GALGAS

Version 3.0.2

Jean-Luc Béchennec
Mikaël Briday
Pierre Molinaro

11 novembre 2014

Table des matières

Table des matières 2

Liste des tableaux 15

Table des figures 16

I Utilisation 17

1 Getting and installing GALGAS 18

2 Using GALGAS 19
- 2.1 Command Line Options . . . 19
- 2.2 Creating a New Project . . . 19

3 Lexical Elements 20

4 Traduction dirigée par la syntaxe 21
- 4.1 Le programme d'exemple . . . 21
- 4.2 Activer la traduction dirigée par la syntaxe . . . 22
- 4.3 Obtenir la chaîne traduite . . . 22
- 4.4 Modifier l'instruction d'appel de terminal . . . 23
- 4.5 Insérer du texte : instruction `send` . . . 24
- 4.6 Modifier l'instruction d'appel de non-terminal . . . 25

II Le système de types 27

5 Présentation du système de types 28
- 5.1 Opérations définies pour tous les types . . . 28
  - 5.1.1 L'opérateur `==` . . . 28
  - 5.1.2 L'opérateur `!=` . . . 28
  - 5.1.3 Le getter `description` . . . 28
  - 5.1.4 Le getter `dynamicType` . . . 28
  - 5.1.5 Le getter `object` . . . 29

6 Types de base 30

7 Le type `@binaryset` 31
- 7.1 Constructeurs . . . 31
  - 7.1.1 Constructeur `binarySetWithBit` . . . 31
  - 7.1.2 Constructeur `binarySetWithEqualComparison` . . . 31
  - 7.1.3 Constructeur `binarySetWithEqualToConstant` . . . 32

7.1.4 Constructeur binarySetWithGreaterOrEqualComparison 32
7.1.5 Constructeur binarySetWithGreaterOrEqualToConstant 33
7.1.6 Constructeur binarySetWithLowerOrEqualComparison 33
7.1.7 Constructeur binarySetWithLowerOrEqualToConstant 33
7.1.8 Constructeur binarySetWithNotEqualComparison 34
7.1.9 Constructeur binarySetWithNotEqualToConstant 34
7.1.10 Constructeur binarySetWithPredicateString 34
7.1.11 Constructeur binarySetWithStrictGreaterComparison 35
7.1.12 Constructeur binarySetWithStrictGreaterThanConstant 36
7.1.13 Constructeur binarySetWithStrictLowerComparison 36
7.1.14 Constructeur binarySetWithStrictLowerThanConstant 36
7.1.15 Constructeur emptyBinarySet 37
7.1.16 Constructeur fullBinarySet 37
7.2 Getters 37
7.2.1 Getter accessibleStates 37
7.2.2 Getter binarySetByTranslatingFromIndex 38
7.2.3 Getter compressedValueCount 38
7.2.4 Getter compressedStringValueList 38
7.2.5 Getter containsValue 38
7.2.6 Getter equalTo 39
7.2.7 Getter existOnBitIndex 39
7.2.8 Getter existsOnBitRange 39
7.2.9 Getter existOnBitIndexAndBeyond 40
7.2.10 Getter forAllOnBitIndex 40
7.2.11 Getter forAllOnBitIndexAndBeyond 40
7.2.12 Getter greaterOrEqualTo 40
7.2.13 Getter isEmpty 41
7.2.14 Getter isFull 41
7.2.15 Getter ITE 41
7.2.16 Getter lowerOrEqualTo 41
7.2.17 Getter notEqualTo 42
7.2.18 Getter predicateStringValue 42
7.2.19 Getter strictGreaterThan 42
7.2.20 Getter strictLowerThan 43
7.2.21 Getter stringValueList 43
7.2.22 Getter stringValueListWithNameList 43
7.2.23 Getter swap021 43
7.2.24 Getter swap01 44
7.2.25 Getter swap102 44
7.2.26 Getter swap120 44
7.2.27 Getter swap201 45
7.2.28 Getter swap210 45
7.2.29 Getter transitiveClosure 45
7.2.30 Getter uint64ValueList 46
7.2.31 Getter valueCount 46
7.3 Logical Operators 46
7.4 Comparison Operators 46
7.5 Shift Operators 47

8 Le type @bool **48**
8.1 Getters . . . 48
8.1.1 Getter cString . . . 48
8.1.2 Getter ocString . . . 48
8.1.3 Getter sint . . . 48
8.1.4 Getter sint64 . . . 49
8.1.5 Getter uint . . . 49
8.1.6 Getter uint64 . . . 49
8.2 Logical Operators . . . 49
8.3 Comparison Operators . . . 50

9 Le type @char **51**
9.1 Constructors . . . 52
9.1.1 Constructeur replacementCharacter . . . 52
9.1.2 Constructeur unicodeCharacterWithUnsigned . . . 52
9.2 Getters . . . 53
9.2.1 Getter isalnum . . . 53
9.2.2 Getter isalpha . . . 53
9.2.3 Getter iscntrl . . . 53
9.2.4 Getter isdigit . . . 53
9.2.5 Getter islower . . . 54
9.2.6 Getter isUnicodeCommand . . . 54
9.2.7 Getter isUnicodeLetter . . . 54
9.2.8 Getter isUnicodeMark . . . 54
9.2.9 Getter isUnicodePunctuation . . . 55
9.2.10 Getter isUnicodeSeparator . . . 55
9.2.11 Getter isUnicodeSymbol . . . 55
9.2.12 Getter isupper . . . 55
9.2.13 Getter string . . . 56
9.2.14 Getter uint . . . 56
9.2.15 Getter unicodeName . . . 56
9.2.16 Getter unicodeToLower . . . 56
9.2.17 Getter unicodeToUpper . . . 57
9.3 Comparison Operators . . . 57

10 Le type @data **58**

11 Le type @double **59**
11.1 Constructor . . . 59
11.1.1 Constructeur doubleWithBinaryImage . . . 59
11.1.2 Constructeur pi . . . 59
11.2 Getters . . . 60
11.2.1 Getter binaryImage . . . 60
11.2.2 Getter cos . . . 60
11.2.3 Getter sin . . . 60
11.2.4 Getter sint . . . 60
11.2.5 Getter sint64 . . . 60
11.2.6 Getter string . . . 61
11.2.7 Getter tan . . . 61
11.2.8 Getter uint . . . 61
11.2.9 Getter uint64 . . . 61
11.3 Arithmetic Operators . . . 62
11.4 Comparison Operators . . . 62

12 Le type `@filewrapper` 63
12.1 Constructor . . . 63
12.2 Setter . . . 63
12.2.1 Setter `setCurrentDirectory` . . . 63
12.3 Getters . . . 63
12.3.1 Getter `allTextFilePathes` . . . 63
12.3.2 Getter `allDirectoryPathes` . . . 63
12.3.3 Getter `currentDirectory` . . . 63
12.3.4 Getter `allFilePathesWithExtension` . . . 63
12.3.5 Getter `directoryExistsAtPath` . . . 63
12.3.6 Getter `fileExistsAtPath` . . . 64
12.3.7 Getter `textFileContentsAtPath` . . . 64
12.3.8 Getter `binaryFileContentsAtPath` . . . 64
12.3.9 Getter `absolutePathForPath` . . . 64

13 Le type `@location` 65
13.1 Le mot réservé `here` . . . 65
13.2 Constructor . . . 65
13.2.1 Constructeur `nowhere` . . . 65
13.3 Getters . . . 66
13.3.1 Getter `column` . . . 66
13.3.2 Getter `isNowhere` . . . 66
13.3.3 Getter `line` . . . 66
13.3.4 Getter `locationIndex` . . . 66
13.3.5 Getter `locationString` . . . 67

14 Le type `@object` 68

15 Le type `@sint` 69
15.1 Constructors . . . 69
15.1.1 Constructeur `min` . . . 69
15.1.2 Constructeur `max` . . . 69
15.2 Getters . . . 70
15.2.1 Getter `double` . . . 70
15.2.2 Getter `sint64` . . . 70
15.2.3 Getter `string` . . . 70
15.2.4 Getter `uint` . . . 70
15.2.5 Getter `uint64` . . . 71
15.3 Incrementation and decrementation . . . 71
15.4 Arithmetic Operators . . . 71
15.5 Shift Operators . . . 72
15.6 Logical Operators . . . 72
15.7 Comparison Operators . . . 72

16 Le type `@sint64` 73
16.1 Constructors . . . 73
16.1.1 Constructeur `min` . . . 73
16.1.2 Constructeur `max` . . . 73
16.2 Getters . . . 74
16.2.1 Getter `double` . . . 74
16.2.2 Getter `sint` . . . 74
16.2.3 Getter `string` . . . 74
16.2.4 Getter `uint` . . . 74
16.2.5 Getter `uint64` . . . 75

16.3 Incrementation and decrementation . . . 75
16.4 Arithmetic Operators . . . 75
16.5 Shift Operators . . . 76
16.6 Logical Operators . . . 76
16.7 Comparison Operators . . . 76

17 Le type `@string` **77**
17.1 Getters . . . 77
17.1.1 Getter `containsCharacter` . . . 77
17.1.2 Getter `subString` . . . 77

18 Le type `@stringset` **78**
18.1 Constructors . . . 78
18.1.1 Constructeur `emptySet` . . . 78
18.1.2 Constructeur `setWithString` . . . 78
18.2 Getters . . . 78
18.2.1 Getter `count` . . . 78
18.2.2 Getter `hasKey` . . . 79
18.2.3 Getter `anyString` . . . 79
18.3 Setter . . . 79
18.3.1 Setter `removeKey` . . . 79
18.4 the += Operator . . . 79
18.5 the & Operator . . . 80
18.6 the | Operator . . . 80
18.7 the – Operator . . . 80
18.8 Enumerating `@stringset` objects . . . 80
18.9 Comparison Operators . . . 81

19 Le type `@type` **82**

20 Le type `@uint` **83**
20.1 Constructors . . . 83
20.1.1 Constructeur `errorCount` . . . 83
20.1.2 Constructeur `max` . . . 83
20.1.3 Constructeur `valueWithMask` . . . 84
20.1.4 Constructeur `warningCount` . . . 84
20.2 Getters . . . 84
20.2.1 Getter `double` . . . 84
20.2.2 Getter `hexString` . . . 84
20.2.3 Getter `isInRange` . . . 85
20.2.4 Getter `isUnicodeValueAssigned` . . . 85
20.2.5 Getter `lsbIndex` . . . 85
20.2.6 Getter `significantBitCount` . . . 86
20.2.7 Getter `sint` . . . 86
20.2.8 Getter `sint64` . . . 86
20.2.9 Getter `string` . . . 86
20.2.10 Getter `uint64` . . . 87
20.2.11 Getter `xString` . . . 87
20.3 Incrementation and decrementation . . . 87
20.4 Arithmetic Operators . . . 87
20.5 Shift Operators . . . 88
20.6 Logical Operators . . . 88
20.7 Comparison Operators . . . 89

21 Le type @uint64 **90**
21.1 Constructeurs 90
21.1.1 Constructeur max 90
21.1.2 Constructeur uint64BaseValueWithCompressedBitString 90
21.1.3 Constructeur uint64MaskWithCompressedBitString 91
21.1.4 Constructeur uint64WithBitString 91
21.2 Getters 92
21.2.1 Getter double 92
21.2.2 Getter hexString 92
21.2.3 Getter sint 92
21.2.4 Getter sint64 93
21.2.5 Getter string 93
21.2.6 Getter uint 93
21.2.7 Getter uintSlice 93
21.2.8 Getter xString 94
21.3 Incrementation and decrementation 94
21.4 Arithmetic Operators 94
21.5 Shift Operators 95
21.6 Logical Operators 95
21.7 Comparison Operators 95

22 Le type list **96**
22.1 List Type Declaration 96
22.2 Constructors 96
22.2.1 The emptyList constructor 96
22.2.2 The listWithValue constructor 96
22.3 Adding elements 97
22.3.1 The += operator 97
22.3.2 L'instruction .= 97
22.3.3 The prependValue setter 97
22.3.4 Setter insertAtIndex 97
22.3.5 The concatenation operator 98
22.4 Removing elements 98
22.4.1 Setter popFirst 98
22.4.2 Setter popLast 98
22.5 Methods 98
22.5.1 The first method 98
22.5.2 The last method 99
22.6 Getters 99
22.6.1 Le getter length 99
22.6.2 Le getter range 99
22.6.3 Le getter subListFromIndex 99
22.6.4 Le getter subListWithRange 99
22.7 Enumerating a list with a foreach instruction 99
22.7.1 Enumeration using the implicitly declared constants 100
22.7.2 Enumeration using the explicitly declared constants 100
22.7.3 Enumeration in the reverse order 100
22.8 Direct Access of an element attribute 100
22.8.1 Read Access 101
22.8.2 Write Access 101
22.8.3 Example of read and write accesses 101

23 Le type `sortedlist` **103**
23.1 Déclaration . . . 103
23.2 Constructeurs . . . 104
23.2.1 Constructeur `emptySortedList` . . . 104
23.2.2 Constructeur `sortedListWithValue` . . . 104
23.3 Opérateurs . . . 104
23.3.1 L'opérateur `+=` . . . 104
23.3.2 L'opérateur `.=` . . . 104
23.3.3 L'opérateur `.` . . . 105
23.4 Getter `length` . . . 105
23.5 Setters . . . 105
23.5.1 Setter `popGreatest` . . . 105
23.5.2 Setter `popSmallest` . . . 105
23.6 Méthodes . . . 106
23.6.1 La méthode `greatest` . . . 106
23.6.2 La méthode `smallest` . . . 106
23.7 Énumération avec l'instruction `for` . . . 106

24 Le type `array` **108**
24.1 Déclaration d'un type tableau . . . 108
24.2 Constructeur d'un type tableau . . . 108
24.3 Accès à un élément . . . 109
24.3.1 Le getter `valueAtIndex` . . . 109
24.3.2 Setter `setValueAtIndex` . . . 109
24.3.3 Setter `forceValueAtIndex` . . . 109
24.4 Validité d'un élément . . . 109
24.4.1 Le getter `isValueValidAtIndex` . . . 110
24.4.2 Setter `invalidateValueAtIndex` . . . 110
24.5 Contrôle des tailles des axes . . . 110
24.5.1 Le getter `axisCount` . . . 110
24.5.2 Le getter `sizeForAxis` . . . 110
24.5.3 Le getter `rangeForAxis` . . . 110
24.5.4 Setter `setSizeForAxis` . . . 111
24.5.5 Setter `setSize` . . . 111
24.6 Comparaison . . . 111

25 Le type `class` **112**
25.1 Déclaration d'une classe . . . 112
25.2 Le constructeur `new` . . . 112
25.3 Lecture d'un attribut . . . 113
25.4 Écriture d'un attribut . . . 113
25.5 Conversions entre objets de classes différentes . . . 114
25.5.1 Affectation polymorphique . . . 114

26 Le type `enum` **116**
26.1 Déclaration . . . 116
26.2 Instanciation . . . 116
26.3 Comparaison . . . 116
26.4 Tester une valeur . . . 117
26.5 L'instruction `switch` . . . 117
26.6 Valeurs associées . . . 117

27 Le type graph **120**
27.1 Constructeur emptyGraph . . . 120
27.2 Construire un graphe . . . 121
27.2.1 Setter d'insertion . . . 121
27.2.2 Entrer un arc : setter addEdge . . . 121
27.2.3 setter noteNode . . . 122
27.3 Enlever des arcs . . . 122
27.3.1 Setter removeEdgesToNode . . . 122
27.3.2 Setter removeEdgesToDominators . . . 122
27.4 Getters . . . 122
27.4.1 Getter edges . . . 122
27.4.2 Getter graphviz . . . 122
27.4.3 Getter keyList . . . 123
27.4.4 Getter lkeyList . . . 123
27.4.5 Getter reversedGraph . . . 123
27.4.6 Getter subgraphFromNodes . . . 123
27.4.7 Getter accessibleNodesFromNodes . . . 123
27.4.8 Getter undefinedNodeCount . . . 123
27.4.9 Getter undefinedNodeKeyList . . . 123
27.4.10 Getter undefinedNodeReferenceList . . . 124
27.5 Méthodes . . . 124
27.5.1 Méthode depthFirstTopologicalSort . . . 124
27.5.2 Méthode nodesWithNoPredecessor . . . 124
27.5.3 Méthode nodesWithNoSucccessor . . . 125
27.5.4 Méthode topologicalSort . . . 125

28 Le type map **126**
28.1 Déclaration . . . 126
28.2 Constructeurs . . . 126
28.2.1 Constructeur emptyMap . . . 126
28.2.2 Constructeur mapWithMapToOverride . . . 126
28.3 Setters d'insertion . . . 127
28.4 Méthodes de recherche . . . 127
28.5 Setters de retrait . . . 128
28.6 Getters . . . 128
28.6.1 Getter count . . . 128
28.6.2 Getter hasKey . . . 128
28.6.3 Getter keyList . . . 128
28.6.4 Getter keySet . . . 129
28.6.5 Getter locationForKey . . . 129
28.6.6 Getter overriddenMap . . . 129
28.7 Énumération . . . 129

29 Le type structure **130**

30 Type extern **131**
30.1 Type externe minimum . . . 131
30.1.1 Déclaration en GALGAS . . . 131
30.1.2 Implémentation en C++ . . . 132
30.2 Constructeur . . . 132
30.3 Setter . . . 132
30.4 Méthode . . . 132
30.5 Getter . . . 132
30.6 Méthode de classe . . . 132

31 Types prédéfinis 133
31.1 Types structure prédéfinis . . . 133
31.1.1 Le type @lbool . . . 133
31.1.2 Le type @lchar . . . 133
31.1.3 Le type @ldouble . . . 133
31.1.4 Le type @lsint . . . 134
31.1.5 Le type @lsint64 . . . 134
31.1.6 Le type @lstring . . . 134
31.1.7 Le type @luint . . . 134
31.1.8 Le type @luint64 . . . 134
31.1.9 Le type @range . . . 134

III Sous-programmes 136

32 Sous-programmes 137
32.1 Arguments formels et paramètres effectifs . . . 137
32.1.1 Argument formel en entrée . . . 137
32.1.2 Argument formel en entrée/sortie . . . 137
32.1.3 Argument formel en sortie . . . 138
32.2 Liste d'arguments formels en entrée, en sortie, ou en entrée/sortie . . . 138
32.3 Liste de paramètres effectifs en entrée . . . 138
32.4 Sélecteur . . . 138

33 Fonctions et procédures 140
33.1 Fonction . . . 140
33.1.1 déclaration d'une fonction . . . 140
33.1.2 Fonction interne à un fichier . . . 140
33.1.3 Fonction %once . . . 141
33.2 Procédure . . . 141
33.2.1 Déclaration d'une procédure . . . 141
33.2.2 Procédure interne à un fichier . . . 141

34 Extensions 142
34.1 Category getter . . . 143
34.2 Category method . . . 143
34.3 Category setter . . . 144
34.4 Categories and classes . . . 146

IV Filewrappers et templates 148

35 Filewrappers 149
35.1 Décllararation d'un filewrapper . . . 149

V Instructions et expressions 151

36 Expressions 152
36.1 Opérandes . . . 152
36.1.1 Identificateur . . . 152
36.1.2 self . . . 152
36.1.3 Expression de conversion polymorphique inverse . . . 152
36.1.4 Test du type dynamique d'une expression . . . 153

36.1.5 Parenthèses . . . 153
36.1.6 true et false . . . 153
36.1.7 Constante Chaîne de caractères . . . 153
36.1.8 Constante caractère . . . 153
36.1.9 Constante entière . . . 153
36.1.10 Constante flottante . . . 154
36.1.11 Expression if . . . 154
36.1.12 Appel de fonction . . . 154
36.1.13 Appel de getter . . . 154
36.1.14 Constructeur . . . 154
Suppression des accolades . . . 154
Inférence du type . . . 154
36.1.15 Constructeur par défaut . . . 154
Intérêt du constructeur par défaut . . . 154
Les constructeurs par défaut pour chaque type . . . 155
36.1.16 Valeur d'une option . . . 155
36.2 Opérateurs . . . 157
36.2.1 Priorité des opérateurs . . . 157
36.2.2 Logique . . . 157
36.2.3 Logique, évaluation en court-circuit . . . 157
36.2.4 Complémentation bit-à-bit . . . 157
36.2.5 Comparaison . . . 157
36.2.6 Décalage . . . 157
36.2.7 Arithmétique . . . 158
36.2.8 Accès à un champ d'une structure . . . 158

37 Instructions sémantiques 159
37.1 Rôle du point-virgule « ; » . . . 159
37.2 Instruction de déclaration de variable . . . 159
37.2.1 Déclaration « var @type variable » . . . 159
37.2.2 Déclaration « var @type variable = expression » . . . 160
37.3 Instruction de déclaration de constante . . . 160
37.4 L'instruction d'affectation . . . 161
37.5 L'instruction cast . . . 162
37.6 L'instruction d'ajout += . . . 162
37.6.1 Instruction d'ajout et le type @string . . . 163
37.6.2 Instruction d'ajout et le type @stringset . . . 163
37.6.3 Instruction d'ajout et les listes . . . 163
37.6.4 Instruction d'ajout et les listes triées . . . 164
37.6.5 Instruction d'ajout et les tables . . . 164
37.7 Décrémentation -- . . . 164
37.8 L'instruction drop . . . 165
37.9 L'instruction error . . . 165
37.10 L'appel de procédure . . . 166
37.11 L'instruction for . . . 166
37.11.1 Les quatre formes d'une énumération . . . 167
37.11.2 Types énumérables et ordre d'énumération . . . 167
37.11.3 Énumération « () in expression » . . . 167
37.11.4 Énumération « () prefixe in expression » . . . 168
37.11.5 Énumération « cst in expression » . . . 169
Type explicite . . . 170
37.11.6 Énumération « (cst1 cst2 ...) in expression » . . . 171
Type explicite . . . 171

Joker . . . 172
Points de suspension . . . 172
37.11.7 Organigramme d’exécution . . . 172
37.11.8 Champs optionnels . . . 172
37.11.9 Modification de la collection . . . 174
37.12 Instruction d’incrémentation . . . 174
37.13 L’instruction `if` . . . 174
37.14 L’instruction `grammar` . . . 176
37.15 L’instruction `log` . . . 177
37.16 L’instruction `loop` . . . 177
37.17 L’instruction d’appel de procédure . . . 178
37.18 L’instruction d’appel de méthode . . . 179
37.19 L’instruction d’appel de procédure de classe . . . 179
37.20 L’instruction d’appel de *setter* . . . 180
37.21 L’instruction `switch` . . . 180
37.22 L’instruction `warning` . . . 182
37.23 L’instruction `with` . . . 182
37.23.1 Accès en lecture tolérant l’échec de la recherche . . . 183
37.23.2 Accès en lecture, signalement d’erreur si échec de la recherche . . . 184
37.23.3 Accès en lecture/écriture tolérant l’échec de la recherche . . . 185
37.23.4 Accès en lecture/écriture, signalement d’erreur si échec de la recherche . . . 186

38 Instructions syntaxiques **187**
38.1 Instruction d’appel de terminal . . . 187
38.2 Instruction d’appel de non terminal . . . 188
38.3 Instruction `select` . . . 188
38.4 Instruction `repeat` . . . 188
38.5 Instruction `parse` . . . 188
38.6 Instruction `send` . . . 188

VI Déclarations **189**

39 Le composant `lexique` **190**
39.1 Définition d’un composant `lexique` . . . 190
39.2 Comment opère un analyseur lexical . . . 191
39.3 Ambiguïtés lexicales . . . 191
39.4 Unexemple . . . 191
39.5 Déclarations lexicales . . . 192
39.5.1 Déclaration d’un symbole terminal . . . 192
39.5.2 Déclaration d’une liste de symboles terminaux . . . 193
39.5.3 Déclaration d’un attribut terminal . . . 193
39.5.4 Déclaration d’un message d’erreur lexicale . . . 194
39.6 Règles lexicales . . . 194
39.6.1 Règle s’appuyant sur une liste . . . 194
39.6.2 Simple règle . . . 194
39.7 Instructions lexicales . . . 194
39.7.1 Instruction lexicale `select` . . . 194
39.7.2 Instruction lexicale `repeat` . . . 195
39.7.3 Appel d’une action lexicale . . . 195
39.7.4 Appel d’une fonction lexicale . . . 196
39.7.5 Instruction lexicale `error` . . . 196
39.7.6 Instruction lexicale `send` . . . 196
39.7.7 Instruction lexicale `drop` . . . 196

39.7.8 Instruction lexicale tag 197
39.7.9 Instruction lexicale rewind 197
39.8 Routines lexicales prédéfinies 197
39.8.1 Routine codePointToUnicode 197
39.8.2 Routine convertDecimalStringIntoSInt 197
39.8.3 Routine convertDecimalStringIntoSInt64 197
39.8.4 Routine convertDecimalStringIntoUInt 197
39.8.5 Routine convertDecimalStringIntoUInt64 198
39.8.6 Routine convertHTMLSequenceToUnicodeCharacter 198
39.8.7 Routine convertHexStringIntoSInt 198
39.8.8 Routine convertHexStringIntoSInt64 198
39.8.9 Routine convertHexStringIntoUInt 198
39.8.10 Routine convertHexStringIntoUInt64 198
39.8.11 Routine convertStringToDouble 198
39.8.12 Routine convertUInt64ToSInt64 199
39.8.13 Routine convertUIntToSInt 199
39.8.14 Routine convertUnsignedNumberToUnicodeChar 199
39.8.15 Routine enterBinDigitIntoUInt 199
39.8.16 Routine enterBinDigitIntoUInt64 199
39.8.17 Routine enterCharacterIntoCharacter 199
39.8.18 Routine enterCharacterIntoString 199
39.8.19 Routine enterDigitIntoASCIIcharacter 200
39.8.20 Routine enterDigitIntoUInt 200
39.8.21 Routine enterDigitIntoUInt64 200
39.8.22 Routine enterHexDigitIntoASCIIcharacter 200
39.8.23 Routine enterHexDigitIntoUInt 201
39.8.24 Routine enterHexDigitIntoUInt64 201
39.8.25 Routine enterOctDigitIntoUInt 201
39.8.26 Routine enterOctDigitIntoUInt64 201
39.8.27 Routine multiplyUInt 201
39.8.28 Routine multiplyUInt64 202
39.8.29 Routine negateSInt 202
39.8.30 Routine negateSInt64 202
39.8.31 Routine resetString 202
39.9 Fonctions lexicales prédéfinies 202
39.9.1 Fonction toLower 202
39.9.2 Fonction toUpper 202
39.10 Définir vos propres actions et fonctions lexicales 202
39.10.1 Où ? 203
39.10.2 Correspondance entre les appels d'actions GALGAS et C++ 203
39.11 Exemples d'analyseurs lexicaux 203
39.11.1 Analyser des identificateurs 203
39.11.2 Analyser des identificateurs et des mots-clés 204
39.11.3 Analyser des délimiteurs 204
39.11.4 Analyser des séparateurs 204
39.11.5 Analyser des commentaires 204
39.11.6 Analyser des entiers décimaux non signés 204
39.11.7 Analyser des entiers hexadécimaux non signés 205
39.11.8 Analyser des constantes caractère 205
39.11.9 Analyser des constantes chaîne de caractères 205
39.11.10 Analyser des constantes flottantes 205
39.12 *Back tracking* avec les instructions tag et rewind 206
39.13 Ajouter la coloration lexicale (sur Mac uniquement) 207

39.13.1 Exemple : les styles de l'analyseur lexical GALGAS . . . 207
39.13.2 Appliquer un style aux commentaires . . . 207

40 Écrire un composant gammaire **209**
40.1 GALGAS and Context-Free Grammars . . . 209
40.2 Analyse en plusieurs phases . . . 209

41 Graphic User Interface Component **210**

42 Le composant `option` **211**
42.1 Déclaration d'une option . . . 211
42.2 Option booléenne . . . 211
42.3 Option entière . . . 212
42.4 Option chaîne de caractères . . . 212

43 Règle d'analyse de fichier source **213**

44 Le composant `project` **214**

45 Cocoa Features **215**
45.1 Generated Cocoa Application . . . 215
45.2 Adding Icons to your Cocoa Application . . . 215
45.3 Customizing Syntax Coloring . . . 216
45.4 Indexing your source files . . . 216

VII Index **219**

Index **220**

Liste des tableaux

32.1 Constructions d'appel de sous programme 138
32.2 Argument formel en entrée, paramètre effectif en sortie 138
32.3 Argument formel en entrée/sortie, paramètre effectif en sortie/entrée 139
32.4 Argument formel en sortie, paramètre effectif en entrée 139

36.1 Constructeur par défaut pour chaque type 156
36.2 Informations relatives à une option de la ligne de commande 156
36.3 Priorité des opérateurs 157

37.1 Les quatre formes d'énumération de l'instruction `for` 167
37.2 Types énumérables par l'instruction `for` 167
37.3 Constantes implicitement déclarées par « `() in expression`» 168
37.4 Type de la constante dans «`cst in expression` » 170
37.5 Constantes à déclarer pour « `(cst1 cst2 ...) in expression` » 171

Table des figures

34.1 inheritance graph and categories 147

37.1 Organigramme d’exécution d’une instruction `for` 173
37.2 Organigramme d’exécution d’une instruction `loop` 178

45.1 Example of a syntax coloring property list 216
45.2 Indexing and cross-referencing in GALGAS Cocoa Application 217

Première partie

Utilisation

Chapitre 1

Getting and installing GALGAS

Chapitre 2

Using GALGAS

2.1 Command Line Options

2.2 Creating a New Project

Chapitre 3

Lexical Elements

Chapitre 4

Traduction dirigée par la syntaxe

GALGAS permet de construire un *traducteur dirigée par la syntaxe*. Ce type de traduction permet de transformer le texte source d'une grammaire en un autre texte source, tout en conservant les commentaires. C'est donc bien adapté pour mettre à jour des textes sources suite à un changement de syntaxe.

Mettre en place une traduction dirigée par la syntaxe en GALGAS fait appel aux constructions suivantes :

- activer la traduction dirigée par la syntaxe pour chaque composant `syntax` ;
- activer la traduction dirigée par la syntaxe pour le composant `grammar` ;
- modifier l'instruction `grammar` , de façon à récupérer les informations de traduction ;
- modifier l'instruction d'appel de terminal, de façon à récupérer les informations relatives à l'occurrence du terminal ;
- modifier l'instruction d'appel de non terminal, de façon à récupérer la traduction du non terminal ;
- appeler l'instruction `send` pour insérer du texte dans la chaîne produite.

4.1 Le programme d'exemple

Pour illustrer les différentes possibilités, on prend pour exemple une grammaire qui analyse les expressions arithmétiques, dont les opérandes sont des identificateurs, et dont les deux opérateurs sont l'addition et la multiplication (l'exemple s'étend facilement à d'autres opérateurs). Les parenthèses sont utilisées pour forcer le groupement.

L'analyseur lexical – non décrit – définit les symboles terminaux `$idf$ !@lstring`, `$+$`, `$*$`, `$($` et `$)$`.

L'analyseur syntaxique est le suivant :

```
syntax expSyntax {
  rule <expression> {
    <terme>
    repeat while $+$ ; <terme> ; end
  }
  rule <terme> {
    <facteur>
    repeat while $*$ ; <facteur> ; end
  }
  rule <facteur> {
    $idf$ ?*
  }
```

```
  rule <facteur> {
    $($
    <expression>
    $)$
  }
}
```

La grammaire :

```
grammar expGrammar "LL1" {
  syntax expSyntax
  <expression>
}
```

La classe de la grammaire (ici `LL1`) n'a pas d'importance pour la traduction dirigée par la syntaxe : celle-ci fonctionne pour toutes les classes de grammaire.

Enfin, le lien entre l'extension des fichiers source et l'analyseur est réalisé par le code suivant :

```
case . "expression"
message "an '.expression' source file"
??@lstring inSourceFile {
  grammar expGrammar in inSourceFile
}
```

4.2 Activer la traduction dirigée par la syntaxe

Activer la traduction dirigée par la syntaxe indique à GALGAS d'engendrer le code supplémentaire qui prend en charge la traduction. L'activation doit être indiquée à la fois sur le composant `syntax` et le composant `grammar` en ajoutant la directive `%translate` dans chaque en-tête[1].

Pour le composant `syntax` :

```
syntax expSyntax %translate {
  ...
```

Et pour la grammaire :

```
grammar expGrammar "LL1" %translate {
  ...
```

Quand la traduction est activée, l'analyse d'un fichier construit une chaîne de caractères, et par défaut celle-ci est identique à la chaîne source. Par défaut, la chaîne construite est perdue, la section suivante va montrer comment l'obtenir.

4.3 Obtenir la chaîne traduite

La chaîne traduite est obtenue en modifiant l'instruction `grammar` (section 37.14 page 176). Comme on l'a vu, celle-ci est :

```
grammar expGrammar in inSourceFile
```

1. Dans le cas où les règles syntaxiques sont réparties dans plusieurs composants `syntax`, l'activation doit être indiquée dans tous.

Obtenir la chaine traduite s'exprime en utilisant l'opérateur `:>` :

```
grammar expGrammar in inSourceFile :> ?@string s
```

L'instruction déclare une variable `s` de type `@string` et lui affecte la chaîne traduite [2].

Par défaut, la chaîne traduite est identique à la chaîne source. Obtenir une chaîne différente est contrôlé par trois instructions :

— l'instruction d'appel de terminal, de façon à récupérer les informations relatives à l'occurrence du terminal ;
— l'instruction d'appel de non terminal, de façon à récupérer la traduction du non terminal ;
— l'instruction `send` pour insérer du texte dans la chaîne produite.

4.4 Modifier l'instruction d'appel de terminal

Une instruction d'appel de terminal a l'allure suivante (par exemple pour `$idf$`) :

```
$idf$ ?*
```

Par défaut, cette instruction recopie à l'identique dans la chaîne produite deux informations :

— les séparateurs qui précèdent le terminal ;
— le terminal lui-même.

Prenons un exemple. On suppose que la chaîne source est : `@1@a+@2@b@3@`, les commentaires étant constitués des séquences `@...@`. Cet exemple considère des commentaires, mais il en est de même pour les séparateurs (espaces, retours à la ligne). La séquence des terminaux encontrés lors de l'analyse de cette phrase est :

Instruction	Séparateurs précédent le terminal	Terminal
`$idf$ ?*`	`@1@`	`a`
`$+$`		`+`
`$idf$ ?*`	`@2@`	`b`

Le dernier commentaire (`@3@`), placé après le dernier symbole non terminal, est toujours ajouté à la fin de la chaîne produite.

Pour obtenir les deux informations attachés à chaque terminal [3], on utilise l'opérateur `:>` :

```
$idf$ ?* :> ?@string separateur ?@string token ;
```

Cette écriture a pour effet que le séparateur précédent le terminal et le terminal lui-même ne sont plus transmis dans la chaîne traduite, mais affectés respectivement à `separateur` et à `token`.

On va prendre un exemple pour illustrer cette construction : produite une chaîne dont les identificateurs et les séparateurs qui les précèdent auront disparus. On modifie le composant `syntax` comme suit [4] :

```
syntax expSyntax {
  rule <expression> {
    <terme> ;
    repeat while $+$ ; <terme> ; end
  }
```

2. Il existe des variantes pour exprimer l'obtention de la chaîne traduite, voir la description de l'instruction `grammar` à la section 37.14 page 176.
3. Il existe d'autres variantes de cet opérateur, voir la description de l'*instruction d'appel de terminal* à la section 38.1 page 187.
4. Il existe une expression plus simple de l'instruction `$idf$ ?* :> ?@string s ?@string t ;`, puisque `s` et `t` ne sont pas utilisés : c'est `$idf$ ?* :> ?* ?* ;`, décrite à la section 38.1 page 187.

```
  rule <terme> {
    <facteur> ;
    repeat while $*$ ; <facteur> ; end
  }
  rule <facteur> {
    $idf$ ?* :> ?@string s ?@string t ;
  }
  rule <facteur> {
    $($ ;
    <expression> ;
    $)$ ;
  }
}
```

Si la chaîne source est `@1@a+@2@b@3@`, alors la chaîne produite est `+@3@`.

Cette première instruction permet donc de ne pas transmettre les informations attachées un terminal. L'instruction `send`, décrite à la section suivante, va montrer comment insérer du texte dans la chaîne produite.

4.5 Insérer du texte : instruction send

L'instruction `send` a la syntaxe suivante [5] :

```
send exp
```

exp est une expression de type `@string`. Son comportement est simple : la valeur de l'expression chaîne de caractères est simplement transmise à la chaîne produite.

Par exemple, supposons que l'on veuille transformer les parenthèses en accolades ; on écrit le composant `syntax` comme suit [6] :

```
syntax expSyntax {
  rule <expression> {
    <terme>
    repeat while $+$ <terme> end
  }
  rule <terme> {
    <facteur>
    repeat while $*$ <facteur> end
  }
  rule <facteur> {
    $idf$ ?*
  }
  rule <facteur> {
    $($ :> ?@string s ?@string t ; send s . "{"
    <expression>
    $)$ :> ?@string s ?@string t ; send s . "}"
  }
}
```

5. L'instruction `send` est décrite à la section 38.6 page 188.
6. Là encore, il existe une forme plus concise de l'instruction `$($ :> ?@string s ?@string t ;`, puisque `t` est inutilisé : c'est `$($ :> ?@string s ?* ;`, décrite à la section 38.1 page 187.

Mentionner `s` dans l'instruction `send` permet de transmettre les séparateurs qui précèdent les parenthèses. Ainsi à partir de la chaîne source `(@1@a+@2@b)@3@`, on obtient `{@1@a+@2@b}@3@`.

L'instruction `send` permet de reconstituer le comportement par défaut de l'instruction d'appel de terminal : par exemple, `$($ :> ?@string s ?@string t ; send s . t ;` a le même effet que `$($ ;`.

Attention, l'instruction `send` est une instruction syntaxique. Cela signifie que le code suivant est incorrect :

```
if condition then
  send A # Erreur
else
  send B # Erreur
end
```

L'analyse des instructions `send A ;` et `send B ;` déclenche une erreur ; en effet, les branches d'une instruction `if` ne peuvent contenir que des instructions sémantiques. Les instructions `send` ne peuvent figurer que directement dans des règles de production, soient dans les branches des instructions `select`, `repeat` ou `parse`. Pour contourner cette interdiction, écrire :

```
@string s
if condition then
  s = A
else
  s = B
end
send s
```

4.6 Modifier l'instruction d'appel de non-terminal

L'instruction d'appel de non terminal capture la chaîne obtenue par la dérivation de ce non terminal :

```
<expression>
```

Par défaut, cette chaîne est ajoutée à la chaîne produite.

