GALGAS

Version 2.5.3

Jean-Luc Béchennec
Mikaël Briday
Pierre Molinaro

24 octobre 2013

Table des matières

Liste des tableaux

Table des figures

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

Première partie

Le système de types

Chapitre 4

Présentation du système de types

4.1 Opérations définies pour tous les types

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

La plupart des types implémentent le constructeur par défaut `default` (voir section 4.2 page 28).

4.1.1 L'opérateur ==

```
operator @T == -> @bool ;
```

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

4.1.2 L'opérateur !=

```
operator @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 `==` .

4.1.3 Le reader `description`

```
reader @T description -> @string ;
```

Le *reader* `description` retourne une description textuelle du receveur, la même que celle affichée par l'instruction `log` (section 36.17 page 191).

4.1.4 Le reader `dynamicType`

```
reader @T dynamicType -> @type ;
```

Le *reader* `dynamicType` retourne un objet de type `@type`, dont la valeur représente le type dynamique du receveur (voir aussi la définition du type `@type` (page 97)).

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 extends @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
```

4.1.5 Le reader `object`

```
reader @T object -> @object ;
```

Le *reader* `object` retourne un objet de type `@object`. Une variable de type `@object` (page 77) peut encapsuler tout type de valeur.

4.2 Constructeur par défaut

Pour la plupart des types, un constructeur par défaut est implicitement défini (voir la définition précise section 4.2.4 page 29). Celui-ci est invoqué par le mot réservé `default`.

Le constructeur par défaut peut être utilisé dans deux constructions :
- la déclaration d'une variable ou d'une constante ;
- dans une expression.

4.2.1 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
  ![@uneMap emptyMap]
  ![@uneListe emptyList]
  ![@stringlist emptyList]
  ![@stringset emptySet]
] ;
```

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`.

4.2.2 Appel dans la déclaration d'une variable ou d'une constante

```
@T variable [default] ;
```

Ceci déclare une variable de type `@T` et l'initialise avec le constructeur par défaut. Pour une constante, la syntaxe est :

```
const @T constante [default] ;
```

4.2.3 Appel dans une expression

L'expression `[@T default]` invoque le constructeur par défaut du type `@T` et renvoie un objet initialisé du type `@T`.

4.2.4 Les constructeurs par défaut pour chaque type

Le tableau 4.1 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.

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 4.1 – *Constructeur par défaut pour chaque type*

Chapitre 5

Types de base

GALGAS predefines several types. This chapter presents all their features, including their constructors, readers, modifiers, methods, ...

Les types prédéfinis sont :

- type `@binaryset` (page 33), binary set objects (implemented with Binary Decision Diagrams) ;
- type `@bool` (page 53), boolean objects ;
- type `@char` (page 57), Unicode characters ;
- type `@double` (page 67), floating point numbers ;
- type `@filewrapper` (page 71), dont les objets permettent d'explorer les *filewrappers* ;
- type `@location` (page 73), whose value points out a location in a source file ;
- type `@object` (page 77), dont une instance peut encapsuler toute valeur ;
- type `@sint` (page 79), the 32-bit signed integers ;
- type `@sint64` (page 85), the 64-bit signed integers ;
- type `@string` (page 91), the Unicode string objects ;
- type `@stringset` (page 93), set of `string` objects ;
- type `@type` (page 97), dont une instance représente un type ;
- type `@uint` (page 99), the 32-bit unsigned integers ;
- type `@uint64` (page 107), the 64-bit unsigned integers.

Chapitre 6

Le type `@binaryset`

The `@binaryset` type encodes sets, binary relations, boolean mathematical expressions. It is implemented with Binary Decision Diagrams (BDD).

6.1 Constructors

6.1.1 `binarySetWithBit` Constructor

Returns an `@binaryset` object whose *inBitIndex* bit is constraint to one.

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

Availability: available in GALGAS 1.6.0 and later.

```
  constructor toto
```

Exemple :

```
@binaryset s [binarySetWithBit !2] ;
log s ; # displays <@binaryset: 1XX>
```

6.1.2 `binarySetWithEqualComparison` Constructor

Returns an `@binaryset` object that encodes a equality relation between two variables.

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the *a* == *b* relation, where *a* is encoded from bit index *inLeftFirstIndex* to bit index *inLeftFirstIndex* + *inBitCount* - *1*, and *b* is encoded from bit index *inRightFirstIndex* to *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

```
@binaryset s [binarySetWithEqualComparison !0 !2 !3] ;
log s; # displays <@binaryset: 00x00, 01X01, 10X10, 11X11>
```

6.1.3 `binarySetWithEqualToConstant` Constructor

Returns an `@binaryset` object that encodes a equality relation between a variable and a constant.

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the *a* == *cst* relation, where *a* is encoded from bit index *inBitIndex* to bit index *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by the *inConstant* argument.

Exemple :

```
@binaryset s [binarySetWithEqualToConstant !0 !6 !23L] ;
log s; # displays <@binaryset: 10111>
```

6.1.4 `binarySetWithGreaterOrEqualComparison` Constructor

Returns an `@binaryset` object that encodes a greater or equal relation between two variables.

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the *a* >= *b* relation, where *a* is encoded from bit index *inLeftFirstIndex* to bit index *inLeftFirstIndex* + *inBitCount* - *1*,

and *b* is encoded from bit index *inRightFirstIndex* to *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

```
@binaryset s [binarySetWithGreaterOrEqualComparison !0 !2 !3] ;
log s; # displays <@binaryset: 00XXX, 01X01, 01X1X, 10X1X, 11X11>
```

6.1.5 `binarySetWithGreaterOrEqualToConstant` Constructor

Returns an `@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 ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the *a* >= *cst* relation, where *a* is encoded from bit index *inBitIndex* to bit index *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by the *inConstant* argument.

6.1.6 `binarySetWithLowerOrEqualComparison` Constructor

Returns an `@binaryset` object that encodes a lower or equal relation between two variables.

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the *a* <= *b* relation, where *a* is encoded from bit index *inLeftFirstIndex* to bit index *inLeftFirstIndex* + *inBitCount* - *1*, and *b* is encoded from bit index *inRightFirstIndex* to *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

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

6.1.7 binarySetWithLowerOrEqualToConstant Constructor

Returns an `@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 ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the *a* <= *cst* relation, where *a* is encoded from bit index *inBitIndex* to bit index *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by the *inConstant* argument.

6.1.8 binarySetWithNotEqualComparison Constructor

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

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the *a* != *b* relation, where *a* is encoded from bit index *inLeftFirstIndex* to bit index *inLeftFirstIndex* + *inBitCount* - *1*, and *b* is encoded from bit index *inRightFirstIndex* to *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

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

6.1.9 binarySetWithNotEqualToConstant Constructor

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

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the *a != cst* relation, where *a* is encoded from bit index *inBitIndex* to bit index *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by the *inConstant* argument.

6.1.10 `binarySetWithPredicateString` Constructor

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

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: the *inBitString* argument string encodes a predicate string, such as those returned by @binaryset predicateStringValue reader (page 45).

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 s
@binaryset s [binarySetWithPredicateString !"␣"] ;
@bool b := [s isEmptySet]; # b is true

A predicate string containing only 'X' characters (at least one) provides an full binary
@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; # Displays <@binaryset: 1100>

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

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

6.1.11 `binarySetWithStrictGreaterComparison` Constructor

Returns an `@binaryset` object that encodes a strict greater than relation between two variables.

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the $a > b$ relation, where a is encoded from bit index *inLeftFirstIndex* to bit index *inLeftFirstIndex* + *inBitCount* - *1*, and b is encoded from bit index *inRightFirstIndex* to *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

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

6.1.12 `binarySetWithStrictGreaterThanConstant` Constructor

Returns an `@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 ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the $a > cst$ relation, where a is encoded from bit index *inBitIndex* to bit index *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by the *inConstant* argument.

6.1.13 `binarySetWithStrictLowerComparison` Constructor

Returns an `@binaryset` object that encodes a strict lower than relation between two variables.

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the *a* < *b* relation, where *a* is encoded from bit index *inLeftFirstIndex* to bit index *inLeftFirstIndex* + *inBitCount* - *1*, and *b* is encoded from bit index *inRightFirstIndex* to *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

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

6.1.14 `binarySetWithStrictLowerThanConstant` Constructor

Returns an `@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 ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: the constructor returns an binary set that encodes the *a* < *cst* relation, where *a* is encoded from bit index *inBitIndex* to bit index *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by the *inConstant* argument.

6.1.15 `emptyBinarySet` Constructor

Returns an empty `@binaryset` object.

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

Availability: available in GALGAS 1.6.0 and later.

6.1.16 `fullBinarySet` Constructor

Returns a full `@binaryset` object.

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

Availability: available in GALGAS 1.6.0 and later.

6.2 Readers

6.2.1 `accessibleStates` Reader

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

```
reader @binaryset accessibleStates -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

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

Exemple :

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

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

6.2.2 `binarySetByTranslatingFromIndex` Reader

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

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

Availability: available in GALGAS 1.6.3 and later.

6.2.3 `compressedValueCount` Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

6.2.4 `compressedStringValueList` Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

6.2.5 `containsValue` Reader

Returns an `@bool` value indicating whether the receiver'value contains a given value.

```
reader @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 ; \# Displays <@binaryset: 000XXX111, 111111111>
@bool b := [s containsValue !127L !0 !7] ;
log b ; \# Displays <@bool:true>
b := [s containsValue !31L !1 !7] ;
log b ; \# Displays <@bool:true>
b := [s containsValue !63L !1 !8] ;
log b ; \# Displays <@bool:false>
b := [s containsValue !7L !0 !9] ;
log b ; \# Displays <@bool:true>
b := [s containsValue !7L !0 !10] ;
log b ; \# Displays <@bool:true>
b := [s containsValue !32767L !1 !12] ;
log b ; \# Displays <@bool:true>
```

6.2.6 `equalTo` Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

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

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

6.2.7 `existOnBitIndex` Reader

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

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

Availability: available in GALGAS 1.6.3 and later.

6.2.8 `existsOnBitRange` Reader

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.

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

Availability: available in GALGAS 1.6.3 and later.

Exemple :

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

6.2.9 `existOnBitIndexAndBeyond` Reader

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

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

Availability: available in GALGAS 1.6.3 and later.

