GALGAS 3

Version GALGASBETAVERSION

Jean-Luc Béchennec

Mikaël Briday

Pierre Molinaro

21 décembre 2023

Table des matières

Liste des tableaux

Table des figures

I

Utilisation

Chapitre 1

Tutorial : le langage LOGO

Le but de ce tutorial est de construire en utilisant GALGAS un compilateur d'un langage inspiré de LOGO, qui fournit en sortie un fichier SVG[1] contenant les tracés définis par un programme source LOGO. La génération des fichiers SVG à partir de GALGAS sera faite par un template.

Il est rédigé pour être réalisé sur Unix (Linux, Mac OS X).

1.1 Description rapide de GALGAS

GALGAS est un générateur de compilateur. Vous allez donc écrire l'analyseur lexical du langage LOGO, son analyseur syntaxique, sa sémantique statique, et sa sémantique dynamique (la génération de code). Ces descriptions sont contenues dans des fichiers texte encodés obligatoirement en UTF-8[2]. Il y a deux types de fichiers source GALGAS :

- le fichier *projet*, d'extension `.galgasProject` ;
- les fichiers contenant les descriptions des analyseurs lexicaux, syntaxiques, sémantiques, d'extension `.galgas` .

Le fichier projet est unique dans un projet et référence tous les fichiers d'extension `.galgas` . Aussi la compilation GALGAS s'effectue en compilant le fichier projet. Cette compilation engendre des fichiers C++, et il faut effectuer une compilation C++ pour obtenir un binaire exécutable.

les descriptions des analyseurs lexicaux, syntaxiques, sémantiques sont disposés dans des fichiers d'extension `.galgas` de manière quelconque, c'est-à-dire que l'ordre des descriptions et leur répartition dans les différents fichiers sont indifférentes : par exemple, la déclaration d'un classe et de ses héritières peuvent être dans un même fichier (et dans un ordre quelconque), ou dans des fichiers différents.

[1] Vous trouverez des informations sur le format SVG sur la page : http://www.canarlake.org/index.cgi?theme=svg.
[2] Vous pourrez ainsi utiliser des caractères accentués ou des lettres grecques, cyrilliques, ... dans vos identificateurs.

Pour simplifier la mise en œuvre du développement, l'option `--create-project` du compilateur GALGAS permet de créer une arborescence de fichiers contenant un projet GALGAS prêt à être compilé. Ce projet contient aussi des scripts Python permet de lancer facilement la compilation C++.

Ce tutorial est organisé comme suit :

- la section **??** page **??** présente de manière informelle le langage LOGO ;
- la section **??** page **??** vous guide pour installer GALGAS sur votre ordinateur ;
- la section **??** page **??** décrit comment utiliser l'option `--create-project` pour créer le projet GALGAS, et le compiler et l'exécuter ;
- la section **??** page **??** vous explique comment coder l'analyseur lexical du langage LOGO ;
- la section **??** page **??** pour coder l'analyseur syntaxique du langage LOGO ;
- la section **??** page **??** pour coder la sémantique statique du langage LOGO ;
- la section **??** page **??** pour coder la sémantique statique du langage LOGO ;
- et pour finir, la section **??** page **??** pour coder la génération de code en obtenant un compilateur qui engendre un fichier SVG.

À l'issue de ce tutorial, vous aurez un compilateur complet du langage LOGO qui effectue des vérifications sémantiques et qui engendre un fichier SVG pour chaque source LOGO qui lui est soumis.

1.2 Présentation du langage LOGO

Vous trouverez une description précise du langage à la fin de cette section. Un programme LOGO décrit le déplacement d'une tortue dans un plan. Celle-ci peut effectuer des déplacements en ligne droite et des rotations sur place. La tortue est munie d'un crayon, qui peut être abaissé ou levé. Un déplacement provoque le tracé d'un segment de droite si le crayon est abaissé.

Un programme LOGO est contenu dans un fichier texte, d'extension `.logo`. Il comprend une liste (éventuellement vide) de sous-programmes, et une liste d'instructions. L'exécution du programme consiste à exécuter cette liste d'instructions, en appelant les sous-programmes qui y figurent. Initialement, la tortue est en (0, 0), sa direction est 0° (horizontale, vers la droite), et le crayon est levé.

1.2.1 Quelques exemples

Voici quelques exemples de programmes LOGO (tableau **??** et tableau **??** page **??**). Le tableau **??** page **??** liste des programmes présentant des erreurs sémantiques : le compilateur qui va être écrit décèlera ces erreurs.

Dessin d'un carré

```
PROGRAM
 ROUTINE trace
 BEGIN
 FORWARD 50 ;
 ROTATE 90 ;
 END
BEGIN
 FORWARD 100 ;
 ROTATE 90 ;
 FORWARD 100 ;
 ROTATE 270 ;
 PEN DOWN ;
 CALL trace ;
 CALL trace ;
 CALL trace ;
 CALL trace ;
END.
```

Dessin d'une étoile

```
PROGRAM
 ROUTINE trace
 BEGIN
 FORWARD 70 ;
 ROTATE 160 ;
 END
 ROUTINE trace3
BEGIN
 CALL trace;
 CALL trace;
 CALL trace;
 END
BEGIN
 FORWARD 200;
 ROTATE 90;
 FORWARD 300;
 ROTATE 270;
 PEN DOWN;
 CALL trace3;
 CALL trace3;
 CALL trace3;
END.
```

Dessin d'un pentagone

```
PROGRAM
 ROUTINE trace
 BEGIN
 FORWARD 70 ;
 ROTATE 72 ;
 END
BEGIN
 FORWARD 200;
 ROTATE 90;
 FORWARD 300;
 ROTATE 270;
 PEN DOWN;
 CALL trace;
 CALL trace;
 CALL trace;
END.
```

Tableau 1.1 – *Carré, étoile et pentagone en LOGO*

1.2.2 Définition lexicale

Les identificateurs sont constitués d'une séquence non vide de lettres minuscules ou majuscules. La casse est significative.

Les mots réservés sont les identificateurs suivants : `PROGRAM`, `ROUTINE`, `BEGIN`, `END`, `FORWARD`, `ROTATE`, `PEN`, `UP`, `DOWN` et `CALL`.

Les constantes littérales entières sont écrites en décimal (une séquence non vide de chiffres décimaux).

Les séparateurs sont tous les caractères compris entre le point de code Unicode `'\u0001'` et l'espace (point de code `'\u0020'`), ce qui inclut la tabulation horizontale et les différentes formes de la fin de ligne.

Les délimiteurs sont le point (« `.` ») et le point virgule (« `;` »).

Un commentaire commence par le caractère dièse « `#` ») et s'étend jusqu'à la fin de la ligne courante.

1.2.3 Définition syntaxique

Un programme LOGO commence le mot réservé `PROGRAM`, est suivi d'une liste éventuellement vide de définition de routines, du mot réservé `BEGIN`, d'une liste éventuellement vide d'instructions, et se termine par le mot réservé `END` suivi d'un point.

Dessin d'un hexagone

```
PROGRAM
 ROUTINE trace
 BEGIN
 FORWARD 70 ;
 ROTATE 60 ;
 END
BEGIN
 FORWARD 100 ;
 ROTATE 90 ;
 FORWARD 100 ;
 ROTATE 270 ;
 PEN DOWN ;
 CALL trace ;
 CALL trace ;
 CALL trace ;
 CALL trace ;
 CALL trace ;
 CALL trace ;
END.
```

Dessin d'un octogone

```
PROGRAM
 ROUTINE trace
 BEGIN
 FORWARD 70 ;
 ROTATE 45 ;
 END
 ROUTINE trace1
 BEGIN
 CALL trace;
 CALL trace;
 END
 ROUTINE trace2
 BEGIN
 CALL trace1;
 CALL trace1;
 END
 ROUTINE trace3
 BEGIN
 CALL trace2;
 CALL trace2;
 END
BEGIN
 FORWARD 100 ;
 ROTATE 90 ;
 FORWARD 100 ;
 ROTATE 270 ;
 PEN DOWN ;
 CALL trace3 ;
END.
```

Tableau 1.2 – *Hexagone et octogone en LOGO*

Une définition de routine est introduite par le mot réservé `ROUTINE`, est suivi d'un identificateur, du mot réservé `BEGIN`, d'une liste éventuellement vide d'instructions, et se termine par le mot réservé `END`.

Une instruction LOGO est une des séquences suivantes :

- le mot réservé `FORWARD` suivi d'un entier littéral et d'un point virgule ;
- le mot réservé `ROTATE` suivi d'un entier littéral et d'un point virgule ;
- le mot réservé `PEN` suivi du mot réservé `UP` et d'un point virgule ;
- le mot réservé `PEN` suivi du mot réservé `DOWN` et d'un point virgule ;
- le mot réservé `CALL` suivi d'un identificateur et d'un point virgule.

Routine récursive

```
PROGRAM
 ROUTINE routine
 BEGIN
 CALL routine ;
 END
BEGIN
END.
```

Routine indéfinie

```
PROGRAM
BEGIN
 CALL routine;
END.
```

Routine définie plusieurs fois

```
PROGRAM
 ROUTINE routine
 BEGIN
 END
 ROUTINE routine
 BEGIN
 END
BEGIN
END.
```

Tableau 1.3 – *Programmes LOGO contenant des erreurs sémantiques*

1.2.4 Sémantique statique

Dans une définition de routine, l'identificateur qui suit le mot réservé `ROUTINE` est le nom de la routine définie. Dans une instruction `CALL`, l'identificateur est le nom de la routine appelée.

