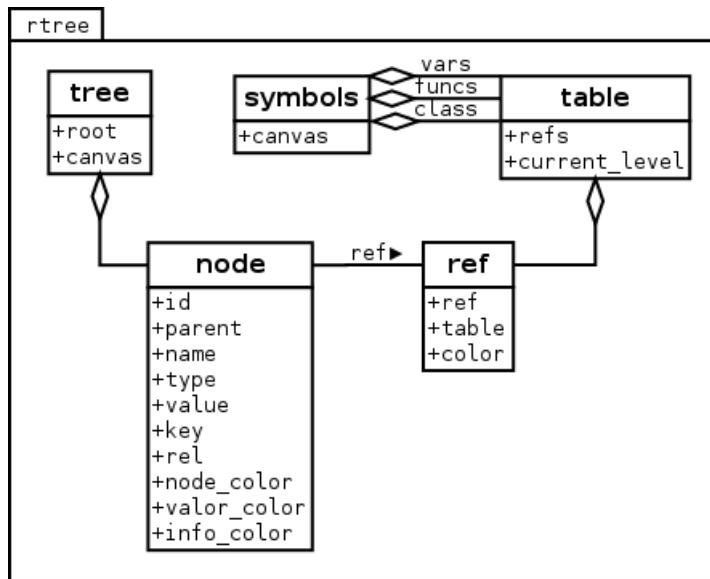


Figura 5.140: Clases rTree

5.2. Modelo de casos de uso

En esta sección se detallan los casos de usos que recogen los sistemas que conforman el proyecto OMI. Un caso de uso describe la secuencia de interacciones entre el sistema y sus actores como consecuencia de un evento.

En primer lugar se describen los actores que hacen uso del sistema, llegándose a distinguir dos que serán nominados usuario y sistema externo. Luego se describen los casos de uso del sistema intérprete y del cliente runTree.

5.2.1. Actores

Los actores humanos que interactúan con los sistemas OMI presentan todos un mismo rol. Así el único actor definido será llamado usuario. Los usuarios que hacen uso del intérprete pueden ser desarrolladores, estudiantes u otros perfiles técnicos, pero todos usarán el sistema de la misma forma.

Por otro lado el intérprete OMI puede ser usado por otros sistemas, viéndose estos actores de determinados casos de uso. Algunos sistemas del proyecto OMI hacen uso del intérprete para llevar a cabo su propósito.

5.2.2. Intérprete

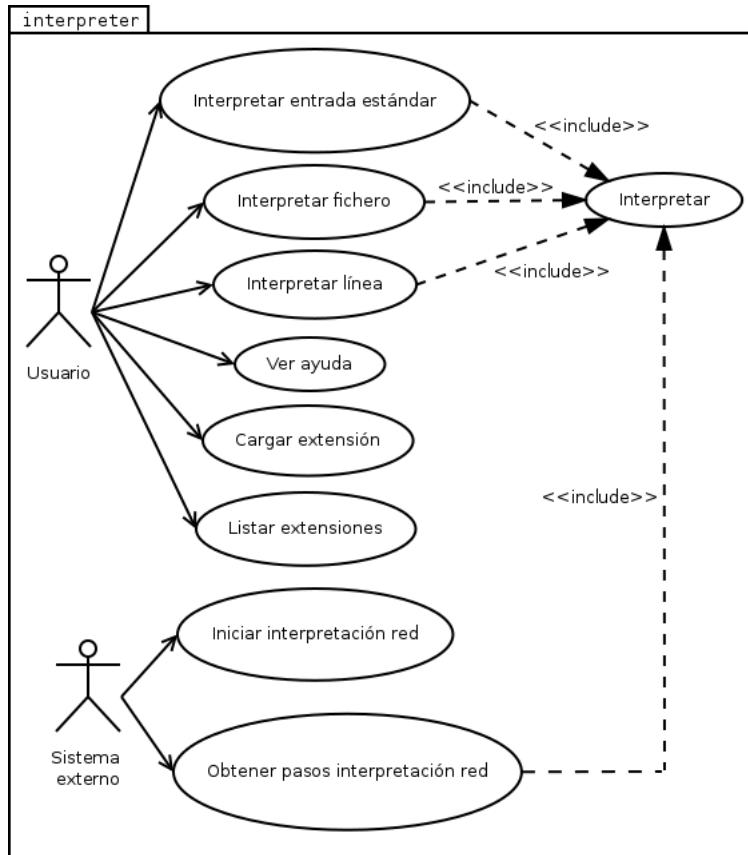


Figura 5.141: Casos de usos intérprete

5.2.2.1. Casos de usos interpretar entrada estándar

Caso de Uso: Interpretar entrada estándar.

Tipo: General.

Descripción: El usuario introduce un bloque de código en forma de cadena de caracteres mediante la entrada estándar. El sistema lo interpreta y ejecuta.

Actores: Usuario.

Precondiciones: El sistema debe estar esperando un bloque de código mediante la entrada estándar.

Postcondiciones: El código introducido es interpretado y ejecutado.

Escenario principal:

1. El usuario inicia el sistema facilitando un listado de argumentos.
2. El sistema asigna como variables cada argumento y solicita contenido fuente al usuario.
3. El usuario introduce un bloque de código en la entrada estándar.

4. El sistema obtiene el bloque de código a interpretar mediante la entrada estándar.
5. Incluir (Interpretar).

5.2.2.2. Interpretar fichero

Caso de Uso: Interpretar fichero.

Tipo: General.

Descripción: El usuario indica un fichero que contiene código y que será interpretado y ejecutado por el sistema.

Actores: Usuario.

Precondiciones: El sistema espera que se le indique un fichero.

Postcondiciones: El fichero es leído y el código en el mismo es interpretado y ejecutado.

Escenario principal:

1. El usuario indica la ruta a un fichero y una serie de argumentos
2. El sistema lee el fichero y obtiene el código en el mismo, además asigna cada argumento a variables.
3. Incluir (Interpretar).

Flujo alternativo:

- 2a. El fichero indicado no se encuentra.
 1. El sistema informa del error y finaliza.

5.2.2.3. Interpretar línea

Caso de Uso: Interpretar línea.

Tipo: General.

Descripción: El usuario introduce bloques de códigos denominados líneas de una forma interactiva. El sistema solicita por la entrada estándar las líneas de código, que serán interpretadas y ejecutadas.

Actores: Usuario.

Precondiciones: El sistema se encuentra en modo interactivo.

Postcondiciones: Se interpreta cada línea introducida por el usuario

Escenario principal:

1. El usuario inicia el sistema facilitando un listado de argumentos y la opción de intérprete de línea.

2. El sistema asigna como variables cada argumento y muestra un prompt que indica que espera una línea de código.
3. El usuario introduce una línea de código.
4. El sistema lee de la entrada estándar la línea introducida.
5. Include (Interpretar).

El sistema repite el caso de uso hasta que se interpreta una sentencia que produzca la salida.

5.2.2.4. Interpretar

Caso de Uso: Interpretar.

Tipo: General.

Nivel: Subfunción.

Descripción: El sistema analiza, interpreta y ejecuta un bloque de código facilitado por el usuario. Para ello comprueba que este cumple con el léxico y la gramática del lenguaje que define, dividiéndolo a su vez en sentencias que serán interpretadas.

Precondiciones: Se dispone de un bloque de código.

Postcondiciones: El bloque de código es interpretado.

Escenario principal:

1. El sistema procesa y comprueba el bloque de código, aplicando la gramática y léxico que define.
2. El sistema obtiene e interpreta cada sentencia en el código, produciéndose el significado semántico que estas encierran.

Flujo alternativo:

- 1a. El código no respeta el léxico del lenguaje.
 1. El sistema informa del error y finaliza.
- 1b. El código no respeta la gramática del lenguaje.
 1. El sistema informa del error y finaliza.
- 2a. El código no contiene ninguna sentencia.
 1. El sistema finaliza.

5.2.2.5. Ver ayuda

Caso de Uso: Ver ayuda.

Tipo: General.

Descripción: Se muestra una ayuda que detalla cada opción disponible en el sistema.

Actores: Usuario.

Precondiciones: Sin precondiciones.

Postcondiciones: El sistema muestra un listado que presenta las distintas opciones.

Escenario principal:

1. El usuario indica que quiere visualizar la ayuda.
2. El sistema muestra un listado completo de las opciones que presenta.

5.2.2.6. Cargar extensión

Caso de Uso: Cargar extensión.

Tipo: General.

Descripción: El usuario indica que una extensión que será cargada por el sistema.

Actores: Usuario.

Precondiciones: Sin precondiciones.

Postcondiciones: El sistema cargar la extensión facilitada.

Escenario principal:

1. El usuario indica la ruta a la extensión que desea cargar.
2. El sistema carga la extensión para disponer de las distintas opciones que ofrece.

Flujo alternativo:

- 2a. La extensión indicada no se encuentra.
 1. El sistema informa del error.
- 2b. La extensión indicada no es una extensión válida.
 1. El sistema informa del error.

5.2.2.7. Listar extensiones

Caso de Uso: Listar extensiones.

Tipo: General.

Descripción: Lista las extensiones que serán cargadas en cada ejecución del sistema.

Actores: Usuario.

Precondiciones: Sin precondiciones.

Postcondiciones: El sistema lista las extensiones que serán cargadas.

Escenario principal:

1. El usuario indica que desea listar las extensiones cargadas.
2. El sistema lista las extensiones cargadas por defecto.

5.2.2.8. Iniciar interpretación red

Caso de Uso: Iniciar interpretación red

Tipo: Red

Descripción: Un sistema externo inicia una interpretación por red, estableciendo el contenido fuente que será interpretado. El sistema procesa la petición y abre un nuevo proceso de interpretación por pasos.

Precondiciones: No tiene.

Postcondiciones: Se establece el código fuente a interpretar y se inicia una interpretación por pasos.

Escenario principal:

1. El sistema espera una petición de interpretación por red.
2. El sistema externo inicia la comunicación y facilita el código fuente.
3. El sistema establece el código fuente y abre un nuevo proceso de interpretación por pasos.

Flujo alternativo:

- 1a. El servicio no se encuentra disponible.
 1. Se devuelve un estado de error.

5.2.2.9. Obtener pasos interpretación red

Caso de Uso: Obtener pasos interpretación red

Tipo: Red

Descripción: Un sistema externo obtiene un nuevo estado dentro de los pasos en el proceso de interpretación del código fuente establecido.

Precondiciones: Se ha establecido código fuente para una interpretación por pasos.

Postcondiciones: Se lleva a cabo un nuevo paso dentro del proceso de interpretación. Obteniéndose el siguiente estado.

Escenario principal:

1. El sistema espera la petición de un nuevo paso en una iterpretación por red.

2. El sistema externo realiza la petición de un nuevo paso.
3. Incluir (Interpretar).
4. El sistema devuelve una estructura de datos que representa el estado actual del proceso de interpretación.

Flujo alternativo:

- 1a. El servicio no se encuentra disponible.
 1. Se devuelve un estado de error.
- 3a. El código fuente presenta errores sintáctico o léxicos.
 1. Se devuelve un estado de error.

5.2.3. runTree

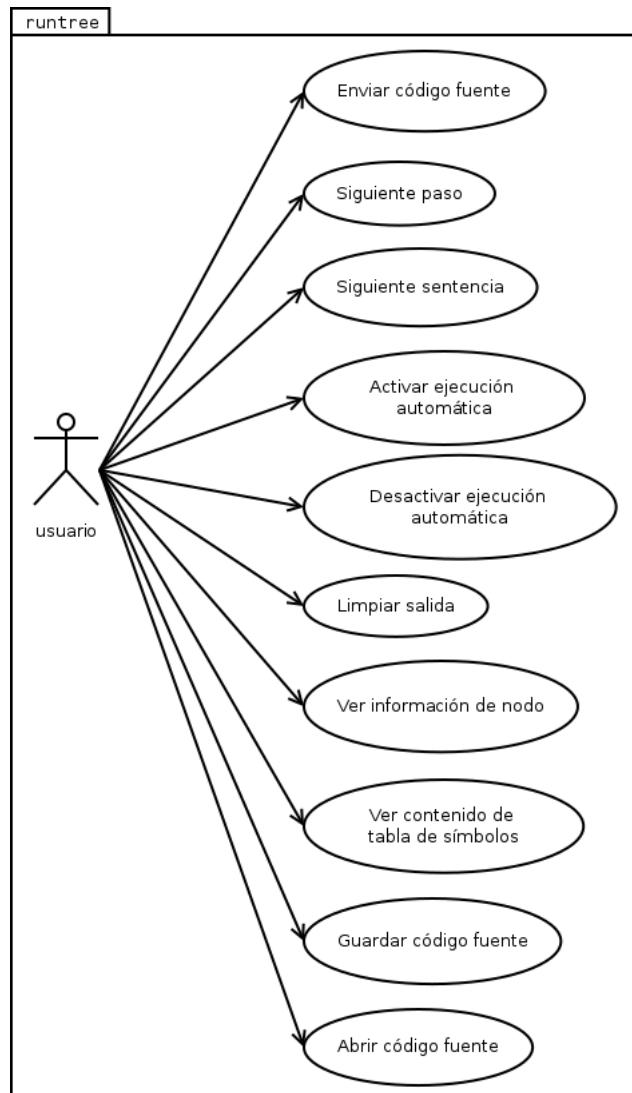


Figura 5.142: Casos de usos runTree

5.2.3.1. Enviar código fuente

Caso de Uso: Enviar código fuente

Tipo: runTree

Descripción: El usuario envía texto correspondiente a código fuente para su interpretación y análisis. El sistema cliente envía el código al servidor que lo establecerá como fuente a interpretar y enviará datos relativos al proceso. El cliente imprime el árbol sintáctico y espera acciones del usuario.

Precondiciones: No tiene.

Postcondiciones: Se ha establecido el código fuente para el análisis del proceso de interpretación y se ha imprimido el árbol sintáctico.

Escenario principal:

1. El sistema solicita código fuente escrito en OMI.
2. El usuario introduce código fuente escrito en el lenguaje OMI.
3. El sistema cliente runtree envía el código fuente al servidor para su interpretación.
4. El sistema servidor devuelve una representación de los datos que describen el árbol sintáctico relativo al código fuente enviado.
5. El sistema cliente imprime el árbol sintáctico.

Flujo alternativo:

- 4a. El código fuente no presenta una sintaxis OMI correcta.
 1. Se devuelve un estado de error.
- 4b. El servicio no se encuentra disponible.
 1. Se devuelve un estado de error.

5.2.3.2. Siguiente paso

Caso de Uso: Siguiente paso

Tipo: runTree

Descripción: El usuario usuario solicita un nuevo paso en el proceso de interpretación. El sistema cliente obtiene el nuevo paso del servidor y lo representa en pantalla.

Precondiciones:

- Existe un código fuente establecido para estudio.
- El estado actual del proceso no es final.
- La ejecución automática se encuentra desactivada.

Postcondiciones: Se representa en nuevo paso.

Escenario principal:

1. El usuario solicita un nuevo paso en el proceso de interpretación
2. El sistema cliente solicita la resolución de un nuevo paso y representa el nuevo estado en pantalla.

Flujo alternativo:

- 2a. El servicio no se encuentra disponible.
 1. Se devuelve un estado de error.

5.2.3.3. Siguiente sentencia**Caso de Uso:** Siguiente sentencia**Tipo:** runTree

Descripción: El usuario solicita la resolución de la siguiente sentencia. El sistema cliente obtiene el estado correspondiente a la interpretación de la siguiente sentencia dentro del proceso de interpretación y lo representa en pantalla.

Precondiciones:

- Existe un código fuente establecido para estudio.
- El estado actual del proceso no es final.
- La ejecución automática se encuentra desactivada.

Postcondiciones: Se representa en nuevo estado correspondiente a la interpretación de una sentencia completa.

Escenario principal:

1. El usuario solicita la resolución de una nueva sentencia en el proceso de interpretación.
2. El sistema cliente solicita la resolución de la sentencia completa y representa el nuevo estado en pantalla.

Flujo alternativo:

- 2a. El servicio no se encuentra disponible.
 1. Se devuelve un estado de error.

5.2.3.4. Activar ejecución automática**Caso de Uso:** Activar ejecución automática**Tipo:** runTree

Descripción: El usuario solicita la ejecución automática. El sistema cliente obtiene y representa cada paso dentro del proceso de interpretación hasta obtenerse un estado final.

Precondiciones:

- Existe un código fuente establecido para estudio.
- El estado actual del proceso no es final.
- La ejecución automática se encuentra desactivada.

Postcondiciones: Se obtiene un estado final.

Escenario principal:

1. El usuario activa la ejecución automática.
2. El sistema obtiene y representa cada paso en el proceso de interpretación hasta que se obtiene un estado final.

Flujo alternativo:

- 2a. El servicio no se encuentra disponible.
 1. Se devuelve un estado de error.

5.2.3.5. Desactivar ejecución automática

Caso de Uso: Desactivar ejecución automática

Tipo: runTree

Descripción: El usuario solicita la detención de la ejecución automática. El sistema cliente detiene la ejecución automática y representa el estado actual.

Precondiciones:

- Existe un código fuente establecido para estudio.
- El estado actual del proceso no es final.
- La ejecución automática se encuentra activada.

Postcondiciones: Se obtiene el estado actual del proceso.

Escenario principal:

1. El usuario desactiva la ejecución automática.
2. El sistema cliente para la ejecución automática y representa el estado actual

5.2.3.6. Limpiar salida

Caso de Uso: Limpiar salida

Tipo: runTree

Descripción: El usuario solicita la limpieza de los datos de salida. El sistema cliente elimina la información disponible en la consola de salida.

Precondiciones: No tiene.

Postcondiciones: La consola de salida queda vacía.

Escenario principal:

1. El usuario solicita la limpieza de la salida.
2. El sistema cliente limpia la información disponible en la consola de salida.

5.2.3.7. Ver información de nodo

Caso de Uso: Ver información de nodo.

Tipo: runTree

Descripción: El usuario marca un nodo para ver su información. El sistema cliente muestra la información relativa al nodo.

Precondiciones: Existe un código fuente establecido para estudio.

Postcondiciones: Se muestra información relativa al nodo.

Escenario principal:

1. El usuario marca un nodo para ver su información.
2. El sistema muestra información relativa al nodo tal como su posición de memoria interna, su tamaño, su tipo y su nombre.

5.2.3.8. Ver contenido de la tabla de símbolos

Caso de Uso: Ver contenido de la tabla de símbolos.

Tipo: runTree

Descripción: El usuario marca una tabla de símbolos para ver su contenido. El sistema cliente muestra la información relativa a la tabla de símbolos.

Precondiciones: Existe un código fuente establecido para estudio.

Postcondiciones: Se muestra información relativa a la tabla de símbolos.

Escenario principal:

1. El usuario indica una tabla de símbolos para ver su contenido.
2. El sistema cliente muestra los nodos referenciados desde la tabla de símbolos dada.

5.2.3.9. Guardar código fuente

Caso de Uso: Guardar código fuente.

Tipo: runTree

Descripción: El usuario solicita guardar el código fuente escrito en un fichero local y el cliente abre el cuadro de diálogo correspondiente.

Precondiciones: Existe un código escrito en el cliente

Postcondiciones: Se guarda el código fuente en un fichero local.

Escenario principal:

1. El usuario solicita guardar el código fuente en un fichero local.
2. El sistema abre el cuadro de diálogo correspondiente.

5.2.3.10. Abrir código fuente

Caso de Uso: Abrir código fuente.

Tipo: runTree

Descripción: El usuario solicita cargar código fuente desde un fichero local.

Precondiciones: No tiene.

Postcondiciones: Se carga el código fuente contenido en el fichero.

Escenario principal:

1. El usuario solicita abrir un fichero de código fuente.
2. El sistema abre el cuadro de diálogo correspondiente.
3. El sistema carga el contenido del código fuente.

Flujo alternativo:

- 2c. El fichero no contiene código en texto plano.
 1. Se devuelve un estado de error.

5.3. Comportamiento del sistema

Partiendo de los casos de usos se presentan los diagramas de secuencias que modelan los eventos que el sistema puede recibir del usuario y los valores de retorno que produce como consecuencia de estos. Esta sección considera tanto al sistema intérprete como el cliente.

A partir de los diagramas de secuencia del sistema se obtienen las operaciones que este presenta. Luego se describen los contratos de cada una de las operaciones.

5.4. Diagramas de secuencia del sistema

5.4.1. Intérprete

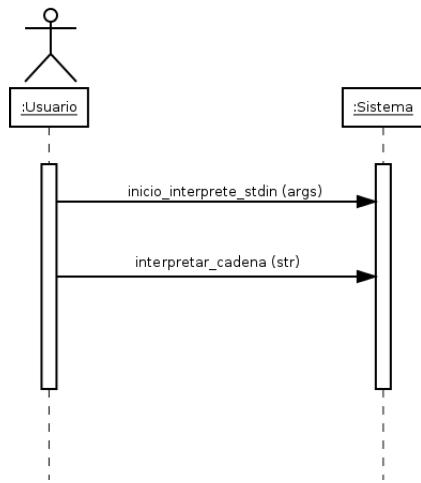


Figura 5.143: Secuencia interpretar entrada estándar

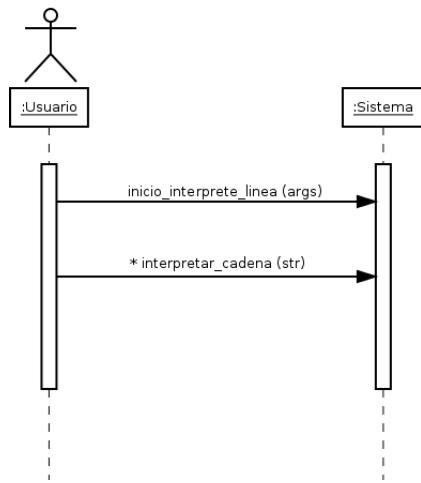


Figura 5.144: Secuencia interpretar línea

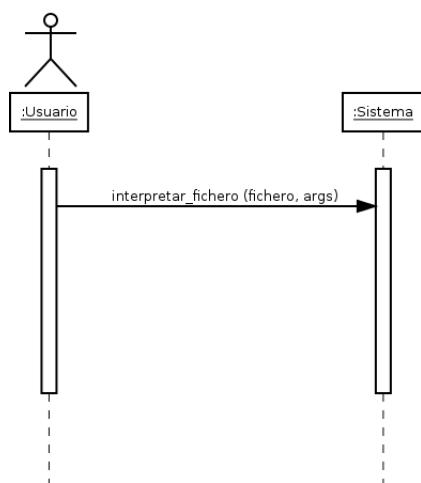


Figura 5.145: Secuencia interpretar fichero

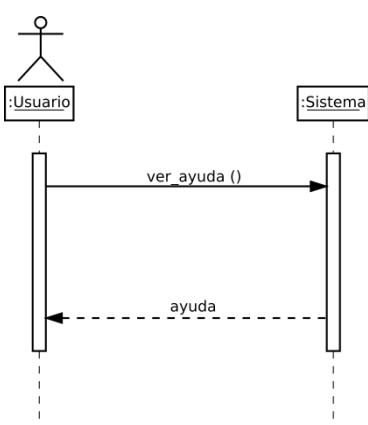


Figura 5.146: Secuencia ver ayuda

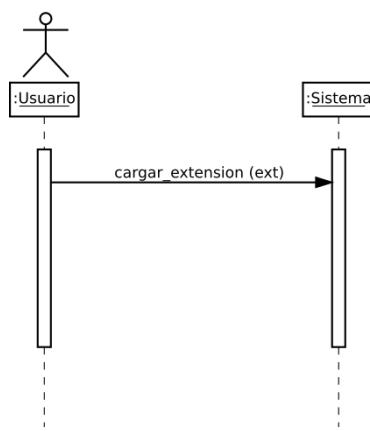


Figura 5.147: Secuencia cargar extensión

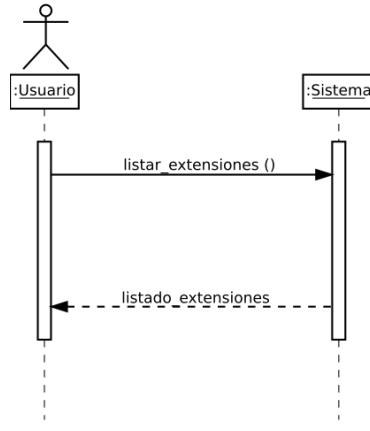


Figura 5.148: Secuencia listar extensiones

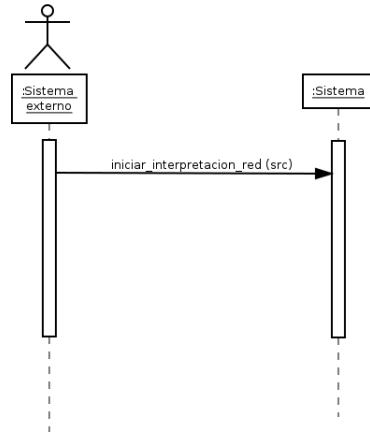


Figura 5.149: Secuencia iniciar interpretación red

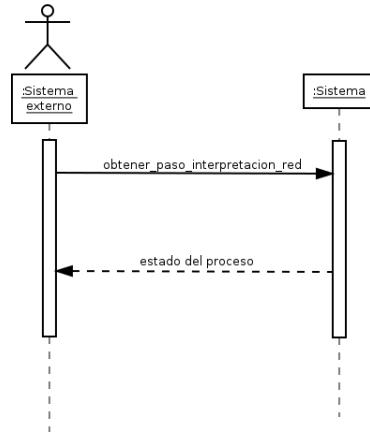


Figura 5.150: Secuencia obtener pasos interpretación red

5.4.2. runTree

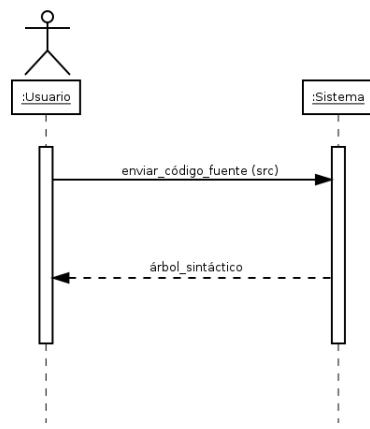


Figura 5.151: Secuencia enviar código fuente

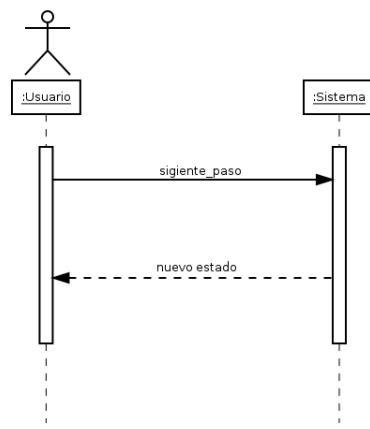


Figura 5.152: Secuencia siguiente paso

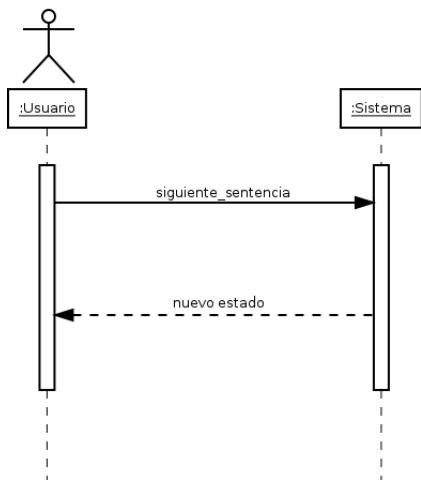


Figura 5.153: Secuencia siguiente sentencia

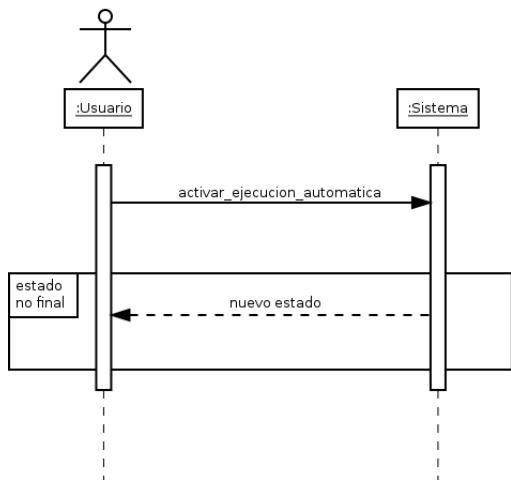


Figura 5.154: Secuencia activar ejecución automática

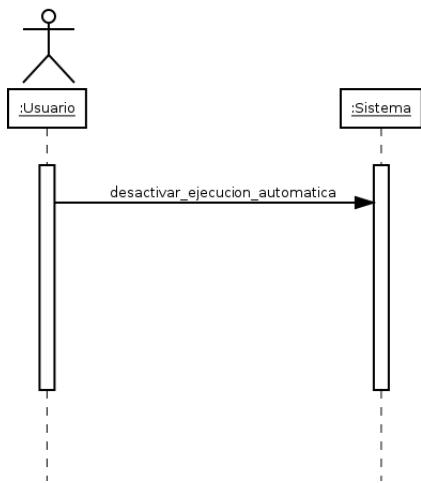


Figura 5.155: Secuencia desactivar ejecución automática

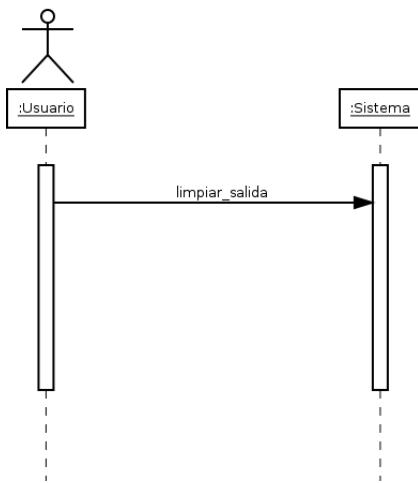


Figura 5.156: Secuencia limpiar salida

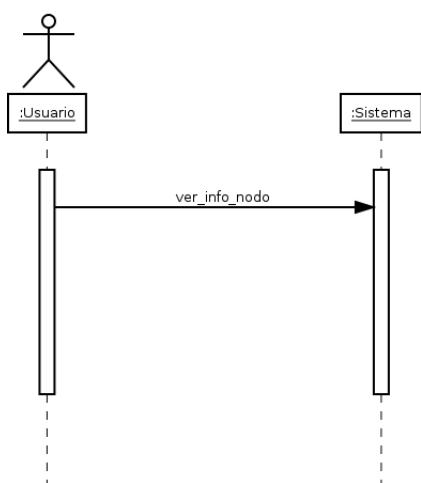


Figura 5.157: Secuencia ver información de nodo

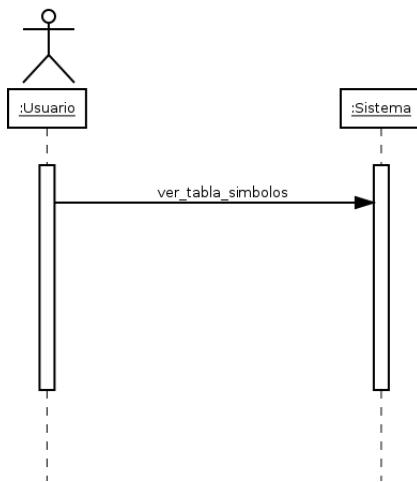


Figura 5.158: Secuencia ver contenido tabla de símbolo

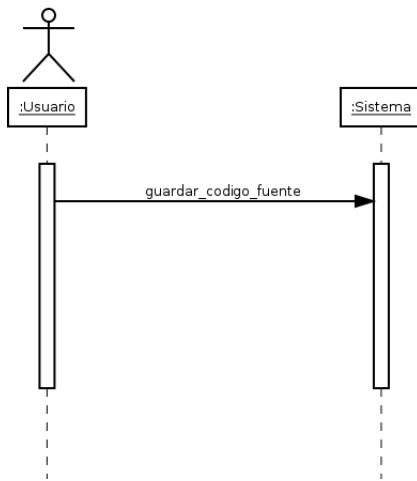


Figura 5.159: Secuencia guardar código fuente

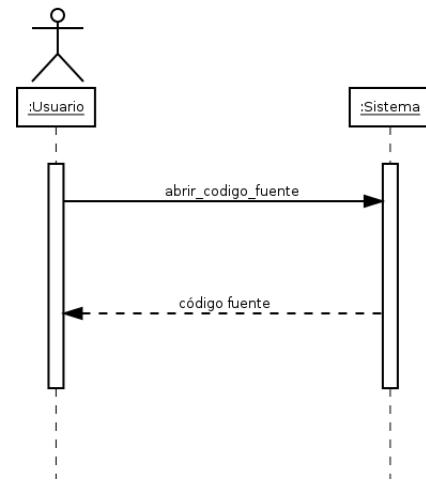


Figura 5.160: Secuencia abrir código fuente

5.5. Contratos de operaciones del sistema

En esta sección se listan las operaciones del sistema y se detallan los contratos de aquellas que implican un cambio en la estructura de datos interna del programa.

5.5.1. Intérprete

Sistema
+inicio_interprete_stdin(args) +inicio_interprete_linea(args) +interpretar_cadena(str) +interpretar_fichero(fichero,args) +ver_ayuda() +cargar_extension(ext) +listar_extensiones() +iniciar_interpretacion_red(src) +obtener_paso_interpretacion_red()

Figura 5.161: Contrato de operaciones intérprete

5.5.1.1. Operación inicio_interprete_stdin

Nombre: inicio_interprete_stdin(args)

Responsabilidades: Iniciar la interpretación de la entrada estándar.

Referencias Cruzadas: Caso de Uso: Interpretar entrada estándar

Precondiciones: No tiene.

Postcondiciones:

- Se creó una instancia “i” de “interpreter” (creación de objeto).
- Se inicializaron los atributos de “i”.
- Se crearon las instancias “a₀...a_n” de “arg” por cada argumento en “args” (creación de objeto).
- Se asignó “args_i” a “a_i.value” (a_i.value = args_i) (modificación de atributos).
- Se asoció los “arg” “a₀...a_n” al objeto “i” de “interpreter” (creación de enlace).
- Se creó una instancia “p” de “parser” (creación de objeto).
- Se inicializaron los atributos de “p”.
- Se creó una instancia “s” de “scanner” (creación de objeto).
- Se inicializaron los atributos de “s”.
- Se asoció el “scanner” “s” al objeto “p” de “parser” (creación de enlace).
- Se asoció el “parser” “p” al objeto “i” de “interpreter” (creación de enlace).
- Se creó una instancia “c” de “context” (creación de objeto).
- Se inicializaron los atributos de “c”.
- Se creó una instancia “var” de “varSymbols” (creación de objeto).
- Se inicializaron los atributos de “var”.
- Se asoció el “varSymbols” “var” al objeto “c” de “context” (creación de enlace).
- Se creó una instancia “func” de “funcSymbols” (creación de objeto).

- Se inicializaron los atributos de “func”.
- Se asoció el “funcSymbols” “func” al objeto “c” de “context” (creación de enlace).
- Se creó una instancia “class” de “classSymbols” (creación de objeto).
- Se inicializaron los atributos de “class”.
- Se asoció el “classSymbols” “class” al objeto “c” de “context” (creación de enlace).
- Se asoció el “context” “c” al objeto “i” de “interpreter” (creación de enlace).

5.5.1.2. Operación inicio_interprete_linea

Nombre: inicio_interprete_linea(args)

Responsabilidades: Iniciar la interpretación interactiva línea a línea.

Referencias Cruzadas: Caso de Uso: Interpretar línea

Precondiciones: No tiene.

Postcondiciones:

- Se creó una instancia “i” de “interpreter” (creación de objeto).
- Se inicializaron los atributos de “i”.
- Se crearon las instancias “ $a_0 \dots a_n$ ” de “arg” por cada argumento en “args” (creación de objeto).
- Se asignó “ $args_i$ ” a “ $a_i.value$ ” ($a_i.value = args_i$) (modificación de atributos).
- Se asoció los “arg” “ $a_0 \dots a_n$ ” al objeto “i” de “interpreter” (creación de enlace).
- Se creó una instancia “p” de “parser” (creación de objeto).
- Se inicializaron los atributos de “p”.
- Se creó una instancia “s” de “scanner” (creación de objeto).
- Se inicializaron los atributos de “s”.
- Se asoció el “scanner” “s” al objeto “p” de “parser” (creación de enlace).
- Se asoció el “parser” “p” al objeto “i” de “interpreter” (creación de enlace).
- Se creó una instancia “c” de “context” (creación de objeto).
- Se inicializaron los atributos de “c”.
- Se creó una instancia “var” de “varSymbols” (creación de objeto).
- Se inicializaron los atributos de “var”.
- Se asoció el “varSymbols” “var” al objeto “c” de “context” (creación de enlace).
- Se creó una instancia “func” de “funcSymbols” (creación de objeto).
- Se inicializaron los atributos de “func”.
- Se asoció el “funcSymbols” “func” al objeto “c” de “context” (creación de enlace).
- Se creó una instancia “class” de “classSymbols” (creación de objeto).
- Se inicializaron los atributos de “class”.
- Se asoció el “classSymbols” “class” al objeto “c” de “context” (creación de enlace).
- Se asoció el “context” “c” al objeto “i” de “interpreter” (creación de enlace).

5.5.1.3. Operación interpretar_cadena

Nombre: interpretar_cadena(str)

Responsabilidades: Interpreta el contenido fuente almacenado en la cadena “str”

Referencias Cruzadas:

- Caso de Uso: Interpretar entrada estándar
- Caso de Uso: Interpretar línea

Precondiciones:

- Se creó un “interpreter” “i”.
- Se creó y asoció una instancia de “parser” y “scanner” a “i”.
- Se creó y asoció una instancia de “varSymbols”, “funcSymbols” y “classSymbols” a “i”.

Postcondiciones:

- Se creó una instancia “s” de “source” (creación de objeto).
- Se asignó “str” a “s.src” ($s.\text{src} = \text{str}$) (modificación de atributos)
- Se asoció “s” al “scanner” componente del “interpreter” “i” (creación de enlace).
- Se creó un conjunto “ $t_0...t_n$ ” de “token” a partir del análisis léxico (creación de objeto).
- Se creó un conjunto “ $n_0...n_n$ ” de “runNode” a partir del análisis sintáctico (creación de objetos).
- Se asoció “ $n_i \in n_0...n_n$ ” a “ $n_k \in n_0...n_n$ ” para construir el árbol sintáctico (creación de enlace).
- Se asoció “ $n_r \in n_0...n_n$ ”, raíz del árbol sintáctico, al “interprter” “i” (creación de enlace).
- Se creó un conjunto “ $var_0...var_n$ ” de “refNode” correspondientes a las variables definidas en el contenido fuente (creación de objetos).
- Se creó un conjunto “ $val_0...val_n$ ” de “runNode” correspondientes a los valores asignados a las variables definidas en el contenido fuente (creación de objetos).
- Se asoció “ $val_i \in val_0...val_n$ ” a “ $var_i \in var_0...var_n$ ” donde val_i es el valor de la variable var_i (creación de enlace).
- Se asoció todo “refNode” “ $var_0...var_n$ ” al componente “varSymbols” de “i” (creación de enlace).
- Se creó un conjunto “ $func_0...func_n$ ” de “refNode” correspondientes a las funciones con identificador definidas en el contenido fuente (creación de objetos).
- Se asoció “ $func_i \in func_0...func_n$ ” a “ $n_i \in n_0...n_n$ ” donde n_i es un “funcNode” correspondiente a la definición de la función $func_i$ (creación de enlaces).
- Se asoció todo “refNode” “ $func_0...func_n$ ” al componente “funcSymbols” de “i” (creación de enlace).
- Se creó un conjunto “ $class_0...class_n$ ” de “refNode” correspondientes a las clases definidas en el contenido fuente (creación de objetos).
- Se asoció “ $class_i \in class_0...class_n$ ” a “ $n_i \in n_0...n_n$ ” donde n_i es un “classNode” correspondiente a la definición de la clase $class_i$ (creación de enlaces).
- Se asoció todo “refNode” “ $class_0...class_n$ ” al componente “classSymbols” de “i” (creación de enlace).

5.5.1.4. Operación interpretar_fichero

Nombre: interpretar_fichero(fichero, args)

Responsabilidades: Interpreta el contenido fuente almacenado en “fichero”

Referencias Cruzadas: Caso de Uso: Interpretar fichero

Precondiciones: No tiene.

Postcondiciones:

- Se creó una instancia “i” de “interpreter” (creación de objeto).
- Se inicializaron los atributos de “i”.
- Se creó el conjunto “ $a_0 \dots a_n$ ” de “arg” según el número de argumentos en “args” (creación de objeto).
- Se asignó “args $_i$ ” a “ $a_i.value$ ” ($a_i.value = args_i$) (modificación de atributos).
- Se asoció los “arg” “ $a_0 \dots a_n$ ” al objeto “i” de “interpreter” (creación de enlace).
- Se creó una instancia “p” de “parser” (creación de objeto).
- Se inicializaron los atributos de “p”.
- Se creó una instancia “s” de “scanner” (creación de objeto).
- Se inicializaron los atributos de “s”.
- Se asoció el “scanner” “s” al objeto “p” de “parser” (creación de enlace).
- Se asoció el “parser” “p” al objeto “i” de “interpreter” (creación de enlace).
- Se creó una instancia “c” de “context” (creación de objeto).
- Se inicializaron los atributos de “c”.
- Se creó una instancia “var” de “varSymbols” (creación de objeto).
- Se inicializaron los atributos de “var”.
- Se asoció el “varSymbols” “var” al objeto “c” de “context” (creación de enlace).
- Se creó una instancia “func” de “funcSymbols” (creación de objeto).
- Se inicializaron los atributos de “func”.
- Se asoció el “funcSymbols” “func” al objeto “c” de “context” (creación de enlace).
- Se creó una instancia “class” de “classSymbols” (creación de objeto).
- Se inicializaron los atributos de “class”.
- Se asoció el “classSymbols” “class” al objeto “c” de “context” (creación de enlace).
- Se asoció el “context” “c” al objeto “i” de “interpreter” (creación de enlace).
- Se creó una instancia “src” de “source” (creación de objeto).
- Se asignó el contenido de “fichero” a “src.src” ($src.src = fichero$) (modificación de atributos)
- Se asoció “src” al “scanner” “s” componente del “interpreter” “i” (creación de enlace).
- Se creó un conjunto “ $t_0 \dots t_n$ ” de “token” a partir del análisis léxico (creación de objeto).
- Se creó un conjunto “ $n_0 \dots n_n$ ” de “runNode” a partir del análisis sintáctico (creación de objetos).

- Se asoció “ $n_i \in n_0...n_n$ ” a “ $n_k \in n_0...n_n$ ” para construir el árbol sintáctico (creación de enlace).
- Se asoció “ $n_r \in n_0...n_n$ ”, raíz del árbol sintáctico, al “interpret” “i” (creación de enlace).
- Se creó un conjunto “ $var_0...var_n$ ” de “refNode” correspondientes a las variables definidas en el contenido fuente (creación de objetos).
- Se creó un conjunto “ $val_0...val_n$ ” de “runNode” correspondientes a los valores asignados a las variables definidas en el contenido fuente (creación de objetos).
- Se asoció “ $val_i \in val_0...val_n$ ” a “ $var_i \in var_0...var_n$ ” donde val_i es el valor de la variable var_i (creación de enlace).
- Se asoció todo “refNode” “ $var_0...var_n$ ” al componente “varSymbols” de “i” (creación de enlace).
- Se creó un conjunto “ $func_0...func_n$ ” de “refNode” correspondientes a las funciones con identificador definidas en el contenido fuente (creación de objetos).
- Se asoció “ $func_i \in func_0...func_n$ ” a “ $n_i \in n_0...n_n$ ” donde n_i es un “funcNode” correspondiente a la definición de la función $func_i$ (creación de enlaces).
- Se asoció todo “refNode” “ $func_0...func_n$ ” al componente “funcSymbols” de “i” (creación de enlace).
- Se creó un conjunto “ $class_0...class_n$ ” de “refNode” correspondientes a las clases definidas en el contenido fuente (creación de objetos).
- Se asoció “ $class_i \in class_0...class_n$ ” a “ $n_i \in n_0...n_n$ ” donde n_i es un “classNode” correspondiente a la definición de la clase $class_i$ (creación de enlaces).
- Se asoció todo “refNode” “ $class_0...class_n$ ” al componente “classSymbols” de “i” (creación de enlace).

5.5.1.5. Operación iniciar_interpretacion_red

Nombre: iniciar_interpretacion_red (src)

Responsabilidades: Iniciar la interpretación de una petición por red.

Referencias Cruzadas: Caso de Uso: Iniciar interpretación red

Precondiciones: No tiene.

Postcondiciones:

- Se creó una instancia “i” de “interpreter” (creación de objeto).
- Se inicializaron los atributos de “i”.
- Se creó una instancia “p” de “parser” (creación de objeto).
- Se inicializaron los atributos de “p”.
- Se creó una instancia “s” de “scanner” (creación de objeto).
- Se inicializaron los atributos de “s”.
- Se asoció el “scanner” “s” al objeto “p” de “parser” (creación de enlace).
- Se asoció el “parser” “p” al objeto “i” de “interpreter” (creación de enlace).

- Se creó una instancia “c” de “context” (creación de objeto).
- Se inicializaron los atributos de “c”.
- Se creó una instancia “var” de “varSymbols” (creación de objeto).
- Se inicializaron los atributos de “var”.
- Se asoció el “varSymbols” “var” al objeto “c” de “context” (creación de enlace).
- Se creó una instancia “func” de “funcSymbols” (creación de objeto).
- Se inicializaron los atributos de “func”.
- Se asoció el “funcSymbols” “func” al objeto “c” de “context” (creación de enlace).
- Se creó una instancia “class” de “classSymbols” (creación de objeto).
- Se inicializaron los atributos de “class”.
- Se asoció el “classSymbols” “class” al objeto “c” de “context” (creación de enlace).
- Se asoció el “context” “c” al objeto “i” de “interpreter” (creación de enlace).
- Se creó una instancia “src” de “source” (creación de objeto).
- Se asignó el contenido de “fichero” a “src.src” (src.src = fichero) (modificación de atributos)
- Se asoció “src” al “scanner” “s” componente del “interpreter” “i” (creación de enlace).
- Se creó un conjunto “ $t_0 \dots t_n$ ” de “token” a partir del análisis léxico (creación de objeto).
- Se creó un conjunto “ $n_0 \dots n_n$ ” de “runNode” a partir del análisis sintáctico (creación de objetos).
- Se asoció “ $n_i \in n_0 \dots n_n$ ” a “ $n_k \in n_0 \dots n_n$ ” para construir el árbol sintáctico (creación de enlace).
- Se asoció “ $n_r \in n_0 \dots n_n$ ”, raíz del árbol sintáctico, al “interprter” “i” (creación de enlace).

5.5.1.6. Operación obtener_paso_interpretacion_red

Nombre: obtener_paso_interpretacion_red

Responsabilidades: Obtiene el siguiente paso de la interpretación de una petición por red abierta.

Referencias Cruzadas: Caso de Uso: Obtener pasos interpretación red

Precondiciones:

- Se creó un “interpreter” “i”.
- Se creó y asoció una instancia de “parser” y “scanner” a “i”.
- Se creó y asoció una instancia de “varSymbols”, “funcSymbols” y “classSymbols” a “i”.
- Se creó una instancia “src” de “source”.
- Se asoció “src” al “scanner” “s” componente del “interpreter” “i”.
- Se creó un conjunto “ $t_0 \dots t_n$ ” de “token” a partir del análisis léxico.
- Se creó un conjunto “ $n_0 \dots n_n$ ” de “runNode” a partir del análisis sintáctico.
- Se asoció “ $n_i \in n_0 \dots n_n$ ” a “ $n_k \in n_0 \dots n_n$ ” para construir el árbol sintáctico.

- Se asoció “ $n_r \in n_0...n_n$ ”, raíz del árbol sintáctico, al “interprter” “i”.

Postcondiciones:

- Se creó un conjunto “ $var_0...var_n$ ” de “refNode” correspondientes a las variables definidas en el paso correspondiente (creación de objetos).
- Se creó un conjunto “ $val_0...val_n$ ” de “runNode” correspondientes a los valores asignados a las variables definidas en el paso correspondiente (creación de objetos).
- Se asoció “ $val_i \in val_0...val_n$ ” a “ $var_i \in var_0...var_n$ ” donde val_i es el valor de la variable var_i (creación de enlace).
- Se asoció todo “refNode” “ $var_0...var_n$ ” al componente “varSymbols” de “i” (creación de enlace).
- Se creó un conjunto “ $func_0...func_n$ ” de “refNode” correspondientes a las funciones con identificador definidas en el paso correspondiente (creación de objetos).
- Se asoció “ $func_i \in func_0...func_n$ ” a “ $n_i \in n_0...n_n$ ” donde n_i es un “funcNode” correspondiente a la definición de la función $func_i$ (creación de enlaces).
- Se asoció todo “refNode” “ $func_0...func_n$ ” al componente “funcSymbols” de “i” (creación de enlace).
- Se creó un conjunto “ $class_0...class_n$ ” de “refNode” correspondientes a las clases definidas en el paso correspondiente (creación de objetos).
- Se asoció “ $class_i \in class_0...class_n$ ” a “ $n_i \in n_0...n_n$ ” donde n_i es un “classNode” correspondiente a la definición de la clase $class_i$ (creación de enlaces).
- Se asoció todo “refNode” “ $class_0...class_n$ ” al componente “classSymbols” de “i” (creación de enlace).

5.5.2. runTree

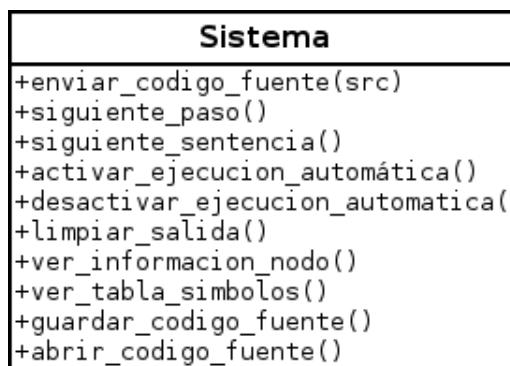


Figura 5.162: Contrato de operaciones runTree

5.5.2.1. Operación enviar_codigo_fuente

Nombre: enviar_codigo_fuente (src)

Responsabilidades: Envía código fuente para una interpretación por red.