6.2.10 `forAllOnBitIndex` Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

6.2.11 `forAllOnBitIndexAndBeyond` Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

6.2.12 greaterOrEqualTo Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

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

6.2.13 isEmpty Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

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

6.2.14 isFull Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

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

6.2.15 ITE Reader

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

```
reader @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)`.

6.2.16 `lowerOrEqualTo` Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

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

6.2.17 `notEqualTo` Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

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

6.2.18 `predicateStringValue` Reader

Returns a string representation of the receiver's value.

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: the returned string is compatible with the binarySetWithPredicateString constructor (page 37).

6.2.19 strictGreaterThan Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

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

6.2.20 strictLowerThan Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

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

6.2.21 stringValueList Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

6.2.22 `stringValueListWithNameList` Reader

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

```
reader @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.

6.2.23 `swap132` Reader

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

```
reader @binaryset swap132
  ?@uint inBitCount1
  ?@uint inBitCount2
  ?@uint inBitCount3
  -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this reader 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*.

6.2.24 `swap21` Reader

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

```
reader @binaryset swap21
  ?@uint inBitCount1
  ?@uint inBitCount2
  -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this reader considers that the receiver encodes an *(x, y)* relation, where *x* is defi-

ned by bits index *0* to *inBitCount1 - 1*, *y* is defined by bits index *inBitCount1* to *inBitCount1* + *inBitCount2 - 1*.

6.2.25 `swap213` Reader

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

```
reader @binaryset swap213
  ?@uint inBitCount1
  ?@uint inBitCount2
  ?@uint inBitCount3
  -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this reader 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*.

6.2.26 `swap231` Reader

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

```
reader @binaryset swap231
  ?@uint inBitCount1
  ?@uint inBitCount2
  ?@uint inBitCount3
  -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this reader 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*.

6.2.27 `swap312` Reader

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

```
reader @binaryset swap312
  ?@uint inBitCount1
  ?@uint inBitCount2
  ?@uint inBitCount3
  -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this reader 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*.

6.2.28 `swap321` Reader

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

```
reader @binaryset swap321
  ?@uint inBitCount1
  ?@uint inBitCount2
  ?@uint inBitCount3
  -> @binaryset ;
```

Availability: available in GALGAS 1.6.0 and later.

Discussion: this reader 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*.

6.2.29 `transitiveClosure` Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: this reader 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*.

6.2.30 uint64ValueList Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

6.2.31 valueCount Reader

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

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

Availability: available in GALGAS 1.6.0 and later.

Discussion: no overflow test in performed.

6.3 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 returns an @binaryset object.

6.4 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 reader (page 42) and @binaryset notEqualTo reader (page 45) that return a `@binaryset` object.

6.5 Shift Operators

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

<<	Left Shift
>>	Right Shift

Exemple :

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

Chapitre 7

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.

7.1 Readers

7.1.1 `cString` Reader

Returns a string representation of the receiver's value.

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

Availability: available in GALGAS 1.8.7 and later.

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

7.1.2 `ocString` Reader

Returns a string representation of the receiver's value.

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

Availability: available in GALGAS 1.8.7 and later.

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

7.1.3 sint Reader

Returns the receiver's value in an type @sint (page 79) (32-bit signed integer) object.

```
reader @bool sint -> @sint ;
```

Availability: available in GALGAS 1.9.4 and later.

Discussion: returns the 1S type @sint (page 79) value if the receiver's value is true, and the 0S type @sint (page 79) value otherwise.

7.1.4 sint64 Reader

Returns the receiver's value in an type @sint64 (page 85) (64-bit signed integer) object.

```
reader @bool sint64 -> @sint64 ;
```

Availability: available in GALGAS 1.9.4 and later.

Discussion: returns the 1LS type @sint64 (page 85) value if the receiver's value is true, and the 0LS type @sint64 (page 85) value otherwise.

7.1.5 uint Reader

Returns the receiver's value in an type @uint (page 99) (32-bit unsigned integer) object.

```
reader @bool uint -> @uint ;
```

Availability: available in GALGAS 1.9.4 and later.

Discussion: returns the 1 type @uint (page 99) value if the receiver's value is true, and the 0 type @uint (page 99) value otherwise.

7.1.6 uint64 Reader

Returns the receiver's value in an type @uint64 (page 107) (64-bit unsigned integer) object.

```
reader @bool uint64 -> @uint64 ;
```

Availability: available in GALGAS 1.9.4 and later.

Discussion: returns the 1L type @uint64 (page 107) value if the receiver's value is true,

and the `0S` type `@uint64` (page 107) value otherwise.

7.2 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.

7.3 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 8

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'
character
```

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.

8.1 Constructors

8.1.1 `replacementCharacter` Constructor

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

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

Availability: available in GALGAS 1.8.3 and later.

8.1.2 `unicodeCharacterWithUnsigned` Constructor

Returns an `@char` object from an Unicode code point.

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

Availability: available in GALGAS 1.8.3 and later.

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 reader (page 101).

8.2 Readers

8.2.1 `isalnum` Reader

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

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

Availability: available in GALGAS 1.7.2 and later.

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.

8.2.2 `isalpha` Reader

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

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

Availability: available in GALGAS 1.7.2 and later.

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

8.2.3 `iscntrl` Reader

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

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

Availability: available in GALGAS 1.7.2 and later.

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

8.2.4 isdigit Reader

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

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

Availability: available in GALGAS 1.7.2 and later.

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

8.2.5 islower Reader

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

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

Availability: available in GALGAS 1.7.2 and later.

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

8.2.6 isUnicodeCommand Reader

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

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

Availability: available in GALGAS 1.8.3 and later.

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

8.2.7 isUnicodeLetter Reader

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

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

Availability: available in GALGAS 1.8.3 and later.

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

8.2.8 `isUnicodeMark` Reader

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

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

Availability: available in GALGAS 1.8.3 and later.

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

8.2.9 `isUnicodePunctuation` Reader

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

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

Availability: available in GALGAS 1.8.3 and later.

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

8.2.10 `isUnicodeSeparator` Reader

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

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

Availability: available in GALGAS 1.8.3 and later.

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

8.2.11 isUnicodeSymbol Reader

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

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

Availability: available in GALGAS 1.8.3 and later.

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

8.2.12 isupper Reader

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

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

Availability: available in GALGAS 1.7.2 and later.

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

8.2.13 string Reader

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

```
reader @char string -> @string ;
```

Availability: available in GALGAS 1.5.5 and later.

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

8.2.14 uint Reader

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

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

Availability: available in GALGAS 1.7.7 and later.

8.2.15 `unicodeName` Reader

Returns the unicode name of the receiver's value.

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

Availability: available in GALGAS 1.8.3 and later.

Discussion: for an decimal string representation of the receiver's value, see the @uint hexString reader (page 101); for a decimal string representation of the receiver's value, see the @uint string reader (page 103).

Exemple :

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

8.2.16 `unicodeToLower` Reader

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

```
reader @char unicodeToLower -> @char ;
```

Availability: available in GALGAS 1.8.3 and later.

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

Exemple :

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

8.2.17 `unicodeToUpper` Reader

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

```
reader @char unicodeToUpper -> @char ;
```

Availability: available in GALGAS 1.8.3 and later.

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

Exemple :

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

8.3 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 9

Le type `@data`

Chapitre 10

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 (.)

10.1 Constructor

10.1.1 `doubleWithBinaryImage` Constructor

Returns a double object from the binary image of the argument

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

Availability: available in GALGAS 2.4.7 and later.

10.1.2 `pi` Constructor

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

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

Availability: available in GALGAS 2.1.1 and later.

10.2 Readers

10.2.1 binaryImage Reader

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

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

Availability: available in GALGAS 2.4.7 and later.

10.2.2 cos Reader

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

```
reader @double cos -> @double ;
```

Availability: available in GALGAS 2.1.1 and later.

10.2.3 sin Reader

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

```
reader @double sin -> @double ;
```

Availability: available in GALGAS 2.1.1 and later.

10.2.4 sint Reader

Returns the receiver's value in an type @sint (page 79) (32-bit signed integer) object.

```
reader @double sint -> @sint ;
```

Availability: available in GALGAS 1.9.9 and later.

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

10.2.5 sint64 Reader

Returns the receiver's value in an type @sint64 (page 85) (64-bit signed integer) object.

```
reader @double sint64 -> @@sint64 ;
```

Availability: available in GALGAS 1.9.9 and later.

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

10.2.6 `string` Reader

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

```
reader @double string -> @string ;
```

Availability: available in GALGAS 1.7.7 and later.

Discussion: this reader never fails.

10.2.7 `tan` Reader

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

```
reader @double tan -> @double ;
```

Availability: available in GALGAS 2.1.1 and later.

10.2.8 `uint` Reader

Returns the receiver's value in an type `@uint` (page 99) (32-bit unsigned integer) object.

```
reader @double uint -> @uint ;
```

Availability: available in GALGAS 1.9.9 and later.

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

10.2.9 `uint64` Reader

Returns the receiver's value in an type `@uint64` (page 107) (64-bit unsigned integer) object.

```
reader @double uint64 -> @uint64 ;
```

Availability: available in GALGAS 1.9.9 and later.

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

10.3 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).

10.4 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 11

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 34 page 173).

11.1 Constructor

11.2 Modifier

11.2.1 Modifier `setCurrentDirectory`

```
modifier setCurrentDirectory ??@string inDirectory ;
```

11.3 Readers

11.3.1 Reader `allTextFilePathes`

```
reader allTextFilePathes -> @stringlist ;
```

11.3.2 Reader `allDirectoryPathes`

```
reader allDirectoryPathes -> @stringlist ;
```

11.3.3 Reader `currentDirectory`

```
reader currentDirectory -> @string ;
```

11.3.4 Reader `allFilePathesWithExtension`

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

11.3.5 Reader `directoryExistsAtPath`

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

11.3.6 Reader fileExistsAtPath

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

11.3.7 Reader textFileContentsAtPath

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

11.3.8 Reader binaryFileContentsAtPath

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

11.3.9 Reader absolutePathForPath

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

Chapitre 12

Le type `@location`

An `@location` object value is a location in a source file. Objects of this type are useful for pointing out an error or a warning location.