Contraintes (voir le tableau **??** pour des exemples de programmes contenant des erreurs sémantiques) :

- le nom d'une routine est unique (on n'a pas le droit de définir plusieurs routines de même nom) ;
- une instruction `CALL` ne peut nommer qu'une routine qui a été définie plus haut dans le texte ;
- la routine courante ne peut pas être appelée récursivement.

1.2.5 Sémantique dynamique

L'espace de déplacement de la tortue est un plan, muni du repère orthonormé direct habituel. À un instant donné, l'état de la tortue est caractérisé par :

- sa position dans le plan ;
- sa direction, mesuré en degrés à partir de l'axe horizontal, et dans le sens trigonométrique ;
- la position du crayon (levé ou abaissé).

Initialement, la position de la tortue est (0, 0), sa direction est 0°, et le crayon est levé.

L'exécution de chaque instruction a l'effet suivant :

- l'instruction `FORWARD` avance la souris dans sa direction d'une longueur égale à la valeur de la constante entière contenue dans l'instruction ; si le crayon est abaissé, un segment de droite délimité par les positions de départ et d'arrivée de la tortue est tracé ;
- l'instruction `ROTATE` fait tourner la tortue dans le sens trigonométrique d'un nombre de degrés égal à la constante contenue dans l'instruction ; aucun tracé n'a lieu, quelque l'état du crayon.
- l'instruction `PEN UP` relève le crayon ;

- l'instruction `PEN DOWN` abaisse le crayon ;
- l'instruction `CALL` exécute le sous-programme nommé dans l'instruction.

1.3 Installation de GALGAS

Aller sur la page http://galgas.rts-software.org/download/.

GALGAS est un utilitaire en ligne de commande (sauf sur Mac, pour lequel une application Cocoa est disponible). Vous pouvez :

- soit télécharger le binaire correspondant à votre plateforme (pour l'installer, aller à la section **??** page **??**) ;
- soit télécharger les sources et les recompiler.

1.3.1 Téléchargement des sources et compilation

Télécharger l'archive contenant les sources pour Unix et Mac.

Décompressez cette archive et placer le répertoire obtenu (galgas) dans un répertoire dont le chemin ne contient ni espace ni caractère accentué. C'est important car les chemins utilisés dans les makefile de GALGAS sont relatifs.

Dans la suite de la compilation GALGAS, tous les chemins sont indiqués relativement à ce répertoire, qui sera appelé `constructionGALGAS`.

Donc, vous devez obtenir à la suite de la décompression le répertoire `constructionGALGAS/galgas`.

Nous allons maintenant compiler GALGAS. Avec le terminal, sur Linux :

```
cd constructionGALGAS/galgas/makefile-unix
./build.py
```

Sur Mac :

```
cd constructionGALGAS/galgas/makefile-macosx
./build.py
```

La compilation de GALGAS peut prendre une dizaine de minutes. Deux exécutables sont produits :

- `constructionGALGAS/galgas/makefile-unix/galgas` ;
- `constructionGALGAS/galgas/makefile-unix/galgas-debug`.

Les deux exécutables sont fonctionnellement identiques. Le premier est celui que vous utiliserez. Le second est la version debug du premier : il est exécuté avec de nombreuses vérifications internes, ce qui fait qu'il

est beaucoup lent. Si le premier plante brutalement, on peut utiliser le second pour déceler si une erreur interne peut être mise en évidence.

La section suivante indique comment installer les binaires obtenus.

1.3.2 Installation

Pour pouvoir appeler les exécutables sans avoir besoin de mentionner un chemin, vous avez plusieurs possibilités :

- le copier dans le répertoire `/bin` : `sudo cp galgas /bin/`
- le copier dans votre répertoire local `bin` : `cp galgas ~/bin/`

Attention, le répertoire `~/bin` n'existe peut-être pas pour votre compte : il faut alors le créer, et l'ajouter dans la variable `$PATH`.

Sur Linux :

```
mkdir ~/bin/
echo 'export PATH=$PATH:~/bin' >> /home/user/.bashrc
```

Sur Mac :

```
mkdir ~/bin/
echo 'export PATH=$PATH:~/bin' >> ~/.bash_profile
```

1.4 Création du squelette du compilateur LOGO

Un projet GALGAS nécessite la mise en place de nombreux fichiers, de créer des makefile pour différentes plateformes, ...

Appeler galgas avec l'option `--create-project` permet de créer automatique un projet prêt à être utilisé.

Pour tout le tutorial vous devez utiliser un répertoire dont le chemin ne contient ni espace ni caractère accentué. C'est important car les chemins utilisés dans les makefile de GALGAS sont relatifs.

Dans toute la suite de ce tutorial, les chemins sont indiqués relativement à ce répertoire, qui sera appelé `chezmoi`.

La création (Unix) :

```
cd chezmoi
galgas --create-project=logo
```

La création (Windows) :

Fichier	Description
`logo-lexique.galgas`	Définit l'analyseur lexical
`logo-semantics.galgas`	Définit les types pour la sémantique
`logo-syntax.galgas`	Définit les règles de production de la grammaire
`logo-grammar.galgas`	Définit la grammaire (axiome, classe)
`logo-program.galgas`	Définit la routine principale
`logo-cocoa.galgas`	Définit l'interface pour Cocoa : utile uniquement sous Mac
`logo-options.galgas`	Définit les options de la ligne de commande

Tableau 1.4 – *Contenu du répertoire `Logo/galgas-sources`*

```
cd chezmoi
galgas --no-dialog --create-project=logo
```

Sous Windows, si aucun fichier source à compiler n'est indiqué dans la ligne de commande, GALGAS affiche un dialogue proposant d'entrer ce fichier. L'option `--no-dialog` (section **??** page **??**) permet de ne pas faire apparaître ce dialogue.

Le message affiché par cette opération est :

```
*** PERFORM PROJECT CREATION (--create-project=logo option) ***
*** DONE ***
```

L'affichage de `DONE` indique que la création s'est effectuée avec succès : un répertoire nommé `logo` a été créé dans le répertoire `chezmoi`.

1.4.1 Visite guidée du répertoire créé

Dans le répertoire `chezmoi/logo` :

- le fichier `+logo.galgasProject` est le fichier projet, c'est lui que vous compilerez ;
- le répertoire `galgas-sources` contient les fichiers sources que vous allez compléter tout au long de ce tutorial ; son contenu est indiqué dans le tableau **??**.

1.4.2 Première compilation du projet

Une compilation s'effectue en deux temps :

1. d'abord une compilation GALGAS qui crée ou met à jour des fichiers C++ ;
2. ensuite une compilation C++.

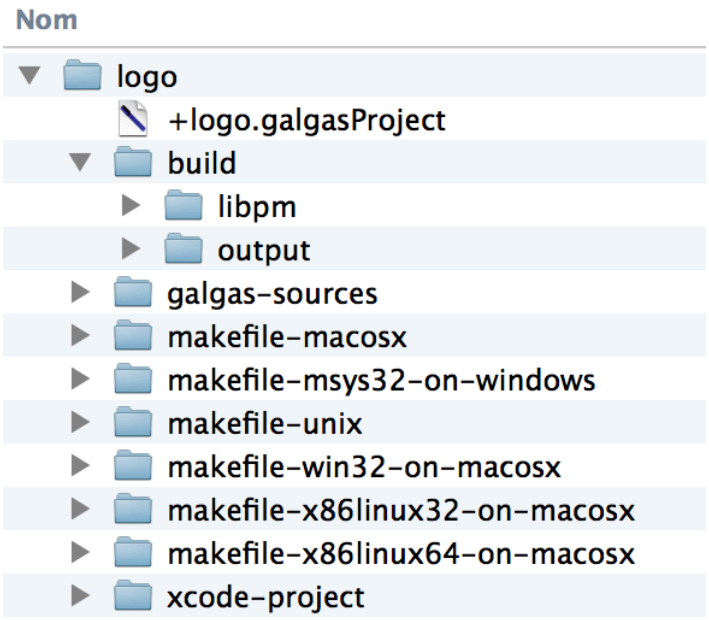


Figure 1.1 – *Répertoire Logo après compilation GALGAS*

Répertoire	**Contenu**
`makefile-macosx`	Makefile pour compiler sur Mac OS X
`makefile-unix`	Makefile pour compiler sur Unix
`makefile-win32-on-macosx`	Makefile pour compiler sur Mac OS X pour Win 32
`makefile-x86linux32-on-macosx`	Makefile pour compiler sur Mac OS X pour x86 Linux 32 bits
`makefile-x86linux64-on-macosx`	Makefile pour compiler sur Mac OS X pour x86 Linux 64 bits
`xcode-project`	Projet Xcode pour compiler sur Mac OS X

Tableau 1.5 – *Contenu des sous-répertoires de Logo après compilation GALGAS*

1.4.2.1 Compilation GALGAS

Vous devez d'abord compiler les sources GALGAS :

```
galgas chezmoi/logo/+logo.galgasProject
```

L'exécution provoque l'affichage de messages : observez ceux qui indiquent la création des fichiers C++. Ceux-ci sont rangés dans le répertoire `chezmoi/logo/build/output` et `chezmoi/logo/build/libpm`.