Là encore, l'opérateur `:>` permet d'effectuer une interception. On écrit :

```
<expression> :> ?@string e
```

La chaîne obtenue par la dérivation du non terminal `<expression>` n'est pas ajoutée à la chaîne produite, mais affectée à la variable `e`. D'une manière analogue à l'instruction d'appel de terminal, l'instruction `send` permet de retrouver le comportement par défaut :

```
<expression> :> ?@string e ; send e
```

On utilise souvent cette construction pour ne pas transmettre la chaîne obtenue par la dérivation d'un non terminal ; par exemple, si on ne veut pas transmettre les expressions entre parenthèses, on modifie la dernière règle `facteur` en [7] :

```
syntax expSyntax {
  ...
  rule <facteur> {
    $($
    <expression> :> ?@string e
    $)$
```

7. Ou encore : `<expression> :> ?*`.

```
  }
}
```

Deuxième partie

Le système de types

Chapitre 5

Présentation du système de types

5.1 Opérations définies pour tous les types

Tout type implémente implicitement :
- l'opérateur `==` ;
- l'opérateur `!=` ;
- le *getter* `description` ;
- le *getter* `dynamicType` ;
- le *getter* `object`.

La plupart des types implémentent le constructeur par défaut **`default`** (voir section 36.1.15 page 154).

5.1.1 L'opérateur ==

```
func == ?@T ?@T -> @bool
```

Cet opérateur permet de tester l'identité entre de deux objets de même type.

5.1.2 L'opérateur !=

```
func != ?@T ?@T -> @bool
```

Cet opérateur permet de tester la non identité entre de deux objets de même type. Il renvoie le complément logique du résultat de l'application de l'opérateur `==`.

5.1.3 Le getter `description`

```
getter @T description -> @string
```

Le *getter* `description` retourne une description textuelle du receveur, la même que celle affichée par l'instruction **`log`** (section 37.15 page 177).

5.1.4 Le getter `dynamicType`

```
getter @T dynamicType -> @type
```

Le *getter* `dynamicType` retourne un objet de type `@type`, dont la valeur représente le type dynamique du receveur (voir aussi la définition du type `@type` (page 82)).

Pour tous les types sauf les classes, leurs instances sont du même type que le type statique :

```
@uint n = 2
@type t = [n dynamicType]
log t # Affiche @uint
```

Pour les instances de classes, le jeu des affectations polymorphiques peut entraîner que le type dynamique soit une classe héritière du type statique.

Par exemple, en déclarant :

```
class @A { }
class @B : @A { }
```

Et avec la séquence d'instructions suivante :

```
@B b = .new
@type t = [b dynamicType]
log t # Affiche @B, type statique de b : @B
@A a = b # Affectation polymorphique
t = [a dynamicType]
log t # Affiche @B, type statique de a : @A
```

5.1.5 Le getter `object`

```
getter @T object -> @object
```

Le *getter* `object` retourne un objet de type `@object`. Une variable de type `@object` (page 68) peut encapsuler tout type de valeur.

Chapitre 6

Types de base

GALGAS predefines several types. This chapter presents all their features, including their constructors, getters, setters, methods, ...

Les types prédéfinis sont :

— `@binaryset` (page 31), binary set objects (implemented with Binary Decision Diagrams) ;
— `@bool` (page 48), boolean objects ;
— `@char` (page 51), Unicode characters ;
— `@double` (page 59), floating point numbers ;
— `@filewrapper` (page 63), dont les objets permettent d'explorer les *filewrappers* ;
— `@location` (page 65), whose value points out a location in a source file ;
— `@object` (page 68), dont une instance peut encapsuler toute valeur ;
— `@sint` (page 69), the 32-bit signed integers ;
— `@sint64` (page 73), the 64-bit signed integers ;
— `@string` (page 77), the Unicode string objects ;
— `@stringset` (page 78), set of `string` objects ;
— `@type` (page 82), dont une instance représente un type ;
— `@uint` (page 83), the 32-bit unsigned integers ;
— `@uint64` (page 90), the 64-bit unsigned integers.

Chapitre 7

Le type `@binaryset`

Le type `@binaryset` encode des ensembles, des relations binaires, des expressions booléennes. Il est implémenté par des BDD (Binary Decision Diagrams).

7.1 Constructeurs

7.1.1 Constructeur `binarySetWithBit`

Retourne un `@binaryset` dont le bit `inBitIndex` est égal à 1.

```
constructor @binaryset binarySetWithBit
 ?@uint inBitIndex
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Exemple :

```
@binaryset s = .binarySetWithBit {!2}
log s # Affiche <@binaryset: 1XX>
```

7.1.2 Constructeur `binarySetWithEqualComparison`

Retourne un `@binaryset` qui encode la relation d'égalité entre deux variables.

```
constructor @binaryset binarySetWithEqualComparison
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint inRightFirstIndex
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : ce constructeur retourne un binary set qui encode la relation $a == b$, où a est encodé à partir du bit d'indice *inLeftFirstIndex* jusqu'au bit d'indice *inLeftFirstIndex* + *inBitCount* - *1*, et b est encodé à partir

du bit d'indice bit *inRightFirstIndex* jusqu'au bit d'indice *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

```
@binaryset s = .binarySetWithEqualComparison {!0 !2 !3}
log s # Affiche <@binaryset: 00x00, 01X01, 10X10, 11X11>
```

7.1.3 Constructeur `binarySetWithEqualToConstant`

Retourne un `@binaryset` object that encodes a equality relation between a variable and a constant.

```
constructor @binaryset binarySetWithEqualToConstant
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint64 inConstant
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : ce constructeur retourne un objet qui encode la relation the *a* == *cst*, où *a* est encodé à partir du bit d'indice *inBitIndex* jusqu'au bit d'indice *inBitIndex* + *inBitCount* - *1*, et *cst* est défini par l'argument *inConstant*.

Exemple :

```
@binaryset s = .binarySetWithEqualToConstant {!0 !6 !23L}
log s # Affiche <@binaryset: 10111>
```

7.1.4 Constructeur `binarySetWithGreaterOrEqualComparison`

Retourne un `@binaryset` object qui encode la relation *supérieur ou égal* entre deux variables.

```
constructor @binaryset binarySetWithGreaterOrEqualComparison
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint inRightFirstIndex
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : ce constructeur retourne un binary set qui encode la relation *a* >= *b*, où *a* est encodé à partir du bit d'indice *inLeftFirstIndex* jusqu'au bit d'indice *inLeftFirstIndex* + *inBitCount* - *1*, et *b* est encodé à partir du bit d'indice bit *inRightFirstIndex* jusqu'au bit d'indice *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

```
@binaryset s = .binarySetWithGreaterOrEqualComparison {!0 !2 !3}
log s # Affiche <@binaryset: 00XXX, 01X01, 01X1X, 10X1X, 11X11>
```

7.1.5 Constructeur `binarySetWithGreaterOrEqualToConstant`

Retourne un `@binaryset` object that encodes a greater or equal relation between a variable and a constant.

```
constructor @binaryset binarySetWithGreaterOrEqualToConstant
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint64 inConstant
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : the constructeur Retourne un binary set that encodes the *a* >= *cst* relation, where *a* est encodé à partir du bit d'indice *inBitIndex* jusqu'au bit d'indice *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by the *inConstant* argument.

7.1.6 Constructeur `binarySetWithLowerOrEqualComparison`

Retourne un `@binaryset` object that encodes a lower or equal relation between two variables.

```
constructor @binaryset binarySetWithLowerOrEqualComparison
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint inRightFirstIndex
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : the constructeur Retourne un binary set that encodes the *a* <= *b* relation, where *a* est encodé à partir du bit d'indice *inLeftFirstIndex* jusqu'au bit d'indice *inLeftFirstIndex* + *inBitCount* - *1*, and *b* est encodé à partir du bit d'indice *inRightFirstIndex* to *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

```
@binaryset s = .binarySetWithLowerOrEqualComparison !0 !2 !3] ;
log s; # Affiche <@binaryset: 00X00, 01X0X, 10X0X, 10X10, 11XXX>
```

7.1.7 Constructeur `binarySetWithLowerOrEqualToConstant`

Retourne un `@binaryset` object that encodes a lower or equal relation between a variable and a constant.

```
constructor @binaryset binarySetWithLowerOrEqualToConstant
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint64 inConstant
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : the constructeur Retourne un binary set that encodes the *a* <= *cst* relation, where *a* est encodé à partir du bit d'indice *inBitIndex* jusqu'au bit d'indice *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by

the *inConstant* argument.

7.1.8 Constructeur `binarySetWithNotEqualComparison`

Retourne un `@binaryset` object that encodes an inequality relation between two variables.

```
constructor @binaryset binarySetWithNotEqualComparison
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint inRightFirstIndex
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : the constructeur Retourne un binary set that encodes the *a != b* relation, where *a* est encodé à partir du bit d'indice *inLeftFirstIndex* jusqu'au bit d'indice *inLeftFirstIndex + inBitCount - 1*, and *b* est encodé à partir du bit d'indice *inRightFirstIndex* to *inRightFirstIndex + inBitCount - 1*.

Exemple :

```
@binaryset s = .binarySetWithNotEqualComparison !0 !2 !3] ;
log s; # Affiche <@binaryset: 00X01, 00X1X, 01X00, 01X1X, 10X0X, 10X11, 11X0X, 11X10>
```

7.1.9 Constructeur `binarySetWithNotEqualToConstant`

Retourne un `@binaryset` object that encodes an inequality relation between a variable and a constant.

```
constructor @binaryset binarySetWithNotEqualToConstant
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint64 inConstant
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : the constructeur Retourne un binary set that encodes the *a != cst* relation, where *a* est encodé à partir du bit d'indice *inBitIndex* jusqu'au bit d'indice *inBitIndex + inBitCount - 1*, and *cst* is defined by the *inConstant* argument.

7.1.10 Constructeur `binarySetWithPredicateString`

Returns the `@binaryset` object described by the *inPredicateString* argument.

```
constructor @binaryset binarySetWithPredicateString
 ?@string inPredicateString
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : the *inBitString* argument string encodes a predicate string, such as those returned by @binaryset predicateStringValue getter (page 42).

The *inBitString* argument string characters should have one of the five following values :
— `'0'` : a bit set to zero ;
— `'1'` : a bit set to one ;
— `'X'` : a don't care bit ;
— `' '` : a separator (non significant character) ;
— `'|'` : the boolean *or* operation (in infix notation).

Exemple :

```
An empty predicate string (or a string containing only spaces) provides an empty binary set:
@binaryset s = .binarySetWithPredicateString !" "] ;
@bool b = = .s isEmptySet]; # b is true
```

```
A predicate string containing only 'X' characters (at least one) provides an full binary set:
@binaryset s = .binarySetWithPredicateString !" X X"] ; # Spaces are non significant
@bool b = [s isFullSet]; # b is true
```

```
A predicate string can encode a binary value (MSB first):
@binaryset s [binarySetWithPredicateString !"1100"] ; # 12 in decimal
log s; # Affiche <@binaryset: 1100>
```

```
You can use the boolean '|' operator for providing an or'ed values:
@binaryset s [binarySetWithPredicateString !" 1100 | 1101"] ;
log s; # Affiche <@binaryset: 110X>
```

```
You can use you can use don't care bits and '|' operator together:
@binaryset s [binarySetWithPredicateString !"11X00X0 | 111XXX"] ;
log s; # Affiche <@binaryset: 1100X0, 111XXX>
```

7.1.11 Constructeur `binarySetWithStrictGreaterComparison`

Retourne un `@binaryset` object that encodes a strict greater than relation between two variables.

```
constructor @binaryset binarySetWithStrictGreaterComparison
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint inRightFirstIndex
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : the constructeur Retourne un binary set that encodes the $a > b$ relation, where a est encodé à partir du bit d'indice *inLeftFirstIndex* jusqu'au bit d'indice *inLeftFirstIndex + inBitCount - 1*, and b est encodé à partir du bit d'indice *inRightFirstIndex* to *inRightFirstIndex + inBitCount - 1*.

Exemple :

```
@binaryset s [binarySetWithStrictGreaterComparison !0 !2 !3] ;
log s; # Affiche <@binaryset: 00X01, 00X1X, 01X1X, 10X11>
```

7.1.12 Constructeur `binarySetWithStrictGreaterThanConstant`

Retourne un `@binaryset` object that encodes a strict greater than relation between a variable and a constant.

```
constructor @binaryset binarySetWithStrictGreaterThanConstant
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint64 inConstant
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : the constructeur Retourne un binary set that encodes the *a* > *cst* relation, where *a* est encodé à partir du bit d'indice *inBitIndex* jusqu'au bit d'indice *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by the *inConstant* argument.

7.1.13 Constructeur `binarySetWithStrictLowerComparison`

Retourne un `@binaryset` object that encodes a strict lower than relation between two variables.

```
constructor @binaryset binarySetWithStrictLowerComparison
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint inRightFirstIndex
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : the constructeur Retourne un binary set that encodes the *a* < *b* relation, where *a* est encodé à partir du bit d'indice *inLeftFirstIndex* jusqu'au bit d'indice *inLeftFirstIndex* + *inBitCount* - *1*, and *b* est encodé à partir du bit d'indice *inRightFirstIndex* to *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

```
@binaryset s [binarySetWithStrictLowerComparison !0 !2 !3] ;
log s; # Affiche <@binaryset: 01X00, 10X0X, 11X0X, 11X10>
```

7.1.14 Constructeur `binarySetWithStrictLowerThanConstant`

Retourne un `@binaryset` object that encodes a strict lower than relation between a variable and a constant.

```
constructor @binaryset binarySetWithStrictLowerThanConstant
 ?@uint inLeftFirstIndex
 ?@uint inBitCount
 ?@uint64 inConstant
 -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion : the constructeur Retourne un binary set that encodes the *a* < *cst* relation, where *a* est encodé à partir du bit d'indice *inBitIndex* jusqu'au bit d'indice *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by the

inConstant argument.

7.1.15 Constructeur `emptyBinarySet`

Retourne un empty `@binaryset` object.

```
constructor @binaryset emptyBinarySet -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

7.1.16 Constructeur `fullBinarySet`

Returns a full `@binaryset` object.

```
constructor @binaryset fullBinarySet -> @binaryset
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

7.2 Getters

7.2.1 Getter `accessibleStates`

Returns the set of accessible states from an initial state set.

```
getter @binaryset accessibleStates -> @binaryset ;
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion: computes the set of accessible states from the *inInitialStateSet* state set using the accessibility relation encoded by the receiver.

Exemple :

```
@binaryset gr [binarySetWithPredicateString !"0001 0000"] ; # Edge 0 -> 1
gr = gr | [@binaryset binarySetWithPredicateString !"0010 0001"] ; # Edge 1 -> 2
gr = gr | [@binaryset binarySetWithPredicateString !"0011 0010"] ; # Edge 2 -> 3
gr = gr | [@binaryset binarySetWithPredicateString !"0100 0011"] ; # Edge 3 -> 4
gr = gr | [@binaryset binarySetWithPredicateString !"0101 0100"] ; # Edge 4 -> 5
@binaryset initialState [binarySetWithPredicateString !"0000"] ; # 0 is the initial state
@binaryset accessibleStates = [gr accessibleStates !initialState !4] ;
message " Accessible:" ;
@uint64list valueList = [accessibleStates uint64ValueList !4] ;
foreach valueList do
  message " " . [mValue string] ;
end foreach ;
message "\n" ;
```

This program Affiche : `Accessible: 0 1 2 3 4 5`.

7.2.2 Getter binarySetByTranslatingFromIndex

Returns a `@binaryset` value computed by translating the receiver's value by *inTranslation* bits from index *inFirstIndex*.

```
getter @binaryset binarySetByTranslatingFromIndex
 ?@uint inFirstIndex
 ?@uint inTranslation
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.3 and later.

7.2.3 Getter compressedValueCount

Returns in an `@uint64` value the number of different compressed string values encoded by receiver's value.

```
getter @binaryset compressedValueCount -> @@uint64 ;
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

7.2.4 Getter compressedStringValueList

Returns the list of compressed string values corresponding to receiver's value, considering it uses *inBitCount* bits.

```
getter @binaryset compressedStringValueList
 ?@uint inBitCount
 -> @stringlist ;
```

Availability: available in GALGAS 1.6.0 and later.

7.2.5 Getter containsValue

Retourne un `@bool` value indicating whether the receiver'value contains a given value.

```
getter @binaryset containsValue
 ?@uint inFirstBit
 ?@uint inBitCount
 -> @bool ;
```

Availability: available in GALGAS 1.6.4 and later.

Discussion: returns `true` if the receiver's contains a value, and `false` otherwise ; this value is computed from the *inBitCount* first bits of *inValue* value, shifted left by *inFirstBit*.

Exemple :

```
@binaryset s [binarySetWithPredicateString !"0 00XX X111\textbar 1 1111 1111"] ;
log s ; \# Affiche <@binaryset: 000XXX111, 111111111>
@bool b = [s containsValue !127L !0 !7] ;
log b ; \# Affiche <@bool:true>
b = [s containsValue !31L !1 !7] ;
log b ; \# Affiche <@bool:true>
b = [s containsValue !63L !1 !8] ;
log b ; \# Affiche <@bool:false>
b = [s containsValue !7L !0 !9] ;
log b ; \# Affiche <@bool:true>
b = [s containsValue !7L !0 !10] ;
log b ; \# Affiche <@bool:true>
b = [s containsValue !32767L !1 !12] ;
log b ; \# Affiche <@bool:true>
```

7.2.6 Getter `equalTo`

Returns the complement of the exclusive or between the receiver's value and the operand's value.

```
getter @binaryset equalTo
 ?@binaryset inOperand
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Note that `[a equalTo !b]` is equivalent to $\sim$ `(a ∧ b)`.

This operation Retourne un `@binaryset` value ; do not confuse with == operator that Retourne un `@bool` value.

7.2.7 Getter `existOnBitIndex`

Returns the binary computed by applying the *exist* operator on the *inBitIndex* bit of the receiver's value.

```
getter @binaryset existOnBitIndex
 ?@uint inBitIndex
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.3 and later.

7.2.8 Getter `existsOnBitRange`

Returns the binary computed by applying the *exist* operator on the receiver's value, from *inFirstBitIndex* bit index until the *inFirstBitIndex* + *inBitCount* - *1* bit index.

```
getter @binaryset existsOnBitRange
 ?@uint inFirstBitIndex
 ?@uint inBitCount
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.3 and later.

Exemple :

```
@binaryset s [binarySetWithPredicateString !"01110010"] ;
log s ; # Affiche <@binaryset: 01110010>
@binaryset ss = [s existsOnBitRange !2 !3] ;
log s ; # Affiche <@binaryset: 011XXX10>
```

7.2.9 Getter `existOnBitIndexAndBeyond`

Returns the binary set computed by applying the *exist* operator on all bits from *inFirstBitIndex* bit index of the receiver's value.

```
getter @binaryset existOnBitIndexAndBeyond
 ?@uint inBitIndex
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.3 and later.

7.2.10 Getter `forAllOnBitIndex`

Returns the binary set computed by applying the *for all* operator on the *inFirstBitIndex* bit index of the receiver's value.

```
getter @binaryset forAllOnBitIndex
 ?@uint inBitIndex
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

7.2.11 Getter `forAllOnBitIndexAndBeyond`

Returns the binary computed by applying the *for all* operator on all bits from *inFirstBitIndex* bit index of the receiver's value.

```
getter @binaryset forAllOnBitIndexAndBeyond
 ?@uint inBitIndex
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

7.2.12 Getter `greaterOrEqualTo`

Returns the complement of the exclusive or between the receiver's value and the operand's value.

```
getter @binaryset greaterOrEqualTo
 ?@binaryset inOperand
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Note that `[a greaterOrEqualTo !b]` is equivalent to `(a | ~b)`.

7.2.13 Getter `isEmpty`

Returns a `@bool` value that indicates whether the receiver's value is the empty set.

```
getter @binaryset isEmpty -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion: returns `true` if receiver's value is the empty set, and `false` otherwise.

7.2.14 Getter `isFull`

Returns a `@bool` value that indicates whether the receiver's value is the full set.

```
getter @binaryset isFull -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion: returns `true` if receiver's value is the full set, and `false` otherwise.

7.2.15 Getter `ITE`

Returns the binary set computed by applying the *ite* operator on the receiver's value, the *inThenOperand* argument, and the *inElseOperand* argument.

```
getter @binaryset ITE
 ?@binaryset inThenOperand
 ?@binaryset inElseOperand
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.3 and later.

Discussion: `ite (x, y, z)` is `(x & y) |(~x & z)`.

7.2.16 Getter `lowerOrEqualTo`

Returns the binary set computed by applying the *lower or equal* operator on the receiver's value and the *inOperand* argument.

```
getter @binaryset lowerOrEqualTo
 ?@binaryset inOperand
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: `[a lowerOrEqualTo !b]` is `((~x) |y)`.

7.2.17 Getter `notEqualTo`

Returns the binary set computed by applying the *not equal* operator on the receiver's value and the *inOperand* argument.

```
getter @binaryset notEqualTo
 ?@binaryset inOperand
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: `[a notEqualTo !b]` is `(x ^ y)`.

7.2.18 Getter `predicateStringValue`

Returns a string representation of the receiver's value.

```
getter @binaryset predicateStringValue -> @string ;
```

Disponibilité : disponible en GALGAS version 1.6.0 et ultérieure.

Discussion: the returned string is compatible with the binarySetWithPredicateString constructor (page 34).

7.2.19 Getter `strictGreaterThan`

Returns the binary set computed by applying the *strict greater* operator on the receiver's value and the *inOperand* argument.

```
getter @binaryset strictGreaterThan
 ?@binaryset inOperand
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: `[a strictGreaterThan !b]` is `(x & ~y)`.

7.2.20 Getter `strictLowerThan`

Returns the binary set computed by applying the *strict lower* operator on the receiver's value and the *inOperand* argument.

```
getter @binaryset strictLowerThan
 ?@binaryset inOperand
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: `[a strictLowerThan !b]` is `(~x & y)`.

7.2.21 Getter `stringValueList`

Returns the list of string values corresponding to receiver's value, considering it uses *inBitCount* bits.

```
getter @binaryset stringValueList
 ?@uint inBitCount
 -> @@stringlist ;
```

Availability: available in GALGAS 1.6.0 and later.

7.2.22 Getter `stringValueListWithNameList`

Returns the list of named values corresponding to receiver's value, considering it uses *inBitCount* bits.

```
getter @binaryset stringValueListWithNameList
 ?@uint inBitCount
 ?@stringlist inNameList
 -> @@stringlist ;
```

Availability: available in GALGAS 1.9.3 and later.

Discussion: first, the receiver is enumerated, considering it uses *inBitCount* bits. Each enumerated value is used as an index of *inNameList*, and the string value at this index is appended at the end of the returned value.

7.2.23 Getter `swap021`

Returns the transposed *(x, z, y)* relation.

```
getter @binaryset swap021
 ?@uint inBitCount1
 ?@uint inBitCount2
 ?@uint inBitCount3
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this getter considers that the receiver encodes an *(x, y, z)* relation, where *x* is defined by bits index *0* to *inBitCount1 - 1*, *y* is defined by bits index *inBitCount1* to *inBitCount1 + inBitCount2 - 1* and *z* is defined by bits index *inBitCount1 + inBitCount2* to *inBitCount1 + inBitCount2 + inBitCount3 - 1*.

7.2.24 Getter `swap01`

Returns the transposed *(y, x)* relation.

```
getter @binaryset swap01
 ?@uint inBitCount1
 ?@uint inBitCount2
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this getter considers that the receiver encodes an *(x, y)* relation, where *x* is defined by bits index *0* to *inBitCount1 - 1*, *y* is defined by bits index *inBitCount1* to *inBitCount1 + inBitCount2 - 1*.

7.2.25 Getter `swap102`

Returns the transposed *(y, x, z)* relation.

```
getter @binaryset swap102
 ?@uint inBitCount1
 ?@uint inBitCount2
 ?@uint inBitCount3
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this getter considers that the receiver encodes an *(x, y, z)* relation, where *x* is defined by bits index *0* to *inBitCount1 - 1*, *y* is defined by bits index *inBitCount1* to *inBitCount1 + inBitCount2 - 1* and *z* is defined by bits index *inBitCount1 + inBitCount2* to *inBitCount1 + inBitCount2 + inBitCount3 - 1*.

7.2.26 Getter `swap120`

Returns the transposed *(y, z, x)* relation.

```
getter @binaryset swap120
 ?@uint inBitCount1
 ?@uint inBitCount2
 ?@uint inBitCount3
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this getter considers that the receiver encodes an *(x, y, z)* relation, where *x* is defined by bits index *0* to *inBitCount1 - 1*, *y* is defined by bits index *inBitCount1* to *inBitCount1 + inBitCount2 - 1* and *z* is defined by bits index *inBitCount1 + inBitCount2* to *inBitCount1 + inBitCount2 + inBitCount3 - 1*.

7.2.27 Getter `swap201`

Returns the transposed *(z, x, y)* relation.

```
getter @binaryset swap201
 ?@uint inBitCount1
 ?@uint inBitCount2
 ?@uint inBitCount3
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this getter considers that the receiver encodes an *(x, y, z)* relation, where *x* is defined by bits index *0* to *inBitCount1 - 1*, *y* is defined by bits index *inBitCount1* to *inBitCount1 + inBitCount2 - 1* and *z* is defined by bits index *inBitCount1 + inBitCount2* to *inBitCount1 + inBitCount2 + inBitCount3 - 1*.

7.2.28 Getter `swap210`

Returns the transposed *(z, y, x)* relation.

```
getter @binaryset swap210
 ?@uint inBitCount1
 ?@uint inBitCount2
 ?@uint inBitCount3
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this getter considers that the receiver encodes an *(x, y, z)* relation, where *x* is defined by bits index *0* to *inBitCount1 - 1*, *y* is defined by bits index *inBitCount1* to *inBitCount1 + inBitCount2 - 1* and *z* is defined by bits index *inBitCount1 + inBitCount2* to *inBitCount1 + inBitCount2 + inBitCount3 - 1*.

7.2.29 Getter `transitiveClosure`

Returns the transitive closure of the relation encoded by the receiver.

```
getter @binaryset transitiveClosure
 ?@uint inBitCount
 -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this getter considers that the receiver encodes an *(x, y)* relation, where *x* is defined by bits index *0* to *inBitCount - 1*, *y* is defined by bits index *inBitCount* to *2 * inBitCount - 1*.

7.2.30 Getter `uint64ValueList`

Returns the list of `@uint64` values corresponding to receiver's value, considering it uses *inBitCount* bits.

```
getter @binaryset uint64ValueList
 ?@uint64 inBitCount
 -> @@uint64list ;
```

Availability: available in GALGAS 1.6.0 and later.

7.2.31 Getter `valueCount`

Returns in an `@uint64` object the number of different values encoded by receiver, considering it uses *inBitCount* bits.

```
getter @binaryset valueCount
 ?@uint inBitCount
 -> @@uint64 ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: no overflow test in performed.

7.3 Logical Operators

The `@binaryset` type supports the three logical operators :

&	Logical And, intersection
|	Logical Or, union
^	Exclusive or

Theses operators require both arguments to be `@binaryset` objects and return an `@binaryset` object.

The `@binaryset` type supports the logical unary operator :

~	Negation, Complementation

This operator Retourne un `@binaryset` object.

7.4 Comparison Operators

The `@binaryset` type supports the two comparison operators :

==	Equality
!=	Non Equality

Theses operators require both arguments to be `@binaryset` objects, and return a `@bool` object. These operations are very fast and are performed in a constant time (integer equality comparison).

Do not confuse with @binaryset equalTo getter (page 39) and @binaryset notEqualTo getter (page 42) that return a `@binaryset` object.

7.5 Shift Operators

The `@binaryset` type supports the two shift operators :

<<	Left Shift
>>	Right Shift

Exemple :

```
@binaryset b [binarySetWithPredicateString !"1010"] ;
log b ; # Affiche: <@binaryset: 1010>
@binaryset bb = b << 3 ;
log bb ; # Affiche: <@binaryset: 1010XXX>
```

Chapitre 8

Le type `@bool`

An `@bool` object has a boolean value. The two keywords **`true`** and **`false`** belong to the `@bool` type, and denotes the true and false values. No constructor is defined for the `@bool` type.

8.1 Getters

8.1.1 Getter `cString`

Returns a string representation of the receiver's value.

```
getter @bool cString -> @string ;
```

Disponibilité : disponible en GALGAS version 1.8.7 et ultérieure.

Discussion: returns the `"true"` string if the receiver's value is true, and the `"false"` string otherwise.

8.1.2 Getter `ocString`

Returns a string representation of the receiver's value.

```
getter @bool ocString -> @string ;
```

Disponibilité : disponible en GALGAS version 1.8.7 et ultérieure.

Discussion: returns the `"YES"` string if the receiver's value is true, and the `"NO"` string otherwise.

8.1.3 Getter `sint`

Returns the receiver's value in an `@sint` (page 69) (32-bit signed integer) object.

```
getter @bool sint -> @sint ;
```

Disponibilité : disponible en GALGAS version 1.9.4 et ultérieure.

Discussion: returns the `1S` `@sint` (page 69) value if the receiver's value is true, and the `0S` `@sint` (page 69) value otherwise.

8.1.4 Getter `sint64`

Returns the receiver's value in an `@sint64` (page 73) (64-bit signed integer) object.

```
getter @bool sint64 -> @sint64 ;
```

Disponibilité : disponible en GALGAS version 1.9.4 et ultérieure.

Discussion: returns the `1LS` `@sint64` (page 73) value if the receiver's value is true, and the `0LS` `@sint64` (page 73) value otherwise.

8.1.5 Getter `uint`

Returns the receiver's value in an `@uint` (page 83) (32-bit unsigned integer) object.

```
getter @bool uint -> @uint ;
```

Disponibilité : disponible en GALGAS version 1.9.4 et ultérieure.

Discussion: returns the `1` `@uint` (page 83) value if the receiver's value is true, and the `0` `@uint` (page 83) value otherwise.

8.1.6 Getter `uint64`

Returns the receiver's value in an `@uint64` (page 90) (64-bit unsigned integer) object.

```
getter @bool uint64 -> @uint64 ;
```

Disponibilité : disponible en GALGAS version 1.9.4 et ultérieure.

Discussion: returns the `1L` `@uint64` (page 90) value if the receiver's value is true, and the `0S` `@uint64` (page 90) value otherwise.

8.2 Logical Operators

The `@bool` type supports the three logical operators :

&	And
|	Or
^	Exclusive or

Theses operators require both arguments to be `@bool` objects and return an `@bool` object.

The `@bool` type supports the logical unary operator :

not	Complementation

This operator returns an `@bool` object.

8.3 Comparison Operators

The `@bool` type supports the six comparison operators :

==	Equality
!=	Non Equality
<	Strict Lower Than
<=	Lower or Equal
>	Strict Greater Than
>=	Greater or Equal

Theses operators require both arguments to be `@bool` objects, and return a `@bool` object.

Chapitre 9

Le type `@char`

An `@char` object value is an Unicode character. You can initialize an `@char` object from a character constant :

```
@char myCharacter := 'A' ;
```

You have several ways for writing a literal character constant. In any case, it should define an assigned Unicode character. A compile-time error is raised if it does not.

A literal character constant is a single character or an escape sequence enclosed by single quotes (').

For an ASCII printable character :

```
@char myCharacter := 'a' ;
```

If you want to get ASCII source text file, any character that does not correspond to an ASCII printable character should be expressed with an escape sequence.

Otherwise, for any printable Unicode character, you can write it directly, without escape sequence, provided your text file encoding supports this character :

```
@char myCharacter := 'æ';
```

The following escape sequences are defined (they begin with a ’’).

Character Constant	Meaning
'\f'	A Form Feed Character
'\n'	A New Line Character
'\r'	A Carriage Return Character
'\v'	A Vertical Tabulation Character
'\\'	A back slash Character
'\0'	A Nul Character
'\''	A Single Quote Character

Character Constant	Meaning
'\uABCD'	An Unicode Character

Where *ABCD* is a four digit hexadecimal number that represents an assigned Unicode point code. For example :

```
@char myCharacter := '\u03A0'; # The 'Σ' character
```

Note : an unassigned point code raises a compile-time error :
```
@char myCharacter := '\uFFFF'; # The \uFFFF point code is not assigned
```

Character Constant	Meaning
'\Uabcdxyzt'	An Unicode Character

Where *abcdxyzt* is a eight digit hexadecimal number that represents an assigned Unicode point code. For example :

```
@char myCharacter := '\U0010170'; # The 'GREEK ACROPHONIC NAXIAN FIVE HUNDRED' cha-
racter
```

Note : an unassigned point code raises a compile-time error :

```
@char myCharacter := '\U0000FFFF'; # Raises a compile-time error: \U0000FFFF is not
assigned.
```

Any point code beyond \U0010FFFF is invalid and not assigned.

9.1 Constructors

9.1.1 Constructeur replacementCharacter

Returns an @char object corresponding to Unicode replacement character ('\uFFFD.

```
constructor @char replacementCharacter -> @char
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

9.1.2 Constructeur unicodeCharacterWithUnsigned

Returns an @char object from an Unicode code point.

```
constructor @char unicodeCharacterWithUnsigned
 ?@uint inValue
 -> @char
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

Discussion : A run-time error is raised if the *inValue* value does not represent an assigned Unicode value. You can check if an @uint value represents an assigned Unicode value with the @uint isUnicodeValueAssigned getter (page 85).

9.2 Getters

9.2.1 Getter `isalnum`

Returns an `@bool` value indicating whether the receiver'value represents an ASCII letter or an ASCII digit.

```
getter @char isalnum -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.7.2 et ultérieure.

Discussion: returns `true` if the receiver'value represents an ASCII letter or an ASCII digit (between `'A'` and `'Z'`, or between `'a'` and `'z'`, or between `'0'` and `'9'`), and `false` otherwise.

9.2.2 Getter `isalpha`

Returns an `@bool` value indicating whether the receiver'value represents an ASCII letter.

```
getter @char isalpha -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.7.2 et ultérieure.

Discussion: returns `true` if the receiver'value represents an ASCII letter (between `'A'` and `'Z'`, or between `'a'` and `'z'`), and `false` otherwise.

9.2.3 Getter `iscntrl`

Returns an `@bool` value indicating whether the receiver'value represents an ASCII control character.

```
getter @char iscntrl -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.7.2 et ultérieure.

Discussion: returns `true` if the receiver'value represents an ASCII control character (strictly before the *SPACE* character), and `false` otherwise.

9.2.4 Getter `isdigit`

Returns an `@bool` value indicating whether the receiver'value represents an ASCII digit.

```
getter @char isdigit -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.7.2 et ultérieure.

Discussion: returns `true` if the receiver'value represents an ASCII digit (between `'0'` and `'9'`), and `false` otherwise.

9.2.5 Getter `islower`

Returns an `@bool` value indicating whether the receiver'value represents an ASCII lowercase ASCII letter.

```
getter @char islower -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.7.2 et ultérieure.

Discussion: returns `true` if the receiver'value represents an ASCII lowercase letter (between `'a'` and `'z'`), and `false` otherwise.

9.2.6 Getter `isUnicodeCommand`

Returns an `@bool` value indicating whether the receiver'value represents an Unicode command.

```
getter @char isUnicodeCommand -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

Discussion: returns `true` if the receiver'value represents an Unicode command, and `false` otherwise.

9.2.7 Getter `isUnicodeLetter`

Returns an `@bool` value indicating whether the receiver'value represents an Unicode letter.

```
getter @char isUnicodeLetter -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

Discussion: returns `true` if the receiver'value represents an Unicode letter, and `false` otherwise.

9.2.8 Getter `isUnicodeMark`

Returns an `@bool` value indicating whether the receiver'value represents an Unicode mark character.

```
getter @char isUnicodeMark -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

Discussion: returns `true` if the receiver'value represents an Unicode mark character, and `false` otherwise.

9.2.9 Getter `isUnicodePunctuation`

Returns an `@bool` value indicating whether the receiver'value represents an Unicode punctuation character.

```
getter @char isUnicodePunctuation -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

Discussion: returns `true` if the receiver'value represents an Unicode punctuation character, and `false` otherwise.

9.2.10 Getter `isUnicodeSeparator`

Returns an `@bool` value indicating whether the receiver'value represents an Unicode separator character.

```
getter @char isUnicodeSeparator -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

Discussion: returns `true` if the receiver'value represents an Unicode separator character, and `false` otherwise.

9.2.11 Getter `isUnicodeSymbol`

Returns an `@bool` value indicating whether the receiver'value represents an Unicode symbol character.

```
getter @char isUnicodeSymbol -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

Discussion: returns `true` if the receiver'value represents an Unicode symbol character, and `false` otherwise.

9.2.12 Getter `isupper`

Returns an `@bool` value indicating whether the receiver'value represents an ASCII uppercase ASCII letter.

```
getter @char isupper -> @bool ;
```

Disponibilité : disponible en GALGAS version 1.7.2 et ultérieure.

Discussion: returns `true` if the receiver'value represents an ASCII uppercase letter (between `'A'` and `'Z'`), and `false` otherwise.

9.2.13 Getter `string`

Returns returns a string representation of the receiver's value.

```
getter @char string -> @string ;
```

Disponibilité : disponible en GALGAS version 1.5.5 et ultérieure.

Discussion: returns a one character `@string` object, containing the receiver's value.

9.2.14 Getter `uint`

Returns an `@uint` object representing the Unicode code point of the receiver's value.

```
getter @char uint -> @uint64 ;
```

Disponibilité : disponible en GALGAS version 1.7.7 et ultérieure.

9.2.15 Getter `unicodeName`

Returns the unicode name of the receiver's value.

```
getter @char unicodeName -> @string ;
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

Discussion: for an decimal string representation of the receiver's value, see the @uint hexString getter (page 84) ; for a decimal string representation of the receiver's value, see the @uint string getter (page 86).

Exemple :

```
['\AE' unicodeName] # returns "LATIN CAPITAL LETTER AE"
```

9.2.16 Getter `unicodeToLower`

Returns the lowercase character corresponding to the receiver's value.

```
getter @char unicodeToLower -> @char ;
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

Discussion: if the receiver's value is an Unicode uppercase character, this getter returns the corresponding lowercase character. Otherwise, it returns the receiver's value.