12.1 The `here` Keyword

The `here` keyword indicates the current parsing location is the current source file. Assigning an `@location` object from the `here` keyword is a way for initializing an `@location` object :

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

12.2 Constructor

12.2.1 `nowhere` Constructor

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

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

Availability: available in GALGAS 2.1.2 and later.

Discussion: The returned object responds `true` to the @location isNowhere reader (page 74).

12.3 Readers

12.3.1 `column` Reader

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

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

Availability: available in GALGAS 1.8.2 and later.

Discussion: this reader raises a run-time error if the receiver's value responds `true` to the @location isNowhere reader (page 74).

12.3.2 `isNowhere` Reader

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

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

Availability: available in GALGAS 2.1.2 and later.

Discussion: this reader 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.

12.3.3 `line` Reader

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

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

Availability: available in GALGAS 1.8.2 and later.

Discussion: this reader raises a run-time error if the receiver's value responds `true` to the @location isNowhere reader (page 74).

12.3.4 `locationIndex` Reader

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

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

Availability: available in GALGAS 1.8.2 and later.

Discussion: this reader raises a run-time error if the receiver's value responds `true` to the

@location isNowhere reader (page 74).

12.3.5 `locationString` Reader

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

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

Availability: available in GALGAS 1.8.2 and later.

Discussion: this reader raises a run-time error if the receiver's value responds `true` to the @location isNowhere reader (page 74).

Chapitre 13

Le type `@object`

Chapitre 14

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.

14.1 Constructors

14.1.1 `min` Constructor

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

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

Availability: available in GALGAS 1.3.0 and later.

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

14.1.2 `max` Constructor

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

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

Availability: available in GALGAS 1.3.0 and later.

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

14.2 Readers

14.2.1 double Reader

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

```
reader @sint double -> @double ;
```

Availability: available in GALGAS 1.9.8 and later.

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

14.2.2 sint64 Reader

Returns the receiver's value in an type @sint64 (page 85) (64-bit signed integer) object.

```
reader @sint sint64 -> @sint64 ;
```

Availability: available in GALGAS 1.6.12 and later.

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

This reader is the only way to convert an type @sint (page 79) object into an type @sint64 (page 85) object.

14.2.3 string Reader

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

```
reader @sint string -> @string ;
```

Availability: available in GALGAS 1.6.12 and later.

Discussion: for an hexadecimal string representation of the receiver's value, see @uint hexString reader (page 101) and @uint xString reader (page 103).

14.2.4 uint Reader

Returns the receiver's value in an type @uint (page 99) (32-bit unsigned integer) object.

```
reader @sint uint -> @uint ;
```

Availability: available in GALGAS 1.3.0 and later.

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

This reader is the only way to convert an type `@sint` (page 79) object into an type `@uint` (page 99) object.

14.2.5 `uint64` Reader

Returns the receiver's value in an type `@uint64` (page 107) (64-bit unsigned integer) object.

```
reader @sint uint64 -> @uint64 ;
```

Availability: available in GALGAS 1.3.0 and later.

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

This reader is the only way to convert an type `@sint` (page 79) object into an type `@uint64` (page 107) object.

14.3 Incrementation and decrementation

The type `@sint` (page 79) 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.

14.4 Arithmetic Operators

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

+	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 :

+	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}.

14.5 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.

14.6 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.

14.7 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 15

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.

15.1 Constructors

15.1.1 `min` Constructor

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

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

Availability: available in GALGAS 1.3.0 and later.

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

15.1.2 `max` Constructor

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

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

Availability: available in GALGAS 1.3.0 and later.

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

15.2 Readers

15.2.1 `double` Reader

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

```
reader @sint64 double -> @double ;
```

Availability: available in GALGAS 1.9.8 and later.

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

15.2.2 `sint` Reader

Returns the receiver's value in an type `@sint` (page 79) (32-bit signed integer) object.

```
reader @sint64 sint -> @sint ;
```

Availability: available in GALGAS 1.3.0 and later.

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

This reader is the only way to convert an type `@sint64` (page 85) object into an type `@sint` (page 79) object.

15.2.3 `string` Reader

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

```
reader @sint64 string -> @string ;
```

Availability: available in GALGAS 1.6.12 and later.

Discussion: this reader never fails.

15.2.4 `uint` Reader

Returns the receiver's value in an type `@uint` (page 99) (32-bit unsigned integer) object.

```
reader @sint64 uint -> @uint ;
```

Availability: available in GALGAS 1.3.0 and later.

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

This reader is the only way to convert an type `@sint64` (page 85) object into an type `@uint` (page 99) object.

15.2.5 `uint64` Reader

Returns the receiver's value in an type `@uint64` (page 107) (64-bit unsigned integer) object.

```
reader @sint64 uint64 -> @uint64 ;
```

Availability: available in GALGAS 1.3.0 and later.

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

This reader is the only way to convert an type `@sint64` (page 85) object into an type `@uint64` (page 107) object.

15.3 Incrementation and decrementation

The type `@sint64` (page 85) 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.

15.4 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}.

15.5 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.

15.6 Logical Operators

The `@sint64` 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 `@sint64` objects.

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

~	Bit-wise complementation

This operator returns an `@sint64` object.

15.7 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 16

Le type `@string`

A `@string` object value is an Unicode character string value. The @string type defines several constructors, readers constant methods and modifiers, 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 reader can be used on it. For example : `["ae" uppercaseString]` returns the "AE" string.

16.1 Readers

16.1.1 `containsCharacter` Reader

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

```
reader @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"
```

16.1.2 `subString` Reader

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.

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

Availability: available in GALGAS 1.7.8 and later.

Chapitre 17

Le type `@stringset`

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

17.1 Constructors

17.1.1 `emptySet` Constructor

Creates and returns an empty `@stringset` object.

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

Availability: available in GALGAS 1.3.0 and later.

17.1.2 `setWithString` Constructor

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

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

Availability: available in GALGAS 1.3.0 and later.

17.2 Readers

17.2.1 count Reader

Returns the number of strings in the set.

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

Availability: available in GALGAS 1.3.0 and later.

17.2.2 hasKey Reader

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

```
reader @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.

17.3 Modifier

17.3.1 removeKey Modifier

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

```
modifier @stringset removeKey
  ?@string inString
```

Availability: available in GALGAS 1.3.0 and later.

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

17.4 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 ;
```

17.5 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
```

17.6 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
```

17.7 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
```

17.8 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 ;
```

17.9 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 18

Le type `@type`

Chapitre 19

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.

19.1 Constructors

19.1.1 `errorCount` Constructor

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

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

Availability: available in GALGAS 1.4.9 and later.

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

Exemple :

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

19.1.2 `max` Constructor

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

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

Availability: available in GALGAS 1.3.0 and later.

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

19.1.3 `valueWithMask` Constructor

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

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

Availability: available in GALGAS 1.6.12 and later.

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)
```

19.1.4 `warningCount` Constructor

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

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

Availability: available in GALGAS 1.4.9 and later.

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

19.2 Readers

19.2.1 `double` Reader

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

```
reader @uint double -> @@double ;
```

Availability: available in GALGAS 1.9.8 and later.

Discussion: as a 32-bit integer value can always be converted in a `@double` value, this reader

never fails.

19.2.2 `hexString` Reader

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

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

Availability: available in GALGAS 1.5.2 and later.

Discussion: for getting an hexadecimal representation string without '0x' prefix, see @uint xString reader (page 103).

19.2.3 `isInRange` Reader

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

```
reader @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.

19.2.4 `isUnicodeValueAssigned` Reader

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

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

Availability: available in GALGAS 1.8.3 and later.

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).
```

19.2.5 lsbIndex Reader

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

```
reader @uint lsbIndex -> @uint ;
```

Availability: available in GALGAS 1.6.12 and later.

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.

19.2.6 significantBitCount Reader

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

```
reader @uint significantBitCount -> @uint ;
```

Availability: available in GALGAS 1.6.12 and later.

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
```

19.2.7 sint Reader

Returns the receiver's value in an type @sint (page 79) (32-bit signed integer) object.

```
reader @uint sint -> @sint ;
```

Availability: available in GALGAS 1.6.12 and later.

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

This reader is the only way to convert an type @uint (page 99) object into an type @sint (page 79) object.

19.2.8 `sint64` Reader

Returns the receiver's value in an type `@sint64` (page 85) (64-bit signed integer) object.

```
reader @uint sint64 -> @sint64 ;
```

Availability: available in GALGAS 1.6.12 and later.

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

This reader is the only way to convert an type `@uint` (page 99) object into an type `@sint64` (page 85) object.

19.2.9 `string` Reader

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

```
reader @uint string -> @string ;
```

Availability: available in GALGAS 1.6.12 and later.

Discussion: for an hexadecimal string representation of the receiver's value, see @uint hexString reader (page 101) and @uint xString reader (page 103).

19.2.10 `uint64` Reader

Returns the receiver's value in an type `@uint64` (page 107) (64-bit unsigned integer) object.

```
reader @uint uint64 -> @uint64 ;
```

Availability: available in GALGAS 1.6.12 and later.

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

This reader is the only way to convert an type `@uint` (page 99) object into an type `@uint64` (page 107) object.

19.2.11 `xString` Reader

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

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

Availability: available in GALGAS 1.9.10 and later.

Discussion: for an decimal string representation of the receiver's value, see the @uint hexString reader (page 101); for a decimal string representation of the receiver's value, see the @uint string reader (page 103).

19.3 Incrementation and decrementation

The type @uint (page 99) 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.

19.4 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).

19.5 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.

19.6 Logical Operators

The `@uint` 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 `@uint` objects.

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

~	Bit-wise complementation

This operator returns an `@uint` object.

19.7 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 20

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.

20.1 Constructeurs

20.1.1 `max` Constructor

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

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

Availability: available in GALGAS 1.3.0 and later.

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

20.1.2 `uint64BaseValueWithCompressedBitString` Constructor

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 ;
```

Availability: available in GALGAS 1.6.4 and later.

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> ;
```

20.1.3 `uint64MaskWithCompressedBitString` Constructor

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 ;
```

Availability: available in GALGAS 1.6.4 and later.

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> ;
```

20.1.4 `uint64WithBitString` Constructor

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

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

Availability: available in GALGAS 1.6.4 and later.