Le répertoire `logo` est complété par de nouveaux répertoires (figure **??** et tableau **??**).

1.4.2.2 Compilation C++

Choisissez le répertoire correspondant à votre plateforme (`makefile-macosx` ou `makefile-unix`) et lancer le script de compilation `build.py` (soit via la ligne de commande, soit en double-cliquant).

Par exemple :

```
chezmoi/logo/makefile-unix/build.py
```

Vous obtenez deux exécutables :

```
chezmoi/logo/makefile-unix/logo
chezmoi/logo/makefile-unix/logo-debug
```

Sous Mac, vous pouvez utiliser le projet Xcode engendré, et ainsi créer une application Cocoa nommée `CocoaLogo`.

Dans tous les cas, comme les analyseurs lexicaux et syntaxiques sont vides après la création, les exécutables ainsi obtenus ne sont pas exploitables.

1.5 Analyseur lexical

Dans cette partie, vous allez écrire l'analyseur lexical du langage LOGO. Pour cela, vous allez modifier le fichier `chezmoi/logo/galgas-sources/logo-lexique.galgas`.

Remarques préliminaires :

1. en GALGAS, tous les symboles terminaux sont notés par une chaîne de caractères non vide délimitée par deux caractères `$` ; par exemple : `$identifier$`, `$integer$`, ...
2. en GALGAS, un nom de type est un identificateur précédé du caractère `@` ; par exemple : `@string`, `@uint`, `@lstring`, `@luint`, ...;
3. le type `@string` définit une valeur chaîne de caractères ;
4. le type `@uint` définit une valeur entière non signée sur 32 bits ;
5. le type `@lstring` définit une valeur composée d'une chaîne de caractères et d'une information de localisation sur la position de la chaîne dans le texte source ;
6. le type `@luint` définit une valeur composée d'une valeur entière non signée et d'une information de localisation sur la position de la chaîne dans le texte source ;
7. ces informations de localisation sont à la base du signalement d'erreur.

1.5.1 Analyse lexicale d'un identificateur et d'un mot réservé

Par défaut, une analyse lexicale des identificateurs et une liste de mots réservés est présente. Tout ce que vous avez à faire est de modifier la liste des mots réservés pour y placer ceux du langage LOGO.

Voici les lignes correspondantes :

```
@string tokenString
style keywordsStyle -> "Keywords"
$identifier$ ! tokenString error message "an identifier"

list keyWordList style keywordsStyle error message "the '%K' keyword" {
  "begin",
  "end"
}

rule 'a'->'z' | 'A'->'Z' {
  repeat
    enterCharacterIntoString (!?tokenString !*)
  while 'a'->'z' | 'A'->'Z' | '_' | '0'->'9' :
  end
  send search tokenString in keyWordList default $identifier$
}
```

Explications :

1. `@string tokenString` déclare l'attribut lexical tokenString de type chaîne de caractères ; au début de l'analyse de chaque token, cet attribut est initialisé à la valeur chaîne vide ;
2. `style keywordsStyle -> "Keywords"` déclare un style (uniquement utile pour l'application Cocoa engendrée, vous pouvez ignorer cette ligne) ;
3. `$identifier$ ! tokenString error message "an identifier"` déclare le terminal `$identifier$` qui sera transmis à l'analyseur syntaxique accompagné de la valeur de `tokenString` ; le message d'erreur qui suit est celui qui est utilisé lors d'une erreur syntaxique ;
4. `list keyWordList style keywordsStyle error ...` déclare une liste de mots réservés associés à un style d'affichage (pour l'application Cocoa sur Mac), un message d'erreur syntaxique ; telle qu'elle est présente, cette définition déclare les deux terminaux `$begin$` et `$end$` ;
5. enfin, `rule 'a'->'z' | 'A'->'Z' ...` effectue l'analyse lexicale des identificateurs en accumulant dans `tokenString` les caractères rencontrés ; la recherche d'un mot réservé est effectuée par `send search tokenString in keyWordList default $identifier$` : par défaut si la chaîne entrée n'est pas un mot réservé, un identificateur est retourné à l'analyseur syntaxique.

Travail à faire. Modifier la liste des mots réservés en y plaçant ceux du langage LOGO.

1.5.2 Analyse lexicale d'une constante entière

L'analyse lexicale d'une constante entière 32 bits non signée est présente par défaut, vous n'avez rien à ajouter.

Voici l'écriture correspondante :

```
style integerStyle -> "Integer Constants"
@uint uint32value
$integer$ !uint32value style integerStyle
                error message "a 32-bit unsigned decimal number"

message decimalNumberTooLarge : "decimal number too large"
message internalError : "internal error"

rule '0'->'9' {
  enterCharacterIntoString (!?tokenString !*)
  repeat
  while '0'->'9' :
    enterCharacterIntoString (!?tokenString !*)
  while '_' :
  end
  convertDecimalStringIntoUInt (
    !tokenString
    !?uint32value
    error decimalNumberTooLarge, internalError
  )
  send $integer$
}
```

Explications :

1. `style integerStyle -> "Integer Constants"` déclare un style (uniquement utile pour l'application Cocoa engendrée, vous pouvez ignorer cette ligne) ;

2. `@uint uint32value` déclare l'attribut lexical `uint32value` de type entier 32 bits non signé ; au début de l'analyse de chaque token, cet attribut est initialisé à la valeur zéro ;

3. `$integer$ !uint32value style integerStyle ...` déclare le terminal `$integer$` qui sera transmis à l'analyseur syntaxique accompagné de la valeur de `uint32value` ; le message d'erreur qui suit est celui qui est utilisé lors d'une erreur syntaxique ;

4. `message decimalNumberTooLarge~: "decimal number too large"` déclare un message d'erreur ;

5. enfin `rule '0'->'9' ...` définit l'analyse lexicale d'une contante entière non signé ; les caractères qui la composent sont accumulés dans `tokenString` , et la conversion de cette chaîne en entier est effectuée par la routine `convertDecimalStringIntoUInt` ; pour finir, `send $integer$` envoie le terminal vers l'analyseur syntaxique.

1.5.3 Analyse des délimiteurs

Par défaut, un certain nombre de délimiteurs sont définis :

```
style delimitersStyle -> "Delimiters"
list delimitorsList style delimitersStyle error message "the '%K' delimitor" {
  ":",    ",",    ";",   "!",  "{",  "}", "->", "@", "*", "-"
}

rule list delimitorsList
```

La règle `list delimitorsList style delimitersStyle error ...` déclare les terminaux `$:$` , `$,$` ... Les messages d'erreur syntaxique sont définis en remplaçant la séquence `%K` par l'épellation du délimiteur.

L'analyse des délimiteurs est définit par la règle `rule list delimitorsList` .

Travail à faire : remplacer la liste des délimiteurs par celle du langage LOGO.

1.5.4 Analyse des chaînes de caractères

Une analyse des chaînes de caractères est disponible par défaut. Comme le langage LOGO n'utilise pas de chaînes de caractères, vous pouvez supprimer les définitions suivantes :

```
style stringStyle -> "String Constants"
$literal_string$ ! tokenString style stringStyle %nonAtomicSelection
                    error message "a character string constant \"...\""

message incorrectStringEnd : "string does not end with '\"'"

rule '"' {
  repeat
  while ' ' | '!' | '#'-> '\uFFFD' :
    enterCharacterIntoString (!?tokenString !*)
  end
  select
  case '"' :
    send $literal_string$
  default
    error incorrectStringEnd
  end
}
```

1.5.5 Analyse des séparateurs

C'est une règle très simple, qui accepte tout caractère de code ASCII compris entre `0x01` et `0x20` (l'espace). Comme il n'y a pas d'instruction send dans la règle lexicale, l'occurrence d'un séparateur est complètement ignorée par l'analyseur syntaxique.

```
rule '\u0001' -> ' ' {
}
```

La séquence d'échappement `\u` permet d'écrire directement des points de code Unicode sous la forme de quatre chiffres hexadécimaux.

1.5.6 Analyse des commentaires

C'est un peu plus compliqué, il faut repérer la fin de la ligne courante. Or, celle-ci peut être un seul caractère `LF` (fichier Unix), un seul caractère `CR` (fichier Mac Classic), une séquence `CRLF` (fichier Windows). D'autre part, une ligne de commentaire peut être la dernière ligne du fichier : notez que GALGAS rajoute automatiquement le caractère `'\0'` à la fin de la chaîne source. L'analyse d'un commentaire consiste donc, une fois le caractère initial `'\#'` repéré, à accepter silencieusement tous les caractères possibles, sauf `'\u000A'` (LF), `'\u000D'` (CR), `'\0'`. L'écriture `drop $comment$` signifie que le terminal `$comment$` n'est pas transmis à l'analyseur syntaxique.

```
style commentStyle -> "Comments"
$comment$ style commentStyle %nonAtomicSelection error message "a comment"
rule '#' {
  repeat
  while '\u0001'->'\u0009' | '\u000B' | '\u000C' | '\u000E'->'\uFFFD':
  end
  drop $comment$
}
```

Remarquez que pour un fichier Windows, le caractère `CR` marque la fin du commentaire, et que le caractère `LF` qui suit est silencieusement absorbé comme délimiteur.