Referencias Cruzadas: Caso de Uso: Enviar código fuente

Precondiciones: No tiene.

Postcondiciones:

- Se creó un “tree” “t” a partir del código enviado a interpretación (creación de objeto).
- Se creó un conjunto “ $n_0...n_n$ ” de “node” a partir de los datos recibidos de la petición (creación de objeto).
- Se asoció “ $n_i \in n_0...n_n$ ” a “ $n_k \in n_0...n_n$ ” para construir el árbol sintáctico (creación de enlace).
- Se asoció “ $n_r \in n_0...n_n$ ”, raíz del árbol sintáctico, al “tree” “t” (creación de enlace).
- Se inicializó el atributo “canvas” del “tree” “t” para dibujar el árbol correspondiente a los nodos “ $n_0...n_n$ ”.
- Se creó un “symbols” “s” (creación de objeto).
- Se creó tres instancias de “table”: “vars”, “funcs” y “class” (creación de objeto).
- Se asociaron “vars”, “funcs” y “class” a “s” (creación de enlace).

5.5.2.2. Operación siguiente _paso

Nombre: siguiente _paso

Responsabilidades: Obtiene el siguiente paso del proceso de interpretación abierto.

Referencias Cruzadas: Caso de Uso: Siguiente paso

Precondiciones:

- Se creó un “tree” “t”.
- Se creó un conjunto “ $n_0...n_n$ ” de “node”.
- Se asoció “ $n_i \in n_0...n_n$ ” a “ $n_k \in n_0...n_n$ ” para construir el árbol sintáctico.
- Se asoció “ $n_r \in n_0...n_n$ ”, raíz del árbol sintáctico, al “tree” “t”.
- Se creó un “symbols” “s”.
- Se creó tres instancias de “table”: “vars”, “funcs” y “class”.
- Se asociaron “vars”, “funcs” y “class” a “s”.

Postcondiciones:

- Se creó el conjunto “ $r_0...r_n$ ” de ‘refs’ según el nuevo paso del proceso de interpretación (creación de objeto).
- Se creó el conjunto “ $v_0...v_m$ ” de ‘node’ correspondiente a los valores generados en el paso del proceso de interpretación (creación de objeto).
- Se asoció “ v_i ” a “ r_i ” como valor de la referencia (creación de enlace).
- Se asoció “ $r_i \in r_0...r_n$ ” a “vars”, “funcs” o “class” según el paso del proceso de interpretación (creación de enlace).
- Se actualizó el valor del atributo “canvas” de “s” para reflejar el nuevo estado tras la ejecución del paso.
- Se actualizó el valor del atributo “canvas” de “t” para reflejar el nuevo estado tras la ejecución del paso.

5.5.2.3. Operación siguiente _ sentencia

Nombre: siguiente _ sentencia

Responsabilidades: Obtiene la siguiente sentencia dentro del proceso de interpretación abierto.

Referencias Cruzadas: Caso de Uso: Siguiente sentencia

Precondiciones:

- Se creó un “tree” “t”.
- Se creó un conjunto “ $n_0...n_n$ ” de “node”.
- Se asoció “ $n_i \in n_0...n_n$ ” a “ $n_k \in n_0...n_n$ ” para construir el árbol sintáctico.
- Se asoció “ $n_r \in n_0...n_n$ ”, raíz del árbol sintáctico, al “tree” “t”.
- Se creó un “symbols” “s”.
- Se creó tres instancias de “table”: “vars”, “funcs” y “class”.
- Se asociaron “vars”, “funcs” y “class” a “s”.

Postcondiciones:

- Se creó el conjunto “ $r_0...r_n$ ” de ‘refs’ según la nueva sentencia interpretada (creación de objeto).
- Se creó el conjunto “ $v_0...v_m$ ” de ‘node’ correspondiente a los valores generados en la sentencia interpretada (creación de objeto).
- Se asoció “ v_i ” a “ r_i ” como valor de la referencia (creación de enlace).
- Se asoció “ $r_i \in r_0...r_n$ ” a “vars”, “funcs” o “class” según la sentencia interpretada (creación de enlace).
- Se actualizó el valor del atributo “canvas” de “s” para reflejar el nuevo estado tras la ejecución de la sentencia.
- Se actualizó el valor del atributo “canvas” de “t” para reflejar el nuevo estado tras la ejecución de la sentencia.

5.5.2.4. Operación activar _ ejecucion _ automatica

Nombre: activar _ ejecucion _ automatica

Responsabilidades: Activa la ejecución automática del proceso de interpretación.

Referencias Cruzadas: Caso de Uso: Activar ejecución automática

Precondiciones:

- Se creó un “tree” “t”.
- Se creó un conjunto “ $n_0...n_n$ ” de “node”.
- Se asoció “ $n_i \in n_0...n_n$ ” a “ $n_k \in n_0...n_n$ ” para construir el árbol sintáctico.
- Se asoció “ $n_r \in n_0...n_n$ ”, raíz del árbol sintáctico, al “tree” “t”.

- Se creó un “symbols” “s”.
- Se creó tres instancias de “table”: “vars”, “funcs” y “class”.
- Se asociaron “vars”, “funcs” y “class” a “s”.

Postcondiciones:

- Se establece a “1” el atributo “auto” de “t”.

5.5.2.5. Operación desactivar_ejecucionAutomatica

Nombre: desactivar_ejecucionAutomatica

Responsabilidades: Desactiva la ejecución automática del proceso de interpretación.

Referencias Cruzadas: Caso de Uso: Desactivar ejecución automática

Precondiciones:

- Se creó un “tree” “t”.
- Se creó un conjunto “ $n_0 \dots n_n$ ” de “node”.
- Se asoció “ $n_i \in n_0 \dots n_n$ ” a “ $n_k \in n_0 \dots n_n$ ” para construir el árbol sintáctico.
- Se asoció “ $n_r \in n_0 \dots n_n$ ”, raíz del árbol sintáctico, al “tree” “t”.
- Se creó un “symbols” “s”.
- Se creó tres instancias de “table”: “vars”, “funcs” y “class”.
- Se asociaron “vars”, “funcs” y “class” a “s”.

Postcondiciones:

- Se establece a “0” el atributo “auto” de “t”.

5.5.2.6. Operación limpiar_salida

Nombre: limpiar_salida

Responsabilidades: Limpia la salida producida por el proceso de interpretación

Referencias Cruzadas: Caso de Uso: Limpiar salida

Precondiciones:

- Se creó un “tree” “t”.
- Se creó un conjunto “ $n_0 \dots n_n$ ” de “node”.
- Se asoció “ $n_i \in n_0 \dots n_n$ ” a “ $n_k \in n_0 \dots n_n$ ” para construir el árbol sintáctico.
- Se asoció “ $n_r \in n_0 \dots n_n$ ”, raíz del árbol sintáctico, al “tree” “t”.
- Se creó un “symbols” “s”.
- Se creó tres instancias de “table”: “vars”, “funcs” y “class”.

- Se asociaron “vars”, “funcs” y “class” a “s”.

Postcondiciones:

- Se restablece los valores del atributo “canvas” de “t”.

5.5.2.7. Operación ver_informacion_nodo

Nombre: ver_informacion_nodo

Responsabilidades: Muestra la información de un nodo

Referencias Cruzadas: Caso de Uso: Ver información nodo

Precondiciones:

- Se creó un “tree” “t”.
- Se creó un conjunto “ $n_0 \dots n_n$ ” de “node”.
- Se asoció “ $n_i \in n_0 \dots n_n$ ” a “ $n_k \in n_0 \dots n_n$ ” para construir el árbol sintáctico.
- Se asoció “ $n_r \in n_0 \dots n_n$ ”, raíz del árbol sintáctico, al “tree” “t”.
- Se creó un “symbols” “s”.
- Se creó tres instancias de “table”: “vars”, “funcs” y “class”.
- Se asociaron “vars”, “funcs” y “class” a “s”.

Postcondiciones:

- Se establece los valores del atributo “canvas” de “t” para representar la información del nodo.

5.5.2.8. Operación ver_tabla_simbolos

Nombre: ver_tabla_simbolos

Responsabilidades: Muestra la información contenida en una tabla de símbolos

Referencias Cruzadas: Caso de Uso: Ver información nodo

Precondiciones:

- Se creó un “tree” “t”.
- Se creó un conjunto “ $n_0 \dots n_n$ ” de “node”.
- Se asoció “ $n_i \in n_0 \dots n_n$ ” a “ $n_k \in n_0 \dots n_n$ ” para construir el árbol sintáctico.
- Se asoció “ $n_r \in n_0 \dots n_n$ ”, raíz del árbol sintáctico, al “tree” “t”.
- Se creó un “symbols” “s”.
- Se creó tres instancias de “table”: “vars”, “funcs” y “class”.
- Se asociaron “vars”, “funcs” y “class” a “s”.

Postcondiciones:

- Se establece los valores del atributo “canvas” de “s” para representar la información de la tabla de símbolos.

5.6. Modelo de interfaz de usuario

En esta sección se procede al análisis de la interfaz de usuario. Para ello se analizará la interfaz precisa para el intérprete y para el cliente runTree.

El intérprete utilizará una interfaz de consola de comandos. Toda la información de salida la presentará como cadenas de caracteres. Así mismo toda la información de entrada la tomará del teclado en formato orden y opciones.

Por otro lado el cliente runTree presentará una interfaz en la que se disponga de una descripción gráfica de todo el proceso de interpretación. Para una descripción del proceso de interpretación se precisa de la visualización del código fuente, un árbol con los nodos que encierran el significado semántico del código, las distintas tablas de símbolos que referenciarán a variables, funciones y clases, y una consola en la que se mostrará información textual.

También un diagrama de navegación en el que se expone las distintas ventanas de la web OMI.

5.6.1. Intérprete

El interprete será accesible como cualquier otro comando de la consola del sistema. Este recibirá una serie de opciones y parámetros. Las opciones del intérprete tendrán los siguientes propósitos:

- Determinar cómo se toma el código fuente, pudiéndo ser desde la entrada estándar, un fichero o la propia línea de comandos
- Ejecutar el intérprete de forma interactiva, mostrando un prompt en el que se introduzca directamente las sentencias
- Listar y cargar los módulos del intérprete.
- Ver la ayuda.
- Abrir un puerto de escucha para peticiones de red.
- Configurar el formato de la salida que describe el proceso de interpretación.

Como cualquier interprete que tome su entrada de la estrada estándar, el interprete OMI puede ser ejecutado por el sistema operativo si se indica al comienzo de un script como el shebang del mismo.

5.6.2. runTree

El cliente runTree será accesible desde un navegador web, y presentará una interfaz gráfica compatible con las versiones actuales de estos. La interfaz gráfica del cliente deberá contener la siguiente información:

- El código fuente introducido por el usuario y que será enviado a interpretar.
- El árbol de nodos resultado del análisis léxico y sintáctico.
- Las distintas tablas de símbolos que guardarán información sobre las variables, las funciones y las clases que serán creadas.
- La explicación del proceso semántico llevado a cabo.
- La salida producida como fruto de la ejecución del código fuente.
- La entrada introducida por el usuario y que ha sido solicitada por el código fuente.
- Información relativa a los nodos y tablas de símbolos que el usuario señale.

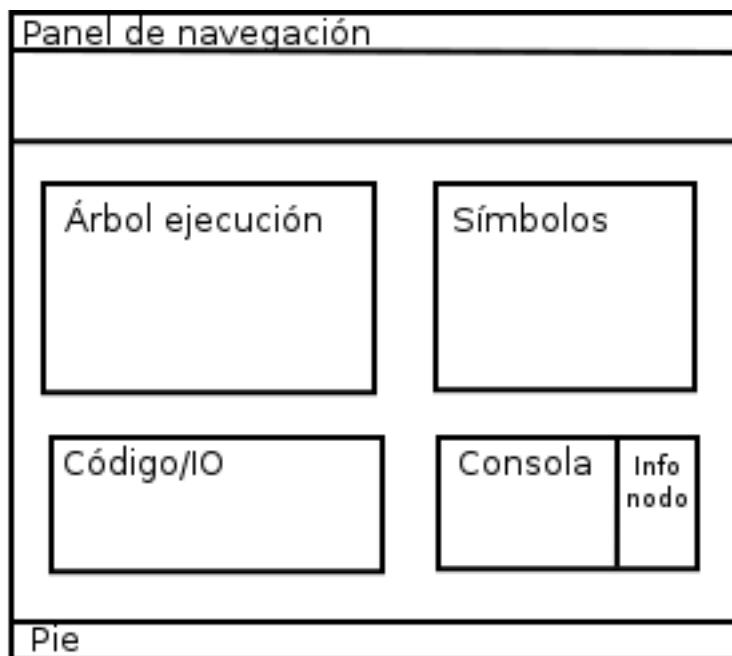


Figura 5.163: Interfaz de usuario runTree

5.6.3. Sitio web

El proyecto OMI incluye un sitio web que sirve como presentación del mismo, además de como medio de acceso a la documentación y el software desarrollado. Todas las páginas web pertenecientes al sitio contienen información relativa al proyecto y a las áreas que este ocupa.

El sitio web OMI se compone de:

Página de inicio: Describe e introduce brevemente el proyecto. Contiene enlaces a las demás secciones del sitio web. Además presenta un listado de noticias y enlaces de descargas a la última versión del intérprete.

Índice de documentación: Página que representa un índice de los documentos que conforman el proyecto.

Documentos: Páginas relativas a la documentación del proyecto en si.

Navegador de clases: Páginas relativas a la documentación de las clases incluidas en la biblioteca.

Navegador de ficheros: Páginas relativas a la documentación de los ficheros que conforman el código fuente de la biblioteca y el intérprete.

Navegador de gramática: Páginas relativas a la documentación gráfica de la gramática del lenguaje.

Descargas: Página que enlaza la descarga de las distintas versiones del software que conforma el proyecto, disponibles en varios formatos de instalación.

Sobre OMI: Página con información relativa a la motivación y circunstancias en las que se ha dado el proyecto. Además da detalles sobre los autores y los organismos implicados en el desarrollo del mismo.

Contacto: Página con información de contacto.

5.6.3.1. Wireframe

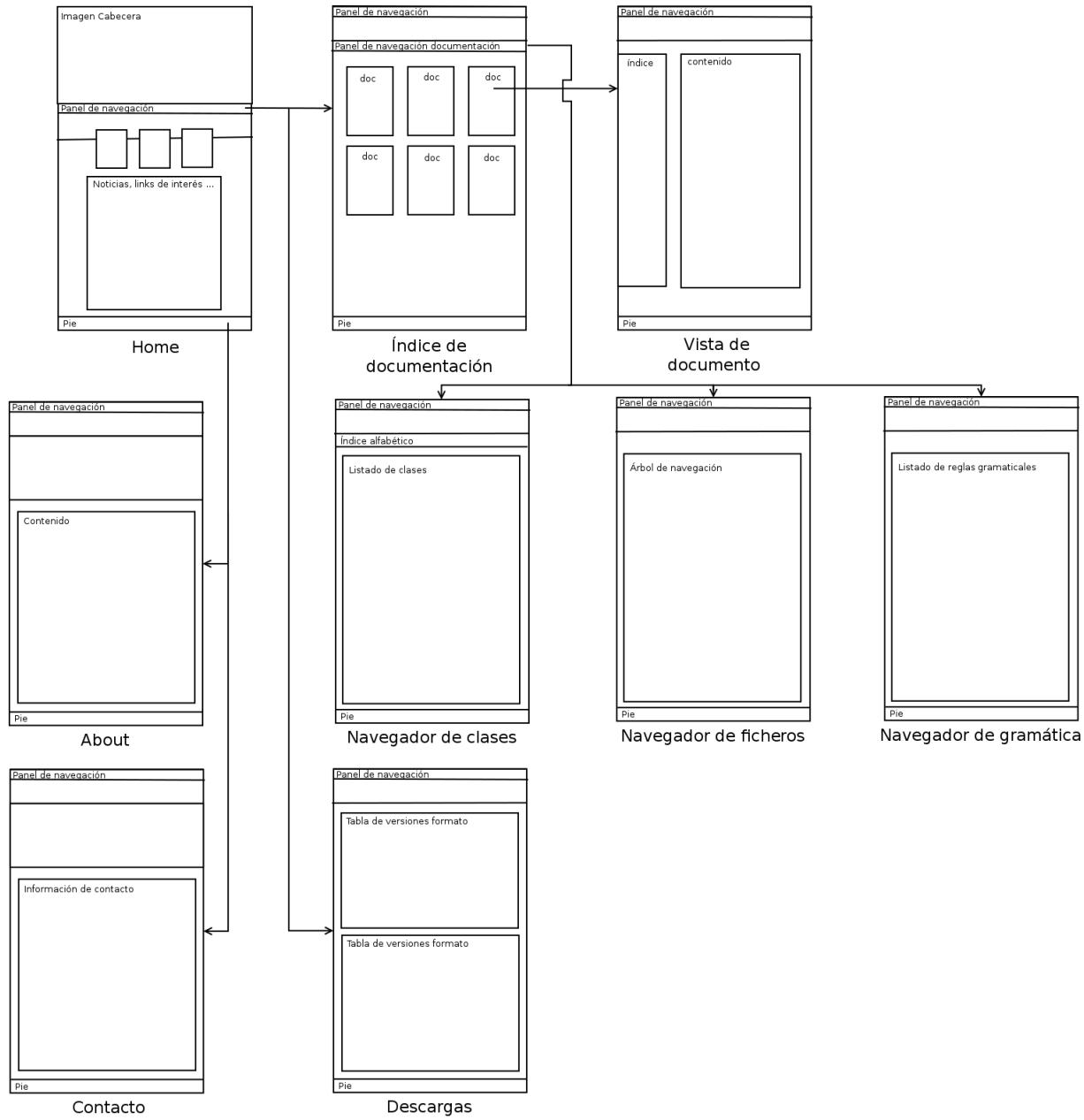


Figura 5.164: Interfaz de usuario web

Capítulo 6

Diseño del Sistema

En esta sección se recoge la arquitectura general del sistema de información, la parametrización del software base (opcional), el diseño físico de datos, el diseño detallado de componentes software y el diseño detallado de la interfaz de usuario.

6.1. Arquitectura del sistema

6.1.1. Arquitectura física

Las herramientas del proyecto OMI pueden ser dispuestas sobre diferentes arquitecturas físicas. Por un lado se puede hacer uso del intérprete como una herramienta de un sistema operativo GNU/Linux, siguiendo así una arquitectura de escritorio. Por otro se puede usar el sistema como una arquitectura cliente/servidor.

6.1.1.1. Escritorio

El cliente OMI puede ser utilizado en una arquitectura de escritorio, funcionando como un comando del sistema. Para ello solo se precisa de un PC con un sistema operativo GNU/Linux.

6.1.1.2. Cliente/servidor

El intérprete OMI puede ser usado en una arquitectura cliente/servidor, en la que hace de servidor. El servidor OMI espera una petición para un proceso de interpretación. Esta petición contendrá el código fuente que se desea interpretar.

El servidor OMI procesa un código fuente para su interpretación devolviendo una estructura de datos en formato JSON que representa el árbol de nodos resultado del análisis léxico y sintáctico.

Luego espera peticiones para resolver cada paso dentro del proceso de interpretación y devolver una estructura de datos en el mismo formato que representa qué ha ocurrido mediante el estado actual.

El sistema servidor es independiente del cliente en el sentido de que pueden usarse distintos clientes para el mismo servicio. El proyecto OMI presenta un cliente web llamado runTree que funciona en cualquier navegador moderno, no obstante se puede usar otra tecnología como cliente.

Para ejecutar el sistema en una arquitectura cliente servidor se precisa de un equipo que tenga conexión a internet con un navegador web que hará de cliente. Por otro lado se necesita de un equipo servidor que presente una alta disponibilidad. Este equipo dispondrá de un sistema GNU/Linux sobre el que se ejecutará el intérprete y un servidor web Apache que servirá de plataforma para dar el servicio web.

6.1.2. Arquitectura lógica

En esta sección se presenta la arquitectura del sistema. Se presentan dos niveles de organización la parte frontal (front-end) y la parte interna o motor (back-end).

La parte frontal se encarga de analizar y procesar la entrada del usuario. Una vez tratada la entrada, se pasa la ejecución a las estructuras que conforman el motor del sistema. Estas llevarán a cabo una serie de tareas que pueden producir un resultado que será enviado al frontal. La parte frontal mostrará los datos como salida del programa en un formato establecido.

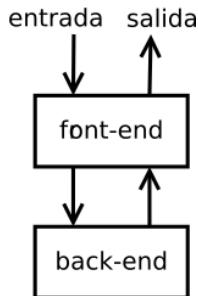


Figura 6.1: Arquitectura lógica

Aunque bien se podría haber especificado otra capa correspondiente a la gestión de datos llevada a cabo por las tablas de símbolos, esta se ha incluido en la parte del motor. Los datos que las tablas de símbolos tratan no mantienen persistencia, están muy ligados al proceso interno de interpretación y no representan los datos correspondientes al modelo de datos, sino que sirve para referenciar componentes de la capa del motor de la aplicación.

6.1.2.1. Frontal (front-end)

El frontal del sistema se compone de un analizador léxico y un analizador sintáctico. Aunque es posible especificar reglas léxicas mediante reglas gramaticales, se ha separado el estos dos

componentes por los siguientes motivos:

- La reglas lexicográficas suelen ser muy simples y para su descripción no se precisa de una notación tan compleja como las gramaticales.
- Las expresiones regulares son más concisas y fáciles de entender que las gramáticas
- A partir de expresiones regulares se puede describir analizadores léxicos eficientes.
- Se modulariza el proceso en dos componentes con objetivos bien definidos.

Además del analizador léxico y sintáctico el front-end contiene estructuras funcionales correspondientes a la toma de datos de entradas y a la impresión de la salida.

6.1.2.1.1 Analizador léxico

El analizador léxico implementa una gramática de tipo 3 en la jerarquía de Chomsky (referencia: Compiladores y procesadores del lenguaje [José Antonio Jiménez Millán]), estas se corresponden con las gramáticas regulares.

Las gramáticas regulares, también llamadas gramáticas finitas, se pueden describir mediante expresiones regulares. Cada expresión regular denota un lenguaje $L(r)$. Una expresión regular se construye a partir de otras expresiones más simples, utilizando un conjunto definidos de reglas. Estas reglas indican la manera de conseguir el conjunto $L(r)$ combinando de varias formas los lenguajes denotados por las subexpresiones de r . Las expresiones regulares se pueden definir de una forma recursiva como sigue:

1. ϵ es una expresión regular que denota el lenguaje que únicamente contiene la cadena vacía.
2. Si a fuese un símbolo del alfabeto, entonces a es una expresión regular que denota el lenguaje formado por la cadena a .
3. Si r y s son dos expresiones regulares que denotan los lenguajes $L(r)$ y $L(s)$ respectivamente entonces:

- $L(r|s) \rightarrow L(r) \cup L(s)$
- $L(rs) \rightarrow L(r)L(s)$
- $L(r*) \rightarrow (L(r))*$
- $L(r+) \rightarrow (L(r))+$
- $L(r?) \rightarrow \{\epsilon\} \cup L(r)$
- $L((r)) \rightarrow (L(r))$

Las gramáticas regulares pueden ser implementadas mediante un autómata finito. Un autómata finito se define formalmente como sigue:

$$AF = (S, \Sigma, \delta, S_0, F)$$

De forma que:

- S es un conjunto de estados.
- Σ es el conjunto de símbolos que conforman el alfabeto.
- δ es una función de transición tal que $(\delta : S \times \Sigma \rightarrow S)$. Donde para cada estado y según cada carácter de entrada del alfabeto le hace corresponder un nuevo estado.
- S_0 es el estado inicial.
- F es un conjunto de estado finales.

El analizador léxico se corresponde con un autómata finito donde, al alcanzarse un estado final, se produce la generación de un token. Un token es un elemento léxico con cierto valor para el lenguaje. Normalmente se corresponde con una cadena de caracteres que se puede corresponder con una palabra reservada, un identificador, un número... Un token puede contener un valor. Los token son utilizados por el analizador sintáctico para llevar a cabo el procesamiento del código fuente.

Las reglas lexicográficas a partir de las cuales se construye el analizador léxico son descritas junto la gramática, haciendo uso de expresiones regulares y diagramas sintácticos.

6.1.2.1.2 Analizador sintáctico

El analizador sintáctico implementa una gramática de tipo 2 en la jerarquía de Chomsky, esta se corresponden con las gramáticas independientes del contexto. Una gramática de este tipo, como su propio nombre indica, no depende del contexto para su resolución, lo que origina que los recursos que permiten analizarlas sean relativamente eficientes y simples. Una gramática de este tipo pueden ser analizadas mediante algoritmos con un orden $O(n^3)$ donde n es el tamaño de la entrada. No obstante existen un subconjuntos de este tipo de gramáticas como las $LL(k)$ y $LR(k)$ que pueden ser analizados en tiempo $O(n)$. Estos últimos son los comúnmente utilizados en los lenguajes de programación.

En el análisis sintáctico se pueden utilizar métodos descendentes o ascendentes, en función de cómo se genera el árbol sintáctico.

Los métodos de análisis descendentes tienen como objetivo construir el árbol de derivación desde la raíz hacia las hojas. Normalmente se utilizan analizadores del tipo LL dado que la entrada es tratada desde la izquierda y las reglas de producción también desde la izquierda. Estos tipos de analizadores tienen el problema de la incertidumbre, debido a que dado un símbolo no terminal debe determinar qué regla de derivación aplicar. La incertidumbre puede ser reducida mediante métodos como el retroceso o la predicción.

Los métodos de análisis de ascendentes pretenden construir el árbol de derivación o de sintaxis desde las hojas hacia la raíz. Para este tipo normalmente se utiliza analizadores del tipo LR dado que la entrada es tratada desde la izquierda, mientras que las reglas de producción son tratadas

desde la derecha. Este tipo de analizadores reducen la incertidumbre dado que se parte de una regla de derivación para obtener el símbolo no terminal que se corresponde a la misma.

Para el lenguaje tratado en este proyecto se tomará un analizador sintáctico ascendente LR , dado que es uno de los más utilizados, son eficientes y existen herramientas que permiten su generación automática a partir de la descripción de la gramática.

Las gramáticas libres de contexto son analizadas mediante un autómata de pila. La configuración en un momento dado del analizador se corresponde con el contenido de la pila y el resto de la entrada que aún está por analizar. La entrada estará constituida por tokens obtenidos por el analizador léxico, teóricamente estos serán almacenados en una estructura de buffer, pero en la práctica estos son generados bajo demanda por el analizador léxico. En la pila se irá almacenando una serie de estados en función del procesamiento realizado sobre la cadena de entrada. El autómata puede llevar a cabo dos operaciones:

Desplazar: Consiste en sacar del buffer de entrada un símbolo terminal e introducir el estado correspondiente en la pila.

Reducir: Consiste en reducir n estados del tope de la pila al estado correspondiente, según las reglas de producción.

El análisis finaliza cuando se produce un estado de aceptación o de error.

En la práctica este tipo de analizador sintáctico se implementa mediante un programa monitor que encierra la lógica descrita, un buffer de entrada en el que se almacenan los tokens, una pila que almacena los estados producidos, y dos tablas. La primera tabla guardará las acciones y determina, para el estado actual y el primer símbolo del buffer, la acción a realizar (desplazar o reducir). La otra tabla, la tabla de saltos, es utilizada tras llevarse a cabo una de reducción, y determina a partir del estado en el tope de la pila tras la reducción y la regla que se utilizó para la misma, el siguiente estado a introducir en la pila.

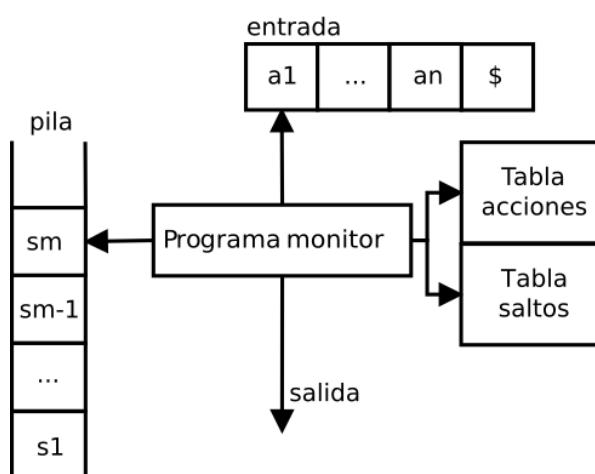


Figura 6.2: Arquitectura analizador sintáctico

Por cada reducción que se lleve a cabo se creará un nodo del árbol sintáctico. De esta forma el árbol es creado desde las hojas hacia la raíz. Cada nodo encierra un significado semántico que será llevado a cabo cuando se ejecuten. Estos nodos constituyen a su vez el motor, o back-end, del sistema.

6.1.2.1.3 Mecanismos de entrada/salida

En el nivel de front-end también se dan los diferentes componentes para capturar la entrada del usuario, así como para facilitar la salida que produce en sistema.

La entrada más común supondrá código fuente que será interpretado y ejecutado. Este se podrá dar de varias formas:

- Mediante la entrada estándar del sistema.
- Contenido en fichero.
- Mediante un intérprete de línea.

Es posible que se den datos de entrada que no sean código fuente, como por ejemplo, una extensión que debe ser cargada, la opción de mostrar ayuda ...

Por otro lado los datos de salida que produce el programa normalmente serán volcados en la salida estándar sistema en forma de cadenas de caracteres. Esta será el resultado visual de la interpretación del código fuente, aunque es posible que un determinado código fuente no produzca salida de este tipo.

El programa debe ser capaz de procesar los errores y mostrarlos por la salida de errores del sistema. Para ello debe controlar las líneas y ficheros procesados en cada momento. Los errores pueden ser de diferente tipos:

- Errores léxicos.
- Errores sintácticos.
- Errores semánticos.

Además los errores pueden presentar diferente grado:

Avisos: Informan de que algún recurso del lenguaje no se utiliza de la forma habitual, y aunque no se produce un error como tal, puede suponer un error de codificación.

Normales: No finalizan la ejecución del código fuente pero provocan que la sentencia en la que sucede no se pueda ejecutar.

Críticos: Finalizan la ejecución del código fuente.

6.1.2.2. Motor (back-end)

El motor del sistema lo conforman los nodos creados mediante el análisis sintáctico. Estos nodos se denominan nodos ejecutables, dado que su procesamiento conlleva la realización operacional de la semántica que encierran. Los nodos ejecutables se categorizan según el tipo de operaciones que conlleva su ejecución, así existen nodos aritméticos, lógicos, de control de flujo, de definiciones...

El sistema presenta una jerarquía de nodos ejecutables que establece la naturaleza de los mismos, dotándolos de características de una forma general y que serán concretadas en los niveles más bajos de la jerarquía. Se presenta de esta forma, nodos ejecutable genéricos como expresiones, referencias, imprimibles....

El procesamiento de un nodo ejecutable, que fue creado a partir del análisis sintáctico, puede originar la creación de otros nodos ejecutables, a estos últimos se les denominará nodos ejecutables dinámicos. Un nodo dinámico es creado y referenciado por una estructura de datos denominada tabla de símbolos. Esta estructura también es considerada un componente del motor del sistema.

Los nodos ejecutables presentan dos niveles de procesamiento semántico: la inicialización y la ejecución. Además también se ha de controlar cómo estos son creados y eliminados de forma dinámica en la tabla de símbolos.

6.1.2.2.1 Inicialización

Todo nodo ejecutable se inicializa a partir de otros nodos ejecutables sobre los que operará. Normalmente estos nodos se corresponderán con los hijos del nodo en cuestión en el árbol sintáctico.

La inicialización de un nodo ejecutable se lleva a cabo durante la creación del árbol sintáctico, por tanto los nodos se inicializan de una forma ascendente, es decir, antes los nodos hijos que los padres.

Durante la inicialización se lleva a cabo algunas comprobaciones de tipos, asignaciones y otras operaciones. No obstante, todo proceso que se lleve a cabo durante la inicialización debe ser independiente de la ejecución. Se ha de tener en cuenta que un nodo ejecutable sólo se inicializa una vez, y sin embargo se puede ejecutar varias veces, siendo esta última dependiente del estado en un momento dado. Se puede decir pues que la inicialización tiene un carácter estático.

6.1.2.2.2 Ejecución

Una vez generado el árbol sintáctico, lo que conlleva la inicialización de los nodos ejecutables que lo componen, se procede a la ejecución del mismo. La ejecución comienza desde el nodo raíz hacia las hojas, se hace pues un recorrido del árbol en profundidad.

La ejecución de un nodo es iniciada normalmente desde su nodo padre, y generalmente conlleva la ejecución de sus nodos hijos antes de proceder a su propia ejecución. Así por ejemplo la ejecución de un nodo suma conllevará el cálculo del lado izquierdo de la expresión, lo que se corresponde

con la ejecución uno de sus nodos hijo, luego se calculará el lado derecho, correspondiente a la ejecución del otro nodo hijo, para, una vez ejecutados sus hijos (los operandos), proceder al cálculo de la suma, lo que se corresponde con su propia ejecución.

Durante la ejecución de un nodo ejecutable lo normal es que primero se resuelvan los nodos referencias, aquellos que se corresponden con nodos dinámicos de tipos no definidos. Estos nodos referencian a otros nodos creados dinámicamente en la interpretación y que se encuentran accesibles desde la tabla de símbolos. Algunos nodos referencias, también denominados nodos de expresiones no definidas, son los correspondientes a: variables, funciones u clases de objetos. Una vez se ha obtenido el nodo al que estos referencian, se procede con la ejecución como si los nodos obtenidos fueran los hijos del nodo en proceso.

Aunque la ejecución del árbol se hace inicialmente con un recorrido en profundidad existen nodos que rompen esta secuencia, haciendo que la ejecución pase a nodos específicos o repitiendo la ejecución de varios de sus hijos. Cabe decir que una vez se salte la secuencia de ejecución el recorrido seguirá siendo en profundidad. Estos nodos se corresponden con sentencias de control de flujo, llamadas a funciones, etc.

El resultado de ejecutar un nodo no tiene por que ser estático, dependiendo en gran medida del contexto, es decir, del estado de la tabla de símbolos. Se puede decir pues que la ejecución de un nodo tiene un carácter dinámico dependiente de la tabla de símbolos.

6.1.2.2.3 Tabla de símbolos

La tabla de símbolos es una estructura de datos que almacena referencias a nodos creados dinámicamente en el proceso de interpretación. Estos nodos serán creados en la ejecución del árbol de sintáctico y serán accedidos desde la ejecución de determinados nodos en la resolución de referencias.

Las tablas de símbolos se componen de pares donde el primer elemento es un identificador que nomina al símbolo y el segundo elemento es una referencia al nodo ejecutable.

Aunque en una tabla de símbolos es posible guardar referencias a cualquier nodo se mantiene diferentes tipos tablas de símbolos según la naturaleza de los nodos a los que se referencian. Esto hace posible que se puedan repetir identificadores para distintos tipos de nodos como variables, funciones o clases.

Dos identificadores que pueden estar en la misma tabla de símbolos o distintas, pueden referenciar al mismo nodo. El propio nodo debe ser capaz de guardar cuantas referencias al mismo tiene. Esto se hace por criterios de optimización, para la reutilización de datos y para realizar una eliminación del nodo de una forma segura cuando sea necesario.

Las tablas de símbolos pueden quedar organizadas en niveles, aunque únicamente un nivel, el actual en un momento dado, es el que se utilizará para la resolución de referencias. Esto hace posible definir diferentes ámbito para el acceso a determinados símbolos. Los niveles quedan dispuestos en una estructura de pila, de forma que el nivel actual es la cima de la pila. Una vez un nivel es eliminado de la pila se eliminan todas las referencias del mismo y el nivel anterior

queda en la cima y por tanto es considerado el nivel actual.

Existen nodos que presentan sus propias tablas de símbolos, esto hace que muchas definiciones sean internas al mismo. Por ejemplo las clases de objetos mantienen sus propias tablas de símbolos para referencias los métodos y atributos, y los arrays presentan una tabla de símbolos para guardar las claves y sus valores. Es común que en la ejecución de uno de estos nodos se sustituya la tabla de símbolos en uso por la del nodo en cuestión.

La tabla de símbolo es una estructura de datos que forma parte del motor de la aplicación dado que gestiona y almacena datos presentes en este nivel.

6.1.2.2.4 Eliminación de referencias

Los nodos creados de forma dinámica por el proceso de interpretación deben ser eliminados cuando ya no se precisen de ellos. Por ejemplo cuando se produce una asignación destructiva. Como los nodos creados de forma dinámica son reutilizados y pueden ser referenciados por más de un símbolo se ha de controlar que no exista ninguna referencia al mismo antes de proceder a su eliminación. Así cada nodo guarda el número de referencias al mismo.

Los nodos correspondientes al árbol sintáctico nunca serán eliminados durante la ejecución y no será hasta la finalización del programa hasta que se proceda a la liberación de los recursos que consumen.

6.1.2.3. runTree

runTree es un sistema que hace de cliente del intérprete OMI, cuando este funciona como servidor. Se encarga de llevar a cabo peticiones para navegar por el proceso de interpretación de un código fuente dado.

El cliente runTree presenta una arquitectura interna en dos capas al igual que el intérprete. El backend de este se comunica con el frontend del intérprete cuando funciona de servidor

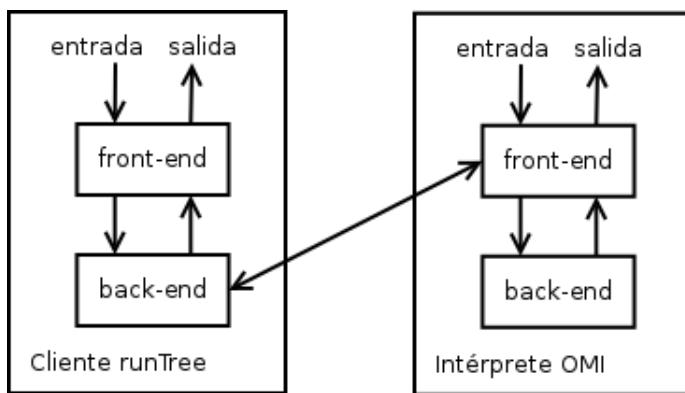


Figura 6.3: Arquitectura runTree

El frontend del cliente runTree lo componen elementos relacionados con la interfaz del usuario. Captura eventos de esta y lo envía al backend para su procesamiento, a su vez representa los datos que recibe del backend.

El backend del cliente runTree lo componen elementos que guardan el estado interno del proceso de interpretación. Normalmente estos elementos son inicializados mediante datos que recibe del servidor como fruto de una petición.

6.2. Diseño de la gramática

En esta sección se presenta la gramática del lenguaje, para ello se procede a una descripción de las reglas gramaticales mediante el lenguaje EBNF (Extended Backus–Naur Form) usado para expresar gramáticas libres de contexto. Además cada regla se acompaña de un diagrama sintáctico o diagrama de carril.

Una gramática G se define formalmente como sigue:

$$G = (V_t, V_n, P, S)$$

De forma que:

V_t es un conjunto finito de símbolos terminales

V_n es un conjunto finito de símbolos no terminales

P es un conjunto finito de reglas de producción

$S \in V_n$ es el símbolo inicial

Las reglas de producción de una gramática libre de contexto tiene la forma siguiente forma:

$$V_n \rightarrow (V_t \cup V_n)^*$$

La gramática libre de contexto abordada tiene como símbolo inicial el no terminal *program*. Se comienza pues describiendo las reglas de producción relacionadas con este símbolo, siguiendo con las reglas de producción derivadas de esta.

Las reglas de producción se organizan en niveles como sigue:

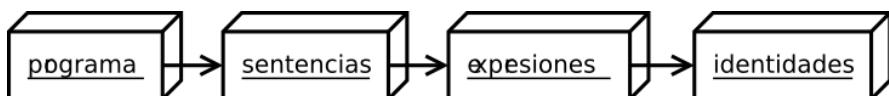


Figura 6.4: Niveles reglas de producción

Estos niveles son dados a partir del nivel de abstracción del significado semántico que encierran las reglas de producción contenidas en los mismos.

Las reglas de producción correspondientes al nivel de programa son las más genéricas y se valen de las de la siguiente nivel para su definición. Estas definen el programa como una secuencia de sentencias. La gramática descrita contempla el programa vacío, es decir, el programa que no contiene ninguna sentencia.

Las reglas de producción del nivel de sentencia se definen a partir de expresiones o de otras sentencias. Una sentencia contiene un significado semántico operativo de valor para el programa. Cabe decir que una expresión por si sola puede constituir una sentencia. La gramática expuesta describe la sentencia vacía, esta es una sentencia que no tienen ningún significado semántico.

Las expresiones son la unidad mínima con significado semántico atribuido por el lenguaje. La mayoría de expresiones se definen a partir de identidades, sin embargo algunos tipos de expresiones, como las funciones, pueden formarse a partir de reglas de más alto nivel de abstracción semántica. Por otro lado las reglas de producción correspondiente a las expresiones están organizadas en niveles según la prioridad atribuida en su resolución.

Las identidades son reglas de producción atómicas, componiéndose únicamente de símbolos no terminales. Tienen un significado semántico asociado de forma directa. Normalmente este valor viene dado por el análisis léxico.

6.2.1. Programa

```
program ::= stmts
          | empty
```

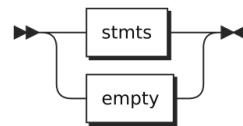


Figura 6.5: Gramática programa

6.2.2. Sentencias

6.2.2.1. Secuencia de sentencias

```
stmts ::= stmt ";" stmts
        | stmt ";" ?
        | stmtb
        | label stmts
        | ";"
```

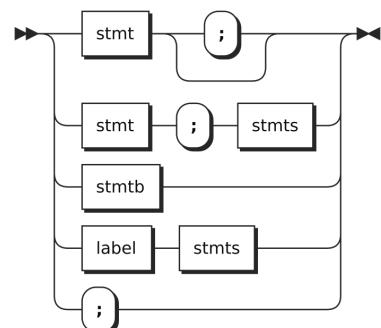


Figura 6.6: Gramática sentencias

6.2.2.2. Sentencia de bloque

```
stmtb ::= if
      | while
      | dowhile
      | for
      | foreach
      | break
      | switch
      | iloop
      | trycatch
      | class
      | with
```

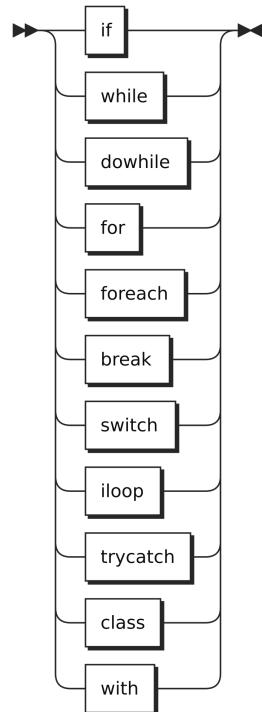


Figura 6.7: Gramática sentencias de bloque

6.2.2.2.1 Sentencia de control: if

```
if ::= "if" exp ("{" stmts? "}" | stmt ";" | stmtb) elif* ("else" ("{" stmts? "}" | stmt ";" | stmtb))?
elif ::= "elif" exp ("{" stmts? "}" | stmt ";" | stmtb)
```

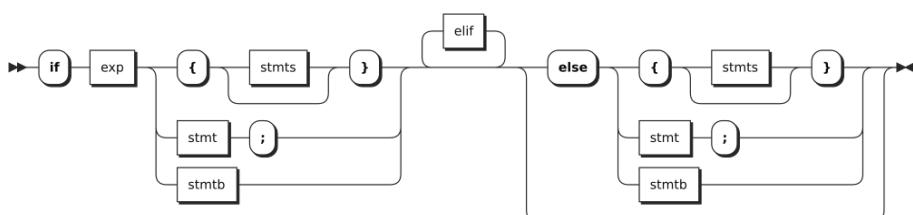


Figura 6.8: Gramática if

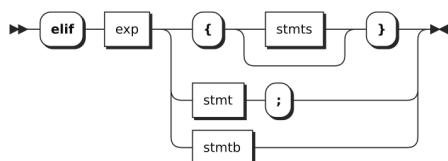


Figura 6.9: Gramática elif

6.2.2.2.2 Sentencia de control: while

```
while ::= "while" exp ("{" stmts? "}" | stmt ";" | stmtb)
```

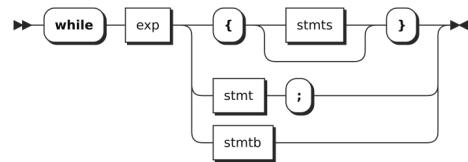


Figura 6.10: Gramática while

6.2.2.2.3 Sentencia de control: do...while

```
dowhile ::= "do" ( "{" stmts? "}" | stmt ";" | stmtb ) "while" exp ";"
```

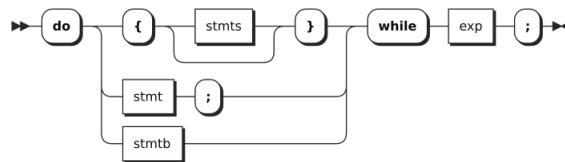


Figura 6.11: Gramática dowhile

6.2.2.2.4 Sentencia de control: for

```
for ::= "for" "("? exp? ";" exp? ";" exp? ")"
( "{" stmts? "}" | stmt ";" | stmtb )
```

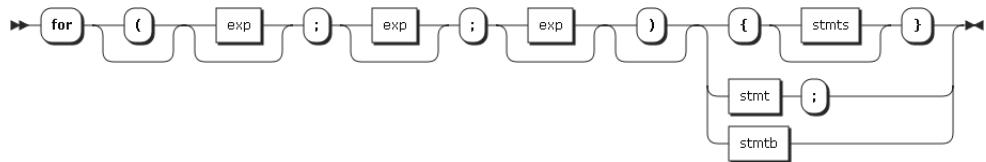


Figura 6.12: Gramática for

6.2.2.2.5 Sentencia de control: foreach

```
foreach ::= "for" "("? id (";" id)? "in" exp | exp "as" id (";" id)?)?"? ( "{" stmts? "}" | stmt ";" | stmtb )
```

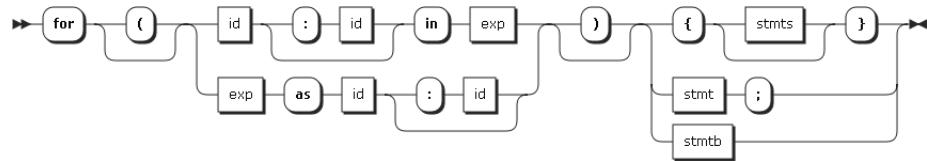


Figura 6.13: Gramática foreach

6.2.2.2.6 Sentencia de control: switch

```
switch ::= "switch" "(" exp ")" "{" cases? "}"  
  
cases ::= "case" exp ":" ( stmts cases | stmts | cases )  
| "default" ":" stmts
```

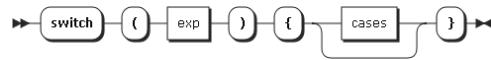


Figura 6.14: Gramática switch

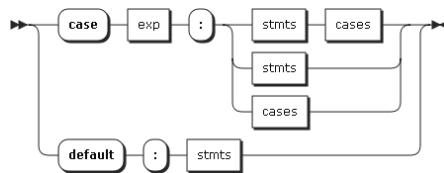


Figura 6.15: Gramática cases

6.2.2.2.7 Sentencia de control: iloop

```
iloop ::= "$" "(" exp ( "as" id (":" id)? )? ( "," exp )? ")" ( "{" stmts? "}" | stmt ";" | stmtb )  
iloop_access ::= "$"  
| "$" "{" NUM "}"
```

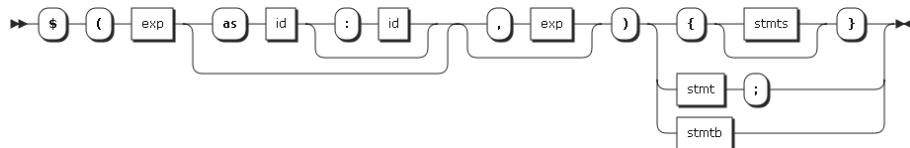


Figura 6.16: Gramática iloop

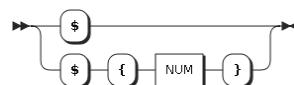


Figura 6.17: Gramática acceso iloop

6.2.2.2.8 Sentencia de control: try...catch

```
trycatch ::= "try" ( "{" stmts? "}" | stmt ";" | stmtb ) catch "(" id ")" ( "{" stmts? "}" | stmt ";" | stmtb )
```

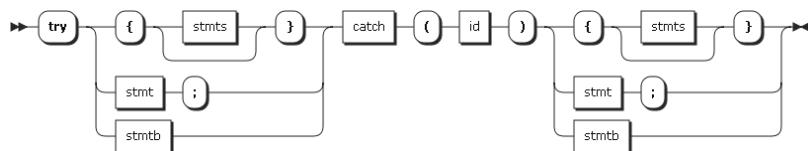


Figura 6.18: Gramática trycatch

6.2.2.2.9 Sentencia de control: with

```
with ::= "with" exp ( "{" stmts? "}" | stmt ";" | stmtb )
```

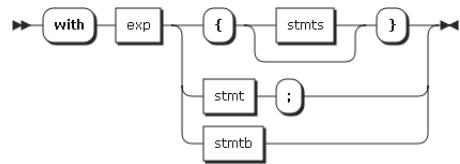


Figura 6.19: Gramática with

6.2.2.3. Sentencia simple

```
stmt ::= exp
      | "<<" exp
      | ">>" ("[" exp "]")? id
      | "goto" exp
      | "include" exp
      | "return" exp?
      | "sleep" exp
      | "load" exp
      | "typeof" id
      | "datinfo" exp
      | "exit"
      | "throw" exp
      | "global" identity
      | "break" num? ";"
      | "continue" num? ";"
```