Exemple :

```
['\AE' unicodeToLower] returns '\ae '
['\ae' unicodeToLower] returns '\ae '
```

9.2.17 Getter `unicodeToUpper`

Returns the uppercase character corresponding to the receiver's value.

```
getter @char unicodeToUpper -> @char ;
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

Discussion: if the receiver's value is an Unicode lowercase character, this getter returns the corresponding uppercase character. Otherwise, it returns the receiver's value.

Exemple :

```
['\AE' unicodeToUpper] returns '\AE'}
['\ae' unicodeToUpper] returns '\AE'}
```

9.3 Comparison Operators

The `@char` type supports the six comparison operators :

$=$	Equality
$!=$	Non Equality
$<$	Strict Lower Than
$<=$	Lower or Equal
$>$	Strict Greater Than
$>=$	Greater or Equal

Theses operators require both arguments to be `@char` objects, and return a `@bool` object. Comparison is done by comparing of the Unicode code point's value.

Chapitre 10

Le type `@data`

Chapitre 11

Le type `@double`

The `@double` object values correspond to the C type `@double` values. You can initialize an `@double` object from a float constant :

```
@double myDouble := 123.456 ;
```

Note that a `@double` constant is characterized by the occurrence of the decimal point (.)

11.1 Constructor

11.1.1 Constructeur `doubleWithBinaryImage`

Returns a double object from the binary image of the argument

```
constructor @double doubleWithBinaryImage
  ?@uint64 inImage
  -> @double
```

Disponibilité : disponible en GALGAS version 2.4.7 et ultérieure.

11.1.2 Constructeur `pi`

Returns an approximation of the π constant value (3.14159265358979323846264338327950288).

```
constructor @double pi -> @double
```

Disponibilité : disponible en GALGAS version 2.1.1 et ultérieure.

11.2 Getters

11.2.1 Getter `binaryImage`

Returns the binary image of the value of receiver's value.

```
getter @double binaryImage -> @uint64 ;
```

Disponibilité : disponible en GALGAS version 2.4.7 et ultérieure.

11.2.2 Getter `cos`

Returns the *cosine* value of receiver's value, expressed in radian.

```
getter @double cos -> @double ;
```

Disponibilité : disponible en GALGAS version 2.1.1 et ultérieure.

11.2.3 Getter `sin`

Returns the *sine* value of receiver's value, expressed in radian.

```
getter @double sin -> @double ;
```

Disponibilité : disponible en GALGAS version 2.1.1 et ultérieure.

11.2.4 Getter `sint`

Returns the receiver's value in an `@sint` (page 69) (32-bit signed integer) object.

```
getter @double sint -> @sint ;
```

Disponibilité : disponible en GALGAS version 1.9.9 et ultérieure.

Discussion: if receiver's value is outside `@sint` bounds, a runtime error is raised.

11.2.5 Getter `sint64`

Returns the receiver's value in an `@sint64` (page 73) (64-bit signed integer) object.

```
getter @double sint64 -> @@sint64 ;
```

Disponibilité : disponible en GALGAS version 1.9.9 et ultérieure.

Discussion: if receiver's value is outside `@sint64` bounds, a runtime error is raised.

11.2.6 Getter `string`

Returns a decimal string representation of the receiver's value.

```
getter @double string -> @string ;
```

Disponibilité : disponible en GALGAS version 1.7.7 et ultérieure.

Discussion: this getter never fails.

11.2.7 Getter `tan`

Returns the *tangent* value of receiver's value, expressed in radian.

```
getter @double tan -> @double ;
```

Disponibilité : disponible en GALGAS version 2.1.1 et ultérieure.

11.2.8 Getter `uint`

Returns the receiver's value in an `@uint` (page 83) (32-bit unsigned integer) object.

```
getter @double uint -> @uint ;
```

Disponibilité : disponible en GALGAS version 1.9.9 et ultérieure.

Discussion: if receiver's value is outside `@uint` bounds, a runtime error is raised.

11.2.9 Getter `uint64`

Returns the receiver's value in an `@uint64` (page 90) (64-bit unsigned integer) object.

```
getter @double uint64 -> @uint64 ;
```

Disponibilité : disponible en GALGAS version 1.9.9 et ultérieure.

Discussion: if receiver's value is outside `@uint64` bounds, a runtime error is raised.

11.3 Arithmetic Operators

The `@double` type supports the five arithmetic diadic operators :

$+$	Addition
$-$	Substraction
$*$	Multiplication
$/$	Division
`mod`	Modulo

Theses operators require both arguments to be `@double` objects.

A run-time error is raised if the operation leads to an overflow.

The `@double` type supports the following arithmetic unary operators :

$+$	No operation
$-$	Negate

This operator returns the receiver's value (an `@double` object).

11.4 Comparison Operators

The `@double` type supports the six comparison operators :

$=$	Equality
$!=$	Non Equality
$<$	Strict Lower Than
$<=$	Lower or Equal
$>$	Strict Greater Than
$>=$	Greater or Equal

Theses operators require both arguments to be `@double` objects, and return a `@bool` object.

Chapitre 12

Le type `@filewrapper`

Le type `@filewrapper` permet d'accéder à un *filewrapper*, c'est à dire à des fichiers embarqués dans l'exécutable (voir chapitre 35 page 149).

12.1 Constructor

12.2 Setter

12.2.1 Setter `setCurrentDirectory`

```
setter setCurrentDirectory ??@string inDirectory ;
```

12.3 Getters

12.3.1 Getter `allTextFilePathes`

```
getter allTextFilePathes -> @stringlist ;
```

12.3.2 Getter `allDirectoryPathes`

```
getter allDirectoryPathes -> @stringlist ;
```

12.3.3 Getter `currentDirectory`

```
getter currentDirectory -> @string ;
```

12.3.4 Getter `allFilePathesWithExtension`

```
getter allFilePathesWithExtension ??@string inExtension -> @stringlist ;
```

12.3.5 Getter `directoryExistsAtPath`

```
getter directoryExistsAtPath ??@string inPath -> @bool ;
```

12.3.6 Getter fileExistsAtPath

```
getter fileExistsAtPath ??@string inPath -> @bool ;
```

12.3.7 Getter textFileContentsAtPath

```
getter textFileContentsAtPath ??@string inPath -> @string ;
```

12.3.8 Getter binaryFileContentsAtPath

```
getter binaryFileContentsAtPath ??@string inPath -> @data ;
```

12.3.9 Getter absolutePathForPath

```
getter absolutePathForPath ??@string inPath -> @string ;
```

Chapitre 13

Le type `@location`

Un objet de type `@location` a pour valeur une position dans un texte source. Les objets de ce types sont utilisés dans les messages d'erreurs et les messages d'alerte pour indiquer à l'utilisateur la position de l'erreur ou de l'alerte.

13.1 Le mot réservé `here`

Le mot réservé `here` contient la position courante de l'analyse du texte source. Il doit être considéré comme un opérande particulier d'une expression, et, à ce titre, peut apparaître dans toute expression. On peut ainsi écrire :

```
const @location currentLocation := here ;
```

Plus précisément, la position capturée est le début du dernier *token* analysé. Ainsi, si l'on écrit :

```
$token$ ;
...
@location currentLocation := here ;
```

La position capturée est la position du premier caractère du token correspondant à `$token$`. Si `here` est appelé avant que le premier token soit analysé, la position capturée est le premier caractère du texte source.

13.2 Constructor

13.2.1 Constructeur `nowhere`

Returns an `@location` that does not point out any location.

```
constructor @location nowhere -> @location
```

Disponibilité : disponible en GALGAS version 2.1.2 et ultérieure.

Discussion: The returned object responds `true` to the @location isNowhere getter (page 66).

13.3 Getters

13.3.1 Getter column

Returns an @uint value containing the column of the receiver's value.

```
getter @location column -> @uint ;
```

Disponibilité : disponible en GALGAS version 1.8.2 et ultérieure.

Discussion: this getter raises a run-time error if the receiver's value responds true to the @location isNowhere getter (page 66).

13.3.2 Getter isNowhere

Returns an @bool value indicating whether the receiver'value points out a source location or does not.

```
getter @location isNowhere -> @bool ;
```

Disponibilité : disponible en GALGAS version 2.1.2 et ultérieure.

Discussion: this getter returns true if the receiver's value does not point out an actual location in a text source (i.e. it has been constructed using the nowhere constructor), and false if the receiver's value points out an actual location in a text source (i.e. it has been constructed using the here keyword.

13.3.3 Getter line

Returns an @uint value containing the line of the receiver's value.

```
getter @location line -> @uint ;
```

Disponibilité : disponible en GALGAS version 1.8.2 et ultérieure.

Discussion: this getter raises a run-time error if the receiver's value responds true to the @location isNowhere getter (page 66).

13.3.4 Getter locationIndex

Returns an @uint value containing the the offset from the the beginning of the source of the location defined by receiver's value.

```
getter @location locationIndex -> @uint ;
```

Disponibilité : disponible en GALGAS version 1.8.2 et ultérieure.

Discussion: this getter raises a run-time error if the receiver's value responds true to the @location is-

Nowhere getter (page 66).

13.3.5 Getter `locationString`

returns an `@string` object that contains a string representation of the location defined by receiver's value.

```
getter @location locationString -> @string ;
```

Disponibilité : disponible en GALGAS version 1.8.2 et ultérieure.

Discussion: this getter raises a run-time error if the receiver's value responds `true` to the @location isNowhere getter (page 66).

Chapitre 14

Le type `@object`

Chapitre 15

Le type `@sint`

An `@sint` object value is a 32-bit signed integer value. You can initialize an `@sint` object from an 32-bit signed integer constant :

```
@sint mySignedInteger := 123_456S ;
```

Note that a 32-bit signed integer constant is characterized by the 'S' suffix.

15.1 Constructors

15.1.1 Constructeur `min`

Returns an `@sint` object that the minimum value of the 32-bit signed range.

```
constructor @sint min -> @sint
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: the returned value is -2^{31}.

15.1.2 Constructeur `max`

Returns an `@sint` object that the maximum value of the 32-bit signed range.

```
constructor @sint max -> @sint
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: the returned value is $2^{31} - 1$.

15.2 Getters

15.2.1 Getter double

Returns the receiver's value converted in a @double object.

```
getter @sint double -> @double ;
```

Disponibilité : disponible en GALGAS version 1.9.8 et ultérieure.

Discussion: as a 32-bit integer value can always be converted in a @double value, this getter never fails.

15.2.2 Getter sint64

Returns the receiver's value in an @sint64 (page 73) (64-bit signed integer) object.

```
getter @sint sint64 -> @sint64 ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: as a 32-bit signed value can always be converted in a 64-bit signed value, this getter never fails.

This getter is the only way to convert an @sint (page 69) object into an @sint64 (page 73) object.

15.2.3 Getter string

Returns a decimal string representation of the receiver's value.

```
getter @sint string -> @string ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: for an hexadecimal string representation of the receiver's value, see @uint hexString getter (page 84) and @uint xString getter (page 87).

15.2.4 Getter uint

Returns the receiver's value in an @uint (page 83) (32-bit unsigned integer) object.

```
getter @sint uint -> @uint ;
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: an error is raised is receiver's value is negative.

This getter is the only way to convert an @sint (page 69) object into an @uint (page 83) object.

15.2.5 Getter uint64

Returns the receiver's value in an @uint64 (page 90) (64-bit unsigned integer) object.

```
getter @sint uint64 -> @uint64 ;
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: an error is raised is receiver's value is negative.

This getter is the only way to convert an @sint (page 69) object into an @uint64 (page 90) object.

15.3 Incrementation and decrementation

The @sint (page 69) supports incrementation and decrementation instructions.

```
@sint n := ...; n ++; # Incrementation
@sint p := ...; p -; # Decrementation
```

The incrementation instruction raises an error if receiver's value is equal to $2^{31} - 1$.

The decrementation instruction raises an error if receiver's value is equal to -2^{31}.

Note that incrementation and decrementation are not available within an expression.

15.4 Arithmetic Operators

The @sint type supports the five arithmetic diadic operators :

Operator	Operation
$+$	Addition
$-$	Substraction
$*$	Multiplication
$/$	Division
mod	Modulo

Theses operators require both arguments to be @sint objects.

A run-time error is raised if the operation leads to a 32-bit signed overflow.

The @sint type supports the following arithmetic unary operators :

Operator	Operation
$+$	No operation
$-$	Negate

This operator returns the receiver's value (an @sint object). A run-time error is raised if "-" operator is invoked on an object whose value is -2^{31}.

15.5 Shift Operators

The `@sint` type supports right and left shift operators :

<<	Left shift
>>	Right shift

Theses operators require the right argument to be `@sint` object, and the left argument to be `@uint` object.

Note the right shift inserts a zero bit in the most significant bit location if the receiver's value is negative, and a one bit otherwise (it is a arithmetic right shift).

The actual amount of the shift is the value of the right-hand operand masked by 31, i.e. the shift distance is always between 0 and 31.

15.6 Logical Operators

The `@sint` type supports the three bit-wise logical operators :

&	Bit-wise and
|	Bit-wise or
^	Bit-wise exclusive or

Theses operators require both arguments to be `@sint` objects.

The `@sint` type supports the bit-wise logical unary operator :

~	Bit-wise complementation

This operator returns an `@sint` object.

15.7 Comparison Operators

The `@sint` type supports the six comparison operators :

=	Equality
!=	Non Equality
<	Strict Lower Than
<=	Lower or Equal
>	Strict Greater Than
>=	Greater or Equal

Theses operators require both arguments to be `@sint` objects, and return a `@bool` object.

Chapitre 16

Le type `@sint64`

An `@sint64` object value is a 64-bit signed integer value. You can initialize an `@sint64` object from an 64-bit signed integer constant :

```
@sint64 mySignedInteger := 123_456LS ;
```

Note that a 64-bit signed integer constant is characterized by the 'LS' suffix.

16.1 Constructors

16.1.1 Constructeur `min`

Returns an `@sint64` object that the minimum value of the 64-bit signed range.

```
constructor @sint64 min -> @sint64
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: the returned value is -2^{63}.

16.1.2 Constructeur `max`

Returns an `@sint64` object that the maximum value of the 64-bit signed range.

```
constructor @sint64 max -> @sint64
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: the returned value is $2^{63} - 1$.

16.2 Getters

16.2.1 Getter double

Returns the receiver's value converted in a @double object.

```
getter @sint64 double -> @double ;
```

Disponibilité : disponible en GALGAS version 1.9.8 et ultérieure.

Discussion: as a 64-bit integer value can always be converted in a @double value, this getter never fails.

16.2.2 Getter sint

Returns the receiver's value in an @sint (page 69) (32-bit signed integer) object.

```
getter @sint64 sint -> @sint ;
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: an error is raised is receiver's value is lower than -2^{31} or greater than $2^{31} - 1$.

This getter is the only way to convert an @sint64 (page 73) object into an @sint (page 69) object.

16.2.3 Getter string

Returns a decimal string representation of the receiver's value.

```
getter @sint64 string -> @string ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: this getter never fails.

16.2.4 Getter uint

Returns the receiver's value in an @uint (page 83) (32-bit unsigned integer) object.

```
getter @sint64 uint -> @uint ;
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: an error is raised is receiver's value is negative or greater than $2^{32} - 1$.

This getter is the only way to convert an @sint64 (page 73) object into an @uint (page 83) object.

16.2.5 Getter `uint64`

Returns the receiver's value in an `@uint64` (page 90) (64-bit unsigned integer) object.

```
getter @sint64 uint64 -> @uint64 ;
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: this getter raises a run-time error if the receiver's value is negative.

This getter is the only way to convert an `@sint64` (page 73) object into an `@uint64` (page 90) object.

16.3 Incrementation and decrementation

The `@sint64` (page 73) supports incrementation and decrementation instructions.

```
@sint64 n := ... ; n ++ ; # Incrementation
@sint64 p := ... ; p - ; # Decrementation
```

The incrementation instruction raises an error if receiver's value is equal to $2^{63} - 1$.

The decrementation instruction raises an error if receiver's value is equal to -2^{63}.

Note that incrementation and decrementation are not available within an expression.

16.4 Arithmetic Operators

The `@sint64` type supports the five arithmetic diadic operators :

$+$	Addition
$-$	Substraction
$*$	Multiplication
$/$	Division
mod	Modulo

Theses operators require both arguments to be `@sint64` objects.

A run-time error is raised if the operation leads to a 64-bit signed overflow.

The `@sint` type supports the following arithmetic unary operators :

$+$	No operation
$-$	Negate

This operator returns the receiver's value (an `@sint` object). A run-time error is raised if "-" operator is invoked on an object whose value is -2^{63}.

16.5 Shift Operators

The `@sint64` type supports right and left shift operators :

<<	Left shift
>>	Right shift

Theses operators require the right argument to be `@sint64` object, and the left argument to be `@uint` object.

Note the right shift inserts a zero bit in the most significant bit location if the receiver's value is negative, and a one bit otherwise (it is a arithmetic right shift).

The actual amount of the shift is the value of the right-hand operand masked by 63, i.e. the shift distance is always between 0 and 63.

16.6 Logical Operators

The `@sint64` type supports the three bit-wise logical operators :

&	Bit-wise and
$\vert$	Bit-wise or
^	Bit-wise exclusive or

Theses operators require both arguments to be `@sint64` objects.

The `@sint64` type supports the bit-wise logical unary operator :

~	Bit-wise complementation

This operator returns an `@sint64` object.

16.7 Comparison Operators

The `@sint64` type supports the six comparison operators :

=	Equality
!=	Non Equality
<	Strict Lower Than
<=	Lower or Equal
>	Strict Greater Than
>=	Greater or Equal

Theses operators require both arguments to be `@sint64` objects, and return a `@bool` object.

Chapitre 17

Le type `@string`

A `@string` object value is an Unicode character string value. The @string type defines several constructors, getters constant methods and setters, described below.

Literal String Constants. Characters strings are written enclosed within quotation marks (") characters, as in many languages. For example : "a string". Note that a literal string constant is an actual @string object, so a getter can be used on it. For example : `["ae"uppercaseString]` returns the "AE" string.

17.1 Getters

17.1.1 Getter `containsCharacter`

Returns true if the receiver contains the given charactezr, and false oteherwise.

```
getter @string containsCharacter
 ?@char inCharacter
 -> @bool ;
```

Availability: available in GALGAS 2.5.0 and later.

```
@string s := "abcdef";
@string s2 := [s rightSubString!3]; # The value of s2 is "def"
```

17.1.2 Getter `subString`

Creates and returns the string built with the *inLength* last characters of the receiver. If the receiver contains less than inLength characters, the receiver's value is returned.

```
getter @string subString
 ?@uint inStart
 ?@uint inLength
 -> @string ;
```

Availability: available in GALGAS 1.7.8 and later.

Chapitre 18

Le type `@stringset`

An `@stringset` object value is a set of `@string` values.

18.1 Constructors

18.1.1 Constructeur `emptySet`

Creates and returns an empty `@stringset` object.

```
constructor @stringset emptySet -> @stringset
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

18.1.2 Constructeur `setWithString`

Creates and returns an `@stringset` object that contains the value of the *inString* argument object.

```
constructor @stringset setWithString
 ?@string inString
 -> @stringset
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

18.2 Getters

18.2.1 Getter `count`

Returns the number of strings in the set.

```
getter @stringset count -> @uint ;
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

18.2.2 Getter `hasKey`

Returns a boolean value that indicates whether the value of *inString* argument is present in the set.

```
getter @stringset hasKey
 ?@string inString
 -> @bool ;
```

Availability: available in GALGAS 1.3.0 and later.

Discussion: returns `true` if the value of *inString* argument is present in the set, `false` otherwise.

18.2.3 Getter `anyString`

Retourne une des chaînes de caractères contenue dans le receveur.

```
getter @stringset anyString -> @string ;
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: si le receiveur est vide, une erreur d'exécution est déclenchée.

18.3 Setter

18.3.1 Setter `removeKey`

Removes the value of *inString* argument from the receiver's value.

```
setter @stringset removeKey
 ?@string inString
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: if the receiver's value does not contain the value of *inString* argument, this setter leaves the receiver's value unchanged.

18.4 the += Operator

The += operator adds a string value to the receiver. If the receiver's value already contains the added value, this operator has no effect.

exemple :

```
@string aString := ... ;
@stringset aStringSet := ... ;
aStringSet += !aString ;
```

18.5 the & Operator

The & operator returns the intersection of its operand values.

exemple :

```
@stringset s1 := ... ;
@stringset s2 := ... ;
@stringset s := s1 & s2 ; # s is the intersection of s1 and s2
```

18.6 the | Operator

The | operator returns the union of its operand values.

exemple :

```
@stringset s1 := ... ;
@stringset s2 := ... ;
@stringset s := s1 | s2 ; # s is the union of s1 and s2
```

18.7 the − Operator

The − operator returns the difference of its operand values.

exemple :

```
@stringset s1 := ... ;
@stringset s2 := ... ;
@stringset s := s1 - s2 ; \# s is the difference of s1 and s2
```

18.8 Enumerating `@stringset` objects

The `foreach` instruction can be used for enumerating `@stringset` values ; enumeration is performed in the ascending order, or in the reverse alphabetical order using the '>' qualifier.

`@stringset s := ...;`
foreach s **do**
`# the key constant has the value of current entry of s stringset`
end foreach ;

18.9 Comparison Operators

The `@stringset` type supports the six comparison operators :

$=$	Equality
$!=$	Non Equality
$<$	Strict Inclusion
$<=$	Inclusion or Equality
$>$	Strict Greater
$>=$	Greater or Equality

Theses operators require both arguments to be `@stringset` objects, and return a `@stringset` object.

Chapitre 19

Le type `@type`

Chapitre 20

Le type `@uint`

An `@uint` object value is a 32-bit unsigned integer value. You can initialize an `@uint` object from an unsigned integer constant :

```
@uint myUnsignedInteger := 123_456 ;
```

Note that a 32-bit unsigned integer constant is characterized by no suffix.

20.1 Constructors

20.1.1 Constructeur `errorCount`

Returns an `@uint` object that contains the number of errors.

```
constructor @uint errorCount -> @uint
```

Disponibilité : disponible en GALGAS version 1.4.9 et ultérieure.

Discussion: The returned value is the cumulative count of errors from the beginning of execution.

Exemple :

```
@uint x := [@uint errorCount] ;
```

20.1.2 Constructeur `max`

Returns an `@uint` object that the maximum value of the 32-bit unsigned range.

```
constructor @uint max -> @uint
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: The returned value is $2^{32} - 1$ (4294967295).

20.1.3 Constructeur `valueWithMask`

Returns an `@uint` object with bits from *inLowerIndex* to *inUpperIndex* equal to 1.

```
constructor @uint valueWithMask
 ?@uint inLowerIndex
 ?@uint inUpperIndex
 -> @uint
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion : a run-time error is raised if *inLowerIndex* $>$ *inUpperIndex* or if *inUpperIndex* $>$ *31*.

Exemple :

```
@uint x := [@uint valueWithMask !2 !4] ; # x is equal to 28 (11100 in binary)
```

20.1.4 Constructeur `warningCount`

Returns an `@uint` object that contains the number of warnings.

```
constructor @uint warningCount -> @uint
```

Disponibilité : disponible en GALGAS version 1.4.9 et ultérieure.

Discussion: The returned value is the cumulative count of warnings from the beginning of execution.

20.2 Getters

20.2.1 Getter `double`

Returns the receiver's value converted in a `@double` object.

```
getter @uint double -> @@double ;
```

Disponibilité : disponible en GALGAS version 1.9.8 et ultérieure.

Discussion: as a 32-bit integer value can always be converted in a `@double` value, this getter never fails.

20.2.2 Getter `hexString`

Returns the an hexadecimal string representation of the receiver value, prefixed by the string "0x".

```
getter @uint hexString -> @@string ;
```

Disponibilité : disponible en GALGAS version 1.5.2 et ultérieure.

Discussion: for getting an hexadecimal representation string without '0x' prefix, see @uint xString getter (page 87).

20.2.3 Getter `isInRange`

Returns an `@range` value indicating whether the receiver'value belongs to a range.

```
getter @uint isInRange
  ?@range
  -> @@bool ;
```

Availability: available in GALGAS 2.3.0 and later.

Discussion: for a receiver's value equal to v and a range of length $length$ starting at $start$, it returns `true` if $((v \geqslant start)\ and\ (v < (start + length)))$, and `false` otherwise.

20.2.4 Getter `isUnicodeValueAssigned`

Returns an `@bool` value indicating whether the receiver'value represents an assigned Unicode character.

```
getter @uint isUnicodeValueAssigned -> @@bool ;
```

Disponibilité : disponible en GALGAS version 1.8.3 et ultérieure.

Discussion: it returns `true` if the receiver value represents an assigned Unicode character, `false` and otherwise.

Exemple :

```
[0xFFFF isUnicodeValueAssigned] # is false, as \uFFFF is not assigned.
[0x41 isUnicodeValueAssigned] # is true, as \u0041 is assigned (LATIN CAPITAL LETTER A).
```

20.2.5 Getter `lsbIndex`

Returns an `@uint` value of the index of the most significant bit of the receiver value.

```
getter @uint lsbIndex -> @uint ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: it raises a run-time error if the receiver value is zero.

Exemple :

```
@uint value := 192 ; # 192 is 11000000 in binary
@uint x := [value lsbIndex] ; # x is equal to 7
```

The most significant bit of 192 is the 7th bit.

20.2.6 Getter `significantBitCount`

Returns the number of bits needed to express the receiver value.

```
getter @uint significantBitCount -> @uint ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: it the receiver value is zero, it returns 0 ; otherwise, it returns the most significant bit index plus one.

Exemple :

```
@uint value := 145 ; # 145 is 10010001 in binary
@uint x := [value significantBitCount] ; # x is equal to 8
```

20.2.7 Getter `sint`

Returns the receiver's value in an `@sint` (page 69) (32-bit signed integer) object.

```
getter @uint sint -> @sint ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: an error is raised is receiver's value is greater than $2^{31} - 1$.

This getter is the only way to convert an `@uint` (page 83) object into an `@sint` (page 69) object.

20.2.8 Getter `sint64`

Returns the receiver's value in an `@sint64` (page 73) (64-bit signed integer) object.

```
getter @uint sint64 -> @sint64 ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: as a 32-bit unsigned value can always be converted in a 64-bit signed value, this getter never fails.

This getter is the only way to convert an `@uint` (page 83) object into an `@sint64` (page 73) object.

20.2.9 Getter `string`

Returns a decimal string representation of the receiver's value.

```
getter @uint string -> @string ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: for an hexadecimal string representation of the receiver's value, see @uint hexString getter (page 84) and @uint xString getter (page 87).

20.2.10 Getter `uint64`

Returns the receiver's value in an `@uint64` (page 90) (64-bit unsigned integer) object.

```
getter @uint uint64 -> @uint64 ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: as a 32-bit unsigned value can always be converted in a 64-bit unsigned value, this getter never fails.

This getter is the only way to convert an `@uint` (page 83) object into an `@uint64` (page 90) object.

20.2.11 Getter `xString`

Returns an hexadecimal string representation of the receiver's value (without any prefix).

```
getter @uint xString -> @string ;
```

Disponibilité : disponible en GALGAS version 1.9.10 et ultérieure.

Discussion: for an decimal string representation of the receiver's value, see the @uint hexString getter (page 84) ; for a decimal string representation of the receiver's value, see the @uint string getter (page 86).

20.3 Incrementation and decrementation

The `@uint` (page 83) supports incrementation and decrementation instructions.

```
@uint n := ... ; n ++ ; # Incrementation
@uint p := ... ; p -- ; # Decrementation
```

The incrementation instruction raises an error if receiver's value is equal to $2^{32} - 1$; the incrementation instruction raises an error if receiver's value is equal to 0.

Note that incrementation and decrementation are not available within an expression.

20.4 Arithmetic Operators

The `@uint` type supports the five arithmetic diadic operators :

$+$	Addition
$-$	Substraction
$*$	Multiplication
$/$	Division
mod	Modulo

Theses operators require both arguments to be `@uint` objects.

A run-time error is raised if the operation leads to a 32-bit unsigned overflow.

The `@uint` type supports the following arithmetic unary operator :

$+$	No operation

This operator returns the receiver's value (an `@uint` object).

20.5 Shift Operators

The `@uint` type supports right and left shift operators :

$<<$	Left shift
$>>$	Right shift

Theses operators require both arguments to be `@uint` objects.

Note the right shift inserts always a zero bit in the most significant bit location (it is a logical right shift).

The actual amount of the shift is the value of the right-hand operand masked by 31, i.e. the shift distance is always between 0 and 31.

20.6 Logical Operators

The `@uint` type supports the three bit-wise logical operators :

$\&$	Bit-wise and
$\vert$	Bit-wise or
ˆ	Bit-wise exclusive or

Theses operators require both arguments to be `@uint` objects.

The `@uint` type supports the bit-wise logical unary operator :

$\sim$	Bit-wise complementation

This operator returns an `@uint` object.

20.7 Comparison Operators

The `@uint` type supports the six comparison operators :

$=$	Equality
$!=$	Non Equality
$<$	Strict Lower Than
$<=$	Lower or Equal
$>$	Strict Greater Than
$>=$	Greater or Equal

Theses operators require both arguments to be `@uint` objects, and return a `@bool` object.

Chapitre 21

Le type @uint64

An @uint64 object value is a 64-bit unsigned integer value. You can initialize an @uint64 object from a 64-bit unsigned integer constant :

```
@uint64 myUnsignedInteger := 123_456L ;
```

Note the 'L' suffix is required for a 64-bit unsigned integer constant.

21.1 Constructeurs

21.1.1 Constructeur max

Returns an @uint64 object that the maximum value of the 64-bit unsigned range.

```
constructor @uint64 max -> @uint64
```

Disponibilité : disponible en GALGAS version 1.3.0 et ultérieure.

Discussion: The returned value is $2^{64} - 1$.

21.1.2 Constructeur uint64BaseValueWithCompressedBitString

Returns an @uint64 object computed from a string containing '0', '1' or 'X' characters, replacing all occurrences of 'X' by '0'.

```
constructor @uint64 uint64BaseValueWithCompressedBitString
 ?@string inBitString
 -> @uint64
```

Disponibilité : disponible en GALGAS version 1.6.4 et ultérieure.

Discussion : the inBitString argument should contain only '0', '1' or 'X' characters. A run time exception is raised if an other character appears.

This constructor considers the *inBitString* argument value as a binary encoding of an integer value. First, it internally replaces all ’X’s by ’0’s, and then converts the resulting string into an integer value that is the one returned by this constructor.

Note that the first character of the *inBitString* argument value corresponds to the most significant bit of the converted value.

Exemple :

```
@uint64 v [uint64BaseValueWithCompressedBitString !"01XX10"] ;
log v ; # Displays <@uint64:18> ;
```

21.1.3 Constructeur `uint64MaskWithCompressedBitString`

Returns an `@uint64` object computed from a string containing ’0’, ’1’ or ’X’ characters, replacing all occurrences of ’0’ by ’1’ and all occurrences of ’X’ by ’0’.

```
constructor @uint64 uint64MaskWithCompressedBitString
 ?@string inBitString
 -> @uint64
```

Disponibilité : disponible en GALGAS version 1.6.4 et ultérieure.

Discussion : the *inBitString* argument should contain only ’0’, ’1’ and ’X’ characters. A run time exception is raised if an other character appears.

This constructor considers the *inBitString* argument value as a binary encoding of an integer value. First, it internally replaces all ’0’s by ’1’s and all ’X’s by ’0’s, and then converts the resulting string into an integer value that is the one returned by this constructor.

Note that the first ’0’ or ’1’ character of the *inBitString* argument value corresponds to the most significant Bit of the converted value.

Exemple :

```
@uint64 v [uint64MaskWithCompressedBitString !"01XX10"] ;
log v ; \# Displays <@uint64:51> ;
```

21.1.4 Constructeur `uint64WithBitString`

Returns an `@uint64` object computed from a string containing ’0’ or ’1’ characters.

```
constructor @uint64 uint64WithBitString
 ?@string inBitString
 -> @uint64
```

Disponibilité : disponible en GALGAS version 1.6.4 et ultérieure.

Discussion : the *inBitString* argument should contain only ’0’ and ’1’ characters. A run time exception is raised if an other character appears.

This constructor considers the *inBitString* argument value as a binary encoding of an integer value. It returns an `@uint64` object containing the converted value.

Note that the first '1' character of the *inBitString* argument value corresponds to the most significant bit of the converted value.

Exemple :

```
@uint64 v [uint64WithBitString !"0101"]] ;
log v ; # Displays <@uint64:5> ;
```

21.2 Getters

21.2.1 Getter `double`

Returns the receiver's value converted in a `@double` object.

```
getter @uint64 double -> @double ;
```

Disponibilité : disponible en GALGAS version 1.9.8 et ultérieure.

Discussion: as a 64-bit integer value can always be converted in a `@double` value, this getter never fails.

21.2.2 Getter `hexString`

Returns the an hexadecimal string representation of the receiver value, prefixed by the string "0x".

```
getter @uint64 hexString -> @string ;
```

Disponibilité : disponible en GALGAS version 1.5.2 et ultérieure.

Discussion: for getting an hexadecimal representation string without "0x" prefix, see @uint64 xString getter (page 94).

21.2.3 Getter `sint`

Returns the receiver's value in an `@sint` (page 69) (32-bit signed integer) object.

```
getter @uint64 sint -> @sint ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: an error is raised is receiver's value is greater than $2^{31} - 1$.

This getter is the only way to convert an `@uint64` (page 90) object into an `@sint` (page 69) object.

21.2.4 Getter `sint64`

Returns the receiver's value in an `@sint64` (page 73) (64-bit signed integer) object.

```
getter @uint64 sint64 -> @sint64 ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: an error is raised is receiver's value is greater than $2^{63} - 1$.

This getter is the only way to convert an `@uint64` (page 90) object into an `@sint64` (page 73) object.

21.2.5 Getter `string`

Returns a decimal string representation of the receiver's value.

```
getter @uint64 string -> @string ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: for an hexadecimal string representation of the receiver's value, see @uint64 hexString getter (page 92) and @uint64 xString getter (page 94).

21.2.6 Getter `uint`

Returns the receiver's value in an `@uint` (page 83) (32-bit unsigned integer) object.

```
getter @uint64 uint -> @uint ;
```

Disponibilité : disponible en GALGAS version 1.6.12 et ultérieure.

Discussion: an error is raised is receiver's value is greater than $2^{32} - 1$.

This getter is the only way to convert an `@uint64` (page 90) object into an `@uint` (page 83) object.

21.2.7 Getter `uintSlice`

Returns an `@uint` (page 83) value, extracted from a bit slice of the receiver's value.

```
getter @uint64 uintSlice
 ?@uint inStartBit
 ?@uint inBitCount
 -> @uint ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: the receiver's value is right shifted by *inStartBit*, and the resulted value is and'ed with a mask equal to $2^{inBitCount} - 1$.

Exemple :

```
@uint64 v := 0x1234_5678_9ABC_DEF0L ;
@uint result := [v uintSlice !4 !5] ; # The result value is 0x8_9ABC
```

21.2.8 Getter `xString`

Returns an hexadecimal string representation of the receiver's value (without any prefix).

```
getter @uint64 xString -> @string ;
```

Disponibilité : disponible en GALGAS version 1.9.10 et ultérieure.

Discussion: for an decimal string representation of the receiver's value, see the @uint64 hexString getter (page 92) ; for a decimal string representation of the receiver's value, see the @uint64 string getter (page 93).

21.3 Incrementation and decrementation

The `@uint64` (page 90) supports incrementation and decrementation instructions.

```
@uint64 n := ...; n ++; # Incrementation
@uint64 p := ...; p -; # Decrementation
```

The incrementation instruction raises an error if receiver's value is equal to $2^{64} - 1$.

The incrementation instruction raises an error if receiver's value is equal to 0.

Note that incrementation and decrementation are not available within an expression.

21.4 Arithmetic Operators

The `@uint64` type supports the five arithmetic diadic operators :

$+$	Addition
$-$	Substraction
$*$	Multiplication
$/$	Division
mod	Modulo

Theses operators require both arguments to be `@uint64` objects.

A run-time error is raised if the operation leads to a 64-bit unsigned overflow.

The `@uint64` type supports the following arithmetic unary operator :

$+$	No operation

This operator returns the receiver's value (an `@uint64` object).

21.5 Shift Operators

The `@uint` type supports right and left shift operators :

Operator	Description
$<<$	Left shift
$>>$	Right shift

Theses operators require the left argument to be `@uint64` object, and the right argument to be `@uint` object.

Note the right shift inserts always a zero bit in the most significant bit location (it is a logical right shift).

The actual amount of the shift is the value of the right-hand operand masked by 63, i.e. the shift distance is always between 0 and 63.

21.6 Logical Operators

The `@uint64` type supports the three bit-wise logical diadic operators :

Operator	Description
&	Bit-wise and
$\vert$	Bit-wise or
^	Bit-wise exclusive or

Theses operators require both arguments to be `@uint64` objects.

The `@uint64` type supports the bit-wise logical unary operator :

Operator	Description
$\sim$	Bit-wise complementation

This operator returns an `@uint64` object.

21.7 Comparison Operators

The `@uint64` type supports the six comparison operators :

Operator	Description
$=$	Equality
$!=$	Non Equality
$<$	Strict Lower Than
$<=$	Lower or Equal
$>$	Strict Greater Than
$>=$	Greater or Equal

Theses operators require both arguments to be `@uint64` objects, and return a `@bool` object.

Chapitre 22

Le type `list`

22.1 List Type Declaration

A `@list` type declaration names all attributes of the list elements :

```
list @MyList {
  @string mFirstAttribute ;
  @bool mSecondAttribute ;
}
```

22.2 Constructors

22.2.1 The `emptyList` constructor

For every list, an `emptyList` constructor is implicitly declared. It returns an empty list :

```
@MyList aList := [@MyList emptyList] ;
```

22.2.2 The `listWithValue` constructor

A list can be constructed directly with one value :

```
@MyList aList := [@myList listWithValue !"c" !3] ;
```

Using this constructor is equivalent to :

```
@MyList aList := [@MyList emptyList] ;
aList += !"c" !3 ;
```

22.3 Adding elements

22.3.1 The += operator

The `+=` operator adds a new element at the end of the list. The right side expressions should correspond to the attributes declared in the `list` declaration :

```
@MyList aList := ... ;
@string aString := ... ;
@bool aBool := ... ;
aList += !aString !aBool ;''
```

22.3.2 L'instruction .=

L'instruction `cible .= expression ;` concatène la liste définie par la valeur de `expression` à la la liste `cible` :

```
@MyList aList := ... ;
@MyList secondList := ... ;
aList .= secondList ;''
```

22.3.3 The `prependValue` setter

Ce setter a été supprimé ; utiliser le setter *insertAtIndex*.

The `prependValue` setter adds a new element at the begining of the list. The right side expressions should correspond to the attributes declared in the `list` declaration :

```
@MyList aList := ... ;
@string aString := ... ;
@bool aBool := ... ;
[!?aList prependValue !aString !aBool];
```

22.3.4 Setter `insertAtIndex`

Le setter `insertAtIndex` permet d'insérer un nouvel élément à une position quelconque de la liste. Si le type `list` correspondant déclare n champs, l'appel du setter comprend $n + 1$ arguments :
- les n premiers correspondent aux valeurs des champs du nouvel élément inséré ;
- le dernier est l'indice d'insertion, une valeur de type `@uint`.