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> ;
```

20.2 Readers

20.2.1 `double` Reader

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

```
reader @uint64 double -> @double ;
```

Availability: available in GALGAS 1.9.8 and later.

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

20.2.2 `hexString` Reader

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

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

Availability: available in GALGAS 1.5.2 and later.

Discussion: for getting an hexadecimal representation string without "0x" prefix, see @uint64 xString reader (page 111).

20.2.3 `sint` Reader

Returns the receiver's value in an type `@sint` (page 79) (32-bit signed integer) object.

```
reader @uint64 sint -> @sint ;
```

Availability: available in GALGAS 1.6.12 and later.

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

This reader is the only way to convert an type @uint64 (page 107) object into an type @sint (page 79) object.

20.2.4 sint64 Reader

Returns the receiver's value in an type @sint64 (page 85) (64-bit signed integer) object.

```
reader @uint64 sint64 -> @sint64 ;
```

Availability: available in GALGAS 1.6.12 and later.

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

This reader is the only way to convert an type @uint64 (page 107) object into an type @sint64 (page 85) object.

20.2.5 string Reader

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

```
reader @uint64 string -> @string ;
```

Availability: available in GALGAS 1.6.12 and later.

Discussion: for an hexadecimal string representation of the receiver's value, see @uint64 hexString reader (page 109) and @uint64 xString reader (page 111).

20.2.6 uint Reader

Returns the receiver's value in an type @uint (page 99) (32-bit unsigned integer) object.

```
reader @uint64 uint -> @uint ;
```

Availability: available in GALGAS 1.6.12 and later.

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

This reader is the only way to convert an type @uint64 (page 107) object into an type @uint (page 99) object.

20.2.7 `uintSlice` Reader

Returns an type @uint (page 99) value, extracted from a bit slice of the receiver's value.

```
reader @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
```

20.2.8 `xString` Reader

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

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

Availability: available in GALGAS 1.9.10 and later.

Discussion: for an decimal string representation of the receiver's value, see the @uint64 hexString reader (page 109); for a decimal string representation of the receiver's value, see the @uint64 string reader (page 110).

20.3 Incrementation and decrementation

The type @uint64 (page 107) 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.

20.4 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).

20.5 Shift Operators

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

<<	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.

20.6 Logical Operators

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

&	Bit-wise and
|	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 :

~	Bit-wise complementation

This operator returns an `@uint64` object.

20.7 Comparison Operators

The `@uint64` 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 `@uint64` objects, and return a `@bool` object.

Chapitre 21

Le type `list`

21.1 List Type Declaration

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

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

21.2 Constructors

21.2.1 The `emptyList` constructor

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

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

21.2.2 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 ;
```

21.3 Adding elements

21.3.1 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 ;''
```

21.3.2 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 ;''
```

21.3.3 The prependValue modifier

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

The `prependValue` modifier 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];
```

21.3.4 Modifier insertAtIndex

Le modifier `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 modifier 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 modifier *prependValue*), et la longueur courante de la liste (insertion à la fin, comme le fait l'opérateur `+=`, section 21.3.1 page 116). 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];
```

21.3.5 The concatenation operator

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

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

21.4 Removing elements

21.4.1 The `popFirst` modifier

The `popFirst` modifier 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` modifier is invoked, a run-time error is raised and the input arguments are not valuated.

21.4.2 The `popLast` modifier

The `popLast` modifier 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.

21.5 Methods

21.5.1 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.

21.5.2 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.

21.6 Readers

21.6.1 Le reader lengthr

```
reader length -> @uint ;
```

Le reader length reader retourne le nombre d'éléments du receveur.

21.6.2 The range reader

```
reader range -> @range ;
```

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

21.6.3 The subListFromIndex reader

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

This reader 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 reader returns an empty list.

21.6.4 The subListWithRange reader

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

This reader 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 reader returns an empty list.

21.7 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 ;
}
```

21.7.1 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 ;
```

21.7.2 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 ;
```

21.7.3 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 ;
```

21.8 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` reader)

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.

21.8.1 Read Access

By default and for every attribute, a reader 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* reader is provided. It accepts one `@uint` argument, the value of the index.

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

For example :

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

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

21.8.2 Write Access

By default, no modifier 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 modifier 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 modifier 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 modifier is available : `@setMFirstAttributeAtIndex` ; the `@setMSecondAttributeAtIndex` modifier is not available.

21.8.3 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 22

Le type `sortedlist`

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

22.1 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 22.2.1 page 124) ;
- le constructeur `sortedListWithValue` qui construit une liste contenant un élément (section 22.2.2 page 124) ;
- l'opérateur `+=` pour ajouter un élément à une liste ordonnée (section 22.3.1 page 124) ;
- l'opérateur `.=` pour ajouter tous les éléments d'une liste à une liste ordonnée (section 22.3.2 page 124) ;
- l'opérateur `.` pour construire une liste ordonnée à partir de deux listes ordonnées (section 22.3.3 page 125) ;
- le *reader* `length`, qui retourne le nombre d'éléments d'une liste (section 22.4 page 125) ;
- le *modifier* `popGreatest`, qui retourne les champs du plus grand élément d'une liste, et retire cet élément de cette liste (section 22.5.1 page 125) ;
- le *modifier* `popSmallest`, qui retourne les champs du plus grand élément d'une liste, et retire cet élément de cette liste (section 22.5.2 page 125) ;
- la *méthode* `greatest`, qui retourne les champs du plus grand élément d'une liste sans

la modifier (section 22.6.1 page 126) ;
— la *méthode* `smallest`, qui retourne les champs du plus petit élément d'une liste sans la modifier (section 22.6.2 page 126).

22.2 Constructeurs

22.2.1 Constructeur `emptySortedList`

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

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

22.2.2 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
] ;
```

22.3 Opérateurs

22.3.1 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
```

22.3.2 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 ;
```

22.3.3 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 ;
```

22.4 Le reader `length`

Le reader `length` retourne un `@uint` contenant le nombre d'éléments de la liste ordonnée.

22.5 Modifiers

22.5.1 Le modifier `popGreatest`

Ce *modifier* 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*.

22.5.2 Le modifier `popSmallest`

Ce *modifier* 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*.

22.6 Méthodes

22.6.1 La méthode `greatest`

Ce *modifier* 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*.

22.6.2 La méthode `smallest`

Ce *modifier* 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*.

22.7 Énumération avec l'instruction `foreach`

L'instruction `foreach` (section 36.11 page 185) 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 23

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.

23.1 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] ;
```

23.2 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 reader `valueAtIndex` déclenche une *run-time error*. Pour être valide, un élément doit avoir été initialisé par un appel au modifier `setValueAtIndex`.

Il est valide d'affecter la valeur 0 à un ou plusieurs axes. Le tableau ne contient alors aucun élément.

23.3 Accès à un élément

L'accès à la valeur d'un élément s'effectue par le reader `valueAtIndex`. La modification de la valeur d'un élément est réalisée par le modifier `setValueAtIndex` ou le modifier `forceValueAtIndex`.

23.3.1 Le reader `valueAtIndex`

Ce reader 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.

23.3.2 Le modifier `setValueAtIndex`

Ce modifier 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] ;
```

23.3.3 Le modifier `forceValueAtIndex`

Ce modifier 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 modifier `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] ;
```

23.4 Validité d'un élément

Le reader `isValueValidAtIndex` permet de savoir si un élément est valide ou non, c'est à dire si sa lecture déclenchera une *run-time error*. Le modifier `invalidateValueAtIndex` invalide un élément.

23.4.1 Le reader `isValueValidAtIndex`

Ce reader 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] ;
```

23.4.2 Le modifier `invalidateValueAtIndex`

Ce modifier 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 reader `valueAtIndex` pour lire cet élément déclenchera une *run-time error*.

Par exemple :

```
[!?t invalidateValueAtIndex !1 !2 !2] ;
```

23.5 Contrôle des tailles des axes

Le reader `axisCount` renvoie la dimension d'un tableau, c'est à dire le nombre de ces axes, le reader `sizeForAxis` renvoie la taille allouée à un axe particulier. Les modifiers `setSizeForAxis` et `setSize` permettent de modifier la taille d'un tableau.

23.5.1 Le reader `axisCount`

Ce reader 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 reader `axisCount` renvoie la valeur 3.

23.5.2 Le reader `sizeForAxis`

Ce reader 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, ... 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.

23.5.3 Le reader `rangeForAxis`

Ce reader 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, ... 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.

23.5.4 Le modifier `setSizeForAxis`

Ce modifier 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 modifier `setValueAtIndex`.

23.5.5 Le modifier `setSize`

Ce modifier 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 modifier `setValueAtIndex`.

23.6 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 24

Le type `class`

24.1 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 *readers* pour lire les attributs, et des *modifiers* pour les écrire. On ne peut pas définir explicitement d'autres constructeurs, *readers* ou *modifiers* à l'intérieur de la classe. Cependant, les catégories (chapitre 33 page 163) permettent de définir *readers*, *méthodes* et *modifiers* associés à une classe.

24.2 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
] ;
```

24.3 Lecture d'un attribut

Par défaut, la lecture d'un attribut est activée par la définition implicite d'un *reader*, 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 *reader* 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.

24.4 Écriture d'un attribut

Par défaut, l'écriture d'un attribut n'est pas activée.

Pour activer la génération d'un *modifier* permettant décrire un attribut, compléter la déclaration de cet attribut par `feature setter`. Un *modifier* 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 *reader* de lecture d'un attribut et engendrer le *modifier* d'écriture, il suffit de déclarer l'attribut par :

```
  @uint mA feature nogetter, setter ;
```

Ou encore :

```
  @uint mA feature setter, nogetter ;
```

24.5 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 := ... ;
```

24.5.1 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 35.1.3 page 177) ;
- l'expression de test du type dynamique (section 35.1.4 page 178) ;
- l'instruction `cast` (section 36.4 page 183).

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 25

Le type `enum`

Galgas permet à l'utilisateur de définir des types énumérés.

25.1 Déclaration

La déclaration d'un type `enum` nomme l'ensemble des constantes associées à ce type.

Par exemple :

```
enum @feuTricolore {
  vert, orange, rouge
}
```

Plusieurs types énumérés peuvent définir des constantes de même nom.