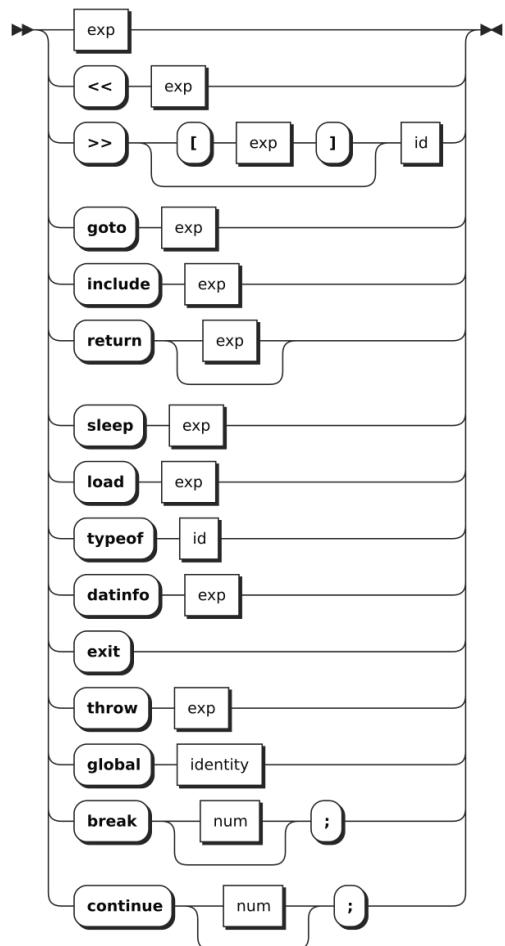


Figura 6.20: Gramática stmt

6.2.2.4. Etiquetas

```
label ::= id ":"
```

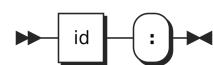


Figura 6.21: Gramática etiquetas

6.2.2.5. Nombres de espacios

```
namespace ::= namespace ":" id
          | id ":" id
          | "parent" ":" id
          | "static" ":" id
```

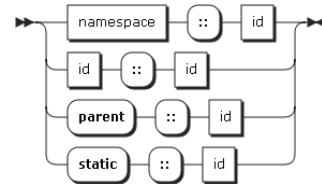


Figura 6.22: Gramática nombres de espacios

6.2.2.6. Clases

```
class ::= "class" id ("extends" id )? "{" class_stmts? "}"
```

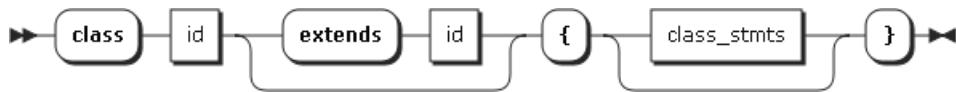


Figura 6.23: Gramática clases

6.2.2.6.1 Métodos y atributos

```
class_stmts ::= ("static"|"private")? ("static"|"private"|"function"|"assig")?
```

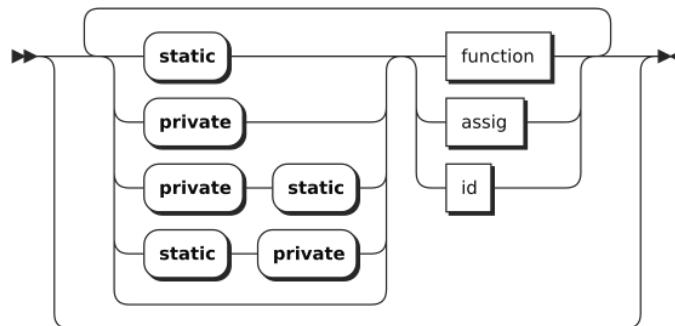


Figura 6.24: Gramática métodos y atributos

6.2.3. Expresiones

```
exp ::= op1
```



Figura 6.25: Gramática expresiones

6.2.3.1. Operadores lógicos

6.2.3.1.1 Or lógico

```
op1 ::= op1 ( "||" | "or" ) op2
      | op2
```

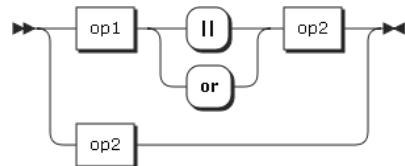


Figura 6.26: Gramática or lógico

6.2.3.1.2 And lógico

```
op2 ::= op2 ( "&&" | "and" ) op3
      | op3
```

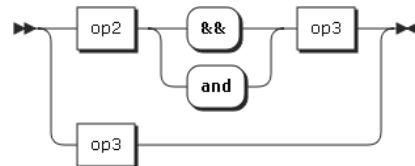


Figura 6.27: Gramática and lógico

6.2.3.1.3 Negación lógica

```
op3 ::= "!" op3
      | op4
```

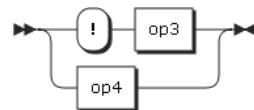


Figura 6.28: Gramática negación lógica

6.2.3.1.4 Comparaciones

```
op4 ::= op4 "<" op5
      | op4 "<=" op5
      | op4 ">" op5
      | op4 ">=" op5
      | op4 "==" op5
      | op4 "!=" op5
      | op4 "====" op5
      | op4 "!===" op5
      | op5
```

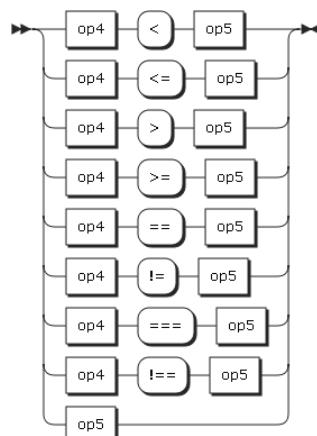


Figura 6.29: Gramática comparaciones

6.2.3.2. Operadores aritméticos

6.2.3.2.1 Suma y diferencia

```
op5 ::= op5 "+" op6
      | op5 "-" op6
      | op6
```

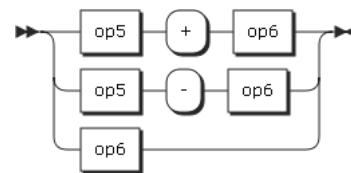


Figura 6.30: Gramática operadores aritméticos

6.2.3.2.2 Producto y división

```
op6 ::= op6 "*" op7
      | op6 "/" op7
      | op7
```

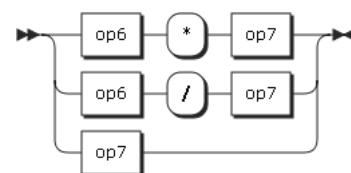


Figura 6.31: Gramática producto y división

6.2.3.2.3 Potencia y módulo

```
op7 ::= op7 "^" op8
      | op7 "%" op8
      | op8
```

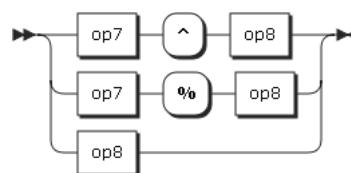


Figura 6.32: Gramática potencia y módulo

6.2.3.3. Operadores cadenas de caracteres

6.2.3.3.1 Concatenación

```
op8 ::= op8 "." op9
      | op9
```

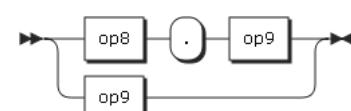


Figura 6.33: Gramática operadores cadenas de caracteres

6.2.3.3.2 Flujo

```
op9 ::= call "<<" op9
      | call
```

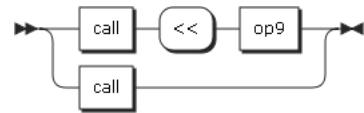


Figura 6.34: Gramática flujo

6.2.3.4. Llamadas

```
call ::= call "(" list? ")"
      | opc
```

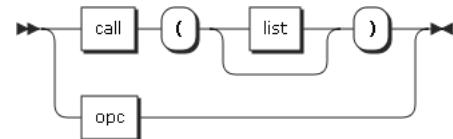


Figura 6.35: Gramática llamadas

6.2.3.5. Operadores condicionales

```
opc ::= tern
      | nullcoalescing
      | unity
```

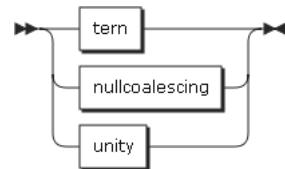


Figura 6.36: Gramática operadores condicionales

6.2.3.5.1 Operador ternario

```
tern ::= exp "?" exp? ":" exp
      | exp "?" exp
```

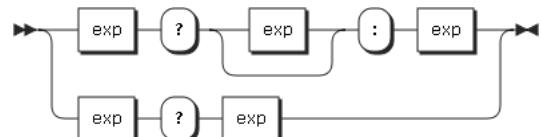


Figura 6.37: Gramática operador ternario

6.2.3.5.2 Null coalescing

```
nullcoalescing ::= "[[" list "]]"
```



Figura 6.38: Gramática fusión de nulos

6.2.3.6. Operadores unitarios

```
unity ::= inc_dec
      | assignation_exp
      | cast
      | logical_func
      | arith_func
      | array_func
      | string_func
      | regexp_func
      | iloop_access
      | class_exp
      | func_exp
      | file
      | date
      | process
      | generator
      | environments
      | array
      | identity
```

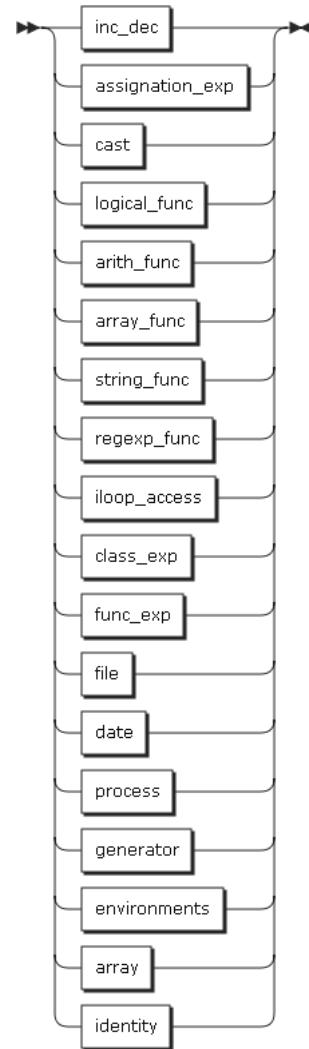


Figura 6.39: Gramática operadores unitarios

6.2.3.6.1 Incrementos y decrementos

```
inc_dec ::= "++" exp
      | exp "++"
      | "--" exp
      | exp "--"
```

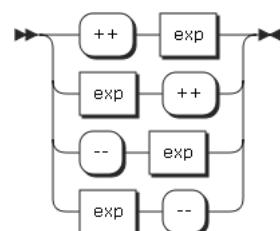


Figura 6.40: Gramática incrementos y decrementos

6.2.3.6.2 Conversión de tipos

```
cast ::= "int" exp
      | "float" exp
      | "bool" exp
      | "str" exp
```

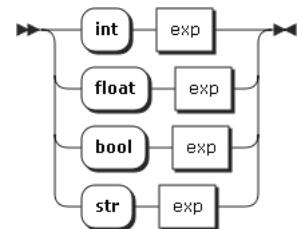


Figura 6.41: Gramática conversión de tipos

6.2.3.6.3 Accesos

```
access ::= access "->" id
       | access "[" exp? "]"
       | call "->" id
       | call "[" exp? "]"
```

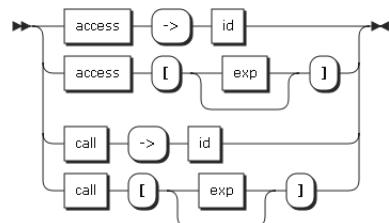


Figura 6.42: Gramática accesos

6.2.3.7. Asignaciones

```
assignment ::= (id | assignment | access) "=" "&"? exp
```

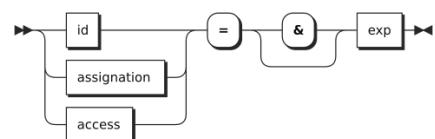


Figura 6.43: Gramática asignaciones

6.2.3.8. Funciones

```
function ::= "~" id "(" params_val? ")" "{" stmts? "}"
```



Figura 6.44: Gramática funciones

6.2.3.8.1 Función lambda

```
function_lambda ::= "~" "(" params_val? ")" "{" stmts? "}"
| "~" params_val ":" exp
| "~&" id
```

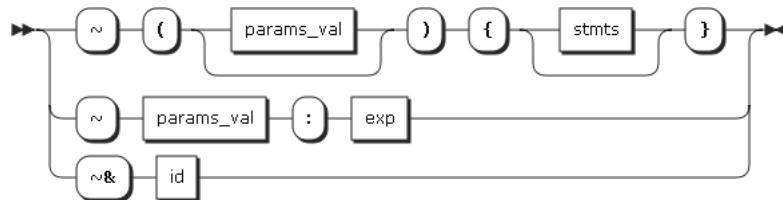


Figura 6.45: Gramática función lambda

6.2.3.8.2 Cálculo parcial

```
function_partial ::= "P" "[" params_val "]" "(" id ")"
| "P" "[" params_val "]" "(" function_exp ")"
| "P" "[" params_val "]" "(" arrayexp ")"
```

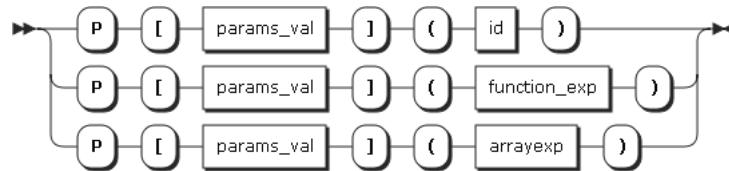


Figura 6.46: Gramática función parcial

6.2.3.8.3 Función de contexto

```
function_context ::= "~>"
```



Figura 6.47: Gramática función de contexto

6.2.3.9. Decoradores

```
decorator ::= "~~" id "(" params_val? ")" "{" stmts? "}"
```

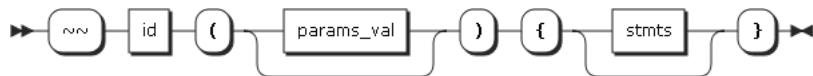


Figura 6.48: Gramática decoradores

6.2.3.9.1 Decorador lambda

```
decorator_lambda ::= "~~" "(" params_val? ")" "{" stmts? "}"
```



Figura 6.49: Gramática decorador lambda

6.2.3.10. Operadores clases y objetos

```
class_exp ::= "new" id ("(" list? ")")?
           | "this"
           | "parent"
           | namespace
```

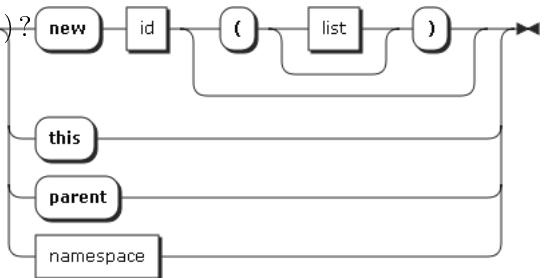


Figura 6.50: Gramática operadores clases y objetos

6.2.3.11. Funciones del lenguaje

6.2.3.11.1 Funciones lógicas

```
logical_func ::= "isnull" exp
               | "empty" exp
```

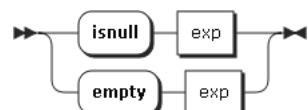


Figura 6.51: Gramática funciones del lenguaje

6.2.3.11.2 Funciones aritméticas

```
arith_func ::= "size" exp
              | "sizeof" exp
```

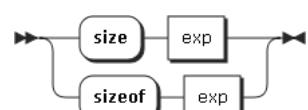


Figura 6.52: Gramática funciones aritméticas

6.2.3.11.3 Funciones cadenas de caracteres

```
string_func ::= "sprintf" "(" exp "," exp ")"
| "str_replace" "(" exp "," exp "," exp (" , " exp )? ")"
| "str_subreplace" "(" exp "," exp "," exp "," exp ")"
| "str_find" "(" exp "," exp (" , " exp )? ")"
| "str_upper" "(" exp ")"
| "str_lower" "(" exp ")"
```

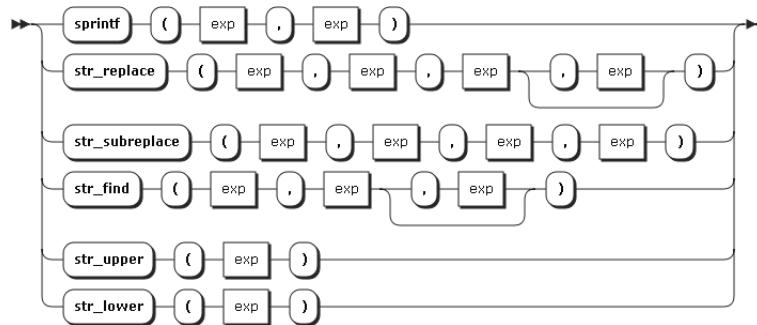


Figura 6.53: Gramática funciones cadenas de caracteres

6.2.3.11.4 Funciones arrays

```
array_func ::= "array_explode" "(" exp "," exp ")"
| "arrayImplode" "(" exp "," exp ")"
| "array_chunk" "(" exp "," exp ")"
| "array_reduce" "(" exp "," exp ")"
array_explode ::= array_explode "(" exp "," exp ")"
arrayImplode ::= arrayImplode "(" exp "," exp ")"
array_chunk ::= array_chunk "(" exp "," exp ")"
array_reduce ::= array_reduce "(" exp "," exp ")"
```

Figura 6.54: Gramática funciones array

6.2.3.11.5 Funciones expresiones regulares

```
regexp_func ::= "regexp" "(" exp ")"
| "search" "(" exp "," exp (" , " list )? ")"
| "match" "(" exp "," exp ")"
```

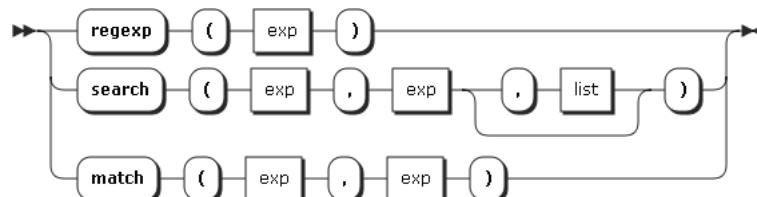


Figura 6.55: Gramática funciones expresiones regulares

6.2.3.11.6 Funciones fechas y tiempo

```
date ::= "date" "(" exp ")"
| "time" "(" ")"
```

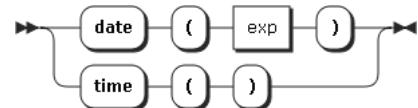


Figura 6.56: Gramática funciones fechas y tiempo

6.2.3.11.7 Funciones acceso a entorno

```
environment ::= "getenv" "(" exp ")"
```



Figura 6.57: Gramática funciones acceso a entorno

6.2.3.11.8 Funciones ficheros

```
file ::= "file" "(" exp ("," exp)? ")"
| "fput" "(" exp "," exp ")"
| "fwrite" "(" exp "," exp ")"
| "fappend" "(" exp "," exp ")"
| "fget" "(" exp ("," exp)? ")"
| "fread" "(" exp ")"
| "fclose" "(" exp ")"
| "fseek" "(" exp "," exp ("," fposs)? ")"
| "ftell" "(" exp ")"

fpos ::= FSET
| FCUR
| FEND
```

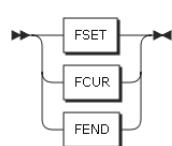


Figura 6.58: Gramática fpos

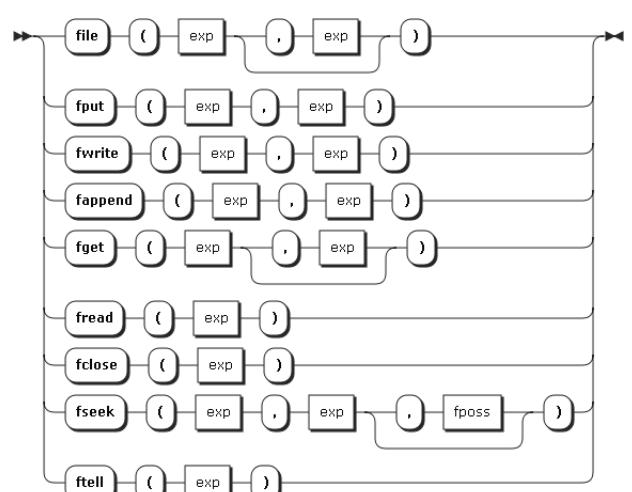


Figura 6.59: Gramática file

6.2.3.11.9 Funciones procesos

```
process ::= "exec" "(" "exp" ")"
| "eval" "(" "exp" ")"
| "fork" "(" ")"
| "wait" "(" "exp" ")"
| "signal" "(" "exp" ", " exp ")"
| "signalhandler" "(" "exp" ", " exp ")"
| "exitProcess" "(" ")"
| "getpid" "(" ")"
| "getppid" "(" ")"
| "process" "(" "exp" (" ", " list)? ")"
```

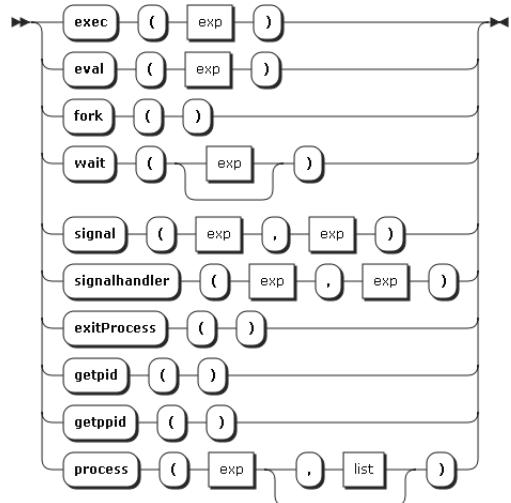


Figura 6.60: Gramática funciones procesos

6.2.3.12. Generadores

```
generator ::= "(" exp (";" exp)? "for" id (";" id)? "in" exp ( "if" exp )? ("{" stmts "}")? ")"
| "(" exp (";" exp)? "for" "(" id (";" id)? "in" exp ")" ( "if" exp )? ("{" stmts "}")? ")"
```

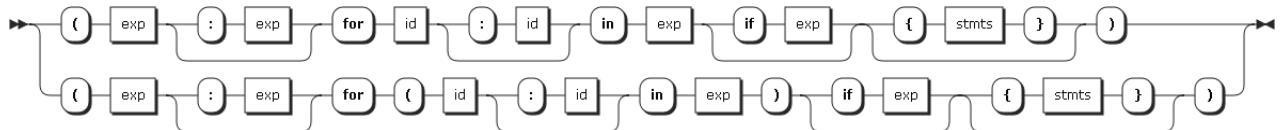


Figura 6.61: Gramática generadores

6.2.4. Identidades

```
identity ::= num
| "true"
| "false"
| "null"
| str
| rexp
| id
```

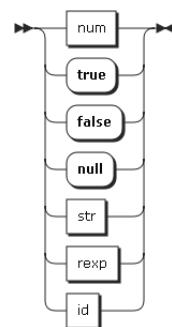


Figura 6.62: Gramática identidades

6.2.4.1. Parámetros

```

params_val ::= params_val "," id "=" identity
|   params "," id "=" identity
|   id "=" identity
|   params

params ::= params "," "&"? id
|   "&"? id
  
```

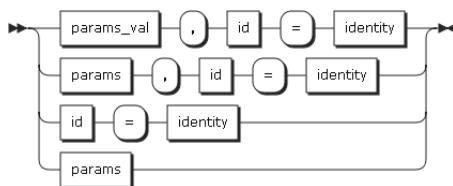


Figura 6.63: Gramática valores de parámetros

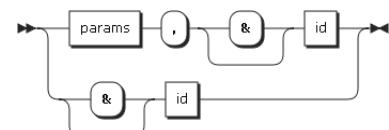


Figura 6.64: Gramática parámetros

6.2.4.2. Listas y pares

```

list ::= list "," exp?
|   exp

map ::= map "," pair?
|   pair

pair ::= exp ":" exp
  
```

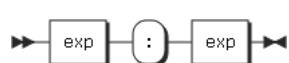


Figura 6.65: Gramática pares

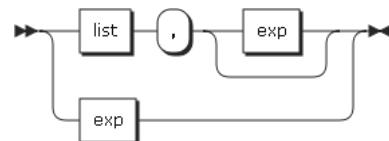


Figura 6.66: Gramática lista

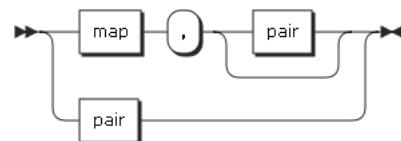


Figura 6.67: Gramática diccionario

6.2.4.3. Identificador

`id ::= [a-zA-Z_] [a-zA-Z0-9_]*`

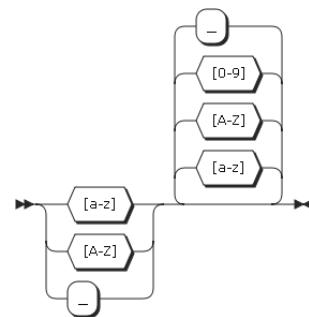


Figura 6.68: Gramática identificador

6.2.4.4. Números

num ::= ("+" | "-") [0-9] + (. [0-9] +)?

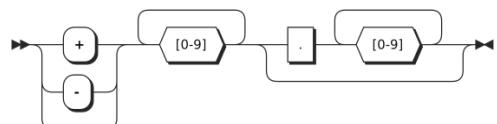


Figura 6.69: Gramática números

6.2.4.5. Cadenas de caracteres

```
str ::= " ".* " "
```

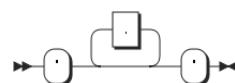


Figura 6.70: Gramática cadena de caracteres

6.2.4.6. Array

```

array ::=      "{" list "}"
             |  "{" map "}"
             |  "{{}}"
             |  "arrayop"
             |  access

```

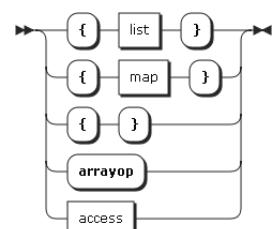


Figura 6.71: Gramática array

6.2.4.7. Expresión regular

```
regexp ::= " ".*" "
```

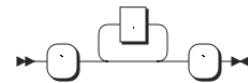


Figura 6.72: Gramática expresión regular

6.2.4.8. Comentarios

```
comments ::= /* [^*]* */  
          | /* /[^\\n]  
          | /* #[^#]
```

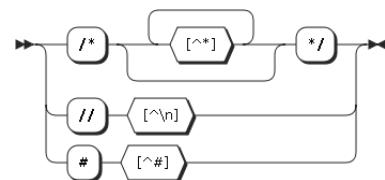


Figura 6.73: Gramática comentarios

6.3. Diseño de comunicaciones

OMI puede ser ejecutado de forma que guarde en un fichero una estructura de datos que representa el proceso de interpretación seguido. Esta estructura de datos tiene un formato JSON. En esta sección se presenta la estructura de estos ficheros mediante el esquema de json-schema.org.

Cuando el intérprete se ejecuta en modo servidor se vale de esta misma estructura para devolver el estado del proceso. Otros sistemas que hagan de cliente pueden interpretar esta estructura de datos para operar con los mismos.

6.3.1. Esquema JSON

6.3.1.1. Nodos ejecutables

La primera estructura de datos JSON que es guardada representa el árbol fruto del análisis léxico y sintáctico. Esta estructura de datos tiene como elemento base un nodo que mantendrá relaciones con otros nodos. A partir del nodo raíz se puede obtener todo el árbol de nodos.

```
1 {  
2   "$schema": "http://omi-project.com/json-schema/node",  
3   "title": "Nodos del árbol sintáctico"  
4   "type": "object",  
5   "properties": {  
6     "id": {  
7       "description": "Posición de memoria del nodo",  
8       "type": "string",  
9     },  
10    "name": {  
11      "description": "Nombre del nodo",  
12      "type": "string",  
13    }  
14  }  
15}
```

```

13 },
14   "type": {
15     "description": "Tipo del nodo",
16     "type": "string",
17   },
18   "size": {
19     "description": "Tamaño del nodo en Bytes",
20     "type": "string",
21   },
22   "rel": {
23     "description": "Relaciones con otros nodos",
24     "type": "array",
25     "items": {
26       "ref": "http://omi-project.com/json-schema/node"
27     }
28   },
29   "relname": {
30     "description": "Nombre de las relaciones con otros nodos",
31     "type": "array",
32     "items": {
33       "type": "string",
34     }
35   },
36 },
37 }

```

Listing 6.1: JSON nodos ejecutables

6.3.1.2. Acciones

Para representar el proceso de interpretación se precisa de una estructura de datos que indique las acciones llevadas a cabo en el proceso.

```

1 {
2   "$schema": "http://omi-project.com/json-schema/process",
3   "title": "Proceso de interpretación",
4   "type": "array",
5   "items": {
6     "type": "object",
7     "properties": {
8       "action": {
9         "description": "Acción que se corresponde con un paso",
10        "type": "enum",
11      },
12      "id": {
13        "description": "Id del nodo sobre el que se lleva a cabo la acción",
14        "type": "string",
15      },
16      "attrs": {
17        "description": "Atributos de la acción. Dependen de la acción",
18        "type": "object",
19      }
20    }
21  }
22 }

```

Listing 6.2: JSON acciones

Los tipos de acciones son los siguientes:

run: Ejecución de un nodo.

runClass: Ejecución de un nodo clase.

endClass: Finaliza la ejecución de una clase.

setParent: Establece el parente de un nodo clase.

setValue: Establece el valor de un nodo.

setRef: Establece el valor de una referencia.

accessVar: Accede al valor de una variable.

accessFunc: Accede al valor de una función.

accessClass: Accede al valor de una clase.

clone: Clona un nodo.

newNode: Crea un nuevo nodo.

changeRef: Cambia el valor de una referencia.

changeValue: Cambia el valor de un nodo.

delete: Elimina un nodo.

print: Imprime en la salida estándar.

input: Toma valores de la entrada estándar.

toSymbols: Añade elementos a la tabla de símbolos interna de un nodo.

removeLevel: Elimina un nivel de la tabla de símbolos.

return: Se devuelve un valor.

error: Se ha producido un error.

sleep: Se espera un evento.

runStatic: Comienza la ejecución de un elemento estático.

endStatic: Finaliza la ejecución de un elemento estático.

runPrivate: Comienza la ejecución de un elemento privado.

endPrivate: Finaliza la ejecución de un elemento privado.

runGlobal: Comienza la ejecución de un elemento global.

endGlobal: Finaliza la ejecución de un elemento global.

line: Especifica la línea actual del código fuente.

6.4. Diseño de componentes

Esta sección constituye el modelo de comportamiento básico. Detalla cómo los distintos objetos interaccionan y se comunican entre sí para analizar la entrada del usuario, correspondiente al código fuente, e interpretarlo para su ejecución.

Se presenta un diagrama de secuencia para la operación interpretación de un código fuente. Luego se presentan diagramas de comunicación correspondientes a algunas sentencias escritas en el lenguaje.

6.4.1. Interpretar código fuente

Para la interpretación de código fuente el sistema crea los objetos encargados del análisis léxico y sintáctico del mismo. El objeto principal del sistema se encarga de crear e inicializar el analizador sintáctico (parser) a partir del código fuente, a su vez este crea el analizador léxico (scanner) que crea una estructura correspondiente al código fuente que será interpretado.

El sistema envía un mensaje para que el parser analice el código fuente. El analizador sintáctico define una serie de reglas gramaticales e irá comprobando que el código fuente cumple estas reglas a la vez que las utiliza para crear un árbol de derivación denominado árbol sintáctico. Para ello hace uso de una serie de tokens que irá solicitando al analizador léxico. Este último obtendrá los tokens a partir del código fuente, mediante una serie de reglas léxicas formadas a partir de lenguajes regulares.

Por cada regla gramatical que se cumpla en el parser, este habrá obtenido del scanner tantos tokens como componentes léxicos sean necesarios para cumplir la regla.

Además por cada regla gramatical se construirá un nodo ejecutable formado a partir del valor asociado a una serie de tokens, o a partir de otros nodos ejecutables construidos por reglas de mayor prioridad y creados en anteriores interacciones. Esto formará un árbol sintáctico formado por nodos ejecutables que contienen la semántica que encierran las construcciones del código fuente.

A partir del nodo raíz del árbol de nodos ejecutables comienza un recorrido en profundidad del árbol que conllevará la ejecución del código fuente, produciéndose así el resultado semántico esperado.

En el caso de que no se de una regla gramatical que se corresponda con el código analizado se producirá un error sintáctico. Por otro lado, en el caso de que alguna cadena contenida en el código fuente no se corresponda con los lenguajes regulares que define el scanner se producirá un error léxico.

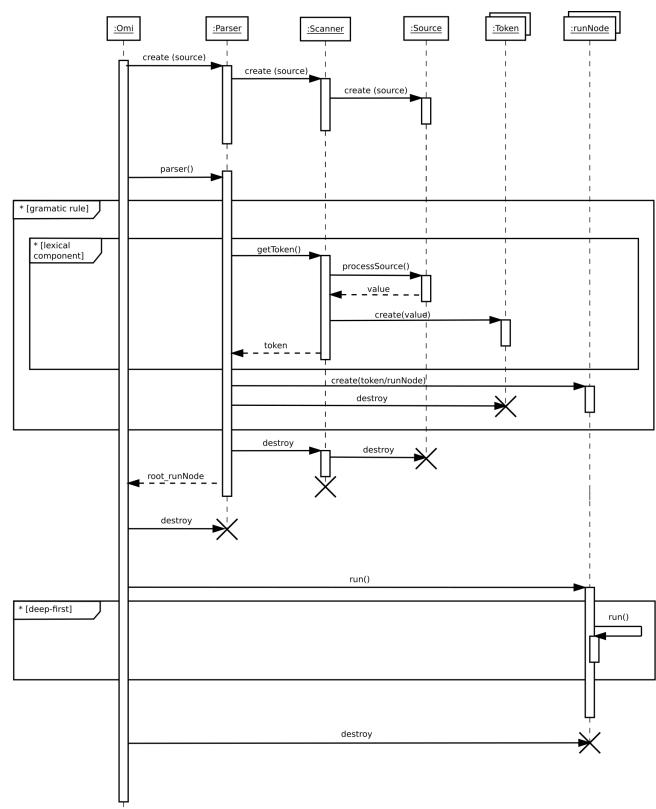


Figura 6.74: Interacción interpretar código fuente

6.4.2. Sentencias

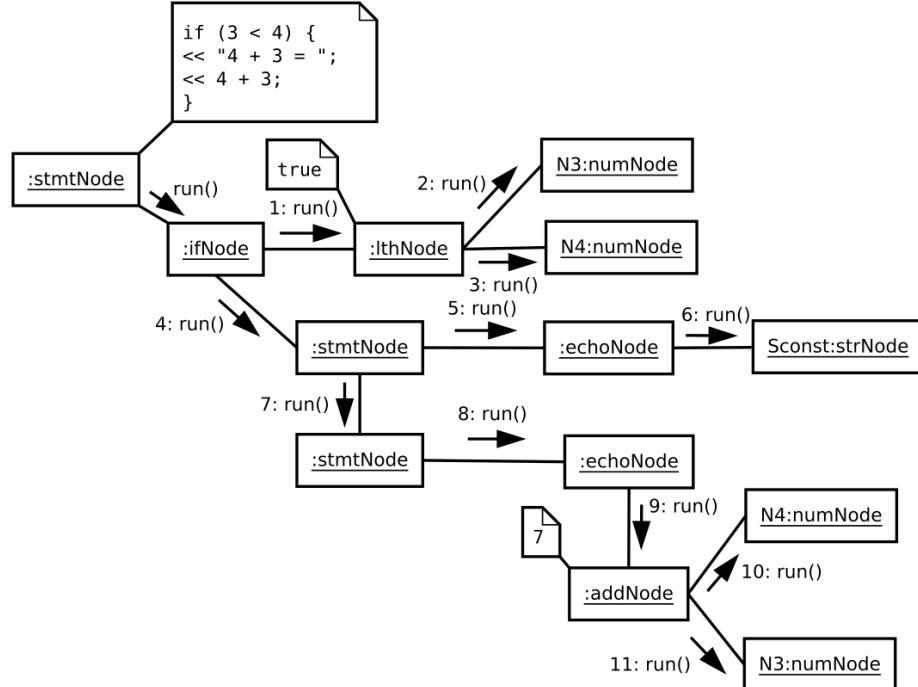


Figura 6.75: Objetos sentencia condicional

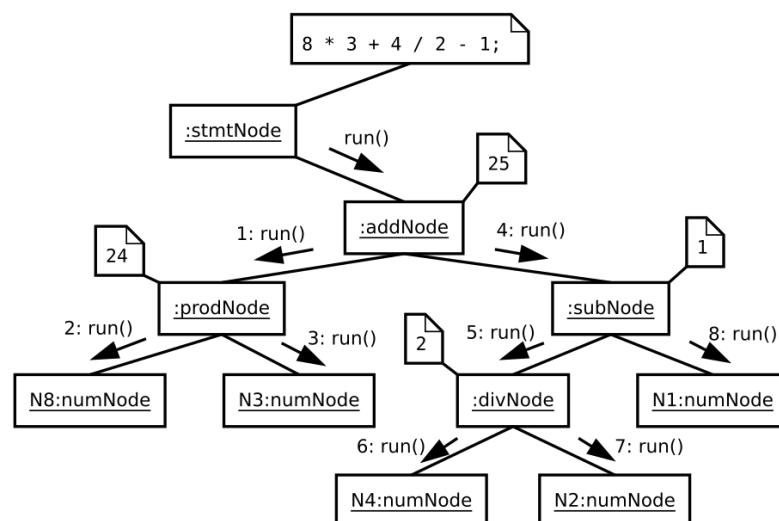


Figura 6.76: Objetos operaciones aritméticas

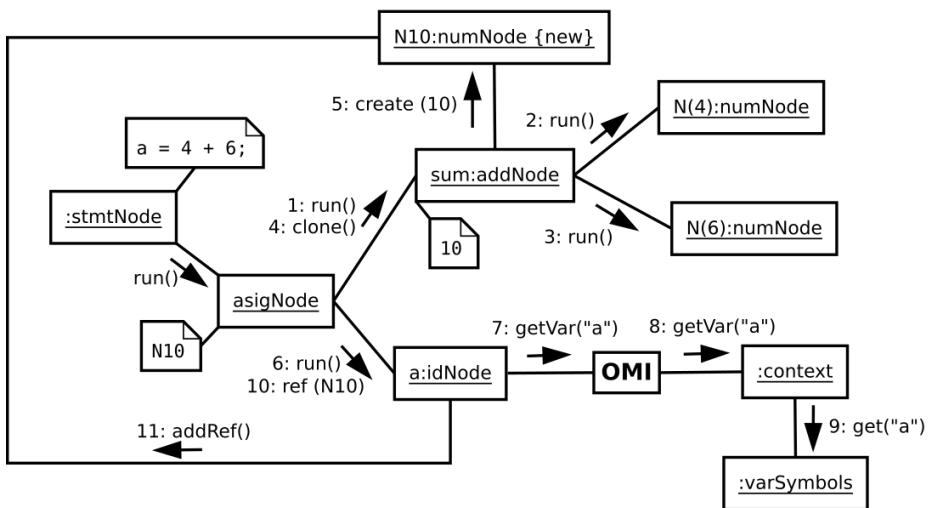


Figura 6.77: Objetos asignaciones

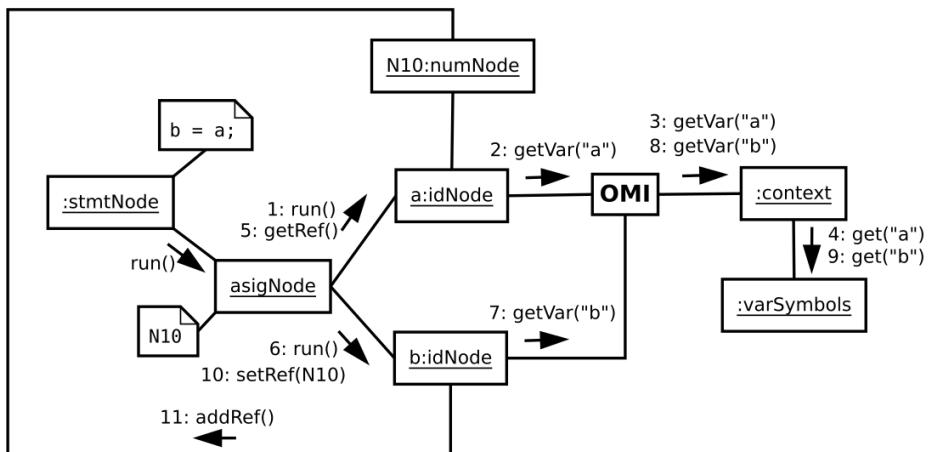


Figura 6.78: Objetos asignaciones (2)

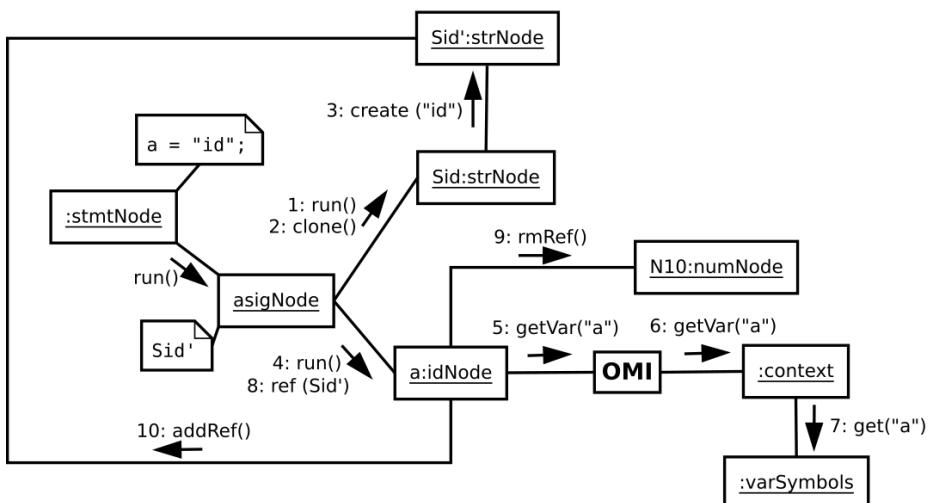


Figura 6.79: Objetos asignaciones (3)

6.5. Diseño de la interfaz de usuario

En esta sección se describe el diseño de la interfaz del usuario a partir del análisis llevado a cabo. El intérprete presenta una interfaz de consola de comandos, por lo que el diseño se corresponde con una descripción de los usos del comando y el listado de las opciones que acepta. El cliente runTree y la web en general presentan una interfaz web. El diseño se corresponde con una descripción gráfica de las páginas que la conforman y de las secciones que las componen.

6.5.1. Intérprete

El intérprete es un programa de consola de comando por lo que la interfaz con el usuario no es gráfica.

El comando “omi” permite ejecutar el intérprete. Se puede usar de las siguientes forma:

- omi [opciones] < *fichero* > [argumentos...]
- omi -c < *codigo* > [argumentos...]
- omi -i [argumentos...]
- omi -sj < *fichero.json* >

El listado de opciones que acepta el comando a continuación:

- i : Ejecuta el intérprete de forma interactiva.
- c < *codigo* > : Interpreta el código dado.
- l < *fichero.so* > : Carga el módulo < *fichero.so* >.
- h : Muestra la ayuda.
- V : Muestra la versión.
- j < *fichero.json* > : Imprime una descripción del procesos en formato json en el fichero < *fichero.json* >.
- x < *pasos* > : Obtiene < *pasos* > pasos del proceso de interpretación en cada petición.
- s : Ejecuta como servidor en el puerto 8888.

6.5.2. runTree

runTree es un cliente web del intérprete OMI. Muestra información sobre el proceso de interpretación y permite navegar por este.

El sistema lo compone una única página web con un diseño de cuadrícula.

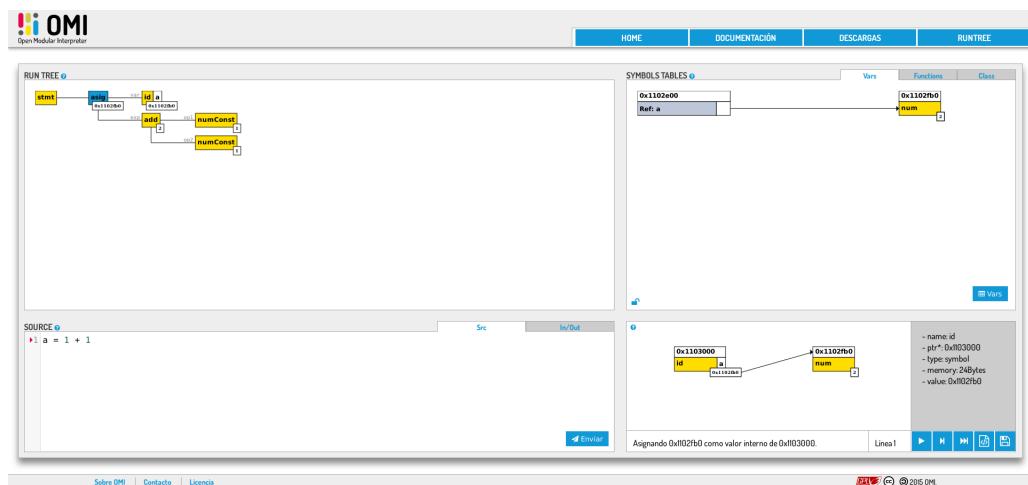


Figura 6.80: Interfaz de usuario runTree

La primera parte de la retícula, superior izquierda, muestra el árbol sintáctico correspondiente al código enviado. En este árbol se marcarán los nodos a medida que se van ejecutando y resolviendo semánticamente, mostrando información directa de cada nodo como el nombre, el valor o el tipo. En la segunda parte, superior derecha, muestra las tablas de símbolos de variables, funciones y clases. Esta sección permite navegar por las tablas de símbolos y los elementos que serán referenciados desde las mismas.

En la tercera parte, inferior izquierda, se muestra el código fuente y la interfaz de entrada/salida del programa.

En la última sección, inferior derecha, se muestra una consola informativa, en la que aparece una descripción del nodo actualmente en ejecución o los nodos seleccionados por el usuario. También se colocan en esta sección las opciones de control para avanzar un paso, una sentencia, la reproducción automática, abrir un fichero local o guardar el código en un fichero local.

6.5.3. Sitio web

Según el análisis de la interfaz de usuario llevado a cabo se procede a detallar gráficamente cada página descrita.

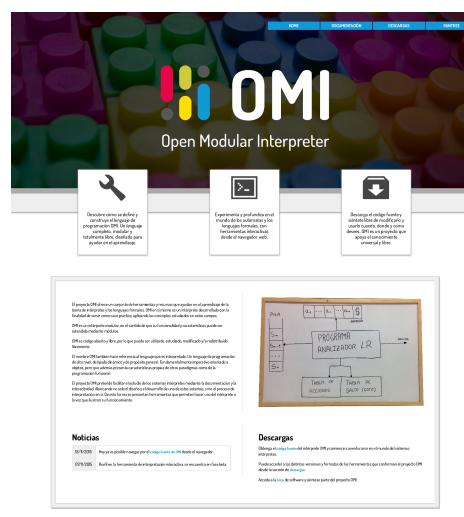


Figura 6.81: Interfaz de usuario home

Figura 6.82: Interfaz de usuario sobre OMI

Figura 6.83: Interfaz de usuario contacto

Figura 6.84: Interfaz de usuario índice de la documentación

Figura 6.85: Interfaz de usuario documento

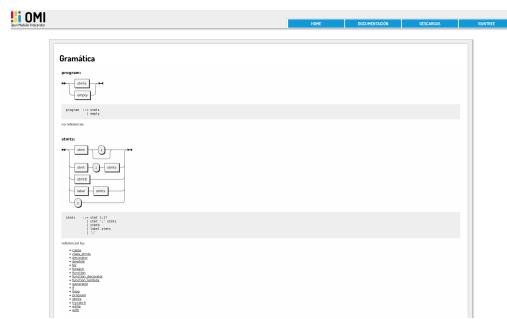


Figura 6.86: Interfaz de usuario navegador de gramática

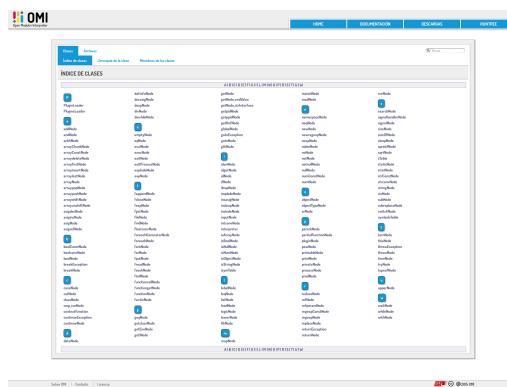


Figura 6.87: Interfaz de usuario navegador de clases

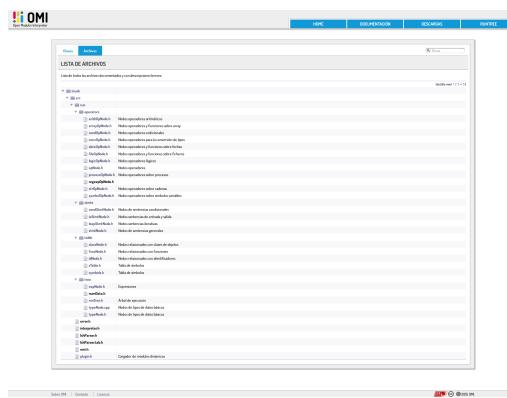


Figura 6.88: Interfaz de usuario navegador de ficheros

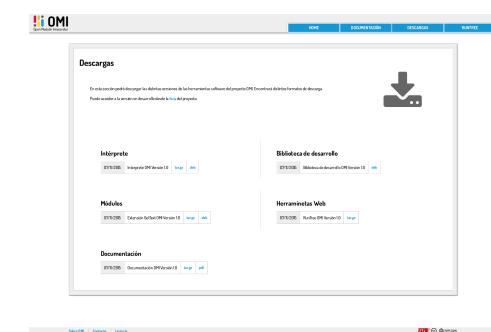


Figura 6.89: Interfaz de usuario descargas

Capítulo 7

Construcción del Sistema

En esta sección se tratan los aspectos relacionados con la implementación del sistema y su codificación. Para ello se describen las herramientas software y hardware utilizadas en el desarrollo, y la estructura del código fuente.