Travail à faire : effectuer la compilation GALGAS, puis la compilation C++ ; les exécutables `logo` et `logo-debug` obtenus sont alors partiellement opérationnels (pas encore d'analyseur syntaxique) : avec l'option `--mode=lexical-only`, vous pouvez faire afficher la liste des symboles terminaux produite par l'analyse lexicale du fichier source passé en argument.

Note : l'option `--help` permet d'afficher la liste des options de l'exécutable.

1.6 Analyseur syntaxique

Deux fichiers sont concernés :

Chaîne	Analyse effectuée
`"LL1"`	Analyse LL (1) de la grammaire ; échoue si la grammaire n'est pas LL (1)
`"SLR"`	Analyse SLR de la grammaire ; échoue si la grammaire n'est pas SLR
`"LR1"`	Analyse LR (1) de la grammaire ; échoue si la grammaire n'est pas LR (1)
`""`	Analyse LL (1) ; en cas d'échec, analyse SLR ; en cas de nouvel échec, analyse LR (1)

Tableau 1.6 – *Spécification de l'analyse de la grammaire*

- `chezmoi/logo/galgas-sources/logo-syntax.galgas`, et
- `chezmoi/logo/galgas-sources/logo-grammar.galgas`.

Le fichier `logo-syntax.galgas` contient une liste de règles de production. Le fichier `logo-grammar.galgas` définit une grammaire.

Le fichier `logo-grammar.galgas` a la composition suivante :

```
grammar logo_grammar "LL1" {
  syntax logo_syntax
  <start_symbol>
}
```

Explications :

1. `"LL1"` est la classe de la grammaire ;
2. `syntax logo_syntax` : les règles de productions sont dans le composant syntaxique `logo_syntax`, situé dans le fichier `logo-syntax.galgas` ;
3. `<start_symbol>` : l'axiome de la grammaire.

A priori, vous n'avez pas besoin de modifier le fichier `logo-grammar.galgas` au cours de ce tutorial. Vous pouvez cependant modifier l'analyse effectuée en suivant les indications du tableau **??** page **??**[3].

Par défaut dans le fichier `logo_syntax.galgas`, seul le non terminal `<start_symbol>` est déclaré, et une règle de production vide est écrite.

C'est à vous d'écrire les règles de production qui définissent le langage LOGO (voir sa définition syntaxique section **??** page **??**).

Voici les indications qui vous permettront d'écrire ces règles :

- vous pouvez déclarer autant de non terminaux que vous voulez ;
- la forme d'une règle de production est : `rule <mon_non_terminal> { partie droite }`
- la partie droite est une séquence éventuellement vide de :
  - terminaux ;

[3] Rappel : toute grammaire LL(1) est SLR, toute grammaire SLR est LR(1).

  - non-terminaux ;
  - d'instructions de répétition syntaxique (section **??** page **??**) ;
  - d'instruction de sélection syntaxique (section **??** page **??**).

- les règles de production peuvent apparaître dans un ordre quelquonque.

Pour vous aider, voici une écriture possible de la dérivation de l'axiome :

```
rule <start_symbol> {
#-- Définition des routines
  $PROGRAM$
  repeat
  while
    <routine_definition>
  end
#--- Programme principal
  $BEGIN$
  <instruction_list>
  $END$
  $.$
}
```

Et la règle de production `<routine_definition>` :

```
rule <routine_definition> {
  $ROUTINE$
  $identifier$ ?*
  $BEGIN$
  <instruction_list>
  $END$
}
```

Noter l'écriture `$identifier$ ?*` : en effet, quand l'analyseur lexical envoie vers l'analyseur syntaxique le token `$identifier$`, celui-ci est accompagné d'une chaîne de caractères. On indique que la valeur de celle-ci n'est pas utilisée (pour le moment) par l'écriture `?*`.

Il en est de même pour le token `$integer$` qui est accompagné d'une valeur entière.

Si des erreurs d'analyse de la grammaire surviennent, vous pouvez ajouter dans la ligne de commande l'option `--output-html-grammar` : celle-ci provoquera la génération du fichier `chezmoi/logo/build/helpers/logo_grammar.html` qui contient tous les détails de l'analyse de la grammaire.

À l'issue de ce travail, l'exécutable obtenu doit analyser correctement les programmes LOGO cités en exemple (tableau **??** page **??**, tableau **??** page **??** et tableau **??** page **??**. Comme l'analyse sémantique n'est pas encore écrite, les erreurs sémantiques ne sont pas détectées.

Vous pouvez utiliser l'option `--mode=syntax-only` pour afficher la trace de l'analyse syntaxique.

1.7 Sémantique statique

Le but de cette étape est d'enrichir les fichiers GALGAS de façon à vérifier la sémantique statique du langage LOGO (section **??** page **??**).

1.7.1 Préliminaire : obtenir la valeur des identificateurs

Dans l'analyseur syntaxique, pour chaque occurrence du token `$identifier$`, nous avons précédemment écrit `$identifier$ ?*` pour signifier que la valeur de la chaîne de caractères n'était pas utilisée.

À partir de maintenant, nous avons besoin de cette valeur. Celle-ci est récupérée en écrivant :

```
$identifier$ ?let @lstring unNom
```

Cette écriture déclare une constante locale, nommée `unNom`, de type `@lstring`.

Notez que le type mentionné est `@lstring`, alors que dans l'analyseur lexical une valeur de type `@string` est associée au terminal `$identifier$`. Le type `@lstring` est une structure composée d'une valeur de type `@string` et d'une valeur de type `@location`. Cette dernière désigne un point dans le texte source analysé. Lors de la transmission des informations de l'analyseur lexical vers l'analyseur syntaxique, la valeur de type `@string` est associée à la position de l'identificateur dans le texte source. Ceci permet de construire facilement des messages d'erreur qui désigne l'endroit dans le texte source où une erreur est apparue.

Pour le moment, on ne modifie pas les terminaux `$integer$`.

Faire les modifications et recompiler. Comme les valeurs récupérées ne sont pas utilisées et perdues, vous obtenez un *warning* pour chaque constante.

Vous pouvez afficher la valeur obtenue en ajoutant une instruction `log` à chacune des séquences précédentes :

```
$identifier$ ?let @lstring unNom
log unNom
```

L'instruction `log` affiche la valeur d'une variable ou d'une constante. Elle est utilisable sur tous les types GALGAS.

1.7.2 Principes d'écriture de la sémantique

Le cadre général est celui des grammaires attribuées. Ceci revient à doter de paramètres formels les non terminaux de la partie gauche d'une règle, de la même façon que la définition d'une fonction C peut présenter des paramètres formels. En conséquence, un non-terminal apparaissant en partie droite d'une règle de production doit présenter des arguments effectifs, de la même façon qu'un appel de fonction doit citer des arguments effectifs en accord avec la déclaration du prototype de la fonction. Dès lors, vous pouvez établir les correspondances listées dans le tableau **??** page **??**.

En C	En GALGAS
Le prototype d'une fonction cite la liste des arguments formels	La déclaration d'un non-terminal cite la liste des attributs (au sens des grammaires attribuées)
L'en tête de l'implémentation d'une fonction cite la liste des arguments formels	Le non terminal de gauche d'une règle de production cite la liste des attributs (au sens des grammaires attribuées)
L'appel d'une fonction cite des paramètres effectifs	Un non terminal apparaissant dans la partie droite d'une règle de production cite une liste des attributs (au sens des grammaires attribuées)

Tableau 1.7 – *Arguments formels, paramètres effectifs en C et en GALGAS*

Délimiteur	Sens de transmission
`?`	Entrée
`?let`	Entrée constant
`!`	Sortie
`?!`	Entrée/sortie

Tableau 1.8 – *Sens de transmission d'un argument formel*

En GALGAS, nous utilisons plutôt le vocabulaire des langages de programmation : *argument formel*, *paramètre effectif*.

1.7.2.1 Arguments formels en GALGAS

Un argument formel cite :

- un délimiteur qui précise le sens de transmission de l'argument formel ;
- son type (par exemple `@lstring` , `@luint` , ...) ;
- son nom.

Le sens de transmission d'un argument formel est défini dans le tableau **??** page **??**.

1.7.2.2 Paramètres effectifs en GALGAS

Un paramètre effectif cite :

- un délimiteur qui précise le sens de transmission du paramètre effectif ;
- une variable locale ou un argument formel de la règle de production.

Le sens de transmission d'un paramètre effectif est défini dans le tableau **??** page **??**.

Délimiteur	Sens de transmission	Argument formel correspondant
`?`	Entrée	`!` (argument formel en sortie)
`!`	Sortie	`?` (argument formel en entrée) ou `?let` (argument formel en entrée constant)
`!?`	Sortie/entrée	`?!` (argument formel en entrée/sortie)

Tableau 1.9 – *Sens de transmission d'un paramètre effectif*

1.7.2.3 Les types en GALGAS

Il existe plusieurs sortes de types :

- les types prédéfinis par le langage, comme `@lstring`, `@luint`, ... ;
- les types définis par l'utilisateur, qui peuvent être :
  - des types *table* ;
  - des types *liste* ;
  - des types *classe*.