L'indice d'insertion peut varier entre 0 (insertion au début, comme le faisait le setter *prependValue*), et la longueur courante de la liste (insertion à la fin, comme le fait l'opérateur `+=`, section 22.3.1 page 97). Si la liste est vide, insérer à l'indice 0 est donc la seule possibilité.

Par exemple :

```
@MyList aList := ... ;
@string aString := ... ;
@bool aBool := ... ;
[!?aList insertAtIndex !aString !aBool !0];
```

22.3.5 The concatenation operator

The « . » operator can be used for concatenating two lists of the same type :

```
@MyList firstList := ... ;
@MyList secondList := ... ;
@MyList thirdList := firstList . secondList ;
```

22.4 Removing elements

22.4.1 Setter popFirst

The popFirst setter removes and returns the first element of the list. The right side expressions should correspond to the attributes declared in the list declaration :

```
@MyList aList := ... ;
@string aString ;
@bool aBool ;
[!?aList popFirst ?aString ?aBool];
```

If the list is empty when popFirst setter is invoked, a run-time error is raised and the input arguments are not valuated.

22.4.2 Setter popLast

The popLast setter removes and returns the last element of the list. The right side expressions should correspond to the attributes declared in the list declaration :

```
@MyList aList := ... ;
@string aString ;
@bool aBool ;
[!?aList popLast ?aString ?aBool];
```

If the list is empty when popLast is invoked, a run-time error is raised and the input arguments are not valuated.

22.5 Methods

22.5.1 The first method

The first method returns the first element of the list. The element is not removed. The right side expressions should correspond to the attributes declared in the list declaration :

```
@MyList aList := ... ;
@string aString ;
@bool aBool ;
[aList first ?aString ?aBool];
```

If the list is empty when first is invoked, a run-time error is raised and the input arguments are not valuated.

22.5.2 The `last` method

The `last` method returns the last element of the list. The element is not removed. The right side expressions should correspond to the attributes declared in the `list` declaration :

```
@MyList aList := ... ;
@string aString ;
@bool aBool ;
[aList last ?aString ?aBool];
```

If the list is empty when `last` is invoked, a run-time error is raised and the input arguments are not valuated.

22.6 Getters

22.6.1 Le getter `length`

```
getter length -> @uint ;
```

Le getter `length` getter retourne le nombre d'éléments du receveur.

22.6.2 Le getter `range`

```
getter range -> @range ;
```

The `range` getter returns a range starting at 0 of length equal to the number of elements of the receiver.

22.6.3 Le getter `subListFromIndex`

```
getter subListFromIndex ?@uint inIndex -> @self
```

This getter returns a new list containing the elements of the receiver from the one at a given index to the end. The `inIndex` value should be lower or equal to the length of the receiver's value. If `inIndex` is equal to the length of the receiver, the getter returns an empty list.

22.6.4 Le getter `subListWithRange`

```
getter subListWithRange
  ?@range inRange
  -> @self
```

This getter returns a list containing the elements of the receiver that lie within a given range. The range must not exceed the length of the receiver's value, that is $range_start + range_length \leqslant list_length$. If the range's length is equal to zero, this getter returns an empty list.

22.7 Enumerating a list with a foreach instruction

The `foreach` instruction can be used for enumerating list objects. By default, lists are enumerated in the insertion order ; enumeration in the reverse order is performed using the `>` qualifier.

There are two ways for accessing element values :
- using the implicitly declared constants that receive the current attribute values ;

— declare explicitly constants that receive the current attribute values.

Given the list declaration :

```
list @MyList {
  @string mFirstAttribute ;
  @bool mSecondAttribute ;
}
```

22.7.1 Enumeration using the implicitly declared constants

For every attribute, a constant of the same name is available in the `do` instruction list. Theses constants receive the value of the corresponding attribute of the current element.

```
foreach aList do
  # the mFirstAttribute constant receives the value
  # of the mFirstAttribute attribute of the current element,
  # and the mSecondAttribute constant receives the value
  # of the mSecondAttribute attribute of the current element.
end foreach ;
```

22.7.2 Enumeration using the explicitly declared constants

The `foreach` header declares a sequence of constants, corresponding to the attribute list of the `do` declaration. Theses constants receive the value of the corresponding attribute of the current element.

```
foreach aList (@string kString @bool kBool) do
  # the kString constant receives the value
  # of the mFirstAttribute attribute of the current element,
  # and the kBool constant receives the value
  # of the mSecondAttribute attribute of the current element.
end foreach ;
```

22.7.3 Enumeration in the reverse order

In GALGAS 1.7.3 and later, you can enumerate a list in the reverse order using the `>` qualifier :

```
foreach > aList (@string kString @bool kBool) do
  ...
end foreach ;
```

22.8 Direct Access of an element attribute

In GALGAS 1.7.5 and later, lists can be used as an array. Each element of a list is associated with an `@uint` index, spanning from 0 to element count (value returned by `length` getter) minus one.

The element retrieved with `first` method is at index 0.

The element retrieved with `last` method is at index equal to element count minus one.

22.8.1 Read Access

By default and for every attribute, a getter is provided to retrieve the value of this attribute for an element at a given index. For example, for an attribute named *name*, the *nameAtIndex* getter is provided. It accepts one `@uint` argument, the value of the index.

You can disable the default getter generation, by using `feature nogetter`.

For example :

```
list @MyList {
  @string mFirstAttribute ;
  @bool mSecondAttribute feature nogetter ;
}
...
@MyList aList := ... ;
@string s := [aList mFirstAttributeAtIndex !1] ;
```

One getter is available : `mFirstAttributeAtIndex` ; the `mSecondAttributeAtIndex` getter is not available.

22.8.2 Write Access

By default, no setter is provided for performing a direct write access to an attribute at a given index. You should use `feature setter` for enabling setter generation for a given attribute.

The setter name is the name of the attribute with the first letter capitalized, prefixed by *set* and suffixed by *AtIndex* : for an attribute named *name*, the setter is named *setNameAtIndex*. It accepts two arguments, the first one is the new attribute's value, the second one an `@uint` argument, the value of the index.

For example :

```
list @MyList {
  @string mFirstAttribute feature setter ;
  @bool mSecondAttribute ;
}
...
@string s := ... ;
[!?aList setMFirstAttributeAtIndex !s !1] ;
```

One setter is available : `@setMFirstAttributeAtIndex` ; the `@setMSecondAttributeAtIndex` setter is not available.

22.8.3 Example of read and write accesses

```
list @myList {
  @string name ;
}
...
@myList strList [emptyList] ;
strList += !"a" ;
strList += !"b" ;
strList += !"c" ;
strList += !"d" ;
@string s := [strList nameAtIndex !0] ;
```

```
log s ; # displays LOGGING s: <@string:"a">
s := [strList nameAtIndex !1] ;
log s ; # displays LOGGING s: <@string:"b">
s := [strList nameAtIndex !2] ;
log s ; # displays LOGGING s: <@string:"c">
s := [strList nameAtIndex !3] ;
log s ; # displays LOGGING s: <@string:"d">
[!?strList setNameAtIndex !"x" !0] ;
[!?strList setNameAtIndex !"y" !1] ;
[!?strList setNameAtIndex !"z" !2] ;
[!?strList setNameAtIndex !"t" !3] ;
s := [strList nameAtIndex !0] ;
log s ; # displays LOGGING s: <@string:"x">
s := [strList nameAtIndex !1] ;
log s ; # displays LOGGING s: <@string:"y">
s := [strList nameAtIndex !2] ;
log s ; # displays LOGGING s: <@string:"z">
s := [strList nameAtIndex !3] ;
log s ; # displays LOGGING s: <@string:"t">
```

Chapitre 23

Le type `sortedlist`

Le type `sortedlist` permet de construire des listes ordonnées de valeurs.

23.1 Déclaration

La déclaration d'une `sortedlist` nomme tous les attributs qui composent un élément de liste et la description du tri. Par Exemple :

```
sortedlist @MaListeOrdonnee {
  @char mCaractere ;
  @uint mEntier ;
}{
  mCaractere <, mEntier >
}
```

La description du tri est exprimée par la liste ordonnée des attributs qui interviennent dans le tri, chacun d'eux étant suivi de l'ordre du tri (`<` pour croissant, et `>` pour décroissant). Ainsi, les élements des instances du type liste ordonnée ci-dessus sont triés par ordre croissant du champ caractère, puis par ordre décroissant du champ entier.

Déclarer une `sortedlist` définit implicitement :
- le constructeur `emptySortedList` qui construit une liste vide (section 23.2.1 page 104) ;
- le constructeur `sortedListWithValue` qui construit une liste contenant un élément (section 23.2.2 page 104) ;
- l'opérateur `+=` pour ajouter un élément à une liste ordonnée (section 23.3.1 page 104) ;
- l'opérateur `.=` pour ajouter tous les éléments d'une liste à une liste ordonnée (section 23.3.2 page 104) ;
- l'opérateur `.` pour construire une liste ordonnée à partir de deux listes ordonnées (section 23.3.3 page 105) ;
- le *getter* `length`, qui retourne le nombre d'éléments d'une liste (section 23.4 page 105) ;
- le *setter* `popGreatest`, qui retourne les champs du plus grand élément d'une liste, et retire cet élément de cette liste (section 23.5.1 page 105) ;
- le *setter* `popSmallest`, qui retourne les champs du plus grand élément d'une liste, et retire cet élément de cette liste (section 23.5.2 page 105) ;
- la *méthode* `greatest`, qui retourne les champs du plus grand élément d'une liste sans la modifier (section 23.6.1 page 106) ;
- la *méthode* `smallest`, qui retourne les champs du plus petit élément d'une liste sans la modifier (section 23.6.2 page 106).

23.2 Constructeurs

23.2.1 Constructeur emptySortedList

Le constructeur `emptySortedList` construit et retourne une liste vide. Par exemple :

```
@MaListeOrdonnee uneListe [emptySortedList] ;
```

23.2.2 Constructeur sortedListWithValue

Le constructeur `sortedListWithValue` construit et retourne une liste comprenant un élément. Cet élément est spécifié par les arguments effectifs de l'appel : ce constructeur présente une séquence d'arguments en entrée correspondant aux champs de l'élément. Par exemple :

```
@MaListeOrdonnee uneListe [sortedListWithValue
  !'a' # Affecte au champ mCaractere
  !10  # Affecte au champ mEntier
] ;
```

23.3 Opérateurs

23.3.1 L'opérateur +=

L'opérateur `+=` ajoute un élément à la liste ordonnée, en maintenant la relation d'ordre. L'élément ajouté est spécifié par la séquences des valeurs à affecter à ses champs. Si il y a un ou plusieurs éléments égaux à l'élément ajouté, ce dernier est placé après les éléments existants.

Cette opération est effectuée en $O(log(n))$ où n est le nombre d'éléments de la liste.

Exemple :

```
@MaListeOrdonnee uneListe [emptySortedList] ;
uneListe += !'b' ! 1 ; # b1
uneListe += !'b' ! 2 ; # b2
uneListe += !'d' ! 1 ; # d1
uneListe += !'f' ! 1 ; # f1
uneListe += !'a' ! 1 ; # a1
uneListe += !'c' ! 1 ; # c1
uneListe += !'f' ! 2 ; # f2
```

23.3.2 L'opérateur .=

L'opérateur `.=` ajoute tous les éléments de l'expression à la liste ordonnée, en maintenant la relation d'ordre. Si il y a un ou plusieurs éléments égaux à chaque élément ajouté, ce dernier est placé après les éléments existants.

Exemple :

```
@MaListeOrdonnee uneListe := ... ;
@MaListeOrdonnee autreListe := ... ;
uneListe .= autreListe ;
```

23.3.3 L'opérateur .

L'opérateur . combine deux listes ordonnées. Les éléments de la seconde liste égaux à ceux de la première liste sont placés après ceux de la première liste.

Exemple :

```
@MaListeOrdonnee uneListe := ... ;
@MaListeOrdonnee autreListe := ... ;
@MaListeOrdonnee troisiemeListe := uneListe . autreListe ;
```

23.4 Getter length

Le getter `length` retourne un `@uint` contenant le nombre d'éléments de la liste ordonnée.

23.5 Setters

23.5.1 Setter popGreatest

Ce *setter* retourne les champs du plus grand élément de la liste ordonnée, et le retire. Si la liste est vide, un message d'erreur est affiché, et les variables destinées à recevoir les valeurs des champs sont placées dans l'état *invalide*. Par exemple :

```
@MaListeOrdonnee uneListe := ... ;
...
[!?uneListe popGreatest
  ?@char c
  ?@uint n
] ;
```

Si `uneListe` est vide, les variables `c` et `n` sont placées dans l'état *invalide*.

23.5.2 Setter popSmallest

Ce *setter* retourne les champs du plus petit élément de la liste ordonnée, et le retire. Si la liste est vide, un message d'erreur est affiché, et les variables destinées à recevoir les valeurs des champs sont placées dans l'état *invalide*. Par exemple :

```
@MaListeOrdonnee uneListe := ... ;
...
[!?uneListe popSmallest
  ?@char c
  ?@uint n
] ;
```

Si `uneListe` est vide, les variables `c` et `n` sont placées dans l'état *invalide*.

23.6 Méthodes

23.6.1 La méthode `greatest`

Cette méthode retourne les champs du plus grand élément de la liste ordonnée, sans le retirer. La liste n'est donc pas modifiée. Si elle est vide, un message d'erreur est affiché, et les variables destinées à recevoir les valeurs des champs sont placées dans l'état *invalide*. Par exemple :

```
@MaListeOrdonnee uneListe := ... ;
...
[uneListe greatest
  ?@char c
  ?@uint n
] ;
```

Si `uneListe` est vide, les variables `c` et `n` sont placées dans l'état *invalide*.

23.6.2 La méthode `smallest`

Cette méthode retourne les champs du plus petit élément de la liste ordonnée, sans le retirer. La liste n'est donc pas modifiée. Si elle est vide, un message d'erreur est affiché, et les variables destinées à recevoir les valeurs des champs sont placées dans l'état *invalide*. Par exemple :

```
@MaListeOrdonnee uneListe := ... ;
...
[uneListe smallest
  ?@char c
  ?@uint n
] ;
```

Si `uneListe` est vide, les variables `c` et `n` sont placées dans l'état *invalide*.

23.7 Énumération avec l'instruction `for`

L'instruction `foreach` (section 37.11 page 166) permet d'énumérer les éléments d'une liste ordonnée, par ordre croissant ou décroissant.

Pour effectuer l'énumération par ordre croissant, écrire :

```
foreach uneListe do
  ...
end foreach ;
```

Pour effectuer l'énumération par ordre décroissant, écrire :

```
foreach > uneListe do
  ...
end foreach ;
```

À l'intérieur de la boucle, pour chaque champ des éléments de la liste, un constante dont le nom est celui du champ est définie et prend la valeur du champ correspondant de l'élément courant.

Par exemple :

```
@MaListeOrdonnee uneListe [emptySortedList] ;
uneListe += !'b' ! 1 ; # b1
uneListe += !'b' ! 2 ; # b2
uneListe += !'d' ! 1 ; # d1
uneListe += !'f' ! 1 ; # f1
uneListe += !'a' ! 1 ; # a1
uneListe += !'c' ! 1 ; # c1
uneListe += !'f' ! 2 ; # f2
@string s := "" ;
foreach uneListe do
  s .= [mCaractere string] . [mEntier string] . " " ;
end foreach ;
message s . "\n" ; # Affiche "a1 b2 b1 c1 d1 f2 f1"
s := "" ;
foreach > uneListe do
  s .= [mCaractere string] . [mEntier string] . " " ;
end foreach ;
message s . "\n" ; # Affiche "f1 f2 d1 c1 b1 b2 a1"
```

Chapitre 24

Le type `array`

Le type *array* permet de réaliser des tableaux dont la dimension et le type de l'élément sont fixés à la compilation.

24.1 Déclaration d'un type tableau

La déclaration d'un type tableau contient les informations suivantes :

- le type `@TypeElement` qui cite le type de l'élément de tableau ;
- la dimension du tableau, qui doit être un nombre entier strictement positif ;
- le type `@TypeTableau` qui est le nom donné au type de tableau.

La déclaration d'un type tableau a la syntaxe suivante :

```
array @TypeTableau : @TypeElement [dimension] ;
```

Par exemple :

```
array @monTableau : @string [3] ;
```

24.2 Constructeur d'un type tableau

Le seul constructeur d'un type tableau est le constructeur `new`. Il a pour but de fixer les dimensions initiales du tableau (il pourra ensuite être redimensionné). Il comporte *dimension* arguments de type `@uint`, qui fixent la taille initiale de chaque axe. Par exemple :

```
  @monTableau t [new !2 !3 !4] ;
```

Cette déclaration crée un tableau à $2 * 3 * 4$ éléments. Ces éléments sont par défaut *invalides*, c'est à dire que leur lecture par le getter `valueAtIndex` déclenche une *run-time error*. Pour être valide, un élément doit avoir été initialisé par un appel au setter `setValueAtIndex`.

Il est valide d'affecter la valeur 0 à un ou plusieurs axes. Le tableau ne contient alors aucun élément.

24.3 Accès à un élément

L'accès à la valeur d'un élément s'effectue par le getter `valueAtIndex`. La modification de la valeur d'un élément est réalisée par le setter `setValueAtIndex` ou le setter `forceValueAtIndex`.

24.3.1 Le getter `valueAtIndex`

Ce getter comporte *dimension* arguments de type `@uint`, qui précisent l'indice pour chaque axe. Les indices sont comptés à partir de zéro (comme en C).

Par exemple :

```
@string s := [t valueAtIndex !1 !2 !2] ;
```

Une *run-time error* est déclenchée si un indice dépasse sa borne correspondante, et la valeur retournée est *invalide*. Si les indices ont des valeurs correctes, l'élément est retourné ; si cet élément est invalide, une *run-time error* est déclenchée, et une valeur *invalide* est retournée.

24.3.2 Setter `setValueAtIndex`

Ce setter comporte (*dimension*+1) arguments :
- le premier argument est type `@TypeElement`, et contient la valeur à écrire ;
- les *dimension* suivants arguments sont de type `@uint` et précisent l'indice pour chaque axe.

Les indices sont comptés à partir de zéro (comme en C). Une *run-time error* est déclenchée si un indice dépasse sa borne correspondante, et le tableau est alors non modifié.

Par exemple :

```
@string s := ... ;
[!?t setValueAtIndex !s !1 !2 !2] ;
```

24.3.3 Setter `forceValueAtIndex`

Ce setter comporte (*dimension*+1) arguments :
- le premier argument est type `@TypeElement`, et contient la valeur à écrire ;
- les *dimension* suivants arguments sont de type `@uint` et précisent l'indice pour chaque axe.

Les indices sont comptés à partir de zéro (comme en C). Contrairement au setter `setValueAtIndex`, aucune *run-time error* n'est déclenchée si un indice dépasse sa borne correspondante : le tableau est d'abord agrandi, ce qui ajoute des éléments invalides, puis l'élément désigné par les indices est affecté.

Par exemple :

```
@string s := ... ;
[}?t forceValueAtIndex !s !5 !4 !4] ;
```

24.4 Validité d'un élément

Le getter `isValueValidAtIndex` permet de savoir si un élément est valide ou non, c'est à dire si sa lecture déclenchera une *run-time error*. Le setter `invalidateValueAtIndex` invalide un élément.

24.4.1 Le getter `isValueValidAtIndex`

Ce getter comporte *dimension* arguments de type `@uint`, qui précisent l'indice pour chaque axe. Les indices sont comptés à partir de zéro (comme en C). Une *run-time error* est déclenchée si un indice dépasse sa borne correspondante, et la valeur retournée est *invalide*. Il renvoie une valeur de type `@bool`, suivant que l'élément est valide ou non.

Par exemple :

```
@bool b := [t isValueValidAtIndex !1 !2 !2] ;
```

24.4.2 Setter `invalidateValueAtIndex`

Ce setter comporte *dimension* arguments de type `@uint`, qui précisent l'indice pour chaque axe. Les indices sont comptés à partir de zéro (comme en C). Une *run-time error* est déclenchée si un indice dépasse sa borne correspondante. Il invalide l'élément correspondant, c'est dire qu'un appel au getter `valueAtIndex` pour lire cet élément déclenchera une *run-time error*.

Par exemple :

```
[!?t invalidateValueAtIndex !1 !2 !2] ;
```

24.5 Contrôle des tailles des axes

Le getter `axisCount` renvoie la dimension d'un tableau, c'est à dire le nombre de ces axes, le getter `sizeForAxis` renvoie la taille allouée à un axe particulier. Les setters `setSizeForAxis` et `setSize` permettent de modifier la taille d'un tableau.

24.5.1 Le getter `axisCount`

Ce getter sans argument renvoie un `@uint` qui contient le nombre d'axes d'un tableau. Comme ce nombre est fixé statiquement par la déclaration de type, la valeur retournée est toujours la même, pour toutes les objets d'un même type tableau.

Par exemple, pour la déclaration :

```
array @monTableau : @string [3] ;
```

Pour tous les objets de type `@monTableau`, l'appel au getter `axisCount` renvoie la valeur 3.

24.5.2 Le getter `sizeForAxis`

Ce getter présente un argument de type `@uint` qui est l'indice de l'axe interrogé. Les axes sont numérotés à partir de zéro, c'est à dire que le premier axe a l'indice 0, le deuxième l'indice $1, \ldots$ Une *run-time error* est déclenchée si la valeur de l'argument est supérieure ou égale à la dimension du tableau, et la valeur renvoyée est invalide. Sinon, il renvoie un `@uint` qui contient la taille attribuée à l'axe correspondant.

24.5.3 Le getter `rangeForAxis`

Ce getter présente un argument de type `@uint` qui est l'indice de l'axe interrogé. Les axes sont numérotés à partir de zéro, c'est à dire que le premier axe a l'indice 0, le deuxième l'indice $1, \ldots$ Une *run-time error* est

déclenchée si la valeur de l'argument est supérieure ou égale à la dimension du tableau, et la valeur renvoyée est invalide. Sinon, il renvoie un `@range` qui commence à 0 et qui a pour longueur la taille attribuée à l'axe correspondant.

24.5.4 Setter `setSizeForAxis`

Ce setter permet de changer la taille d'un axe sans changer les tailles attribuées aux autres axes. Il présente deux arguments de type `@uint` :
- le premier est la nouvelle taille ;
- le second est l'indice de l'axe concerné.

Les axes sont numérotés à partir de zéro, c'est à dire que le premier axe a l'indice 0, le deuxième l'indice 1, ... Une *run-time error* est déclenchée si la valeur de l'argument est supérieure ou égale à la dimension du tableau, et le tableau n'est pas modifié.

Diminuer la taille d'un axe fait disparaître des éléments, qui sont alors perdus. Si la nouvelle taille est zéro, le tableau est vidé de tous ses éléments.

Augmenter la taille fait apparaître de nouveaux éléments, qui sont invalides par défaut. Il faudra alors explicitement les initialiser individuellement par un appel au setter `setValueAtIndex`.

24.5.5 Setter `setSize`

Ce setter permet de changer les tailles de tous les axes. Il présente `@uint` arguments de type `@uint` qui contiennent les nouvelles tailles de chaque axe.

Diminuer la taille d'un axe fait disparaître des éléments, qui sont alors perdus. Si une des nouvelles tailles est zéro, le tableau est vidé de tous ses éléments.

Augmenter une taille fait apparaître de nouveaux éléments, qui sont invalides par défaut. Il faudra alors explicitement les initialiser individuellement par un appel au setter `setValueAtIndex`.

24.6 Comparaison

Un type tableau implémente les opérateurs `==` et `!=`. L'égalité de deux tableaux est testé comme suit :
- les tailles de chaque axe doivent être identiques ;
- les éléments doivent être identiques.

Chapitre 25

Le type `class`

25.1 Déclaration d'une classe

Voici différents exemples de déclaration de classes :

```
abstract class @A {
  @uint mA ;
}
class @B extends @A {
  @string mB ;
}
class @C extends @B {
  @data mC ;
}
```

La classe `@A` est abstraite (c'est-à-dire qu'elle ne peut pas être instanciée), la classe `@B` hérite de `@A`. Une classe déclare zéro, un ou plusieurs attributs. L'héritage multiple n'est pas implémenté en GALGAS.

Une classe qui hérite d'une autre peut être abstraite :

```
abstract class @D extends @C {
  ...
 }
```

Une classe non abstraite définit implicitement le constructeur `new`, et des *getters* pour lire les attributs, et des *setters* pour les écrire. On ne peut pas définir explicitement d'autres constructeurs, *getters* ou *setters* à l'intérieur de la classe. Cependant, les extensions (chapitre 34 page 142) permettent de définir *getters*, *méthodes* et *setters* associés à une classe.

25.2 Le constructeur `new`

Le constructeur `new` est implicitement pour toute classe non abstraite (c'est à dire les classes `@B` et `@C`). Ce constructeur présente un argument par attribut déclaré dans la classe instanciée et dans toutes les classes mère. L'ordre des arguments est celui obtenu en parcourant la hiérarchie de classes, en commençant par la classe racine. Par exemple on écrira :

```
@B b [new
  !0 # Attribut mA de @A
```

```
  !"Hello" # Attribut mB de @B
] ;
@C c [new
  !0 # Attribut mA de @A
  !"Hello" # Attribut mB de @B
  ![@data emptyData] # Attribut mC de @C
] ;
```

25.3 Lecture d'un attribut

Par défaut, la lecture d'un attribut est activée par la définition implicite d'un *getter*, dont le nom est le nom de l'attribut. Ainsi, pour une variable `b` de type `@B`, on pourra écrire :

```
@uint v := [b mA] ;
@string s := [b mB] ;
```

Il est possible d'inhiber la génération implicite d'un *getter* de lecture d'un attribut en complétant sa déclaration par `feature nogetter`, comme par exemple :

```
abstract class @A {
  @uint mA feature nogetter ;
}
```

L'écriture `[b mA]` sera alors rejetée par le compilateur.

25.4 Écriture d'un attribut

Par défaut, l'écriture d'un attribut n'est pas activée.

Pour activer la génération d'un *setter* permettant décrire un attribut, compléter la déclaration de cet attribut par `feature setter`. Un *setter* est alors engendré, et porte le nom `set<Attribut>`, c'est à dire le nom de l'attribut avec sa première lettre en majuscule, précédé par `set`. Par exemple :

```
abstract class @A {
  @uint mA feature setter ;
}
```

Pour modifier l'attribut `mA`, on écrira :

```
[!?b setMA !12] ;
```

Si on veut à la fois inhiber la génération implicite d'un *getter* de lecture d'un attribut et engendrer le *setter* d'écriture, il suffit de déclarer l'attribut par :

```
  @uint mA feature nogetter, setter ;
```

Ou encore :

```
  @uint mA feature setter, nogetter ;
```

25.5 Conversions entre objets de classes différentes

Pour toute cette section, nous illustrons les constructions décrites en nous basant sur les trois variables suivantes :

```
@A a ;
@B b := ... ;
@C c := ... ;
```

25.5.1 Affectation polymorphique

GALGAS accepte l'affectation polymorphique qui est par exemple `a := b ;`. Elle est autorisée aussi lors de l'affectation d'une expression effective à un paramètre formel dans une instruction d'appel (de routine, de fonction, de méthode, ...)

L'affectation polymorphique inverse (qui consisterait à écrire `b := a ;`) est logiquement refusée par le compilateur.

Il y a trois constructions qui permettent d'effectuer cette opération :
— l'expression de conversion polymorphique inverse (section 36.1.3 page 152) ;
— l'expression de test du type dynamique (section 36.1.4 page 153) ;
— l'instruction `cast` (section 37.5 page 162).

Pour effectuer ponctuellement une affectation polymorphique inverse, on écrit (les parenthèses sont obligatoires) :

```
@T resultat := (cast expression : @T) ;
```

Si le type dynamique de l'`expression` est `@T` ou une de ses classes héritières, l'expression de conversion polymorphique renvoie un objet de type `@T` contenant la valeur de `expression`. Dans le cas contraire, un message d'erreur est affiché, et la variable `resultat` est non construite.

L'exécution échoue donc avec émission de message d'erreur si la conversion n'est pas possible.

Grâce à l'*expression de test du type dynamique*, il est possible de tester si une conversion est possible. On peut donc écrire :

```
if (expression is @B) then
  const @B variable := (cast expression : @B) ;
  ...
elsif (expression is @C) then
  const @C variable := (cast expression : @C) ;
  ...
else
  message "conversion impossible" ;
end if ;
```

L'instruction `cast` permet simplement d'exprimer de manière plus élégante une série de test de conversion. La forme équivalent à l'instruction `if` précédente est :

```
cast expression
when >= @B variable :
  ...
when >= @C variable :
  ...
else
```

```
  message "conversion impossible" ;
end cast ;
```

Chapitre 26

Le type `enum`

Galgas permet à l'utilisateur de définir des types énumérés.

26.1 Déclaration

La déclaration d'un type `enum` nomme l'ensemble des constantes associées à ce type.

Par exemple :

```
enum @feuTricolore {
  case vert
  case orange
  case rouge
}
```

Plusieurs types énumérés peuvent définir des constantes de même nom.

26.2 Instanciation

Chaque constante définit un constructeur de même nom. On peut ainsi écrire :

```
@feuTricolore feu := [@feuTricolore vert] ;
```

Ou encore :

```
@feuTricolore feu [vert] ;
```

26.3 Comparaison

Un type enuméré accpete les six opérateurs de comparaison (`==`, `!=`, `<`, `<=`, `>` et `>`). L'ordre est celui de la déclaration, c'est-à-dire que :

```
  [@feuTricolore vert] < [@feuTricolore orange] < [@feuTricolore rouge]
```

26.4 Tester une valeur

Il y a deux façons de tester une valeur d'un type énuméré. La première consiste à comparer avec une valeur obtenue par un constructeur, par exemple :

```
if feu == [@feuTricolore orange] then ...
```

La seconde possibilité est d'appeler les *getter* implicitement déclarés : pour chaque constante, un *getter* sans argument nommé `is<Constante>` (le préfixe `is` suivi du nom de la constante, dont le premier caractère est en majuscule) est déclaré ; il renvoie une valeur de type `@bool` qui est vrai si le récepteur a la valeur correspondante :

```
if [feu isOrange] then ...
```

26.5 L'instruction `switch`

L'instruction `switch` (section 37.21 page 180) est dédiée aux types énumérés. On écrit par exemple :

```
@feuTricolore feu = ...
switch feu
case vert : message "vert"
case orange : message "orange"
case rouge : message "rouge"
end
```

26.6 Valeurs associées

Il est possible d'associer des valeurs à chaque constante, ce qui permet d'alléger dans certains cas le code à écrire. Supposons par exemple que l'on ait dans un langage une construction optionnelle :

```
rule regleProduction {
  select
  or
    $option$
    $identifier$ ?let nomOption
  end
}
```

Comment construire l'arbre syntaxique abstrait ? Il y a en fait trois possibilités.

Première solution. La première consiste à considérer la chaîne vide comme significative de l'absence d'option :

```
rule regleProduction {
  @lstring nomOption
  select
    nomOption = ["" nowhere]
  or
    $option$
    $identifier$ ?nomOption
  end
}
```

Évidemment, cette solution est acceptable uniquement si l'information associée est simple, et si une valeur particulière peut être considéré comme l'absence d'option.

Deuxième solution. La deuxième solution fait appel à trois classes :

```
abstract class @abstractOption {}

class @noOption : @abstractOption {}

class @option : @abstractOption { @lstring mOptionName }
```

La construction de l'arbre est réalisée par :

```
rule regleProduction {
  @abstractOption optionAST
  select
    optionAST = @noOption.new
  or
    $option$
    $identifier$ ?let nomOption
    optionAST = @option.new {!nomOption}
  end
}
```

Cette solution, plus générale, est plus lourde à mettre en œuvre : trois classes, et analyser l'option nécessite d'écrire un *getter* abstrait ou une méthode abstraite pour la classe abstraite de base `@abstractOption`, et les redéfinir dans les deux classes héritières `@noOption` et `@option`.

Troisième solution. La troisième et dernière solution consiste à écrire un type énuméré possédant des valeurs associées :

```
enum @option {
  case noOption
  case optionPresente (@lstring optionName)
}
```

À la constante `optionPresente` est associée une valeur de type `@lstring`, identifiée par le nom `optionName`. Ce nom est optionnel, on pourrait écrire `optionPresente (@lstring)`. La construction de l'arbre syntaxique est maintenant réalisée par :

```
rule regleProduction {
  @option optionAST
  select
    optionAST = @option.noOption
  or
    $option$
    $identifier$ ?let nomOption
    optionAST = @option.optionPresente {!optionName:nomOption}
  end
}
```

À la constante `optionPresente` correspond un constructeur de même nom, avec un argument qui correspond à la valeur associée `@lstring optionName`. Le nom `optionName` est utilisé comme sélecteur. Si on avait déclaré la valeur associée sans nom par `optionPresente (@lstring)`, alors l'appel du constructeur serait `@option.optionPresente {!nomOption}`.

Pour tester un type énuméré avec des valeurs associées, on peut appliquer les *getter* décrits à section 26.4 page 117, mais on n'a pas accès aux valeurs associées.

Les six opérateurs de comparaison (`==`, `!=`, `<`, `<=`, `>` et `>`) sont définis sur des types énumérés avec des valeurs associées : l'ordre est celui de la déclaration des constantes, et, en cas d'égalité, les valeurs associées sont comparées les unes après les autres, dans leur ordre de déclaration.

Il n'y a qu'une façon d'extraire les valeurs associées, l'instruction `switch` :

```
switch optionAST
case noOption : ...
case optionPresente (@lstring nomOption) : ...
end
```

`nomOption` est une constante dont la portée s'étend jusqu'à la fin de la branche `case` courante.

Chapitre 27

Le type `graph`

Le type `graph` permet de faire des opérations sur les graphes orientés.

Chaque nœud est identifié par un nom qui est une chaîne de caractères (de type `@string`) ; à chaque nœud sont associées types d'informations :
- une liste de positions dans des textes sources (liste de `@location`) ;
- une information utilisateur dont le type est défini par la déclaration du type graphe.

Un arc est identifié par un couple de nœuds.

Un type `graph` se déclare comme suit :

```
graph @nom_du_type_graph (@nom_liste_information) {
}
```

Le nom `@nom_du_type_graph` est le nom donné au type. Le nom `@nom_liste_information` nomme un type qui spécifie l'information utilisateur associée à chaque nœud.

Attention, le type `@nom_liste_information` est un type *liste*, et l'information utilisateur a pour type l'élement associé, c'est à dire `@nom_liste_information-element`.

Par exemple, si l'on veut manipuler des graphes dont l'information associée est un entier `@uint`, on déclarera :

```
graph @monGraphe (@uintlist) {
}
```

Si l'information associée à chaque nœud est composée d'un entier et d'une chaîne de caractères, il faut déclarer un type liste particulier :

```
list @maListe {
  @uint monInfo1 ;
  @string monInfo2 ;
}
graph @monGraphe (@maListe) {
}
```

27.1 Constructeur `emptyGraph`

```
constructor emptyGraph -> @self ;
```

`emptyGraph` est le seul constructeur d'un graphe. Il instancie un graphe vide.

```
var gr := [@monGraphe emptyGraph] ;
```

27.2 Construire un graphe

Trois setters permettent de construire un graphe :
- un *setter d'insertion* (section 27.2.1 page 121), défini par l'utilisateur, réalise l'ajout d'un nœud ;
- le *setter* `addEdge` (section 27.2.2 page 121), implicitement défini, réalise l'ajout d'un arc ;
- le *setter* `noteNode` (section 27.2.3 page 122), implicitement défini, indique qu'un nœud doit être défini.

Ces trois setters peuvent être appelés dans un ordre quelconque. Il est possible d'entrer un arc alors que ni le nœud origine, ni le nœud destination ne sont définis. Il faudra simplement qu'ils le soient avant que les calculs soient entrepris sur le graphe.

27.2.1 Setter d'insertion

Pour pouvoir entrer un nœud, il faut déclarer explicitement un *setter d'insertion* :

```
graph @monGraphe (@maListe) {
  insert addNode error message "the '%K' node is already declared at %L" ;
}
```

`addNode` est le nom donné au *setter* d'insertion de nœud. Comme dans un graphe, un nœud est unique, la chaîne de caractères qui suit `error message` est le message d'erreur qui est affiché en cas de tentative d'insertion d'un nœud déjà existant. Dans cette chaîne, deux séquences particulières peuvent être utilisées :
- `%K`, qui est remplacée par le nom du nœud ;
- `%L`, qui est remplacée par la désignation de l'endroit dans le texte source où est déclaré le nœud déjà existant.

Un setter d'insertion présente que des arguments d'entrée :
- le premier est toujours de type `@lstring` ; la composante `@string` est le nom du nœud (qui est unique dans un graphe donné), et la composante `@location` est la position dans le texte source où est déclaré le nœud, et est ajoutée à la liste correspondante du nœud ;
- Les arguments suivants correspondent en nombre et en type aux champs du type liste qui définit les informations associées.

Ainsi, le setter d'insertion `addNode` du type de graphe `@monGraphe` possède trois arguments en entrée ;
- le premier de type `@lstring` ;
- le deuxième de type `@uint`, qui correspond au premier champ `monInfo1` du type liste `@maListe` ;
- le troisième de type `@string`, qui correspond au second champ `monInfo2` du type liste `@maListe`.

Il est par exemple appelé comme suit :

```
var @lstring lstr := ... ;
var gr := [@monGraphe emptyGraph] ;
[!?gr addNode !lstr !0 !"xyz"] ;
```

27.2.2 Entrer un arc : setter `addEdge`

```
setter addEdge ??@lstring inSourceNode ??@lstring inTargetNode ;
```

Pour entrer un arc, appeler le setter prédéfini `addEdge`. Celui-ci possède deux arguments d'entrée de type `@lstring` :

— le premier spécifie le nœud origine de l'arc ; la composante `@string` est le nom du nœud origine, et la composante `@location` est ajoutée à la liste du nœud origine ;
— le second spécifie le nœud destination de l'arc ; la composante `@string` est le nom du nœud destination, et la composante `@location` est ajoutée à la liste du nœud destination.

27.2.3 setter noteNode

```
setter noteNode ??@lstring inNode ;
```

Le setter prédéfini `noteNode` permet d'indiquer qu'un nœud doit être défini : il possède un seul argument en entrée, de type `@lstring`, dont la composante `@string` désigne le nom du nœud, et dont la composante `@location` est ajoutée à la liste de ce nœud.

27.3 Enlever des arcs

Deux *setters* permettent d'enlever des arcs à un graphe :
— le *setter* `removeEdgesToNode` (section 27.3.1 page 122) retire tous les arcs qui arrivent à un nœud ;
— le *setter* `removeEdgesToDominators` (section 27.3.2 page 122) retire tous les arcs qui arrivent à un nœud *dominateur*.