25.2 Instanciation

Chaque constante définit un constructeur de même nom. On peut ainsi écrire :

```
@feuTricolore feu := [@feuTricolore vert] ;
```

Ou encore :

```
@feuTricolore feu [vert] ;
```

25.3 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]
```

25.4 L'instruction `switch`

L'instruction `switch` (section 36.21 page 191) est dédiée aux types énumérés.

Chapitre 26

Le type `graph`

Chapitre 27

Le type `map`

Un objet de type `map` est une table de symboles, chaque symbole étant associé à des valeurs.

27.1 Déclaration

La déclaration d'un type `map` nomme :
- les attributs qui sont associés à une clé ;
- les *modifiers* d'insertion ;
- les *méthodes* de recherche ;
- les *modifiers* de retrait ;

Les clés sont déclarées implicitement et sont du type `@lstring` (page 152).

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" ;
}
```

27.2 Modifiers d'insertion

Une `map` peut déclarer zéro, un ou plusieurs *modifiers* d'insertion. Un *modifier* 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 *modifier* d'insertion est déclaré par :

```
insert nom error message "message_erreur" ;
```

L'identificateur `nom` donne un nom au *modifier* d'insertion ; ce nom doit être unique parmi les *modifiers* 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 *modifier* d'insertion est appelé dans une *instruction d'appel de modifier*, 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 [emptyMap] ;
@lstring clef := ... ;
@string s := ... ;
@uint v := ... ;
[!?uneTable insertKey !clef !s !v] ;
```

27.3 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 [emptyMap] ;
...
@lstring clef := ... ;
[!?uneTable searchKey !clef ?@string s ?@uint v] ;
```

27.4 Modifiers de retrait

Une `map` peut déclarer zéro, un ou plusieurs *modifiers* de retrait. Un *modifier* 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 *modifier* de retrait est déclaré par :

```
remove nom error message "message_erreur" ;
```

L'identificateur `nom` donne un nom au *modifier* de retrait ; ce nom doit être unique parmi les *modifiers* 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 *modifier* de retrait est appelé dans une *instruction d'appel de modifier* :

— 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 [emptyMap] ;
...
@lstring clef := ... ;
[!?uneTable removeKey !clef ?@string s ?@uint v] ;
```

27.5 Constructeurs

27.5.1 Constructeur `emptyMap`

```
constructor @T emptyMap -> @T ;
```

Ce constructeur permet d'instancier une table vide. Exemple :

```
@MaTable uneTable [emptyMap] ;
```

27.5.2 Constructeur `mapWithMapToOverride`

```
constructor @T 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 [inMapToOverride !uneTable] ;
```

27.6 Readers

27.6.1 Le reader `count`

```
reader @T count -> @uint ;
```

Le *reader* `count` retourne un `@uint` qui contient le nombre d'entrées de la table de premier niveau du receveur.

27.6.2 Le reader `hasKey`

```
reader @T hasKey ??@string inKey -> @bool ;
```

Le *reader* `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.

27.6.3 Le reader `keyList`

```
reader @T keyList -> @lstringlist ;
```

Le *reader* `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.

27.6.4 Le reader `keySet`

```
reader @T keySet -> @stringset ;
```

Le *reader* `keySet` retourne l'ensemble de toutes les clés de la table de premier niveau du receveur.

27.6.5 Le reader `locationForKey`

```
reader @T locationForKey ??@string inKey -> @location ;
```

Le *reader* `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.

27.6.6 Le reader `overriddenMap`

```
reader @T overriddenMap -> @T ;
```

Le *reader* `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.

27.7 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 [emptyMap] ;
[!?uneTable insertKey ![@lstring new !"z" !here] !"world" !5] ;
[!?uneTable insertKey ![@lstring new !"a" !here] !"hello" !10] ;
foreach aMap do
  message lkey->string . "␣" . mPremier . "␣" . mSecond . "\n" ;
end foreach ;
```

L'affichage produit est :

```
a hello 10
z world 5
```

Chapitre 28

Structure type

Chapitre 29

Types prédéfinis

The following types are predefined, as particular structure types.

29.1 Types structure prédéfinis

Les types suivants sont des types structures prédéfinis, dont les champs sont des types de base.

29.1.1 Le type `@lbool`

The `@lbool` is predefined as :

```
struct @lbool {
  @bool bool ;
  @location location ;
}
```

29.1.2 Le type `@lchar`

The `@lchar` is predefined as :

```
struct @lchar {
  @char char ;
  @location location ;
}
```

29.1.3 Le type `@ldouble`

The `@ldouble` is predefined as :

```
struct @ldouble {
  @double double ;
  @location location ;
}
```

29.1.4 Le type `@lsint`

The `@lsint` is predefined as :

```
struct @lsint {
  @sint sint ;
  @location location ;
}
```

29.1.5 Le type `@lsint64`

The `@lsint64` is predefined as :

```
struct @lsint64 {
  @sint64 sint64 ;
  @location location ;
}
```

29.1.6 Le type `@lstring`

The `@lstring` is predefined as :

```
struct @lstring {
  @string string ;
  @location location ;
}
```

29.1.7 Le type `@luint`

The `@luint` is predefined as :

```
struct @luint {
  @uint uint ;
  @location location ;
}
```

29.1.8 Le type `@luint64`

The `@luint64` is predefined as :

```
struct @luint64 {
  @uint64 uint64 ;
  @location location ;
}
```

29.1.9 Le type `@range`

The `@range` is equivalent to the declaration :

```
struct @range {
  @uint start ;
  @uint length ;
}
```

Deuxième partie

Sous-programmes

Chapitre 30

Sous-programmes

GALGAS définit les sous-programmes suivants :
- les *routines* (chapitre 32 page 161) ;
- les *fonctions* (chapitre 31 page 159) ;
- les *méthodes* (section 33.2 page 165) ;
- les *readers* (section 33.1 page 164) ;
- les *modifiers* (section 33.3 page 165).

En GALGAS, *méthodes*, *readers* et *modifiers* s'appliquent sur un objet d'un type quelconque (qui n'est donc pas forcément un type *classe*). Pour les types définis par le programmeur, *méthodes*, *readers* et *modifiers* sont toujours déclarés en dehors de la déclaration du type auquel ils s'appliquent.

Un sous-programme peut donc être directement déclaré dans :
- un composant *semantics* ;
- un composant *syntax* ;
- un composant *program*.

À chaque nature de sous-programme correspond une construction particulière pour l'appeler (tableau 30.1).

Sous-programme	**Construction**	**Référence**
routine	Instruction d'appel de routine	section 36.9 page 185
fonction	Appel de fonction (dans une expression)	section 35.1.13 page 179
méthode	Instruction d'appel de méthode	section 36.19 page 191
reader	Appel de reader (dans une expression)	section 35.1.14 page 179
modifier	Instruction d'appel de modifier	section 36.20 page 191

Tableau 30.1 – *Constructions d'appel de sous programme*

Syntaxe	Remarque
`?@T var`	`var` est modifiable localement
`?@T unused var`	`var` n'est pas utilisée
`??@T var`	`var` est une constante
`?@T unused var`	`var` est une constante inutilisée

(a) Argument formel

Syntaxe
`!expression`

(b) Paramètre effectif

Tableau 30.2 – *Argument formel en entrée, paramètre effectif en sortie*

30.1 Arguments formels et paramètres effectifs

30.1.1 Argument formel en entrée

30.1.2 Argument formel en entrée/sortie

Syntaxe	Remarque
`?!@T var`	`var` est modifiable
`?!@T unused var`	`var` n'est pas utilisée

(a) Argument formel

Syntaxe
`!?cible`

(b) Paramètre effectif

Tableau 30.3 – *Argument formel en entrée / sortie, paramètre effectif en sortie / entrée*

30.1.3 Argument formel en sortie

Syntaxe
`!@T var`

(a) Argument formel

Syntaxe	Remarque
`?nom`	Affectation d'une variable
`?@T nom`	Déclaration et affectation d'une variable
`?*`	Variable anonyme
`??@T nom`	Déclaration et affectation d'une constante

(b) Paramètre effectif

Tableau 30.4 – *Argument formel en sortie, paramètre effectif en entrée*

Chapitre 31

Fonctions

Chapitre 32

Routines

Chapitre 33

Catégories

Categories are the way for adding *readers*, *methods* and *modifiers* to any type. They are defined outside type declarations.

You can declare for any type :
- *category readers* ;
- *category methods* ;
- *category modifiers*.

Additional features are available for classes and are described in section 33.4 page 167.

A *category reader* is called in an expression. As expressions have no side-effect, a category reader cannot change current object's value.

A *category method* is called by the *method call instruction* (section 36.19 page 191). A category method cannot modify current object's value.

A *category modifier* is called by the *modifier call instruction* (section 36.20 page 191). A category modifier can modify current object's value.

Within the category reader, method and modifier instruction list, the `selfcopy` 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 readers since they cannot modify the current object. Using `self` in category modifiers is explained in section 33.3 page 165.

A category reader, method and modifier can be declared in :
- a *semantics* component ;
- a *syntax* component ;
- a *program* component.

A declared category reader, method and modifier 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 readers 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 reader with the same name than a predefined reader : the compiler issues an error on on a such declaration. The same rules apply on category methods and category modifiers.

However, it is safe to declare for a given type a category reader, a category method and a category modifier with the same name. GALGAS compiler uses different naming spaces for them, and call syntax are different, so there is no ambiguity.

33.1 Category reader

A category reader is declared like a function, but its header names a type and a reader name. As a function, it accepts zero, one or more input and constant input formal parameters.