7.1. Entorno de construcción

Entorno de desarrollo (IDE): Geany 1.24

Lenguaje de programación: C++ 03

Compilador: GCC 4.9

Configuración automática: Autoconf 2.69

Construcción automática: Automake 1.14

Gestión de dependencias: Make 4.0

Control de versiones: Subversion 1.8.10

Generador de analizador léxico: Flex 2.5.39

Generador de analizador sintáctico: Bison 3.0.4

Depurador: GDB 7.7.1

Bibliotecas de desarrollo

Editor de línea e histórico: Readline 5.2

Expresiones regulares: BoostRegex 1.55

Matemáticas: Biblioteca estándar de C 2.19

Enlaces dinámicos: Biblioteca del sistema GNU/Linux libdl 3.1.8

Desarrollo Web

Programación en servidor: PHP 5.6

Programación en cliente: JavaScript 1.5

Estructura del contenido: HTML5

Presentación del contenido: CSS 3

7.2. Ficheros de código fuente

El sistema software se constituye de una serie de módulos o componentes en forma de ficheros, cada uno de los cuales contiene las estructuras de programación y el código fuente necesario para implementar cada una de las funcionalidades del sistema.

interpreter: Interprete.

lshScanner: Analizador léxico.

lshParser: Analizador sintáctico.

error: Sistema de errores.

plugins: Sistema de extensiones.

run/runTree: Abstracción de nodo ejecutable.

run/expNode: Abstracción de nodos ejecutables expresiones.

run/symbols: Estructura de datos tabla de símbolos.

run/sTable: Gestión de tabla de de símbolos y definiciones.

run/typeNode: Nodos ejecutables para cada tipo de dato.

run/numData: Representación interna de datos numéricos.

run/stmtNode: Nodos ejecutables sentencias de control.

run/operatorBaseNode: Nodos ejecutables operadores básicos.

run/operatorLogicNode: Nodos ejecutables operadores lógicos.

run/operatorArithNode: Nodos ejecutables operadores aritméticos.

run/operatorStrNode: Nodos ejecutables operadores sobre cadenas.

run/operatorArrayNode: Nodos ejecutables operadores sobre arrays.

run/operatorRegexpNode: Nodos ejecutables operadores sobre expresiones regulares.

run/operatorDateNode: Nodos ejecutables operadores sobre fechas y tiempo.

run/operatorFileNode: Nodos ejecutables operadores sobre ficheros.

run/operatorProcessNode: Nodos ejecutables operadores sobre procesos.

A continuación se describen las dependencias entre ficheros mediante una serie de paquetes que contienen diagramas de componentes. Este aspecto del sistema queda completamente descrito mediante la combinación de estos paquetes.

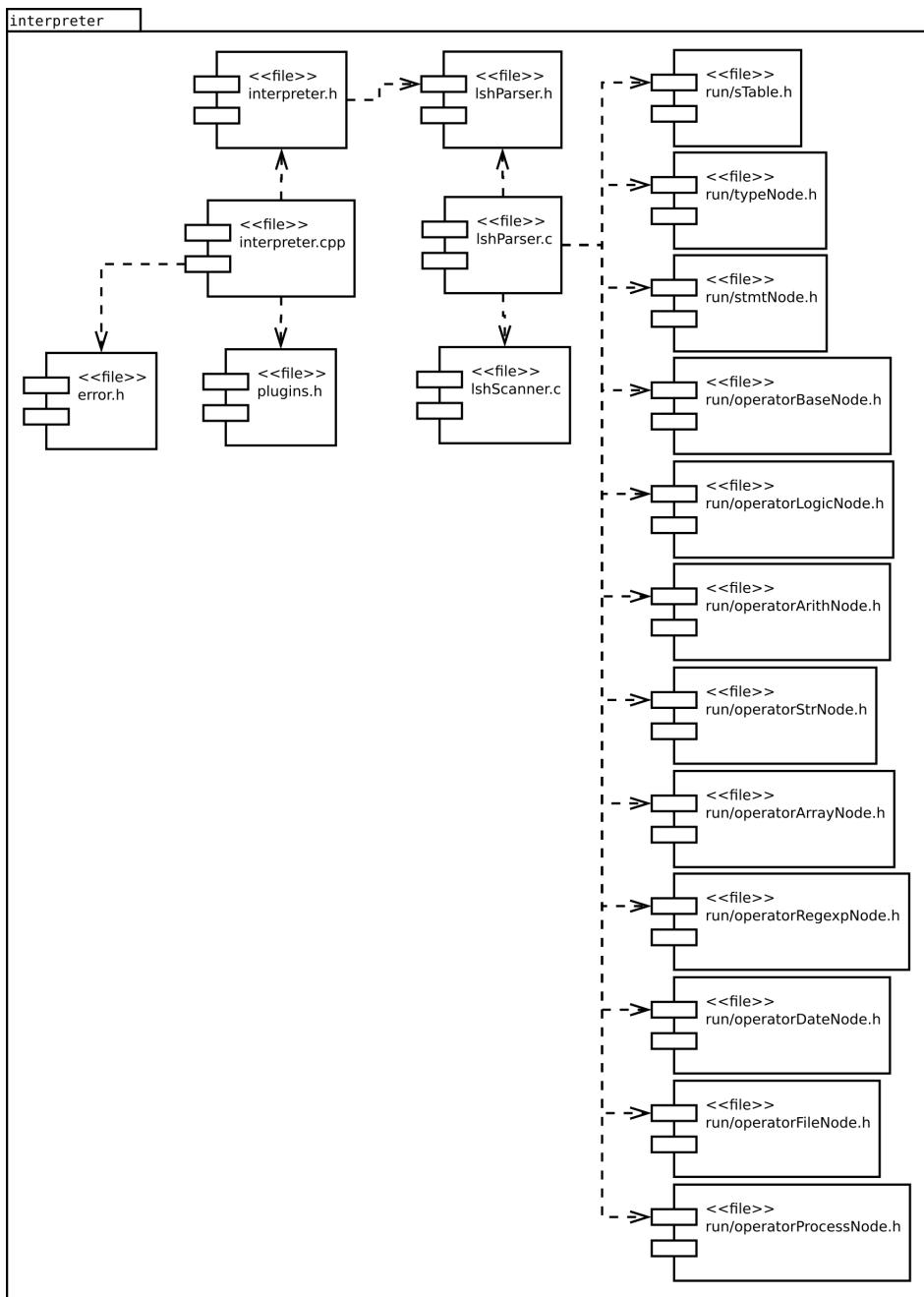


Figura 7.1: Ficheros intérprete

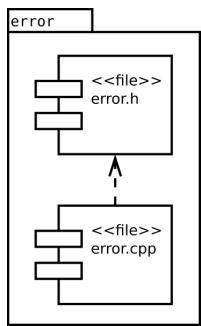


Figura 7.2: Ficheros error

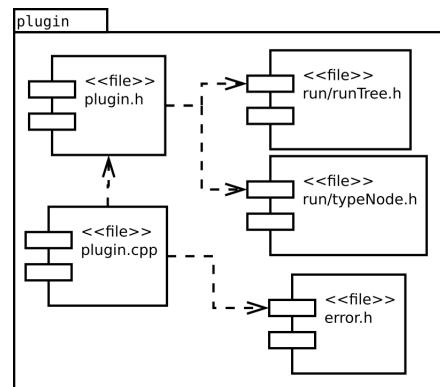


Figura 7.3: Ficheros plugin

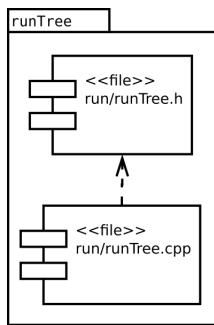


Figura 7.4: Ficheros runTree

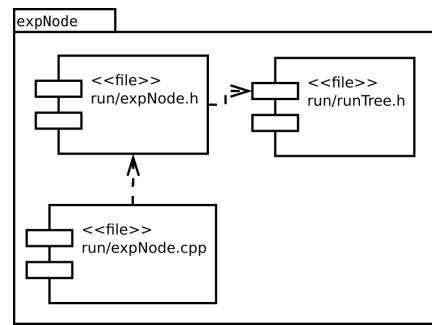


Figura 7.5: Ficheros expNode

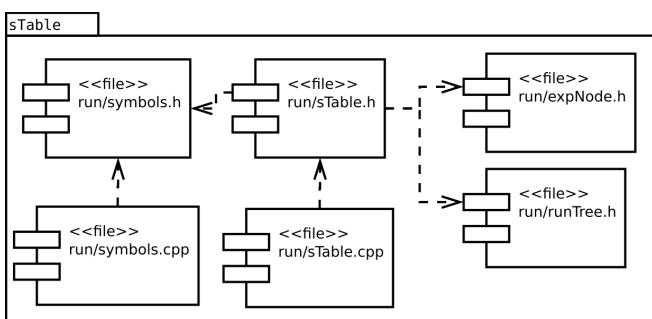


Figura 7.6: Ficheros sTable

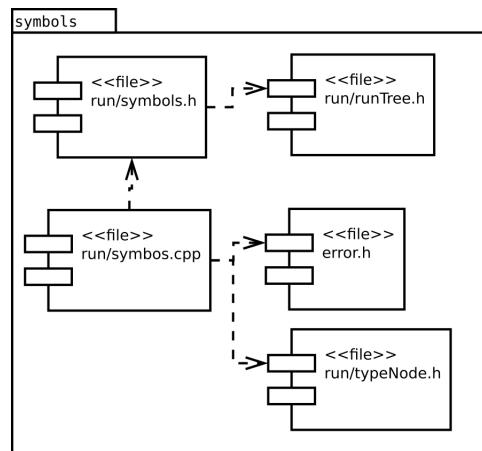


Figura 7.7: Ficheros symbols

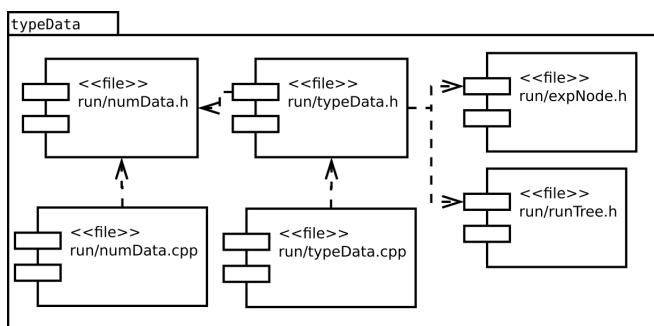


Figura 7.8: Ficheros typeData

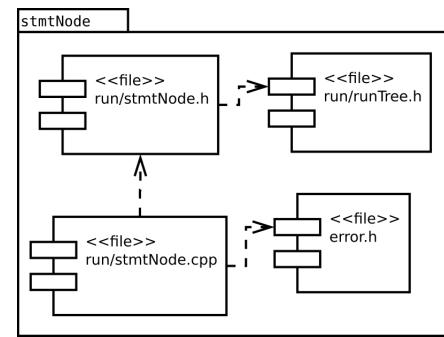


Figura 7.9: Ficheros stmtNode

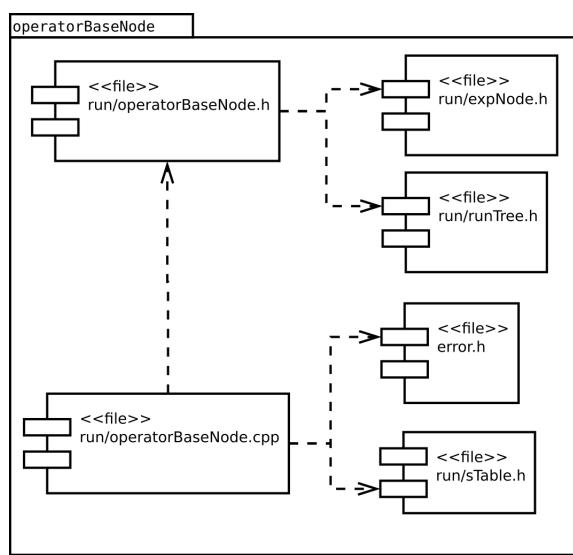


Figura 7.10: Ficheros operatorBaseNode

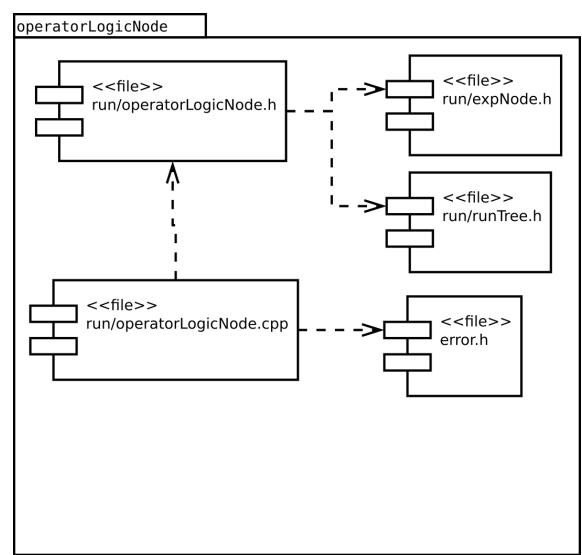


Figura 7.11: Ficheros operatorLogicNode

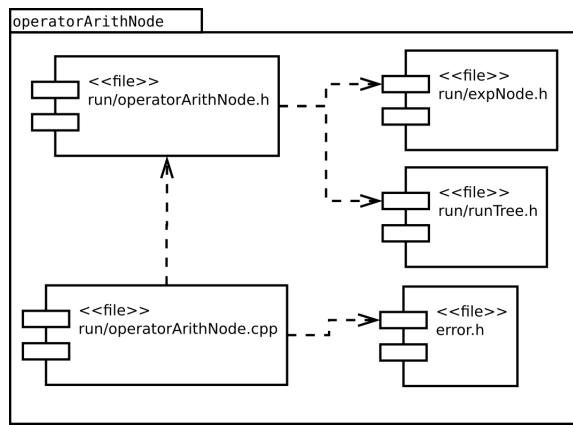


Figura 7.12: Ficheros operatorArithNode

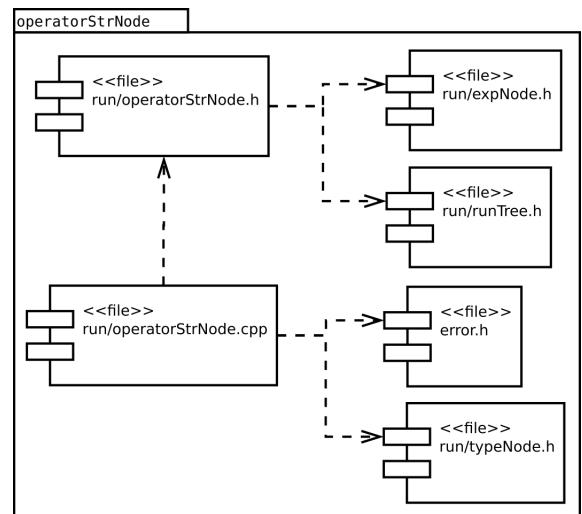


Figura 7.13: Ficheros

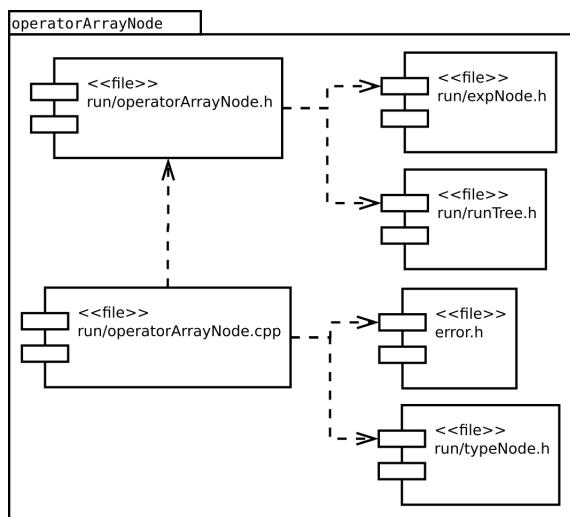


Figura 7.14: Ficheros operatorArrayNode

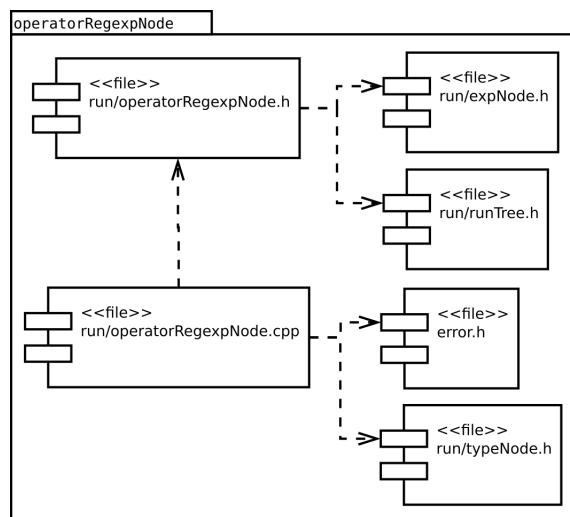


Figura 7.15: Ficheros operatorRegexpNode

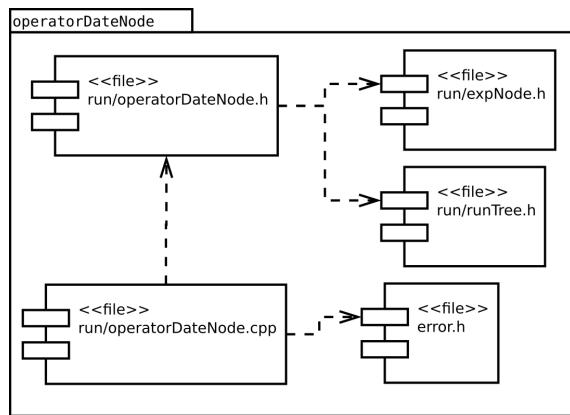


Figura 7.16: Ficheros operatorDatteNode

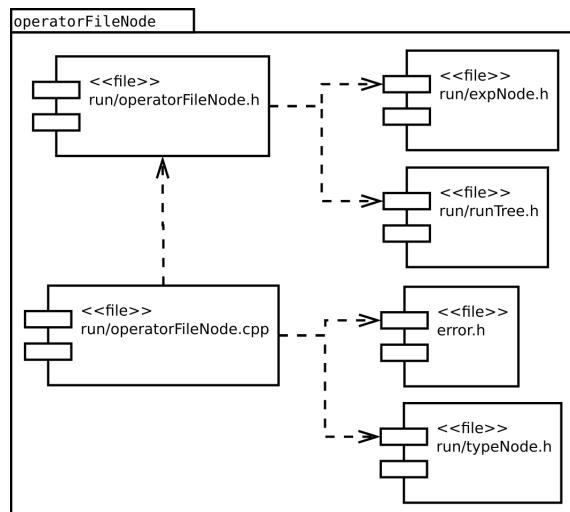


Figura 7.17: Ficheros operatorFileNode

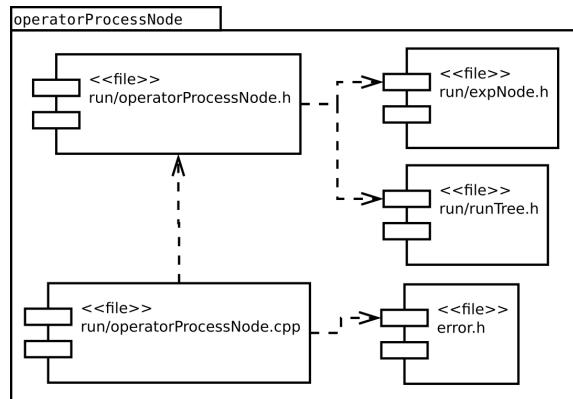


Figura 7.18: Ficheros operatorProcessNode

7.3. Código fuente

En esta sección se presentan extractos de código fuentes pertenecientes al intérprete OMI. Se ha tomado los métodos “run” de nodos significativos dado que es el método que se encarga de producir el significado semántico. Se presentan nodos relativos al acceso a la tabla de símbolos variables, a la asignación y la operación suma debido a su importancia y su simplicidad.

7.3.1. Acceso a variable

```
1 void idNode::run (bool resolvkey) {
2     #if JSON==1
3         interpreter::to_jsonRun(this);
4     #endif
5     sTable *table_ = sTable::sTable_use;
6     refNode *str = new refNode(id_);
7     exist_ = table_->exist(str);
8     ref_ = table_->access(str);
9     delete str;
10    runNode *val = ref_->getRef();
11    refNode * key = NULL;
12    ref_ = refNode::resolved(val);
13    if (privateNode::is)
14        table_->setPrivate(new refNode(id_));
15    if (!ref_->isprivate_)
16        ref_->isprivate_ = privateNode::is;
17    #if JSON==1
18        interpreter::to_jsonSetValue(this, val);
19    #endif
20 }
```

Listing 7.1: Código acceso a variable

7.3.2. Asignación

```
1 void asigNode::run () {
2     runNode *node_aux = node2_, *nodeR_ = NULL;
3     #if JSON==1
4         interpreter::to_jsonRun(this);
5     #endif
6     ref_ = NULL;
7     nexpNode::resolvedRef(node_aux);
8     #if JSON==1
9         if (node2_ != node_aux)
10            interpreter::to_jsonRun(this);
11     #endif
12     if (!(bool)dynamic_cast<functionNode*>(node_aux) && node2_ == node_aux) {
13         node_aux->run();
14         #if JSON==1
15             interpreter::to_jsonRun(this);
16         #endif
17     }
18     nodeR_ = this->isRefNode(node_aux)?node_aux:expNode::clone(node_aux);
19     noderef(nodeR_);
20     #if JSON==1
21         interpreter::to_jsonSetValue(this, nodeR_);
22     #endif
23     setValue(nodeR_);
24 }
```

Listing 7.2: Código asignación

7.3.3. Operación suma

```
1 void addNode::run () {
2     #if JSON==1
3         interpreter::to_jsonRun(this);
4     #endif
5     runNode* op1 = node1_;
6     runNode* op2 = node2_;
7     nexpNode::resolved (op1);
8     #if JSON==1
9         interpreter::to_jsonRun(this);
10    #endif
11    nexpNode::resolved (op2);
12    #if JSON==1
13        interpreter::to_jsonRun(this);
14    #endif
15    numvalue_ = addNode::do_add (op1, op2);
16    #if JSON==1
17        interpreter::to_jsonSetValue(this, numvalue_);
18    #endif
19 }
```

Listing 7.3: Código operación suma

Capítulo 8

Pruebas del Sistema

El objetivo de esta sección del documento es presentar el plan de pruebas seguido, incluyendo los diferentes tipos de pruebas que se han realizado.

En primer lugar se describe la estrategia de pruebas seguida. Para ello se describe el alcance de estas y el procedimiento de pruebas de regresión realizado.

Otro aspecto tratado es la descripción del entorno que se deberá utilizar para la realización de las pruebas. Esto incluye los requisitos de software y hardware necesarios.

Además se incluyen los perfiles y participantes necesarios para llevar a cabo los casos de prueba. Esto son los roles desde los que se llevarán a cabo las pruebas.

Por último se documentan las diferentes pruebas realizadas según el tipo al que estas pertenecen.

8.1. Estrategia

Las pruebas llevadas a cabo comprenden toda la funcionalidad del sistema. En cada iteración del ciclo de vida se realizan pruebas relativas a las funcionalidades comprendidas en la misma.

Para el diseño de las pruebas se ha tomado un enfoque funcional o de caja negra, centrada en la especificación de las funciones, la entrada y salida. La técnica utilizada consiste en definir los casos de pruebas a partir de clases de equivalencias:

1. Identificar las restricciones de formato y contenido de los datos de entrada.
2. A partir de las restricciones identificar las clases de equivalencias. Contemplando tanto datos válidos como erróneos.
3. Se identifican y definen los casos de prueba a partir de las clases de equivalencias.

Para asegurar la estabilidad en el resultado de las pruebas realizadas en cada iteración, se ha implementado un mecanismo que lleve a cabo las pruebas de forma regresiva, ejecutándose todo el conjunto de estas cada vez que una nueva funcionalidad del software es desarrollada.

Además de realizar pruebas funcionales en cada iteración, se han realizado una serie de pruebas no funcionales. Esta comprende aquellas que aseguran que se cumplen los requisitos no funcionales.

8.2. Entorno de pruebas

En este punto se define el entorno utilizado para las pruebas, tanto a nivel de software como a nivel de hardware.

8.2.1. Hardware

Para las pruebas se ha tomado un PC de características medias, en el momento de la realización del proyecto. Este se conforma de:

CPU: Intel Core i5-4460 3.2Ghz

Disco Duro: 1TB Western/Seagate

Memoria Ram: 4GB DDR3 1333 PC3-10600 CL9 Kingston

Placa Base: Gigabyte GA-H81M-S2H

Tarjeta gráfica: 1GB Integrada Intel

8.2.2. Software

Para las pruebas se ha utilizado un sistema GNU/Linux, instalado y configurado desde una distribución de paquetes Debian 8 Jessie.

Sistema operativo: GNU/Linux

Distribución de paquetes: Debian Jessie 8

Entorno gráfico: Xfce 4

Interprete de comando: Bash 4.3.8

Interprete OMI: OMI 0.1

8.3. Roles

Las pruebas se han llevado a cabo desde la perspectiva del único rol que interactuará con el sistema correspondiente al usuario que normalmente será un programador que hará uso del mismo.

El usuario programador hace uso integro del sistema en función a los programas que desarrolle en el lenguaje. Un único programa, o conjunto finito de estos no son suficientes para probar todos los aspectos del sistema. Así aunque las pruebas son llevadas a cabo desde un perfil programador, estas no constituyen programas completos con un objetivo específico, sino baterías de pruebas que pretenden probar cada aspecto funcional del sistema.

8.4. Niveles de pruebas

Las pruebas se presentan en distintos niveles, según el tipo de prueba realizado. Dado la gran cantidad de pruebas estas en su mayoría han sido automatizadas, presentándose junto al sistema y siendo una característica más del mismo, la cual es recomendable ejecutar tras cada instalación.

8.4.1. Pruebas unitarias

Son pruebas llevadas a cabo sobre cada artefacto o pieza software producido en cada iteración del ciclo de desarrollo. Estas aseguran la correcta implementación de las piezas desarrolladas, comprobando que están libre de errores y que cada entrada es procesada correctamente. Además cada caso de prueba se ha estructurado en función los tipos de entradas para una mejor organización de las mismas.

Las pruebas unitarias están recogidas en un subsistema de la aplicación que automatizan su ejecución y asegura que son llevadas a cabo de forma regresiva. Este subsistema consiste en un script bash que se encarga de ejecutar varios script OMI que implementan pruebas unitarias, la salida de la ejecución será comparada con ficheros de texto plano que guardan el resultado correcto de las mismas.

A continuación se expone algunos ejemplos de las pruebas unitarias realizadas:

8.4.1.1. Asignación de booleanos

Estos casos de prueba se centran en el operador asignación cuando el elemento asignado es de valor booleano.

Entrada:

```
a = true ;
```

Descripción: Asignación sobre la variable *a* el valor booleano true.

Salida esperada: *a* tiene el valor true.

Salida obtenida: *a* tiene el valor true.

Entrada:

```
a = false ;
```

Descripción: Asignación sobre la variable *a* el valor booleano false.

Salida esperada: *b* tiene el valor false.

Salida obtenida: *b* tiene el valor false.

Entrada:

```
for ( i = 0; i < 10; ++i )
    a = true ;
```

Descripción: A la variable *a* se le asigna el valor booleano true en cada iteración del bucle

Salida esperada: *a* tiene el valor true.

Salida obtenida: *a* tiene el valor true.

Entrada:

```
a = b = true;
```

Descripción: A la variable *b* se le asigna el valor booleano true, el valor de *b* es asignado a *a*

Salida esperada: Tanto *b* como *a* tienen el valor true.

Salida obtenida: Tanto *b* como *a* tienen el valor true.

Entrada:

```
array [0] = true;
```

Descripción: A el array *array* se le asigna en el índice 0 el valor true

Salida esperada: *array* contiene en la para la clave 0 el valor true. Si *array* no existe es creado.

Salida obtenida: *array* contiene en la para la clave 0 el valor true. Si *array* no existe es creado.

Entrada:

```
array [1] = true
```

Descripción: A el array *array* se le asigna en el índice 1 el valor false

Salida esperada: *array* contiene en la para la clave 1 el valor false. Si *array* no existe es creado.

Salida obtenida: *array* contiene en la para la clave 1 el valor false. Si *array* no existe es creado.

Entrada:

```
for ( i = 0; i < 10; ++i)
    array [ i ] = true;
```

Descripción: A el array *array* se le asigna el valor true a los índices que van desde 0 a 9.

Salida esperada: *array* contiene en la para las claves del 0 al 9 el valor true. Si *array* no existe es creado.

Salida obtenida: *array* contiene en la para las claves del 0 al 9 el valor true. Si *array* no existe es creado.

Entrada:

```
str = "ABCDEF";
str[0] = false;
```

Descripción: A la cadena *str* se le asigna en la posición 0 el valor *false*.

Salida esperada: La cadena *str* queda con el índice 0 con el valor *false*. La cadena resultante es "BCDEF".

Salida obtenida: La cadena *str* queda con el índice 0 con el valor *false*. La cadena resultante es "BCDEF".

Entrada:

```
str = "ABCDEF";
str[0] = true;
```

Descripción: A la cadena *str* se le asigna en la posición 0 el valor *true*.

Salida esperada: La cadena *str* queda con el índice 0 con el valor *true*. La cadena resultante es "1BCDEF".

Salida obtenida: La cadena *str* queda con el índice 0 con el valor *true*. La cadena resultante es "1BCDEF".

Entrada:

```
while (size str)
    str[0] = false;
```

Descripción: A la cadena *str* se le asigna en la posición 0 el valor *false*, mientras que la cadena sea distinta a la cadena vacía.

Salida esperada: La cadena *str* queda vacía.

Salida obtenida: La cadena *str* queda vacía.

Entrada:

```
a = false;
b = &a;
a = true;
```

Descripción: A *a* se le asigna el valor false, a *b* se le asigna una referencia a *a*, a *a* se le asigna el valor true.

Salida esperada: Tanto el valor de *b* como el de *a* es true.

Salida obtenida: Tanto el valor de *b* como el de *a* es true.

Entrada:

```
true = true;
```

Descripción: A la constante true se le asigna la constante true.

Salida esperada: Error asignación a constante.

Salida obtenida: Error asignación a constante.

Entrada:

```
true = false;
```

Descripción: A la constante true se le asigna la constante false.

Salida esperada: Error asignación a constante.

Salida obtenida: Error asignación a constante.

Entrada:

```
a = true = true;
```

Descripción: A *a* se le asigna el valor de asignar la constante true se le asigna la constante true.

Salida esperada: Error asignación a constante. *a* permanece con el valor que tenía.

Salida obtenida: Error asignación a constante. *a* permanece con el valor que tenía.

8.4.1.2. Operador de igualdad con operandos numéricos

Estos casos de prueba se enfoca en el operador de igualdad cuando las entradas son de tipo numéricas.

Entrada:

```
0 == 0;
```

Descripción: Operador igualdad donde el primer operando es 0 y el segundo operando es 0.

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

```
0 == 1;
```

Descripción: Operador igualdad donde el primer operando es 0 y el segundo operando es 1.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 == 0;
```

Descripción: Operador igualdad donde el primer operando es 1 y el segundo operando es 0.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
10 == 10;
```

Descripción: Operador igualdad donde el primer operando es 10 y el segundo operando es 10.

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

10 == 2;

Descripción: Operador igualdad donde el primer operando es 10 y el segundo operando es 2.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

2 == 10;

Descripción: Operador igualdad donde el primer operando es 2 y el segundo operando es 10.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

-12 == -12;

Descripción: Operador igualdad donde el primer operando es -12 y el segundo operando es -12.

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

-12 == 12;

Descripción: Operador igualdad donde el primer operando es -12 y el segundo operando es 12.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

12 == -12;

Descripción: Operador igualdad donde el primer operando es 12 y el segundo operando es -12.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

18.7 == 18.7;

Descripción: Operador igualdad donde el primer operando es 18.7 y el segundo operando es 18.7.

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

18.7 == 18;

Descripción: Operador igualdad donde el primer operando es 18.7 y el segundo operando es 18.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

18 == 18.7;

Descripción: Operador igualdad donde el primer operando es 18 y el segundo operando es 18.7.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

$-18.7 == -18;$

Descripción: Operador igualdad donde el primer operando es -18.7 y el segundo operando es -18.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

$-18.69 == -18.7;$

Descripción: Operador igualdad donde el primer operando es -18.69 y el segundo operando es -18.7.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

$18.069 == -18.0;$

Descripción: Operador igualdad donde el primer operando es 18.069 y el segundo operando es -18.0.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

$-11.79999999999999 == -11.79999999999999;$

Descripción: Operador igualdad donde el primer operando es -11.79999999999999 y el segundo operando es -11.79999999999999.

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

$-11.79999999999999 == -11.79999999999999;$

Descripción: Operador igualdad donde el primer operando es -11.79999999999999 y el segundo operando es -11.79999999999999.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

$-11.79999999999999 == -11.79999999999998;$

Descripción: Operador igualdad donde el primer operando es -11.79999999999999 y el segundo operando es -11.79999999999998.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

$-11.79999999999998 == -11.79999999999999;$

Descripción: Operador igualdad donde el primer operando es -11.79999999999998 y el segundo operando es -11.79999999999999.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

$-11.79999999999999 == -11.79999999999998;$

Descripción: Operador igualdad donde el primer operando es -11.79999999999999 y el segundo operando es -11.79999999999998.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano true. Aviso de precisión sobrepasada por representación numérica finita.

Entrada:

$-11.799999999999999 == -11.8;$

Descripción: Operador igualdad donde el primer operando es -11.799999999999999 y el segundo operando es -11.8 .

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano true. Aviso de precisión sobrepasada por representación numérica finita.

Entrada:

$9999999999999 == 10000000000001;$

Descripción: Operador igualdad donde el primer operando es 9999999999999 y el segundo operando es 10000000000001 .

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

$1000000000000001 == 1000000000000001;$

Descripción: Operador igualdad donde el primer operando es 1000000000000001 y el segundo operando es 1000000000000001 .

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

$1000000000000001 == 1000000000000002;$

Descripción: Operador igualdad donde el primer operando es 10000000000000001 y el segundo operando es 10000000000000002.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1000000000000002 == 1000000000000001;
```

Descripción: Operador igualdad donde el primer operando es 1000000000000002 y el segundo operando es 1000000000000001.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1000000000000002 == 1000000000000001;
```

Descripción: Operador igualdad donde el primer operando es 1000000000000002 y el segundo operando es 1000000000000001.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
999999999999999 == 1000000000000001;
```

Descripción: Operador igualdad donde el primer operando es 999999999999999 y el segundo operando es 1000000000000001.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano true. Aviso de precisión sobrepasada por representación numérica finita.

Entrada:

```
10000000000000001 == 10000000000000002;
```

Descripción: Operador igualdad donde el primer operando es 10000000000000001 y el segundo operando es 10000000000000002.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano true. Aviso de precisión sobrepasada por representación numérica finita.

Entrada:

```
10000000000000001 == 10000000000000002;
```

Descripción: Operador igualdad donde el primer operando es 10000000000000002 y el segundo operando es 10000000000000001.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano true. Aviso de precisión sobrepasada por representación numérica finita.

8.4.1.3. Definición y llamadas de funciones con cuerpo vacío

Los casos de pruebas expuestos a continuación comprenden la definición y posterior llamada de funciones con cuerpo vacío.

Entrada:

```
function empty_fun () { }
empty_fun ();
```

Descripción: Definición de función sin parámetros y cuerpo vacío. Llamada respetando el número de parámetros

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado.

Entrada:

```
function empty_fun () { }
empty_fun ("param");
```

Descripción: Definición de función sin parámetros y cuerpo vacío. Llamada sin respetar el número de parámetros (más de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
function empty_fun (param) { }
empty_func ("param");
```

Descripción: Definición de función con un único parámetro y cuerpo vacío. Llamada respetando número de parámetros.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param) { }
empty_func ();
```

Descripción: Definición de función con un único parámetro y cuerpo vacío. Llamada sin respetar el número de parámetros (menos de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
function empty_fun (param) { }
empty_func ("param1", "param2");
```

Descripción: Definición de función con un único parámetro y cuerpo vacío. Llamada sin respetar el número de parámetros (más de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
param = "param";
function empty_fun (&param) { }
empty_func (param);
```

Descripción: Definición de función con un único parámetro pasado por referencia y cuerpo vacío. Llamada dando una variable como valor de la referencia.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Entrada:

```
param = {"param"};
function empty_fun (&param) { }
empty_func (param[0]);
```

Descripción: Definición de función con un único parámetro pasado por referencia y cuerpo vacío. Llamada dando la posición de un array como valor de la referencia.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Entrada:

```
function empty_fun (&param) { }
empty_func (param);
```

Descripción: Definición de función con un único parámetro pasado por referencia y cuerpo vacío. Llamada dando una variable no definida como valor de la referencia.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Entrada:

```
function empty_fun (&param) { }
empty_func ("const");
```

Descripción: Definición de función con un único parámetro pasado por referencia y cuerpo vacío. Llamada dando una constante como valor de la referencia.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de constante como referencia.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de constante como referencia.

Entrada:

```
function empty_fun (param1, param2) { }
empty_func ("param1", "param2");
```

Descripción: Definición de función con dos parámetros y cuerpo vacío. Llamada respetando el número de parámetros.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param1, param2) { }
empty_func ("param1");
```

Descripción: Definición de función con dos parámetros y cuerpo vacío. Llamada sin respetar el número de parámetros (menos de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
function empty_fun (param1, param2) { }
empty_func ("param1", "param2", "param3");
```

Descripción: Definición de función con dos parámetros y cuerpo vacío. Llamada sin respetar el número de parámetros (más de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
function empty_fun (param1, param2 = "default") { }
empty_function ("param1", "param2");
```

Descripción: Definición de función con dos parámetros, uno con valor por defecto, y cuerpo vacío. Llamada facilitando todos los parámetros.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param1, param2 = "default") { }
empty_function ("param1");
```

Descripción: Definición de función con dos parámetros, uno con valor por defecto, y cuerpo vacío. Llamada facilitando los parámetros que no tienen valor por defecto.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param1, param2 = "default") { }
empty_function ();
```

Descripción: Definición de función con dos parámetros, uno con valor por defecto, y cuerpo vacío. Llamada sin facilitar los parámetros que no tienen valor por defecto.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
function empty_fun (param1, param2 = "default") { }
empty_function ("param1", "param2", "param3");
```

Descripción: Definición de función con dos parámetros, uno con valor por defecto, y cuerpo vacío. Llamada sin respetar el número de parámetros (más de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
function empty_fun (param1 = "default", param2 = "default") { }
empty_fun ("param1", "param2");
```

Descripción: Definición de función con dos parámetros, todos con valor por defecto, y cuerpo vacío. Llamada dando valor a todos los parámetros

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param1 = "default", param2 = "default") { }
empty_fun ("param1");
```

Descripción: Definición de función con dos parámetros, todos con valor por defecto, y cuerpo vacío. Llamada dando valor a algunos de los parámetros

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param1 = "default", param2 = "default") { }
empty_fun ();
```

Descripción: Definición de función con dos parámetros, todos con valor por defecto, y cuerpo vacío. Llamada tomando valor por defecto de todos los parámetros.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param1 = "default", param2 = "default") { }
empty_fun ("param1", "param2", "param3");
```

Descripción: Definición de función con dos parámetros, todos con valor por defecto, y cuerpo vacío. Llamada sin respetar el número de parámetros (más de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

8.4.2. Pruebas de integración

Las pruebas de integración son llevadas a cabo tras iteración del ciclo de vida. Incluyen casos de prueba correspondiente a la interacción de varios módulos o artefactos, desarrollados en la misma interacción del ciclo de desarrollo o anteriores. En este tipo de pruebas se ha de repasar las características desarrolladas en iteraciones anteriores con el objetivo de localizar errores en la integración de los módulos desarrollados con el resto del sistema.

En esta sección se recogen algunos casos de pruebas de integración. Junto el software se facilita un sistema de pruebas que comprueba automáticamente todos los casos.

Entrada:

```
<< 22 / 2 + 8 - 5 * 2;
```

Descripción: Impresión de expresión aritmética compuesta de varios operadores y operandos todos constantes.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: 9.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: 9.

Entrada:

```
array = {22};  
var = 5;  
<< array[0] / 2 + 8 - var * 2;
```

Descripción: Impresión de expresión aritmética compuesta de varios operadores, algunos operandos variables y/o posiciones de array.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: 9.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: 9.

Entrada:

```
function const2 () {  
    return 2;  
}  
array = {22};  
var = 5;  
<< array[0] / const2 () + 8 - var * const2 ();
```

Descripción: Impresión de expresión aritmética compuesta de varios operadores, algunos operandos variables, llamadas a funciones y/o posiciones de array.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: 9.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: 9.

Entrada:

```
<< !(( true && true ) || false );
```

Descripción: Impresión de expresión booleana compuesta de varios operadores y operandos todos constantes.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: false.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: false.

Entrada:

```
array = { true };
var = true;
<< !(( array [0] && var ) || false );
```

Descripción: Impresión de expresión booleana compuesta de varios operadores, algunos operandos variables y/o posiciones de array.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: false.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: false.

Entrada:

```
function identity (param) {
    return param;
}
array = { true };
var = true;
<< !(( array [0] && var ) || identity (false));
```

Descripción: Impresión de expresión booleana compuesta de varios operadores, algunos operandos variables, llamadas a función y/o posiciones de array.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: false.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: false.

Entrada:

```
class fac {  
    function factorial (a) {  
        fac = 1;  
        this->recursiva (a, fac);  
        return fac;  
    }  
    private function recursiva (a, &factorial){  
        if (a > 0){  
            factorial *= a;  
            this->recursiva (a - 1, factorial);  
        }  
    }  
}  
f = new fac ();  
<< f->factorial (4);
```

Descripción: Definición de clases con métodos públicos y privados. Paso de parámetros por referencia. Operadores aritméticos.

Salida esperada: Calcula el factorial de 4 de forma recursiva: 24.

Salida obtenida: Calcula el factorial de 4 de forma recursiva: 24.

8.4.3. Pruebas funcionales del sistema

Este tipo de pruebas tienen como objetivo comprobar la funcionalidad del sistema. La funcionalidad principal que debe cumplir el interprete es recibir un programa en forma de código fuente, interpretarlo y producir el resultado esperado.

Para comprobar que el sistema cumple la funcionalidad para el que fue diseñado se han realizado una serie de programas tipos, recogiendo diversos estilos de programación, y de una naturaleza distinta. Estos programas han sido codificados en el lenguaje reconocido por el intérprete y se han realizado distintas comprobaciones sobre los mismos.

8.4.3.1. Calculadora

Este programa representa una calculadora sencilla en la que se le pide al usuario dos operandos numéricos y una operación (suma, resta, producto o cociente). El programa muestra el resultado de la operación. El programa se ejecuta hasta que la operación dada no es reconocida. Es posible consultar el código correspondiente a la calculadora en el anexo 12.1.

8.4.3.2. Sistema de cuestionarios

El siguiente programa representa un sistema de cuestionarios. Es en si mismo un DSL (lenguaje específico de dominio) definido de forma interna, por lo que presenta una estructura y gramática similar al lenguaje reconocido por el intérprete.

Un cuestionario se define mediante preguntas y posibles respuestas. Cada pregunta presenta una valor, la suma de los valores de todas las preguntas se corresponde con el valor del cuestionario. El usuario que realice el cuestionario sacará una nota que se corresponderá con una parte del total según las preguntas que responda correctamente.

Cada pregunta del cuestionario tendrá una serie de posibles respuestas, las respuestas pueden ser de dos tipos:

De selección: En este caso tras la pregunta se dará una serie de respuestas acompañadas de un valor que indique si es correcta o falsa. Para una misma pregunta pueden existir más de una respuesta correcta. En la ejecución del cuestionario al usuario se le permitirá elegir entre todas las respuestas aquella (solo una) que considere es correcta.

De texto: En este caso tras la pregunta se dará una serie de respuestas todas correctas. En la ejecución del cuestionario se le permitirá al usuario introducir textualmente la solución, y solo en el caso de que se coincida con alguna de las repuesta esta será dada como correcta.

Es posible consultar el código fuente correspondiente al sistema de cuestionarios en el anexo 12.2.

8.4.3.3. Tic-Tac-Toe

El programa que se presenta a continuación se corresponde con el juego del Tic-Tac-Toe. Este juego, también llamado el juego de tres en raya, enfrenta a dos jugadores en una cuadrícula de 3x3. Cada jugador se corresponde con un símbolo, y se turnan para ponerlo o dibujarlo en una posición vacía de la cuadrícula. El jugador que consiga poner tres de sus símbolos en línea gana la partida.

No es difícil darse cuenta que si ambos jugadores utilizan la estrategia más óptima el juego terminará en empate. Al ser un juego sencillo se utiliza para enseñar conceptos de teoría de juegos y, dentro de la inteligencia artificial, la búsqueda de árboles de juego.

El programa en primer lugar solicita el nombre y tipo de los jugadores, pudiéndose ser estos humanos o máquinas. Luego turno a turno va solicitando a cada jugador una posición vacía de la cuadrícula en la que efectuar el movimiento, esto se hace hasta que se da una línea ganadora o hasta que se completa la cuadrícula.

Para determinar el movimiento o acción llevada en cada turno se prosigue de la siguiente forma:

- Si el jugador es humano el sistema solicita la posición (fila y columna) en la que poner el símbolo.
- Si el jugador es de tipo máquina se utiliza el algoritmo recursivo minimax para calcular el mejor movimiento a partir de búsquedas en árboles de juego y el estado del tablero. Esto se hace con la salvedad de los primeros turnos, ya que en estos la estrategia óptima es fija.

Es posible consultar el código correspondiente al programa Tic-Tac-Toe en el anexo 12.3.

8.4.4. Pruebas no funcionales del sistema

Estas pruebas están enfocadas a comprobar que el sistema cumple con los requisitos no funcionales determinado en las fases de especificación.

El interprete debe presentar un rendimiento óptimo en cuanto tiempo de interpretación. Dado que su objetivo no es constituir una herramienta para la producción de software este aspecto no es crítico, no obstante debe cumplir unos mínimos para que sea operativo.

Por otro lado un programa interpretado no puede excederse en la memoria física que ocupa. Para ello se debe medir la cantidad de memoria de las entidades que conforma el programa.

Además se debe asegurar dentro de unos márgenes que el intérprete está libre de vulnerabilidades y que no es posible hacer un uso indebido del mismo para explotar la plataforma sobre la que se ejecuta o el propio sistema software.

8.4.4.1. Rendimiento de tiempo

Para comprobar que el rendimiento que ofrece el software en relación a los tiempos tomados para la interpretación se ha sometido al sistema a una serie benchmarks conocidos, comparando los resultados con los obtenidos con otros lenguajes de programación interpretados.

Se considera que el sistema supera las pruebas de rendimiento siempre y cuando el tiempo en pasar los benchmarks sea inferior al doble de los lenguajes pensados para la producción software.

8.4.4.1.1 Fibonacci

Esta prueba consiste en medir el tiempo para distintas entradas de un programa que trata de calcular, de forma recursiva, el número correspondiente a una determinada posición de la sucesión de Fibonacci. Es posible consultar el código fuente correspondiente al programa de Fibonacci en el anexo 12.4.

Los tiempos obtenidos frente otros lenguajes son los siguientes:

Entrada	Tiempo (s)
10	0.020
20	0.104
30	8.720

Cuadro 8.1: Tiempos Fibonacci OMI

Entrada	Tiempo (s)
10	0.043
20	0.054
30	5.622

Cuadro 8.2: Tiempos Fibonacci PHP

Entrada	Tiempo (s)
10	0.023
20	0.029
30	4.963

Cuadro 8.3: Tiempos Fibonacci Python

8.4.4.1.2 N-body

El primer benchmark al que ha sido sometido el sistema se denomina “n-body”. Este consiste en una simulación de un sistema dinámico de partículas que se encuentran bajo la influencia de fuerzas físicas como la gravedad. Es posible consultar el código fuente del programa N-body en el anexo 12.5

Los tiempos obtenidos frente otros lenguajes son los siguientes:

Entrada	Tiempo (s)
500.000	14.10
5.000.000	240.4
50.000.000	1243.02

Cuadro 8.4: Tiempos N-body OMI

Entrada	Tiempo (s)
500.000	7.10
5.000.000	69.00
50.000.000	719.66

Cuadro 8.5: Tiempos N-body PHP

Entrada	Tiempo (s)
500.000	9.86
5.000.000	96.17
50.000.000	967.81

Cuadro 8.6: Tiempos N-body Python

Como se pude apreciar los tiempos obtenidos en OMI son significativamente mayores. Esto es debido a que OMI no compila el código fuente durante la ejecución mediante técnicas como la compilación justo a tiempo. El intérprete OMI se ha desarrollado para que describa el proceso de interpretación de una forma clara a todos los niveles, mientras que otros intérpretes son optimizados para ejecutar programas en un entorno de producción real.

8.4.4.2. Espacio de memoria

El espacio de memoria que ocupa un determinado programa en ejecución es de vital importancia. La memoria es un recurso físico limitado, y los programas deben hacer un buen uso de la misma. Dado que el interprete es una herramienta con la que se van a escribir otros programas, el uso de memoria que haga este influye en gran medida en la cantidad que ocuparán los programas con son interpretados. De esta forma si el interprete no hace un uso óptimo de la memoria los programas que este procesará tampoco.

Las unidades mínimas sobre las que opera el interprete son los nodos ejecutables. Así en primer lugar se va a medir cuánto ocupa los nodos ejecutables básicos y más comunes. Cabe decir que la representación interna de los distintos tipos de datos puede ser configurada como opciones de compilación, es por ello que se presenta la medición en función las distintas configuraciones posibles, indicándose solo aquellas que presentan un esquema óptimo debido a factores como el alineamiento.

La longitud del alineamiento en todos los casos viene dado por el puntero a la tabla de métodos virtuales. Este elemento referencia a una tabla que indexa los métodos de los que dispone un objeto debido a la jerarquía de herencia con la que se definió. Al ser un puntero su tamaño dependerá de la arquitectura del equipo.

8.4.4.2.1 Nodos lógicos

En el siguiente diagrama se presenta la memoria ocupada por un nodo de tipo lógico según las distintas combinaciones para la representación interna de los datos:

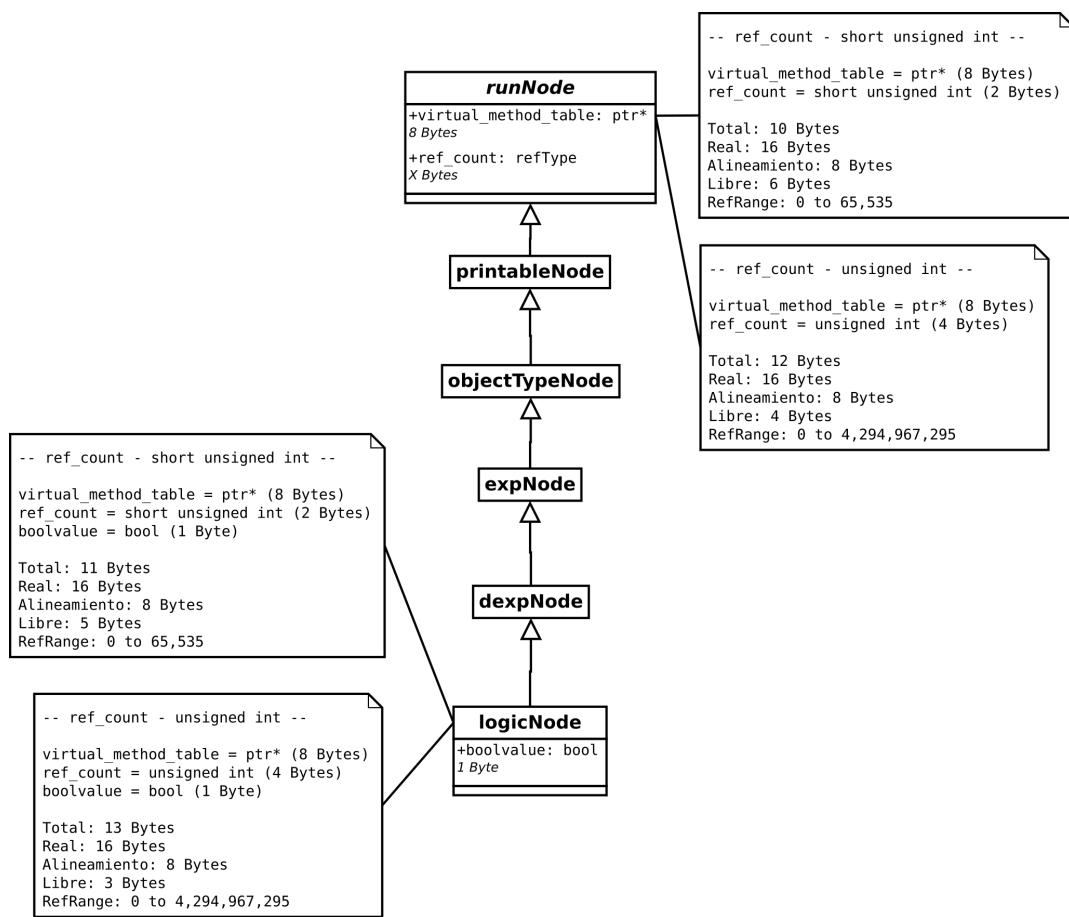


Figura 8.1: Espacio de memoria de nodos lógicos

8.4.4.2.2 Nodos aritméticos

En el siguiente diagrama se presenta la memoria ocupada por un nodo de tipo aritmético según las distintas combinaciones para la representación interna de los datos:

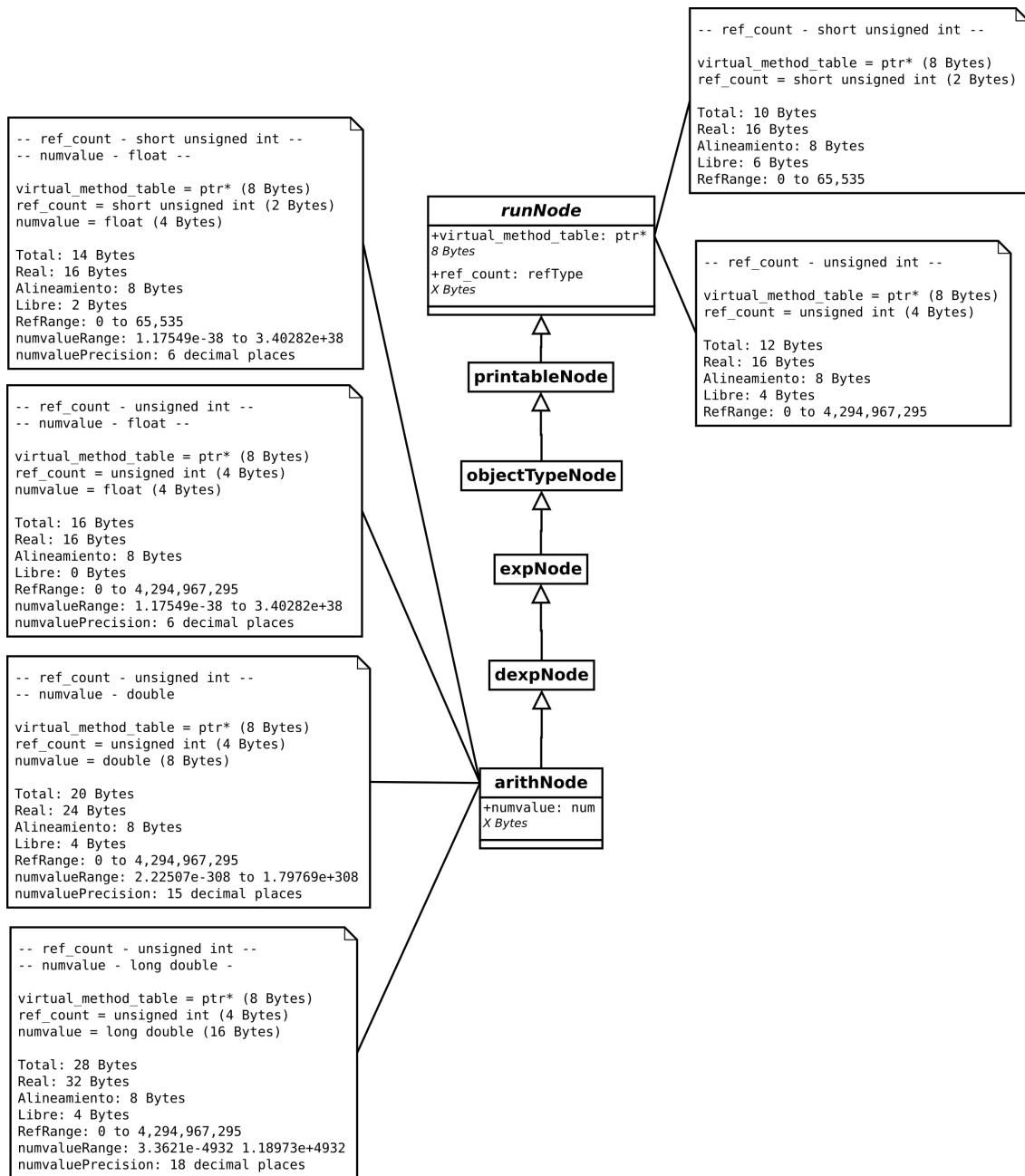


Figura 8.2: Espacio de memoria de nodos aritméticos

8.4.4.2.3 Nodos cadenas de caracteres

En el siguiente diagrama se presenta la memoria ocupada por un nodo de tipo cadena de caracteres según las distintas combinaciones para la representación interna de los datos:

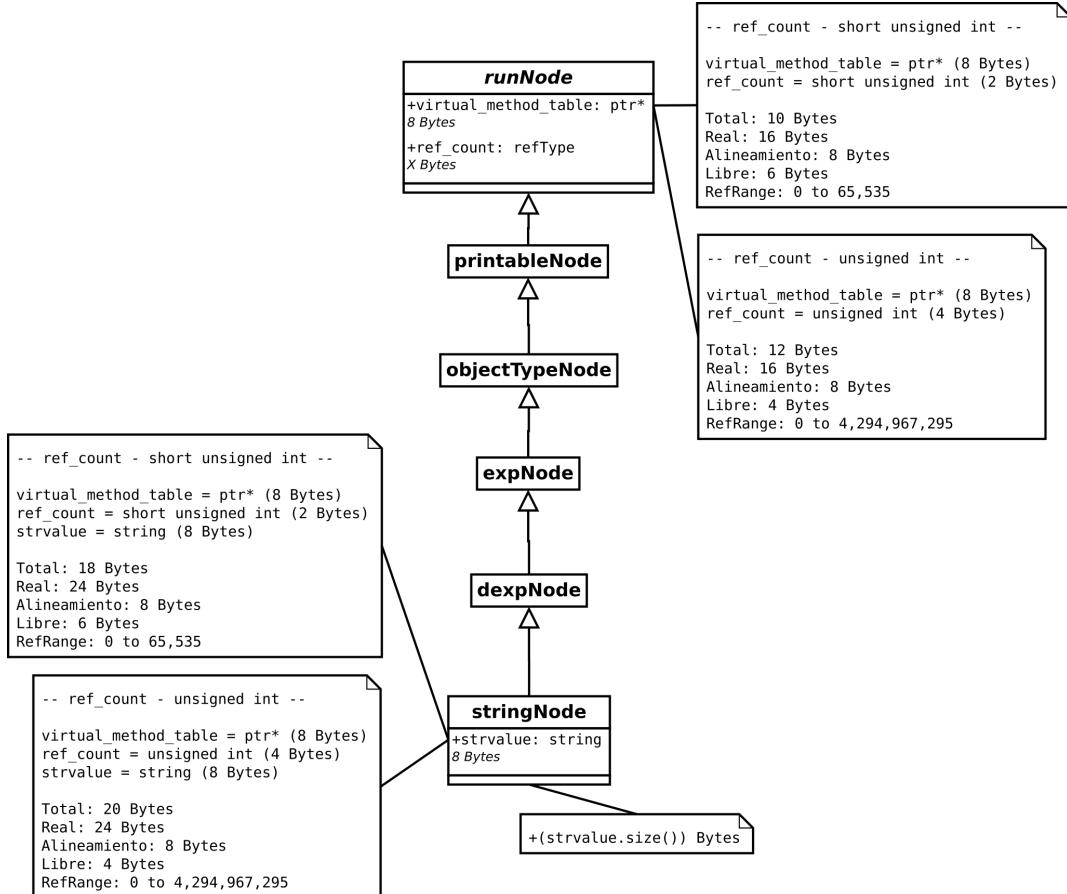


Figura 8.3: Espacio de memoria de nodos de cadenas de caracteres

8.4.4.3. Seguridad

El objetivo de este punto es medir la calidad en función de la seguridad del software desarrollado. Así, para asegurar que la aplicación cumple un mínimo de seguridad se han realizado una serie de auditorías al software.

En el tipo de pruebas realizadas solo se ha tenido en cuenta el software correspondiente al interprete, dejando fuera todo el sistema web que conforma la plataforma de distribución. Un sistema web se ve afectado por tipos de vulnerabilidades tales como DoS, XSS, CSRF, SQL injection, sistemas de autentificación... Por otro lado en un software de escritorio, como puede ser el interprete, se ve afectado por otros tipos de vulnerabilidades comunes.

Las pruebas que se han realizado sobre el software se enfocan en la entrada del usuario y se pueden categorizar de la siguiente forma:

Desbordamiento de buffer: Se ha comprobado las entradas directas del programa, controlando el tamaño máximo de estas para evitar el desbordamiento de la zona de memoria de datos.

Desbordamiento de buffer por variables de entorno: Se han controlado el uso de las variables de entorno para que estas no puedan ser usadas para desbordar el espacio de me-

moria destinada a los datos y sobrescribir punteros de control como el de la próxima instrucción.

Inyección de código: En instrucciones el las que se hace uso de otros lenguajes como por ejemplo para base de datos, se ha comprobado la entrada para que no sea posible modificar la consulta interna.

Salto de directorio: Se ha comprobado que la inclusión de ficheros no dependa directamente de una entrada sin verificar. Se ha intentado enjaular la aplicación para que no sea posible la inclusión de ficheros del sistema por parte del usuario.

Parte III

Epílogo

En esta última parte quedarán recogidas las conclusiones y los manuales necesarios para el manejo de la aplicación resultado del desarrollo. Así se desarrollan el manual de integración y explotación, el manual de usuario y las conclusiones del trabajo realizado.

Capítulo 9

Manual de implantación y explotación

En esta sección se detallan las instrucciones de instalación y explotación del sistema

9.1. Introducción

Los objetivos del proyecto OMI es servir como una plataforma para el aprendizaje de cómo se construyen los lenguajes de programación. Esto lo hace mediante el estudio de un intérprete propio del proyecto, desarrollado desde cero y con características de los lenguajes actuales. Para facilitar toda esta información el proyecto OMI pone a disposición del usuario una serie de herramientas que ayudan al aprendizaje, así mismo como toda la información relativa al proyecto. El sistema OMI pretende ser un sistema distribuido con una alta disponibilidad y de fácil acceso. Es por ello que todas las herramientas y recursos del proyecto son ofrecida vía online y pueden ser accedidas desde cualquier navegador o dispositivo.

La implantación del sistema OMI comprende la configuración del dominio, del servidor web y del sistema intérprete.

9.2. Requisitos previos

Para el correcto funcionamiento de la plataforma se precisa de un equipo con alta disponibilidad que hará de servidor web. Este equipo tendrá una una distribución GNU/Linux Debian o derivada.

9.3. Inventario de componentes

En esta sección se listan los componentes software incluidos en el proyecto, disponibles en el CD o en la forja del mismo:

- Web omi-project 1.0
- Intérprete OMI 1.0
- Biblioteca OMI 1.0

9.4. Procedimiento de instalación

Se procede con la instalación de los paquetes necesarios

```

1 apt-get install apache2 php5 bind9 autoconf automake build-essential libreadline-dev
  libboost-regex-dev

```

Listing 9.1: Instalación de paquetes

Se configura la zona del dominio y los registros asociados al mismo.

```

1 ; Zone file for omi-project.com
2 $TTL 14400
3 omi-project.com. 14400 IN SOA ns.omi-project.com. omi-project.com. (
4     10000000001
5     86400
6     7200
7     3600000
8     86400 )
9
10 omi-project.com. 14400 IN NS ns1.omi-project.com.
11 omi-project.com. 14400 IN NS ns2.omi-project.com.
12 omi-project.com. 14400 IN A $IP$
13 localhost 14400 IN A 127.0.0.1
14 omi-project.com. 14400 IN MX 10 mail
15 omi-project.com. 14400 IN TXT "v=spf1 mx ptr ip4:$IP$ mx:mail.omi-project.com +all"
16 omi-project.com. 14400 IN SPF      "v=spf1 mx ptr ip4:$IP$ mx:mail.omi-project.
   com +all"
17
18 mail 14400 IN A $IP$
19 www 14400 IN CNAME omi-project.com.
20 ftp 14400 IN A $IP$
21 ns 14400 IN A $IP$
22 ns1 14400 IN A $IP$
23 ns2 14400 IN A $IP$
24 admin 14400 IN A $IP$
25
26 _adsp._domainkey.omi-project.com. IN TXT "dkim=all; atps=y; asl=omi-project;"
```

Listing 9.2: Configuración del servidor de dominio

Descarga y compilación del proyecto OMI.

```

1 wget http://www.omi-project.com/download/code/omi_1.0.tar.gz
2 tar -xzvf omi_1.0.tar.gz
3 cd omi_1.0.tar.gz
4 ./configure JSON=1 SERVER=1
5 make
6 make install
7 make clean
```

Listing 9.3: Instalación de OMI

Configuración del servidor web Apache. Se define un nuevo host virtual

```

1 <VirtualHost *:80>
2   ServerAdmin admin@omi-project.com
3   ServerName omi-project.com
4   DocumentRoot $PATH$
5   ErrorLog $PATH$/error.log
6   TransferLog $PATH$/access.log
7   <Directory $PATH$ >
8     Require all granted
9     AllowOverride all
10  </Directory>
11
```

```

12   <IfModule php5_module>
13     php_flag session.cookie_httponly on
14   </IfModule>
15
16   <IfModule mod_headers.c>
17     Header always append X-Frame-Options SAMEORIGIN
18     Header set Server "securesauce http/html processor"
19     Header unset X-Powered-By
20   </IfModule>
21</VirtualHost>

```

Listing 9.4: Configuración del servidor web

Se ha de mover la web al directorio de producción y dar los permisos de acceso correspondientes

```

1 mv OMI_1.0/web $PATH$
2 chown -R omi:www-data $PATH$
3 chmod -R 750 $PATH$

```

Listing 9.5: Instalación web del proyecto OMI

Para finalizar se han de reiniciar los servicios.

```

1 service omi restart
2 service apache2 restart
3 service bind9 restart

```

Listing 9.6: Reinicio de servicios

9.5. Pruebas de implantación

Para comprobar que la implantación se ha llevado a cabo correctamente se puede llevar a cabo un chequeo de las pruebas automáticas. Para ello:

```

1 cd OMI_1.0
2 make check

```

Listing 9.7: Pruebas de implantación

También se puede acceder a las distintas secciones de la web. Navegando por los distintos páginas y comprobando la disponibilidad de los recursos.

Para comprobar que el servidor OMI se encuentra funcionando correctamente se puede usar el cliente runTree y comprobar que el código fuente enviado se interpreta correctamente.

9.6. Procedimientos de operación y nivel de servicio

Para comprobar el correcto funcionamiento del sistema en cuanto a rendimiento y disponibilidad se puede usar un navegador web y las herramientas de desarrollo que la mayoría de estos tienen integradas. También se puede automatizar un chequeo de disponibilidad desde una maquina tercera o la propia máquina

La web OMI no guarda ningún dato del usuario ni de los proceso asociados por lo que no es necesario hacer back-ups periódicos.

Es posible chequear errores en el servicio web mediante el log \$PATH\$/error.log. También es posible chequear las transferencias de datos y las peticiones atendidas en el fichero \$PATH\$/access.log.

Capítulo 10

Manual de usuario

OMI, acrónimo de Open Modular Interpreter, es un lenguaje para la creación de ‘scripts’ que presenta un propósito general y es de código abierto. El objetivo del proyecto OMI es servir de guía en la aplicación práctica de la teoría de compiladores e intérpretes. Para ello presenta un caso práctico en el que se construye un lenguaje de programación moderno a partir de los conceptos teóricos de autómatas y lenguajes formales.

La plataforma OMI representa un sistema software desarrollado para la comunidad académica. Pretende ser una herramienta utilizada en el aprendizaje de los sistemas intérpretes y traductores modernos, además de cualquier otro basado en los conceptos sobre los que estos se construyen. La plataforma OMI se constituye mediante los siguientes elementos:

- Documentación completa del proceso aplicado en el desarrollo de un lenguaje de programación.
- Intérprete de un lenguaje de programación denominado OMI, el cual da nombre al proyecto.
- Aplicación web que permite consultar la documentación e interactuar con el intérprete.

Este manual describe al lenguaje, las funciones que integra, y las características del intérprete. Además detalla cómo se estructura el contenido dentro de la aplicación web y las herramientas que la componen.

10.1. Características

OMI representa una plataforma constituida por una serie de herramientas. Estas ayudan al aprendizaje en la aplicación práctica de la teoría de autómatas y lenguajes formales para el desarrollo de un lenguaje de programación moderno. Sus características son las siguientes:

- Intérprete OMI
 - Abierto
 - Modular
 - Interactivo
 - Configurable
 - Paso de argumentos
 - Ejecución en modo servidor
 - Autodescriptivo.

- Lenguaje OMI

- Propósito general
- Interpretado
- Sintaxis simple y cercana a los lenguajes modernos
- Tipado dinámico y débil
- Tipos de datos simples y compuestos
- Referencia de datos
- Funciones y operadores sobre los distintos tipos de datos
- Sentencias de control
- Variable de ámbito global y local
- Definición de funciones
- Paso de parámetros por valor y por referencia
- Funciones de orden superior
- Clausura de funciones
- Funciones anónimas
- Aplicación parcial de funciones
- Decoradores
- Definición de clases de objetos
- Creación e instanciación de objetos
- Tipos de datos como clases de objetos
- Visibilidad de métodos y atributos
- Definición estática de métodos y atributos
- Polimorfismo
- Duck typing
- Herencia simple
- Métodos mágicos
- Dynamic binding
- Excepciones
- Evaluación por cortocircuito devolviendo último valor
- Operadores condicionales
- Funciones de fechas y tiempo

- Funciones de creación y acceso a ficheros
- Concurrente
- Recolector de basura

- Web OMI project.

- Contenido estático
- Acceso a la documentación del proyecto

- Navegación por las distintas clases y módulos que conforman el software
- Acceso a la descarga del software
- Información sobre el proyecto y la autoría
- Información de contacto
- runTree
 - Cliente OMI
 - Alta disponibilidad y facilidad de acceso
 - Interactivo
 - Interfaz gráfica
 - Descripción de diagramas
 - Descripción del árbol sintáctico producido
 - Descripción de las tablas de símbolos
 - Descripción de la entrada/salida producidos
 - Ayuda en pantalla

10.2. Requisitos previos

El intérprete OMI puede ser ejecutado sobre cualquier hardware actual. El software por si mismo solo necesita unos pocos kilobytes de memoria para funcionar, sin embargo el hardware necesario dependerá en gran medida del código fuente que será interpretado.

Para usar el intérprete OMI es necesario disponer de un sistema GNU/Linux. El intérprete depende de una serie de bibliotecas de programación. Dependiendo de la instalación que se realice será necesario la instalación previa de estas o no.

- readline
- boost-regex

Por otro lado para el correcto uso de la aplicación web se precisa de un navegador web con soporte HTML5 y JavaScript.

10.3. Uso del sistema

10.3.1. Obtener OMI

El intérprete OMI puede ser descargado desde la web OMI project (<http://omi-project.com/download>). Se ofrecen varias alternativas para la descarga:

- Código fuente.
- Precompilado y empaquetado para algunas distribuciones GNU/Linux.

Se recomienda la descarga de la última versión estable del sistema software.

10.3.2. Instalación

El intérprete OMI puede ser instalado sobre un sistema GNU/Linux mediante varios procesos.

10.3.2.1. Compilación e instalación desde código fuente

Para instalar OMI mediante el código fuente de la aplicación previamente se deberá llevar a cabo la compilación de este. Es posible obtener el código fuente de la aplicación desde la sección de descargas de la web OMI project.

El proceso de compilación se vale de una serie de herramientas que lo simplifican y parametrizan, permitiendo la configuración del ejecutable obtenido.

- autoconf
- automake
- make
- g++

Además se mantiene dependencia con las siguientes bibliotecas de programación:

- Biblioteca estándar de C/C++
- Biblioteca GNU readline
- Biblioteca Boost regex

Instalación de dependencias en una distribución Debian:

```
1 $PS1 apt-get install autoconf automake build-essential \
2 $PS1 libreadline-dev libboost-regex-dev
```

Listing 10.1: Instalación de dependencias

Para la compilación es necesario abrir una sesión de terminal de comandos en el directorio sobre el que se desea llevar a cabo el proceso. Se supone que el usuario tiene permisos suficientes en el sistema para llevar a cabo la instalación. Descarga del código fuente de la versión \$VERSION:

```
1 $PS1 wget http://www.omi-project.com/download/code/omi_$VERSION.tar.gz
```

Listing 10.2: Descarga de código fuente

Descompresión y desempaquetado del fichero:

```
1 $PS1 tar -xzvf omi_$VERSION.tar.gz
2 $PS1 cd omi_$VERSION
```

Listing 10.3: Descompresión y desempaquetado

Es posible establecer determinados aspectos del intérprete durante el proceso de compilación. Las opciones más significativas son las siguientes:

numType: Tipo de dato utilizado para la representación interna de los datos numéricos. El valor por defecto es “double”. Su valor puede ser cualquier tipo de dato estándar de C/C++ o algún tipo específico facilitado por el usuario. También admite el valor en número de bytes, siendo los posibles valores “4bytes”, “8bytes” y “16bytes”.

numPrecision: Precisión numérica utilizada en al escribir en la salida estándar. Su valor por defecto es 15, y puede ser cualquier número entero.

refCType: Tipo de dato utilizado para contar el número de referencias de los nodos. Determina el número de referencias máximas que puede llegar a tener un dato. Su valor por defecto “unsigned int”. Puede ser cualquier tipo de dato estándar de C/C++ o algún tipo específico facilitado por el usuario.

sizeNode: Esta opción permite configurar el intérprete dando valores óptimos a las opciones anteriores. Puede presentar uno de los siguientes valores:

small: numType=“float” refCType=“unsigned int” numPrecision=6

normal: numType=“double” refCType=“unsigned int” numPrecision=15

big: numType=“long double” refCType=“unsigned int” numPrecision=18

Para ver un listado completo de las opciones de configuración para la compilación:

```
| 1 $PS1 ./configure --help
```

Listing 10.4: Listado de las opciones de configuración para la compilación

Configurar el proceso de compilación con las opciones por defecto:

```
| 1 $PS1 ./configure
```

Listing 10.5: Proceso de compilación

Durante la ejecución de este script se comprobará si se satisfacen las dependencias, y se crearán los scripts necesarios para la compilación. Para llevar a cabo la compilación:

```
| 1 $PS1 make
```

Listing 10.6: Compilación

Una vez compilado con éxito es posible proceder con la instalación:

```
| 1 $PS1 make install
```

Listing 10.7: Instalación

Para comprobar la integridad de la instalación:

```
1 $PS1 make check
```

Listing 10.8: Comprobación de la integridad de la instalación

Para limpiar el entorno de compilación:

```
1 $PS1 make clean
```

Listing 10.9: Limpiar el entorno de compilación

Para desinstalar el software:

```
1 $PS1 make uninstall
```

Listing 10.10: Desinstalar el software

10.3.2.2. Instalación mediante paquete .deb

Cada versión del intérprete es precompilada y empaquetada. Las bibliotecas externas se compilan de forma estática por lo que se elimina la dependencia con otros paquetes.

Con una sesión de terminal y los permisos en el sistema suficientes es posible descargar e instalar el paquete .deb. Para descargar el paquete:

```
1 $PS1 wget http://www.omi-project.com/download/deb/omi_.$VERSION.deb
```

Listing 10.11: Descargar el paquete

Para instalar el paquete:

```
1 $PS1 dpkg -i omi_.$VERSION.deb
```

Listing 10.12: Instalar el paquete

Para desinstalar el paquete:

```
1 $PS1 dpkg -r omi
```

Listing 10.13: Desinstalar el paquete

10.3.3. Intérprete

El intérprete es el sistema software sobre el que gira el proyecto OMI. Se trata del software encargado de leer y ejecutar el código fuente escrito en OMI.

Para ejecutar el intérprete se utiliza el comando con el mismo nombre.

```
| 1 $PS1 omi
```

Listing 10.14: Ejecutar intérprete

Ejecutar el comando sin argumento hará que el intérprete lea y procese la entrada estándar.

La sintaxis del comando es la siguiente:

```
omi [options] [file] [args...]
```

El fichero será un documento escrito en el lenguaje OMI. La lista de argumentos serán datos accesibles desde el código fuente contenido en el fichero.

Las posibles opciones son:

- c cmd: Interpreta la cadena cmd.
- i: Modo interactivo. Las expresiones y sentencias son interpretadas en tiempo real y solicitadas mediante un prompt.
- h: Muestra la ayuda del comando.
- V: Muestra la versión del software.

Los argumentos facilitados serán accesible desde el código fuente mediante el array “args”, donde la primera posición del array es el nombre del fichero y las siguientes los argumentos facilitados. Se pone a disposición del usuario una hoja de manual que detalla el uso del comando.

```
| 1 $PS1 man omi
```

Listing 10.15: Uso del comando

Con la instalación por defecto el intérprete es ubicado en la siguiente ruta:

```
/usr/local/bin/omi
```

Además los recursos, como por ejemplo las extensiones, serán ubicados en el siguiente directorio:

```
/usr/local/share/omi
```

10.3.4. Referencias del lenguaje

10.3.4.1. Sentencias

Como muchos otros lenguajes de programación toda sentencia OMI debe acabar en punto y coma “;”.

Es posible delimitar un bloque de sentencias entre llaves. Muchos recursos del lenguaje operan o trabajan sobre un bloque de sentencias. Estas no necesitan acabar en “;”.

Los cierre de bloque o fin de la entrada automáticamente implica un punto y coma. En el modo interactivo no es necesario si la línea introducida sólo se compone de una sentencia.

```
1  << "Hola mundo"; // Sentencia acabada
2  if (true) { << "Y un ejemplo"; } // Bloque de sentencias
3  << "Soy un programa OMI"
```

Listing 10.16: Ejemplo sentencias

10.3.4.2. Comentarios

Los comentarios son fragmentos de texto que son incluidos en el código fuente y que serán omitidos por el intérprete. Se utilizan normalmente para aclarar y/o documentar las piezas de código.

En OMI es posible realizar un comentario de línea al estilo de C++ o de consola de comando UNIX.

```
1  x = 1; //x vale 1
2  y = 2; #y vale 2
```

Listing 10.17: Ejemplo comentario línea

Por otro lado, es posible escribir un bloque de una o varias líneas de comentarios delimitándolo mediante “/*” y “*/”.

```
1  /*
2   *      x vale 1
3   *      y vale 2
4  */
5  x = 1;
6  y = 2;
```

Listing 10.18: Ejemplo comentario bloque

10.3.4.3. Ejecución

La ejecución de un programa escrito en OMI comenzará desde un fichero, y desde la primera sentencia contenida en este. Las sentencias en este fichero que no estén contenidas en ninguna función o método son consideradas el flujo principal del programa.

La ejecución del programa normalmente se realiza sentencia a sentencia de forma secuencial. Algunas sentencias y expresiones del lenguaje podrán cambiar este flujo.

La ejecución finaliza cuando se llega al final de este fichero o se resuelve una sentencia que produzca la salida. También es posible que se de algún error sintáctico o semántico

10.3.4.4. Identificadores

Los identificadores en OMI sirven para nombrar elementos del script definidos por el programador, tales como variables, funciones, etiquetas, clases...

Un identificador debe comenzar por una letra (mayúscula o minúscula) o un subguión (_), seguidos de tantas letras, números o subguiones como se desee.

```
1  foo = 1; // foo identifica una variable que vale uno
2  ~ bar () { // bar identifica una función que imprime "Hola mundo".
3      << "Hola mundo";
4 }
```

Listing 10.19: Ejemplo identificadores

OMI es sensible a mayúsculas, por lo que considera dos identificadores distintos si no mantienen la misma capitalización.

Un identificador no puede coincidir con una palabra reservada del lenguaje o se producirá un error sintáctico.

10.3.4.4.1 Palabras reservadas

- null
- NULL
- true
- false
- global
- getenv
- function
- P
- return
- class
- new
- private
- extends
- this
- extends
- parent
- static
- include
- if
- else
- elseif
- elif
- switch
- case
- default
- with
- do
- while
- for
- as
- in
- break
- continue
- exit
- goto
- throw
- try
- catch
- print
- echo
- input
- inputline
- typeof
- typeOf
- datinfo
- datInfo
- empty
- isnull
- isbool
- isnum
- isstring
- isArray
- isobject
- AND
- and
- OR
- or

■ size	■ array_search	■ ftell
■ str_explode	■ int	■ fseek
■ str implode	■ float	■ fSET
■ array implode	■ bool	■ fCUR
■ str implode	■ string	■ fEND
■ str find	■ date	■ exec
■ str pos	■ time	■ eval
■ str replace	■ sleep	■ fork
■ str replace_sub	■ file	■ wait
■ str upper	■ fopen	■ getpid
■ str lower	■ fput	■ getppid
■ str search	■ fget	■ process
■ str match	■ fread	■ signal
■ sprintf	■ fwrite	■ kill
■ regexp	■ fappend	■ shandler
■ reduce	■ fapp	■ exit_process
■ array chunk	■ fclose	■ load

En OMI se nombran de igual forma las variables, funciones, clases... Es el contexto en el que se da un identificador el que determina el recurso que este nombra. Por regla general todos los operadores, funciones y demás construcciones del lenguaje toman, a no ser que se indique lo contrario, los identificadores como variables. Las excepciones a esta regla son:

- Operador llamada a función
- Operador aplicación parcial
- Operador *new*

OMI dispone de mecanismos para indicar explícitamente el tipo de recurso que es nombrado.

10.3.4.5. Tipos de datos

Haciendo uso del lenguaje OMI se puede operar sobre varios tipos de datos simples y compuestos. OMI es un lenguaje de programación que presenta un tipado dinámico, es decir, las variables se consideran del mismo tipo que al dato al que referencian. Así no es necesario declarar el tipo de dato que contendrán.

Los tipos de datos sobre los que opera OMI son:

- Simples:

- Booleanos.
 - Numéricos.
 - Cadenas de caracteres.
- Compuestos:
 - Arrays.
 - Objetos.
 - Especiales:
 - Nulo.
 - Función.
 - Expresión regular.

Algunas funciones y operadores del lenguaje tratan tipos de datos concretos. A pesar de ello OMI es un lenguaje débilmente tipado, por lo que es posible usar valores de un tipo de dato en expresiones que requieran otro tipo de dato. Para ello se lleva a cabo una conversión de tipo de forma implícita.

10.3.4.5.1 Booleano

Este tipo de dato puede contener dos únicos valores: verdadero o falso. Es el tipo de dato más simple. El valor verdadero se expresa mediante el literal “true”, mientras que el falso se representa mediante “false”.

```

1  a = true; // a presenta el valor verdadero
2  b = false; // b presenta el valor falso

```

Listing 10.20: Ejemplo de booleano

10.3.4.5.2 Numérico

En OMI únicamente existe un tipo de dato numérico. Mediante este tipo de dato es posible representar números enteros y aquellos decimales que puedan ser expresados en punto flotante. A pesar de que solo presenta un tipo de dato numérico ofrece mecanismos para operar con enteros y decimales, por ejemplo es posible obtener la parte entera de un número decimal.

Los números son expresados en OMI mediante cualquier combinación de dígitos, separando la parte decimal mediante un punto si existiera. Además es posible preceder la expresión de un signo “+” o “-”.

```

1  a = 1; // a presenta el valor 1
2  b = -1; // b presenta el valor -1
3  c = +1; // c presenta el valor +1
4  d = 0.001; // d presenta el valor 0.001
5  e = -0.001; // e presenta el valor -0.001

```

Listing 10.21: Ejemplo de datos numéricos

En OMI es posible configurar cómo los tipos de datos numéricos son almacenados y tratados de forma interna. Así es posible alterar el valor numérico máximo que se puede representar y la precisión. No obstante, esto sólo es posible en tiempo de compilación, por lo que se deberá usar la misma configuración para todos las ejecuciones.

10.3.4.5.3 Cadena de caracteres

En general, un carácter se refiere a una letra, número u otro símbolo o signo. Las cadenas de caracteres son una sucesión de estos. Normalmente representan un contenido textual.

En OMI las cadenas de caracteres son consideradas como un tipo de dato simple. Esto es debido a que no existe el tipo de dato carácter, y por tanto los datos de este tipo no puede ser descompuesto en datos de tipos más simples. De esta forma se considera al carácter simple como una cadena de un solo elemento.

Una cadena de caracteres queda delimitada entre comillas simples o comillas dobles.

```
1  a = "Una cadena de caracteres"; // a es una cadena definida entre "
2  b = 'Y otra'; // b es una cadena definida entre '
```

Listing 10.22: Ejemplo de cadena de caracteres

10.3.4.5.4 Array

En OMI un array es una estructura de datos que almacena valores de forma contigua. Los datos que guarda un array pueden ser de diferente tipo.

Se pueden dar dos tipos de arrays:

- Secuenciales.
- Asociativos.

Los arrays secuenciales son accesibles mediante índices numéricos y de una forma posicional. Se definen mediante una secuencia de expresiones que representan un valor. Estas serán separadas por el carácter “,” y delimitadas entre llaves “{}” y “{}”. El primer elemento del array se corresponde con el índice numérico 0.

```
1  a = {0,1,2}; // a es un array que contiene los números 0, 1 y 2
2  /*
3      b es un array que contiene el número 0, la
4      cadena "str" y otro array con los números 0 y 1
5  */
6  b = {0, "str", { 0, 1 } },
```

Listing 10.23: Ejemplo de arrays secuenciales

En arrays asociativos cada elemento está referenciado mediante una clave, de forma que se puede utilizar esta para acceder al elemento dentro del array. En OMI las claves de los arrays asociativos pueden ser cualquier expresión válida que represente un valor de tipo simple. Un array asociativo se define igual que uno secuencial salvo que los elementos que lo componen son

pares de expresiones correspondientes a la clave y al valor y separadas por el carácter “:”.

```
1  /*
2   a es un array asociativo con las cadenas
3   'k0', 'k1' y 'k2' como claves. Sus valores
4   son 0, 1 y 2 respectivamente.
5  */
6  a = { 'k0' : 0, 'k1' : 1, 'k2' : 2};
7  /*
8   b es un array asociativo con los números 0 y
9   10, y la cadena "K" como claves. Sus valores son
10  el número 0, la cadena "str" y otro array con
11  los números 0 y 1.
12 */
13 b = { 10: 0, 0: "str", "K": { 0, 1 } };
```

Listing 10.24: Ejemplo de arrays asociativos

10.3.4.5.5 Objetos

En programación orientada a objetos, un objeto es una estructura de datos que presenta un comportamiento asociado. Tiene un estado interno determinado por el valor de los datos que contiene, y se encarga de realizar una serie de tareas en tiempo de ejecución. Una clase de objetos es un modelo que define las propiedades y el comportamiento que tienen objetos afines. El uso de las clases y los objetos es descrito en profundidad como una referencia del lenguaje ([Subsubsección 10.3.4.13](#)).

En OMI todo dato es considerado de tipo objeto, los otros tipos se ven como clases de objetos. Un dato, además de tener asociado un valor de un tipo concreto, presenta una serie de acciones u operaciones que se pueden realizar con él y que están determinadas por su tipo. Por ejemplo sobre un dato array se puede realizar la operación implode para obtener una cadena con todos los elementos del array separado por una subcadena separadora.

```
1  /*
2   A la variable "a" se le asigna
3   el resultado de ejecutar la operación
4   implode sobre el array.
5  */
6  a = { "hola", "mundo"}->implode(" ");
```

Listing 10.25: Ejemplo array como objeto

OMI permite al usuario definir, construir y utilizar sus propios objetos. Para ello se vale de las clases y de mecanismos para instanciar los objetos a partir de estas.

10.3.4.5.6 Nulo

Nulo representa la ausencia de valor. Es un tipo de dato especial que se utiliza para expresar referencias que no tienen valor asociado. Paradójicamente el valor nulo conlleva la ausencia de valor.

En OMI cualquier referencia sin valor se considera como nulo. Las variables no inicializadas, o las posiciones de arrays inexistentes tienen el valor nulo.

Es posible expresar un dato nulo mediante el literal “null” (o en mayúsculas “NULL”).

```
1 a = null; // a no contiene valor
```

Listing 10.26: Ejemplo de nulo

10.3.4.5.7 Función

En OMI una función hace referencia a un bloque de sentencias que puede recibir una serie de parámetros y tomar un valor. Una función realiza una tarea específica dentro del programa. Cuando una función es llamada se ejecuta el bloque de sentencias al que referencia, pasándole como parámetros los valores en la llamada.

El uso de las funciones es descrito en detalle más adelante, en este manual, como una referencia del lenguaje ([Subsubsección 10.3.4.13](#)).

En OMI las funciones son consideradas tipos de datos, siendo posible operar sobre ellas. Así una función puede ser asignada a una variable, pasada como parámetro o ser devuelta por otra función.

```
1 /*
2      a es una función que recibe un
3      parámetro y lo devuelve como valor
4 */
5 a = ~(param){ return param; };
```

Listing 10.27: Ejemplo función

10.3.4.5.8 Expresión regular

Una expresión regular es un patrón definido por un lenguaje regular. Normalmente se aplica a una cadena de caracteres, para determinar si pertenece al lenguaje que define. Se utiliza para realizar búsquedas de subcadenas o llevar a cabo reemplazos.

Con OMI se puede expresar cualquier expresión regular usando la sintaxis PERL. Para ello se ha de delimitar entre acento grave (`).

```
1 /*
2      "exp" es la expresión regular correspondiente
3      al lenguaje descrito por cero o mas caracteres "a"
4      seguidos de dos caracteres "b"
5 */
6 exp = `a*bb`;
```

Listing 10.28: Ejemplo de expresión regular

El lenguaje dispone de funciones para crear y operar sobre expresiones regulares.

10.3.4.5.9 Conversión implícita entre tipos de datos

OMI es un lenguaje de tipado débil, es decir, dado un valor de un tipo concreto es posible utilizarlo como otro tipo sin necesidad de llevar a cabo una conversión de forma explícita. Al usar un dato en una operación que espera otro tipo OMI realizará una conversión automática.

Nulo

Booleano: false

Numérico: 0

Cadena de caracteres: ""

Booleanos

false

Numérico: 0

Cadena de caracteres: ""

true

Numérico: 1

Cadena de caracteres: "1"

Numéricos

Número 0

Booleano: false

Cadena de caracteres: "0"

Número distinto a 0

Booleano: true

Cadena de caracteres: Cadena que representa el número.

Cadena de caracteres

Cadena vacía

Booleano: false

Numérico: 0

Cadena numérica

Booleano: true

Numérico: Número que representa la cadena

Otra cadena

Booleano: true

Numérico: Tamaño de la cadena

Normalmente no es posible usar un dato simple en lugar de uno compuesto. Las funciones y operaciones que esperan recibir como argumento un dato compuesto devolverán un error si no es así. No obstante un dato compuesto puede utilizarse como uno simple sin que se produzcan errores semánticos.

Array

Array vacío

Booleano: false

Numérico: 0

Cadena de caracteres: ""

Array con elementos

Booleano: true

Numérico: 0

Cadena de caracteres: ""

En el caso de los objetos, es posible definir métodos que serán invocados cuando sean utilizados como otro tipo de dato. Estos métodos son denominados métodos mágicos (Párrafo 10.3.4.13.7).

10.3.4.6. Operadores

OMI soporta una serie de operadores de naturaleza muy variada. Mediante un operador se puede construir una expresión que, al ser evaluadas, se aplicará una función sobre un conjunto de datos de entrada denominados operandos, y obtener así un valor.

Un operador normalmente es de naturaleza binaria, presentando dos entradas, aunque existe operadores unitarios o ternarios. Además presenta una sintaxis y semántica propia.

```
1 foo = 1 + 1; // foo es el resultado de evaluar la operación suma
```

Listing 10.29: Ejemplo de operador

Los operadores pueden combinarse para formar expresiones más complejas. Para la evaluación se establece una precedencia de operadores implícita, que determina en qué orden deben de resolverse. Es posible alterar la precedencia de operadores encerrando la expresión entre paréntesis.

```
1 foo = 2 * 8 + 3; // se evalúa la operación producto y luego la suma.
2 bar = 2 * (8 + 3); // se evalúa la operación suma y luego el producto.
```

Listing 10.30: Ejemplos combinación de operadores

La mayoría de operadores que se pueden utilizar en OMI tienen una naturaleza matemática y se resuelven igual forma. No obstante también soporta otros operadores propios de los lenguajes de programación.

10.3.4.6.1 Operadores lógicos

Al aplicarse un operador lógico se produce un resultado booleano derivado de la evaluación de una condición.

Los operadores lógicos son los siguientes:

Negación: !

AND lógico: &&, and

OR lógico: `||`, `or`

```
1  foo = !true; // foo vale falso
2  foo = true && false; // foo vale falso
3  foo = true or false; // foo vale verdadero
```

Listing 10.31: Ejemplos de operadores lógicos

Los operadores AND y OR son evaluados de izquierda a derecha en corto circuito. Se puede producir el resultado mediante una evaluación parcial, al conocerse de antemano el valor que se obtendrá mediante la evaluación total. En este caso el resto de la expresión no será evaluada. Además el resultado siempre será el último operando en ser evaluado.

```
1  foo = null and 3 + 4; // foo vale null, evaluación parcial
2  foo = {} or "10"; // foo vale "10", evaluación total
3  foo = "10" and 3 + 4; // foo vale 7, evaluación total
4  foo = "5" or {}; // foo vale "5", evaluación parcial
```

Listing 10.32: Ejemplos de operadores lógicos en cortocircuito

10.3.4.6.2 Operadores de comparación

Se encargan de evaluar el orden entre dos valores. El resultado es un valor booleano de verdadero si se satisface la comparación o falso en caso contrario.

Los operadores de comparación son los siguientes:

Igualdad: `==`

Desigualdad: `!=`

Menor que: `<`

Menor o igual que: `<=`

Mayor que: `>`

Mayor o igual que: `>=`

Idéntico: `====`

No idéntico: `!====`

```
1  foo = 5 == 5; // foo vale verdadero
2  foo = "5" == 5; // foo vale verdadero
3  foo = 5 != 5; // foo vale falso
4  foo = "3" < 4; // foo vale verdadero
5  foo = "3" <= 4; // foo vale verdadero
6  foo = 4 > 4; // foo vale falso
7  foo = 4 >= 4; // foo vale verdadero
8  foo = 5 === 5; // foo vale verdadero
9  foo = "5" === 5; // foo vale falso
10 foo = "5" !== 5; // foo vale verdadero
```

Listing 10.33: Ejemplos de operadores de comparación

Para los tipos booleanos se considera que verdadero (true) es mayor que falso (false). Para los datos de tipo numéricos se toma su valor aritmético. Las cadenas de caracteres son ordenadas alfabéticamente.

Cuando se comparan datos de distintos tipos se realiza una conversión al tipo de dato más simple. Por ejemplo al comparar un dato numérico con un booleano se convertirá el numérico a booleano.

```
1 foo = true <= 5; // foo vale verdadero
2 foo = "5" == true; // foo vale verdadero
3 foo = "word" < 4; // foo vale falso
```

Listing 10.34: Ejemplo de conversión implícita

Cuando se realiza una comparación de forma idéntica, además de comparar el valor de los datos, se comprueban los tipos.

10.3.4.6.3 Operadores aritméticos

Se corresponden con las operaciones aritméticas básicas que son la suma, la diferencia, el producto y la división, y aquellas derivadas de estas. El resultado de la operación será un número.

Los operadores aritméticos toman como operandos expresiones tratadas como numéricos. Todo dato que sea sometido a una operación aritmética será convertido a numérico.

Los operadores aritméticos soportados por OMI son:

Suma: +

Diferencia: -

Producto: *

División: /

Módulo: %

Potencia: ^

```
1 foo = 1 + 1; // foo vale 2
2 foo = "5" + 5; // foo vale 10
3 foo = "word" + 4; // foo vale 8
4 foo = true + 10; // foo vale 11
5 foo = 4 - "2"; // foo vale 2
6 foo = 3 * 4; // foo vale 12
7 foo = 17 / 4; // foo vale 4.25
8 foo = 2.1 / "8"; // foo vale 0.2625
9 foo = 4 % 3; // foo vale 1
10 foo = 4 ^ 2; // foo vale 16
```

Listing 10.35: Ejemplos de operadores aritméticos

El resultado del operador módulo mantendrá el mismo signo que el operando que hace de dividiendo.

10.3.4.6.4 Operadores sobre cadenas de caracteres

OMI soporta una serie de operadores que pueden ser aplicados sobre cadenas de caracteres.

Concatenación: .

Acceso a posición: [n]

```
1   foo = "ABC" . "DEF"; // foo vale "ABCDEF"
2   foo = "ABCDE"[0]; // foo vale "A"
```

Listing 10.36: Ejemplos de operadores sobre cadena de caracteres

La concatenación se realiza después de que se determinen los números decimales presentes en la expresión. Todo carácter punto no suponga un separador decimal será considerado un operador de concatenación.

```
1   foo = "AB".2.1.1."CD"; // foo vale "AB2.11CD"
2   foo = "AB".2.1.1.1."CD"; // foo vale "AB2.11.1CD"
```

Listing 10.37: Ejemplos de operadores de concatenación

El operador de acceso a posición recibe un índice que indica el elemento al que acceder dentro de la cadena. Si la posición indicada no existe se toma el valor nulo. El índice siempre será considerado como un dato numérico.

El operador de acceso también puede ser utilizado para escribir en una determinada posición de la cadena. No obstante, en este caso, debe existir un elemento en la posición indicada o se producirá un error.

```
1   foo = "ABCD";
2   foo[0] = "E"; // foo vale "EBCD"
3   foo[0] = "FG"; // foo vale "FGBCD"
4   foo[10] = "X"; // se produce un error.
```

Listing 10.38: Ejemplos de operador de acceso

No es posible utilizar el operador de acceso para escribir en una determinada posición de una cadena constante.

```
1   "ABCD"[0] = "X"; // se produce un error.
```

Listing 10.39: Ejemplo de error de un operador de acceso

El operador de acceso es utilizado para otros tipos de datos como los arrays.

10.3.4.6.5 Operadores de asignación

La asignación es la operación mediante la cual el valor de una expresión puede ser atribuido a un símbolo variable. De esta forma se puede hacer uso del valor en otras expresiones. El operador se

representa mediante un signo igual “=” y toma una expresión (operando derecho) para asignarla a un símbolo variable (operando izquierdo).

En OMI se lleva a cabo una asignación destructiva. Cuando una expresión es asignada, por ejemplo a una variable, el valor de esta última cambia, indistintamente de lo que contenía anteriormente.