27.3.1 Setter removeEdgesToNode

```
setter removeEdgesToNode ??@string inTargetNode ;
```

Ce *setter* ne présente qu'un seul argument, une chaîne qui désigne un nœud du graphe. Son exécution supprime tous les arcs dont le nœud destination est le nœud nommé en argument.

27.3.2 Setter removeEdgesToDominators

```
setter removeEdgesToDominators ;
```

Dans un graphe, un nœud d domine un autre nœud n si chaque chemin à partir du nœud d'entrée vers le nœud n doit passer par d [1].

Ce *setter* considère que les nœuds d'entrée sont les nœuds sans prédécesseur, et retire tous les arcs d'un nœud vers son dominateur.

27.4 Getters

27.4.1 Getter edges

```
getter edges -> @2stringlist ;
```

Ce getter retourne la liste des arcs.

27.4.2 Getter graphviz

```
getter graphviz -> @string ;
```

1. http://en.wikipedia.org/wiki/Dominator_(graph_theory)

Ce getter retourne une chaîne de caractères contenant une description compatible *GraphViz* [2] du graphe.

27.4.3 Getter keyList

```
getter keyList -> @stringlist ;
```

Ce getter retourne la liste des noms de nœuds, aussi bien les nœuds définis (c'est à dire les nœuds pour lesquels un setter d'insertion a été appelé, section 27.2.1 page 121), que les nœuds indéfinis.

27.4.4 Getter lkeyList

```
getter lkeyList -> @stringlist ;
```

Ce getter retourne la liste des noms de nœuds, aussi bien les nœuds définis (c'est à dire les nœuds pour lesquels un setter d'insertion a été appelé, section 27.2.1 page 121), que les nœuds indéfinis. Chaque nœud défini est accompagné de sa position de définition, les nœuds indéfinis sont accompagnés d'une position non définie (équivalente à celle obtenue par le constructeur `nowhere`).

27.4.5 Getter reversedGraph

```
getter reversedGraph -> @T ;
```

Ce getter retourne un graphe de même type que le receiveur, mais dont les arcs sont inversés.

27.4.6 Getter subgraphFromNodes

```
getter subgraphFromNodes
  ??@lstringlist inStartNodes
  ??@stringset inNodesToExclude
  -> @self ;
```

Ce getter retourne le graphe défini par la partie accessible du receveur à partir des nœuds nommés dans `inStartNodes`, en excluant les nœuds nommés dans `inNodesToExclude`.

27.4.7 Getter accessibleNodesFromNodes

```
getter accessibleNodesFromNodes
  ??@lstringlist inStartNodes
  -> @lstringlist ;
```

Ce getter retourne la liste des nœuds accessibles à partir des nœuds nommés dans `inStartNodes`.

27.4.8 Getter undefinedNodeCount

```
getter undefinedNodeCount -> @uint ;
```

Ce getter retourne le nombre de nœuds indéfinis (c'est à dire les nœuds pour lesquels un setter d'insertion n'a été appelé, section 27.2.1 page 121).

27.4.9 Getter undefinedNodeKeyList

```
getter undefinedNodeKeyList -> @stringlist ;
```

2. http://www.graphviz.org/

Ce getter retourne la liste des noms des nœuds indéfinis (c'est à dire les nœuds pour lesquels un setter d'insertion n'a été appelé, section 27.2.1 page 121).

27.4.10 Getter undefinedNodeReferenceList

```
getter undefinedNodeReferenceList -> @lstringlist ;
```

Ce getter retourne la liste des références des nœuds indéfinis (c'est à dire les nœuds pour lesquels un setter d'insertion n'a été appelé, section 27.2.1 page 121). Une référence à un nœud est un `@lstring` dont la chaîne est le nom du nœud, et la composante `@location` une position dans le texte source, définie par un appel à `addEdge` (section 27.2.2 page 121) ou `noteNode` (section 27.2.3 page 122).

27.5 Méthodes

27.5.1 Méthode depthFirstTopologicalSort

```
method depthFirstTopologicalSort
  !@nom_liste_information outSortedInformationList
  !@lstringlist outSortedKeyList
  !@nom_liste_information outUnsortedInformationList
  !@lstringlist outUnsortedKeyList
;
```

Cette méthode effectue un tri topologique du graphe. Tous les arguments sont en sortie :

— le premier argument `outSortedInformationList` est la liste triée des informations utilisateur liées aux nœuds ;
— le deuxième `outSortedKeyList` est la liste triée des noms de nœuds ;
— le troisième `outUnsortedInformationList` est la liste des informations utilisateur liées aux nœuds qui n'ont pas pu être triés ;
— le dernier `outUnsortedKeyList` est la liste des noms de nœuds qui n'ont pas pu être triés.

Si le tri échoue, aucun message d'erreur n'est émis ; il suffit de tester le nombre d'éléments du troisième ou du quatrième argument pour savoir si le tri a réussi.

Les deux premiers arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice. Le premier élément désigne un nœud qui n'a pas de prédecesseur, et le dernier un nœud qui n'a pas de successeur.

Les deux derniers arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice. L'ordre dans lequel les nœuds non triés apparaissent n'est pas défini.

Cette méthode diffère de `topologicalSort` (section 27.5.4 page 125) par le fait que la liste triée est présentée en privilégiant un parcours en profondeur.

27.5.2 Méthode nodesWithNoPredecessor

```
method nodesWithNoPredecessor
  !@nom_liste_information outInformationList
  !@lstringlist outKeyList
;
```

Cette méthode renvoie la liste de tous les nœuds sans prédecesseur. Les deux arguments sont en sortie :

— le premier argument `outInformationList` est la liste des informations utilisateur liées aux nœuds sans prédécesseur ;

— le second `outKeyList` est la liste des noms de nœuds sans prédécesseur.

Les deux arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice.

27.5.3 Méthode nodesWithNoSucccessor

```
method nodesWithNoSucccessor
  !@nom_liste_information outInformationList
  !@lstringlist outKeyList
;
```

Cette méthode renvoie la liste de tous les nœuds sans successeur. Les deux arguments sont en sortie :

— le premier argument `outInformationList` est la liste des informations utilisateur liées aux nœuds sans successeur ;
— le second `outKeyList` est la liste des noms de nœuds sans successeur.

Les deux arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice.

27.5.4 Méthode topologicalSort

```
method topologicalSort
  !@nom_liste_information outSortedInformationList
  !@lstringlist outSortedKeyList
  !@nom_liste_information outUnsortedInformationList
  !@lstringlist outUnsortedKeyList
;
```

Cette méthode effectue un tri topologique du graphe. Tous les arguments sont en sortie :

— le premier argument `outSortedInformationList` est la liste triée des informations utilisateur liées aux nœuds ;
— le deuxième `outSortedKeyList` est la liste triée des noms de nœuds ;
— le troisième `outUnsortedInformationList` est la liste des informations utilisateur liées aux nœuds qui n'ont pas pu être triés ;
— le dernier `outUnsortedKeyList` est la liste des noms de nœuds qui n'ont pas pu être triés.

Si le tri échoue, aucun message d'erreur n'est émis ; il suffit de tester le nombre d'éléments du troisième ou du quatrième argument pour savoir si le tri a réussi.

Les deux premiers arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice. Le premier élément désigne un nœud qui n'a pas de prédecesseur, et le dernier un nœud qui n'a pas de successeur.

Les deux derniers arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice. L'ordre dans lequel les nœuds non triés apparaissent n'est pas défini.

Cette méthode différe de `depthFirstTopologicalSort` (section 27.5.1 page 124) par le fait que l'ordre topologique n'est pas défini.

Chapitre 28

Le type `map`

Un objet de type `map` est une table de symboles, chaque symbole étant associé à des valeurs.

28.1 Déclaration

La déclaration d'un type `map` nomme :
- les attributs qui sont associés à une clé ;
- les *setters* d'insertion ;
- les *méthodes* de recherche ;
- les *setters* de retrait ;

Les clés sont déclarées implicitement et sont du `@lstring` (page 134).

Par exemple :

```
map @MaTable {
  @string mPremier
  @bool mSecond
  insert insertKey error message "the '%K' key is already declared in %L"
  search searchKey error message "the '%K' key is not defined"
  remove removeKey error message "the '%K' key is not defined"
}
```

28.2 Constructeurs

28.2.1 Constructeur `emptyMap`

```
constructor emptyMap -> @T
```

Ce constructeur permet d'instancier une table vide. Exemple :

```
@MaTable uneTable = .emptyMap
```

28.2.2 Constructeur `mapWithMapToOverride`

```
constructor mapWithMapToOverride ?@T inMapToOverride -> @T
```

Ce constructeur permet d'instancier une table vide, qui surcharge la table `inMapToOverride` citée en argument. Exemple :

```
@MaTable uneTable = .emptyMap
@MaTable autreTableTable = .mapWithMapToOverride {!uneTable}
```

28.3 Setters d'insertion

Une `map` peut déclarer zéro, un ou plusieurs *setters* d'insertion. Un *setter* d'insertion permet d'insérer une nouvelle entrée à une table. Une erreur est déclenchée en cas de tentative d'une clé déjà existante.

Un *setter* d'insertion est déclaré par :

```
insert nom error message "message_erreur"
```

L'identificateur `nom` donne un nom au *setter* d'insertion ; ce nom doit être unique parmi les *setters* d'insertion et de retrait. La chaîne de caractères `"message_erreur"` définit le message d'erreur qui est affiché en cas de tentative d'une clé déjà existante. Cette chaîne accepte deux séquences d'échappement :

— `%K`, qui est remplacée par la chaîne de caractères de la clé existante ;
— `%L`, qui est remplacée par la chaîne décrivant la position de la clé existante dans les fichiers source.

Un *setter* d'insertion est appelé dans une *instruction d'appel de setter*, comprenant tous ses arguments en sortie :

— le premier argument est une expression de type `@lstring` qui caractérise la clé à insérer ;
— ensuite, pour chaque attribut déclaré, une expression du type de cet attribut.

Par exemple :

```
@MaTable uneTable = {}
@lstring clef = ...
@string s = ...
@uint v = ...
[!?uneTable insertKey !clef !s !v]
```

28.4 Méthodes de recherche

Une `map` peut déclarer zéro, une ou plusieurs *méthodes* de recherche. Une *méthode* de recherche permet de rechercher une entrée d'une table, et retourne la valeur de ses attributs associés. Une erreur est déclenchée si la clé n'existe pas.

Une *méthode* de recherche est déclarée par :

```
search nom error message "message_erreur"
```

L'identificateur `nom` donne un nom à la *méthode* de recherche ; ce nom doit être unique parmi ces *méthodes*. La chaîne de caractères `"message_erreur"` définit le message d'erreur qui est affiché en cas de recherche d'une clé inexistante. Cette chaîne accepte une séquence d'échappement :

— `%K`, qui est remplacée par la chaîne de caractères de la clé inexistante recherchée ;

Une *méthode* de recherche est appelée dans une *instruction d'appel de méthode* :

— le premier argument (sortie) est une expression de type `@lstring` qui caractérise la clé à rechercher ;
— ensuite, pour chaque attribut déclaré, un argument en entrée nommant une variable destinée à recevoir la valeur de l'attribut correspondant.

Par exemple :

```
@MaTable uneTable = {}
...
@lstring clef = ...
[!?uneTable searchKey !clef ?@string s ?@uint v]
```

28.5 Setters de retrait

Une `map` peut déclarer zéro, un ou plusieurs *setters* de retrait. Un *setter* de recherche permet de retirer une entrée d'une table, et retourne la valeur des attributs de la clé retirée. Une erreur est déclenchée si la clé n'existe pas.

Un *setter* de retrait est déclaré par :

```
remove nom error message "message_erreur"
```

L'identificateur nom donne un nom au *setter* de retrait ; ce nom doit être unique parmi les *setters* d'insertion et de retrait. La chaîne de caractères `"message_erreur"` définit le message d'erreur qui est affiché en cas de recherche d'une clé inexistante. Cette chaîne accepte une séquence d'échappement :

— K% , qui est remplacée par la chaîne de caractères de la clé inexistante à retirer ;

Un *setter* de retrait est appelé dans une *instruction d'appel de setter* :

— le premier argument (sortie) est une expression de type `@lstring` qui caractérise la clé à retirer ;
— ensuite, pour chaque attribut déclaré, un argument en entrée nommant une variable destinée à recevoir la valeur de l'attribut correspondant de la clé retirée.

Par exemple :

```
@MaTable uneTable = {}
...
@lstring clef = ...
[!?uneTable removeKey !clef ?@string s ?@uint v]
```

28.6 Getters

28.6.1 Getter count

```
getter count -> @uint
```

Le *getter* `count` retourne un `@uint` qui contient le nombre d'entrées de la table de premier niveau du receveur.

28.6.2 Getter hasKey

```
getter hasKey ??@string inKey -> @bool
```

Le *getter* `hasKey` retourne un `@bool` qui est `true` si la clé `inKey` est dans la table de premier niveau du receveur, `false` dans le cas contraire.

28.6.3 Getter keyList

```
getter keyList -> @lstringlist
```

Le *getter* `keyList` retourne la liste construite avec toutes les clés de la table de premier niveau du receveur. L'ordre de la liste est l'ordre alphabétique croissant des clés.

28.6.4 Getter keySet

```
getter keySet -> @stringset
```

Le *getter* `keySet` retourne l'ensemble de toutes les clés de la table de premier niveau du receveur.

28.6.5 Getter locationForKey

```
getter locationForKey ??@string inKey -> @location
```

Le *getter* `locationForKey` retourne un `@location` qui contient l'information de position de la clé `inKey` dans la table de premier niveau du receveur. Une erreur d'exécution est déclenchée si cette clé n'existe pas.

28.6.6 Getter overriddenMap

```
getter overriddenMap -> @T
```

Le *getter* `overriddenMap` retourne la table obtenue en amputant de la valeur du receveur la table de premier niveau. Si le receveur n'a pas de table surchargée, une erreur d'exécution est déclenchée.

28.7 Énumération

L'instruction `foreach` permet d'énumérer des objets de type `map`. Uniquement la table de premier niveau est énumérée. Par défaut, l'énumération s'effectue dans l'ordre croissant des clés. Pour énumérer dans l'ordre décroissant, utiliser le qualifier `>`.

À l'intérieur du coprs de la boucle, sont implicitement définies :
- la constante `lkey`, de type `@lstring`, qui a pour valeur la clé de l'entrée courante ;
- pour chaque attribut, une constante du type de l'attribut, et portant le nom de cet attribut, qui a pour valeur la valeur de cet attribut de l'entrée courante.

Par exemple :

```
@MaTable uneTable = {}
[!?uneTable insertKey ![@lstring new !"z" !here] !"world" !5]
[!?uneTable insertKey ![@lstring new !"a" !here] !"hello" !10]
for () in aMap do
  message lkey->string . " " . mPremier . " " . mSecond . "\n" ;
end
```

L'affichage produit est :

```
a hello 10
z world 5
```

Chapitre 29

Le type structure

Chapitre 30

Type `extern`

Un type `extern` est déclaré et spécifié en GALGAS, et implémenté par une classe C++. Ceci permet de définir des types qui seraient difficilement exprimables en GALGAS.

On va voir sur un exemple comment déclarer et implémenter :

— un type externe minimum ;
— un constructeur ;
— un *setter* ;
— une *méthode* ;
— un *getter* ;
— une *méthode* de classe.

L'exemple consiste à implémenter le type `@complex` qui représente les nombres complexes.

30.1 Type externe minimum

L'implémentation minimum ne sera pas opérationnelle, car elle ne comprendra pas de constructeur : on ne pourra donc pas instancier d'objet du type `@complex`. L'ajout de constructeur sera présenté à la section suivante. De même, cette implémentation minimum ne définira ni *setter*, ni *méthode*, ni *getter*.

30.1.1 Déclaration en GALGAS

La description minimum est la suivante :

```
extern @complex {
  "// No Predeclaration\n"
}{
  "  private : bool mIsValid ;\n"
  "  private : double mReal ;\n"
  "  private : double mImaginary ;\n"
}{
}
```

Cette description est divisée en trois parties, délimitées par les accolades `{` et `}`.

Première partie. Elle cite une séquence de chaînes de caractères, qui seront écrites telles quelles dans le fichier d'en-tête C++ engendré, juste avant la déclaration de la classe C++ ; on peut y placer là des pré-déclarations de classe, des inclusions de fichier, … Pour le type `@complex`, aucune pré-déclaration n'est

nécessaire, aussi on place un simple commentaire C++, de façon à le localiser dans le fichier d'en-tête C++ engendré.

30.1.2 Implémentation en C++

30.2 Constructeur

30.3 Setter

30.4 Méthode

30.5 Getter

30.6 Méthode de classe

Chapitre 31

Types prédéfinis

The following types are predefined, as particular structure types.

31.1 Types structure prédéfinis

Les types suivants sont des types structures prédéfinis, dont les champs sont des types de base.

31.1.1 Le type `@lbool`

The `@lbool` is predefined as :

```
struct @lbool {
  @bool bool
  @location location
}
```

31.1.2 Le type `@lchar`

The `@lchar` is predefined as :

```
struct @lchar {
  @char char
  @location location
}
```

31.1.3 Le type `@ldouble`

The `@ldouble` is predefined as :

```
struct @ldouble {
  @double double
  @location location
}
```

31.1.4 Le type `@lsint`

The `@lsint` is predefined as :

```
struct @lsint {
  @sint sint
  @location location
}
```

31.1.5 Le type `@lsint64`

The `@lsint64` is predefined as :

```
struct @lsint64 {
  @sint64 sint64
  @location location
}
```

31.1.6 Le type `@lstring`

The `@lstring` is predefined as :

```
struct @lstring {
  @string string
  @location location
}
```

31.1.7 Le type `@luint`

The `@luint` is predefined as :

```
struct @luint {
  @uint uint
  @location location
}
```

31.1.8 Le type `@luint64`

The `@luint64` is predefined as :

```
struct @luint64 {
  @uint64 uint64
  @location location
}
```

31.1.9 Le type `@range`

The `@range` is equivalent to the declaration :

```
struct @range {
  @uint start
  @uint length
}
```

Troisième partie

Sous-programmes

Chapitre 32

Sous-programmes

GALGAS définit les sous-programmes suivants :
- les *fonctions* (dans ce chapitre, section 33.1 page 140) ;
- les *procédures* (dans ce chapitre, section 33.2 page 141) ;
- les *méthodes* (section 34.2 page 143) ;
- les *getters* (section 34.1 page 143) ;
- les *setters* (section 34.3 page 144).

En GALGAS, *méthodes*, *getters* et *setters* s'appliquent sur un objet d'un type quelconque (qui n'est donc pas forcément un type *classe*). Pour les types définis par l'utilisateur, *méthodes*, *getters* et *setters* sont toujours déclarés en dehors de la déclaration du type auquel ils s'appliquent.

À chaque nature de sous-programme correspond une construction particulière pour l'appeler (tableau 32.1).

32.1 Arguments formels et paramètres effectifs

GALGAS distingue trois sortes d'arguments formels :
- en entrée (section 32.1.1 page 137) ;
- en entrée/sortie (section 32.1.2 page 137) ;
- en sortie (section 32.1.3 page 138).

32.1.1 Argument formel en entrée

Le tableau 32.2 liste les différentes formes d'un argument formel en entrée. Le paramètre effectif correspondant est une expression précédée par `!`.

32.1.2 Argument formel en entrée/sortie

Le tableau 32.3 liste les différentes formes d'un argument formel en entrée. Le paramètre effectif correspondant est une *cible* précédée par `!?`. Une *cible* est soit une variable, soit l'accès à un champ d'une variable de type `struct`.

Sous-programme	Construction	Référence
routine	Instruction d'appel de routine	section 37.10 page 166
fonction	Appel de fonction (dans une expression)	section 36.1.12 page 154
méthode	Instruction d'appel de méthode	section 37.18 page 179
getter	Appel de getter (dans une expression)	section 36.1.13 page 154
setter	Instruction d'appel de setter	section 37.20 page 180

***Tableau 32.1** – Constructions d'appel de sous programme*

Argument formel en entrée	Remarque	Paramètre effectif en sortie
`?selector: @T variable`	Variable (modifiable)	`!selector:expression`
`?selector: @T unused variable`	Variable inutilisée	
`?selector: let @T constante`	Constante	
`?selector: let @T unused constante`	Constante inutilisée	

***Tableau 32.2** – Argument formel en entrée, paramètre effectif en sortie*

32.1.3 Argument formel en sortie

Le tableau 32.4 liste l'unique forme d'un argument formel en sortie. Le compilateur vérifie que les instructions du sous-programme fixent une valeur à chaque argument formel en sortie.Le paramètre effectif correspondant est une *cible* précédée par `?`. Une *cible* est soit une variable, soit l'accès à un champ d'une variable de type `struct`.

32.2 Liste d'arguments formels en entrée, en sortie, ou en entrée/-sortie

32.3 Liste de paramètres effectifs en entrée

32.4 Sélecteur

Il est possible d'associer un nom avec chaque symbole `?`, `?!`, `!` et `!?`. Ce nom est appelé *sélecteur*.

Un sélecteur commence par une lettre et est suivi par zéro, un ou plusieurs lettres ou chiffres, et termine obligatoirement par deux points `:`. Aucun espace n'est autorisé. Par exemple :

`?valeur:`

`!par2:`

Les sélecteurs peuvent être utilisés à chaque fois que le symbole `?`, `?!`, `!` et `!?` apparaît. Quand un argument formel est déclaré avec un sélecteur, alors le paramètre effectif doit nommer le même sélecteur.

Par exemple, si on considère une routine déclarée par :

```
proc aireRectangle
  ?longueur: @uint inA
  ?largeur: @uint inA
  !aire: @uint outAire
{
  ...
```

Argument formel en entrée/sortie	Paramètre effectif en sortie/entrée	Remarque
`?!selector: @T variable`	`!?selector: cible`	
`?!selector: @T unused variable`	`!?selector:*`	Variable anonyme
	`!?` n `*`	n variables anonymes

***Tableau 32.3** – Argument formel en entrée/sortie, paramètre effectif en sortie/entrée*

Argument formel en sortie	Paramètre effectif en entrée	Remarque
`!selector:@T variable`	`?selector: variable`	Affectation
	`?selector: @T variable`	Déclaration et affectation de variable
	`?selector: let @T constante`	Déclaration et affectation
	`?selector: let constante`	Déclaration et affectation
	`?selector: *`	Variable anonyme
	`?` n `*`	n variables anonymes

***Tableau 32.4** – Argument formel en sortie, paramètre effectif en entrée*

```
}
```

Alors son appel s'exprimera par :

```
aireRectangle (!longueur: 2 !largeur: 3 ?aire: let aire)
```

Chapitre 33

Fonctions et procédures

GALGAS définit les sous-programmes suivants :
- les *fonctions* (dans ce chapitre, section 33.1 page 140) ;
- les *procédures* (dans ce chapitre, section 33.2 page 141) ;
- les *méthodes* (section 34.2 page 143) ;
- les *getters* (section 34.1 page 143) ;
- les *setters* (section 34.3 page 144).

33.1 Fonction

Une fonction GALGAS n'accepte que des arguments en entrée, et retourne une valeur. Elle est appelée dans une expression (section 36.1.12 page 154).

33.1.1 déclaration d'une fonction

```
private # Optionnel
func nom_fonction liste_arguments_entree -> @T var_resultat {
  liste_instructions
}
```

Une fonction est désignée par `nom_fonction`. Ce nom est unique dans un projet GALGAS. La liste des paramètres d'entrée peut être vide (section 32.3 page 138). La valeur renvoyée par l'exécution de la fonction est la valeur de `var_resultat` à l'issue de l'exécution de la `liste_instructions`. Aussi, l'exécution de la `liste_instructions` doit valuer `var_resultat`.

Exemple :

```
func produit ?@uint a ?@uint b -> @uint resultat {
  resultat = a * b
}
```

33.1.2 Fonction interne à un fichier

En préfixant la déclaration d'une fonction par `private`, on limite son appel aux expressions situées dans le même fichier que la déclaration.

33.1.3 Fonction %once

Une fonction sans argument accepte le qualificatif `%once` :

```
func %once masque-> @uint resultat {
  resultat = 1 << 16
}
```

Le qualificatif `%once` organise le cache du résultat : celui-ci est calculé lors du premier appel, est mémorisé internement, et est retourné directement lors des appels ultérieurs.

Une fonction `%once` peut être déclarée interne en la préfixant par **`private`**.

```
private func %once masque-> @uint resultat {
  resultat = 1 << 16
}
```

33.2 Procédure

Une procédure GALGAS accepte que des arguments en entrée, en sortie, en entrée/sortie. Elle est appelée dans une instruction (section 37.17 page 178).

33.2.1 Déclaration d'une procédure

```
private # Optionnel
proc nom_procedure liste_arguments {
  liste_instructions
}
```

Une procédure est désignée par `nom_procedure`. Ce nom est unique dans un projet GALGAS. La liste des paramètres en entrée, en sortie ou en entrée/sortie est décrite à la section 32.2 page 138).

Exemple :

```
proc produit ?@uint a ?@uint b !@uint resultat {
  resultat = a * b
}
```

33.2.2 Procédure interne à un fichier

En préfixant la déclaration d'une procédure par **`private`**, on limite son appel aux instructions situées dans le même fichier que la déclaration.

Chapitre 34

Extensions

Categories are the way for adding *getters*, *methods* and *setters* to any type. They are defined outside type declarations.

You can declare for any type :
- *category getters* ;
- *category methods* ;
- *category setters*.

Additional features are available for classes and are described in section 34.4 page 146.

A *category getter* is called in an expression. As expressions have no side-effect, a category getter cannot change current object's value.

A *category method* is called by the *method call instruction* (section 37.18 page 179). A category method cannot modify current object's value.

A *category setter* is called by the *setter call instruction* (section 37.20 page 180). A category setter can modify current object's value.

Within the category getter, method and setter instruction list, the `self` key word is allowed in any expression. It represents a copy of the current object. Of course, the current is lazily copied only when required.

The `self` key word is just a syntactic tag for representing a write or a read/write access to the current object. Using `self` is not allowed in category methods and category getters since they cannot modify the current object. Using `self` in category setters is explained in section 34.3 page 144.

A category getter, method and setter can be declared in :
- a *semantics* component ;
- a *syntax* component ;
- a *program* component.

A declared category getter, method and setter has a global scope, meaning it is available in the current component, and in any component that includes it directly or indirectly.

A type does not accept several category getters with the same name. During compilation of the project file, the project global checking mechanism detects such declarations and issues an error. Consequently, it is forbidden to declare a category getter with the same name than a predefined getter : the compiler issues an error on on a such declaration. The same rules apply on category methods and category setters.

However, it is safe to declare for a given type a category getter, a category method and a category setter with the same name. GALGAS compiler uses different naming spaces for them, and call syntax are different, so there is no ambiguity.

34.1 Category getter

A category getter is declared like a function, but its header names a type and a getter name. As a function, it accepts zero, one or more input and constant input formal parameters.

For example, the following code add a getter to the `@uint64` (page 90) that computes the square of its value :

```
getter @uint64 square -> @uint64 outResult {
  outResult = self * self
}
```

This getter is called like a predefined getter :

```
@uint64 v = 7L
log "Square of 7": [v square] # LOGGING Square of 7 : <@uint64:49>
```

You can add a category getter to a list :

```
getter @uintlist sum -> @uint outResult {
  outResult = 0
  for self do
    outResult = outResult + mValue
  }
}
```

For counting the number of element values greater than the value given in argument :

```
getter @uintlist countValuesGreaterThan
  ?let @uint inTestValue -> @uint outResult
{
  outResult = 0
  for self do
    if mValue > inTestValue then
      outResult ++
    end if
  }
}
```

When used with a struct or class type, current object attributes values can be read by naming the attribute in an expression. For example, the `@lstring` (page 134) has an attribute `string` whose type is `@string`. The following getter returns the value of this attribute, appended with the `" !"` string :

```
getter @lstring op -> @string outResult {
  outResult = string . " !"
}
```

34.2 Category method

A category method is declared like a routine, but its header names a type and a method name. As a routine, it accepts zero, one or more input, output, input/output constant input formal parameters.

For example, the following code add a method to the `@uint64` (page 90) that computes the square of its value :

```
method @uint64 square !@uint64 outResult {
  outResult = self * self
}
```

This getter is called like a predefined method :

```
@uint64 v
[7L square ?v]
log "Square of 7": v # LOGGING Square of 7 : <@uint64:49>
```

You can add a category method to a list :

```
method @uintlist sum !@uint outResult {
  outResult = 0
  for self do
    outResult = outResult + mValue
  }
}
```

For counting the number of element values greater than the value given in argument :

```
method @uintlist countValuesGreaterThan
  ?let @uint inTestValue
  !@uint outResult
{
  outResult = 0
  for self do
    if mValue > inTestValue then
      outResult ++
    end if
  }
}
```

When used with a struct or class type, current object attributes values can be read by naming the attribute in an expression. For example, the `@lstring` (page 134) has an attribute `string` whose type is `@string`. The following method returns the value of this attribute, appended with the `" !"` string :

```
method @lstring op !@string outResult {
  outResult = string . " !"
}
```

34.3 Category setter

A category method is declared like a routine, but its header names a type and a setter name. As a routine, it accepts zero, one or more output, input/output, input and constant input formal parameters. Unlike a category method, a category setter can change the value of the current object.

For structure and classes types, attributes can be read, written, read / written. For example :

```
setter @lstring appendInt ?let @uint inValue {
  string .= [inValue string]
}
```

The `self` key word is used as a syntactic tag for denoting a read/write or a write access on the current object. This key word is syntactically accepted in the following constructs :

1. the *setter call instruction* (section 37.20 page 180) ;
2. the *concat instruction* (section 37.6 page 162) ;
3. the *increment instruction* (section 37.12 page 174) ;
4. the *decrement instruction* (section 37.7 page 164) ;
5. the *assignment instruction* (section 37.4 page 161).

Example of using `self` in setter call instruction ; this setter prepends the square of argument value to the `@uint64list` value :

```
setter @uint64list prependSquare ?let @uint64 inValue {
  [!?self prependValue !inValue * inValue]
}
```

Example of using `self` in the append instruction ; this setter appends the square of argument value to the `@uintlist` value :

```
setter @uintlist appendSquare ?let @uint inValue {
  self += !inValue * inValue
}
```

This construct is valid only for types that handle the += operator.

Example of using `self` in the concat instruction ; this setter appends to the string all items of the `@stringlist` argument value :

```
setter @string concatList ?let @stringlist inList {
  for inList do
    self .= mValue
  }
}
```

This construct is valid only for types that handle the `.=` operator.

Example of using `self` in the increment instruction ; this setter increments the receiver's value :

```
setter @uint increment {
  self ++
}
```

This construct is valid only for types that handle the ++ operator, such as `@uint` (page 83), `@uint64` (page 90), `@sint` (page 69), `@sint64` (page 73).

Example of using `self` in the assignment instruction ; this setter removes all odd values of the receiver :

```
setter @uintlist removeOddValues {
  @uintlist listWithEvenValues [emptyList]
  for self do
    if (mValue & 1) == 0 then
      listWithEvenValues += !mValue
    end if
  }
  self = listWithEvenValues
}
```

This construct is valid only for types, but class types.

34.4 Categories and classes

Additional features are available only for classes ; in addition to category getters, methods and setters described in the above sections, you can declare :
- *abstract* category getters, methods, setters ;
- *overriding* category getters, methods, setters ;
- *overriding abstract* category getters, methods, setters.

Abstract category getters, methods, setters and *overriding abstract* category getters, methods, setters do not contain any instruction list : they act as *prototypes*.

Examples of *abstract* category getters, methods, setters declarations :

```
abstract getter @aType getterName
  ?anOtherType aParameter
  -> @resultType outResult
;

abstract method @aType methodName
  ?anOtherType aParameter
;

abstract setter @aType setterName
  ?anOtherType aParameter
;
```

Examples of *overriding* category getters, methods, setters declarations :

```
override getter @aType getterName
  ?anOtherType aParameter
  -> @resultType outResult
{
  instructions
}

override method @aType methodName
  ?anOtherType aParameter
{
  instructions
}

override setter @aType setterName
  ?anOtherType aParameter
{
  instructions
}
```

Examples of *overriding abstract* category getters, methods, setters declarations :

```
override abstract getter @aType getterName
  ?@anOtherType aParameter
  -> @resultType outResult
;

override abstract method @aType methodName
  ?anOtherType aParameter
```

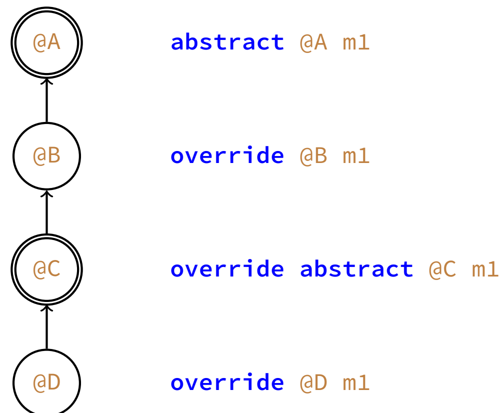

FIGURE 34.1 – *inheritance graph and categories*

```
;

override abstract setter @aType setterName
  ?anOtherType aParameter
;
```

Neither *abstract* category getters, methods, setters, neither *overriding abstract* category getters, methods, setters cannot be declared for concrete classes. Any kind of category getter, method, setter can be declared for abstract classes.

If an *abstract* category getter, method, setter, or an *overriding abstract* category getter, method, setter is declared for an abstract class, it should be also declared as *overriding* with the same name for every first concrete successor class.

A category getter, method, setter that has the same name as a category getter, method, setter declared for one of its super classes should be declared as *overriding*.

An abstract category getter, method, setter that has the same name as a category getter, method, setter declared for one of its super classes should be declared as *overriding abstract*.

The following example illustrates how theses rules should be applied. In the figure 34.1, four classes are shown. An arrow links a class to its super class. The `@A` and `@C` classes are abstract. `m1` is a name for any getter, method or setter.

`m1` is declared as **`abstract`** for the `@A` class. It is allowed since `@A` is abstract. Consequently, the concrete `@B` class should override `m1`. The `@C` class is also abstract, and `m1` can be declared as **`abstract`** for this class. But as it has been also declared for one of theses super class, it should also declared as **`override`**. As `@D` is concrete, `m1` should be declared for this class with **`override`** tag.

Quatrième partie

Filewrappers et templates

Chapitre 35

Filewrappers

Un *filewrapper* permet d'embarquer dans le code engendré une arborescence de fichiers. Comme on va le voir dans la section suivante, la déclaration d'un *filewrapper* désigne un répertoire, qui va être exploré au moment de la compilation GALGAS de façon à embarquer dans le code engendré la copie de certains fichiers. Ces fichiers peuvent être de trois sortes :

— des fichiers *texte* ; ils sont sélectionnés par leur extension : la déclaration d'un *filewrapper* liste toutes les extensions des fichiers texte embarqués ;

— des fichiers *binaires* ; de même, ils sont sélectionnés par leur extension, et la déclaration d'un *filewrapper* liste toutes les extensions des fichiers binaires embarqués ;

— des *templates*, qui sont sélectionnés par leur nom ; ils sont analysés lors de leur lecture.

L'exploration des fichiers embarqués peut s'effectuer soit de manière statique, soit dynamique à l'aide d'un objet de `@filewrapper` (page 63).

35.1 Déclararation d'un `filewrapper`

Un *filewrapper* peut être déclaré dans un composant *syntax*, *semantics* ou *program*. Sa déclaration est la suivante :

```
filewrapper nom in "chemin" {
 "extension_texte", ...
}{
 "extension_binaire", ...
}{
 declaration_de_templates
}
```

Où :

— `nom` est le nom, interne à GALGAS, donné au *filewrapper* ; ce nom doit être unique à chaque *filewrapper* ;

— `"chemin"` est le chemin du répertoire qui va être exploré récursivement au moment de la compilation ; c'est soit un chemin absolu (il commence par un `/`), soit un chemin relatif, par rapport au répertoire qui contient le fichier source qui déclare le *filewrapper*.

La déclaration est divisée en trois parties délimitées par des accolades `{ ... }` :

— la première partie (`"extension_texte", ...`) liste les extensions des fichiers texte qui sont embarqués ; à la compilation GALGAS, le répertoire désigné est exploré récursivement, et les fichiers dont l'extension est l'une des extensions citées sont embarqués, ainsi que leurs chemins relatifs ;

- la deuxième partie (`"extension_binaire", ...`) liste les extensions des fichiers binaires qui sont embarqués ; de même, à la compilation GALGAS, le répertoire désigné est exploré récursivement, et les fichiers dont l'extension est l'une des extensions citées sont embarqués, ainsi que leurs chemins relatifs ;
- la troisième et dernière partie (`declaration_de_templates`) contient les déclarations de *templates*.

Chacune de ces parties peut être vide si on ne veut pas embarquer de ficher ou ne définir aucun template.

Cinquième partie

Instructions et expressions

Chapitre 36

Expressions

D'une manière classique, une expression est constituée d'*opérandes* (section 36.1 page 152) et d'*opérateurs* (section 36.2 page 157). La priorité des opérateurs est définie dans le tableau 36.3 page 157.

36.1 Opérandes

36.1.1 Identificateur

36.1.2 `self`

Dans une expression, `self` représente une copie de l'objet courant. On ne peut donc utiliser `self` que dans une expression à l'intérieur d'une *méthode*, d'un *getter*, d'un *setter*, ou d'une extension (chapitre 34 page 142). Sont donc exclues les procédures et les fonctions.

`self` effectue un accès en lecture seule de l'objet courant.

Voici un exemple extrait de la section décrivant les *extensions de getter* (section 34.1 page 143) :

```
getter @uint64 square -> @uint64 outResult {
  outResult = self * self
}
```

36.1.3 Expression de conversion polymorphique inverse

La syntaxe de l'*expression de conversion polymorphique inverse* est : `expression as @T`. Elle permet de renvoyer la valeur de `expression` sous la forme d'un objet de type statique `@T`. À l'exécution, la conversion échoue si le type dynamique de l'`expression` n'est pas `@T` ou une de ses classes héritières ; une erreur sémantique est alors déclenchée, et l'expression renvoie un objet *non construit*.

Pour tester le type dynamique de l'expression avant d'effectuer la conversion, utiliser la construction décrite à la section 36.1.4 page 153. On peut aussi utiliser l'instruction `cast` (section 37.5 page 162).