1.7.3 Écriture de la sémantique statique

Pour décrire la sémantique statique (section **??** page **??**), le plus simple est de créer un type table de symboles, dont une instance contiendra tous les noms de routines d'un programme LOGO.

1.7.3.1 Ajout du type de table des routines

éditez le fichier `chezmoi/logo/galgas-sources/logo-semantics.galgas` et ajouter la définition suivante :

```
map @routineMap {
  insert insertKey error message "the '%K' routine has been already declared"
  search searchKey error message "the '%K' routine is not declared"
}
```

Ceci déclare le type `@routineMap`, avec une méthode d'insertion `insertKey` accompagnée de son message d'erreur, et une méthode de recherche `searchKey` accompagnée de son message d'erreur. Implicitement, la clé de la table est du type `@lstring`.

Cette définition sera complétée dans l'étape suivante afin de prendre en compte les instructions des routines (on n'en a pas besoin pour le moment).

À cet instant, vous pouvez recompiler le fichier `logo-semantics.galgas`.

Instructions sur les objets de type table Voici quatre instructions relatives aux tables dont vous allez avoir besoin :

- la déclaration d'un objet de type table ;
- l'initialisation d'un objet de type table ;
- l'instruction d'insertion dans une table ;
- l'instruction de recherche dans une table.

La déclaration d'un objet de type table se fait simplement en nommant le type puis l'objet ; par exemple :

```
@routineMap maTable
```

L'initialisation d'un objet de type table s'effectue en créant une table vide :

```
maTable = {}
```

Les deux écritures précédentes peuvent être condensées en une seule par :

```
@routineMap maTable = {}
```

L'instruction d'insertion dans une table est :

```
[!?maTable insertKey !clef]
```

où `insertKey` est le nom d'une méthode d'insertion déclarée dans le type table ; clef doit être une variable de type `@lstring` valuée. Si il existe déjà une entrée de même nom, le message d'erreur associé à la méthode d'insertion est affiché.

L'instruction de recherche dans une table est :

```
[maTable searchKey !clef]
```

où `searchKey` est le nom d'une méthode de recherche déclarée dans le type table ; `clef` doit être une variable de type `@lstring` valuée. Si il n'existe pas d'entrée de même nom, le message d'erreur associé à la méthode de recherche est affiché.

1.7.3.2 Ajout de la sémantique dans les règles de productions

Éditer le fichier `chezmoi/logo/galgas-sources/logo-syntax.galgas` et modifiez la dérivation de l'axiome :

```
rule <start_symbol> {
  $PROGRAM$
  @routineMap tableRoutines = {}
  repeat
  while
    <routine_definition> !?tableRoutines
  end
  $BEGIN$
  <instruction_list>
  $END$
```

```
  $.$
}
```

L'appel du non terminal `<routine_definition>` impose que son en-tête doit être modifiée en conséquence :

```
rule <routine_definition> ?!@routineMap ioTableRoutines {
  ...
}
```

1.7.3.3 Travail à faire

Maintenant, à vous de compléter les règles de façon à prendre en compte toutes les contraintes édictées à la section **??** page **??**.

Vérifiez que votre analyseur détecte correctement les erreurs. Pour cela, vous pouvez utiliser les exemples du tableau **??** page **??**.

1.8 Sémantique dynamique

Dans la sémantique dynamique (section **??** page **??**), nous allons prendre en compte la signification de l'exécution d'une instruction. En préliminaire, nous allons compléter l'analyseur lexical pour qu'il envoie la valeur d'une constante entière.

1.8.1 Préliminaire : les constantes entières

Modifier maintenant l'analyse syntaxique des constantes entières, à l'image de ce qui a été fait pour les identificateurs :

```
$integer$ ?let @luint unNom
```

Le type `@luint` est une structure composée d'une valeur de type `@uint` et d'une valeur de type `@location`.

1.8.2 Mise à plat de la liste des instructions

Le but ultime est d'obtenir la liste des instructions du programme principal. Mais quelles sont les instructions qui devront apparaître dans cette liste ? A priori, toutes les instructions décrites dans la section **??** page **??**. En fait, vous pouvez vous passer de l'instruction `CALL` en insérant dans la liste des instructions non pas cette instruction, mais la liste des instructions de la routine correspondante. Il faut procéder de même lors de construction de la liste de chaque routine.

1.8.3 Hiérarchie des classes des instructions

Une solution classique pour ce type de situation est de définir une classe abstraite `@instruction`, et des classes concrètes `@penUp`, `@penDown`, `@rotate` et `@forward` qui héritent de cette classe abstraite.

Éditez le fichier `chezmoi/logo/galgas-sources/logo-semantics.galgas` et insérer le texte suivant (n'importe où, dans n'importe quel ordre, GALGAS est indifférent à l'ordre des déclarations) :

```
abstract class @instruction {
}
class @penUp : @instruction {
}
class @penDown : @instruction {
}
class @forward : @instruction {
  @luint mLength
}
class @rotate : @instruction {
  @luint mAngle
}
```

Les trois premières classes n'ont pas de propriété, et les deux dernières une propriété de type `@luint`.

1.8.4 Instructions sur les objets de type `class`

Vous avez besoin de deux instructions relatives aux classes :

- la déclaration d'une variable de type classe ;
- l'instanciation d'un objet de type classe.

La déclaration d'un référence de type classe se fait simplement en nommant le type puis l'objet ; par exemple :

```
@instruction instruction
```

L'instanciation d'un objet de type classe s'effectue en appelant le constructeur new d'une classe concrète avec les paramètres effectifs correspondants aux attributs de la classe, précédés des paramètres effectifs correspondants aux attributs des classes héritées :

```
instruction = @rotate.new {!valeurAngle}
```

Les deux instructions peuvent réduites en :

```
@instruction instruction = @rotate.new {!valeurAngle}
```

1.8.5 Travail à faire

Compléter les règles de productions pour chaque instruction (sauf l'instruction `CALL`).

1.8.6 Le type liste d'instructions

Pour construire la liste des instructions, il faut définir un nouveau type dont les valeurs sont des listes.

Éditez le fichier `chezmoi/logo/galgas-sources/logo-semantics.galgas` et insérer le texte suivant (n'importe où, GALGAS est indifférent à l'ordre des déclarations) :

```
list @instructionList {
  @instruction mInstruction
}
```

Ceci déclare le type de liste `@instructionList`, dont chaque élément contient un objet instance d'une classe héritière de `@instruction`.

Instructions sur les objets de type liste Voici trois instructions relatives aux listes dont vous allez avoir besoin :

- la déclaration d'un objet de type liste ;
- l'initialisation d'un objet de type liste ;
- l'instruction d'ajout d'une valeur à une liste.

La déclaration d'un objet de type liste se fait simplement en nommant le type puis l'objet ; par exemple :

```
@instructionList maListe
```

L'initialisation d'un objet de type liste s'effectue en créant une liste vide :

```
maListe = {}
```

Les deux écritures précédentes peuvent condensées en une seule par :

```
@instructionList maListe = {}
```

L'instruction d'ajout d'une valeur dans une liste est :

```
maListe += !instruction
```

L'ajout s'effectue toujours à la fin de la liste.

1.8.7 Travail à faire

Compléter les règles de productions construire la liste des instructions d'une routine et la liste des instructions du programme principal (les instructions CALL sont toujours ignorées).

1.8.8 L'instruction CALL

Pour prendre en compte l'instruction `CALL`, nous allons procéder comme suit : d'abord, la définition du type table `@routineMap` va être modifier de façon à associer à chaque routine la liste mise à plat des instructions. Ensuite, nous prendrons en compte l'instruction `CALL` en extrayant de la table des routines la liste des instructions de la routine appelée, et en l'insérant à la fin de la liste courante des instructions.

1.8.9 Modification du type table @routineMap

Il faut maintenant modifier la définition du type table `@routineMap` de façon qu'à chaque nom de routine soit associée sa liste d'instructions :

```
map @routineMap {
  @instructionList mInstructionList
  insert insertKey  error message "the '%K' routine has been already declared"
  search searchKey error message "the '%K' routine is not declared"
}
```

Recompiler les sources GALGAS, et examiner les erreurs produites. Corrigez les en vous aidant des explications suivantes :

- l'instruction d'insertion doit maintenant nommer un argument effectif en sortie supplémentaire, de type `@instructionList` :

  ```
  [!?maTable insertKey !clef !maListe]
  ```

- l'instruction de recherche doit maintenant nommer un argument effectif en entrée, dont le type est `@instructionList` :

  ```
  [maTable searchKey !clef ?maListe]
  ```

Prise en compte de l'instruction `CALL`. Il suffit d'ajouter à la liste courante des instructions toutes les instructions de la routine appelée par `CALL` :

```
...
[maTable searchKey !nomRoutine ?listeInstructionRoutine]
for i in listeInstructionRoutine do
  listeCouranteInstructions += !i.mInstruction
end
...
```

L'instruction `for` permet d'énumérer un objet de type liste. Le corps de la boucle (entre `do` et `end`) est exécuté une fois pour chaque élément `i` de la liste.

1.9 Génération de code

Dans ce TP, la génération de code est divisée en deux étapes : d'abord, la succession des segments à tracer est simplement affichée sur le terminal ; dans un second temps, un fichier SVG est engendré au moyen d'un template.