```
1 foo = 1; // foo vale 1
2 foo = 2; // foo a cambiado al valor 2
3 bar = foo; // bar vale el valor actual de foo que es 2
4 foo = 10; // foo a cambiado al valor 10 y bar sigue valiendo 2
```

Listing 10.40: Ejemplos de operadores de asignación

OMI, como otros lenguajes, soporta una serie de operadores que combinan una operación determinada y una asignación. Estos simplifican muchas expresiones.

Suma y asignación: `+ =`

Diferencia y asignación: `- =`

Producto y asignación: `* =`

División y asignación: `/ =`

Potencia y asignación: `^ =`

Módulo y asignación: `% =`

Concatenación y asignación: `. =`

```
1 foo = 1; // foo vale 1
2 foo += 2; // foo a cambiado al valor 3
3 foo -= 4; // foo a cambiado al valor -1
4 foo *= -9; // foo a cambiado al valor 9
5 foo /= 3; // foo a cambiado a 3
6 foo ^= 3; // foo a cambiado a 27
7 foo %= 2; // foo a cambiado a 1
8 foo .= "F"; // foo a cambiado a "1F"
```

Listing 10.41: Ejemplos de operación y asignación

10.3.4.6.6 Operadores de incremento y decremento

Estos operadores se aplican a una expresión variable, incrementando o decrementando su valor numérico. Este tipo de operaciones implican una asignación.

Se distinguen entre si, además de que si el valor es incrementado o decrementado, en el momento en el momento en el que se tomará el valor. Pudiendo aplicarse la operación antes o después de evaluar el resto de la expresión.

Preincremento: `++ i`

Posincremento: `i ++`

Predecremento: $--i$

Posdecremento: $i--$

```
1  foo = 1; // foo vale 1
2  bar = ++foo; // bar vale 2 y foo vale 2
3  bar = foo++; // bar vale 2 y foo vale 3
4  bar = --foo; // bar vale 2 y foo vale 2
5  bar = foo--; // bar vale 2 y foo vale 1
```

Listing 10.42: Ejemplo de incremento y decremento

10.3.4.6.7 Operadores de conversión de tipo

En OMI se definen una serie de operadores que permiten realizar una conversión explícita a un determinado tipo de dato.

Booleano: *bool*

Entero: *int*

Punto flotante: *float*

Cadena de caracteres: *string*

```
1  foo = bool 4; // foo vale el booleano true
2  foo = bool 0; // foo vale el booleano false
3  foo = int "4.5"; // foo vale el numérico 4
4  foo = int "AA"; // foo vale el numérico 2
5  foo = float "4.5"; // foo vale el numérico 4.5
6  foo = string 55; // foo vale la cadena "55"
7  foo = string true; // foo vale la cadena "1"
```

Listing 10.43: Ejemplos de operadores de conversión explícita

A pesar de que OMI soporta la conversión a enteros o flotantes, internamente estos se representan de la misma forma. Por lo que se consideran del mismo tipo.

10.3.4.6.8 Operadores sobre arrays

El lenguaje OMI soporta una serie de operadores que pueden ser aplicados sobre un dato array.

Acceso a posición: $[k]$

Acceso a final: $[]$

Tamaño: *size*

```
1  foo = {"A", "B"}; // foo es un array que contiene las cadenas "A" y "B"
2  bar = {"A", "B"}[0]; // bar vale "A"
3  bar = foo[1]; // bar vale "B"
4  bar = size foo; // bar vale 2
5  foo[] = "C"; // foo es un array que contiene las cadenas "A", "B" y "C"
6  bar = size foo; // bar vale 3
7  foo = { "k0": "v0", "k1": "v1" }; // foo es un array asociativo con dos claves
```

```

8  bar = { "k0": "v0", "k1": "v1" }["k0"]; // bar vale "v0"
9  bar = foo["k1"]; // bar vale "v1"
10 bar = size foo; // bar vale 2
11 foo["k2"] = "v2"; // foo es un array asociativo con tres claves
12 foo[] = "v3"; // foo es un array asociativo con cuatro claves

```

Listing 10.44: Ejemplo de operadores sobre array

Con el operador de acceso a posición es posible acceder a un determinado elemento dentro del array. Para ello se debe facilitar el índice o la clave a la que se desea acceder.

El operador de acceso a posición en array no realiza ninguna conversión de tipos sobre el índice o clave. Las claves son comparadas con el operador de igualdad (Párrafo 10.3.4.6.2).

Si se accede mediante un índice o clave que no existe en el array se toma el valor nulo.

El operador de acceso a última posición por si solo toma el valor nulo, pero puede ser utilizado para añadir elementos al final del array.

No es posible asignar a posiciones de un array constante.

```

1  {1,2}[0] = 10; // se produce un error
2  {"key": "val"}["key"] = "other"; // se produce un error
3  {1,2}[] = 10; // se produce un error

```

Listing 10.45: Ejemplo de errores sobre operadores de array

El operador de acceso es utilizado en otros tipos de datos como las cadenas de caracteres.

10.3.4.6.9 Operadores sobre clases y objetos

OMI es un lenguaje de programación orientado a objetos, por lo que pone a disposición del usuario un conjunto de operadores relacionados con estos.

Acceso a atributos y métodos: ->

Instanciación de clases: *new*

Acceso al objeto en ejecución: *this*

Acceso al objeto padre en ejecución: *parent*

Acceso a la clase en ejecución: *static*

El uso de las clases y objetos quedan descritos en la sección correspondiente (Subsubsección 10.3.4.13).

10.3.4.6.10 Operadores sobre funciones

OMI define una serie de operadores que son aplicables a funciones.

Llamada a función: *(p0,...)*

Aplicación parcial: *P[v1,...]*

Acceso a función de contexto: *~>*

El uso de las funciones queda descritos en la sección correspondiente (Subsubsección 10.3.4.11).

10.3.4.6.11 Operadores comprobación de tipos

En OMI existen una serie de operadores encargados de comprobar el tipo de un dato dado. Estos devolverán un valor booleano que dependerá de que el dato sea del tipo indicado o no.

Comprobación de nulo: *isnull*

Comprobación de booleano: *isbool*

Comprobación de numérico: *isnum*

Comprobación de cadena: *isstring*

Comprobación de array: *isarray*

Comprobación de objeto: *isobject*

```
1  foo = isnull null; // foo es true
2  foo = isnull 4; //foo es false
3  foo = isbool true; //foo es true
4  foo = isbool false; //foo es true
5  foo = isbool 5; //foo es false
6  foo = isnum 4; // foo es true
7  foo = isnum "4"; // foo es false
8  foo = isnum 4.6; // foo es true
9  foo = isstring "4"; // foo es true
10 foo = isstring 4; // foo es false
11 foo = isarray {}; // foo es true
12 foo = isarray {1,2}; // foo es true
13 foo = isarray "{1,2}"; // foo es false
14 foo = isobject new Class1(); // foo es true
15 foo = isobject 5; // foo es false
```

Listing 10.46: Ejemplo de operadores de comprobación de tipos

10.3.4.6.12 Operadores condicionales

Los operadores condicionales evalúan una expresión que hace de condición, y en función del resultado obtenido realizan una determinada operación.

Operador ternario: *cond ? op1 : op2*

Fusión de nulos: *[[op1, op2, ..., opn]]*

En el operador ternario si la expresión de condición se cumple se toma el valor del operando precedido por "?", en caso contrario se toma el del último, que es precedido por ":".

```
1  foo = (4 == 4) ? "OP1": "OP2"; // foo vale "OP1"
2  foo = (4 != 4) ? "OP1": "OP2"; // foo vale "OP2"
3  foo = 5? "OP1": "OP2"; // foo vale "OP1"
```

Listing 10.47: Ejemplo de operador ternario

Existe distintas formas del operador ternario además de la descrita. El operador ternario simplificado en valor verdadero omite el operando que se devolverá en el caso de que la condición

sea cierta, en este caso se devolverá el valor de la expresión de condición. Por otro lado existe el ternario simplificado en valor falso, que omite el último operando y que tomará el valor nulo.

```
1  foo = 5 ?: "OP2"; // foo vale 5
2  foo = 0 ?: "OP2"; // foo vale "OP2"
3  foo = "COND"? "OP1"; // foo vale "OP1"
4  foo = ""? "OP1"; // foo vale null
```

Listing 10.48: Ejemplo de operador ternario simplificado

La fusión de nulos opera sobre una lista de operandos. El operador comprueba los elementos secuencialmente y devuelve el primero que no sea nulo o nulo si todos los son.

```
1  foo = [[ null, null, 4 ]]; // foo vale 4
2  foo = [[ null, 4, 5 ]]; // foo vale 4
3  foo = [[ null, null, null ]]; // foo vale null
```

Listing 10.49: Ejemplo de operador fusión de nulos

10.3.4.6.13 Operadores de entrada/salida

OMMI presenta una serie de operadores que permiten leer de la entrada estándar o escribir en la salida estándar.

Entrada estándar: `>>, input`

Salida estándar: `<., <<, echo`

El operador de entrada estándar tiene como operando una expresión variable. Este se encarga de leer un flujo de caracteres de la entrada estándar hasta que lee un carácter fin de línea. El conjunto de caracteres leídos es asignado a la variable como una cadena.

```
1  >> foo; // Se lee de la entrada estándar hasta un carácter fin de linea y se introduce
   en foo
2  input foo; // Igual que el caso anterior
```

Listing 10.50: Ejemplo de operador de entrada

El operador de entrada tiene una segunda forma en la que se puede indicar una cadena de caracteres que se mostrará como prompt.

```
1  /*
2   * Se lee de la entrada estándar hasta un carácter fin de linea
3   * y se introduce en foo. Para indicar que se espera una entrada
4   * se utiliza la cadena "Example:"
5  */
6  >>["Example:"] foo;
```

Listing 10.51: Ejemplo de operador de entrada con prompt

Los operadores de salida estándar se encargan de escribir en esta una cadena de caracteres. Estos operadores se diferencian entre si, además de en su forma léxica, en si escriben un carácter de fin de línea automático o no.

```

1 echo "Hola mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
2 <. "Hola mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
3 << "Hola mundo"; // Escribe la cadena "Hola mundo" en la salida estándar y un salto de
    linea

```

Listing 10.52: Ejemplo de operadores de salida

Es posible utilizar los operadores `< .` y `<<` para concatenar un flujo de cadenas de caracteres en la salida estándar. En el caso del operador `< .` se resolverá como el operador de concatenación `.,` introduciendo las cadenas una seguida a la otra. Por otro lado, usar el operador `<<` implica que se va a concatenar un salto de línea antes que la cadena

```

1 echo "Hola" . "mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
2 echo "Hola" <. "mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
3 echo "Hola" << "mundo"; // Escribe la cadena "Hola \mundo" en la salida estándar
4
5 <. "Hola" . "mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
6 <. "Hola" <. "mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
7 <. "Hola" << "mundo"; // Escribe la cadena "Hola \mundo" en la salida estándar
8
9 << "Hola" . "mundo"; // Escribe la cadena "Hola mundo\n" en la salida estándar
10 << "Hola" <. "mundo"; // Escribe la cadena "Hola mundo\n" en la salida estándar
11 << "Hola" << "mundo"; // Escribe la cadena "Hola \mundo\n" en la salida estándar

```

Listing 10.53: Ejemplo de operadores de salida para concatenación

10.3.4.6.14 Precedencia de operadores

Una expresión puede estar compuesta por varios operadores, se establece pues prioridades entre estos que determinan cómo ha de resolverse la expresión.

Los operadores quedan ordenados en el siguiente listado de más prioritarios a menos:

Llamadas: `(p0,..), new`

Operadores de acceso: `[]`, `->`, `~>`, `this`, `static`

Incrementos y decrementos: `++`, `--`

Conversión de tipos: `bool`, `int`, `float`, `string`

Operadores condicionales: `[[op1,...]]`, `? :`

Operaciones aritméticas de primer nivel: `%`, `^`

Operaciones aritméticas de segundo nivel: `*`, `/`

Operaciones aritméticas de tercer nivel: `+`, `-`

Operaciones comparación: ==, !=, <, <=, >, >=, ===, !==

Operaciones lógicas de primer nivel: !

Operaciones lógicas de segundo nivel: &&

Operaciones lógicas de tercer nivel: ||

Asignaciones: =, +=, -=, *=, /=, %=, .=, ^=

Los operadores en el mismo nivel de prioridad son resueltos mediante una asociación desde la izquierda, excepto los operadores de asignación que son asociativos desde la derecha.

10.3.4.7. Etiquetas

Es posible etiquetar sentencias dentro del código fuente para poder cambiar el flujo de ejecución al punto que esta ocupa.

Una sentencia está formada por un identificador seguido del carácter ':' y la sentencia etiquetada.

```
1  label: << "Sentencia";
```

Listing 10.54: Ejemplo de etiquetas

Las etiquetas normalmente son referenciadas por la sentencia de control *goto* la cual cambia el flujo de ejecución a la sentencia etiquetada.

Normalmente se desaconseja el uso de etiquetas y sentencias *goto* dado que dificultan la legibilidad del código. No obstante OMI soporta este mecanismo de control.

10.3.4.8. Sentencias de control

Las sentencias de control son construcciones del lenguaje que permiten alterar el flujo de ejecución del programa. Son muy comunes y utilizadas en los lenguajes de programación.

Se clasifican según su naturaleza, así es posible ver sentencias de control condicionales, iterativas, de salto, inclusivas o de excepción.

Muchas sentencias de control están formadas por bloques de sentencias cuya ejecución controlan. Estos bloques pueden estar formados por una sola sentencia.

10.3.4.8.1 Sentencia condicional *if... else...*

La sentencia condicional *if... else...* está formada por una expresión de condición, además de dos bloques de sentencias que serán ejecutadas en función la condición sea evaluada como verdadera o falsa.

```
1  >>["Dime el valor de foo:"] foo;
2  if ( foo < 10) {
3      << foo << " es menor que 10"; // Se ejecuta si la condición es verdadera
4  }
5  else {
6      << foo << " no es menor que 10"; // Se ejecuta si la condición es falsa
7  }
```

Listing 10.55: Ejemplo if...else...

Si el caso de evaluación negativa no implica la ejecución de ninguna sentencia, el bloque *else* puede omitirse.

```
1  >>["Dime el valor de foo:"] foo;
2  if (foo < 10) {
3      << foo << " es menor que 10"; // Se ejecuta si la condición es verdadera
4 }
```

Listing 10.56: Ejemplo if...

Otra forma de esta sentencia condicional es *if... elif... else...*, en donde se anidan varias sentencias de este tipo. Se pueden anidar tantos *elif* como sea necesario.

```
1  >>["Dime el valor de foo:"] foo;
2  if (foo < 10) {
3      << foo << " es menor que 10"; // La primera condición es verdadera
4  }
5  elif (foo == 10) {
6      << foo << " es igual que 10"; // La primera es falsa y la segunda verdadera
7  }
8  else {
9      << foo << " es mayor que 10"; // Todas las condiciones son falsas
10 }
```

Listing 10.57: Ejemplo if ... elif...else...

10.3.4.8.2 Sentencia condicional *switch... case...*

La sentencia *switch... case...* agiliza la toma de decisiones múltiples. Opera de igual forma que varios *if... else...* anidados, pero presenta una sintaxis que en muchos casos favorece la legibilidad y la rapidez de programación.

Se compone de un dato a comparar y un bloque de sentencias etiquetadas con casos. Los casos están formados por una expresión que será comparada con el dato. La ejecución pasará a la primera sentencia etiquetada con el caso que sea igual al dato comparado, procediéndose a ejecutar todas las sentencias siguientes.

```
1  >>["Sentencia inicial:"] foo;
2  switch (foo) {
3      case 0:
4          << "Sentencia 0"; // Se ejecuta si foo es 0
5      case 1:
6          << "Sentencia 1"; // Se ejecuta si foo es 0 o 1
7      case 2:
8          << "Sentencia 2"; // Se ejecuta si foo es 0, 1 o 2
9  }
```

Listing 10.58: Ejemplo switch

Es una práctica muy común, para omitir la ejecución de algunas sentencias, el utilizar la sentencia *break*, la cual finaliza la ejecución del bloque de sentencias actual.

```

1  >>["Sentencia inicial:"] foo;
2  switch (foo) {
3      case 0:
4          << "Sentencia 0"; // Se ejecuta si foo es 0
5          break;
6      case 1:
7          << "Sentencia 1"; // Se ejecuta si foo es 1
8          break;
9      case 2:
10         << "Sentencia 2"; // Se ejecuta si foo es 2
11     }

```

Listing 10.59: Ejemplo switch break

Si ningún caso es igual al dato comparado no se ejecuta ninguna sentencia del bloque, a menos que se encuentre la etiqueta *default*.

```

1  >>["Sentencia a ejecutar:"] foo;
2  switch (foo) {
3      case 0:
4          << "Sentencia 0"; // Se ejecuta si foo es 0
5          break;
6      case 1:
7          << "Sentencia 1"; // Se ejecuta si foo es 1
8          break;
9      case 2:
10         << "Sentencia 2"; // Se ejecuta si foo es 2
11         break;
12     default:
13         << "Sentencia por defecto"; // Se ejecuta si foo no es 0, 1 ni 2
14   }

```

Listing 10.60: Ejemplo switch default

10.3.4.8.3 Sentencia iterativa *while*...

La sentencia *while*... es una estructura de control iterativa que permite la ejecución de un bloque de sentencias repetidamente mientras que se cumpla una determinada expresión de condición. Se compone pues de una expresión que será evaluada como un dato booleano y un bloque de sentencias.

```

1  /*
2   * Imprime la cadena
3   * "Sentencia" 10 veces.
4  */
5  foo = 0;
6  while (foo < 10) {
7      << "Sentencia";
8      foo++;
9  }

```

Listing 10.61: Ejemplo while

La expresión de condición es evaluada al comienzo de cada ejecución del bloque de sentencias, por lo que si alguna sentencia vuelve la condición falsa se seguirá con la ejecución del bloque hasta la próxima iteración.

10.3.4.8.4 Sentencia iterativa *do... while*

La sentencia *do... while* es una estructura de control iterativa que permite la ejecución de un bloque de sentencias repetidamente mientras que se cumpla una determinada expresión de condición, y al menos una vez. Se compone pues de una expresión que será evaluada como un dato booleano y un bloque de sentencias.

```
1  /*
2   *      Imprime la cadena
3   *      "Sentencia" 10 veces.
4   */
5   foo = 0;
6   do {
7       << "Sentencia";
8       foo++;
9   } while (foo < 10);
```

Listing 10.62: Ejemplo *do...while*

A diferencia de la sentencia *do... while* la condición es evaluada al final de la ejecución del bloque de sentencias por lo que se asegura que este se ejecutará al menos una vez. Al igual que *do... while* si alguna sentencia vuelve falsa la condición se proseguirá con la ejecución hasta que se produzca la evaluación.

10.3.4.8.5 Sentencia iterativa *for...*

La sentencia *for...* es una estructura de control iterativa que permite la ejecución de un bloque de sentencias repetidamente mientras que se cumpla una determinada expresión de condición. La sentencia *for...* conlleva la ejecución de la expresión de inicialización al inicio de la sentencia, y de una expresión de iteración al final de cada ejecución del bloque.

```
1  /*
2   *      Imprime la cadena
3   *      "Sentencia" 10 veces.
4   */
5   for (foo = 0; foo < 10; ++foo){
6       << "Sentencia";
7   }
```

Listing 10.63: Ejemplo *for...*

Es OMI es necesario especificar todas las expresiones que conforman la sentencia *for...*, si alguno es omitido se producirá un error sintáctico.

10.3.4.8.6 Sentencia iterativa *for... as... y for... in...*

OMI ofrece unas formas de sentencias *for...* que permiten ejecutar un bloque de sentencias para cada elemento contenido en una determinada expresión. Para ello se debe especificar la

expresión a recorrer, el símbolo variable con el que se referenciará al elemento actual y el bloque de sentencias que se ejecutará para cada elemento contenido en la expresión.

```

1  /*
2   * Imprime la cadena
3   * "Sentencia" y su índice
4   * 10 veces.
5  */
6  for ( 10 as i ){
7      << "Sentencia" . i;
8  }
9  /*
10    Imprime la cadena
11    "Sentencia" y su índice
12    10 veces.
13 */
14 for ( i in 10 ){
15     << "Sentencia" . i;
16 }
```

Listing 10.64: Ejemplo for... conjunto

La diferencias entre *for... as...* y *for... in...* son puramente sintácticas, siendo el orden en que se especifican la expresión conjunto y el símbolo variable su principal diferencia.

El comportamiento de estas estructuras cambia en función el tipo de dato de la expresión conjunto.

Booleano: Si es true se ejecutará el bloque indefinidamente y el símbolo variable tendrá el valor true. Si es falso el bloque de sentencias no se ejecutará.

Numéricos: Si es mayor que se 0 el bloque se ejecutará desde 0 hasta el valor de la expresión y el símbolo variable tendrá el valor numérico de cada iteración. En otro caso no se llega a ejecutar el bloque.

Cadenas de caracteres: Se ejecuta el bloque de sentencias por cada carácter en la cadena, el símbolo variable tomará como valor la cadena correspondiente al carácter de cada iteración.

Array: Se ejecuta el bloque de sentencias por cada elemento en el array, el símbolo variable tomará como valor el elemento de cada iteración.

Es posible especificar un par de símbolos variables separados por el carácter “`:`”. Esto hará que si el elemento actual en el recorrido tiene una clave que lo referencia esta se guarde en el primero y el valor en el segundo. Si no tuviera clave solo se asignará el valor al segundo símbolo variable.

```

1  /*
2   * Imprime todas las
3   * claves y valores del array
4  */
5  array = { 'k0': 'v0', 'k1': 'v1', 'k2': 'v2'};
6  for ( array as key:value ){
7      << key . " => " . value;
8  }
```

Listing 10.65: Ejemplo for... conjunto con clave

10.3.4.8.7 Sentencia iterativa ágil

OMI ofrece una estructura de control iterativa cuya sintaxis ha sido simplificada, su funcionalidad extendida. La sentencia de control de iteración ágil funciona igual que una sentencia *for... in...*, pero sin que se produzca una asignación de cada elemento a un símbolo variable. En su lugar se puede acceder al elemento en curso mediante un operador de acceso especial.

La sentencia de iteración ágil se compone de una expresión a recorrer y de un bloque de sentencias que se ejecutará para cada elemento en la expresión.

```
1  /*
2   *      Imprime la cadena
3   *      "Sentencia" 10 veces.
4   */
5  $(10){
6      << "Sentencia";
7 }
```

Listing 10.66: Ejemplo iteración ágil

Durante la ejecución del bloque de sentencias es posible acceder al elemento actual mediante el operador “\$”.

```
1  /*
2   *      Imprime la cadena
3   *      "Sentencia" y su índice
4   *      10 veces.
5   */
6  $(10){
7      << "Sentencia " . $;
8 }
```

Listing 10.67: Ejemplo iteración ágil con acceso

Es posible anidar dos o más sentencias de iteración ágil y acceder al índice de cada una de ellas, añadiendo una expresión que se corresponde con el nivel de anidamiento tras el operador “\$” y delimitada entre llaves.

```
1  /*
2   *      Imprime las tablas de multiplicar del
3   *      0 al 9.
4   */
5  $(10){ // Nivel 0
6      << "Tabla de multiplicar del " . ${0};
7      ${10} { // Nivel 1
8          << ${0} . " x " . ${1} . " = " . (${0} * ${1});
9      }
10 }
```

Listing 10.68: Ejemplo iteración ágil con acceso indexado

Si el operador de acceso no presenta índice este se corresponderá con el nivel actual.

10.3.4.8.8 Sentencia de salto *break*

La sentencia *break* permite terminar la ejecución de un bloque de sentencias correspondientes a una estructura iterativa y saltar a la siguiente sentencia tras el bloque.

```
1  /*
2   * Imprime el valor dado
3   * hasta que este es 0.
4  */
5  while(true){
6      >>["Dame un valor (0 para salir)"] foo;
7      if (foo === "0")
8          break;
9      << foo;
10 }
11 << "Finalizando";
```

Listing 10.69: Ejemplo sentencia *break*

Es posible salir de dos bloques iterativos que se encuentren anidados, indicando tras *break* cuantos satos se desean realizar.

```
1  /*
2   * Imprime el valor dado
3   * hasta que este es 0.
4   * Imprimiendo cuantas
5   * solicitudes se han
6   * realizado cada 10 veces
7  */
8  count = 0;
9  while (true) {
10     i = 0;
11     while(i < 10){
12         >>["Dame un valor (0 para salir)"] foo;
13         count++;
14         if (foo === "0")
15             break 2;
16         << foo;
17         i++;
18     }
19     << "Se han solicitado " . count . " valores ";
20 }
21 << "Finalizando";
```

Listing 10.70: Ejemplo sentencia *break* con índice

10.3.4.8.9 Sentencia de salto *continue*

La sentencia *continue* permite terminar la ejecución actual de un bloque de sentencias correspondientes a una estructura iterativa. De esta forma se salta a la comprobación y ejecución (si fuera el caso) de la siguiente iteración.

```
1  /*
2   * Imprime el valor dado,
3   * excepto si este es "jump",
4   * hasta que es 0.
```

```

5     */
6     while(true){
7         >>[ "Dame un valor (0 para salir, jump para saltar)"] foo;
8         if (foo === "0")
9             break;
10        if (foo === "jump")
11            continue;
12        << foo;
13    }
14    << "Finalizando";

```

Listing 10.71: Ejemplo sentencia continue

Es posible utilizar la sentencia *continue* para saltar a la siguiente iteración de una estructura iterativa concreta de varias anidadas. Para ello tras el literal se debe indicar una expresión cuyo valor sea el número de bloques anidados que se desea saltar.

```

1   /*
2      Imprime el valor dado hasta que este es 0.
3      Imprimiendo cuantas solicitudes se han realizado cada 10 veces.
4      Es posible forzar el reinicio del contador mediante el valor "restart"
5  */
6  count = 0;
7  while (true) {
8      i = 0;
9      while(i < 10){
10         >>[ "Dame un valor (0 para salir, restart para reiniciar contador)"] foo;
11         count++;
12         if (foo === "0")
13             break 2;
14         if (foo === "restart"){
15             << "Reiniciando contador ";
16             count = 0;
17             continue 2;
18         }
19         << foo;
20         i++;
21     }
22     << "Se han solicitado " << count << " valores ";
23 }
24 << "Finalizando";

```

Listing 10.72: Ejemplo sentencia continue con índice

10.3.4.8.10 Sentencia de salto *goto*

La sentencia *goto* se compone de un identificador y permite cambiar el flujo de ejecución a la sentencia etiquetada con dicho identificador.

```

1   /*
2      Se imprime la cadena
3      "Sentencia" indefinidamente
4  */
5  label: << "Sentencia";
6  goto label;

```

Listing 10.73: Ejemplo sentencia goto

En general se desaconseja el uso de etiquetas y sentencias *goto* dado que dificultan la legibilidad del código.

Lo habitual es que la sentencia *goto* se encuentre condicionada para que el salto no se ejecute siempre.

```
1  /*
2   * Se imprime la cadena
3   * "Sentencia" 10 veces
4  */
5  foo = 0;
6  label: << "Sentencia " . foo;
7  if (foo < 10){
8      foo++;
9      goto label;
10 }
```

Listing 10.74: Ejemplo sentencia goto (II)

10.3.4.8.11 Sentencia *include*

La sentencia *include* permite separar el código fuente en diferentes ficheros. Se compone de una expresión que se corresponde con el fichero a incluir. Al ser evaluada, la ejecución pasa a la primera sentencia en el fichero.

```
1  /*
2   * fichero1.omi
3   * Se imprime el contenido de este fichero
4   * y se incluye el fichero 2
5  */
6  << "Contenido fichero 1";
7  include "fichero2.omi";
```

Listing 10.75: Ejemplo sentencia include

```
1  /*
2   * fichero2.omi
3   * Se imprime el contenido de este fichero
4  */
5  << "Contenido fichero 2";
```

Listing 10.76: Ejemplo sentencia include (II)

Si la cadena facilitada como expresión no se corresponde con un fichero del sistema se producirá un error y finalizará la ejecución del script.

10.3.4.8.12 Sentencia *exit*

La sentencia *exit* permite finalizar la ejecución del script en cualquier parte del mismo.

```

1  /*
2   * Finaliza el script sin llegar
3   * a ejecuta la ultima sentencia
4   */
5 << "SENTENCIA EJECUTADA";
6 exit;
7 << "SENTENCIA NO EJECUTADA";

```

Listing 10.77: Ejemplo sentencia exit

A diferencia de otros lenguajes de programación en OMI no existe una forma de finalizar el script devolviendo un estado de error.

10.3.4.8.13 Sentencia *with*

La sentencia *with* permite establecer un objeto como contexto dentro de un bloque de sentencias. Se forma mediante una expresión correspondiente al objeto y un bloque de sentencias que será ejecutada con el objeto como contexto.

Cuando se establece un objeto como contexto todas las funciones que se llamen, y que no se encuentren definidas, serán llamadas como métodos del objeto.

```

1  class Foo {
2      ~ identity () {
3          << "Soy Foo";
4      }
5  }
6  foo = new Foo ();
7  with (foo) {
8      identity(); // Llama al método identity del objeto foo
9  }

```

Listing 10.78: Ejemplo sentencia with

10.3.4.8.14 Sentencia *try... catch...*

La sentencia *try... catch...* permite delimitar mediante un bloque de sentencias un comportamiento en el que se pueden dar excepciones. Además, mediante otro bloque de sentencias y un símbolo variable, se puede especificar cómo serán tratadas las excepciones que se den.

A diferencia de otros lenguajes de programación OMI solo permite un bloque de sentencias *catch*. No comprueba la clase de la excepción que se ha dado, esto recae en responsabilidad del programador.

```

1  class Exc {
2      var = null;
3      ~ Exc (num) {
4          this->var = num;
5      }
6      ~ printError () {
7          << "Error " . this->var;
8      }
9  }
10 try {
11     >>["Dame un valor:"] foo;
12     if ( foo === "") {

```

```

13         throw new Exc (404);
14     }
15     << foo;
16 }catch (exc) {
17     exc->printError ();
18 }
```

Listing 10.79: Ejemplo sentencia try...catch...

10.3.4.8.15 Sentencia *throw*

La sentencia *throw* permite lanzar una excepción que puede ser atrapada por una sentencia *try... catch....* Esta sentencia se construye mediante una expresión cuyo valor será asociado al símbolo variable del bloque de sentencias *catch*.

```

1  class Exc {
2     var = null;
3     ~ Exc (num) {
4         this ->var = num;
5     }
6     ~ printError () {
7         << "Error " << this ->var;
8     }
9 }
10 try {
11     >>["Dame un valor:"] foo;
12     if ( foo === "") {
13         throw new Exc (404);
14     }
15     << foo;
16 }catch (exc) {
17     exc->printError ();
18 }
```

Listing 10.80: Ejemplo sentencia *throw*

Si una excepción lanzada con *throw* no sucede dentro de una sentencia *try... catch* controlada se producirá un error y se detendrá la ejecución del script.

10.3.4.8.16 Sentencia *sleep*

La sentencia *sleep* hace que la ejecución del programa se pare un número de segundos dados.

```

1   << "Antes de parar";
2   sleep (10);
3   << "Tras 10 segundos";
```

Listing 10.81: Ejemplo sentencia *sleep*

10.3.4.8.17 Sentencia *typeof*

La sentencia *typeof* imprime en la salida estándar el tipo de dato del valor referenciado por un símbolo variable.

```

1  foo = 10;
2  typeof foo; // Imprime Arithmetic
3  foo = "STR";
4  typeof foo; // Imprime String
5  foo = {1,2};
6  typeof foo; // Imprime Array(2)

```

Listing 10.82: Ejemplo sentencia `typeof`

10.3.4.8.18 Sentencia `datInfo`

La sentencia `datInfo` permite consultar el estado interno de un dato. Esto es la posición de memoria que ocupa, el tipo y el número de referencias que tiene.

```

1  datInfo 5; // Imprime ptr(0x1d72e00), type(Arithmetic: 5), refs(0)
2  datInfo "A"; // Imprime ptr(0x1d72f70), type(String: A), refs(0)
3  foo = 10;
4  datInfo foo; // ptr(0x1d73f00), type(Arithmetic: 10), refs(1)
5  bar = foo;
6  datInfo bar; // ptr(0x1d73f00), type(Arithmetic: 10), refs(2)

```

Listing 10.83: Ejemplo sentencia `datInfo`

La sentencia `datInfo` también admite la forma `datinfo`.

10.3.4.9. Variables

En programación una variable se define como un espacio que es asociado a un nombre o identificador, el contenido alojado en dicho espacio es llamado el valor de la variable. Mediante el nombre es posible utilizar la variable en expresiones con independencia de la información exacta que esta referencia. Cuando una expresión que contiene una variable es resuelta se obtiene el valor contenido en la misma.

El valor de una variable puede cambiar durante la ejecución del programa. Para alterar el contenido de una variable lo habitual es utilizar el operador de asignación. En OMI la asignación de una variable es destructiva, es decir, la variable modifica su valor sobrescribiendo el valor anterior.

Las variables suponen un recurso esencial para los lenguajes de programación imperativos, ya que son el mecanismo básico para guardar el estado del sistema. Las variables en matemáticas forman parte de definiciones y una vez atribuido su valor este no variará, esto difiere del concepto de variable que presentan los lenguajes de programación imperativos.

OMI es un lenguaje de programación de tipado dinámico, y por tanto, el tipo de dato de una variable está determinado por el valor actual de la misma, y no se encuentra asociado a la variable. En un momento dado una variable podría contener un valor entero, y en otro una cadena de caracteres.

En OMI el tipo de dato del valor almacenado en una variable es desconocido hasta que es obtenido en la resolución de una expresión. Los operadores, funciones y demás recursos del lenguaje usan el valor para operar, con independencia de la variable que lo contuviera.

En OMI no es necesario declarar ni inicializar una variable antes de su uso. Si una variable sin valor es usada en una expresión se toma el valor nulo.

Internamente OMI representa las variables como referencias a los valores, más que como contenedores de estos. En OMI no es necesario llevar una gestión los datos que han sido creados y asignados a las variables de forma dinámica, un subsistema denominado recolector de basura será el encargado de esta tarea.

Las variables deben ser nombradas mediante un identificador válido, y no deben coincidir con ninguna palabra reservada.

Las variables en OMI pueden presentar un ámbito local y global. Una variable puede ser local al flujo principal o a una función o método. Por defecto las variables son locales. Una variable local solo puede ser accesible desde donde se utilizó inicialmente. Si en otro lugar se hace uso del mismo identificador se consideran variables distintas. Las variables globales son accesibles desde cualquier parte de la aplicación. Para usar una variable como global debe ser declarada como tal con el literal *global*.

```
1  foo = 10; // foo es una variable local al flujo principal
2  global bar; // bar es una variable global.
3  bar = 20;
4  ~ func () {
5      foo = 30; // foo es una variable local a la función func
6      bar = 40; // Se asigna a la variable global bar
7 }
```

Listing 10.84: Ejemplo variables

10.3.4.10. Referencias

Una referencia es un valor que permite acceder indirectamente a un dato almacenado en un símbolo variable. Cuando se opera sobre una referencia se hace sobre el dato almacenado en la variable.

Es posible obtener una referencia anteponiendo el símbolo “&” ante el símbolo variable.

```
1  foo = 10;
2  bar = &foo; // Se obtiene una referencia de foo
3  << foo . " - " . bar; // Imprime 10 - 10
4  bar = 20;
5  << foo . " - " . bar; // Imprime 20 - 20
```

Listing 10.85: Ejemplo referencias

En OMI es posible obtener una referencia de cualquier expresión que sea variable, y acceder su valor de forma indirecta. Así por ejemplo se puede obtener una referencia a una posición de un array, a un atributo de un objeto, etc.

```
1  foo = {{1,2}, 3};
2  bar = &foo[0][1];
3  bar = 4;
4  << foo[0][1];
```

Listing 10.86: Ejemplo referencias a elemento de array

No es posible obtener una referencia a una posición dentro de una cadena, esto es debido a que en OMI las cadenas de caracteres son un tipo de dato simple. En su lugar se copiará el valor del carácter en la posición como otra cadena, por lo que la referencia no tendrá efecto.

```

1   foo = "ABCD";
2   bar = &foo[0];
3   bar = "Z";
4   << bar; // Imprime "Z"
5   << foo; // Imprime "ABCD". La referencia no tiene efecto.

```

Listing 10.87: Ejemplo referencia a elemento constante

Sintácticamente es posible obtener una referencia de un valor constante, no obstante esto hará que se tome el valor en si, sin que tenga efecto alguno.

Las referencias pueden ser utilizadas como un dato, por lo que pueden ser pasadas como parámetro. En este caso cualquier acceso al parámetro dentro de la función se hará sobre las variables a las que referencia.

```

1   ~ inc (param) {
2     param += 1;
3   }
4
5   a = 20;
6   inc(&a);
7   << a; // Imprime 21

```

Listing 10.88: Ejemplo referencias como parámetros

Es posible hacer que algunos parámetros de una función sean pasados siempre por referencia. Para ello se antepone & al parámetro en la definición de la función. Ver paso de parámetros por referencia (Párrafo 10.3.4.11.2).

Las funciones y métodos pueden devolver referencias. De esta forma si una llamada a función es asignada se hará la operación sobre la referencia devuelta, por lo que cualquier acceso se hará sobre el símbolo variable a la que esta apunta. Si la referencia es a un símbolo variable de ámbito local, el cual es liberado al terminar la llamada, el valor será copiado.

```

1   class example {
2     private attr = null;
3
4     ~ example (attr) {
5       this->attr = attr;
6     }
7
8     ~ getRef () {
9       return &this->attr;
10    }
11
12    ~ printAttr () {
13      << this->attr;
14    }
15  }
16
17  foo = new example (20);
18  foo->printAttr(); // Imprime 20
19  << foo->attr; // Error acceso a elemento privado
20  bar = foo->getRef(); // bar es una referencia a foo->attr

```

```

21     bar = 40;
22     foo->printAttr(); // Imprime 40

```

Listing 10.89: Ejemplo referencias devueltas como valor

10.3.4.11. Funciones

Una función es un conjunto de sentencias que tienen un objetivo particular. Se corresponde con un subalgoritmo dentro del algoritmo principal.

Una función se define normalmente mediante un identificador, una lista de parámetros (delimitados por paréntesis y separados por coma) y un bloque de sentencias. La definición de una función debe empezar por el carácter “~” o la palabra reservada *function*.

```

1   ~ func1 (param1, param2) {
2     << "Soy func1 con dos parámetros:";
3     << "param1 => " . param1;
4     << "param2 => " . param2;
5   }
6
7   function func2 (param1, param2) {
8     << "Soy func2 con dos parámetros:";
9     << "param1 => " . param1;
10    << "param2 => " . param2;
11  }

```

Listing 10.90: Ejemplo funciones

Los parámetros de la función suponen un mecanismo para que esta pueda recibir datos con los que operar y realizar su tarea. Los parámetros son variables que pueden ser utilizadas en el bloque de sentencias y cuyos valores serán dado cuando la función es llamada.

Las funciones pueden ser llamadas desde cualquier parte del código, lo que supondrá la ejecución del bloque de sentencias. Para realizar una llamada a una función se utilizará el nombre de la función seguido de los valores que serán atribuidos a los parámetros, delimitados por paréntesis y separados por coma. La atribución de valores a los parámetros se hace de forma posicional.

```

1   ~ func1 (param1, param2) {
2     << "Soy func1 con dos parámetros:";
3     << "param1 => " . param1;
4     << "param2 => " . param2;
5   }
6
7   func1 ("valor1", "valor2");

```

Listing 10.91: Ejemplo llamada a función

En una llamada los parámetros son pasados, a no ser que se indique lo contrario, por valor. Esto quiere decir que los valores son copiados a los parámetros, lo que implica que cualquier modificación de estos solo tendrá efecto en el bloque que define la función.

```

1   ~ func1 (foo) {
2     foo = 20;

```

```

3     << foo; // Imprime 20
4 }
5
6 foo = 40;
7 func1 (foo);
8 << foo; // Imprime 40

```

Listing 10.92: Ejemplo llamada a función con parámetros

Una función puede devolver un valor que será atribuido a la llamada. Para ello se hace uso de la sentencia *return*. Esta sentencia, que puede formar parte del bloque de sentencias de una función, esta formada por el valor que será devuelto y atribuido a la llamada. La ejecución de la sentencia *return* implica la finalización de la función.

La llamada a un función es considerada un operador y puede formar parte de expresiones. El valor de la llamada es dado por la ejecución de la sentencia *return* en el bloque de sentencias de esta. Si la ejecución de la función termina sin una sentencia *return* se tomará el valor nulo.

```

1 ~ sum (op1, op2) {
2     return op1 + op2;
3 }
4
5 << sum(1,2) * 4; // Imprime 12

```

Listing 10.93: Ejemplo valor devuelto por función

Las llamadas a función son resueltas antes que cualquier otro operador, ya que presentan el mayor nivel de prioridad.

Si una función es llamada sin que esta se encuentre definida se producirá un error semántico. También se producirá un error si el número de valores facilitados como parámetros en la llamada no se corresponden con la definición de la función. Cabe destacar que las definiciones de funciones son ejecutadas de forma secuencial dentro del flujo del programa, por lo que realizar un llamada de un función que se encuentre definida en sentencias posteriores producirá un error como si esta no se encontrara definida, por ello es buena práctica disponer las definiciones de funciones al comienzo del programa.

Los nombres de las funciones deben seguir las reglas normales de uso de identificadores, y no corresponderse con ninguna palabra reservada.

Si dos o más definiciones de función comparten el mismo identificador la última definición interpretada será la que se corresponda con dicho identificador.

```

1 ~ func () {
2     << "FUNC 1";
3 }
4
5 ~ func () {
6     << "FUNC 2";
7 }
8
9 func(); // Imprime "FUNC 2"

```

Listing 10.94: Ejemplo redefinición de funciones

A pesar de ser considerado un lenguaje imperativo, OMI presenta algunos recursos y mecanismos propios de la programación funcional. En las subsecciones siguientes se presentan los recursos y mecanismos que pueden ser utilizados y aplicados a las funciones.

10.3.4.11.1 Parámetros con valores por defecto

En OMI es posible definir funciones cuyos parámetros presenten un valor por defecto, es decir, en caso de ser omitido en una llamada dicho parámetro tendrá el valor especificado.

En una llamada es obligatorio dar el valor de todos los parámetros excepto los que tienen definido un valor por defecto. Dado que la atribución de valores a los parámetros se hace de forma posicional, los parámetros con valores por defecto serán los últimos para que puedan omitirse en la llamada.

En la definición de una función se puede especificar que determinados parámetros tendrán valores por defecto siguiendo estos mediante el signo igual y el valor que tendrán.

```
1 ~ sum (op1, op2 = 4) {
2     return op1 + op2;
3 }
4
5 << sum(1,2) * 4; // Imprime 12
6 << sum(1) * 4; // Imprime 20
```

Listing 10.95: Ejemplo funciones con valores por defecto

10.3.4.11.2 Paso de parámetros por referencia

Por defecto los parámetros de una llamada a función son pasados por valor, lo que implica que cualquier modificación en los mismos dentro de la función no tendrá efecto fuera de esta.

Es posible definir funciones donde algunos parámetros sean pasado como referencia en las llamadas. Esto es que si en la llamada se facilita un símbolo variable como valor de dicho parámetro, cualquier modificación realizada en el mismo será aplicado a la variable fuera de la función.

En la definición de una función se puede indicar que un determinado parámetro será pasado por referencia anteponiendo el símbolo “&” a dicho parámetro.

```
1 ~ func1 (&foo) {
2     foo = 20;
3     << foo; // Imprime 20
4 }
5
6 foo = 40;
7 func1 (foo);
8 << foo; // Imprime 20
```

Listing 10.96: Ejemplo funciones con parámetros por referencia

Si un valor constante es dado como parámetro por referencias se producirá un error.

10.3.4.11.3 Definición como parte de expresiones

En OMI las funciones son consideradas tipos de datos, por lo que es posible asignarlas a variables, operar sobre ellas, etc. Sin embargo, no se puede realizar una conversión de una función a otro

tipo de dato y viceversa, por lo que la mayoría de operadores producirán un error al resolverse si operan con una función y otro tipo de dato.

Es posible utilizar la definición de una función como expresión en una operación de asignación.

```
1  foo = ~ sum (param1, param2) { return param1 + param2; };
2  << foo (4, 5); // Imprime 9
3  ((x < 10)?~(){<<"A";}:~(){<<"B"})(); // Si x < 10 imprime "A", si no imprime "B"
```

Listing 10.97: Ejemplo definición de función como parte de expresión

10.3.4.11.4 Referencias a funciones

En OMI se nombran de igual forma a las funciones, las variables y las clases. Para determinar si un identificador en una expresión hace referencia a una función u otro elemento se utiliza el contexto.

En muchos casos es necesario indicar explícitamente que el objeto que se desea utilizar es una función ya definida, y no una variable u otro elemento. Para ello se antepone al identificador de la función el par de símbolos “ $\sim \&$ ”.

```
1  ~ foo () {
2    return "FUNCIÓN";
3  }
4  foo = "VARIABLE";
5  bar = foo; // bar vale "VARIABLE";
6  bar = ~&foo; // var vale la función que devuelve "FUNCIÓN"
```

Listing 10.98: Ejemplo acceso a función

Por regla general todos los operadores toman, a no ser que se indique lo contrario, los identificadores como variables. Las excepciones a esta regla son los siguientes operadores:

- Llamada a función
- Aplicación parcial
- *new*.

10.3.4.11.5 Funciones anónimas

OMI permite definir funciones sin identificador que las nombre, lo que se denomina funciones anónimas. Normalmente las funciones anónimas son utilizadas como dato en otras expresiones. Para definir una función anónima solo se ha de omitir el identificador.

```
1  bar = foo = ~() { << "FUNCIÓN ANÓNIMA"; };
2  foo(); // IMPRIME "FUNCIÓN ANÓNIMA";
3  bar(); // IMPRIME "FUNCIÓN ANÓNIMA";
```

Listing 10.99: Ejemplo funciones anónimas

10.3.4.11.6 Expresiones parametrizadas

Una expresión parametrizada es similar a una función. Permite definir una expresión a partir de unos parámetros.

La expresión parametrizada puede ser utilizada como dato para llevar a cabo operaciones sobre ella. Así es posible asignarla, llamarla con unos valores dado, ser devuelta como valores de función, etc.

Para definir una expresión parametrizada se utiliza el símbolo “~” seguido de una lista de parámetros separados por coma, el símbolo “:” y la expresión a parametrizar, la cual normalmente hará uso de los parámetros. Es posible obtener el valor para unos determinados parámetros utilizando el operador de llamada a función.

```
1 sum = ~ x, y : x + y;
2 << sum (4,5);
```

Listing 10.100: Ejemplo expresiones parametrizadas

10.3.4.11.7 Funciones de orden superior

Las funciones de orden superior son funciones que cumplen una de las siguientes premisas:

- Reciben como parámetros una o más funciones.
- Devuelven como valor una función.

En OMI las funciones son tipos de datos, por lo que es posible crear funciones de orden superior de forma natural a la sintaxis del lenguaje.

```
1 ~ call (f) { // Función de orden superior que recibe una función como parámetro
2   f();
3 }
4
5 ~ printA () { << "A" }
6 ~ printB () { << "B" }
7
8 call (~&printA); // Imprime A
9 call (~&printB); // Imprime B
```

Listing 10.101: Ejemplo función de orden superior que recibe una función

```
1 ~ getOp (code) { // Función de orden superior que devuelve una función
2   switch (code) {
3     case 0: return ~(a,b) { return a + b }; break;
4     case 1: return ~(a,b) { return a - b }; break;
5     default: return ~(a,b) { << "WRONG CODE"; return null; }
6   }
7 }
8
9 func = getOp (0);
10 << func (4, 5); // Imprime 9
11
12 func = getOp (1);
13 << func (4, 5); // Imprime -1
14
15 func = getOp ("*");
```

```
| 16    << func (4, 5); // Imprime "WRONG CODE"
```

Listing 10.102: Ejemplo función de orden superior que devuelve una función

10.3.4.11.8 Clausura de funciones

Si una función es definida en el bloque de sentencias de otra, cuando la función interna sea llamada, en su ejecución podrá acceder a las variables definidas en el bloque en el momento en el que se definió.

La clausura permite que funciones que sean definidas en un entorno puedan acceder a las variables de dicho entorno.

```
1   ~ addX (x) {
2       return ~(num) {
3           return num + x; // x pertenece al entorno de addX
4       }
5   }
6
7   add1 = addX(1);
8   add2 = addX(2);
9
10  << add1 (4); // Imprime 5
11  << add2 (4); // Imprime 6
```

Listing 10.103: Ejemplo clausura de funciones

10.3.4.11.9 Función de contexto

Por regla general la función de contexto se refiere a la función en ejecución. En OMI es posible acceder a la función de contexto mediante el operador “~>”, esto permite definir funciones recursivas de forma simple.

```
1   ~ factorial (num) {
2       if (num == 2) return 2;
3       return num * ~>(num-1);
4   }
```

Listing 10.104: Ejemplo función de contexto

En otros casos la función de contexto puede mantener otras referencias diferentes a la función en ejecución.

10.3.4.11.10 Decoradores

Un decorador es una función que:

- Toma una función como parámetro.
- Devuelve otra función que utiliza la función recibida como parámetro en su bloque de sentencias.

Es posible crear un decorador mediante funciones gracias al principio de clausura.

```
1 ~ generateFileHTML (func) { //Decorador mediante funciones
2     return ~(title, params){
3         << "<html>";
4         << "<head>";
5         << "<title>" << title << "</title>";
6         << "</head>";
7         << "<body>";
8         func(params);
9         << "</body>";
10        << "</html>";
11    }
12 }
13
14 search = ~(params) { << "Buscando: " << params; };
15 info = ~(params) { << "Información sobre: " << params; };
16
17 generateSearchHTML = generateFileHTML (search);
18 generateInfoHTML = generateFileHTML (info);
19
20 generateSearchHTML("Búsqueda", "concepto");
21 generateInfoHTML("Información", "producto");
```

Listing 10.105: Ejemplo implementación de decorador

En OMI existe una construcción propia del lenguaje denominada decorador. Esta simplifica el proceso anterior. Un decorador se define de forma parecida a una función: presenta un identificador que lo nombra, una lista de parámetros y un bloque de sentencias. La lista de parámetros se corresponde con los que recibirá la función devuelta por el decorador, la cual se definirá a partir del bloque de sentencias. En el bloque de sentencias se podrá hacer referencia a la función a decorar mediante el operador de función de contexto “~>”. Para definir un decorador se utiliza el literal “~~”.

```
1 ~~ generateFileHTML (title, params) { // Decorador mediante construcciones del
2     lenguaje
3     << "<html>";
4     << "<head>";
5     << "<title>" << title << "</title>";
6     << "</head>";
7     << "<body>";
8     ~>(params);
9     << "</body>";
10    << "</html>";
11
12 search = ~(params) { << "Buscando: " << params; };
13 info = ~(params) { << "Información sobre: " << params; };
14
15 generateSearchHTML = generateFileHTML (search);
16 generateInfoHTML = generateFileHTML (info);
17
18 generateSearchHTML("Búsqueda", "concepto");
19 generateInfoHTML("Información", "producto");
```

Listing 10.106: Ejemplo estructura decorador

10.3.4.11.11 Aplicación parcial

La aplicación parcial toma una función y devuelve otra basada en esta, donde algunos de sus parámetros presentan ya un valor.

Para la aplicación parcial se utiliza el operador con el mismo nombre. Este consiste en el literal “*P*” seguidos de una lista con los valores que se aplicarán y la función sobre la que se realizará la operación. La lista de parámetros consiste en pares parámetro/valor separados por el signo “=”, y entre si por coma. Los parámetros se han de corresponder con parámetros de la función sobre la que se realizará la operación. La función resultante será una función que recibe tantos parámetros como la función original exceptuando aquellos que tienen aplicados un valor.