36.1.4 Test du type dynamique d'une expression

L'opérande `expression is conversion @T` teste le type dynamique de `expression` vis à vis du type `@T` :

- si `conversion` est `==`, la valeur renvoyée est `true` si le type dynamique de l' `expression` est exactement `@T`, et `false` dans le cas contraire ;
- si `conversion` est `>=`, la valeur renvoyée est `true` si le type dynamique de l' `expression` est `@T` ou une de ses classes héritières, et `false` dans le cas contraire ;
- si `conversion` est `>`, la valeur renvoyée est `true` si le type dynamique de l' `expression` n'est pas `@T` mais une de ses classes héritières, et `false` dans le cas contraire.

Alliée à la construction précédente, elle permet de lancer une conversion uniquement si elle est possible :

```
if expression is == @B then
  let @B cst = expression as @B
  ...
elsif expression is >= @C then
  let @C cst = expression as @C
  ...
else
  message "conversion impossible"
end
```

36.1.5 Parenthèses

Les parenthèses `(` et `)` permettent de forcer le groupement d'opérandes.

36.1.6 `true` et `false`

`true` et `false` sont les constantes du type `@bool`.

36.1.7 Constante Chaîne de caractères

36.1.8 Constante caractère

36.1.9 Constante entière

Une constante entière est une séquence de chiffres décimaux, éventuellement séparés par le caractère de soulignement `_`, et terminé par un suffixe. Ce suffixe détermine le type de la constante :

- pas de suffixe : `@uint` ;
- suffixe `S` : `@sint` ;
- suffixe `L` : `@uint64` ;
- suffixe `LS` : `@sint64`.

36.1.10 Constante flottante

36.1.11 Expression `if`

36.1.12 Appel de fonction

36.1.13 Appel de getter

36.1.14 Constructeur

L'appel d'un constructeur instancie un nouvel objet. Sa syntaxe est :

```
@T.constructeur {!exp0 !exp1 ...}
```

Par exemple :

```
@lstring ls = @lstring.new {!"" !@location.here{}}
@stringlist str = @lstringlist.emptyList {}
```

Deux simplifications syntaxiques sont proposées :
— si la liste des arguments est vide, les accolades peuvent être omises ;
— si le type peut être inféré, il peut être omis.

Suppression des accolades

Si la liste des arguments est vide, les accolades peuvent être omises.

```
@lstring ls = @lstring.new {!"" !@location.here}
@stringlist str = @lstringlist.emptyList
```

Inférence du type

Si le type peut être inféré, il peut être omis (remarquer que ceci est valable aussi pour `@location.here` qui peut être simplifié en `.here`.

```
@lstring ls = .new {!"" !.here}
@stringlist str = .emptyList
```

36.1.15 Constructeur par défaut

Pour la plupart des types, un constructeur par défaut est implicitement défini (voir la liste précise tableau 36.1 page 156).

L'expression `@T.default` invoque le constructeur par défaut du type `@T` et renvoie un objet initialisé du type `@T`. Le type `@T` peut être inféré et l'appel du constructeur par défaut s'écrit simplement `.default`.

Intérêt du constructeur par défaut

L'intérêt du constructeur par défaut est qu'il allège l'écriture de l'initialisation des variables de certains types. Ce n'est pas une construction qui apporte de l'expressivité au langage (on peut très bien se passer d'appeler des constructeurs par défaut).

Pour un type comme `@uint`, écrire `@uint v = .default` est sémantiquement équivalent à écrire `@uint v = 0`. On voit que le constructeur par défaut présente peu d'utilité ici.

Par contre, si l'on a un type structure :

```
struct @T {
  @uneMap mMap
  @uneListe mList
  @stringlist mStringList
  @stringset mStringSet
}
```

Déclarer et initialiser une variable de ce type s'écrit :

```
@T variable = .new {
  !{}
  !{}
  !{}
  !{}
}
```

Avec le constructeur par défaut, cette instruction s'écrit simplement :

```
@T variable = .default
```

Pour une structure, comme on va le voir plus bas, le constructeur par défaut appelle le constructeur par défaut pour chaque champ ; le constructeur par défaut d'une `map` est équivalent à `emptyMap`, celui d'une `list` équivalent à `emptyList`, et celui d'un `@stringset` équivalent à `emptySet`.

Les constructeurs par défaut pour chaque type

Le tableau 36.1 précise par chaque type l'existence du constructeur par défaut.

Remarques :
— une classe abstraite ne possède pas de constructeur par défaut ;
— une classe concrète possède un constructeur par défaut si tous les attributs (ceux déclarés dans la classe et les super classes) en possèdent un ; la valeur par défaut est celle définie par l'appel du constructeur par défaut sur tous ces attributs ;
— une structure possède un constructeur par défaut si tous ces champs en possèdent un ; la valeur par défaut est celle définie par l'appel du constructeur par défaut sur tous les champs.

36.1.16 Valeur d'une option

Les options de la ligne de commande sont définies dans un composant `option` (chapitre 42 page 211). L'opérande *appel d'option* permet d'obtenir des informations sur une option.

Sa syntaxe est `[option nom_composant_option.nom_option nom_info]`, où :
— `nom_composant_option` est le nom du composant `option` qui déclare l'option ;
— `nom_option` est le nom donné à l'option lors de sa déclaration ;
— `nom_info` est le nom de l'information dont la valeur sera retournée par l'opérande.

Les informations qui peuvent être ainsi obtenues sont décrites dans le tableau 36.2.

Par exemple, si un composant `option` est déclaré comme suit :

Type	Constructeur par défaut
abstract class @T	*Pas de constructeur par défaut*
@bool	Initialisation à false
@application	*Pas de constructeur par défaut*
array @T	*Pas de constructeur par défaut*
@char	Initialisation au caractère NULL
class @T	Oui, si tous les attributs possèdent un constructeur par défaut
@data	Équivalent au constructeur emptyData
@double	Initialisation à 0.0
@filewrapper	*Pas de constructeur par défaut*
@function	*Pas de constructeur par défaut*
graph @T	Équivalent au constructeur emptyGraph
list @T	Équivalent au constructeur emptyList
map @T	Équivalent au constructeur emptyMap
listmap @T	Équivalent au constructeur emptyMap
@object	*Pas de constructeur par défaut*
@sint	Initialisation à 0S
@sint64	Initialisation à 0LS
sortedlist @T	Équivalent au constructeur emptySortedList
@string	Initialisation à chaîne vide ""
@stringset	Équivalent au constructeur emptySet
struct @T	Oui, si tous les attributs possèdent un constructeur par défaut
@type	*Pas de constructeur par défaut*
@uint	Initialisation à 0
@uint64	Initialisation à 0L

Tableau 36.1 *– Constructeur par défaut pour chaque type*

nom_info	Commentaire	Type de la valeur retournée
value	Valeur de l'option	@T (le type de l'option)
char	Caractère d'appel de l'option	@char
string	Chaîne d'appel de l'option	@string
comment	Description de l'option	@string

Tableau 36.2 *– Informations relatives à une option de la ligne de commande*

```
option mesOptions {
  @bool extractOption : 'S', "asm" -> "Extract assembly code"
}
```

Alors :
- [option mesOptions.extractOption value] renvoie un @bool qui vaut true si l'option a été activée, false dans le cas contraire ;
- [option mesOptions.extractOption char] renvoie un @char qui vaut 'S' ;
- [option mesOptions.extractOption string] renvoie un @string qui vaut "asm" ;
- [option mesOptions.extractOption comment] renvoie un @string qui vaut "Extract assembly code".

<table>
<tr><th>Priorité</th><th>Opérateur</th><th>Commentaire</th><th>Référence</th></tr>
<tr><td>0 (plus faible)</td><td><code>|</code></td><td>« ou » logique</td><td>section 36.2.2 page 157</td></tr>
<tr><td></td><td><code>||</code></td><td>« ou », évaluation en court-circuit</td><td>section 36.2.3 page 157</td></tr>
<tr><td></td><td><code>^</code></td><td>« ou exclusif » logique</td><td>section 36.2.2 page 157</td></tr>
<tr><td>1</td><td><code>&</code></td><td>« et » logique</td><td>section 36.2.2 page 157</td></tr>
<tr><td></td><td><code>&&</code></td><td>« et », évaluation en court-circuit</td><td>section 36.2.3 page 157</td></tr>
<tr><td>2</td><td><code>==</code>, <code>!=</code></td><td>Test d'identité</td><td>section 36.2.5 page 157</td></tr>
<tr><td></td><td><code><</code>, <code><=</code></td><td>Comparaison</td><td>section 36.2.5 page 157</td></tr>
<tr><td></td><td><code>></code>, <code>>=</code></td><td>Comparaison</td><td>section 36.2.5 page 157</td></tr>
<tr><td>3</td><td><code><<</code>, <code>>></code></td><td>Décalage</td><td>section 36.2.6 page 157</td></tr>
<tr><td></td><td><code>+</code></td><td>Addition, concaténation</td><td>section 36.2.7 page 158</td></tr>
<tr><td></td><td><code>-</code></td><td>Soustraction</td><td>section 36.2.7 page 158</td></tr>
<tr><td>4</td><td><code>*</code></td><td>Multiplication</td><td>section 36.2.7 page 158</td></tr>
<tr><td></td><td><code>/</code></td><td>Division</td><td>section 36.2.7 page 158</td></tr>
<tr><td></td><td><code>mod</code></td><td>Modulo</td><td>section 36.2.7 page 158</td></tr>
<tr><td>5</td><td><code>-</code></td><td>Négation arithmétique</td><td>section 36.2.7 page 158</td></tr>
<tr><td>6</td><td><code>not</code></td><td>Complémentation booléenne</td><td>section 36.2.2 page 157</td></tr>
<tr><td>7</td><td><code>~</code></td><td>Complémentation bit-à-bit</td><td>section 36.2.4 page 157</td></tr>
<tr><td>8 (plus forte)</td><td><code>.</code></td><td>Accès à un champ d'une structure</td><td>section 36.2.8 page 158</td></tr>
</table>

Tableau 36.3 *– Priorité des opérateurs*

36.2 Opérateurs

36.2.1 Priorité des opérateurs

La priorité des opérateurs est définie dans le tableau 36.3. Pour des opérateurs de même priorité, le groupement s'effectue de gauche à droite. Les parenthèses permettent de forcer l'ordre d'évaluation. Par exemple, `4 + 3 - 2 - 3` est équivalent à `((4 + 3) - 2) - 3`.

36.2.2 Logique

`|`, `^`, `&`, `not`

36.2.3 Logique, évaluation en court-circuit

`||` et `&&`

36.2.4 Complémentation bit-à-bit

`~`

36.2.5 Comparaison

36.2.6 Décalage

`<<` et `>>`

36.2.7 Arithmétique

`+`, `-`, `*`, `/`, `mod`.

`-` unaire.

36.2.8 Accès à un champ d'une structure

`.`

Chapitre 37

Instructions sémantiques

37.1 Rôle du point-virgule « ; »

Le point-virgule n'est pas un terminateur d'instruction. Il représente une instruction vide. Aussi, il peut être utilisé en nombre quelconque entre deux instructions consécutives. Ainsi, on peut écrire :

```
instruction1  instruction2
```

Ou encore :

```
instruction1 ; instruction2
```

```
instruction1 ;;;; instruction2
```

37.2 Instruction de déclaration de variable

Une déclaration de variable peut citer une `expression` qui lui fournit sa valeur initiale. Dans le cas constraire, la variable est *non construite* jusqu'à ce qu'une valeur lui soit affectée.

Les deux formes de déclaration sont donc :
- déclaration d'une variable *non construite* : « `var @type variable` » ;
- déclaration d'une variable *construite* : « `var @type variable = expression` ».

La forme générale de déclaration d'une variable *non construite* est :

```
var @type variable
```

La forme générale de déclaration d'une variable *construite* est :

```
var @type variable = expression
```

37.2.1 Déclaration « `var @type variable` »

On peut alléger l'écriture en ommettant le mot clé `var` :

```
@type variable
```

37.2.2 Déclaration « var @type variable = expression »

On peut alléger l'écriture en ommettant le mot clé `var` :

```
@type variable = expression
```

On peut omettre le type (mais il faut alors garder le mot-clé `var`), à la condition que l'expression puisse fournir l'information de type. Par exemple, dans l'écriture suivante :

```
var @string s = "Hello"
```

L'expression est une chaîne de caractères, dont le type est `@string`. On peut donc omettre l'annotation de type dans l'instruction :

```
var s = "Hello"
```

Prenons un autre exemple, celui de l'initialisation d'une liste :

```
var @stringlist sl = @stringlist {}
```

Le type est annoté de façon redondante, puisqu'il apparaît à la fois dans le membre de gauche et dans l'expression ; l'une des deux annotations peut être omise :

```
var sl = @stringlist {}
```

Ou :

```
var @stringlist sl = {}
```

Par contre, omettre les deux annotations ne permet pas de déduire le type de la variable : c'est donc une erreur détectée par le compilateur :

```
var sl = {} # Erreur : le type est indetermine
```

Un dernier exemple, avec un constructeur :

```
var @location sl = @location.here
```

On peut écrire :

```
var sl = @location.here
```

Ou :

```
var @location sl = .here
```

Mais en aucun cas :

```
var sl = .here # Erreur : le type est indetermine
```

37.3 Instruction de déclaration de constante

La forme générale de cette instruction est la suivante :

```
let @type nom = expression
```

Le mot clé `let` caractérise une déclaration de constante. L'annotation de type peut être omis si le type peut être déduit de l'expression, comme pour l'instruction de déclaration de variable (section 37.2.2 page 160). On peut donc reprendre les exemples de la section précédente :

```
let @string s = "Hello"
```

L'expression est une chaîne de caractères, dont le type est `@string`. On peut donc omettre l'annotation de type dans l'instruction :

```
let s = "Hello"
```

Prenons un autre exemple, celui de l'initialisation d'une liste :

```
let @stringlist sl = @stringlist {}
```

Le type est annoté de façon redondante, puisqu'il apparaît à la fois dans le membre de gauche et dans l'expression ; l'une des deux annotations peut être omise :

```
let sl = @stringlist {}
```

Ou :

```
let @stringlist sl = {}
```

Par contre, omettre les deux annotations ne permet pas de déduire le type de la constante : c'est donc une erreur détectée par le compilateur :

```
let sl = {} # Erreur : le type est indetermine
```

Un dernier exemple, avec un constructeur :

```
let @location sl = @location.here
```

On peut écrire :

```
let sl = @location.here
```

Ou :

```
let @location sl = .here
```

Mais en aucun cas :

```
let sl = .here # Erreur : le type est indetermine
```

37.4 L'instruction d'affectation

L'instruction d'affectation peut prendre plusieurs formes. La plus courante est :

```
variable = expression
```

Si `variable` est une instance de structure (chapitre 29 page 130), on peut directement en modifier un champ en utilisant l'opérateur `.` :

```
variable.champ = expression
```

Si `champ` est lui-même une structure, on peut accéder à l'un de ses champs (et ainsi de suite) :

```
variable.champ.champ1 = expression
```

37.5 L'instruction `cast`

L'instruction `cast` permet simplement d'exprimer de manière élégante une série de tests de conversions polymorphiques inverses. Sa syntaxe est :

```
cast expression
case conversion @T1 nom1 :
  ...
case conversion @T2 nom2 :
  ...
else
  ...
end
```

L'instruction accepte une ou plusieurs branches `case`, et zéro ou une branche `else`. `conversion` est soit `==`, soit `>=`. `nom1` et `nom2` sont des constantes dont le type est le type nommé dans la branche `case` qui la déclare, et dont la portée est limitée à cette branche `case`.

Lors de l'exécution, le type dynamique de `expression` est comparé successivement aux types (`@T1`, `@T2`) nommés dans les branches `case` ; dès que ce type dynamique est :

- exactement la classe `@T` (`conversion` est `==`),
- la classe `@T` ou de l'une de ses classes héritières (`conversion` est `>=`),
- une classe héritière de la classe `@T`, mais pas la classe `@T` (`conversion` est `>`),

alors la constante prend la valeur de `expression` et les instructions de la branche correspondante sont exécutées.

Si toutes les comparaisons échouent, la branche `else` est exécutée (si elle existe). La forme typique de cette instruction est donc :

```
cast expression
case >= @B v1 :
  ...
case >= @C v2 :
  ...
else
  message "conversion impossible"
end
```

Note : si la variable `nom1` ou `nom2` n'est pas utilisée dans la branche correspondante, une alerte est émise. Pour la supprimer, ne pas mentionner la variable en écrivant `case >= @T :`.

37.6 L'instruction d'ajout `+=`

Cette instruction permet d'ajouter un élément à une collection ; elle présente deux syntaxes, suivant que l'élément à ajouter est défini par :

- une expression : `cible += expression ;`
- une liste d'expressions : `cible += !expression1 ... !expressionN.`

La `cible` est une variable ou un champ de structure.

Cette instruction s'applique aux types suivants :

- `@string` (section 37.6.1 page 163) ;
- `@stringset` (section 37.6.2 page 163) ;

— `list @T` (section 37.6.3 page 163) ;
— `sortedlist @T` (section 37.6.4 page 164) ;
— `map @T` (section 37.6.5 page 164).

37.6.1 Instruction d'ajout et le type `@string`

Sous la forme `cible += expression`, l'instruction permet de concaténer des chaînes de caractères :

```
var s = "abc"
s += "def" # s vaut "abcdef"
```

La forme `cible += !expression1 ... !expressionN` n'est pas prise en compte pour le type `@string`.

37.6.2 Instruction d'ajout et le type `@stringset`

Sous la forme `cible += expression`, l'instruction permet de concaténer d'effectuer l'union des ensembles de chaînes :

```
var strset1 = @stringset {!"a", !"b"} # Valeur : "a", "b"
var strset2 = @stringset {!"b", !"c"} # Valeur : "b", "c"
strset1 += strset2 # strset1 vaut "a", "b", "c"
```

La forme `cible += !expression1 ... !expressionN` permet d'ajouter une chaîne à l'ensemble :

```
var strset1 = @stringset {!"a", !"b"} # Valeur : "a", "b"
strset1 += !"c" # strset1 vaut "a", "b", "c"
strset1 += !"b" # strset1 vaut "a", "b", "c"
```

37.6.3 Instruction d'ajout et les listes

Sous la forme `cible += expression`, l'instruction effectue une concaténation de listes : `cible` et `expression` doivent avoir le même type `list @T`, et l' `expression` est ajoutée à la fin de la `cible`.

```
var liste1 = @stringlist {!"a", !"b"}
var liste2 = @stringlist {!"c", !"d"}
liste1 += liste2 # liste1 vaut "a" "b" "c" "d"
```

Sous la forme `cible += !expression1 ... !expressionN`, l'instruction ajoute un élément à la fin de `cible`. L'élément est défini par la liste des valeurs de ses champs.

Avec la liste :

```
list @maListe {
  @uint mProperty1
  @string mProperty2
}
```

On a :

```
var liste = @maListe {}
liste += !2 !"a" # liste vaut contient un element 2, "a"
```

37.6.4 Instruction d'ajout et les listes triées

Sous la forme `cible += expression`, l'instruction effectue une concaténation de listes : `cible` et `expression` doivent avoir le même type `list @T`, et chaque élément de `expression` est ajouté à la `cible` en respectant l'ordre de tri.

Avec la liste triée :

```
sortedlist @maListeTriee {
  @uint mProperty1
  @string mProperty2
}{
  mProperty1 <
}
```

```
var liste1 = @maListeTriee {!3 !"a", !1 !"c"} # Valeur : (1, "c"), (3, "a")
var liste2 = @maListeTriee {!4 !"d", !2 !"b"} # Valeur : (2, "b"), (4, "d")
liste1 += liste2 # liste1 vaut (1, "c"), (2, "b"), (3, "a"), (4, "d")
```

Sous la forme `cible += !expression1 ... !expressionN`, l'instruction ajoute un élément à la fin de `cible`. L'élément est défini par la liste des valeurs de ses champs.

On a :

```
var liste = @maListeTriee {}
liste += !2 !"a" # Valeur : (2, "a")
liste += !1 !"b" # Valeur : (1, "b"), (2, "a")
liste += !3 !"c" # Valeur : (1, "b"), (2, "a"), (3, "c")
```

37.6.5 Instruction d'ajout et les tables

La forme `cible += expression` n'est pas prise en charge.

Sous la forme `cible += !expression1 ... !expressionN`, l'instruction ajoute un élément à la table `cible`. L'élément est défini par la clé et suivie de la liste des valeurs de ses champs.

Avec la table :

```
map @maTable {
  @uint mProperty1
  @string mProperty2
}
```

on a :

```
var table = @maTable {}
@lstring clef = ...
table += !clef !2 !"a"
```

37.7 Décrémentation --

L'instruction de décrémentation s'applique aux types `@sint` (page 69), `@sint64` (page 73), `@uint` (page 83) et `@uint64` (page 90) ; sa syntaxe est la suivante :

```
variable --
```

Les champs de structure peuvent être décrémentés :

```
variable.champ --
```

Une erreur d’exécution est déclenchée en cas de dépassement de capacité.

37.8 L’instruction drop

La syntaxe de l’instruction `drop` est la suivante :

```
drop variable, ...
```

Chaque variable nommée est placée dans l’état *non construit*.

37.9 L’instruction error

L’instruction `error` permet de signaler une erreur à l’utilisateur. Elle est constituée de trois champs séparés par un double-point (:) :

```
error localisation : message_erreur : variable1, ..., variableN
```

Le champ `localisation` signale à l’utilisateur la position de l’erreur dans le texte source. C’est donc une expression de type `@location`, ou d’un type possédant un *getter* sans argument nommé `location` et renvoyant un objet de type `@location` : c’est le cas de tous les types prédéfinis `@luint` (page 134), `@luint64` (page 134), `@lsint` (page 134), `@lsint64` (page 134), `@lbool` (page 133), `@lchar` (page 133) et `@lstring` (page 134). Le `message_erreur` est le message affiché à l’utilisateur : c’est donc une expression de type `@string`. Enfin, le troisième champ liste les variables `variable1`, ..., `variableN` qui ne peuvent être construites du fait de l’erreur. Si il n’y a pas de variable à citer, l’instruction s’écrit :

```
error localisation : message_erreur
```

Par exemple :

```
$identifier$ ??@lstring nom
...
error nom.location : message_erreur
```

Comme `nom` est de type `lstring` (voir ci-dessus), on peut simplement écrire :

```
$identifier$ ??@lstring nom
...
error nom : message_erreur
```

Lister des variables qui ne peuvent pas être construites est indispensable dans certains cas. Examinons le code suivant (qui ne compile pas) :

```
$identifier$ ??@lstring nom
@unType resultat
if ok then
  resultat = ...
else
  error nom : message_erreur
end # Erreur : 'resultat' est value par une branche
```

En effet, une des branches du `if` donne une valeur à `resultat`, et l'autre pas. Or, en cas d'erreur, on veut que `resultat` ne soit pas valué à l'exécution. On écrit alors le texte suivant (qui compile) :

```
$identifier$ ??@lstring nom
@unType resultat
if ok then
  resultat = ...
else
  error nom : message_erreur : resultat
end
```

Mentionner `resultat` à la fin de l'instruction `error` permet de faire croire au compilateur que `resultat` est valué.

37.10 L'appel de procédure

Cette instruction permet d'exécuter une procédure. Si par exemple celle-ci est définie par :

```
proc maRoutine !@uint a ?!@string b {
  ...
}
```

L'instruction d'appel de cette routine est (il y a plusieurs variantes possibles pour le premier paramètre qui est en entrée) :

```
@string x = ...
maRoutine (?@uint y !?x)
```

Note : les parenthèses sont obligatoires, même si il n'y a aucun argment.

La correspondance entre arguments formels et paramètres effectifs est décrite à la section 32.1 page 137.

37.11 L'instruction `for`

L'instruction `for` permet d'énumérer :

- une collection ;
- plusieurs collections de manière synchrone.

Pour énumérer une collection, la syntaxe est la suivante :

```
for enumeration_collection
while condition # Optionnel
before instructions_before  # Optionnel
do
  (nom_index) # Optionnel
  instructions_do
between instructions_between  # Optionnel
after instructions_after  # Optionnel
end
```

Énumérer plusieurs collections s'exprime en séparant les différentes énumérations par une virgule :

```
for enumeration_collection1, enumeration_collection2, ...
```

Énumération	Signification
`sens () in expression`	Utilisation de constantes définies implicitement qui représentent les champs de l'élément courant (section 37.11.3 page 167).
`sens () prefixe in expression`	Utilisation de constantes préfixées définies implicitement qui représentent les champs de l'élément courant (section 37.11.4 page 168).
`sens cst in expression`	Déclaration d'une constante représentant l'élément courant (section 37.11.5 page 169).
`sens (cst1 cst2 ...) in expression`	Déclaration de constantes représentant les champs de l'élément courant (section 37.11.6 page 171).

***Tableau 37.1** – Les quatre formes d'énumération de l'instruction* `for`

Type	Ordre d'énumération
`@data`	Ordre croissant des indices
`list @T`	Ordre croissant des indices
`map @T`	Ordre alphabétique des clés
`listmap @T`	Ordre alphabétique des clés
`sortedlist @T`	Ordre du tri
`@stringset`	Ordre alphabétique

***Tableau 37.2** – Types énumérables par l'instruction* `for`

```
while condition # Optionnel
before instructions_before  # Optionnel
do
  (nom_index) # Optionnel
  instructions_do
between instructions_between  # Optionnel
after instructions_after  # Optionnel
end
```

37.11.1 Les quatre formes d'une énumération

Le tableau 37.1 liste les quatre façons d'exprimer l'énumération `enumeration_collection`.

37.11.2 Types énumérables et ordre d'énumération

Les types pouvant être énumérés sont listés dans le tableau 37.2, ainsi que leur ordre d'énumération par défaut. Si le champ `sens` est vide, c'est l'ordre par défaut qui est adopté. Utiliser `>` fixe le sens inverse.

37.11.3 Énumération « `() in expression` »

Des constantes correspondants à chaque champ de l'élément courant sont implicitement déclarées (tableau 37.3).

Voici quelques exemples :

Type	Constantes implicitement déclarées
`@data`	`data`, de type `@uint`
`list @T`	À chaque champ de la liste, correspond une constante de même nom.
`map @T`	`lkey`, de type `@lstring`, qui représente la clé, et à chaque champ de la table, correspond une constante de même nom.
`listmap @T`	`key`, de type `@string`, qui représente la clé, et `mList`, qui représente la liste associée.
`sortedlist @T`	À chaque champ de la liste, correspond une constante de même nom.
`@stringset`	`key`, de type `@string`

***Tableau 37.3** – Constantes implicitement déclarées par «`() in expression`»*

```
@data d = ...
for () in d do
  log data
end
```

```
@stringset v = ...
for () in v do
  log key # Affichage des cles dans l'ordre alphabetique
end
```

Avec la liste :

```
list @maListe {
  @uint mProperty1
  @string mProperty2
}
```

On peut écrire :

```
@maListe lst = ...
for () in lst do
  log mProperty1, mProperty2
end
```

Avec la table :

```
map @maTable {
  @uint mProperty1
  @string mProperty2
}
```

On peut écrire :

```
@maTable tab = ...
for () in tab do
  log lkey, mProperty1, mProperty2
end
```

37.11.4 Énumération « `() prefixe in expression` »

Cette écriture est une extension de celle de la section précédente : `prefixe` est utilisé pour préfixer le nom des constantes implicitement déclarées. En reprenant les exemples de la section précédente :

```
@data d = ...
for () xyz_ in d do
  log xyz_data
end
```

```
@stringset v = ...
for () pre in v do
  log prekey # Affichage des cles dans l'ordre alphabetique
end
```

```
@maListe lst = ...
for () lst in lst do
  log lstmProperty1, lstmProperty2
end
```

```
@maTable tab = ...
for () tb_ in tab do
  log tb_lkey, tb_mProperty1, tb_mProperty2
end
```

Utiliser un préfixe permet de lever les collisions des noms des constantes implicites quand on énumère des collections ayant des champs de même nom :

```
@maListe v1 = ...
@maListe v2 = ...
for () in v1, () in v2 do # Erreur !
  ...
end
```

Le compilateur GALGAS déclenche une erreur, car il y a ambiguïté sur la signification de `mProperty1` et de `mProperty2` à l'intérieur de la boucle : désigne-t'elle l'élément courant de `v1` ou l'élément courant de `v2` ?

Pour lever l'ambiguïté, on complète l'instruction en précisant un préfixe pour l'une des deux listes (par exemple la seconde) :

```
@maListe v1 = ...
@maListe v2 = ...
for () in v1, () l2_ in v2 do
  log mProperty1 # Designe sans ambiguite le champ de la premiere liste
  log l2_mProperty1 # Designe sans ambiguite le champ de la seconde liste
end
```

37.11.5 Énumération « `cst in expression` »

Dans cette forme, une seule constante est déclarée (`cst`), et son type est donné par le tableau 37.4. Le type `@T-element` est implicitement déclaré avec la déclaration de la collection correspondante (liste, table), et est une structure : on accède donc à ses champs par l'opérateur `.`.

En reprenant les exemples de la section 37.11.3 page 167 :

```
@data d = ...
for v in d do
  log v
end
```

Type	Type implicite de la constante
`@data`	`@uint`
`list @T`	`@T-element`
`map @T`	`@T-element`
`listmap @T`	`@T-element`
`sortedlist @T`	`@T-element`
`@stringset`	`@string`

***Tableau 37.4** – Type de la constante dans* «`cst in expression`»

```
@stringset v = ...
for s in v do
  log s
end
```

```
@maListe lst = ...
for x in lst do
  log x.mProperty1, x.mProperty2
end
```

```
@maTable tab = ...
for entry in tab do
  log entry.lkey, entry.mProperty1, entry.mProperty2
end
```

Type explicite

On peut annoter le nom de la constante en la faisant précéder par un nom de type. Le compilateur GALGAS vérifie alors l'identité entre le type explicitement déclaré, et le type implicitement déduit du type de l'expression enumérée. Il est possible de déclarer explicitement le type de la constante en écrivant l'énumération sous la forme :

```
@unType cst in expression
```

Les exemples précédents deviennent alors :

```
@data d = ...
for @uint v in d do
  log v
end
```

```
@stringset v = ...
for @string s in v do
  log s
end
```

```
@maListe lst = ...
for @maListe-element x in lst do
  log x.mProperty1, x.mProperty2
end
```

Type	Constantes à déclarer
`@data`	*Ce type n’est pas pris en charge par cette forme.*
`list @T`	Une constante pour chaque champ, dans l’ordre de déclaration.
`map @T`	Une constante de type `@lstring`, qui représente la clé, suivi d’une constante pour chaque champ de la table, dans l’ordre de déclaration.
`listmap @T`	Une constante de type `@string`, qui représente la clé, et une constante qui représente la liste associée.
`sortedlist @T`	Une constante pour chaque champ, dans l’ordre de déclaration.
`@stringset`	*Ce type n’est pas pris en charge par cette forme.*

***Tableau 37.5 –** Constantes à déclarer pour «`(cst1 cst2 ...) in expression`»*

```
@maListe tab = ...
for @maListe-element entry in tab do
  log entry.lkey, entry.mProperty1, entry.mProperty2
end
```

37.11.6 Énumération « `(cst1 cst2 ...) in expression` »

Le tableau 37.5 liste pour chaque type le nombre et la signification des constantes qui doivent être déclarées.

Prenons comme exemple le type liste suivant :

```
list @maListe {
  @uint mProperty1
  @string mProperty2
  @char mProperty3
  @bool mProperty4
}
```

L’énumération s’écrit :

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (p1 p2 p3 p4) in uneListe do
  log p1, p2, p3, p4
end
```

Plusieurs variantes sont possibles, et sont décrites ci-après.

Type explicite

Il est possible d’annoter chaque constante en précisant son type.

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (@uint p1 @string p2 @char p3 @bool p4) in uneListe do
  log p1, p2, p3, p4
end
```

Le compilateur vérifie alors que le type cité est égal au type déduit du type de l’expression énumérée.

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (@uint p1 @string p2 @char p3 @bool p4) in uneListe do
```

```
  log p1, p2, p3, p4
end
```

Joker

Si certaines constantes ne sont pas utiles, on peut les remplacer par un joker (*). Le nom du type ne doit alors pas figurer.

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (@uint p1 * * @bool p4) in uneListe do
  log p1, p4
end
```

Plusieurs jokers peuvent être rassemblés en mentionnant leur nombre d'occurrences.

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (@uint p1 2* @bool p4) in uneListe do
  log p1, p4
end
```

Points de suspension

Trois points consécutifs (...) peuvent être utilisés pour signifier que les dernières constantes ne sont pas utiles.

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (@uint p1 ...) in uneListe do
  log p1
end
```

Et si aucune constante n'est utile, on écrit :

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (...) in uneListe do

end
```

37.11.7 Organigramme d'exécution

L'organigramme illustrant l'exécution de l'instruction `for` est donné à la figure 37.1. Si plusieurs collections sont énumérées, l'instructions se termine dès que la collection la moins nombreuse est complètement enumérée.

37.11.8 Champs optionnels

Plusieurs champs de l'instruction `for` sont optionnels.

`sens` . Ce champ peut prendre trois valeurs, et fixe l'ordre dans lequel les éléments sont énumérés :
- si le champ est vide, dans l'ordre indiqué par le tableau 37.2 ;
- `>` , dans l'ordre inverse à celui indiqué par le tableau 37.2.

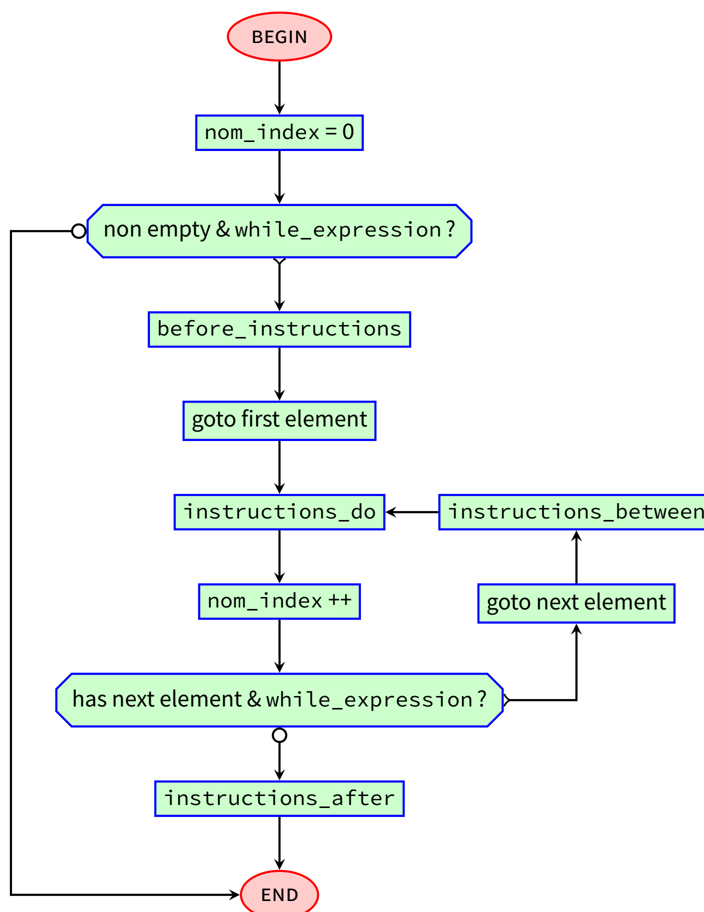

FIGURE 37.1 – *Organigramme d'exécution d'une instruction* for

(nom_index). Vous pouvez mentionner un identificateur entre parenthèses après le mot réservé **do**. Cet identificateur est le nom d'une constante qui a implicitement le type @uint et qui est initialisée à 0 avant toute exécution de la boucle, et incrémentée après chaque exécution des instructions_do, et avant l'exécution des instructions_between. Sa visibilité est la branche **do**.

while expression. L'énumération est exécutée tant que l'expression est vraie. L'absence de cette construction est équivalent à **while true** et permet d'énumérer toutes les valeurs.

before instructions_before. Ces instructions sont exécutées une seule fois, au début de l'exécution de l'instruction. Aucun accès aux objets énumérés n'est possible. Si l'énumération est vide, ces instructions ne sont pas exécutées.

between instructions_between. Ces instructions sont exécutées entre deux exécutions consécutives des instructions_do. Aucun accès aux objets énumérés n'est possible.

after instructions_after. Ces instructions sont exécutées une seule fois, à la fin de l'exécution de l'instruction. Aucun accès aux objets énumérés n'est possible. Si l'énumération est vide, ces instructions ne sont pas exécutées.

37.11.9 Modification de la collection

Au début de l'exécution de l'instruction `for`, les valeurs des collections enumérées sont capturées et mémorisées. L'énumération s'effectue sur ces valeurs mémorisées. Aussi, il est possible de modifier la collection en cours d'énumération sans que cela affecte l'exécution :

```
@stringlist v = {}
v += !"A"
v += !"B"
v += !"C"
log v # "A", "B", "C"
for s in v do
  v += !s
end for
log v # "A", "B", "C", "A", "B", "C"
```

37.12 Instruction d'incrémentation

L'instruction d'incrémentation s'applique aux types `@sint` (page 69), `@sint64` (page 73), `@uint` (page 83) et `@uint64` (page 90) ; sa syntaxe est la suivante :

```
variable ++
```

Les champs de structure peuvent être incrémentés :

```
variable.champ ++
```

Une erreur d'exécution est déclenchée en cas de dépassement de capacité.

37.13 L'instruction `if`

Dans sa forme la plus générale, l'instruction `if` a la syntaxe suivante :

```
if condition then
  instructions
elsif condition2 then
  instructions2
...
else
  instructions_else
end
```

Plus précisement, elle contient :
— zéro, une ou plusieurs branches `elsif` ;
— zéro ou une branche `else`.

Aucune branche `else` est équivalent à une branche `else` sans aucune instruction.

Les branches `elsif` sont simplement du sucre syntaxique : il est sémantiquement équivalent d'utiliser des instructions `if` imbriquées. Par exemple :

```
if condition then
  instructions
```

```
elsif condition2 then
  instructions2
else
  instructions_else
end
```

est équivalent à :

```
if condition then
  instructions
else
  if condition2 then
    instructions2
  else
    instructions_else
  end
end
```

Le langage permet que le type des `condition` et `condition2` soit différent du type `@bool` (page 48), sous certaines conditions. La règle complète est que le type des `condition` et `condition2` est :

— soit le type `@bool` ;
— soit un type *structure*, possèdant une propriété nommée `bool`, dont le type est `@bool` ;
— soit un type possédant un *getter* sans argument nommé `bool` et renvoyant une valeur de type `@bool`.

Voici un exemple illustrant le deuxième cas ; le type `@bool` (page 48) est une structure possèdant une propriété nommée `bool`, dont le type est `@bool`. Aussi, écrire :

```
@lbool variable = ...
if variable then
  instructions
else
  else_instructions
end
```

est équivalent à :