L'allure du calcul des tracés est la suivante (à placer à la fin de la règle `<start_symbol>`) dans `logo-syntax.galgas` :

```
  ...
  @bool pendown = false
  @double x = 0.0
  @double y = 0.0
  @double angle = 0.0 # Angle en degrés
  for i in instructionList do
    ...
  end
```

Pour exprimer l'action à réaliser, des méthodes (définies et implémentées en dehors de leurs classes) vont être utilisées.

1.9.1 Déclaration de la méthode abstraite

Elle est nommée par exemple `codeDisplay` et on peut la déclarer dans n'importe que fichier ; par souci de simplicité, on choisit le fichier qui contient les déclarations sémantiques, c'est-à-dire `chezmoi/logo/galgas-sources/logo`

```
abstract method @instruction codeDisplay
  ?!@bool ioPenDown
  ?!@double ioX
  ?!@double ioY
  ?!@double ioAngle
```

1.9.2 Implémentation d'une héritière concrète

Par exemple, pour la classe `@penUp`, la surcharge de la méthode `codeDisplay` est la suivante. Pour la même raison que précédemment, on place cette déclaration dans le fichier de définitions sémantiques `chezmoi/logo/galgas-sources/logo-semantics.galgas`.

```
override method @penUp codeDisplay
  ?!@bool ioPenDown
  ?!@double unused ioX
  ?!@double unused ioY
  ?!@double unused ioAngle
{
  ioPenDown = false
```

```
}
```

L'implémentation de la méthode héritière concrète pour `@penDown` est élémentaire.

1.9.3 Implémentation de l'héritière concrète pour `@rotate`

Il faut accumuler l'angle de rotation dans l'argument `ioAngle`. Or l'attribut `mAngle` de la classe `@rotate` n'est pas du type `@uint`, mais du type `@luint`. Pour extraire la composante `@uint` d'un `@luint`, on écrit `[mAngle uint]`. Pour transformer un objet `unUint` de type `@uint` en `@double`, on écrit de la même façon `[unUint double]`.

Il faut donc écrire :

```
ioAngle = ioAngle + [[mAngle uint] double]
```

1.9.4 Implémentation de l'héritière concrète pour `@forward`

La méthode complète est alors :

```
override method @forward codeDisplay
  ?!@bool ioPenDown
  ?!@double ioX
  ?!@double ioY
  ?!@double ioAngle
{
  let @double x = ioX + [mLength double] * [ioAngle cosDegree]
  let @double y = ioY + [mLength double] * [ioAngle sinDegree]
  if ioPenDown then
    message "[" + ioX + ", " + ioY + "] -> ["+ x + ", " + y + "]\n"
  end
  ioX = x
  ioY = y
}
```

1.9.5 Calcul des tracés

Le calcul des tracés dans `logo-syntax.galgas` peut être complété par l'appel de la méhode `codeDisplay` pour chaque instruction.

```
  ...
  @bool pendown = false
  @double x = 0.0
  @double y = 0.0
  @double angle = 0.0 # Angle en degrés
```

```
  for i in instructionList do
    [i.mInstruction codeDisplay !?penDown !?x !?y !?angle]
  end
```

Maintenant vous pouvez effectuer la compilation GALGAS et la compilation C++.

1.9.6 Exemple de fichier SVG

Voici à titre d'exemple le fichier SVG qui doit être engendré par la compilation de l'exemple `carre.logo` :

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
                            "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="100%" height="100%" version="1.1" xmlns="http://www.w3.org/2000/svg">
<title>carre.logo</title>
<line x1="100" y1="100" x2="150" y2="100" style="stroke:#1F56D2" />
<line x1="150" y1="100" x2="150" y2="150" style="stroke:#1F56D2" />
<line x1="150" y1="150" x2="100" y2="150" style="stroke:#1F56D2" />
<line x1="100" y1="150" x2="100" y2="100" style="stroke:#1F56D2" />
</svg>
```

1.9.7 Template de génération du fichier SVG

Créer un fichier `chezmoi/logo/galgas-sources/logo-svg.galgasTemplate` et y insérer le contenu suivant :

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
                              "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="100\%" height="100\%" version="1.1" xmlns="http://www.w3.org/2000/svg">
<title>%!TITLE%</title>
%!DRAWINGS%</svg>
```

Notez :

- l'échappement des caractères `%` : pour obtenir `"100%"`, on écrit `"100\%"` ;
- les deux symboles `TITLE` et `DRAWINGS` .

1.9.8 Déclarer un template en GALGAS

Dans le fichier `chezmoi/logo/galgas-sources/logo-semantics.galgas` , insérer la déclaration du template :

```
filewrapper generationTemplate in "." {
}{
}{
```

```
  template svg "logo-svg.galgasTemplate"
    ?@string TITLE
    ?@string DRAWINGS
}
```

En GALGAS, un filewrapper est une structure de données qui est l'image d'un répertoire contenant des fichiers et des sous répertoires. Un fichier particulier est un template ; la déclaration mentionne les symboles (ici `TITLE` et `DRAWINGS`) comme arguments d'entrée, et le contenu est analysé par GALGAS de façon à vérifier qu'il est bien formé (usage correct des caractères `%`).

1.9.9 Construire la liste des instructions SVG

La liste des instructions de tracé est accumulée dans une chaîne de caractères. Modifier toutes les méthodes `codeDisplay` de façon à construire cette chaîne en ajoutant un argument formel en entrée/sortie : `?!@string SVG`. Il faut modifier la méthode `codeDisplay` de la classe `@forward` pour ajouter la génération de code :

```
override method @forward codeDisplay
  ?!@bool ioPenDown
  ?!@double ioX
  ?!@double ioY
  ?!@double ioAngle
  ?!@string SVG
{
  let @double x = ioX + [mLength double] * [ioAngle cosDegree]
  let @double y = ioY + [mLength double] * [ioAngle sinDegree]
  if ioPenDown then
    SVG += "<line x1=\"" + ioX + "\" y1=\"" + ioY + "\" x2=\""
        + x + "\" y2=\"" + y
        + "\" style=\"stroke:#1F56D2\" stroke-linecap=\"round\" />\n"
  end
  ioX = x
  ioY = y
}
```

Pour terminer, voici le code complet de l'axiome `<start_symbol>`, qui enchaîne analyse syntaxique, sémantique et génération du fichier SVG :

```
rule <start_symbol> {
#-- Definition des routines
  $PROGRAM$
  @routineMap tableRoutines = {}
  @instructionList instructions = {}
  repeat
```

```
  while
    <routine_definition> !? tableRoutines
  end
#--- Programme principal
  $BEGIN$
  <instruction_list> !? tableRoutines !? instructions
  $END$
  $.$
#--- Calcul des instructions SVG
  @bool pendown = false
  @double x = 0.0
  @double y = 0.0
  @double angle = 0.0 # Angle en degrés
  @string SVG = ""
  for i in instructions do
    [i.mInstruction codeDisplay !?pendown !?x !?y !?angle !?SVG]
  end
#--- Fichier de sortie
  let @string sourceFilePath = @string.stringWithSourceFilePath
  let @string code = [filewrapper generationTemplate.svg
    ![sourceFilePath lastPathComponent]
    !SVG
  ]
  [code writeToFile ![sourceFilePath stringByDeletingPathExtension] + ".svg"]
}
```

Compiler et essayer l'exécutable : un fichier SVG doit être produit lors de chaque exécution.

Le tutorial est terminé.

Chapitre 2

Options de la ligne de commande

GALGAS accepte un certain nombre d'options, qui sont détaillées dans les pages suivantes.

L'analyse des arguments de la ligne de commande est simple :

- tout argument qui commence par un « - » est une option ;
- tout argument qui ne commence pas par un « - » est considéré comme un fichier source GALGAS ;
- les extensions acceptables par le compilateur GALGAS sont :
  - « `.galgas` », un fichier source ;
  - « `.galgasProject` », un fichier de description de projet ;
  - « `.galgasTemplate` », un fichier de description de template.

L'ordre des options et des fichiers sources est quelconque. La ligne de commande est complètement analysée avant le traitement des fichiers sources. Si plusieurs fichiers sources apparaissent dans la ligne de commande, ils sont traités dans leur ordre d'apparition.

Note pour Windows. L'outil GALGAS pour Windows propose par défaut un dialogue invitant à entrer les références d'un fichier source si la ligne ne contient aucun fichier source (c'est le cas quand on double-clique sur l'icône de l'application). L'option **`--no-dialog`**, spécifique à cette plate forme, permet d'inhiber l'apparition du dialogue.

2.1 Options générales

`--help` Affiche la liste des options.

`--version` Affiche le numéro de version.

--no-color Les messages émis sur le terminal sont en texte pur, sans coloration.

--no-dialog (*uniquement sur Windows*) L'outil GALGAS pour Windows propose par défaut un dialogue invitant à entrer la référence d'un fichier source si la ligne ne contient aucun fichier source (c'est le cas quand on double-clique sur l'icône de l'application). Cette option permet d'inhiber l'apparition du dialogue.

2.2 Options *quiet* et *verbose*

-v , **--verbose** Affiche des messages complémentaires sur le terminal. Par défaut, quand toutes les étapes se déroulent correctement, aucun message n'est affiché.