```
1  ~ addX (num, x) {
2      return num + x;
3  }
4
5  add4 = P[X=4]addX;
6  add8 = P[X=8]addX;
7
8  << add4(5); // Imprime 9
9  << add8(5); // Imprime 13
```

Listing 10.107: Ejemplo aplicación parcial

10.3.4.12. Listas por comprensión

En OMI es posible definir una lista o un diccionario por compresión. Para ello se utiliza una construcción del lenguaje similar a la sentencia iterativa *for*, salvo que en cada iteración genera un elemento del array asociado a la lista o diccionario.

La definición de una lista por compresión debe ir entre paréntesis.

La forma más simple de crear una lista por compresión es mediante una expresión de generación, el literal *for* y una variable que iterará sobre una expresión conjunto. En cada iteración la variable iteradora tomará un valor del conjunto. Para cada elemento se calculará el valor de la expresión generadora, la cual normalmente contendrá la variable iteradora, y será introducido en el array resultado. El conjunto sobre el que se itera dependerá del tipo de dato obtenido al evaluar la expresión conjunto:

Booleano: Se itera hasta que se evalúa como falso. En cada iteración se asigna el entero correspondiente al número de la misma.

Numérico: Si es positivo se itera desde cero hasta el valor sin incluirlo. Si es negativo no se itera.

Cadena de caracteres: Se itera por cada carácter en la cadena.

Array: Se itera por cada elemento en el array.

```
1  /*
2   *      Array de 10 elementos: desde el 0 al 9
3   */
4  foo = ( y++ for x in y < 10 )
5  /*
6   *      Array de 10 elementos donde las
7   *      posiciones pares valen 0 y las
8   *      impares 1
9  */
10 foo = ( x % 2 for x in 10 );
```

```

11  /*
12   *     Array que contiene "A", "B", "C" y "D"
13  */
14  foo = ( x for x in "ABCD" );
15  /*
16   *     Array que contiene 1, 2 y 4
17  */
18  foo = ( x/2 for x in {2,4,8} );

```

Listing 10.108: Ejemplo listas por comprensión

En cada iteración la expresión de conjunto es evaluada antes de obtener el elemento correspondiente.

Es posible filtrar los datos que serán incluidos en el array, añadiendo el literal *if* y la condición que se debe cumplir.

```

1  /*
2   *     Array de los pares
3   *     del 0 al 9
4  */
5  foo = ( x for x in 10 if x % 2 == 0 );

```

Listing 10.109: Ejemplo listas por compresión con filtro

Dado un array asociativo como conjunto es posible iterar sobre los pares clave/valor que lo conforman , para ello se define el par de variables que se van a utilizar separadas por “:”.

```

1  /*
2   *     Array que contiene "key0 => val0"
3   *     y "key1 => val1"
4  */
5  foo = ( k. " => ".v for k:v in {"key0": "val0", "key1": "val1"} );

```

Listing 10.110: Ejemplo listas por compresión con clave

El array generado puede ser asociativo utilizando dos expresiones de generación separadas por “:” que se utilizarán para construir las claves y el valores.

```

1  /*
2   *     Array asociativo con las claves "key0", "key1" y "key2",
3   *     cuyos valores son 2, 4 y 6 respectivamente .
4  */
5  foo = ( "key".k : x for k:x in {2,4,6} );

```

Listing 10.111: Ejemplo listas por compresión generando array asociativo

Es posible indicar un bloque de sentencias que será ejecutado con cada iteración. Este bloque irá después del conjunto a iterar.

```

1  /*
2   *     Array con los valores dados .

```

```

3     Se solicita valores y se van introduciendo
4     en el array hasta que no se da valor
5     */
6     foo = ( x for i in x != "" {>>["Posición ".i.": "] x} if x != "" );

```

Listing 10.112: Ejemplo listas por compresión con bloque de sentencias

10.3.4.13. Clases de objeto

OMI es un lenguaje orientado a objetos, por tanto permite hacer uso de los conceptos y recursos enmarcados dentro de este paradigma de programación.

Una clase es una construcción del lenguaje que modela un concepto del dominio del problema. Se trata de la definición de un conjunto de entidades u objetos con características y propiedades comunes. Describe los datos necesarios para representarlos y el comportamiento que tienen asociado.

Los objetos son casos concretos de los conceptos modelados mediante las clases. Guardan un estado dando valores a los datos descritos. Además presentan el comportamiento ligado al concepto.

Los datos que representan el estado de un objeto son modelados mediante atributos, mientras que el comportamiento es definido mediante métodos.

En OMI una clase se construye mediante la palabra reservada “*class*”, un identificador que la nombra y un bloque de sentencias en la que se definirán los atributos y métodos. Un atributo se define de forma similar a una variable, y un método a una función.

```

1  /*
2   Se define una clase gato. Un gato
3   se representa mediante un nombre y tiene
4   asociado el comportamiento de aullar.
5   */
6   class gato {
7     nombre;
8     ~ aullar () {
9       << "Miauuu";
10    }
11 }

```

Listing 10.113: Ejemplo clases de objeto

Los objetos pueden ser creados mediante la instanciación de una clase. El objeto dará valores a los atributos y se le podrá aplicar los métodos definidos. Al crear una instancia se obtendrá un objeto llamando al método constructor, que es aquel que tenga el mismo nombre que la clase. Para crear una instancia se utiliza el operador “*new*” seguido del nombre de la clase y los parámetros del método constructor.

```

1  /*
2   Se define una clase gato. Un gato
3   se representa mediante un nombre y tiene
4   asociado el comportamiento de aullar.
5   */
6   class gato {
7     nombre;
8     ~ gato (nombre) {
9       << "Construyendo el gato ".nombre;

```

```

10      }
11      ~ aullar () {
12          << "Mi auuu";
13      }
14  }
15 /*
16     Se instancia un gato
17     llamado "MacAlistair"
18 */
19 mac = new gato ("MacAlistair"); // Imprime "Construyendo el gato MacAlistair"

```

Listing 10.114: Ejemplo instantiación de objeto

Si se instancia una clase que no se encuentra definida se produce un error semántico y se continua con la ejecución del programa.

Es posible acceder a los atributos y métodos de un objeto mediante el operador “`->`”. También se puede hacer referencia al objeto en ejecución dentro de los métodos, para ello se utiliza el operador “`this`”.

```

1  /*
2     Se define una clase gato. Un gato
3     se representa mediante un nombre y tiene
4     asociado el comportamiento de aullar.
5 */
6 class gato {
7     nombre;
8     ~ gato (nombre) {
9         this->nombre = nombre;
10    }
11    ~ aullar () {
12        << "Mi auuu";
13    }
14 }
15 /*
16     Se instancia un gato
17     llamado "MacAlistair"
18 */
19 mac = new gato ("MacAlistair");
20 mac->aullar(); // Imprime "Mi auuu"
21 << mac->nombre; // Imprime "MacAlistair"

```

Listing 10.115: Ejemplo acceso a atributos y métodos

Si se accede a un atributo o método no existente se llaman al método “`_get`” o “`_call`” respectivamente (ver Métodos mágicos Párrafo 10.3.4.13.7). Si estos no existen se produce un error semántico y se continua con la ejecución.

Si no existe ningún método llamado igual que la clase, se considera que esta tiene un constructor vacío. Este es un método con el bloque de sentencias vacío y sin parámetros.

Una clase solo puede disponer de un constructor, si se dan varios métodos llamados igual que la clase solo tendrá efecto el último definido.

10.3.4.13.1 Accesibilidad de métodos y atributos

Los métodos y atributos definidos en un clase pueden tener restricciones a la accesibilidad de los mismos. Estos pueden ser públicos o privados.

Los atributos y métodos públicos pueden ser accedidos desde un ámbito externo a la clase. A no

ser que se especifique lo contrario todos los métodos y atributos de una clase son públicos.

```
1  /*
2   * Se define una clase gato. Un gato
3   * se representa mediante un nombre y tiene
4   * asociado el comportamiento de aullar.
5   */
6  class gato {
7      nombre; // Atributo público
8      ~ gato (nombre) { // Método público
9          this->nombre = nombre; // Acceso a atributo público dentro de la clase
10     }
11     ~ aullar () { // Método público
12         << "Miawuu";
13     }
14 }
15 /*
16 * Se instancia un gato
17 llamado "MacAlistair"
18 */
19 mac = new gato ("MacAlistair");
20 mac->aullar(); // Llamada a método público fuera de la clase
21 << mac->nombre; // Acceso a atributo público fuera de la clase
```

Listing 10.116: Ejemplo accesibilidad métodos y atributos públicos

Los métodos o atributos privados no son accesibles desde fuera de la propia clase, solo desde los métodos definido dentro de esta. Para declarar un atributo o método como privado se antepone la palabra clave *private* a la definición.

```
1  /*
2   * Se define una clase gato. Un gato
3   * se representa mediante un nombre y tiene
4   * asociado el comportamiento de aullar.
5   */
6  class gato {
7      private nombre; // Atributo privado
8      ~ gato (nombre) { // Método público
9          this->nombre = nombre; // Acceso a atributo privado dentro de la clase
10     }
11     ~ aullar () { // Método público
12         << "Miawuu";
13     }
14 }
15 /*
16 * Se instancia un gato
17 llamado "MacAlistair"
18 */
19 mac = new gato ("MacAlistair");
20 mac->aullar(); // Llamada a método público fuera de la clase
21 << mac->nombre; // Acceso a atributo privado fuera de la clase (Error)
```

Listing 10.117: Ejemplo accesibilidad métodos y atributos privados

Si el constructor de una clase es privado solo se podrán crear instancias desde los métodos de esta, los cuales normalmente serán métodos estáticos.

10.3.4.13.2 Atributos y métodos estáticos

En OMI es posible declarar un método o atributo como estático, esto hará que pertenezca a la clase y no a los objetos instanciados. Los atributos y métodos estáticos son accesibles utilizando la clase y no utilizando un objeto. Generalmente guardan datos o definen un comportamiento aplicables a un conjunto de objetos definido por la clase.

Para declarar un atributo o método como estático se utiliza la palabra reservada “*static*”. Para acceder a un método o atributo se utiliza el nombre de la clase, el operador de resolución de nombre de dominio “`::`” y el nombre del atributo o método. Es posible especificar que un atributo o método será a la vez estático y privado.

```
1  class gato {
2      private nombre;
3      private raza;
4      private energia = 100;
5      private hambre = 100;
6
7      static private energia_min = -10;
8      static private hambre_min = -10;
9
10     static ~ pasar_tiempo (gatos, h = 1) {
11         if (h > 0) {
12             $(gatos) {
13                 $->energia -= 10;
14                 $->hambre -= (30 * h);
15                 if ($->energia < gato::energia_min || $->hambre < gato::hambre_min)
16                     << $->nombre. " a pasado a un lugar mejor";
17             }
18         } else{
19             << "Imposible";
20         }
21     }
22
23     ~ gato (nombre, raza) {
24         this->nombre = nombre;
25         this->raza = raza;
26     }
27
28     ~ getNombre () {
29         return this->nombre;
30     }
31
32     ~ getRaza () {
33         return this->raza;
34     }
35
36     ~ aullar () {
37         << "Miauuu";
38     }
39
40     ~ comer () {
41         this->hambre += 10;
42         << "El animal come";
43     }
44
45     ~ dormir () {
46         this->energia += 10;
47         << "El animal duerme";
48     }
49
50     ~ jugar () {
51         if (this->energia <= 0 or this->hambre <= 0)
52             << "El gato ".this->nombre. " no quiere jugar.";
53         else{
54             << "A ".this->nombre. " le encanta jugar";
55             this->energia -= 10;
```

```

56         this->hambre -= 10;
57     }
58 }
59 }
60 }
61 }
62 mac = new gato ("MacAlstair", "Siamés");
63 ada = new gato ("Ada", "Persa");
64 bab = new gato ("Babbage", "Persa");
65
66 $(15) {
67     << "Gatos disponibles";
68     << "[0]=> ".mac->getNombre(). " [".mac->getRaza(). "]";
69     << "[1]=> ".ada->getNombre(). " [".ada->getRaza(). "]";
70     << "[2]=> ".bab->getNombre(). " [".bab->getRaza(). "]";
71     << "[otro]=> Salir ";
72     >>["Seleccione un gato para interactuar:"] s;
73     switch (s) {
74         case 0:
75             g = mac;
76             break;
77         case 1:
78             g = ada;
79             break;
80         case 2:
81             g = bab;
82             break;
83         default:
84             exit;
85     }
86     << "-----";
87     << "Acciones disponibles";
88     << "[0]=> Dormir ";
89     << "[1]=> Comer ";
90     << "[2]=> Jugar ";
91     << "[otro]=> Aullar ";
92     >>["Seleccione una acción:"] s;
93     switch (s) {
94         case 0:
95             g->dormir();
96             break;
97         case 1:
98             g->comer();
99             break;
100        case 2:
101            g->jugar();
102            break;
103        default:
104            g->aullar();
105    }
106    gato::pasar_tiempo ({mac, ada, bab});
107 }

```

Listing 10.118: Ejemplo atributos y métodos estáticos

10.3.4.13.3 Herencia de clases

En OMI es posible definir una relación de herencia entre dos clases. Que una clase herede de otra significa que es una especialización de esta.

La herencia permite establecer una jerarquía entre clases, tal que la clase que especializa es denominada hija de la clase que más genérica, denominada padre. La clase hija deriva de la madre y extiende su funcionalidad y definición. Una clase que extiende a otra toma todos sus métodos y atributos, pudiendo añadir nuevos o redefinir los existentes.

La herencia de clases permite que se dé la propiedad de polimorfismo, de forma que es posible mandar mensajes sintácticamente iguales a objetos de tipos distintos.

Para definir una relación de herencia entre dos clases se sigue el nombre de la clase hija de la palabra reservada “*extends*” y del nombre de clase a la que extiende.

```
1  class mascota {
2      private nombre;
3      private raza;
4
5      ~ mascota (nombre, raza) {
6          this->nombre = nombre;
7          this->raza = raza;
8      }
9
10     ~ getNombre () {
11         return this->nombre;
12     }
13
14     ~ getRaza () {
15         return this->raza;
16     }
17
18 }
19
20 class gato extends mascota {
21     ~ aullar () {
22         << "Miauuu";
23     }
24 }
25
26 class perro extends mascota {
27     ~ aullar () {
28         << "Guauuu";
29     }
30 }
31
32 mac = new gato ("MacAlistair", "Siamés");
33 ada = new perro ("Ada", "Bulldog");
34 mac->aullar(); // Imprime "Miauuu"
35 ada->aullar(); // Imprime "Guauuu"
```

Listing 10.119: Ejemplo herencia de clases

Si alguna clase hija no define constructor se toma el constructor de la clase padre si esta tuviera. Es posible definir una serie de clases de forma que presenten varios niveles de jerarquía.

Desde un método definido en la clase hija es posible hacer referencia a los métodos definidos en la clase padre mediante la palabra reservada “*parent*”.

```
1  class mascota {
2      private nombre;
3      private raza;
4
5      ~ mascota (nombre, raza) {
6          this->nombre = nombre;
7          this->raza = raza;
8      }
9
10     ~ getNombre () {
11         return this->nombre;
12     }
13
14     ~ getRaza () {
15         return this->raza;
```

```

16     }
17
18 }
19
20 class gato extends mascota {
21     ~ aullar () {
22         << "Miawuu";
23     }
24 }
25
26 class gato_persa extends gato {
27     ~ gato_persa (nombre) {
28         parent->gato (nombre, "Persa"); // Llama al método constructor de la clase padre
29     }
30 }
31
32 mac = new gato_persa ("MacAlistar");
33 mac->aullar(); // Imprime "Miawuu"
34 << mac->getRaza(); // Imprime "Persa"

```

Listing 10.120: Ejemplo herencia uso de parent

Cuando un método de una clase padre utiliza el operador “*this*” se accede al objeto en si, de la misma forma que se haría desde un método de la clase hija.

```

1  class mascota {
2      private nombre;
3      private raza;
4
5      ~ mascota (nombre, raza) {
6          this->nombre = nombre;
7          this->raza = raza;
8      }
9
10     ~ getNombre () {
11         return this->nombre;
12     }
13
14     ~ getRaza () {
15         return this->raza;
16     }
17
18     ~ saludar () {
19         // Accede al método getType definido en la clase hija
20         << "Hola soy ".this->nombre.", un ".this->getType()." ".this->raza;
21     }
22 }
23
24
25 class gato extends mascota {
26     ~ getType () {
27         return "gato";
28     }
29
30     ~ aullar () {
31         << "Miawuu";
32     }
33 }
34
35 class perro extends mascota {
36     ~ getType () {
37         return "perro";
38     }
39
40     ~ aullar () {

```

```

41         << "Guauuu";
42     }
43 }
44
45 mac = new gato ("MacAlistair", "Persa");
46 mac->aullar(); // Imprime "Miauuu"
47 mac->saludar(); // Imprime "Hola soy MacAlistair, un gato Persa"
48
49 ada = new perro ("Ada", "Bulldog");
50 ada->aullar(); // Imprime "Guauuu"
51 ada->saludar(); // Imprime "Hola soy Ada, un perro Bulldog"

```

Listing 10.121: Ejemplo herencia uso de this

En OMI es posible acceder a un atributo o método estático que es definido en la clase hija, desde un método de la clase padre. Para ello se utiliza la palabra reservada “*static*” seguido del operador “`::`” y el nombre del atributo o método al que se accederá.

```

1  class mascota {
2     private nombre;
3     private raza;
4     ~ mascota (nombre, raza) {
5         this->nombre = nombre;
6         this->raza = raza;
7     }
8     ~ getNombre () {
9         return this->nombre;
10    }
11    ~ getRaza () {
12        return this->raza;
13    }
14    ~ saludar () {
15        // Accede al atributo estático type definido en la clase hija
16        << "Hola soy ".this->nombre.", un ".static::type.".this->raza;
17    }
18 }
19 class gato extends mascota {
20     static type = "gato";
21     ~ aullar () {
22         << "Miauuu";
23     }
24 }
25 class perro extends mascota {
26     static type = "perro";
27     ~ aullar () {
28         << "Guauuu";
29     }
30 }
31
32 mac = new gato ("MacAlistair", "Persa");
33 mac->aullar(); // Imprime "Miauuu"
34 mac->saludar(); // Imprime "Hola soy MacAlistair, un gato Persa"
35 ada = new perro ("Ada", "Bulldog");
36 ada->aullar(); // Imprime "Guauuu"
37 ada->saludar(); // Imprime "Hola soy Ada, un perro Bulldog"

```

Listing 10.122: Ejemplo herencia uso de static

En OMI no es posible llevar a cabo herencia múltiple. Tampoco dispone de ningún mecanismo con el que se pueda simular tales como traits.

10.3.4.13.4 Redefinición de clases

En OMI es posible redefinir una clase, de forma que se pueden añadir métodos o sobrescribir los existentes.

Para redefinir una clase basta con definirla nuevamente con el mismo identificador. La nueva clase tomará todos los atributos y métodos anteriores.

```
1  class miClase {
2      ~ meth1 () {
3          << "Método 1";
4      }
5  }
6
7  class miClase {
8      ~ meth2 () {
9          << "Método 2";
10     }
11 }
12 foo = new miClase ();
13 foo->meth1(); // Imprime "método 1"
14 foo->meth2(); // Imprime "método 2"
15 class miClase {
16     ~ meth1 () {
17         << "Redefinición de método 1";
18     }
19 }
20 foo = new miClase ();
21 foo->meth1(); // Imprime "Redefinición de método 1"
22 foo->meth2(); // Imprime "Método 2"
```

Listing 10.123: Ejemplo redefinición de clases

La redefinición de clases es un mecanismo que ofrece flexibilidad a la hora de construir clases de objetos, no obstante debe ser utilizado con cuidado ya que hace que la definición de una clase sea dinámica y pueda cambiar en la ejecución pudiendo ocasionar confusión en el código e inestabilidad del programa.

10.3.4.13.5 Clases bases

En OMI existen una serie de clases denominadas clases bases, a partir de las cuales se definen los tipos de datos básicos. Estas son:

- logicClass
- arithClass
- stringClass
- arrayClass

Es posible redefinir una clase base y modificar su comportamiento, añadiendo o sustituyendo métodos.

```
1  class stringClass {
2      ~ OReq (elems) {
3          $(elems)
4              if (this == $)
5                  return true;
```

```

6     return false;
7 }
8 ~ concatX () {
9     return this .= "X";
10}
11}
12
13 foo = "ABC" -> ØReq({ "A", "AB", "ABC" }); // foo vale true
14 foo = "ABCD" -> ØReq({ "A", "AB", "ABC" }); // foo vale false
15 foo = "ABCD" -> concatX(); // foo vale "ABCDX"
16
17 str = "ABCD";
18 << str->concatX(); // Imprime "ABCDX"
19 << str; // Imprime "ABCDX"

```

Listing 10.124: Ejemplo clase base

En una clase base el operador “*this*” obtiene el dato en si, por lo que este será tratado como una variable de este tipo.

Es posible extender una clase base para especializarla mediante otras subclases.

```

1 class stringA extends stringClass {
2     ~ stringA () {
3         this = "A";
4     }
5 }
6 class stringB extends stringClass {
7     ~ stringB () {
8         this = "B";
9     }
10}
11
12 strA = new stringA ();
13 strB = new stringB ();
14 << strA; // Imprime "A"
15 << strB; // Imprime "B"
16 << strA . strB; // Imprime "AB"

```

Listing 10.125: Ejemplo extensión de clase base

Las clases bases logicClass y arithClass no disponen inicialmente de métodos. Las clases stringClass y arrayClass tienen como métodos los equivalentes a las funciones aplicables a estos tipos de datos.

10.3.4.13.6 Duck typing

OMI es un lenguaje de tipado dinámico que presenta un estilo duck typing. La validez semántica en el uso de un objeto viene determinada por el conjunto de atributos y métodos de este.

Si en una expresión se accede a un método o atributo de un objeto únicamente se comprobará que el elemento exista, indistintamente de la clase a la que pertenezca el objeto. De esta forma es posible utilizar un objeto independientemente de la clase a la que pertenece, sin que sea necesario un chequeo de tipos.

```

1 class pato {
2     ~ sonar () {
3         << "Cuack";
4     }

```

```

5     }
6
7     class persona {
8         ~ sonar () {
9             << "La persona imita el sonido de un pato";
10        }
11    }
12
13
14    ~ sonar (obj) {
15        /*
16         * Sea cual sea la clase del objeto,
17         * si tiene el método sonar
18         * será llamado
19         */
20        obj->sonar();
21    }
22
23    obj1 = new pato ();
24    obj2 = new persona ();
25    sonar (obj1); // Imprime "Cuack"
26    sonar (obj2); // Imprime "La persona imita el sonido de un pato"

```

Listing 10.126: Ejemplo duck typing

El estilo de tipado duck typing permite que se dé polimorfismo sin herencia de clases, de forma que a objetos de distinto tipos se le puede enviar mensajes sintácticamente iguales.

10.3.4.13.7 Métodos mágicos

En OMI se definen una serie de métodos mágicos. Estos son métodos que serán llamados en distintas circunstancias, como cuando se utilice al objeto como una cadena o se acceda a un atributo no existente.

Los métodos mágicos comienzan por el signo “_” y son los siguientes:

- _str:** Se llama cuando se accede a un objeto como una cadena de caracteres.
- _get:** Se llama cuando se accede a un atributo que no existe.
- _call:** Se llama cuando se accede a un método que no existe.

El método “`_str`” no tendrá parámetros y devolverá la cadena de caracteres que represente el objeto.

```

1   class persona {
2       private nombre;
3       private apellidos;
4       ~ persona (nombre, apellidos) {
5           this->nombre = nombre;
6           this->apellidos = apellidos;
7       }
8       ~ _str () {
9           return this->apellidos . ' ' . this->nombre;
10      }
11  }
12
13  p = new persona ('Fco. Javier', 'Bohórquez Ogalla');
14  << p; // Imprime "Bohórquez Ogalla, Fco. Javier"

```

Listing 10.127: Ejemplo método mágico `_str`

El método “_get” tendrá un único parámetro que se corresponderá con el nombre del atributo no existente al que se ha accedido. Y devolverá el valor para dicho atributo.

```

1  class persona {
2      private nombre;
3      private apellidos;
4
5      ~ persona (nombre, apellidos) {
6          this->nombre = nombre;
7          this->apellidos = apellidos;
8      }
9
10     ~ _get (attr) {
11         << "El atributo ".attr. " no existe";
12         return "No existe";
13     }
14
15 }
16
17 p = new persona ('Fco. Javier', 'Bohórquez Ogalla');
18 foo = p->test; // Imprime "El atributo test no existe"
19 << foo; // Imprime "No existe"
```

Listing 10.128: Ejemplo método mágico `_get'`

El método “_call” tendrá dos parámetros que se corresponderán con el nombre del método no existente al que se ha accedido y con un array que contendrá los parámetros pasados en la llama. El valor devuelto será el valor de la llamada.

```

1  class persona {
2      private nombre;
3      private apellidos;
4
5      ~ persona (nombre, apellidos) {
6          this->nombre = nombre;
7          this->apellidos = apellidos;
8      }
9
10     ~ _call (meth, params) {
11         <"El método ".meth. " no existe [ ";
12         $(params) <.$. " ";
13         << "]";
14         return "No existe";
15     }
16
17 }
18
19 p = new persona ('Fco. Javier', 'Bohórquez Ogalla');
20 foo = p->test("1", 2, "4"); // Imprime "El método test no existe [ 1 2 4 ]"
21 << foo; // Imprime "No existe"
```

Listing 10.129: Ejemplo método mágico `_call`

10.3.4.14. Entorno del programa

En OMI es posible acceder al entorno en el que se ejecuta el programa, definido por el sistema operativo y la entrada del programa.

La mayoría de sistemas operativos permiten definir una serie de variables de entorno que pueden ser usadas por los procesos y por el propio sistema operativo. En OMI es posible acceder a estas variables mediante la función *getenv* y una cadena que representa el nombre de la variable a la que se desea acceder.

```
1 << getenv ("USER"); // Imprime el nombre de usuario
```

Listing 10.130: Ejemplo variables de entorno

Un programa escrito en OMI puede recibir una serie de parámetros cuando es ejecutado. Los argumentos serán almacenados en una variable denominada *args*. El valor de esta será un array donde el primer elemento es el nombre del script y los siguientes los argumentos dados.

```
1 // File: args.omi
2 $args << $; // Imprime el nombre del script y los argumentos.
```

Listing 10.131: Ejemplo acceso a argumentos

```
1 omi args.omi "ARG1" "ARG2" # Imprime args.omi ARG1 y ARG2
```

Listing 10.132: Ejemplo paso de argumentos

10.3.4.15. Excepciones

Las excepciones son un mecanismo de programación que permite tratar casos no habituales, normalmente de error, durante la ejecución de un programa. Las excepciones permiten separar el código correspondiente al caso de normal o de éxito, del código extraordinario o de error.

El uso de excepciones permite realizar programas robustos y tolerantes a errores, a la vez que se gana claridad en el código.

Para hacer uso del manejo de excepciones se utiliza la sentencia *try...catch...*, esta consiste en la palabra reservada *try* seguida del bloque de sentencias en la que puede darse la excepción. Tras lo cual se utiliza la palabra reservada *catch*, un símbolo variable al que se le va a asignar el valor de la excepción y un bloque de sentencias en la que se va a tratar. Para lanzar la excepción se utilizar la palabra reservada *throw* seguida de una expresión que le da valor.

```
1 class Exc {
2     var = null;
3     ~ Exc (num) {
4         this->var = num;
5     }
6     ~ printError () {
7         << "Error" << this->var;
8     }
9 }
10 try {
```

```

11     >>["Dame un valor:"] foo;
12     if ( foo === "" ){
13         throw new Exc (404);
14     }
15     << foo;
16 }catch (exc) {
17     exc->printError ();
18 }
```

Listing 10.133: Ejemplo excepciones

A diferencias de otros lenguajes de programación OMI solo permite un bloque catch que será ejecutado sea cual sea la excepción que se produzca. No se comprobará el tipo de dato del valor asociado a la excepción.

10.3.4.16. Reflexión

OMI es un lenguaje que presenta características reflexivas. Un programa escrito en OMI puede hacer uso del entorno construido durante su ejecución para cambiar su propia estructura y comportamiento.

En OMI es posible utilizar expresiones cuyo valor sea una cadena de caracteres como un identificador para una variable, una función, una clase, un método, etc. Para ello se utiliza el símbolo "@" seguido de la expresión entre llaves.

```

1  class Foo {
2      ~ hello () {
3          << "Hola mundo";
4      }
5  }
6
7  class_pre = "Fo";
8  class_post = "o";
9  var = "object";
10 method = "hello";
11 @{var} = new @{class_pre.class_post}();
12 object->@{method}(); // Imprime "Hola mundo"
```

Listing 10.134: Ejemplo reflexión

En OMI también es posible evaluar una cadena de caracteres como si de una sentencia de código fuente se tratase. Para ello se utiliza la sentencia "eval".

```

1  foo = " x = 4 + 3 ";
2  eval (foo);
3  << x; // Imprime 7
```

Listing 10.135: Ejemplo eval

10.3.4.17. Introspección de tipos

OMI es un lenguaje en el que es posible llevar a cabo introspección de tipos. Permite examinar, en tiempo de ejecución, el tipo de los objetos creados.

Para obtener la clase de un objeto dado se utiliza la palabra clave “*getclass*” seguida de la expresión, cuyo valor representará el objeto, entre paréntesis. Se devolverá la cadena de caracteres correspondiente al nombre de la clase, o la cadena vacía si la expresión no es un objeto.

```
1  class Foo {
2      ~ hello () {
3          << "Hola mundo";
4      }
5  }
6  foo = new Foo ();
7  if (getclass(foo) == 'Foo')
8      foo->hello(); // Imprime "Hola mundo"
```

Listing 10.136: Ejemplo introspección de tipos

10.3.4.18. Errores

Cuando un programa escrito en OMI es interpretado se pueden dar diferentes tipos de errores. Un error de interpretación puede tener diferentes efectos, pero en cualquier caso se presentará información sobre el error ocurrido. Si el código fuente está contenido en un fichero se indicará el nombre del mismo y la línea de código en la que se ha producido el error.

10.3.4.18.1 Léxicos

Los errores léxicos son originados durante el procesado del código fuente, cuando existe un carácter o cadena que no es reconocida por el intérprete. En este caso se detendrá el programa no será ejecutado y fallará el proceso de interpretación.

```
1  æ = 10;
2  Error: lexical error, unexpected character (æ)
```

Listing 10.137: Ejemplo error léxico

10.3.4.18.2 Sintácticos

Los errores sintácticos tienen lugar durante el procesado del código fuente, cuando una sentencia o expresión no está construida de forma correcta y no respeta las reglas establecidas por el lenguaje. Si se detecta un error sintáctico al procesarse el código fuente este no será ejecutado y el programa se detendrá.

```
1  [ id );
2  Error: syntax error, unexpected ')', expecting ',' or ']'
```

Listing 10.138: Ejemplo error sintáctico

10.3.4.18.3 Semánticos

Los errores semánticos se dan durante la ejecución del programa. Estos son dependientes del entorno y los datos.

Cuando una sentencia o expresión es interpretada es posible que se dé un error semántico si no es capaz de operar sobre los datos facilitados, o no se da el contexto adecuado.

Existe gran variedad de posibles errores semánticos, estos son descritos en este manual al tratar las sentencias, expresiones y demás construcciones en las que se pueden dar. Cuando se produce un error semántico puede detenerse o no la ejecución del programa, dependerá del error en sí. Si no se indica lo contrario en el manual, cuando se produce un error semántico, se prosigue con la ejecución del programa.

```
1  a = new noExist ()  
2  Error: data type error, worng class identifier
```

Listing 10.139: Ejemplo error semántico

OMI es un lenguaje de tipado dinámico, por lo que cualquier error de tipo se da en tiempo de ejecución. Las funciones y operadores del lenguaje esperan tratar con unos tipos de datos concretos. Si una función u operador recibe un valor de un tipo que no pueda manipular producirá un error semántico. En este manual se indica los tipos de datos con los que esperan operar las funciones y métodos, se entiende que otro tipo distinto producirá un error.

10.3.5. Funciones del lenguaje

10.3.5.1. Cadenas de caracteres

OMI pone a disposición del programador un conjunto de funciones que operan sobre cadenas de caracteres.

En OMI las cadenas de caracteres pertenecen a una clase de objetos, así toda función aplicable a una cadena puede ser llamada como un método de la cadena.

Toda función aplicable a una cadena comienza con el prefijo “str” y toman como primer parámetro la cadena sobre la que operar. Por otro lado los métodos de cadena no disponen de prefijo y se toma implícitamente como primer parámetro el objeto cadena.

10.3.5.1.1 str_explode

La función str_explode divide en partes una cadena de caracteres según una subcadena denominada separador. El resultado será un array de elementos que contendrá las cadenas correspondientes a cada una de las partes.

array str_explode (string str, string delimiter)

Parámetros:

str: Cadena de caracteres de entrada.

delimiter: Cadena delimitadora.

Valores devueltos:

Array de elementos que contiene las subcadenas resultado de dividir **str** usando el delimitador **delimiter**.

Si el separador no se encuentra en la cadena el resultado será un array de un elemento que contendrá la cadena completa.

```
1 // foo es el array {"A", "B", "C", "D", "E"}  
2 foo = str_explode ("A,B,C,D,E", ",");  
3  
4 // foo es el array {"A", "B", "C", "D", "E"}  
5 foo = "A,B,C,D,E"->explode (",");
```

Listing 10.140: Ejemplo str_explode

10.3.5.1.2 str_find

La función str_find obtiene la primera posición de una subcadena dentro de una cadena de caracteres.

numeric str_find (string str, mixed substr [, numeric position])

Parámetros:

str: Cadena de caracteres de entrada.

substr: Expresión de búsqueda. Puede ser una cadena de caracteres o una expresión regular.

position: Posición dentro de la cadena **str** a partir de la cual se comenzará la búsqueda.
Si no es dado se comenzará desde la primera posición de la cadena.

Valores devueltos:

Devuelve el número entero correspondiente a la primera posición en la que se encuentre **substr** dentro de **str**. Se ha de tener en cuenta que la primera posición de la cadena se corresponde con el entero 0.

Si **substr** es una expresión regular esta se corresponderá con la subcadena de mayor longitud perteneciente al lenguaje definido por esta.

Si la subcadena no se encuentra devuelve el valor -1.

Si la posición de inicio **position** es mayor que la longitud de la cadena **str** se devuelve el valor -1.

```
1 foo = str_find ("A,B,C,D,C,F", "C"); // foo vale 4  
2 foo = "A,B,C,D,C,F"->find ("C"); // foo vale 4  
3  
4 foo = str_find ("A,B,C,D,C,F", "C", 5); // foo vale 8  
5 foo = "A,B,C,D,C,F"->find ("C", 5); // foo vale 8  
6  
7 foo = str_find ("A,B,C,D,C,F", 'C(.*)C'); // foo vale 4  
8 foo = "A,B,C,D,C,F"->find ('C(.*)C'); // foo vale 4
```

Listing 10.141: Ejemplo str_find

10.3.5.1.3 str_replace

La función `str_replace` busca las ocurrencias de una subcadena dada en una cadena de caracteres, sustituyéndolas por una cadena de remplazo. El valor devuelto por la función es la cadena con los reemplazos efectuados.

string str_replace (string str, mixed search, string replace [, numeric n])

Parámetros:

str: Cadena de caracteres de entrada.

search: Expresión de búsqueda. Puede ser una cadena de caracteres o una expresión regular.

replace: Cadena de remplazo.

n: Número de reemplazos a efectuar. Si no es dado se realizan todos los reemplazos.

Valores devueltos:

Devuelve un cadena de caracteres resultado de remplazar la subcadena `search` por `replace` en la cadena de caracteres `str`, tantas veces como indique el parámetro `n`.

Si `search` es una expresión regular se buscarán subcadenas que pertenezcan al conjunto de palabras del lenguaje definido por esta. La expresión regular puede estar compuesta por subexpresiones delimitadas por paréntesis, si es así es posible hacer referencia a las subcadenas correspondientes a estas usando el carácter “\” seguido de la posición que ocupa la subexpresión.

```
1 // foo vale "A,B,P,D,P,F"
2 foo = str_replace ("A,B,C,D,C,F", "C", "P");
3
4 // foo vale "A,B,P,D,P,F"
5 foo = "A,B,C,D,C,F"->replace ("C", "P");
6
7 // foo vale "A,B,P,D,C,F"
8 foo = str_replace ("A,B,C,D,C,F", "C", "P", 1);
9
10 // foo vale "A,B,P,D,C,F"
11 foo = "A,B,C,D,C,F"->replace ("C", "P", 1);
12
13 // foo vale "A,B,D,F"
14 foo = str_replace ("A,B,C,D,C,F", "C,(.)", "|1");
15
16 // foo vale "A,B,D,F"
17 foo = "A,B,C,D,C,F"->replace ("C,(.)", "|1");
```

Listing 10.142: Ejemplo str_replace

10.3.5.1.4 str_replace_sub

La función `str_replace_sub` remplaza la subcadena dada por una posición inicial y una longitud dentro de un cadena de caracteres por otra cadena.

string str_replace_sub (string str, numeric ini, numeric len, string replace)

Parámetros:

str: Cadena de caracteres de entrada.
ini: Posición inicial para el remplazo.
len: Longitud de la subcadena de remplazo.
replace: Cadena de remplazo.

Valores devueltos:

Devuelve un cadena de caracteres resultado de remplazar la subcadena dada por la posición **ini** y la longitud **len** dentro de la cadena de caracteres **str** por la cadena **replace**.

Si la posición inicial dada se encuentra fuera de la cadena se produce un error semántico.

```
1 // foo vale "A,B,P,C,F"  
2 foo = str_replace_sub ("A,B,C,D,C,F", 4, 3, "P");  
3  
4 // foo vale "A,B,P,C,F"  
5 foo = "A,B,C,D,C,F"->replace_sub (4, 3, "P");
```

Listing 10.143: Ejemplo str_replace_sub

10.3.5.1.5 str_upper

La función **str_upper** convierte todos los caracteres alfabéticos de una cadena en mayúsculas.

string str_upper (string str)

Parámetros:

str: Cadena de caracteres de entrada.

Valores devueltos:

Devuelve la cadena de caracteres **str** con todos los caracteres alfabéticos convertidos a mayúsculas.

```
1 foo = str_upper ("a,b,c,D"); // foo vale "A,B,C,D"  
2 foo = "a,b,c,D"->upper (); // foo vale "A,B,C,D"
```

Listing 10.144: Ejemplo str_upper

10.3.5.1.6 str_lower

La función **str_lower** convierte todos los caracteres alfabéticos de una cadena en minúsculas.

string str_lower (string str)

Parámetros:

str: Cadena de caracteres de entrada.

Valores devueltos:

Devuelve la cadena de caracteres **str** con todos los caracteres alfabéticos convertidos a minúsculas.

```

1  foo = str_lower ("A,B,C,d"); // foo vale "a,b,c,d"
2  foo = "A,B,C,d"->lower (); // foo vale "a,b,c,d"

```

Listing 10.145: Ejemplo str_lower

10.3.5.1.7 str_search

La función str_search permite llevar a cabo una búsqueda aplicando un patrón sobre una cadena de texto. Se obtendrá un array con los resultados de la búsqueda.

array str_search (mixed str, regexp pattern [, string keys...])

Parámetros:

str: Cadena de entrada. Puede ser un array de cadenas.

pattern: Patrón de búsqueda.

keys...: Listado de claves para las coincidencias. Si no se da las claves serán numéricas.

Valores devueltos:

Devuelve un array que contiene todas las coincidencias del patrón dado por la expresión regular **pattern** en la cadena **str**, donde las claves del array vienen dadas por la lista de cadenas **keys...**

La expresión regular puede estar formada por subexpresiones delimitadas por "()". En dicho caso se buscará en la cadena **str** subcadenas que pertenezcan al conjunto delimitado por la expresión regular. Por cada subcadena encontrada se creará un array con las correspondencias de cada subexpresión.

Si **str** es un array de cadenas se aplicará el algoritmo de búsqueda de forma iterativa a cada cadena en el mismo.

```

1  atag = '(?i)<a[^>]*href\s*=\s*\"([^\"]*)\\"[^>]*>(.+?)<\s*/\s*a\s*>';
2  web = "<a href=|"url01|">link01 -> web01</a>";
3  web .= "|n<div><a id=|"id|" href=|"url02|" class=|"clase|">link02 -> web02</a></div>";
4  /*
5   links es el siguiente array:
6   {
7     {
8       "url" : "url01",
9       "label" : "link01 -> web01"
10    },
11    {
12      "url" : "url02",
13      "label" : "link02 -> web02"
14    }
15  }
16 */
17 links = str_search (web, atag, "url", "label");

```

Listing 10.146: Ejemplo str_search

Las cadenas de caracteres tienen el método search, que opera de igual forma que la función str_search, salvo por que no acepta la lista de claves.

10.3.5.1.8 str_match

La función str_match comprueba si una cadena de caracteres pertenece o no al lenguaje definido por una expresión regular.

```
bool str_match ( string str, regexp pattern)
```

Parámetros:

str: Cadena de entrada.

pattern: Expresión regular patrón.

Valores devueltos:

Devuelve un valor booleano verdadero si la cadena de caracteres str pertenece al conjunto de palabras del lenguaje definido por la expresión regular pattern. Falso en caso contrario.

```
1  foo = str_match ( "ABCD", 'A[^D]*D'); // foo vale true
2  foo = "ABCD"->match('A[^D]*D'); // foo vale true
```

Listing 10.147: Ejemplo str_match

10.3.5.1.9 regexp

La función regexp convierte una cadena de caracteres dada en una expresión regular

```
regexp regexp ( string str)
```

Parámetros:

str: Cadena de entrada.

Valores devueltos:

Devuelve la expresión regular correspondiente a la cadena str.

```
1  pattern = regexp ( "A[^D]*D");
2  foo = "ABCD"->match(pattern); // foo vale true
```

Listing 10.148: Ejemplo expresión regular

10.3.5.1.10 sprintf

La función sprintf permite obtener una cadena de caracteres formateada a partir de una serie de valores.

```
string sprintf ( string format [, mixed values...] )
```

Parámetros:

format:

La cadena formato contendrá una serie de directivas de formato. Estas directivas

serán sustituidas por el valor correspondiente, según posición, de la lista. Cuando se realiza cada sustitución el valor es formateado según la directiva.

Las directivas de formato tienen el siguiente forma:

$\%[operador][precisión][formato]$

Los posibles operadores serán los siguientes:

- +: Fuerza la impresión del símbolo + cuando se formatean números positivos.
- ^ : Convierte el caracteres a mayúsculas cuando se formatean cadenas de texto.
- #: Añade el carácter 0x cuando se formatean números hexadecimales y el carácter 0 cuando se formatean octales.

La precisión se refiere al número de decimales que se imprimirán en el caso de formatear números o el número de caracteres en el caso de formatear cadenas.

El carácter de formato indica que tipo de formato se le dará al valor:

- i|d: Número entero.
- u: Sin signo.
- f: Coma flotante.
- %: Carácter %.
- e: Notación científica.
- o: Octal.
- x: Hexadecimal.
- s|c: Cadena de texto.

values:

Lista de valores. Se deben de dar tantos como directivas existan en la cadena de formato.

Valores devueltos:

Devuelve la cadena de caracteres resultante de sustituir las directivas de formato de la cadena **format**, por los valores de la lista **values** formateados según la directiva correspondiente posicionalmente.

```
1 << sprintf("Esto es una %s formateada.", "cadena"); // Esto es una cadena formateada.
2 << sprintf("Ya son %2d", 2.003); // Ya son 02
3 << sprintf("Y %4s", "tres gatos"); // Y tres
4 << sprintf("Y %+4i", "4"); // Y +0004
5 << sprintf("Y %u", -5); // Y 5
6 << sprintf("Y %+2f", 6.6666 ); // Y +6.7
7 << sprintf("Y %+5e", 777.77777777); // Y +7.77778e+02
8 << sprintf("Y %F", "8888888888888888888888888888"); // Y 8.888889E+25
9 << sprintf("Y %3E", "99.99"); // Y 9.999E+01
10 << sprintf("Y %-s", "diez"); // Y DIEZ
11 << sprintf("%d en octal es %#o", 31, 31); // 31 en octal es 037
12 // 31 en hexadecimal es 0x1f o 0X1F
13 << sprintf("%d en hexadecimal es %#x o %#X", 31, 31, 31);
14 << sprintf("Texto sin expresiones de formato"); // Texto sin expresiones de formato
15 << sprintf("%2sdías %3stas", "Cadiz", "junio"); // Cadenas juntas
```

```

16 // El 100 % de aciertos en 100 intento
17 << sprintf("El %s %% de aciertos en %d intentos", 100, 100);
18 << sprintf("%%%%%%%%%"); // %%%%%%

```

Listing 10.149: Ejemplo sprintf

10.3.5.2. Arrays

OMI pone a disposición del programador un conjunto de funciones que operan sobre arrays. En OMI los arrays pertenecen a una clase de objetos, así toda función aplicable a un array puede ser llamada como un método del mismo.

Toda función aplicable a un array con el prefijo “array” y toman como primer parámetro el array sobre el que operar. Por otro lado los métodos de array no disponen de prefijo y se toma implícitamente como primer parámetro el objeto array.

10.3.5.2.1 array implode

Forma una cadena de caracteres a partir de un array de cadenas y una cadena separadora.

string array implode (array elements, string sep)

Parámetros:

elements: Array de elementos

sep: Cadena separadora

Valores devueltos:

Devuelve un string correspondiente a todos los elementos del array **elements** en el mismo orden, con el string **sep** entre cada elemento.

```

1 << array implode ({ "Hola", "Mundo"}, " "); // Imprime "Hola Mundo"
2 << { "Hola", "Mundo"}->implode (" "); // Imprime "Hola Mundo"

```

Listing 10.150: Ejemplo array implode

10.3.5.2.2 array first

Obtiene el primer elemento de un array.

mixed array first (array elements)

Parámetros:

elements: Array de elementos

Valores devueltos:

Devuelve el primer elemento del array **elements**. Si **elements** es un array numérico con claves numéricas devuelve el elemento con la posición 0. Si es un array asociativo devuelve el elemento correspondiente a la primera clave que se introdujo. Si el array es vacío devuelve un valor nulo.

```
1 << array_first ({ "Hola", "Mundo" }); // Imprime "Hola"
2 << { "Hola", "Mundo" }->first (); // Imprime "Hola"
```

Listing 10.151: Ejemplo array_first

10.3.5.2.3 array_last

Obtiene el último elemento de un array.

mixed array_last (array elements)

Parámetros:

elements: Array de elementos

Valores devueltos:

Devuelve el último elemento del array **elements**. Si **elements** es un array numérico con claves numéricas devuelve el elemento con la posición *size* – 1. Si es un array asociativo devuelve el elemento correspondiente a la última clave que se introdujo. Si el array es vacío devuelve un valor nulo.

```
1 << array_last ({ "Hola", "Mundo" }); // Imprime "Mundo"
2 << { "Hola", "Mundo" }->last (); // Imprime "Mundo"
```

Listing 10.152: Ejemplo array_last

10.3.5.2.4 array_insert

Inserta un elemento en una determinada posición de un array dado.

array array_insert (array &elements, numeric position, mixed element)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

position: Entero sin signo que determina la posición en la que se insertará el elemento. Debe ser un valor entero entre 0 y el tamaño del array, cualquier otro valor dará error de índice.

element: Elemento a insertar

Valores devueltos:

Devuelve el array resultado de insertar **element** en la posición **position** del array **elements**. Si **elements** es un array de índices numéricos inserta el elemento en la posición dada. Si es un array asociativo inserta el elemento en la clave que ocupa la posición dada, desplazando los demás valores y creando una nueva clave correspondiente al número de elementos.

```

1  /*
2   * foo vale el array { "Hola", "Mundo", "Digital" }
3   */
4  foo = array_insert ({ "Hola", "Mundo" }, 2, "Digital");
5  /*
6   * foo vale el array { "Hola", "Mundo", "Digital" }
7   */
8  foo = { "Hola", "Mundo" }->insert (2, "Digital");
9
10 foo = { "A", "C" };
11 array_insert (foo, 1, "B"); // foo vale {"A", "B", "C"}
12 foo->insert (3, "D"); // foo vale {"A", "B", "C", "D"}

```

Listing 10.153: Ejemplo array_insert

10.3.5.2.5 array_delete

Elimina el elemento ocupá una determinada posición en un array dado.

array array_delete (array &elements, mixed position)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

position: Puede ser un entero sin signo que determina la posición que será eliminada. Debe ser un valor entero mayor que 0 y menor que el tamaño del array, cualquier otro valor no tendrá efecto. Además puede ser una cadena de caracteres que represente la clave del array que será eliminada. Si la clave no existe en el array la función no realizará ninguna acción.

Valores devueltos:

Si el array **elements** presenta índices numéricos devuelve el array resultado de eliminar la posición **position** del array.

Si el array **elements** es asociativo devuelve el array resultado de eliminar la clave **position** del array.

```

1  /*
2   * foo vale el array { "Hola" }
3   */
4  foo = array_delete ({ "Hola", "Mundo" }, 1);
5  /*
6   * foo vale el array { "Hola" }
7   */
8  foo = { "Hola", "Mundo" }->delete (1);
9
10 foo = { "A", "C" };
11 array_delete (foo, 0); // foo vale {"C"}
12 foo->delete (0); // foo vale {}
13
14 foo = { 'key0' : 'val0', 'key1' : 'val1' };
15 foo->delete ('key0'); // foo vale { 'key1' : 'val1' }

```

Listing 10.154: Ejemplo array_delete

10.3.5.2.6 array_push

Inserta un elemento al final de un array dado.

array array_push (array &elements, mixed element)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

element: Elemento a insertar al final del array

Valores devueltos:

Devuelve el array **elements** con el elemento **element** insertado en la última posición. Si **elements** es asociativo la clave del elemento insertado será el número de elementos del array antes de la inserción.