```
@lbool variable = ...
if variable.bool then
  instructions
else
  else_instructions
end
```

Pour illustrer le troisième cas, on prend l'exemple de la classe suivante :

```
class @myClass { ... }

getter @myClass bool -> @bool outResult { ... }
```

Ainsi, on peut écrire :

```
@myClass myObject = ...
if myObject then
  instructions
else
  else_instructions
end
```

Il est équivalent d'écrire :

```
@myClass myObject = ...
if [myObject bool] then
  instructions
else
  else_instructions
end
```

37.14 L'instruction grammar

L'instruction grammar permet d'exécuter l'analyse d'un texte par une grammaire. Le texte peut être contenu dans un fichier ou dans une chaîne de caractères.

Si le texte est contenu dans un fichier :

```
grammar
  nom_grammaire
  label_grammaire # Optionnel
  in expression_de_type_lstring # Chemin du fichier source
  liste_parametres
  traduction_dirigee_par_la_syntaxe # Optionnel
```

Si le texte est contenu dans une chaîne de caractères :

```
grammar
  nom_grammaire
  label_grammaire # Optionnel
  on expression_de_type_string # Texte source
  liste_parametres
  traduction_dirigee_par_la_syntaxe # Optionnel
```

Dans ces écritures :
- nom_grammaire est un identificateur nommant la grammaire, c'est le nom d'un composant grammaire du projet ;
- label_grammaire est optionnel, et permet d'exécuter une variante des règles de production (voir section 40.2 page 209) ;
- expression_de_type_lstring est une valeur de type @lstring, dont le champ string désigne un fichier source, par un chemin relatif ou absolu, et dont le champ location est la position de signalement d'erreur si le fichier source ne peut pas être lu ;
- expression_de_type_string est directement la chaîne source à analyser ;
- liste_parametres est une liste de paramètres effectifs (en entrée, sortie, ou sortie/entrée), en accord avec la liste des arguments formels de l'axiome de la grammaire ;
- traduction_dirigee_par_la_syntaxe est optionnel, et permet d'obtenir la chaîne source traduite lors d'une *traduction dirigée par la syntaxe* (voir section 4.3 page 22).

Prenons l'exemple d'un composant grammaire :

```
grammar maGrammaire "SLR" {
  syntax ...
   <start_symbol> ?!@declarationAST ioDeclarations
}
```

L'instruction grammaire s'écrit alors :

```
grammar maGrammaire in fichierSource !?ioDeclarations
```

Cette instruction est typiquement utilisé dans une règle d'analyse de fichier source (chapitre 43 page 213) :

```
case . "monExtension"
message "un fichier source"
??@lstring inSourceFile {
  var declaration = @declarationList {}
  grammar maGrammaire in inSourceFile !?ioDeclarations
  ...
}
```

37.15 L'instruction log

L'instruction `log` permet d'afficher le détail de la valeur d'une variable, d'une constante ou d'une expression :

— pour une variable ou une constante, `log nom` ;
— pour une expression, `log "message": expression` ;

Par exemple :

```
let x = 2
log x # Affiche LOGGING x: <@uint:2>
log "valeur" : x * 2 # Affiche LOGGING valeur: <@uint:4>
```

Plusieurs variables ou constantes peuvent être affichées par une même instruction `log`, en les séparant par une virgule :

```
let x = 2
log x, "valeur" : x * 2
```

37.16 L'instruction loop

L'instruction `loop` a la syntaxe suivante :

```
loop (variant_expression)
  instructions_1
while expression do
  instructions_2
end
```

Les `instructions_1` et `instructions_2` sont des listes d'instructions qui peuvent être vides.

Le `variant_expression` est une expression de type `@uint` qui assure que la boucle n'est pas sans fin : elle est calculée au début de l'exécution de l'instruction, et décrémentée après chaque itération. Si sa valeur atteint zéro, une erreur d'exécution est déclenchée.

L' `expression` est une expression de type `@bool` qui exprime la continuation de l'exécution de la boucle.

L'exécution de l'instruction `loop` est illustrée par l'organigramme de la figure 37.2.

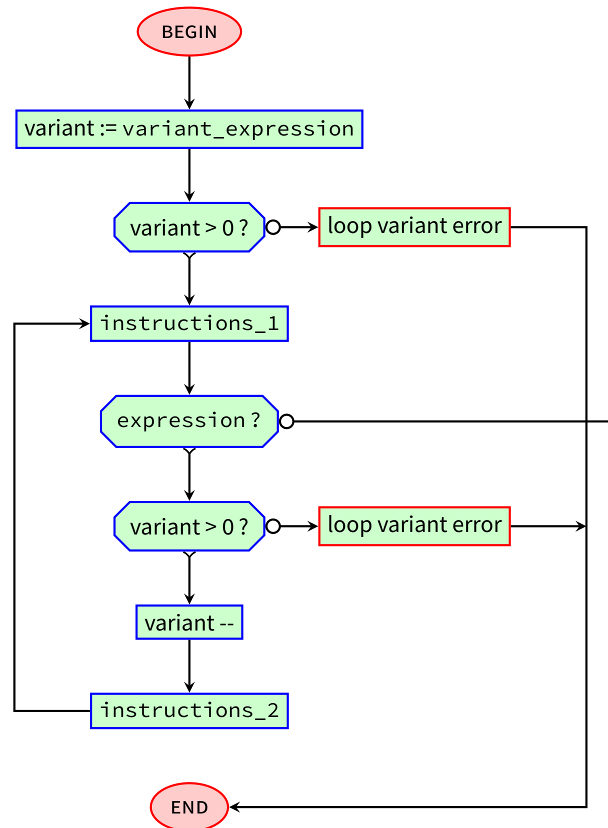

FIGURE 37.2 – *Organigramme d'exécution d'une instruction* `loop`

37.17 L'instruction d'appel de procédure

La syntaxe de l'appel d'une procédure est :

```
nom_procedure (liste_parametres_effectifs)
```

Les parenthèses sont obligatoires. La déclaration d'un procédure est présentée à la section 33.2 page 141.

`nom_procedure` est le nom de la procédure, et `liste_parametres_effectifs` est la liste des paramètres effectifs de l'appel, en accord avec l'en-tête de la procédure.

Par exemple, la procedure suivante :

```
proc produit ?@uint a ?@uint b !@uint resultat {
  resultat = a * b
}
```

Peut être appelée par :

```
produit (!2 !3 ??@uint resultat)
```

37.18 L'instruction d'appel de méthode

En GALGAS, une *méthode* est un sous-programme qui s'applique à un objet, et qui ne modifie pas cet objet. La syntaxe de l'appel d'une méthode est :

```
[expression nom_methode liste_parametres_effectifs]
```

La valeur d'`expression` est l'objet sur lequel la méthode est appelée, `nom_methode` est le nom de la méthode, et `liste_parametres_effectifs` est la liste des paramètres effectifs, en accord avec l'en-tête de la méthode.

Avec l'option `-T`, un fichier HTML qui contient les caractéristiques de tous les types d'un projet est engendré dans le répertoire `build/helpers`. Ainsi, les en-têtes de toutes les méthodes sont listés. Par exemple, pour le type `@string`, la méthode `writeToExecutableFileWhenDifferentContents` est présentée comme suit :

```
method writeToExecutableFileWhenDifferentContents
    ??@string inFilePath
    !@bool outFileModified
```

Aussi, cette méthode peut être appelée par :

```
@string contents = ...
@string filePath = ...
[contents writeToExecutableFileWhenDifferentContents
  !filePath
  ??@bool fileChanged
]
```

37.19 L'instruction d'appel de procédure de classe

En GALGAS, une *procédure de classe* est est une procédure définie dans un type. Au contraire d'une méthode, elle ne s'applique pas à un objet. La syntaxe de l'appel d'une procédure de classe est :

```
[@T nom_procedure liste_parametres_effectifs]
```

`nom_procedure` est le nom de la procédure, et `liste_parametres_effectifs` est la liste des paramètres effectifs, en accord avec l'en-tête de la procédure.

Avec l'option `-T`, un fichier HTML qui contient les caractéristiques de tous les types d'un projet est engendré dans le répertoire `build/helpers`. Ainsi, les en-têtes de toutes les méthodes sont listés. Par exemple, pour le type `@string`, la procédure de classe `deleteFile` est présentée comme suit :

```
proc @string deleteFile
    ??@string inFilePath
```

Aussi, elle peut être appelée par :

```
@string filePath = ...
[@string deleteFile !filePath]
```

37.20 L'instruction d'appel de *setter*

En GALGAS, un *setter* est un sous-programme qui s'applique à un objet, et qui peut modifier cet objet. La syntaxe de l'appel d'un *setter* est :

```
[!?cible nom_setter liste_parametres_effectifs]
```

Le délimiteur `!?` devant `cible` permet de distinguer syntaxiquement un appel de *setter* d'un appel de méthode. `cible` désigne l'objet sur lequel le *setter* est appelé, et peut être une variable, ou le champ d'une variable (`variable.champ`), ou le champ d'un champ d'une variable (`variable.champ.champ2`), ... `nom_setter` est le nom du *setter*, et `liste_parametres_effectifs` est la liste des paramètres effectifs, en accord avec l'en-tête du *setter*.

Avec l'option `-T`, un fichier HTML qui contient les caractéristiques de tous les types d'un projet est engendré dans le répertoire `build/helpers`. Ainsi, les en-têtes de toutes les *setters* sont listés. Par exemple, pour le type `@string`, le *setter* `setCharacterAtIndex` est présenté comme suit :

```
setter setCharacterAtIndex
         ??@char inChar
         ??@uint inIndex
```

Aussi, ce *setter* peut être appelée par :

```
@string s = ...
[!?s setCharacterAtIndex !'a' !4]
```

37.21 L'instruction `switch`

L'instruction `switch` est dédiée aux types énumérés. Elle présente la syntaxe suivante :

```
switch expression
case constante, constante, ... :
  liste_instructions
case constante, constante, ... :
  liste_instructions
...
end
```

Où `expression` est une expression d'un type énuméré. Toutes les constantes de ce type doivent être nommées dans les branches `case`, une et une seule fois.

Par exemple, avec la déclaration :

```
enum @feuTricolore {
  case vert
  case orange
  case rouge
}
```

On peut écrire :

```
@feuTricolore feu = ... ;

switch feu
case vert, orange:
```

```
  ...
case rouge :
  ...
end
```

Si des constantes déclarées dans l'énumération ont des valeurs associées, alors les branches `case` nommant ces constantes doivent adopter une syntaxe particulière.

En prenant pour exemple une constante possédant deux valeurs associées, la forme la plus générale est :

```
switch expression
case constante (@type1 nom1 @type2 nom2) :
...
end
```

`nom1` et `nom2` sont des constantes qui reçoivent les valeurs associées.

Si on n'est pas intéressé par une valeur, on peut substituer `*` au nom :

```
switch expression
case constante (@type1 nom1 @type2 *) :
...
end
```

De même, l'annotation du type est optionnel : les types `@type1`, `@type2` peuvent être déduits de la déclaration de la valeur associée :

```
switch expression
case constante (nom1 *) :
...
end
```

Ainsi, si l'on n'est pas intéressé par les valeurs associées, on peut écrire :

```
switch expression
case constante (* *) :
...
end
```

Enfin, on peut mentionner dans le même branche `case` plusieurs constantes déclarant des valeurs associées, à la condition que ces valeurs associées soient de même nombre et de même type. Par exemple :

```
enum @erreur {
  case ok
  case erreur1 (@string unMessage)
  case erreur2 (@string autreMessage)
}
```

On peut écrire l'instruction `switch` correspondante :

```
@erreur erreur = ... ;

switch erreur
case ok:
  ...
case erreur1, erreur2 (@string m) :
  ...
end
```

37.22 L'instruction warning

L'instruction warning permet de signaler une alerte à l'utilisateur. Elle est constituée de deux champs séparés par un double-point (:) :

```
warning localisation : message_alerte
```

Le champ localisation signale à l'utilisateur la position de l'erreur dans le texte source. C'est donc une expression de type @location, ou d'un type possédant un *getter* sans argument nommé location et renvoyant un objet de type @location : c'est le cas de tous les types prédéfinis @luint (page 134), @luint64 (page 134), @lsint (page 134), @lsint64 (page 134), @lbool (page 133), @lchar (page 133) et @lstring (page 134). Le message_alerte est le message affiché à l'utilisateur : c'est donc une expression de type @string.

Par exemple :

```
$identifier$ ??@lstring nom
...
warning nom.location : message_alerte
```

Comme nom est de type lstring (voir ci-dessus), on peut simplement écrire :

```
$identifier$ ??@lstring nom
...
warning nom : message_alerte
```

37.23 L'instruction with

L'instruction with permet d'associer un test de recherche dans une table et l'accès aux champs correspondants si succès. Elle peut prendre quatre formes différentes, suivi que l'on veuille modifier la table ou non, et suivant que l'on veut tolérer l'échec de la recherche ou non.

Accès en lecture, tolérance de l'échec de la recherche (section 37.23.1 page 183) :

```
with expression_cle prefixe_optionnel in expression_table
do
  liste_instructions_do
else
  liste_instructions_else # Optionnel
end
```

Accès en lecture, signalement d'erreur si échec de la recherche (section 37.23.2 page 184) :

```
with expression_cle prefixe_optionnel in expression_table
error message methode_recherche
do
  liste_instructions_do
end
```

Accès en lecture/écriture, tolérance de l'échec de la recherche (section 37.23.3 page 185) :

```
with expression_cle prefixe_optionnel in !?cible_table
do
  liste_instructions_do
else
```

```
  liste_instructions_else # Optionnel
end
```

Accès en lecture/écriture, signalement d'erreur si échec de la recherche (section 37.23.4 page 186) :

```
with expression_cle prefixe_optionnel in !?cible_table
error message methode_recherche
do
  liste_instructions_do
end
```

37.23.1 Accès en lecture tolérant l'échec de la recherche

```
with expression_cle prefixe_optionnel in expression_table
do
  liste_instructions_do
else
  liste_instructions_else # Optionnel
end
```

Où :

- `expression_cle` est une expression de type `@string` dont la valeur définit la clé ;
- `prefixe_optionnel` est soit vide, soit est constitué d'un double-point `:` suivi d'un identificateur qui préfixe les noms des champs dans la liste d'instructions `liste_instructions_do` ;
- `expression_table` est une expression qui désigne la table.

La clé désignée par la valeur de `expression_cle` est recherchée dans la table désignée par la valeur de `expression_table` :

- en cas d'échec, les instructions `liste_instructions_else` sont exécutées ;
- en cas de succès, ce sont les instructions `liste_instructions_do` qui sont exécutées ; les propriétés de l'élément recherché sont alors disponibles en lecture, sous leur nom éventuellement préfixé ; la clé est disponible sous le nom `lkey` éventuellement préfixé, et est de type `@lstring`.

Par exemple, on veut disposer d'une table possédant une propriété identifiant de manière unique la clé. On déclare :

```
map @maTable {
  @uint mIndex
  insert insertKey error message "entree deja presente"
}
```

Et on effectue des recherches / insertions de la façon suivante :

```
@uint idx ;
@lstring cle = ...
with cle.string in table do
  idx = mIndex
else
  idx = [table count]
  [!?table insertKey !cle !idx]
end
```

En utilisant un préfixe, le code devient :

```
@uint idx ;
@lstring cle = ...
with cle.string : xyz_ in table do
  idx = xyz_mIndex
else
  idx = [table count]
  [!?table insertKey !cle !idx]
end
```

37.23.2 Accès en lecture, signalement d'erreur si échec de la recherche

```
with expression_cle prefixe_optionnel in expression_table
error message methode_recherche
do
  liste_instructions_do
end
```

Où :

- `expression_cle` est une expression de type `@lstring` dont la valeur définit la clé ;
- `prefixe_optionnel` est soit vide, soit est constitué d'un double-point `:` suivi d'un identificateur qui préfixe les noms des champs dans la liste d'instructions `liste_instructions_do` ;
- `expression_table` est une expression qui désigne la table ;
- `methode_recherche` est une méthode de recherche de la table (c'est-à-dire déclarée par `search`) dont le message associé sera utilisé pour le signalement d'erreur.

La clé désignée par la valeur de `expression_cle` est recherchée dans la table désignée par la valeur de `expression_table` :

- en cas d'échec, une erreur est affichée, son message étant fourni par la méthode de recherche `methode_recherche`, et sa localisation par le champ `location` de l'`expression_cle` ;
- en cas de succès, ce sont les instructions `liste_instructions_do` qui sont exécutées ; les propriétés de l'élément recherché sont alors disponibles en lecture, sous leur nom éventuellement préfixé ; la clé est disponible sous le nom `lkey` éventuellement préfixé, et est de type `@lstring`.

Par exemple, on veut disposer d'une table qui implémente un *counted set*, c'est à dire que l'on associe à la clé son nombre de citations, et la doit avoir été entrée auparavant. On déclare :

```
map @maTable {
  @uint mOccurrenceCount
  search searchKey error message "entree %K absente"
}
```

Et on effectue des recherches de la façon suivante :

```
@lstring cle = ...
@uint occurenceCount
with cle in table error message searchKey do
  occurenceCount = mOccurrenceCount ++
end
```

En utilisant un préfixe, le code devient :

```
@lstring cle = ...
```

```
@uint occurenceCount
with cle : abc_ in table error message searchKey do
  occurenceCount = abc_mOccurrenceCount ++
end
```

37.23.3 Accès en lecture/écriture tolérant l'échec de la recherche

```
with expression_cle prefixe_optionnel in !?cible_table
do
  liste_instructions_do
else
  liste_instructions_else # Optionnel
end
```

Où :

- `expression_cle` est une expression de type `@string` dont la valeur définit la clé ;
- `prefixe_optionnel` est soit vide, soit est constitué d'un double-point `:` suivi d'un identificateur qui préfixe les noms des champs dans la liste d'instructions `liste_instructions_do` ;
- `cible_table` est de la forme `variable`, ou `variable.champ`, ou `variable.champ.champ2`, ... et désigne la table ; la cible est accédée en lecture/écriture.

La clé désignée par la valeur de `expression_cle` est recherchée dans la table désignée par la valeur de `cible_table` :

- en cas d'échec, les instructions `liste_instructions_else` sont exécutées ;
- en cas de succès, ce sont les instructions `liste_instructions_do` qui sont exécutées ; les propriétés de l'élément recherché sont alors disponibles en lecture / écriture, sous leur nom éventuellement préfixé ; la clé est disponible en lecture seule sous le nom `lkey` éventuellement préfixé, est de type `@lstring`.

Par exemple, on veut disposer d'une table qui implémente un *counted set*, c'est à dire que l'on associe à la clé son nombre de citations. On déclare :

```
map @maTable {
  @uint mOccurrenceCount
  insert insertKey error message "entree deja presente"
}
```

Et on effectue des recherches / insertions de la façon suivante :

```
@lstring cle = ...
with cle.string in !?table do
  mOccurrenceCount ++
else
  [!?table insertKey !cle !1]
end
```

En utilisant un préfixe, le code devient :

```
@lstring cle = ...
with cle.string : xyz_ in !?table do
  xyz_mOccurrenceCount ++
else
  [!?table insertKey !cle !1]
end
```

37.23.4 Accès en lecture/écriture, signalement d'erreur si échec de la recherche

```
with expression_cle prefixe_optionnel in !?cible_table
error message methode_recherche
do
  liste_instructions_do
end
```

Où :

- expression_cle est une expression de type @lstring dont la valeur définit la clé ;
- prefixe_optionnel est soit vide, soit est constitué d'un double-point : suivi d'un identificateur qui préfixe les noms des champs dans la liste d'instructions liste_instructions_do ;
- expression_table est une expression qui désigne la table ;
- methode_recherche est une méthode de recherche de la table (c'est-à-dire déclarée par search) dont le message associé sera utilisé pour le signalement d'erreur ;
- cible_table est de la forme variable, ou variable.champ, ou variable.champ.champ2, ... et désigne la table ; la cible est accédée en lecture/écriture.

La clé désignée par la valeur de expression_cle est recherchée dans la table désignée par la valeur de expression_table :

- en cas d'échec, une erreur est affichée, son message étant fourni par la méthode de recherche methode_recherche, et sa localisation par le champ location de l'expression_cle ;
- en cas de succès, ce sont les instructions liste_instructions_do qui sont exécutées ; les propriétés de l'élément recherché sont alors disponibles en lecture/écriture, sous leur nom éventuellement préfixé ; la clé est disponible en lecture seule sous le nom lkey éventuellement préfixé, et est de type @lstring.

Par exemple, on veut disposer d'une table qui implémente un *counted set*, c'est à dire que l'on associe à la clé son nombre de citations, et la clé doit avoir été entrée auparavant. On déclare :

```
map @maTable {
  @uint mOccurrenceCount
  search searchKey error message "entree %K absente"
}
```

Et on effectue des recherches de la façon suivante :

```
@lstring cle = ...
with cle in !?table error message searchKey do
  mOccurrenceCount ++
end
```

En utilisant un préfixe, le code devient :

```
@lstring cle = ...
with cle : abc_ in !?table error message searchKey do
  abc_mOccurrenceCount ++
end
```

Chapitre 38

Instructions syntaxiques

Les six instructions décrites dans ce chapitre ne peuvent être utilisées qu'à l'intérieur des règles de production, elles-mêmes obligatoirement placées dans un composant `syntax`.

38.1 Instruction d'appel de terminal

Cette instruction permet de vérifier l'occurrence d'un terminal. Sa syntaxe présente deux options :

```
$terminal$ parametres_entree
indexing nom_index # Optionnel
traduction_dirigee_par_la_syntaxe # Optionnel
```

Où :
- `$terminal$` est le nom du terminal à vérifier ; il doit être l'un des terminaux déclarés par le lexique importé par le composant syntaxique ;
- `parametres_entree` est une liste de zéro, un ou plusieurs paramètres effectifs en entrée, en accord avec la déclaration du `$terminal$` dans le lexique ; comment écrire une liste de paramètres en entrée est présenté à la section 32.3 page 138 ;
- `nom_index` est le nom d'un index, tel que déclaré dans le lexique ; cet index sert à peupler le menu contextual de cross référence en Cocoa (section 45.4 page 216) ;
- `traduction_dirigee_par_la_syntaxe` permet de préciser les options de *traduction dirigée par la syntaxe* ; celle-ci est présentée en détail au chapitre 4 page 21.

38.2 Instruction d’appel de non terminal

38.3 Instruction `select`

38.4 Instruction `repeat`

38.5 Instruction `parse`

38.6 Instruction `send`

Sixième partie

Déclarations

Chapitre 39

Le composant `lexique`

Le rôle d'un analyseur lexical est de grouper les caractères de la chaîne d'entrée en *symboles terminaux*, ou encore *terminaux*, en écartant les séparateurs comment les espaces ou les commentaires.

En GALGAS, un analyseur lexical est défini par un composant `lexique`. Les composants `syntax`, qui définissent un ensemble de règles de production, font référence à un composant `lexique`.

39.1 Définition d'un composant `lexique`

En GALGAS, un composant `lexique` a la structure suivante :

```
lexique nom {
  declarations
}
```

Le nom est le nom donné au composant ; il est utilisé pour référencer le composant `lexique` dans un composant `syntax`.

Dans un composant `lexique`, cinq types de déclarations sont définies :
- déclaration d'attribut lexical ;
- déclaration d'un symbole terminal ;
- déclaration d'une liste de symboles terminaux ;
- déclaration d'un message d'erreur lexical ;
- déclaration d'un style ;
- déclaration de règles d'analyse.

A //lexical attribute// carries the value associated with a terminal symbol : for example, the integer value of a literal integer constant, the string value of a character string constant, ...

In GALGAS, all terminal symbols must be declared either by a //single terminal symbol declaration//, either by a //list of terminal symbols declaration//. This defines the set of defined terminal symbols of your grammar.

Lexical error messages need also to be explicitly declared by //lexical error message declaration//.

A //style declaration// declares a style identifier, for defining automatic coloring in a text editor. Currently, coloring is only available for Mac OS X Cocoa applications.

The order of declarations is not significant, but any entity must be declared before being used.

==== Lexical Rules Overview ==== The //lexical rules// define the executable part of a lexical component. Every lexical rule define //matching strings// that are are tested against substring from current location in input string. A matching string has a one character or more.

39.2 Comment opère un analyseur lexical

You can consider the lexical analyzer as an autonomous thread which analyzes the input string and which sends the sequence of the terminal symbols to the parser. Of course, for efficiency, the lexical analyzer is actually a parser subroutine.

The flowchart of a GALGAS lexical analyzer execution is :

how_works_a_lexical_analyzer.png

When the input string is loaded from source file, a "NUL" character is appended as End Of String (eos) mark.

During execution, the lexical analyzer maintains a //current location// that designates the next character of the input string to be analyzed. Initially, current location points out the first character of the input string.

The lexical analyzer loops until the end of input string is reached. At the beginning of every loop, lexical attributes are reset to their default value.

Then, the first lexical rule matching expressions are tested against substring at current location in input string : * on match success, the first lexical rule is executed ; usually, this execution sends a terminal symbol to the parser ; however, in some cases as parsing a delimitor or a comment, no terminal symbol is sent ; * on match failure, the lexical analyzer tries to find a match with the second lexical rule, and so on.

If no lexical rule matches, the character at current location is tested against eos character. On match success, the lexical analyzer sends once a predefined terminal symbol (denoted by "

") to the parser, for telling it the end of input string is reached. On match failure, the //unknow character// lexical error is raised. The character at current location is discarded, that is the current location points out the next character of the input string.

39.3 Ambiguïtés lexicales

GALGAS does not currently check that the set of lexical rules is unambiguous. So, if the set is unambiguous, the rule order is not significant ; if two or more rules introduce an ambiguity, the first defined one is used.

39.4 Unexemple

This is very simple scanner, from "galgas/samples/notSLRgrammar.ggs" :

|"**lexique** my_scanner_for_not_SLR_grammar :
#— Identifiers

id
error **message** **rule** 'a'-> 'z'| 'A'-> 'Z' :
send
id
;

end **rule** ;
#— Delimitors
list delimitorsList error **message** **rule** **list** delimitorsList ;
#— Separators
rule '\1'-> **end** **rule** ;
end **lexique** ;"|

This `lexique` component defines the following set of terminal symbols : "
id
" (explicitly declared), "
$=$
" and "
$*$
" (declared by "delimitorsList" list.

The first rule sends the "
id
" terminal symbol each time a lower case or upper case character is found. The second rule names the "delimitorsList" list and sends the "
$=$
" or "
$*$
" terminal symbol each time the corresponding character is found. The last rule discards silently the space character and any control character.

Note that this scanner considers identifiers of only one character : "ab" is scanned as two consecutive identifiers.

===== Finding Sample Code =====

You can find examples of `lexique` components in : * "galgas/sample/alt_sample.ggs" file ; this is a very basic scanner that handles one-letter identifier and four delimitors ; * "galgas/sample/arith_expression.ggs" file (for scanning literal integers) ; * "galgas/sample/test_LR1_grammar.ggs" file gives an example of a small scanner for "toy" parser ; * "galgas/galgas/galgas_sources/galgas_scanner.ggs" file : this is the actual scanner of the GALGAS language, and scans identifiers, keywords, delimiters, literal integers, literal characters, literal character strings, galgas type names (the '@' character followed by a sequence of letters), comments, ...

39.5 Déclarations lexicales

39.5.1 Déclaration d'un symbole terminal

The //single terminal symbol declaration// declares a name used for naming a terminal symbol. This declaration just performs declaration, not scanning. For sending this terminal symbol to the parser, it must be named in a "send" lexical instruction within a lexical rule.

The declaration associates to the terminal symbol a possibly empty list of lexical attributes and a syntax error message (not a //lexical// error message), defined by a character string.

First example :

|"$literal_integer$ error **message**

This declaration names no lexical attribute. Consequently, when the lexical send instruction "send $literal_integer$;" will be called from a lexical rule, only the terminal symbol will be sent to the parser, but not the literal integer value. The parser has no way to get the actual value : all integer values share the same

terminal symbol. It is sufficient for a pure parser, however a real compiler needs the actual value.

Second example :

|”@uint unsignedValueAttribute ;
$literal_integer$!unsignedValueAttribute error **message**

In this declaration, the ”unsignedValueAttribute” attribute is named in the terminal symbol declaration. So, when the lexical send instruction ”send $literal_integer$;” will be called from a lexical rule, the terminal symbol will be sent to the parser together with the unsigned value of the ”unsignedValueAttribute” attribute, enabling the semantic instructions to catch it.

39.5.2 Déclaration d’une liste de symboles terminaux

The //list of terminal symbol declaration// associates to a name a list of terminal symbols with a generic syntax error message. It is typically used for declaring the keywords and the delimiters.

An example of key words declaration :

| ”**list** keywordList error **message**

The declared terminal symbols are : ”if”, ”$then$”, ”$else$”. The actual syntax error message is built from generic error message by replacing ”

An other example is a delimitor list declaration :

|”**list** delimitorList error **message**

Actual scanning of a delimitor is done by a ”**rule** **list**” lexical instruction.

39.5.3 Déclaration d’un attribut terminal

Lexical attributes carry values associated with terminal symbol. GALGAS handles string, unsigned, character, float lexical attributes. Every lexical attribute needs to be declared and its declaration names a GALGAS type name.

The following table summerizes the attributes features and type notation :

| ASCII String | ”@string” | ”| ASCII Character | ”@char” | ”| 32-bit Unsigned Integer | ”@uint” | ”0” | ”uint32” | | 32-bit Signed Integer | ”@sint” | ”0” | ”sint32” | | Float | ”@double” | ”0.0” | ”double” |

In GALGAS, type names are identifiers prefixed by a ”@” character.

An ”@string”, ”@char”, ”@uint”, ”@sint”, ”@double” lexical attribute carry a string, character, unsigned, signed, double value.

In a ”**syntax**” component, information that defines the location of the scanned terminal symbol in the input string is added to attribute value : so an ”@string” object in the lexique component corresponds to an ”@lstring” object in the syntax component. Location information is used by the parser and the semantic instructions for building syntax and semantic error messages that indicates //where// the error is located.

The //default value// is the one used at the beginning of every scanning loop for resetting lexical attribute.

The //corresponding C type// is useful if you want to write your own lexical actions (in C++). Please note that this correspondance is **only** available for lexical actions, and not for semantic action. The ”C_String” type is a C++ class that handles mutable character strings, without being worried about memory management. It is declared in the ”libpm/strings/C_string.h” file. The ”uint32” type is the 32-bit unsigned integer type, and the ”sint32” type is the 32-bit signed integer type.

39.5.4 Déclaration d'un message d'erreur lexicale

The //lexical error message declaration// associates a name to a string. These error messages are used in lexical actions, and define the message that are displayed when a lexical error occurs.

| "**message** decimalNumberTooLarge :

39.6 Règles lexicales

There are two kinds of //lexical rules// : - the //list lexical rule// ; - the //single lexical rule//.

39.6.1 Règle s'appuyant sur une liste

This is the simpliest form : it just names a previously defined list of terminal symbols ; for example :

|"**rule** **list** delimitorList ;"|

//Matching expressions// are the set of strings defined by the list. This rule tries to find a substring from input string at current location that matches a terminal symbol string defined in the list, sorted by decreasing length (so longest strings are tested first). On match success, //executing the rule// consists of sending the corresponding terminal symbol.

This kind of rule is typically used for scanning for a delimitor.

39.6.2 Simple règle

A //single lexical rule// has the following form :

|"**rule** //matching_expression// :
//lexical_instructions//
end **rule** ;"|

The //matching expression// defines a set of matching strings, that are tested against the substring from input string at current location. On match, the //lexical instructions// are executed.

A matching expression can be : - a one-character string (for example, "'a'" matches the "a" character) ; - an union of one-character strings, defined by a character subrange (for example, "'a'-> 'z'" matches a lower case letter) ; - a one or more characters string (for example, "- an union of above (for example : "'A'-> 'Z'| 'a'-> 'z'" matches a lower or upper case letter).

On match success, the current location is moved to designate the character after the matching string.

39.7 Instructions lexicales

39.7.1 Instruction lexicale `select`

The //lexical select instruction// is the following :

|"**select**
when //matching_expression_1_in_select// : //lexical_instructions_1//
when //matching_expression_2_in_select// : //lexical_instructions_2//
...

default //default_lexical_instructions//
end **select** ;”|

A //lexical select instruction// has one or more ”**when**” branches.

//matching expression_1_in_select//, //matching expression_2_in_select// conform to the defined above //matching_expression//.

This instruction tries to match the different //matching expressions// until a matching success is found. In such case, the corresponding //lexical instructions// are executed. If all matching fail, the //default lexical instructions// are executed.

39.7.2 Instruction lexicale `repeat`

The //lexical repeat instruction// is the following :

|”**repeat**
//lexical_instructions_0//
while //matching_expression_1_in_repeat// : //lexical_instructions_1//
while //matching_expression_2_in_repeat// : //lexical_instructions_2//
...
end **repeat** ;”|

A //lexical while instruction// has one or more ”**while**” branches.

//matching expression_1_in_repeat//, //matching expression_2_in_repeat// can be : - an expression conform to the defined above //matching_expression// ; - the ” //string//” construct : the match succeeds when the //string// **is not** the current string ; - the ” //string1//, //string2//, ...” construct : the match succeeds when neither of //string1//, //string2//, ... are the current string.

This instruction first executes the //lexical instructions 0//. Then, it tries to match the different //matching expressions// until a matching success is found. In such case, the corresponding //lexical instructions// are executed, then the instruction is executed again (from //lexical instructions 0//). If all matching fail, execution of this instruction is complete (excution goes on the next instruction).

39.7.3 Appel d’une action lexicale

The //lexical action call instruction// calls a C++ defined method for performing computation and checking on lexical attributes. Its syntax is the following :

|”lexical_action_name (parameter, ...) ;”|

or

|”lexical_action_name (parameter, ...) error message_name, ... ;”|

A lexical action is designated by its name. It accepts one or more parameters, and zero, one or more messages names.

A parameter is : - either a lexical attribute, - either a lexical function call ; - either the joker character ”'*'” that represents the character at current location.

A lexical action can be predefined or defined by the user. Predefined lexical actions are actually methods of ”C_Lexique” class (the generated scanner is a class that inherits from this class). User defined lexical actions must be implemented as methods of the generated scanner class.

**Note that no parameter type checking, no error message count checking is performed by GALGAS. ** A parameter type error or a message count error is detected at C++ compilation stage.

39.7.4 Appel d'une fonction lexicale

The //lexical function call// calls a C++ defined method for performing computation on lexical attributes. It can only appear as parameter of a lexical action call or a parameter of an other lexical function call. Its syntax is the following :

|"lexical_function_name (parameter, ...) ;"|

A lexical function is designated by its name. It accepts one or more parameters.

A lexical function parameter is : - either a lexical attribute, - either a lexical function call ; - either the joker character "'*'" that represents the character at current location.

A lexical function can be predefined or defined by the user. Predefined lexical actions are actually methods of "C_Lexique" class (the generated scanner is a class that inherits from this class). User defined lexical functions must be implemented as methods of the generated scanner class.

**Note that no parameter type checking is performed by GALGAS. ** A parameter type error is detected at C++ compilation stage.

39.7.5 Instruction lexicale `error`

The //lexical error instruction// raises a lexical error. Its syntax is :

|"error message_name ;"|

The //message name// is the name of a previously declared lexical error message.

39.7.6 Instruction lexicale `send`

The //lexical send instruction// sends a terminal symbol to the parser. It has several forms :

=== First Form ===

|"send terminal_symbol ;"|

This instruction sends inconditionnaly the //terminal symbol// to the parser.

=== Second Form ===

|"send search //attribute_name// in //lexical_list// default terminal_symbol ;"|

This instruction first search for //attribute name// value in the //lexical list//. If found, the corresponding terminal symbol is sent to the parser. If not found, the default //terminal symbol// is sent.

Several consecutive "search" are accepted, allowing sequential searching in different lists :

|"send search //attribute_name_1// in //lexical_list_1// default search //attribute_name_2// in //lexical_list_2// default terminal_symbol ;"|

39.7.7 Instruction lexicale `drop`

|Available in GALGAS 1.5.6 and later.|

The //lexical drop instruction// does not send any terminal symbol to the parser. It is only significant for lexical coloring (see [[#coloring_comments|coloring comments]]).

This instruction names a terminal symbol : |"**drop** //terminal_symbol// ;"|

39.7.8 Instruction lexicale `tag`

|Available in GALGAS 1.5.6 and later.|

This instruction declares a new //tag identifier//.

|"**tag** //tag_identifier// ;"|

A "**tag**" instruction records a location in the scanned file. The only way to use the declared tag identifier is the [[#lexical_rewind_instruction|lexical rewind instruction]].

39.7.9 Instruction lexicale `rewind`

|Available in GALGAS 1.5.6 and later.|

|"**rewind** //tag_identifier// send //terminal_symbol// ;"|

This instruction rewinds the scanned location from the tag identifier value, and sends the terminal symbol to the parser.

39.8 Routines lexicales prédéfinies

Lexical routine calls are instructions. Lexical function calls can appear as actual output parameters of routine calls and function calls. GALGAS predefines several lexical routines and several lexical functions (listed below).

A lexical routine accepts : * zero, one or more input/output or input formal arguments ; * zero, one or more error messages.

Running the `--print-predefined-lexical-actions` command line option lists all predefined routines and functions prototype.

39.8.1 Routine `codePointToUnicode`

```
codePointToUnicode !@string inCodePointString
                   ?!@string ioString
```

39.8.2 Routine `convertDecimalStringIntoSInt`

```
convertDecimalStringIntoSInt !@string inString
                             ?!@sint ioSignedNumber
                             error inNumberTooLargeError,
                                   inCharacterIsNotDecimalDigitError
```

39.8.3 Routine `convertDecimalStringIntoSInt64`

```
convertDecimalStringIntoSInt64 !@string inString
                               ?!@sint64 ioSignedNumber
                               error inNumberTooLargeError,
                                     inCharacterIsNotDecimalDigitError
```

39.8.4 Routine `convertDecimalStringIntoUInt`

```
convertDecimalStringIntoUInt !@string inString
                             ?!@uint ioUnsignedNumber
                             error inNumberTooLargeError,
                                   inCharacterIsNotDecimalDigitError
```

39.8.5 Routine convertDecimalStringIntoUInt64

```
convertDecimalStringIntoUInt64 !@string inString
                               ?!@uint64 ioUnsignedNumber
                               error inNumberTooLargeError,
                                     inCharacterIsNotDecimalDigitError
```

39.8.6 Routine convertHTMLSequenceToUnicodeCharacter

```
convertHTMLSequenceToUnicodeCharacter ?!@string inString
                                      ?!@char ioUnicodeCharacter
                                      error inUnassignedHTMLSequenceError
```

39.8.7 Routine convertHexStringIntoSInt