-q , **--quiet** N'affiche aucun message complémentaire sur le terminal. Par défaut, des messages complémentaires sur le terminal sont affichés.

Ces deux options s'excluent, c'est-à-dire qu'un exécutable définit soit l'option *quiet*, soit l'option *verbose*, mais par les deux :

- le compilateur GALGAS implémente l'option *quiet*, mais pas l'option *verbose* ;
- par défaut, un compilateur engendré par GALGAS implémente l'option *quiet*, mais pas l'option *verbose* ;
- Si la déclaration `%quietOutputByDefault` parmi les déclarations du fichier projet (voir section **??** page **??**), le compilateur engendré par GALGAS implémente l'option *verbose*, mais pas l'option *quiet*.

2.3 Option de création d'un projet

--create-project=nom Crée un nouveau projet GALGAS nommé nom dans le répertoire courant.

2.4 Options contrôlant le compilateur

-W , **--Werror** Tout *warning* est considéré comme une erreur. Cela peut être important dans un script, l'outil de commande renvoyant un code non nul si une ou plusieurs erreurs ont été détectées.

--max-errors=n Stoppe après n erreurs.

--max-warnings=n Stoppe après n alertes.

--check-usefulness Calcul de l'utilité des constructions. L'utilisation de cette option est décrite dans le chapitre **??** à partir de la page **??**.

--property-access-requires-self Quand cette option est activée, `self` est requis pour accéder aux propriétés de l'objet courant dans les *méthodes*, *getter* et *setter*. Par exemple, étant donné la classe `@maClasse` :

```
class @maClasse {
  @uint maPropriété
}
```

Si on écrit :

```
setter @maClasse incrémenter {
  maPropriété ++
}
```

Cette écriture du `setter` n'est acceptée que si l'option `--property-access-requires-self` est désactivée. Si elle est activée, il faut écrire :

```
setter @maClasse incrémenter {
  self.maPropriété ++
}
```

À noter que cette dernière écriture est toujours acceptée, que l'option `--property-access-requires-self` soit activée ou non.

--warns-anonymous-for-instruction Quand cette option est activée, un warning est émis pour chaque instruction `for` dont la variable énumérée est anonyme (construction `for () in ...`, voir section **??** page **??**).

2.5 Options contrôlant la génération de fichiers

--emit-issue-json-file=$fichier$ Écrit dans un $fichier$ au format JSON la liste des erreurs et des alertes.

--log-file-read Affiche sur la console tout accès en lecture à un fichier.

--no-file-generation Inhibe l'écriture de tout fichier.

--mode=nom Contrôle l'opération du compilateur : si nom est vide, le compilateur opère normalement. Si nom est `lexical-only`, le compilateur affiche le résultat de l'analyse lexicale et s'arrête ; aucun fichier n'est engendré. Si nom est `syntax-only`, le compilateur affiche le résultat de l'analyse syntaxique et s'arrête ; aucun fichier n'est engendré.

--compile=nom Enchaîne une compilation C++ après une compilation GALGAS sans erreur. Le nom est le nom d'une cible de type *makefile* ; par exemple, `--compile=makefile-macosx` enchaîne la compilation C++ de la cible *makefile-macosx*.

--macosx=n Force la génération d'un projet Xcode dont le SDK et le *macos Deployement Target* sont fixés à 10.n. Attention, cette option ne vous dispense pas de préciser `%applicationBundleBase` (section **??** page **??**).

2.6 Options de débogage du compilateur

Ces options ne sont pas destinées à être utilisées lors de l'exploitation de GALGAS : elles permettent de déboguer le compilateur lui-même, et non pas le fichier source compilé.

--generate-many-cpp-files Engendre le code C++ dans une multitude de fichiers. Ceci permet un débogage plus simple du compilateur GALGAS lui-même, mais ralentit ensuite l'étape de compilation C++.

--generate-one-cpp-header Engendre un seul fichier d'en-tête C++ pour tout le projet. Ceci permet un débogage plus simple du compilateur GALGAS lui-même, mais ralentit ensuite l'étape de compilation C++.

--check-gmp Exécute au démarrage une série de calculs afin de vérifier si la librairie GMP s'exécute correctement.

2.7 Options de documentation

Ces options produisent des fichiers qui facilitent la documention LaTeX de votre compilateur.

--emit-syntax-diagrams Cette option provoquent l'émission de fichiers LaTeX qui contiennent les diagrammes syntaxiques des grammaires des projets compilés. Son utilisation est détaillée au chapitre **??** à partir de la page **??**.

--print-predefined-lexical-actions Affiche sur la console la liste des routines lexicales prédéfinies.

--generate-shared-map-automaton-dot-files Exporte les automates d'états finis associés à chaque table de symboles de type `shared map` . Les fichiers de sortie sont placés dans le répertoire `build/helpers`, et portent le nom du type table postfixé par l'extension `.dot`.

--output-concrete-syntax-tree Exporte dans un fichier l'arbre syntaxique concret du code source analysé sous la forme d'un graphe dont le format est compatible avec *Graphviz*. Le nom du fichier de sortie est le nom du fichier source doté de l'extension complémentaire `.dot`.

--output-keyword-list-file=$nomLexique$**:**$nomListe$**:**$colonnes$**:**$prefixe$**:**$postfixe$**:**$fichier$ Cette option permet d'engendrer un fichier au format contenant la liste des mots réservés de votre langage. L'argument qui suit le signe « = » est une séquence de six champs :

- $nomLexique$ est le nom du lexique ;
- $nomListe$ est le nom de la liste ;
- $colonnes$ est un nombre entier naturel, qui représente le nombre de colonnes de la sortie ;
- $prefixe$ est une chaîne (éventuellement vide) qui est placée avant chaque élément de liste ;
- $postfixe$ est une chaîne (éventuellement vide) qui est placée après chaque élément de liste ;
- $fichier$ est une chaîne qui désigne le fichier de sortie.

Prenons un exemple ; supposons que le composant `lexique` de votre langage soit :

```
lexique lex {
  ...
  list mots ... { "a", "b", "c" }
  ...
}
```

En appelant votre compilateur avec l'option **--output-keyword-list-file=lex :mots :2 :::motsreserves.tex**, la liste des mots réservés définies par la liste `mots` du lexique `lex` sera écrite dans le fichier `motsreserves.tex`. Ce fichier aura le contenu suivant :

```
a & b \\
c & \\
```

C'est un fichier qui peut être inclus dans une définition de tableau à deux colonnes. Si le nombre d'éléments n'est pas un multiple du nombre de colonnes, la dernière ligne est complétée par des champs vides. Par exemple, on écrit en LaTeX :

```
\begin{table}[!t]
  \centering
  \begin{tabular}{ll}
    \input{motsreserves.tex}
  \end{tabular}
\end{table}
```

On peut utiliser les champs $prefixe$ et $postfixe$ pour afficher de manière particulière chaque élément : avec l'option **--output-keyword-list-file=lex :mots :2 :\texttt{:} :motsreserves.tex**, le fichier `motsreserves.tex` aura le contenu suivant :

```
  \texttt{a} & \texttt{b} \\
  \texttt{c} & \\
```

Chapitre 3

Élements lexicaux

Les éléments lexicaux du langage GALGAS sont :

- les identificateurs (section **??** page **??**);
- les mots réservés (section **??** page **??**);
- les délimiteurs (section **??** page **??**);
- les sélecteurs (section **??** page **??**);
- les séparateurs (section **??** page **??**);
- les commentaires (section **??** page **??**);
- les non terminaux (section **??** page **??**);
- les terminaux (section **??** page **??**);
- les constantes littérales entières (section **??** page **??**);
- les constantes littérales flottantes (section **??** page **??**);
- les caractères littéraux (section **??** page **??**);
- les constantes chaînes de caractères (section **??** page **??**);
- les noms de types (section **??** page **??**);
- les attributs (section **??** page **??**).

`abstract`	`after`	`array`	`as`	`bang`
`before`	`between`	`block`	`boolset`	`case`
`cast`	`class`	`constructor`	`default`	`dict`
`do`	`drop`	`else`	`elsif`	`end`
`enum`	`error`	`extension`	`extern`	`false`
`fileprivate`	`filewrapper`	`fixit`	`for`	`func`
`getter`	`grammar`	`graph`	`gui`	`if`
`in`	`indexing`	`insert`	`is`	`label`
`let`	`lexique`	`list`	`listmap`	`log`
`loop`	`map`	`message`	`method`	`mod`
`mutating`	`not`	`on`	`operator`	`option`
`or`	`override`	`parse`	`private`	`proc`
`project`	`protected`	`public`	`refclass`	`remove`
`repeat`	`replace`	`rewind`	`rule`	`search`
`select`	`self`	`send`	`setter`	`sortedlist`
`spoil`	`struct`	`style`	`switch`	`syntax`
`tag`	`template`	`then`	`true`	`typealias`
`unused`	`var`	`warning`	`while`	`with`

Tableau 3.1 – *Mots réservés du langage GALGAS*

3.1 Les identificateurs

Un identificateur commence par une lettre minuscule ou majuscule, suivie de zéro, un ou plusieurs chiffres décimaux, lettres minuscules ou majuscules ou caractères `'_'`. Par exemple :

`element`, `element0`, `element_0`, `instructionList`, `instruction_list`.