For example, the following code add a reader to the type `@uint64` (page 107) that computes the square of its value :

```
reader @uint64 square -> @uint64 outResult :
  outResult := selfcopy * selfcopy ;
end reader ;
```

This reader is called like a predefined reader :

```
@uint64 v := 7L ;
log "Square␣of␣7": [v square] ; # LOGGING Square of 7 : <@uint64:49>
```

You can add a category reader to a list :

```
reader @uintlist sum -> @uint outResult :
  outResult := 0 ;
  foreach selfcopy do
    outResult := outResult + mValue ;
  end foreach ;
end reader ;
```

For counting the number of element values greater than the value given in argument :

```
reader @uintlist countValuesGreaterThan
  ??@uint inTestValue -> @uint outResult
:
  outResult := 0 ;
  foreach selfcopy do
    if mValue > inTestValue then
      outResult ++ ;
    end if ;
  end foreach ;
end reader ;
```

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 type `@lstring` (page 152) has an attribute `string` whose type is `@string`. The following reader returns the value of this attribute, appended with the `"␣!"` string :

```
reader @lstring op -> @string outResult :
  outResult := string . "␣!" ;
end reader ;
```

33.2 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 type @uint64 (page 107) that computes the square of its value :

```
method @uint64 square !@uint64 outResult :
  outResult := selfcopy * selfcopy ;
end method ;
```

This reader 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 ;
  foreach selfcopy do
    outResult := outResult + mValue ;
  end foreach ;
end method ;
```

For counting the number of element values greater than the value given in argument :

```
method @uintlist countValuesGreaterThan
  ??@uint inTestValue
  !@uint outResult
:
  outResult := 0 ;
  foreach selfcopy do
    if mValue > inTestValue then
      outResult ++ ;
    end if ;
  end foreach ;
end method ;
```

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 type @lstring (page 152) 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 . "␣!" ;
end method ;
```

33.3 Category modifier

A category method is declared like a routine, but its header names a type and a modifier 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 modifier can change the value of the current object.

For structure and classes types, attributes can be read, written, read / written. For example :

```
modifier @lstring appendInt ??@uint inValue :
  string .= [inValue string] ;
end modifier ;
```

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 *modifier call instruction* (section 36.20 page 191) ;
2. the *append instruction* (section 36.2 page 183) ;
3. the *concat instruction* (section 36.5 page 184) ;
4. the *increment instruction* (section 36.12 page 188) ;
5. the *decrement instruction* (section 36.6 page 184) ;
6. the *assignment instruction* (section 36.3 page 183).

Example of using `self` in modifier call instruction ; this modifier prepends the square of argument value to the `@uint64list` value :

```
modifier @uint64list prependSquare ??@uint64 inValue :
  [!?self prependValue !inValue * inValue] ;
end modifier ;
```

Example of using `self` in the append instruction ; this modifier appends the square of argument value to the `@uintlist` value :

```
modifier @uintlist appendSquare ??@uint inValue :
  self += !inValue * inValue ;
end modifier ;
```

This construct is valid only for types that handle the += operator.

Example of using `self` in the concat instruction ; this modifier appends to the string all items of the `@stringlist` argument value :

```
modifier @string concatList ??@stringlist inList :
  foreach inList do
    self .= mValue ;
  end foreach ;
end modifier ;
```

This construct is valid only for types that handle the .= operator.

Example of using `self` in the increment instruction ; this modifier increments the receiver's value :

```
modifier @uint increment :
  self ++ ;
end modifier ;
```

This construct is valid only for types that handle the ++ operator, such as type `@uint` (page 99), type `@uint64` (page 107), type `@sint` (page 79), type `@sint64` (page 85).

Example of using `self` in the assignment instruction ; this modifier removes all odd values of the receiver :

```
modifier @uintlist removeOddValues :
  @uintlist listWithEvenValues [emptyList] ;
  foreach selfcopy do
    if (mValue & 1) == 0 then
      listWithEvenValues += !mValue ;
    end if ;
  end foreach ;
  self := listWithEvenValues ;
end modifier ;
```

This construct is valid only for types, but class types.

33.4 Categories and classes

Additional features are available only for classes ; in addition to category readers, methods and modifiers described in the above sections, you can declare :

— *abstract* category readers, methods, modifiers ;
— *overriding* category readers, methods, modifiers ;
— *overriding abstract* category readers, methods, modifiers.

Abstract category readers, methods, modifiers and *overriding abstract* category readers, methods, modifiers do not contain any instruction list : they act as *prototypes*.

Examples of *abstract* category readers, methods, modifiers declarations :

```
abstract reader @aType readerName
  ?anOtherType aParameter
  -> @resultType outResult
;

abstract method @aType methodName
  ?anOtherType aParameter
;

abstract modifier @aType modifierName
  ?anOtherType aParameter
;
```

Examples of *overriding* category readers, methods, modifiers declarations :

```
override reader @aType readerName
  ?anOtherType aParameter
  -> @resultType outResult
:
  instructions
;

override method @aType methodName
  ?anOtherType aParameter
:
  instructions
;

override modifier @aType modifierName
  ?anOtherType aParameter
:
```

```
  instructions
;
```

Examples of *overriding abstract* category readers, methods, modifiers declarations :

```
override abstract reader @aType readerName
  ?anOtherType aParameter
  -> @resultType outResult
;

override abstract method @aType methodName
  ?anOtherType aParameter
;

override abstract modifier @aType modifierName
  ?anOtherType aParameter
;
```

Neither *abstract* category readers, methods, modifiers, neither *overriding abstract* category readers, methods, modifiers cannot be declared for concrete classes. Any kind of category reader, method, modifier can be declared for abstract classes.

If an *abstract* category reader, method, modifier, or an *overriding abstract* category reader, method, modifier 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 reader, method, modifier that has the same name as a category reader, method, modifier declared for one of its super classes should be declared as *overriding*.

An abstract category reader, method, modifier that has the same name as a category reader, method, modifier 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 33.1, 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 reader, method or modifier.

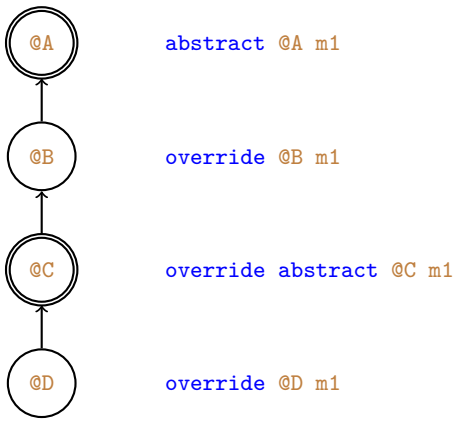

FIGURE *33.1* – *inheritance graph and categories*

`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.

Troisième partie

Filewrappers et templates

Chapitre 34

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 type `@filewrapper` (page 71).

34.1 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.

Quatrième partie

Instructions et expressions

Chapitre 35

Expressions

D'une manière classique, une expression est constituée d'*opérandes* (section 35.1 page 177) et d'*opérateurs* (section 35.2 page 180). La priorité des opérateurs est définie dans le tableau 35.2 page 180.

35.1 Opérandes

35.1.1 Identificateur

35.1.2 `selfcopy`

`selfcopy` représente une copie de l'objet courant. On ne peut donc utiliser `selfcopy` que dans une expression à l'intérieur d'une *méthode*, d'un *reader*, d'un *modifier*, ou d'une catégorie (chapitre 33 page 163). Sont donc exclues les routines et les fonctions.

`selfcopy` effectue un accès en lecture seule de l'objet courant.

Voici un exemple extrait de la section décrivant les *reader catégorie* (section 33.1 page 164) :

```
reader @uint64 square -> @uint64 outResult :
  outResult := selfcopy * selfcopy ;
end reader ;
```

35.1.3 Expression de conversion polymorphique inverse

La syntaxe de l'*expression de conversion polymorphique inverse* est : `(cast expression : @T)`. Les parenthèses sont obligatoires. 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 35.1.4 page 178. On peut aussi utiliser l'instruction `cast` (section 36.4 page 183).

35.1.4 Test du type dynamique d'une expression

L'opérande `(expression is conversion @T)` (les parenthèses sont obligatoires) 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
  @B var := (cast expression : @T) ;
  ...
elsif (expression is >= @C) then
  @C var := (cast expression : @C) ;
  ...
else
  message "conversion␣impossible" ;
end if ;
```

35.1.5 Parenthèses

Les parenthèses `(` et `)` permettent de forcer le groupement d'opérandes.

35.1.6 `true` et `false`

`true` et `false` sont les constantes du type `@bool`.

35.1.7 `here`

`here` est une constante de type `@location`. Elle a pour valeur la position courante de la lecture du fichier source.

35.1.8 Constante Chaîne de caractères

35.1.9 Constante caractère

35.1.10 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`.

35.1.11 Constante flottante

35.1.12 Expression `if`

35.1.13 Appel de fonction

35.1.14 Appel de reader

35.1.15 Appel de constructeur

35.1.16 Constructeur par défaut

L'expression `[@T default]` invoque le constructeur par défaut du type `@T` et renvoie un objet initialisé du type `@T`.

Pour la plupart des types, un constructeur par défaut est implicitement défini (voir le détail section 4.2 page 28).

35.1.17 Valeur d'une option

Les options de la ligne de commande sont définies dans un composant `option` (chapitre 41 page 205). 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 35.1.

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 35.1 – *Informations relatives à une option de la ligne de commande*

Par exemple, si un composant `option` est déclaré comme suit :

```
option mesOptions :
  @bool extractOption : 'S', "asm" -> "Extract␣assembly␣code" ;
end  option ;
```

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"`.

35.2 Opérateurs

35.2.1 Priorité des opérateurs

La priorité des opérateurs est définie dans le tableau 35.2. 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`.

Priorité	Opérateur	Commentaire	Référence
0 (plus faible)	`.`	Concaténation	section 35.2.2 page 180
1	`|`	« ou » logique	section 35.2.3 page 180
	`^`	« ou exclusif » logique	section 35.2.3 page 180
2	`&`	« et » logique	section 35.2.3 page 180
3	`==` , `!=`	Comparaison	section 35.2.5 page 181
	`<` , `<=`	Comparaison	section 35.2.5 page 181
	`>` , `>=`	Comparaison	section 35.2.5 page 181
4	`<<` , `>>`	Décalage	section 35.2.6 page 181
	`+`	Addition	section 35.2.7 page 181
	`-`	Soustraction	section 35.2.7 page 181
5	`*`	Multiplication	section 35.2.7 page 181
	`/`	Division	section 35.2.7 page 181
	`mod`	Modulo	section 35.2.7 page 181
6	`-`	Négation arithmétique	section 35.2.7 page 181
7	`not`	Complémentation booléenne	section 35.2.3 page 180
8	`~`	Complémentation bit-à-bit	section 35.2.4 page 181
9 (plus forte)	`->`	Accès à un champ d'une structure	section 35.2.8 page 181

Tableau 35.2 – *Priorité des opérateurs*

35.2.2 Concaténation

`.`

35.2.3 Logique

`|` , `^` , `&` , `not`

35.2.4 Complémentation bit-à-bit

`~`

35.2.5 Comparaison

35.2.6 Décalage

`<<` et `>>`

35.2.7 Arithmétique

`+`, `-`, `*`, `/`, `mod`.

`-` unaire.

35.2.8 Accès à un champ d'une structure

`->`

Chapitre 36

Instructions sémantiques

36.1 Cible

La notation *cible* apparaît dans plusieurs instructions :
— l'instruction d'affectation (section 36.3 page 183) ;

Cette notation est décrite par le diagramme syntaxique de la figure 36.1.

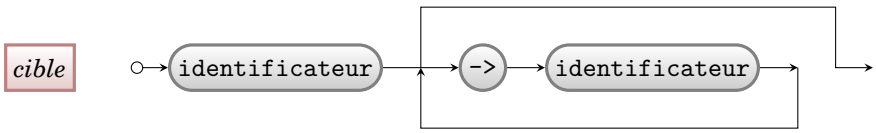

FIGURE *36.1* – *Diagramme syntaxique de* `cible`

Un identificateur seul représente une variable.

36.2 Append Instruction

36.3 Assignment Instruction

36.4 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
when conversion @T1 variable :
  ...
when conversion @T2 variable :
  ...
else
  ...
end cast ;
```

L'instruction accepte une ou plusieurs branches `when`, et zéro ou une branche `else`. *conversion* est soit `==`, soit `>=`. *variable* est une constante dont le type est le type nommé dans la branche `when` qui la déclare, et dont la portée est limitée à cette branche `when`.