```
convertHexStringIntoSInt !@string inString
                         ?!@sint ioSignedNumber
                         error inNumberTooLargeError,
                               inCharacterIsNotHexDigitError
```

39.8.8 Routine convertHexStringIntoSInt64

```
convertHexStringIntoSInt64 !@string inString
                           ?!@sint64 ioSignedNumber
                           error inNumberTooLargeError,
                                 inCharacterIsNotHexDigitError
```

39.8.9 Routine convertHexStringIntoUInt

```
convertHexStringIntoUInt !@string inString
                         ?!@uint ioUnsignedNumber
                         error inNumberTooLargeError,
                               inCharacterIsNotHexDigitError
```

39.8.10 Routine convertHexStringIntoUInt64

```
convertHexStringIntoUInt64 !@string inString
                           ?!@uint64 ioUnsignedNumber
                           error inNumberTooLargeError,
                                 inCharacterIsNotHexDigitError
```

39.8.11 Routine convertStringToDouble

```
convertStringToDouble !@string inString
                      ?!@double ioDouble
                      error inConversionError
```

This action tries to convert the string value of the first argument into a double value. On success, the resulting double is set to the second argument. The conversion error message is displayed on conversion error.

39.8.12 Routine `convertUInt64ToSInt64`

```
convertUInt64ToSInt64 !@uint64 inUnsignedNumber
                      ?!@sint64 ioSignedNumber
                      error inNumberTooLargeError
```

If the unsigned value of the ”inUnsignedNumber” argument is greater than ”2⁶³-1”, the error is raised. Otherwise, the value is assigned to the ”ioSignedNumber” argument.

39.8.13 Routine `convertUIntToSInt`

```
convertUIntToSInt !@uint inUnsignedNumber
                  ?!@sint ioSignedNumber
                  error inNumberTooLargeError
```

If the unsigned value of the ”inUnsignedNumber” argument is greater than ”2³¹-1”, the error is raised. Otherwise, the value is assigned to the ”ioSignedNumber” argument.

39.8.14 Routine `convertUnsignedNumberToUnicodeChar`

```
convertUnsignedNumberToUnicodeChar ?!@uint inUnsignedNumber
                                   ?!@char ioUnicodeCharacter
                                   error inUnassignedUnicodeValueError
```

39.8.15 Routine `enterBinDigitIntoUInt`

```
enterBinDigitIntoUInt !@char inCharacter
                      ?!@uint ioUnsignedNumber
                      error inNumberTooLargeError,
                            inCharacterIsNotBinDigitError
```

39.8.16 Routine `enterBinDigitIntoUInt64`

```
enterBinDigitIntoUInt64 !@char inCharacter
                        ?!@uint64 ioUnsignedNumber
                        error inNumberTooLargeError,
                              inCharacterIsNotBinDigitError
```

39.8.17 Routine `enterCharacterIntoCharacter`

```
enterCharacterIntoCharacter ?!@char ioCharacter
                            !@char inCharacter
```

This routine performs ”ioCharacter := inCharacter” assignment.

39.8.18 Routine `enterCharacterIntoString`

```
enterCharacterIntoString ?!@string ioString
                         !@char inCharacter
```

Appends the character value of the second argument to the string value of the first argument. The resulting string is set to the first argument.

39.8.19 Routine enterDigitIntoASCIIcharacter

```
enterDigitIntoASCIIcharacter ?!@char ioASCIICharacter
                             !@char inDecimalDigitCharacter
                             error inErrorCodeGreaterThan255,
                                   inErrorNotDecimalDigitCharacter
```

Build an ASCII character from its decimal definition.

First, the character value of the ”inDecimalDigitCharacter” argument is tested to be a valid decimal digit, that is in one range ”['0', '9']”. On failure, the ”inErrorNotDecimalDigitCharacter” error message is displayed. On success, the unsigned value of the ”ioASCIICharacter” argument is multiplied by ten, and is added the decimal value corresponding to second argument. If the result is lower or equal to ”2⁸-1”, it is set to the ”ioASCIICharacter” argument. Otherwise, the ”inErrorCodeGreaterThan255” error is raised.

Note : this lexical action treats characters as unsigned values.

39.8.20 Routine enterDigitIntoUInt

```
enterDigitIntoUInt !@char inDecimalDigitCharacter
                   ?!@uint ioUnsignedNumber
                   error inNumberTooLargeError,
                         inCharacterIsNotDecimalDigitError
```

First, the value of ”inDecimalDigitCharacter” argument is tested to be in the range ”['0', '9']”. On failure, the ”inCharacterIsNotDecimalDigitError” error message is displayed. On success, the unsigned value of the first argument is multiplied by ten, and is added the decimal value corresponding to the ”ioUnsignedNumber” argument. If the result is lower or equal to ”2³²-1”, it is set to the ”ioUnsignedNumber” argument. Otherwise, the ”inNumberTooLargeError” error is raised.

39.8.21 Routine enterDigitIntoUInt64

```
enterDigitIntoUInt64 !@char inDecimalDigitCharacter
                     ?!@uint64 ioUnsignedNumber
                     error inNumberTooLargeError,
                           inCharacterIsNotDecimalDigitError
```

First, the value of ”inDecimalDigitCharacter” argument is tested to be in the range ”['0', '9']”. On failure, the ”inCharacterIsNotDecimalDigitError” error message is displayed. On success, the unsigned value of the first argument is multiplied by ten, and is added the decimal value corresponding to the ”ioUnsignedNumber” argument. If the result is lower or equal to ”2⁶⁴-1”, it is set to the ”ioUnsignedNumber” argument. Otherwise, the ”inNumberTooLargeError” error is raised.

39.8.22 Routine enterHexDigitIntoASCIIcharacter

```
enterHexDigitIntoASCIIcharacter ?!@char ioASCIICharacter
                                !@char inHexDigitCharacter
                                error inErrorCodeGreaterThan255,
                                      inErrorNotHexDigitCharacter
```

Build an ASCII character from its hexadecimal definition.

First, the character value of the ”inHexDigitCharacter” argument is tested to be a valid hexadecimal digit, that is in one of the ranges ”['0', '9']”, ”['a', 'f']”, ”['A', 'F']”. On failure, the ”inErrorNotHexDigitCharacter”

error message is displayed. On success, the unsigned value of the first argument is multiplied by sixteen, and is added the hexadecimal value corresponding to ”ioASCIICharacter” argument. If the result is lower or equal to ”2⁸-1”, it is set to the ”ioASCIICharacter” argument. Otherwise, the ”inErrorCodeGreaterThan255” error is raised.

Note : this lexical action treats characters as unsigned values.

39.8.23 Routine `enterHexDigitIntoUInt`

```
enterHexDigitIntoUInt !@char inHexDigitCharacter
                      ?!@uint ioUnsignedNumber
                      error inNumberTooLargeError,
                            inCharacterIsNotHexDigitError
```

First, the character value of the ”inHexDigitCharacter” argument is tested to be a valid hexadecimal digit, that in one of the the ranges ”['0', '9']”, ”['a', 'f']”, ”['A', 'F']”. On failure, the ”inCharacterIsNotHexDigitError” error message is displayed. On success, the unsigned value of the ”ioUnsignedNumber” argument is multiplied by sixteen, and is added the hexadecimal value corresponding to second argument. If the result is lower or equal to ”2³²-1”, it is set to the ”ioUnsignedNumber” argument. Otherwise, the first error is raised.

39.8.24 Routine `enterHexDigitIntoUInt64`

```
enterHexDigitIntoUInt64 !@char inHexDigitCharacter
                        ?!@uint64 ioUnsignedNumber
                        error inNumberTooLargeError,
                              inCharacterIsNotHexDigitError
```

First, the character value of the ”inHexDigitCharacter” argument is tested to be a valid hexadecimal digit, that in one of the the ranges ”['0', '9']”, ”['a', 'f']”, ”['A', 'F']”. On failure, the ”inCharacterIsNotHexDigitError” error message is displayed. On success, the unsigned value of the ”ioUnsignedNumber” argument is multiplied by sixteen, and is added the hexadecimal value corresponding to second argument. If the result is lower or equal to ”2⁶⁴-1”, it is set to the ”ioUnsignedNumber” argument. Otherwise, the first error is raised.

39.8.25 Routine `enterOctDigitIntoUInt`

```
enterOctDigitIntoUInt !@char inString
                      ?!@uint ioUnsignedNumber
                      error inNumberTooLargeError,
                            inCharacterIsNotOctDigitError
```

39.8.26 Routine `enterOctDigitIntoUInt64`

```
enterOctDigitIntoUInt64 !@char inString
                        ?!@uint64 ioUnsignedNumber
                        error inNumberTooLargeError,
                              inCharacterIsNotOctDigitError
```

39.8.27 Routine `multiplyUInt`

```
multiplyUInt !@uint inUnsignedNumber
             ?!@uint ioUnsignedNumber
             error inResultTooLargeError
```

Multiply the ”ioUnsignedNumber” value by ”inUnsignedNumber” value. Detection of overflow is performed.

39.8.28 Routine multiplyUInt64

```
multiplyUInt64 !@uint inUnsignedNumber
               ?!@uint64 ioUnsignedNumber
               error inResultTooLargeError
```

Multiply the ”ioUnsignedNumber” value by ”inUnsignedNumber” value. Detection of overflow is performed.

39.8.29 Routine negateSInt

```
negateSInt ?!@sint ioNumber
           error inNumberTooLargeError
```

39.8.30 Routine negateSInt64

```
negateSInt64 ?!@sint64 ioNumber
             error inNumberTooLargeError
```

39.8.31 Routine resetString

```
resetString ?!@string ioString
```

39.9 Fonctions lexicales prédéfinies

A lexical function accepts : * zero, one or more input formal arguments.

Running the `--print-predefined-lexical-actions` command line option lists all predefined routines and functions prototype.

39.9.1 Fonction toLower

```
toLower ?@char inCharacter -> @char
```

If the character value of the argument is an upper case letter, this function returns the corresponding lower case letter. Otherwise, it returns the unchanged character value of the argument.

39.9.2 Fonction toUpper

```
toUpper ?@char inCharacter -> @char
```

If the character value of the argument is an lower case letter, this function returns the corresponding upper case letter. Otherwise, it returns the unchanged character value of the argument.

39.10 Définir vos propres actions et fonctions lexicales

You can define your own lexical actions and functions in C++ and make them available to called by lexical action call instructions.

39.10.1 Où ?

You must define your lexical actions and functions as a method of the C++ class generated by compilation of the `lexique` component. You need to modify the generated code, adding method prototype declaration in class declaration.

So that the method declaration that you added is not deleted at the time of a future compilation, define it in user zone 2 of the generated header file. For more details, see [[generated_files |file generation process page]].

For implementing your method, you can insert it in user zone 2 of the generated implementation file (for more details, see [[generated_files |file generation process page]]). Alternatively, you can implement it in any other file, provided you include the needed header files.

39.10.2 Correspondance entre les appels d'actions GALGAS et C++

This table gives the correspondance between lexical argument types and C++ types. **Note this correspondance is only available for lexical arguments**.

|" ? @string" |"**const** C_String &"| |" ? ! @string" |"C_String &"| |" ? @char" |"**const** **char**"| |" ? ! @char" |"**char** &"| |" ? @uint" |"**const** uint32"| |" ? ! @uint" |"uint32 &"| |" ? @sint" |"**const** sint32"| |" ? ! @sint" |"sint32 &"| |" ? @double" |"**const** **double**"| |" ? ! @double" |"**double** &"|

" ?" means the formal argument has input passing mode : it cannot be modified by the lexical action. " ? !" means the formal argument has in/out passing mode : its value is got from the caller, can modified by the lexical action and is returned to the caller.

An error message argument corresponds to the C++ type "**const** **char** *".

In C++ generated code, the method call instruction generated by lexical action call names the lexical action name, prefixed by "scanner_routine_".

For example, consider the "convertStringToDouble" lexical action described below. This corresponds to the following method prototype :

"**void** scanner_routine_convertStringToDouble (**const** C_String &, **double** &, **const char** *) ;"
==== Defining Action and Function Prototype ====

The prototype must conform to the rules presented in the [[#Correspondance between Lexical Action Calls and C++ Called Methods|above]] section.

39.11 Exemples d'analyseurs lexicaux

39.11.1 Analyser des identificateurs

|"@string identifierString ;
$identifier$!identifierString error **message** **rule** **repeat**
enterCharacterIntoString ! ?identifierString !* ;
while **end** **repeat** ;
send $identifier$;
end **rule** ;"|

|"@string identifierString ;
$identifier$!identifierString error **message** **rule** **repeat**
enterCharacterIntoString ! ?identifierString !toLower (!*) ;

while **end** **repeat** ;
send $identifier$;
end **rule** ;"|

39.11.2 Analyser des identificateurs et des mots-clés

|"@string identifierString ;

$identifier$!identifierString error **message**
list keywordList error **message**
rule **repeat**
enterCharacterIntoString ! ?identifierString !* ;
while **end** **repeat** ;
send search identifierString in keywordList default $identifier$;
end **rule** ;"|

39.11.3 Analyser des délimiteurs

|"**list** galgasDelimitorsList **error message**
rule list galgasDelimitorsList ;"|

39.11.4 Analyser des séparateurs

|"**rule** **end rule** ;"|

39.11.5 Analyser des commentaires

|"**rule** '#' :
repeat
while **end repeat** ;
end rule ;"|

39.11.6 Analyser des entiers décimaux non signés

|"$unsigned_literal_integer$!ulongValue **error message** $signed_literal_integer$!longValue error **message**
message decimalNumberTooLarge :
message internalError :
rule enterDigitIntoUlong ! ?ulongValue !* error decimalNumberTooLarge, internalError ;
repeat
while enterDigitIntoUlong ! ?ulongValue !* error decimalNumberTooLarge, internalError ;
while **end repeat** ;
select
when convertUlongToLong ! ?longValue !ulongValue send $signed_literal_integer$;
default
send $unsigned_literal_integer$;
end select ;
end rule ;"|

39.11.7 Analyser des entiers hexadécimaux non signés

39.11.8 Analyser des constantes caractère

```
|"$literal_char$ ! charValue **error message**
**message** incorrectCharConstant :
**message** ASCIIcodeTooLargeError :
**rule** **select**
**when** **select**
**when** enterCharacterIntoCharacter ! ?charValue !**when** enterCharacterIntoCharacter ! ?charValue !**when**
enterCharacterIntoCharacter ! ?charValue !**when** enterCharacterIntoCharacter ! ?charValue !**when** en-
terCharacterIntoCharacter ! ?charValue !**when** enterCharacterIntoCharacter ! ?charValue !**when** en-
terCharacterIntoCharacter ! ?charValue !**when** enterCharacterIntoCharacter ! ?charValue !**when** **re-
peat**
enterHexDigitIntoASCIIcharacter ! ?charValue !* error ASCIIcodeTooLargeError, internalError ;
**while** **end repeat** ;
default
error incorrectCharConstant ;
**end select** ;
**when** enterCharacterIntoCharacter ! ?charValue !* ;
default
error incorrectCharConstant ;
**end select** ;
**select**
**when** send $literal_char$ ;
default
error incorrectCharConstant ;
**end select** ;
**end rule** ;"|
```

39.11.9 Analyser des constantes chaîne de caractères

39.11.10 Analyser des constantes flottantes

```
|"$literal_double$ !floatValue !tokenString **error message**
$.$ **error message**
**message** floatNumberConversionError :
**rule** **select**
**when** enterCharacterIntoString ! ?tokenString !enterCharacterIntoString ! ?tokenString !enterCharacterIntoString ! ?tokenS
**repeat**
**while** enterCharacterIntoString ! ?tokenString !* ;
**while** **end repeat** ;
convertStringToDouble !tokenString ! ?floatValue error floatNumberConversionError ;
send $literal_double$ ;
default
send $.$ ;
**end select** ;
**end rule** ;"|
```

39.12 *Back tracking* avec les instructions tag et rewind

|Available in GALGAS 1.5.6 and later.|

The ”**tag**” and ”**rewind**” instructions can be used for performing back tracking.

The first example is the way the non terminal symbols are scanned in GALGAS 1.5.6 (and later).

A non terminal is composed of a single ’<’ character, followed by a letter, zero, one or more letters, digits or underscore characters, is ended by a single ’>’ character. For example ”<abcdef>” is a valid non terminal. However, ”<abcdef >” is //not// a valid non terminal (because of the space before the final ’>’ character) : it is considered as a ’<’ delimitor, followed by the ”abcdef” identifier and by the ’>’ delimitor.

In the file ”galgas/galgas_sources/galgas_scanner.ggs”, the three delimitors befgging with a ’<’ character and the non terminal symbols are scanned by the following code :

”$<$ **error message** "the ’<’ delimitor" **style** delimitersStyle ;”
”””$non_terminal_symbol$! tokenString **error message** "a non terminal symbol <...>" **style** nonTerminalStyle ;”

”**rule** ’<’ :”
” **tag** onlyInfDelimiter ;”
” **select**”
” **when** ’=’ :”
” send ” **when** ’<’ :”
” send ” **when** ” **repeat**”
” enterCharacterIntoString ! ?tokenString !* ;”
” **while** ” **end repeat** ;”
” **select**”
” **when** ’>’ :”
” send $non_terminal_symbol$;”
” default”
” **rewind** onlyInfDelimiter send $<$;”
” **end select** ;”
” default”
” send $<$;”
” **end select** ;”
”**end rule** ;”

The ”**tag**” instruction records a scanning location. When the final ’>’ character is not found, the scanner is rewinded at the character following the ’<’ character, and the ”$<$” terminal is sent. On next scanning, an identifier (or a key word) will be found.

The second examples shows how to scan for integer constants, float constants, and array bounds in Pascal : * an integer constant is a (non empty) sequence of digits ; * a float constant is a (non empty) sequence of digits, following by a dot and a (possibly empty) sequence of digits ; * an array bound is an integer constant, followed by the ’..’ delimitor (two dots) and an integer constant.

The problem is that ”1..2” should not be scanned as a float constant, a single dot delimitor, and an integer constant.

This can be achieved by the following code :

”**rule** ” **repeat**”
” **while** ” **end repeat** ;”
” **tag** endOfIntegerConstant ;”
” **select**”

” **when** ” **select**”
” **when** ” **rewind** endOfIntegerConstant send $integer_constant$;”
” **when** ” **repeat**”
” **while** ” **end repeat** ;”
” send $float_constant$;”
” default”
” send $float_constant$;”
” **end select** ;”
” default”
” send $integer_constant$;”
” **end select** ;”
”**end rule** ;”

39.13 Ajouter la coloration lexicale (sur Mac uniquement)

With GALGAS, you can easily embbed your compiler in a GUI application (currently available only for Mac OS X). This application has a built-in text editor, from which you can modify, save and compile source file. With //style declarations//, you can add automatic coloring in the built-in text editor.

A //style declaration// associates a message to a style identifier. For example :

|”**style** keywordsStyle ->

The associated message is used in application preferences window as a comment of each color selection item.

A //style declaration// does not link a style identifier to any terminal symbol. You need to add this information to //single terminal symbol declaration// and //list of terminal symbols declaration// by naming the style identifier after the syntax error message :

|”$literal_integer$ error **message**

|”**list** delimitorList error **message**

39.13.1 Exemple : les styles de l’analyseur lexical GALGAS

As an example, you can take a look on GALGAS scanner, in ”galgas/galgas_sources/galgas_scanner.ggs” file. The style declarations are the following :

|”**style** keywordsStyle ->

You can search for the occurrence of style identifiers, to see how they are used.

In Cocoa GALGAS application, the Color tab of the Preferences window lists all style comments, each of them being associated to a ”NSColorWell” for color selection :

cocoa_galgas_color_styles.png

Note that no default color is defined in style declaration. Until you define yourself a color from Preference window, it defaults to black color.

39.13.2 Appliquer un style aux commentaires

|Available in GALGAS 1.5.6 and later.|

In GALGAS 1.5.6 and later, you can define a color for comments. Proceed as follows : - declare a new terminal symbol, for example ”$comment$” ; - declare a style for this new terminal symbol ; - when a comment is scanned, use the ”**drop**” instruction for naming the new terminal symbol (instead of the usual ”send” instruction).

The ”**drop**” instruction is only significant for syntax coloring.

For example, GALGAS comments are defined in ”galgas/galgas_sources/galgas_scanner.ggs” in this way :

”**style** commentStyle -> "Comments :" ;”
”...”
”$comment$ error **message** ”**rule** ” **repeat**”
” **while** ” **end repeat** ;”
” **drop** $comment$;”
”**end rule** ;”

Chapitre 40

Écrire un composant gammaire

40.1 GALGAS and Context-Free Grammars

40.2 Analyse en plusieurs phases

Chapitre 41

Graphic User Interface Component

Chapitre 42

Le composant `option`

Le composant **`option`** permet de définir des options qui sont appelables à partir de la ligne de commande. Dans le code, la valeur d'une option est obtenue à partir de l'opérande *appel d'une option*, décrit dans la section 36.1.16 page 155.

Voici l'exemple d'un composant **`option`** qui déclare une option (évidement, un composant **`option`** peut déclarer un nombre quelconque d'options) :

```
option nom_composant {
  @bool nom_option : 'S', "asm" -> "Extract assembly code"
}
```

42.1 Déclaration d'une option

La déclaration d'une option présente le syntaxe suivante :

```
  @T nom_option : caractere, chaine -> description
```

Les cinq champs qui définissent une option sont :
- `@T` : le type de l'option ; trois types sont autorisés : `@bool`, `@uint` et `@string` ;
- `nom_option` : c'est le nom, interne à GALGAS, qui permettra de désigner l'option dans l'*appel d'une option* (section 36.1.16 page 155) ;
- `caractere` : le caractère qui activera l'option dans la ligne de commande ; par exemple, en écrivant `'A'`, l'option sera activée par `-A` dans la ligne de commande ; si vous ne voulez pas d'activation par un caractère, écrivez `'\0'` ;
- `chaine` : la chaîne de caractères qui activera l'option dans la ligne de commande ; par exemple, en écrivant `"ABEDEF"`, l'option sera activée par `--ABCDEF` dans la ligne de commande ; si vous ne voulez pas d'activation par une chaîne, écrivez `""` ;
- `description` : une chaîne de caractère qui contient une description de l'option, qui sera affichée par l'option `--help` de votre compilateur.

42.2 Option booléenne

Le champ qui définit le type de l'option est `@bool` ; par exemple :

```
  @bool nom_option : 'S', "asm" -> "Extract assembly code"
```

Dans la ligne de commande, l'option est activée par -A ou --asm.

Par défaut, l'option n'est pas activée, et sa valeur associée est **false**. Quand l'option est activée dans la ligne de commande, sa valeur associée est **true**.

42.3 Option entière

Le champ qui définit le type de l'option est @uint ; par exemple :

```
@uint nom_option : 'M', "max-iterations-count" -> "Max of iteration count"
```

Dans la ligne de commande, l'option est activée par -N=xxx ou --max-iterations-count=xxx, où xxx est un nombre entier positif ou nul (et inférieur ou égal à $2^{32} - 1$).

Par défaut, l'option n'est pas activée, et sa valeur associée est 0. Quand l'option est activée dans la ligne de commande, sa valeur associée est la valeur xxx. Ainsi, l'option -N=0, comme l'option --max-iterations-count=0 n'a aucun effet.

42.4 Option chaîne de caractères

Le champ qui définit le type de l'option est @string ; par exemple :

```
@string nom_option : 'F', "file-name" -> "File name"
```

Dans la ligne de commande, l'option est activée par -F=abc ou --file-name=abc, où abc est une chaîne de caractères sans espaces. Si vous voulez entrer une chaîne de caractères qui comprend des espaces, écrivez : "-F=abc" ou "--file-name=abc".

Par défaut, l'option n'est pas activée, et sa valeur associée est la chaîne vide. Quand l'option est activée dans la ligne de commande, sa valeur associée est la chaîne abc. Ainsi, l'option -F=, comme l'option --file-name= n'a aucun effet.

Chapitre 43

Règle d’analyse de fichier source

Chapitre 44

Le composant `project`

Chapitre 45

Cocoa Features

45.1 Generated Cocoa Application

When a project component is compiled with a Xcode project target, a project_xcode directory is created. This directory contains :

— the Xcode project file ;
— a build.command file ;
— an Info.plist file ;
— an English.lproj directory ;
— an empty userResources directory.

The Info.plist, the English.lproj directory and the userResources directory are used by the Cocoa target of the Xcode project. The build.command file is a command file that builds the Xcode project.

All files you put in the userResources directory are added to the Cocoa target of the Xcode project when the GALGAS Project component is compiled. When the Cocoa target of the Xcode project is compiled, theses files are put in the Resources directory within the application bundle.

Adding files to the userResources directory is the way of customizing the Cocoa Application :

— adding icons to your Application (section 45.2 page 215) ;
— customizing syntax coloring (section 45.3 page 216).

45.2 Adding Icons to your Cocoa Application

For setting an icon for your Cocoa application and its documents, proceed as following.

1 Design icons, for example with names myApplicationIcon.icns and myDocumentIcon.icns.

2 Put the icons in the userResources directory.

3 Compile the GALGAS project : this updates the Xcode project, adding the icons files to its Cocoa target.

4 Under Xcode, edit the Info.plist file for assigning icon to the application and to the document (see http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Documents/Concepts/DocTypePList.html).

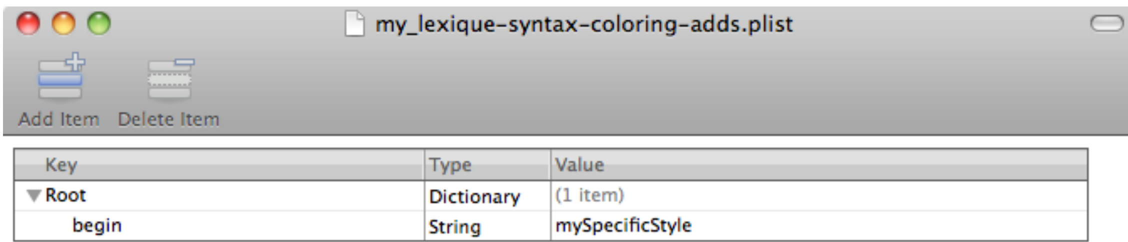

FIGURE 45.1 – *Example of a syntax coloring property list*

45.3 Customizing Syntax Coloring

This feature enables to set particular display attributes to a given list of tokens. This list is defined by a plist file located in the *Resources* directory of the application bundle.

1 Edit the GALGAS lexique component, and add one (or more) `style` entries. For example :

```
lexique my_lexique :
  ...
style mySpecificStyle -> "My Style" ;
  ...
end lexique ;
```

This new style's feature can be edited as other styles, by the Preferences setting of your Cocoa application.

2 Create a plist file with the *Property List Editor* application. This file should be named with the lexique component name, suffixed by `-syntax-coloring-adds` : so, for the example, the file name is `my_lexique-syntax-coloring-adds.plist`. Put this file in the *userResources* directory : so when the GALGAS project document is compiled, this file is added to the Cocoa Target of the Xcode project.

3 Edit the `my_lexique-syntax-coloring-adds.plist` with the *Property List Editor* application or Xcode. Add one entry for every custom syntax coloring case : the *key* is the terminal spelling, the *value* has the *String* type, and the specific style name. For example, the figure 45.1 shows the assignment of the terminal which spelling is `begin` by the `mySpecificStyle` style.

If your provides an undefined style name, you will be warned every time you open a document by a beep and a small explanation window.

45.4 Indexing your source files

You can configure your project for enabling cross-referencing entities with your Cocoa application. This has been done in GALGAS, providing such feature (figure 45.2). The contextual menu is displayed with a `cmd`-click.

For configuring your project, you will have to modify the lexique component, the syntax component, the grammar component and the program component. This section presents the six configuring steps.

1 **Lexique component header.** You need to change the lexique header, adding the « `indexing in` » declaration :

```
lexique my_lexique indexing in "INDEXING" :
  ...
```

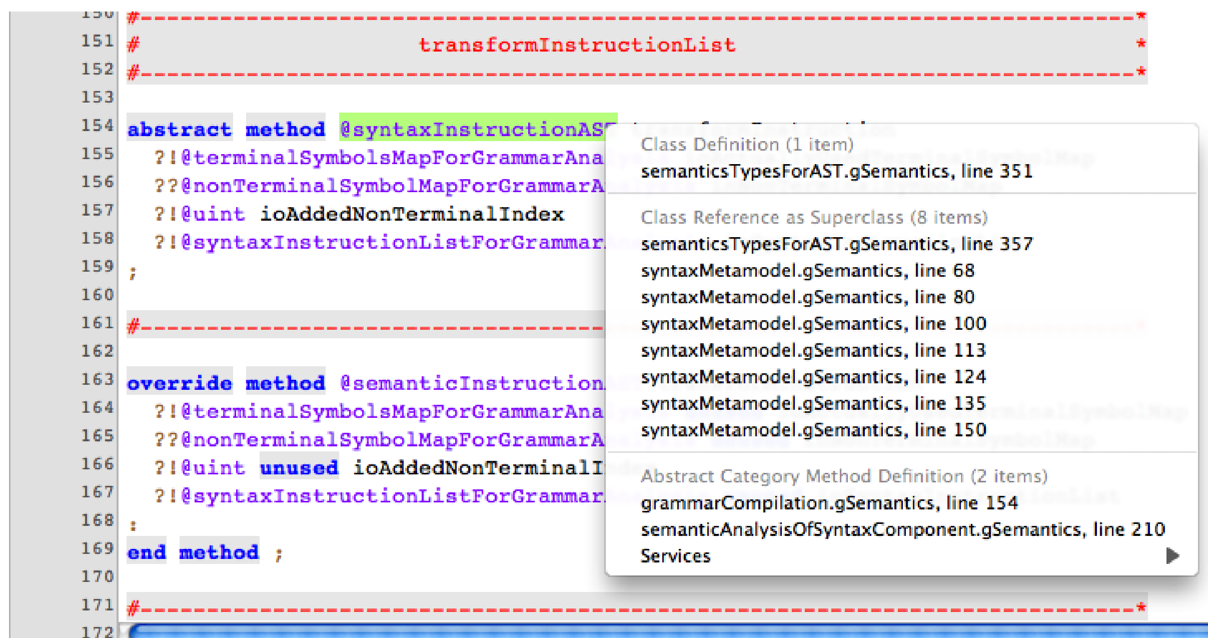

FIGURE 45.2 – *Indexing and cross-referencing in GALGAS Cocoa Application*

The « `"INDEXING"` » character string defines the name of the directory that will contains the indexing cache files. This directory is relative to the source file.

2 **Indexing classes declarations.** Add one or more indexing classes declarations in the lexique component body.

```
lexique my_lexique indexing in "INDEXING" :
  ...
indexing myIndexClass1 : "Awesome objects";
  ...
indexing myIndexClass2 : "Smart Kinds";
  ...
```

Each declaration consists of two parts :
— the index class name (for example `myIndexClass1`, `myIndexClass2`) is an identifier that assigns an internal name to the indexing declaration ;
— the index class title (for example `"Awesome objects"`, `"Smart Kinds"`) is a literal string that will be used as title in the Cocoa Application ; in the figure 45.2, the titles are *Class Definition*, *Class Reference as Superclass* and *Abstract Category Getter Definition*.

3 **Grammar component configuration.** Just prefix by « `indexing` » keyword the grammar header :

```
indexing grammar my_grammar ... :
  ...
```

4 **Program component configuration.** Insert the « `indexing with ...` » declaration after the « `message ...` » declaration in every program rule concerned by indexing :

```
  ...
when ...
message ...
indexing with my_grammar
```

```
??@lstring inSourceFile {
  ...
```

5 **Define indexing entries.** The indexing entries are defined within the rules of syntax components. The *terminal check* instruction is the unique way for definition, by naming an index class name :

```
syntax ... ("my_lexique.gLexique") :
  ...
rule ... :
  ...
  $identifier$ ? ... indexing myIndexClass1 ;
  ...
end rule ;
  ...
```

Any kind of terminal symbol accepts an « `indexing` » attribute : keywords, delimiters, literal string, integers, identifiers, ...

Several index class names can be named, using a comma as separator :

```
  ...
  $identifier$ ? ... indexing myIndexClass1, myIndexClass2 ;
  ...
```

6 **Compile and play.** Now, you can compile and run the Cocoa Application. With a `cmd`-click on an indexed terminal symbol, the contextual menu is displayed. You can delete the indexing directory at any moment, it will be rebuilt as needed.

Septième partie

Index

Index

- A -

accessibleStates
 @binaryset getter, 37
anyString
 @stringset getter, 79

- B -

binaryImage
 @double getter, 60
binarySetByTranslatingFromIndex
 @binaryset getter, 38
binarySetWithBit
 binaryset constructor, 31
binarySetWithEqualComparison
 @binaryset constructor, 31
binarySetWithEqualToConstant
 @binaryset constructor, 32
binarySetWithGreaterOrEqualComparison
 @binaryset constructor, 32
binarySetWithGreaterOrEqualToConstant
 @binaryset constructor, 33
binarySetWithLowerOrEqualComparison
 @binaryset constructor, 33
binarySetWithLowerOrEqualToConstant
 @binaryset constructor, 33
binarySetWithNotEqualComparison
 @binaryset constructor, 34
binarySetWithNotEqualToConstant
 @binaryset constructor, 34
binarySetWithPredicateString
 binaryset constructor, 34
binarySetWithStrictGreaterComparison
 @binaryset constructor, 35
binarySetWithStrictGreaterThanConstant
 @binaryset constructor, 36
binarySetWithStrictLowerComparison
 @binaryset constructor, 36
binarySetWithStrictLowerThanConstant
 @binaryset constructor, 36

- C -

column
 @location getter, 66
Component
 Graphic User Interface, 210
 Project, 214
compressedStringValueList
 @binaryset getter, 38
compressedValueCount
 @binaryset getter, 38
containsCharacter
 @string getter, 77
containsValue
 @binaryset getter, 38
cos
 @double getter, 60
count
 @stringset getter, 78
cString
 @bool getter, 48

- D -

double
 @sint getter, 70
 @sint64 getter, 74
 @uint getter, 84
 @uint64 getter, 92
doubleWithBinaryImage
 double constructor, 59

- E -

emptyBinarySet
 @binaryset constructor, 37
emptySet
 @stringset constructor, 78
equalTo
 @binaryset getter, 39
errorCount
 @uint constructor, 83
existOnBitIndex
 @binaryset getter, 39
existOnBitIndexAndBeyond
 @binaryset getter, 40
existsOnBitRange
 @binaryset getter, 39
Extensions, 142

- F -

Fonction
 Déclaration, 140
forAllOnBitIndex
 @binaryset getter, 40
forAllOnBitIndexAndBeyond
 @binaryset getter, 40

fullBinarySet
    @binaryset constructor, 37

- G -

greaterOrEqualTo
    @binaryset getter, 40

- H -

hasKey
    @stringset getter, 79
hexString
    @uint getter, 84
    @uint64 getter, 92

- I -

isalnum
    @char getter, 53
isalpha
    @char getter, 53
iscntrl
    @char getter, 53
isdigit
    @char getter, 53
isEmpty
    @binaryset getter, 41
isFull
    @binaryset getter, 41
isInRange
    @uint getter, 85
islower
    @char getter, 54
isNowhere
    @location getter, 66
isUnicodeCommand
    @char getter, 54
isUnicodeLetter
    @char getter, 54
isUnicodeMark
    @char getter, 54
isUnicodePunctuation
    @char getter, 55
isUnicodeSeparator
    @char getter, 55
isUnicodeSymbol
    @char getter, 55
isUnicodeValueAssigned
    @uint getter, 85
isupper
    @char getter, 55
ITE
    @binaryset getter, 41

- L -

line
    @location getter, 66
locationIndex
    @location getter, 66
locationString
    @location getter, 67
lowerOrEqualTo
    @binaryset getter, 41
lsbIndex
    @uint getter, 85

- M -

max
    @sint constructor, 69
    @sint64 constructor, 73
    @uint constructor, 83
    @uint64 constructor, 90
min
    @sint constructor, 69
    @sint64 constructor, 73

- N -

notEqualTo
    @binaryset getter, 42
nowhere
    @location constructor, 65

- O -

ocString
    @bool getter, 48

- P -

pi
    @double constructor, 59
predicateStringValue
    @binaryset getter, 42
Procédure
    Déclaration, 141

- R -

removeKey
    @stringset setter, 79
replacementCharacter
    @char constructor, 52

- S -

self, 152
setWithString
    stringset constructor, 78
significantBitCount
    @uint getter, 86
sin
    @double getter, 60
sint
    @bool getter, 48
    @double getter, 60
    @sint64 getter, 74
    @uint getter, 86
    @uint64 getter, 92
sint64
    @bool getter, 49
    @double getter, 60
    @sint getter, 70
    @uint getter, 86
    @uint64 getter, 93

Sous-programmes, 137
strictGreaterThan
@binaryset getter, 42
strictLowerThan
@binaryset getter, 43
string
@char getter, 56
@double getter, 61
@sint getter, 70
@sint64 getter, 74
@uint getter, 86
@uint64 getter, 93
stringValueList
@binaryset getter, 43
stringValueListWithNameList
@binaryset getter, 43
subString
@string getter, 77
swap01
@binaryset getter, 44
swap021
@binaryset getter, 43
swap102
@binaryset getter, 44
swap120
@binaryset getter, 44
swap201
@binaryset getter, 45
swap210
@binaryset getter, 45

- T -

tan
@double getter, 61
transitiveClosure
@binaryset getter, 45
Type
@binaryset, 31
@bool, 48
@char, 51
@data, 58
@double, 59
@filewrapper, 63
@lbool, 133
@lchar, 133
@ldouble, 133
@location, 65
@lsint, 134
@lsint64, 134
@lstring, 134
@luint, 134
@luint64, 134
@object, 68
@range, 134
@sint, 69
@sint64, 73
@string, 77
@stringset, 78
@type, 82
@uint, 83
@uint64, 90

- U -

uint
@bool getter, 49
@char getter, 56
@double getter, 61
@sint getter, 70
@sint64 getter, 74
@uint64 getter, 93
uint64
@bool getter, 49
@double getter, 61
@sint getter, 71
@sint64 getter, 75
@uint getter, 87
uint64BaseValueWithCompressedBitString
uint64 constructor, 90
uint64MaskWithCompressedBitString
uint64 constructor, 91
uint64ValueList
@binaryset getter, 46
uint64WithBitString
uint64 constructor, 91
uintSlice
@uint64 getter, 93
unicodeCharacterWithUnsigned
char constructor, 52
unicodeName
@char getter, 56
unicodeToLower
@char getter, 56
unicodeToUpper
@char getter, 57

- V -

valueCount
@binaryset getter, 46
valueWithMask
@uint constructor, 84

- W -

warningCount
@uint constructor, 84

- X -

xString
@uint getter, 87
@uint64 getter, 94