```
1  /*
2   * foo vale el array { "Hola", "Mundo", "Digital" }
3   */
4  foo = array_push ({ "Hola", "Mundo", "Digital" });
5  /*
6   * foo vale el array { "Hola", "Mundo", "Digital" }
7   */
8  foo = { "Hola", "Mundo" }->push ("Digital");
9
10 foo = { "A", "B" };
11 array_push (foo, "C"); // foo vale { "A", "B", "C" }
12 foo->push ("D"); // foo vale { "A", "B", "C", "D" }
13
14 foo = { 'key0' : 'val0', 'key1' : 'val1' };
15 foo->push ('val2'); // foo vale { 'key0' : 'val0', 'key1' : 'val1', '2' : 'val2' }
```

Listing 10.155: Ejemplo array_push

10.3.5.2.7 array_pop

Elimina y devuelve el último elemento de un array dado.

mixed array_pop (array &elements)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

Valores devueltos:

Devuelve el último elemento del array **elements** y lo elimina del mismo. Si el array es asociativo devuelve el elemento de la última clave introducida.

```
1  /*
2   * foo vale la cadena "Mundo"
3   */
4  foo = array_pop ({ "Hola", "Mundo" });
5  /*
6   * foo vale la cadena "Mundo"
7   */
```

```

8     foo = { "Hola", "Mundo"}->pop ();
9
10    foo = { "A", "B"};
11    array_pop (foo); // foo vale { "A" }
12    foo->pop (); // foo vale { }
13
14    foo = { 'key0' : 'val0', 'key1' : 'val1'};
15    foo->pop (); // foo vale { 'key0' : 'val0' }

```

Listing 10.156: Ejemplo array_pop

10.3.5.2.8 array_unshift

Inserta un elemento al inicio de un array dado.

array array_unshift (array &elements, mixed element)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

element: Elemento a insertar al inicio del array

Valores devueltos:

Devuelve el array **elements** con el elemento **element** insertado en la primera posición. Si **elements** es asociativo la clave del elemento insertado será la misma que la anterior, llevándose a cabo un desplazamiento.

```

1  /*
2   * foo vale el array { "Digital", "Hola", "Mundo" }
3   */
4  foo = array_unshift ({ "Hola", "Mundo" }, "Digital");
5  /*
6   * foo vale el array { "Digital", "Hola", "Mundo" }
7   */
8  foo = { "Hola", "Mundo"}->unshift ("Digital");
9
10 foo = { "B", "C"};
11 array_unshift (foo, "A"); // foo vale { "A", "B", "C" }
12 foo->unshift (""); // foo vale { "", "A", "B", "C" }
13
14 foo = { 'key0' : 'val0', 'key1' : 'val1'};
15 foo->unshift ('val2'); // foo vale { 'key0' : 'val2', 'key1' : 'val0', '2' : 'val1' }

```

Listing 10.157: Ejemplo array_unshift

10.3.5.2.9 array_shift

Elimina y devuelve el primer elemento de un array dado.

mixed array_shift (array &elements)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

Valores devueltos:

Devuelve el primer elemento del array **elements** y lo elimina del mismo. Si el array es asociativo devuelve el elemento de la primera clave introducida.

```
1  /*
2   * foo vale la cadena "Hola"
3   */
4  foo = array_shift ({ "Hola", "Mundo" });
5  /*
6   * foo vale la cadena "Hola"
7   */
8  foo = { "Hola", "Mundo" }->shift ();
9
10 foo = { "A", "B" };
11 array_shift (foo); // foo vale { "B" }
12 foo->shift (); // foo vale { }
13
14 foo = { 'key0' : 'val0', 'key1' : 'val1' };
15 foo->shift (); // foo vale { 'key1' : 'val1' }
```

Listing 10.158: Ejemplo array_shift

10.3.5.2.10 array_reduce

Lleva a cabo la reducción de un array a un único valor, aplicando iterativamente una función.

mixed array_reduce (array elements, function iterative)

Parámetros:

elements: Array de elementos.

iterative: Función de iteración para la reducción. La función debe recibir dos parámetros donde el primero será el acumulador y el segundo el elemento de la iteración actual, y devolverá el acumulador para la próxima iteración. La iteración comienza con el primer elemento como valor del acumulador y el segundo como primer elemento para iterar.

Valores devueltos:

Devuelve el resultado de aplicar la función **iterative** iterativamente sobre los elementos del array **elements**.

```
1  /*
2   * foo vale la cadena "Hola Mundo Digital"
3   */
4  foo = array_reduce (
5    { "Hola", "Mundo", "Digital" },
6    ~(acumulador, elemento) {
7      return acumulador . " " . elemento;
8    }
9  );
10 /*
11  * foo vale la cadena "Hola Mundo Digital"
12 */
13 foo = { "Hola", "Mundo", "Digital" }->reduce (
14   ~(acumulador, elemento) {
```

```

15         return acumulador + ".elemento;
16     }
17 );

```

Listing 10.159: Ejemplo array_reduce

10.3.5.3. Procesos

OMI proporciona una serie de funciones para gestionarlos procesos. Así es posible abrir varios hilos de ejecución creando procesos.

Es posible enviar señales a los procesos creados, y estos podrán manejarlas según sea necesario.

10.3.5.3.1 exec

Ejecuta un comando del sistema y devuelve la salida.

string exec (string cmd)

Parámetros:

cmd: Cadena de caracteres que representa el comando a ejecutar

Valores devueltos:

Devuelve el contenido de la salida estándar al ejecutar el comando del sistema indicado por **cmd**.

```

1  /*
2   * foo vale el listado de procesos en ejecución
3   */
4  foo = exec ('ps -e');

```

Listing 10.160: Ejemplo exec

10.3.5.3.2 fork

Crea un proceso hijo del proceso en ejecución.

numeric fork ()

Valores devueltos:

Produce una bifurcación en la ejecución, creando un nuevo proceso copia del proceso actual.

La ejecución en el proceso hijo prosigue en la misma sentencia y con el mismo estado que el proceso padre.

El valor devuelto en el proceso padre será de el identificador del proceso hijo creado.

El valor devuelto en el proceso hijo será 0.

```

1  /*
2   * Se lleva a cabo la bifurcación de la ejecución mediante la
3   * creación de un proceso que imprimirá "Soy el proceso Hijo",
4   * mientras que en el proceso principal imprimirá

```

```

5   * "Soy el proceso Padre de PID" donde PID es el identificador
6   * del proceso hijo.
7   */
8   if (pid = fork ())
9     << "Soy el proceso Padre de " . pid;
10  else
11    << "Soy el proceso Hijo";

```

Listing 10.161: Ejemplo fork

10.3.5.3.3 wait

Espera la finalización de todos o algunos de los procesos hijos creados.

numeric **wait** ([*numeric* pid])

Parámetros:

pid: Identificador del proceso hijo cuya ejecución se desea esperar a que finalice. Si no es dado se esperará a que finalicen todos los procesos hijos.

Valores devueltos:

Devuelve el código de salida producido por el último proceso en finalizar.

```

1 //Se crea un proceso y se espera a que este finalice
2   if (pid = fork ()) {
3     wait( pid );
4     << "Finaliza proceso padre";
5   } else{
6     sleep (10);
7     << "Finaliza proceso hijo";
8   }

```

Listing 10.162: Ejemplo wait

10.3.5.3.4 getpid

Obtiene el identificador del proceso actual.

numeric **getpid** ()

Valores devueltos:

Devuelve el identificador del proceso actual

```

1 //Se crea un proceso y se imprimen los identificadores.
2   if (pid = fork ()) {
3     << "PID del Padre " . getpid () ; // Imprime el pid del padre
4     << "PID del Hijo " . pid; // Imprime el pid del hijo
5   } else {
6     << "PID del Hijo " . getpid () ; // Imprime el pid del hijo
7   }

```

Listing 10.163: Ejemplo getpid

10.3.5.3.5 getppid

Obtiene el identificador del proceso padre.

numeric getppid ()

Valores devueltos:

Devuelve el identificador del proceso padre del actual.

```
1  /*
2   * Se crea un proceso y se imprimen
3   * los identificadores.
4   */
5  if (pid = fork ()) {
6      << "PID del Padre " . getpid () ; // Imprime el pid del padre
7      << "PID del Hijo " . pid; // Imprime el pid del hijo
8  } else {
9      << "PID del Padre " . getppid () ; // Imprime el pid del padre
10     << "PID del Hijo " . getpid () ; // Imprime el pid del hijo
11 }
```

Listing 10.164: Ejemplo getppid

10.3.5.3.6 signal

Envía una señal a un proceso.

bool signal (numeric pid, numeric code)

Parámetros:

pid: Identificador de proceso al que se le envía la señal

code: Código numérico de señal POSIX

Valores devueltos:

Un valor booleano que indica si la señal se envió correctamente.

```
1  /*
2   * Se crea un proceso y envía una señal pasados
3   * unos segundos para finalizar su ejecución
4   */
5  if (pid = fork ()) {
6      << "Padre: Esperando para finalizar";
7      sleep (2);
8      << "Padre: Enviando señal";
9      signal (pid, 9);
10 } else {
11     << "Hijo: Esperando señal";
12     sleep (30);
13     << "Hijo: Finaliza normal",
14 }
```

Listing 10.165: Ejemplo signal

10.3.5.3.7 shandler

Permite establecer una función para manejar señales enviadas al proceso actual.

bool shandler (numeric code, function handler)

Parámetros:

code: Código numérico de señal POSIX a la que se le atribuirá el manejador.

handler: Función que manejará la ejecución cuando se dé la señal.

Valores devueltos:

Un valor booleano que indica si el manejador de señal se estableció correctamente.

```
1  /*
2   * Se crea un proceso y envía una señal
3   * que será manejada
4   */
5  if (pid = fork ()) {
6      << "Padre: Esperando para interrumpir";
7      sleep (2);
8      << "Padre: Enviando interrupción";
9      signal (pid, 2);
10 } else {
11     if (shandler (2, ~() { << "Interrupción"; })) {
12         sleep (30);
13         << "Hijo: Finaliza normal";
14     } else {
15         << "Error estableciendo manejador de señal";
16     }
17 }
```

Listing 10.166: Ejemplo shandler

10.3.5.3.8 exit_process

Finaliza el proceso en ejecución y todos los hijos de este. Si se da en el proceso principal se sale del programa.

void exit_process ()

```
1  /*
2   * Se crea un proceso y si el identificador de
3   * proceso es menor que 20000 se cierra.
4   */
5  if (pid = fork ()) {
6      sleep (30);
7  } else {
8      if (getpid () < 20000) {
9          exit_process ();
10     }
11     << "Exit";
12 }
```

Listing 10.167: Ejemplo exit_process

10.3.5.3.9 process

Realiza una llamada a función como un proceso.

bool process (function func [, mixed args ...])

Parámetros:

func: Función que será llamada como un proceso.

args...: Lista de parámetros que serán pasados a la función.

Valores devueltos:

Devuelve un valor verdadero si el proceso fue llamado con éxito.

```
1 process ( ~() { $(100) << "P1:". $; } );
2 process ( ~(i) { $(i) << "P2:". $; }, 100 );
```

Listing 10.168: Ejemplo process

10.3.5.4. Ficheros

OMI dispone de una serie de funciones para gestionar ficheros. Estas funciones permiten abrir un fichero, leer su contenido, escribir en el mismo, etc.

OMI presenta un tipo de dato especial que se corresponde con un puntero a fichero. Este tipo de dato sólo puede ser usado en las funciones de fichero, en cualquier otro contexto tomará un valor booleano que indica si el fichero se encuentra abierto.

Un puntero a fichero referencia una posición dentro del mismo, posición a partir de la cual se llevarán a cabo las operaciones de lectura/escritura. Esta posición puede ser manipulada.

OMI solo es capaz de operar directamente con ficheros en texto plano. No es posible crear ficheros de tipo binario que no sea texto plano.

10.3.5.4.1 file

Abre un fichero según el modo especificado.

file file (string name, string mode)

Parámetros:

name: Nombre del fichero que se desea abrir. Se puede utilizar una ruta absoluta o relativa al programa

mode: Modo en el que se abrirá el fichero:

r: Solo lectura

r+: Lectura y/o escritura

w: Escritura truncando el fichero

w+: Lectura y/o escritura truncando el fichero

a: Escritura posicionando el puntero al final del fichero

a+: Lectura y/o escritura posicionando el puntero al final del fichero

Todos los modos excepto el solo lectura crean el fichero si este no existe.

Valores devueltos:

Devuelve un puntero a fichero si el fichero ha sido abierto correctamente. Y un valor falso si no se ha abierto el fichero con éxito

```
1 if (f = file ("example.file", "w+")){
2     << "Fichero abierto con éxito";
3 } else {
4     << "Error abriendo el fichero";
5 }
```

Listing 10.169: Ejemplo file

10.3.5.4.2 fclose

Cierra un fichero abierto mediante la función “file”

bool fclose (file fpointer)

Parámetros:

fpointer: Puntero a fichero.

Valores devueltos:

Devuelve un valor booleano que indica si el fichero se ha cerrado correctamente.

```
1 if (f = file ("example.file", "w+")){
2     fclose (f);
3 } else {
4     << "Error abriendo el fichero";
5 }
```

Listing 10.170: Ejemplo fclose

10.3.5.4.3 fput

Escribe un contenido dado en la posición de lectura/escritura a la que apunta un puntero a fichero. Esta función presenta un alias “f>:”.

numeric fput|f>: (file fpointer, string content)

Parámetros:

fpointer: Puntero a fichero.

content: Cadena de caracteres que será escrita en el fichero a partir de la posición en la que se encuentre el puntero.

Valores devueltos:

Devuelve un valor numérico que representa los bytes que han sido escritos.

```

1   if (f = file ("example.file", "w+")){
2       f>: (f, "Contenido a escribir");
3       fclose(f);
4   } else {
5       << "Error abriendo el fichero";
6   }

```

Listing 10.171: Ejemplo fput

10.3.5.4.4 fget

Lee el un número de caracteres de un fichero, a partir de la posición de lectura/escritura dada por el puntero al mismo. Esta función presenta un alias “f<:”.

string fget|f<: (file fp [, numeric size])

Parámetros:

fpointer: Puntero a fichero.

size: Número de caracteres que serán leídos. Si no es dado se lee hasta el primer carácter de salto de línea encontrado o hasta el final del fichero

Valores devueltos:

Devuelve la cadena de caracteres leída.

```

1   if (f = file ("example.file", "w")){
2       f>: (f, "Contenido escrito");
3       fclose(f);
4   } else {
5       << "Error abriendo el fichero";
6   }
7   if (f = file ("example.file", "r")){
8       << f<:(f); // Imprime "Contenido escrito"
9       fclose (f);
10 } else {
11     << "Error abriendo el fichero";
12 }

```

Listing 10.172: Ejemplo fget

10.3.5.4.5 fseek

Cambia la posición de lectura/escritura de un puntero a fichero.

bool fseek (file fp [, numeric offset] [, enum position])

Parámetros:

fpointer: Puntero a fichero.

offset: Número de bytes que será desplazado el puntero

position: Posición de referencia desde la que mover el puntero. Puede ser algún valor FSET, FCUR o FEND, para referirse al inicio, la posición actual o el final. Si no es dado se toma como referencia el inicio del fichero.

Valores devueltos:

Devuelve un booleano que indica si la posición cambió correctamente. Si la posición nueva se encuentra fuera del fichero el puntero se colocará al final del mismo.

```
1 if (f = file ("example.file", "w+")){
2     f>:(f, "ABCD");
3     fseek (f, 2);
4     << f<:(f, 1); // Imprime "C"
5     fseek (f, -1, FEND);
6     << f<:(f, 1); // Imprime "D"
7     fseek (f, -1, FCUR);
8     << f<:(f, 1); // Imprime "C"
9     fseek (f, -1, FCUR);
10    << f<:(f, 1); // Imprime "B"
11    fseek (f, -1, FCUR);
12    << f<:(f, 1); // Imprime "A"
13    fclose(f);
14 } else {
15     << "Error abriendo el fichero";
16 }
```

Listing 10.173: Ejemplo fseek

10.3.5.4.6 ftell

Obtiene la posición de lectura/escritura a la que apunta un puntero a fichero.

numeric ftell (file fp pointer)

Parámetros:

fp pointer: Puntero a fichero.

Valores devueltos:

Devuelve un la posición del puntero al fichero como un número que representa el número de bytes desde el inicio del mismo.

```
1 if (f = file ("example.file", "w+")){
2     f>:(f, "ABCD");
3     fseek (f, 2);
4     << "Antes de leer: " . ftell (f); // Imprime 2
5     << f<:(f, 1); // Imprime "C"
6     << "Después de leer: " . ftell (f); // Imprime 3
7     fclose(f);
8 } else {
9     << "Error abriendo el fichero";
10 }
```

Listing 10.174: Ejemplo ftell

10.3.5.4.7 fread

Lee el contenido de un fichero.

string fread (mixed file)

Parámetros:

file: Fichero que será leído. Puede ser dado con ruta relativa al programa o absoluta, o como un puntero a fichero. Si el fichero no existe devolverá el valor nulo.

Valores devueltos:

Devuelve una cadena de caracteres que representa el contenido del fichero.

```
| 1  foo = fread ('/etc/hosts'); // foo vale el contenido del fichero /etc/hosts
```

Listing 10.175: Ejemplo fread

10.3.5.4.8 fwrite

Escribe el contenido de un fichero.

string fwrite (mixed file, string content)

Parámetros:

file: Fichero que será leído. Puede ser dado con ruta relativa al programa o absoluta, o como un puntero a fichero. Si el fichero no existe será creado.

content: Cadena de caracteres que será el contenido del fichero.

Valores devueltos:

Devuelve la cadena de caracteres escrita en el fichero.

```
| 1  /*
| 2   * El fichero /etc/hosts queda
| 3   * con el contenido:
| 4   * 127.0.0.1 localhost
| 5   */
| 6  fwrite ('/etc/hosts', '127.0.0.1 localhost');
```

Listing 10.176: Ejemplo fwrite

10.3.5.4.9 fappend

Escribe el contenido al final de un fichero.

string fappend (mixed file, string content)

Parámetros:

file: Fichero que será leído. Puede ser dado con ruta relativa al programa o absoluta, o como un puntero a fichero. Si el fichero no existe será creado.

content: Cadena de caracteres que será añadida al contenido del fichero.

Valores devueltos:

Devuelve la cadena de caracteres escrita en el fichero.

```

1  /*
2  * Al fichero /etc/hosts se le
3  * ha añadido el contenido:
4  * 127.0.0.1 localhost
5  */
6  fappend ('/etc/hosts', '127.0.0.1 localhost');

```

Listing 10.177: Ejemplo fappend

10.3.5.5. Fechas y tiempo

OMI dispone de un conjunto de funciones de tiempo que permiten operar con fechas y con marcas de tiempo.

10.3.5.5.1 date

Obtiene una fecha formateada.

string date (mixed format [, numeric timestamp])

Parámetros:

format: Cadena de formato. Establece el formato de la fecha mediante una serie de directivas.

%d: Día del mes con dos dígitos.

%j: Día del mes sin ceros iniciales.

%l: Día de la semana de forma alfabética completa.

%D: Día de la semana de forma alfabética y con tres letras.

%w: Día de la semana de forma numérica (0-domingo,6-sábado).

%z: Día del año de forma numérica.

%F: Mes de forma alfabética.

%m: Mes de forma numérica con dos dígitos.

%n: Mes de forma numérica sin ceros iniciales.

%M: Mes de forma alfabética con tres letras.

%Y: Año con cuatro dígitos.

%y: Año con dos dígitos.

%a: Periodo del día (am/pm) en minúsculas.

%A: Periodo del día (am/pm) en mayúsculas.

%g: Hora en formato 12h sin ceros iniciales.

%G: Hora en formato 24h sin ceros iniciales.

%h: Hora en formato 12h con dos dígitos.

%H: Hora en formato 24h con dos dígitos.

%i: Minutos con dos dígitos.

%U: Segundos desde la Época Unix (1 de Enero del 1970 00:00:00 GMT).

% %: Carácter %.

timestamp: Número entero que representa la marca de tiempo Unix a la que se desea dar formato. Si no es dado se toma el momento actual.

Valores devueltos:

Devuelve la cadena de caracteres correspondiente a la marca de tiempo **timestamp** formateada según **format**.

```
1 << date ("%d/%m/%Y %H:%i:%s"); // Fecha y hora actual
2 << date ("%d/%m/%Y %H:%i:%s", 0); // 01/01/1970 00:00:00
```

Listing 10.178: Ejemplo date

10.3.5.5.2 time

Obtiene la marca de tiempo Unix actual.

numeric time ()

Valores devueltos:

Devuelve el entero correspondiente a la marca de tiempo Unix actual.

```
1 << time(); // Imprime la marca de tiempo Unix actual
2 << date ("%d/%m/%Y %H:%i:%s", time()); // Fecha y hora actual
```

Listing 10.179: Ejemplo time

Esta es una función interna de OMI que puede ser utilizada sin paréntesis.

10.3.6. Extensiones del lenguaje

OMI es un lenguaje modular que puede ser extendido. Los módulos o extensiones permiten añadir funciones al lenguaje, ampliando así su funcionalidad y uso.

Los módulos son bibliotecas dinámicas que son cargadas en tiempo de ejecución. Estas han sido programadas en un lenguaje de más bajo nivel (C/C++) y compiladas a código máquina. Es posible obtener módulos de OMI desde la web del proyecto (<http://www.omi-project.com/download/extensions>).

10.3.6.1. Carga de módulos

Para cargar un módulo desde el código fuente se puede utilizar la sentencia *load* seguida de una cadena entre paréntesis que representa la biblioteca dinámica que se usará como módulo.

```
1 load ('libomi_gettext.so');
2 <<_('cadena traducida');
```

Listing 10.180: Ejemplo carga de módulos

Se dispone de un mecanismo para autocargar módulos de extensiones del lenguaje. Al iniciarse el intérprete se cargarán todos los módulos indicados en el fichero “libs.ini” localizado en el directorio de datos de la aplicación (por defecto /usr/local/share/omi). Cada línea de este fichero será un módulo que será cargado, exceptuando las líneas que comiencen por “;” que son consideradas comentarios.

10.3.6.2. Creación de módulos

OMI permite al programador extender el lenguaje haciendo uso de una biblioteca para el desarrollo de módulos. Esta dispone de los elementos sobre los que se construye el intérprete.

Para construir un módulo se ha de disponer de la biblioteca OMI correctamente instalada. Para ello se dispone de dos opciones:

Paquete .deb: [http://www.omi-project.com/download/deb/libomi-dev_\\$.VERSION.deb](http://www.omi-project.com/download/deb/libomi-dev_$.VERSION.deb)

Código fuente: A partir del código fuente de OMI se puede construir e instalar la biblioteca mediante las reglas *dev* e *install-dev*

Un módulo OMI está escrito en C++ y se define mediante unas series de clases cada una de las cuales representará una función que se desea añadir.

El intérprete OMI se construye mediante una colección de nodos ejecutables. Todo es un nodo ejecutable: los booleanos, los números, las variables, las funciones... Existen diferentes niveles de abstracción en los nodos ejecutables, así por ejemplo un número es representado mediante un nodo ejecutable numérico, que a su vez es un nodo expresión aritmética, que a su vez es una expresión de tipo definido... Las clases de un módulo extenderán los tipos de nodos definidos por la biblioteca de desarrollo OMI, normalmente nodos expresiones.

Las clases que integran un modulo OMI deben cumplir lo siguiente:

- Ser un tipo de nodo ejecutable.
- Ser un nodo extensión e inheritar su constructor.
- Implementar un método *run* que lleve a cabo la funcionalidad.
- Ser añadidas al cargador de módulos.

Dependiendo el tipo de nodo ejecutable que se tome de base se dispondrá de un atributo que guardará el valor interno del nodo. Así un nodo de tipo cadena de caracteres tendrá un atributo que representa la cadena guardada. El método *run* llevará a cabo la funcionalidad y dará valor a este atributo.

Normalmente el método *run* obtendrá los parámetros con los que se ha llamado a la función, les aplicará la operativa asociada y atribuirá el valor interno del nodo. Mediante una llamada al método *getParams* es posible obtener un vector con los parámetros pasados a la función en una ejecución.

```
1 // libomi_strings.h
2 #include <omi/omi.h>
3
4 class ucfirstNode : public stringNode, public extensionNode {
5     public:
6         using extensionNode::extensionNode; // Si std=c++11
7         // ucfirstNode (listNode* list) : extensionNode(list) {} //Si std=c++98
8         void run ();
9 }
```

Listing 10.181: Ejemplo cabecera de módulo

```
1 // libomi_strings.cpp
2 #include "libomi_strings.h"
3
```

```

4   void ucfirstNode::run () {
5     vector<runNode*> v = this->getParams ();
6     if (v.size() == 1) {
7       strvalue_ = stringNode::to_str (v[0]);
8       strvalue_[0] = toupper(strvalue_[0]);
9     } else
10      throw errorException(
11        "One parameter is required",
12        "ucfirstNode::run, size " + to_string (v.size()),
13        ERROR_PARAMS
14      );
15  }
16
17 extern "C" void load (PluginsLoader* loader) {
18   loader->add ("ucfirst", create<ucfirstNode>());
19 }
```

Listing 10.182: Ejemplo implementación de módulo

```

1  $PS1 # Compilación del módulo
2  $PS1 g++ -c -fPIC \
3  $PS2   -DNUMTYPE='double' -DNUMPRECISION='15' -DREFCTYPE='unsigned int' \
4  $PS2   -ansi -pedantic -g -std=c++11 libomi_strings.cpp -o libomi_strings.o
5  $PS1 g++ -g -O2 -shared -rdynamic -g -o libomi_strings.so libomi_strings.o -lomi
```

Listing 10.183: Ejemplo compilación de módulo

```

1 // user.omi
2 load ("./libomi_strings.so")
3 << ucfirst ("example"); // Imprime "Example"
```

Listing 10.184: Ejemplo carga de módulo

10.3.7. Arquitectura cliente/servidor

El intérprete OMI puede funcionar bajo una arquitectura cliente/servidor para ello solo se ha de ejecutar este con la opción “-s” y el puerto de escucha.

En este caso el intérprete leerá por el puerto establecido una cadena de caracteres que representa el código a interpretar. El intérprete procesará la cadena y producirá un resultado.

10.3.8. Descripción del proceso de interpretación

Es posible iniciar el intérprete de forma que produzca una salida en formato JSON que describa los procesos de interpretación llevados a cabo. La estructura de datos JSON devuelta es descrita en su sección correspondiente de diseño del software.

Cuando se ejecuta el intérprete en de esta forma en conjunción con una arquitectura cliente/servidor esta estructura es devuelta por el puerto establecido.

10.3.9. runTree

runTree es un cliente para el intérprete OMI. Lee la estructura JSON producida por este y la representa en una interfaz gráfica.

La interfaz de runTree se divide como sigue:

- Árbol de ejecución: Describe el árbol sintáctico producido y visualiza su procesamiento.
- Tablas de símbolos: Describe el contenido de las tablas de símbolos.
- Código fuente: Permite al usuario introducir el código fuente que será interpretado.
- Consola de entrada/salida: Muestra la salida producida y la recoge la entrada del usuario.
- Consola de estado: Describe el estado del proceso o de los elementos seleccionados.
- Consola de mandos: Contiene las opciones de control del proceso (avanzar, repetir, cargar código y guardar).

Capítulo 11

Conclusiones

En esta sección se detallan las lecciones aprendidas tras el desarrollo del proyecto OMI y se identifican las posibles mejoras futuras sobre el desarrollo del software.

11.1. Objetivos alcanzados

Como se introdujo se ha construido un intérprete completo de un lenguaje de programación de alto nivel, capaz de describir mediante estructuras de datos el proceso de interpretación llevado a cabo. También se ha construido un cliente web capaz de interpretar estos datos y representarlos gráficamente y de forma interactiva.

Se ha recorrido todo el proceso de desarrollo y construcción del proyecto. La documentación generada ha sido dispuesta en una plataforma web que permite su acceso y navegación de una forma fácil y cómoda.

11.2. Lecciones aprendidas

- Se ha profundizado en los estudios de los intérpretes de lenguajes de formales y los conceptos que los hacen posibles tales como los autómatas, el léxico, la sintaxis, las gramáticas, los autómatas, los árboles...
- Se han estudiado y asimilado características avanzadas de los lenguajes de programación y cómo estas se implementan.
- Se ha profundizado en otros lenguajes de programación para construir una visión amplia de características y paradigmas (PHP, Python, JavaScript, Java, Haskell, Ruby...).
- Se han afianzados nuevos conocimientos en el desarrollo C++ y algunas características avanzadas de este, como la inclusión de bibliotecas dinámicas, bibliotecas de líneas de comandos o las directivas del compilador.
- Se ha estudiado y profundizado en herramientas de compilación automática y la paquetización de binarios precompilados.
- Se ha profundizado en la instalación y configuración de servidores.
- Se han estudiado estructuras de datos estándar utilizadas en comunicaciones como JSON.
- Se ha profundizado en el proceso de desarrollo de un proyecto software y las metodologías aplicadas.

- Se ha profundizado en el desarrollo de diagramas y el lenguaje UML.
- Se han estudiado estándares ISO para asegurar la calidad de proyectos.
- Se ha adquirido conocimientos más afianzados en los distintos tipos de pruebas aplicables al software.
- Se ha profundizado en la planificación y desarrollo de tareas.
- Se ha profundizado en el desarrollo de webs dinámicas escritas en PHP.
- Se ha adquirido nuevos conocimientos en tecnologías de navegadores tales como HTML5 y JavaScript.
- Se ha adquirido destreza para el trabajo en equipo y una comunicación óptima.

11.3. Trabajo futuro

- Añadir características pertenecientes a los paradigmas abordados como número de parámetros arbitrarios, acceso a variables no locales, herencia múltiple, traits, prototipos...
- Añadir características de otros paradigmas no abordados como la programación dirigida por eventos, aspectos, lógica ...
- Añadir características avanzadas al intérprete y su funcionamiento interno como la compilación justo a tiempo.
- OMI es un lenguaje de ámbito general, pero puede ser especializado para cumplir un propósito más concreto como la explotación de servicios web o la creación de juegos.
- Desarrollar en mayor detalle una descripción del proceso de interpretación entrando en materias como el análisis léxico y sintáctico llevado a cabo.
- Integrar un DSL para la creación de gramáticas dentro del propio lenguaje.
- Construir bibliotecas de funciones útiles que añadan recursos matemáticos, de gestión de bases de datos...

Capítulo 12

Anexos

12.1. Calculadora

```
1#!/usr/local/bin/omi
2#Calculadora sencilla.
3while ( true ) {
4    << "=====";
5    << "Calculadora";
6    << "Dame un número";
7    >> a;
8    << "Dame otro";
9    >> b;
10   << "Dame una operación [0=>suma], [1=>resta], [2=>multi], [3=>divide], [otro=>sale]";
11   >> op;
12   if (op == 0)
13       << a << " + " << b << " = " << ( a + b );
14   elif (op == 1)
15       << a << " - " << b << " = " << (a - b);
16   elif (op == 2)
17       << a << " * " << b << " = " << (a * b);
18   elif (op == 3) {
19       if (op == 0)
20           << "Error: no es posible dividir entre 0";
21       else
22           << a << " / " << b << " = " << (a / b);
23   }
24   else {
25       << "Adiós";
26       break;
27   }
28 }
```

Listing 12.1: Calculadora

12.2. Cuestionarios

El diagrama de clases que ilustra el diseño del sistema de cuestionarios es el siguiente:

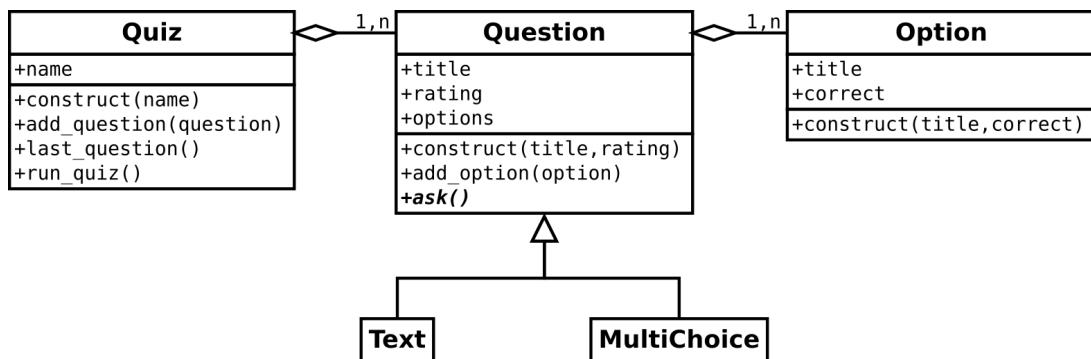


Figura 12.1: Diseño cuestionarios

Este programa ha sido modelado usando programación orientada a objetos. Se presenta un fichero de código fuente por cada clase que conforma el sistema de cuestionarios, otro fichero la definición del DSL junto con el flujo principal que conforma el motor de cuestionarios, y otro fichero que se corresponde con un cuestionario de ejemplo.

```

1#!/usr/local/bin/omi
2#Sistema de cuestionarios
3#=====
4include "quiz.class.omi";
5#=====
6global quiz;
7#=====
8#Funciones del DSL
9~multichoice (text, rating) {
10    quiz->add_question (new MultiChoice (text, rating));
11}
12
13~text (text, rating) {
14    quiz->add_question (new Text (text, rating));
15}
16
17~option (text, correct = null) {
18    quiz->last_question()->add_option(new Option (text, correct));
19}
20#=====
21if (args[1]){
22    title = args[2]?:"Sin titulo";
23    << title;
24    quiz = new Quiz (title);
25    include args[1];
26    quiz->run_quiz();
27}
28else
29    << "Debe indicar un cuestionario";
30#=====
```

Listing 12.2: Cuestionarios: quiz.omi

```

1#quiz.class.omi
2#=====
3include "question.class.omi";
4#=====
5class Quiz {
6    name = '';
7    questions = {};
8}
```

```

8     ~ Quiz (name) {
9         this->name = name;
10    }
11    ~ add_question (question) {
12        this->questions[size this->questions] = question;
13    }
14    ~ last_question () {
15        return this->questions [(size this->questions) - 1];
16    }
17    ~ run_quiz () {
18        count = 0;
19        total = 0;
20        $(this->questions) {
21            if ($->ask ()) count += $->rating;
22            total += $->rating;
23        }
24        << "Tienes " << count << " respuestas correctas de " << total;
25    }
26}
27 # =====

```

Listing 12.3: Cuestionarios: quiz.class.omi

```

1 #question.class.omi
2 #=====
3 include "option.class.omi";
4 #=====
5 class Question {
6     title = "";
7     rating = 0;
8     options = {};
9     ~ Question (title, rating) {
10        this->title = title;
11        this->rating = rating;
12    }
13    ~ add_option (option) {
14        this->options [(size this->options)] = option;
15    }
16}
17 #
18 class Text extends Question {
19     ~ Text (title, rating) {
20         this->Question (title, rating);
21     }
22     ~ ask () {
23         << "";
24         << this->title << " (" << this->rating << ")";
25         << "Introducir respuesta: ";
26         >> ans;
27         n = size (this->options);
28         for (i = 0; i < n; ++i) {
29             if (ans == this->options[i]->title)
30                 return true;
31         }
32         return false;
33     }
34}
35 #
36
37 class MultiChoice extends Question {
38     ~ MultiChoice (title, rating) {
39         this->Question (title, rating);
40     }
41     ~ ask () {
42         << this->title << " (" << this->rating << ")";
43         count = 0;
44         $(this->options) << (++count) << " - " << $->title;

```

```

45     << "Introducir respuesta: ";
46     >> ans;
47     return (this->options[ans - 1]) && this->options[ans - 1]->correct;
48 }
49 }
50 # =====

```

Listing 12.4: Cuestionarios: question.class.omi

```

1 #option.class.omi
2 #=====
3 class Option {
4     title = "";
5     correct = "";
6     ~Option (title, correct){
7         this->title = title;
8         this->correct = correct;
9     }
10 }
11 #=====

```

Listing 12.5: Cuestionarios: option.class.omi

```

1 #Ejemplo de cuestionario.
2 multichoice("Cuanto tiempo duro la guerra de los 100 años?", 2.5);
3 option (100, false);
4 option (116, true);
5 option (90, false);
6 option (102, false);
7 multichoice("Un símil es ...", 2.5);
8 option ("Una comparación", true);
9 option ("Una duda", false);
10 option ("Un aparato para medir el tiempo", false);
11 text ("En que provincia desemboca el río Guadalquivir?", 2.5);
12 option ("Cádiz", true);
13 option ("cádiz", true);

```

Listing 12.6: Cuestionarios: ejemplo

Para ejecutar el cuestionario en una terminal de comandos:

```
1 prompt$ ./quiz_system.omi quiz.q "Cuestionario"
```

Listing 12.7: Cuestionarios: ejecución

12.3. Tic-Tac-Toe

El programa se divide en tres módulos, cada uno correspondiente a un fichero. En un fichero se disponen las funciones de entrada y salida, tales como solicitar los datos de los jugadores, imprimir el tablero, etc. En otro fichero se encuentran las funciones de inteligencia artificial para el caso de jugadores de tipo máquina. Y en otro se encuentra el flujo principal correspondiente al bucle de juego.

```

1 # IO.omi
2 # =====
3 ~ IOJugadores () {
4     jugadores = {};
5     for (i = 1; i <= 2; ++i) {
6         << "Nombre Jugador " << i;
7         >> nombre;
8         << "Tipo Jugador " << i << " [ 0 => Humano, otro => Maquina ] ";
9         >> tipo;
10        if (tipo != 0)
11            tipo = 1;
12        jugadores [] = {
13            'nombre' : nombre,
14            'tipo' : tipo,
15            'token' : (( i == 1)?1:-1),
16        };
17    }
18    return jugadores;
19 }
20 #
21 ~ IOTablero (tablero) {
22     for (i = 0; i < 3; ++i)
23         << IOToken(tablero[i][0]) << " or "
24         << IOToken(tablero[i][1]) << " or "
25         << IOToken(tablero[i][2]);
26 }
27 #
28 ~ IOToken (pos) {
29     if (pos == 1)
30         return 'X';
31     elif (pos == -1)
32         return 'O';
33     else
34         return '#';
35 }
36 #
37 ~ IOMover (tablero) {
38     do {
39         << "Dame la fila";
40         >> row;
41         << "Dame la columna";
42         >> col;
43     }while (tablero[row][col] != 0);
44     return {row, col};
45 }
46 #
47 ~ IOGanador (jugadores, ganador) {
48     if (ganador == 1 ){
49         << "El ganador es " << jugadores[0]["nombre"];
50     }elif (ganador == -1){
51         << "El ganador es " << jugadores[1]["nombre"];
52     }else{
53         << "La partida ha quedado en empate";
54     }
55 }
56 #
57 # =====

```

Listing 12.8: Tic-Tac-Toe: IO.omi

```

1 # AI.omi
2 # =====
3 ~ primerosMov (tablero) {
4     if (!tablero[1][1])
5         return {1,1};
6     do {

```

```

7     col = row = time () % 2;
8     if (row == 1) row = 2;
9     if (col == 1) col = 2;
10    } while (tablero[row][col] != 0);
11    return {row, col};
12}
13#-----
14~ miniMax (A, turno){
15    mejor = turno * -1;
16    minMov = 9;
17    poda = 1;
18    Mov = 0;
19    posicion = {0, 0};
20    if (! (t_ganador = procesarTablero (A)) && ! tableroLleno (A)){
21        for (cont = 0; cont < 3 && poda; cont ++){
22            for (cont2 = 0; cont2 < 3 && poda; cont2 ++){
23                if ( A [cont] [cont2] == 0){
24                    A [cont] [cont2] = turno;
25                    actual = miniMax_R (A, turno * -1, 0, Mov);
26                    if (turno == 1 ){
27                        if ( actual >= mejor && Mov <= minMov){
28                            mejor =actual;
29                            posicion [0] = cont;
30                            posicion [1] = cont2;
31                            if (mejor == turno){
32                                minMov = Mov;
33                                if (mejor == 1 && minMov == 0)
34                                    poda = 0;
35                            }
36                        }
37                    } else
38                        if ( actual <= mejor && Mov <= minMov){
39                            mejor =actual;
40                            posicion [0] = cont;
41                            posicion [1] = cont2;
42                            if (mejor == turno){
43                                minMov = Mov;
44                                if (mejor == 1 && minMov == 0)
45                                    poda = 0;
46                            }
47                        }
48                    A [cont] [cont2] = 0;
49                }
50            }
51        }
52        return posicion;
53    }
54#-----
55~ miniMax_R (&A, turno, nMov, &Mov){
56    mejor = turno * -1;
57    poda = 1;
58    minMov = 9 ;
59    if (! (t_ganador = procesarTablero (A)) && ! tableroLleno (A)){
60        for (cont = 0; cont < 3 && poda; cont ++){
61            for (cont2 = 0; cont2 < 3 && poda; cont2 ++){
62                if ( A [cont] [cont2] == 0){
63                    A [cont] [cont2] = turno;
64                    actual = miniMax_R (A, turno * -1, nMov +1, Mov);
65                    if (turno == 1 ){
66                        if ( actual >= mejor && Mov <= minMov){
67                            mejor =actual;
68                            if (mejor == turno){
69                                minMov = Mov;
70                                if (mejor == 1 && minMov == 0)
71                                    poda = 0;
72                            }
73                        }
74                    } else
75                        if ( actual <= mejor && Mov <= minMov){
76                            mejor =actual;

```

```

77         if (mejor == turno){
78             minMov = Mov;
79             if (mejor == 1 && minMov == 0)
80                 poda = 0;
81         }
82     }
83     A [cont] [cont2] = 0;
84 }
85 }
86 Mov = minMov;
87 return mejor;
88 }
89 Mov = nMov;
90 return t_ganador;
91 }
92 #
93 ~procesarTablero (A) {
94     ganador = 0;
95     cont = 0;
96     for (cont = 0; cont < 3 && ganador == 0; cont ++){
97         if (A [cont] [0] == A[cont] [1] && A [cont] [1] == A [cont] [2])
98             ganador = A [cont] [1];
99     }
100    for (cont = 0; cont < 3 && ganador == 0; cont ++){
101        if (A [0] [cont] == A [1] [cont] && A [1] [cont] == A [2] [cont])
102            ganador = A [0] [cont];
103        if (A [0][0] == A [1][1] && A [1][1] == A [2][2] && ganador == 0 )
104            ganador = A [0][0];
105        if (A [0][2] == A [1][1] && A [1][1] == A [2][0] && ganador == 0 )
106            ganador = A [0][2];
107    }
108    return ganador;
109 }
110 #
111 ~ tableroLleno (A){
112     resp = 1;
113     for (cont = 0; cont < 3 && resp; cont ++){
114         for (cont2 = 0; cont2 < 3 && resp; cont2 ++)
115             resp = A [cont] [cont2] != 0;
116     }
117 }
118 =====

```

Listing 12.9: Tic-Tac-Toe: AI.omi

```

1#!/usr/local/bin/omi
2=====
3#include "IO.omi";
4#include "AI.omi";
5#
6~juego () {
7    tablero = {{0,0,0},{0,0,0},{0,0,0}};
8    posicion = {0,0};
9    turno = 0;
10   jugadores = IOJugadores ();
11   while (!(ganador = procesarTablero (tablero)) && !tableroLleno (tablero)){
12       << "-----";
13       IOTablero (tablero);
14       << "|nTurno " << jugadores[turno%2]['nombre'];
15       if (jugadores[turno%2]['tipo'] == 0) {
16           posicion = IOMover (tablero);
17       } else {
18           << "Calculando movimiento... ";
19           if (turno <= 1) {
20               posicion = primerosMov (tablero);
21           } else {
22               posicion = miniMax (tablero, jugadores[turno%2]['token']);

```

```

23         }
24     }
25     tablero[posicion[0]][posicion[1]] = jugadores[turno%2]['token'];
26     turno++;
27 }
28 IOTablero (tablero);
29 IOGanador (jugadores, ganador);
30 }
31 #-----
32 juego ();
33 #=====

```

Listing 12.10: Tic-Tac-Toe: tic-tac.toe.omi

12.4. Fibonacci

```

1#!/usr/local/bin/omi
2#fibonacci.omi
3#=====
4~ fibonacci (n) {
5    if (n == 1 n == 2)
6        return 1;
7    else
8        return fibonacci (n - 1) + fibonacci (n - 2);
9}
10#-----
11<< fibonacci (args[1]);
12#=====

```

Listing 12.11: Fibonacci

12.5. N-body

```

1#!/usr/local/bin/omi
2#=====
3~ energy(&b) {
4    e = 0.0;
5    m = size (b);
6    for (i=0; i < m; ++i) {
7        b1=b[i];
8        e += 0.5*b1[6]*(b1[3]*b1[3]+b1[4]*b1[4]+b1[5]*b1[5]);
9        for (j=i+1; j<m; j++) {
10            b2=b[j];
11            dx=b1[0]-b2[0]; dy=b1[1]-b2[1]; dz=b1[2]-b2[2];
12            e -= (b1[6]*b2[6])/sqrt(dx*dx + dy*dy + dz*dz);
13        }
14    }
15    return e;
16}
17
18pi=3.141592653589793;
19solar_mass=4*pi*pi;
20days_per_year=365.24;
21
22bodies = [
23    {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, solar_mass }, //Sun
24    {
25        4.84143144246472090, //Jupiter

```

```

26     -1.16032004402742839,
27     -0.103622044471123109,
28     0.00166007664274403694 * days_per_year,
29     0.00769901118419740425 * days_per_year,
30     -0.0000690460016972063023 * days_per_year,
31     0.0009.54791938424326609 * solar_mass
32 },
33 {
34     8.34336671824457987, // Saturn
35     4.12479856412430479,
36     -0.403523417114321381,
37     -0.00276742510726862411 * days_per_year,
38     0.00499852801234917238 * days_per_year,
39     0.00002.30417297573763929 * days_per_year,
40     0.000285885980666130812 * solar_mass
41 },
42 {
43     12.8943695621391310, // Uranus
44     -15.1111514016986312,
45     -0.223307578892655734,
46     0.00296460137564761618 * days_per_year,
47     0.00237847173959480950 * days_per_year,
48     -0.0000296589568540237556 * days_per_year,
49     0.0000436624404335156298 * solar_mass
50 },
51 {
52     15.3796971148509165, // Neptune
53     -25.9193146099879641,
54     0.179258772950371181,
55     0.00268067772490389322 * days_per_year,
56     0.00162824170038242295 * days_per_year,
57     -0.0000951592254519715870 * days_per_year,
58     0.0000515138902046611451 * solar_mass
59 }
60 };
61
62 // offset_momentum
63 px=py=pz=0.0;
64 for (bodies as e) {
65     px+=e[3]*e[6];
66     py+=e[4]*e[6];
67     pz+=e[5]*e[6];
68 }
69 bodies[0][3] = -1 * (px) / solar_mass;
70 bodies[0][4] = -1 * (py) / solar_mass;
71 bodies[0][5] = -1 * (pz) / solar_mass;
72
73 pairs = {};
74 m=size(bodies);
75 for (i=0; i<m; ++i)
76     for (j=i+1; j<m; j++)
77         pairs[] = {bodies[i], bodies[j]};
78
79 n = args[1];
80
81 << energy(bodies);
82
83 i=0;
84 do {
85     for (pairs as p) {
86         a=p[0]; b=p[1];
87         dx=a[0]-b[0]; dy=a[1]-b[1]; dz=a[2]-b[2];
88
89         dist = sqrt(dx*dx + dy*dy + dz*dz);
90         mag = 0.01/(dist*dist*dist);
91         mag_a = a[6]*mag; mag_b = b[6]*mag;
92
93         a[3]-=dx*mag_b; a[4]-=dy*mag_b; a[5]-=dz*mag_b;
94         b[3]+=dx*mag_a; b[4]+=dy*mag_a; b[5]+=dz*mag_a;
95     }

```

```
96     for (bodies as b) {
97         b[0] += 0.01 * b[3]; b[1] += 0.01 * b[4]; b[2] += 0.01 * b[5];
98     }
99 }
100 } while (++i < n);
102
103 << energy(bodies);
```

Listing 12.12: N-body

Bibliografía

Referencias bibliográficas:

- Compiladores y procesadores del lenguaje (José Antonio Jiménez Millám) [Servicios publicaciones universidad de Cádiz].
- Compiladores. Principios, técnicas y herramientas 2^aed. (Alfred Aho, Ravi Sethi, Jeffrey Ullman y Monica S. Lam) [Pearson Addison Wesley].
- El lenguaje unificado de modelado (Booch, Rumbaugh y Jacobson) [Pearson Addison Wesley].

Portales webs sobre lenguajes de programación:

- isocpp.org: Estándar C++.
- cplusplus.com: C++ referencias y tutoriales.
- docs.oracle.com: Java Platform, Standard Edition.
- php.net: PHP documentación y referencias.
- python.org: Python documentación y referencias.
- ruby-lang.org: Ruby documentación y referencias.
- w3schools.com: JavaScript tutoriales y referencias W3C.
- developer.mozilla.org: JavaScript tutoriales y referencias Mozilla.
- nodejs.org: Node.js documentación y referencias.
- uam.es: Manual básico LISP.
- haskell.org: Haskell documentación y referencias.
- swi-prolog.org: Prolog documentación y referencias.
- perl.org: Perl documentación y referencias.
- scala-lang.org: Scala documentación y referencias.

Otros recursos generales:

- wikipedia.org
- stackoverflow.com

Información sobre Licencia

Información sobre Licencia

Todo el software desarrollado para el proyecto OMI se encuentra bajo licencia GPLv3. Por otro lado toda la documentación se encuentra bajo Creative Commons.

GPLv3

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

The GNU General Public License is a free, copyleft license for software and other kinds of works. The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such

abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users’ Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to

limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
- c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
- d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a

fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the

party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you

are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <text><year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <<http://www.gnu.org/licenses/>>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program> Copyright (C) <year> <name of author>
```

```
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

Creative Commons



Usted es libre para:

Compartir — copiar y redistribuir el material en cualquier medio o formato.

Adaptar — remezclar, transformar y crear a partir del material.

El licenciatario no puede revocar estas libertades en tanto usted siga los términos de la licencia



Atribución — Usted debe darle crédito a esta obra de manera adecuada, proporcionando un enlace a la licencia, e indicando si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo del licenciatario.



NoComercial — Usted no puede hacer uso del material con fines comerciales.



CompartirIgual — Si usted mezcla, transforma o crea nuevo material a partir de esta obra, usted podrá distribuir su contribución siempre que utilice la misma licencia que la obra original.

No hay restricciones adicionales — Usted no puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otros hacer cualquier uso permitido por la licencia.