Lors de l'exécution, le type dynamique de *expression* est comparé successivement aux types (`@T1`, `@T2`) nommés dans les branches `when` ; 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 `>`),

variable 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
when >= @B variable :
  ...
when >= @C variable :
  ...
else
  message "conversion␣impossible"
end if ;
```

Note : si la variable `var` n'est pas utilisée dans la branche correspondante, une alerte est émise. Pour la supprimer, ne pas mentionner la variable en écrivant `when >= @T :`.

36.5 Concat Instruction

36.6 Decrement Instruction

36.7 L'instruction `drop`

La syntaxe de l'instruction `drop` est la suivante :

```
drop variable, ... ;
```

Chaque variable nommée est placée dans l'état *non construit*.

36.8 Error Instruction

36.9 L'instruction d'appel de routine

36.10 L'instruction `for`

36.11 L'instruction `foreach`

L'instruction `foreach` permet d'énumérer :
- une collection ;
- plusieurs collections de manière synchrone.

Cette instruction s'est présentée sous plusieurs formes au cours des versions successives de GALGAS, seule la version non obsolète est exposée.

36.11.1 Présentation

Pour énumérer une collection, la syntaxe est la suivante :

```
foreach sens expression
index nom_index # Optionnel
while condition # Optionnel
before instructions_before  # Optionnel
do instructions_do
between instructions_between  # Optionnel
after instructions_after  # Optionnel
end foreach ;
```

Pour énumérer plusieurs collections, la syntaxe est :

```
foreach sens1 expression1, sens2 expression2, ...
index nom_index # Optionnel
while condition # Optionnel
before instructions_before  # Optionnel
do instructions_do
between instructions_between  # Optionnel
after instructions_after  # Optionnel
end foreach ;
```

Les collections à énumérer sont les valeurs de *expression*, *expression1*, *expression2*. Les types pouvant être énumérés sont listés dans le tableau 36.1. Pour accéder aux valeurs courantes énumérées, à chaque *expression* correspond des constantes implicitement déclarées dont les noms sont indiqués dans la dernière colonne du tableau 36.1. Cette caractéristique peut provoquer des conflits de noms, que l'on résoud en indiquant explicitement un préfixe (voir section 36.11.4 page 187).

Par exemple, pour énumérer une valeur de type `@stringset`, on écrira :

```
@stringset v := ... ;
foreach v do
  log key ; # Affichage des cles dans l'ordre alphabetique
end foreach ;
```

Type	Ordre d'énumération	Constantes déclarées
`@data`	Ordre croissant des indices	`data`, de type `@uint`
`list @T`	Ordre croissant des indices	À chaque champ de la liste, correspond une constante de même nom.
`map @T`	Ordre alphabétique des clés	`lkey`, de type `@lstring`, qui représente la clé, et à chaque champ de la table, correspond une constante de même nom.
`listmap @T`	Ordre alphabétique des clés	`key`, de type `@string`, qui représente la clé, et `mList`, qui représente la liste associée.
`sortedlist @T`	Ordre croissant des indices	À chaque champ de la liste, correspond une constante de même nom.
`@stringset`	Ordre alphabétique	`key`, de type `@string`

Tableau 36.1 – *Types énumérables par l'instruction* `foreach`

36.11.2 Organigramme d'exécution

L'organigramme illustrant l'exécution de l'instruction `foreach` est donné à la figure 36.2.

36.11.3 Champs optionnels

Plusieurs champs de l'instruction `foreach` 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 tableau 36.1 ;
- `<`, dans l'ordre indiqué par le tableau tableau 36.1 ;
- `>`, dans l'ordre inverse à celui indiqué par le tableau tableau 36.1.

`index nom_index`. Vous pouvez mentionner un identificateur après le mot réservé `index`. Cet identificateur est le nom d'une variable 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`. Vous ne pouvez pas vous même changer la valeur de cette variable. Sa visibilité inclut l'ensemble des constructions optionnelles.

`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.

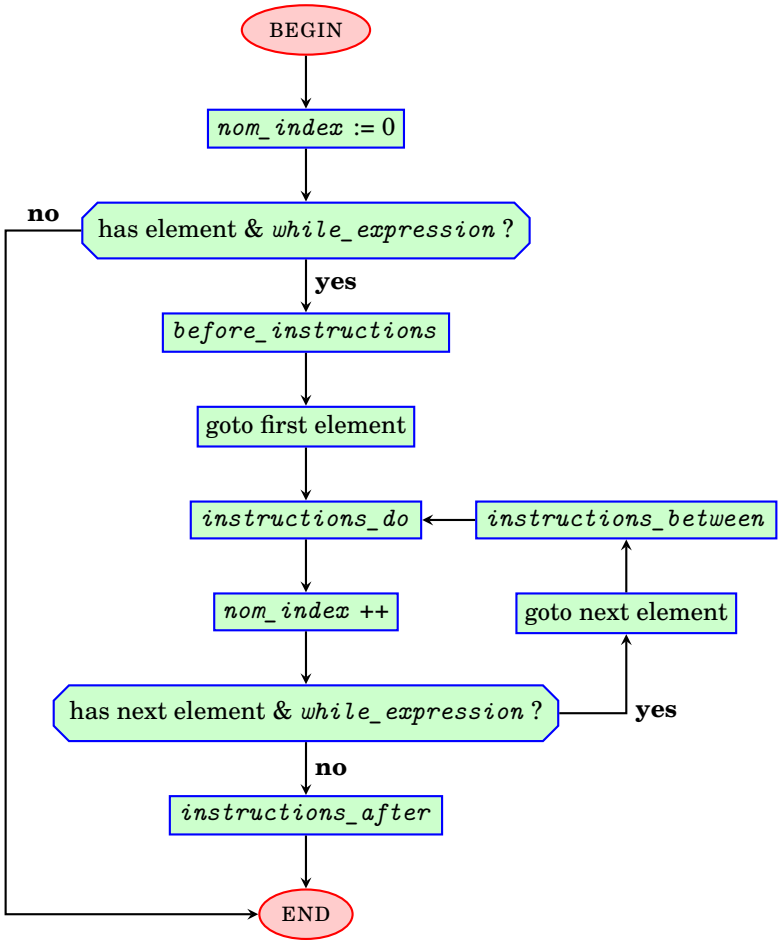

FIGURE 36.2 – *Organigramme d'exécution d'une instruction* `foreach`

36.11.4 Préfixage des constantes

Considérons l'exemple suivant :

```
@stringlist v1 := ... ;
@stringlist v2 := ... ;
foreach v1, v2 do # Erreur !
  ...
end foreach ;
```

Le compilateur GALGAS déclenche une erreur, car il y a ambiguïté sur la signification de `mValue` 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) :

```
@stringlist v1 := ... ;
@stringlist v2 := ... ;
foreach v1, v2 : l2_ do
  ...
end foreach ;
```

La déclaration du préfixe `l2_` signifie que les constantes associées à la seconde liste auront leur nom préfixé par `l2_`. De cette façon, `l2_mValue` désigne la valeur courante de la seconde liste, et `mValue` désigne sans ambiguïté la valeur courante de la première liste.

```
@stringlist v1 := ... ;
@stringlist v2 := ... ;
foreach v1, v2 : l2_ do
  log mValue, l2_mValue ;
end foreach ;
```

36.11.5 Modification de la collection

Au début de l'exécution de l'instruction `foreach`, les valeurs des *expression* 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 [emptyList] ;
v += !"A" ;
v += !"B" ;
v += !"C" ;
log v ; # "A", "B", "C"
foreach v do
  v += !mValue ;
end foreach ;
log v ; # "A", "B", "C", "A", "B", "C"
```

36.12 Increment Instruction

36.13 L'instruction `if`

36.13.1 Syntax

The *if* instruction has the following syntax :

```
if expression then
  instructions
elsif expression2 then
  instructions2
...
else
  else_instructions
end if ;
```

More precisely, it contains :
- zero, one or more *elsif* branches ;
- zero or one *else* branch.

36.13.2 Static semantics

No *else* branch is equivalent to an *else* branch without any instruction.

The *elsif* branches are just syntactic sugar : it is semantically equivalent to use embedded *if* instructions instead. For example :

```
if expression then
  instructions
elsif expression2 then
  instructions2
else
  else_instructions
end if ;
```

is equivalent to :

```
if expression then
  instructions
else
  if expression2 then
    instructions2
  else
    else_instructions
  end if ;
end if ;
```

So, for describing *if* instruction static and dynamic semantics, we only need to describe an *if* instruction without any *elsif* branch and with an *else* branch :

```
if expression then
  instructions
else
  else_instructions
end if ;
```

The static semantics evaluates the *expression* type, and applies the following rules until success :

1. the *expression* type is type `@bool` (page 53) ;
2. the *expression* type is an *structure* type, it has a attribute named *bool*, whose type is type `@bool` (page 53) ;
3. the *expression* type has a reader without any argument named *bool* that returns a type `@bool` (page 53) value.

Most expressions you write fall in the first case.

Applying the second rule enables to use an type `@lbool` (page 151) expression as an *if* expression. For example :

```
@lbool var := ... ;
if var then
  instructions
else
  else_instructions
end if ;
```

The *var* object belongs to the type `@lbool` (page 151) type : so first rule fails. But type `@lbool` (page 151) is a *structure* type, it has a *bool* attribute with the type `@bool` (page 53) type, so the second rule succeeds. It is semantically equivalent to write :

```
@lbool var := ... ;
if var->bool then
  instructions
else
  else_instructions
end if ;
```

The third rule applies on a *class* type that defines a category reader with argument named *bool* that returns a type `@bool` (page 53) type. For example, declaring :

```
class @myClass { ... }

reader @myClass bool -> @bool outResult : ... end reader ;
```

enables to write :

```
@myClass myObject := ... ;
if myObject then
  instructions
else
  else_instructions
end if ;
```

It is semantically equivalent to write :

```
@myClass myObject := ... ;
if [myObject bool] then
  instructions
else
  else_instructions
end if ;
```

36.13.3 Dynamic semantics

According to the preceding section, we only need to describe the dynamic semantic of an *if* instruction without any *elsif* branch and with an *else* branch :

```
if expression then
  instructions
else
  else_instructions
end if ;
```

The *expression* is first computed :

- if the evaluation fails, neither the *if* instructions, neither the *else* instructions are executed ;
- if the evaluation result is *true*, the *if* instructions are executed ;
- if the evaluation result is *false*, the *else* instructions are executed.

36.14 Grammar Instruction

36.15 Local Variable Declaration Instruction

```
@type variable ;

@type variable := expression ;

@type variable [constructor arguments] ;
```

36.16 Local Constant Declaration Instruction

36.17 L'instruction `log`

36.18 L'instruction `loop`

36.18.1 Syntaxe

L'instruction `loop` a la syntaxe suivante :

```
loop variant_expression :
  instructions_1
while expression do
  instructions_2
end loop ;
```

Les *instructions_1* et *instructions_2* sont des listes d'instructions qui peuvent être vides.

36.18.2 Sémantique

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 36.3.

36.19 L'instruction d'appel de méthode

36.20 L'instruction d'appel de modifier

36.21 L'instruction `switch`

L'instruction `switch` est dédiée aux types énumérés. Elle présente la syntaxe suivante :

```
switch expression
when constante, constante, ... :
  liste_instructions
when constante, constante, ... :
  liste_instructions
```

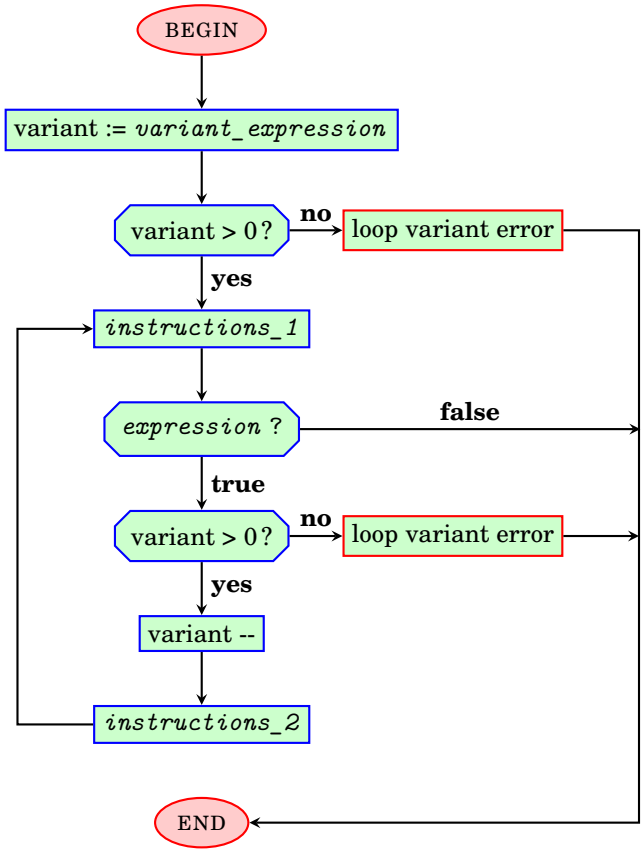

FIGURE 36.3 – *Organigramme d'exécution d'une instruction* `loop`

```
...
end switch ;
```

Où `expression` est une expression d'un type énuméré. Toutes les constantes de ce type doivent être nommées dans les branches `when`, une et une seule fois.

Par exemple, avec la déclaration :

```
enum @feuTricolore {
  vert, orange, rouge
}
```

On peut écrire :

```
@feuTricolore feu := ... ;

switch feu
when vert, orange:
  ...
when rouge :
  ...
end switch ;
```

36.22 Send Instruction

36.23 Warning Instruction

36.24 L'instruction `with`

Chapitre 37

Instructions syntaxiques

37.1 Vérification de l'occurrence d'un terminal

37.2 Appel d'une règle de production

37.3 Instruction `select`

37.4 Instruction `repeat`

37.5 Instruction `parse`

Cinquième partie

Composants

Chapitre 38

Le composant `lexique`

Chapitre 39

Syntax and Grammar Components

39.1 GALGAS and Context-Free Grammars

39.2 Writing a Syntax Component

39.3 Syntax Instructions

39.3.1 Terminal Symbol Instruction

39.3.2 Non Terminal Symbol Instruction

39.3.3 Repeat Instruction

39.3.4 Select Instruction

39.3.5 Parse Instruction

Parse do ... Instruction

Parse loop ... Instruction

Parse when ... Instruction

39.4 Writing a Grammar Component

Chapitre 40

Graphic User Interface Component

Chapitre 41

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 35.1.17 page 179.

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" ;
end  option ;
```

41.1 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 35.1.17 page 179) ;
- `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.

41.2 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`.

41.3 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.

41.4 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 42

Le composant `program`

Chapitre 43

Le composant `project`

Chapitre 44

Cocoa Features

44.1 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 44.2 page 211) ;
- customizing syntax coloring (section 44.3 page 212).

44.2 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 docu-

ment (see http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Documents/Concepts/DocTypePList.html).

44.3 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 44.1 shows the assignment of the terminal which spelling is `begin` by the `mySpecificStyle` style.

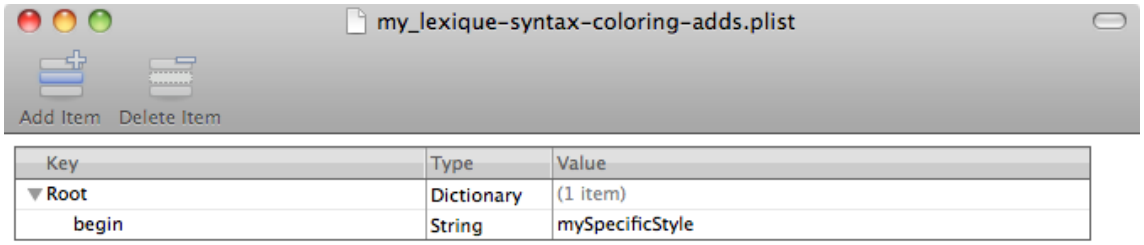

FIGURE *44.1* – *Example of a syntax coloring property list*

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.

44.4 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 44.2). The contextual menu is displayed with a `cmd`-click.

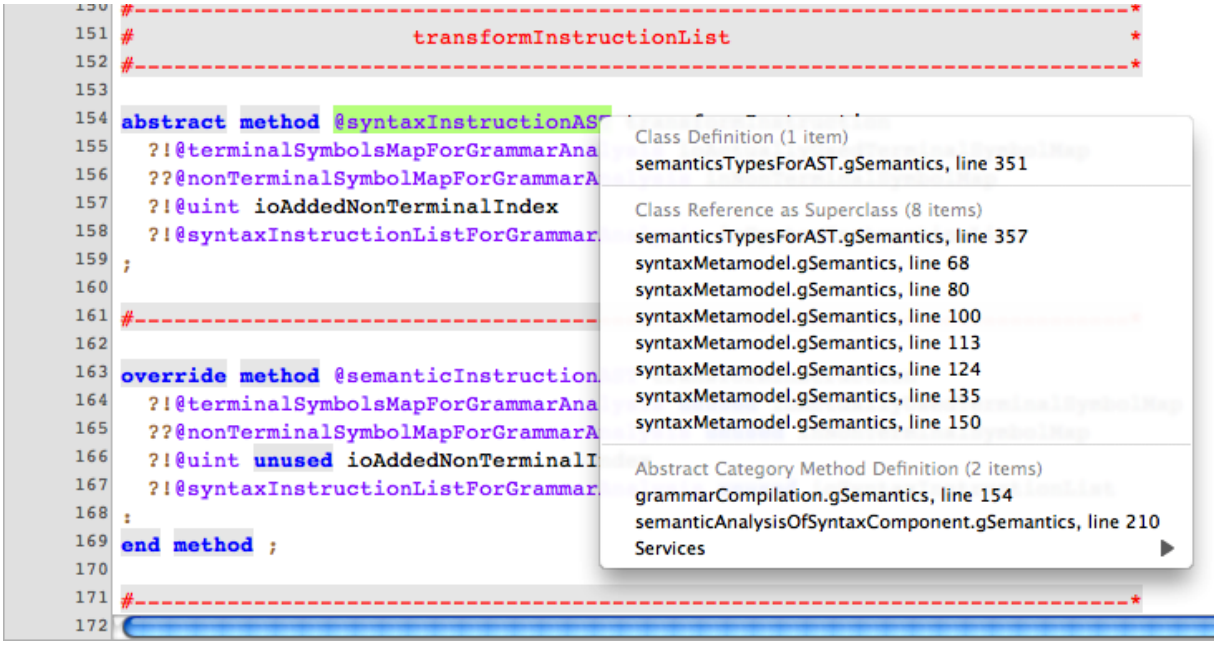

FIGURE *44.2* – *Indexing and cross-referencing in GALGAS Cocoa Application*

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" :
  ...
```

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 44.2, the titles are *Class Definition*, *Class Reference as Superclass* and *Abstract Category Reader 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.

Index