Toutes les lettres Unicode sont acceptées : il est possible d'utiliser des lettres accentuées, des lettres grecques, ... Par exemple :

```
let constanteAccentuée = 12
let π = 3.14
let α = 1
var переменная = 7
```

3.2 Les mots réservés

Les mots réservés de GALGAS sont les identificateurs listés dans le tableau **??** page **??**.

!=	!==	!∧	&	&&	&*	&+	&++	&-	&--	&/	(	)	*	*=	
+	++	+=	,	-	--	-=	->	/	/=	:	:>	;	=	==	
===	>	>=	>>	?∧	[	]	∧	`	{	|	||	}	~		

Tableau 3.2 – *Délimiteurs du langage GALGAS*

!	!selecteur:	!?	!?selecteur:	?	?selecteur:	?!	?!selecteur:

Tableau 3.3 – *Sélecteurs du langage GALGAS*

3.3 Les délimiteurs

Les délimiteurs du langage GALGAS sont listés dans le tableau **??** page **??**.

3.4 Les sélecteurs

Les sélecteurs du langage GALGAS sont listés dans le tableau **??** page **??**.

3.5 Les séparateurs

Les séparateurs du langage GALGAS sont :

- le caractère *espace* ;
- tout caractère dont le point de code est compris entre `U+0000` et `U+001F`.

3.6 Les commentaires

Un commentaire commence par le caractère «#» s'étend jusqu'à la fin de la ligne courante.

3.7 Les non terminaux

Un *non terminal* d'une grammaire est un identificateur placé entre les caractères < et >. Exemple :

```
<expression>, <instruction>
```

Les lettres Unicode y sont acceptées.

Suffixe	Type	Exemples
Pas de suffixe	`@uint`	`1_234` , `0x1234_5678`
L	`@uint64`	`1_234L` , `0x1234_5678L`
S	`@sint`	`1_234S` , `0x1234_5678S`
LS	`@sint64`	`1_234LS` , `0x1234_5678LS`
G	`@bigint`	`1_234G` , `0x1234_5678G`

Tableau 3.4 – *Suffixes et types des constantes littérales entières*

3.8 Les terminaux

Un *terminal* d'une grammaire est une chaîne de caractères placée entre deux caractères « `$` ». Exemple :

```
$identifier$, $constant$
```

Tout caractère Unicode dont le point de code est compris entre `0x21` (« ! ») et `0xFFFD` peut apparaître dans un terminal :

```
$=$, $($, $--$, $≠$
```

Deux échappements sont définis :

- « `\\` » qui permet de définir un unique « `\` »;
- « `\$` » qui permet de définir un « `$` ».

Ceci permet par exemple de définir les terminaux suivants :

```
$\\$, $\$terminal\$$
```

3.9 Les constantes littérales entières

Une constante littérale entière peut être écrite :

- en *décimal* : elle est constituée de un ou plusieurs chiffres décimaux; exemple : `123` , `9` , `05` ;
- en *hexadécimal* : elle commence par `0x`, suivi d'un ou plusieurs chiffres hexadécimaux; exemple : `0x12A` , `0xabcd` .

Le caractère « _ » peut être utilisé pour séparer les chiffres décimaux ou hexadécimaux : `1_234` , `0x123_4567` .

Une constante littérale entière est typée; son type est fixé par son suffixe (tableau **??**).

3.10 Les constantes littérales flottantes

Une constante littérale flottante comprend toujours un point. Elle est constituée :

Échappement	Caractère	Point de code
`'\f'`	Nouvelle page	U+0C
`'\n'`	Passage à la ligne (*Line Feed*)	U+0A
`'\r'`	Retour chariot	U+0D
`'\t'`	Tabulation horizontale	U+09
`'\v'`	Tabulation verticale	U+0B
`'\\'`	Barre oblique inversée	U+5C
`'\0'`	Caractère nul	U+0
`'\''`	Apostrophe	U+27
`'\uabcd'`	Caractère du plan de base (4 chiffres hexadécimaux)	U+ABCD
`'\Uabcdefgh'`	Caractère Unicode (8 chiffres hexadécimaux)	U+abcdefgh

Tableau 3.5 – *Séquence d'échappement des constantes littérales caractère*

- d'un ou plusieurs chiffres décimaux ;
- suivis d'un point ;
- suivi de zéro, un ou plusieurs chiffres décimaux.

Par exemple : `0.` , `12.34` .

Le caractère « _ » peut être utilisé pour séparer les chiffres : `1_234.567_890` .

Une constante littérale flottante est du type `@double` .

3.11 Les caractères littéraux

Un *caractère littéral* est un caractère Unicode placé entre deux apostrophes « ' ». Exemple :

```
'a', 'æ', 'Œ'
```

Plusieurs séquences d'échappements sont définies et listées dans le tableau **??**.

3.12 Les constantes chaînes de caractères

Un *chaîne de caractères littérale* est une séquence de caractères Unicode placé entre deux guillemets « " ». Exemple :

```
"une chaîne", "Œnologie"
```

Plusieurs séquences d'échappements sont définies et listées dans le tableau **??**.

Échappement	Caractère	Point de code
`"\f"`	Nouvelle page	U+0C
`"\n"`	Passage à la ligne (*Line Feed*)	U+0A
`"\r"`	Retour chariot	U+0D
`"\t"`	Tabulation horizontale	U+09
`"\v"`	Tabulation verticale	U+0B
`"\\"`	Barre oblique inversée	U+5C
`"\""`	Guillemet	U+22
`"\uabcd"`	Caractère du plan de base (4 chiffres hexadécimaux)	U+ABCD
`"\Uabcdefgh"`	Caractère Unicode (8 chiffres hexadécimaux)	U+abcdefgh

Tableau 3.6 – *Séquence d'échappement des constantes littérales chaîne de caractères*

3.13 Les noms de types

Un nom de type :

- commence par un caractère «@»;
- est suivi par un ou plusieurs chiffres ou lettres;
- est suivi éventuellement par un tiret « - », lui-même suivi par un ou plusieurs chiffres ou lettres.

Par exemple :

```
@string, @stringlist-element, @2stringlist
```

3.14 Les attributs

Un attribut :

- commence par un caractère « %»;
- est suivi par une lettre Unicode;
- est suivi par une ou plusieurs lettres Unicode, chiffres décimaux, « - » ou « _».

La liste des attributs est donnée dans le tableau **??** page **??**. Un attribut est un mot réservé « secondaire ».

`%MacOS`	`%MacOSDeployment`	`%app-link`
`%app-source`	`%applicationBundleBase`	`%codeblocks-linux32`
`%codeblocks-linux64`	`%codeblocks-windows`	`%generatedInSeparateFile`
`%libpmAtPath`	`%macCodeSign`	`%makefile-macosx`
`%makefile-unix`	`%makefile-win32-on-macosx`	`%makefile-x86linux32-on-macosx`
`%makefile-x86linux64-on-macosx`	`%nonAtomicSelection`	`%once`
`%preserved`	`%quietOutputByDefault`	`%selector`
`%templateEndMark`	`%tool-source`	`%translate`
`%useGrammar`	`%usefull`	

Tableau 3.7 – *Attributs du langage GALGAS*

Chapitre 4

Calcul des entités utiles

Le compilateur GALGAS implémente l'option `--check-usefulness` qui permet de déceler si des constructions sont inutiles. Au fur et à mesure de l'évolution de la conception de votre compilateur, il se peut que des constructions (type, functions, ...) deviennent inutilisées. Cette option permet de déceler ces constructions.

L'option `--check-usefulness` demande au compilateur GALGAS de construire le graphe d'utilité. La construction n'est entreprise que si analyse lexicale, syntaxique et sémantique sont effectuées sans erreur.

Pour chaque construction inutile, un *warning* est déclenché, qui indique l'endroit de la déclaration de la construction inutile.

4.1 Le calcul d'utilité

Les constructions qui sont évaluées sont :

- les routines `extension getter` ;
- les routines `extension setter` ;
- les routines `extension method` ;
- les composants `lexique` ;
- les composants `grammar` ;
- les composants `syntax` ;
- les composants `option` ;
- les `filewrapper` ;

- les fonctions ;
- les procédures ;
- les types ;
- les routines `after` ;
- les routines `before` ;
- les routines d'analyse `case .fileExtension` .

Le compilateur GALGAS calcule l'utilité des constructions en se partant des nœuds racines :

- les routines `after` sont utiles ;
- les routines `before` sont utiles ;
- les routines d'analyse `case .fileExtension` sont utiles ;
- tous les types prédéfinis sont utiles ;
- toutes les fonctions marquées `%usefull` sont utiles (section **??** page **??**).

Les relations d'utilité sont :

- une routine `extension getter` , `extension setter` , `extension method` est utile si son type est utile ;
- un composant `grammar` est utile si il apparaît dans une instruction `grammar` utile ;
- un composant `syntax` est utile si il est nommé par un composant `grammar` utile ;
- un composant `lexique` est utile si il est nommé par un composant `syntax` utile ;
- un composant `option` est utile si il est nommé dans les instructions d'une routine utile ;
- un `filewrapper` est utile si il est nommé dans les instructions d'une routine utile ;
- une fonction est utile si elle est nommée dans les instructions d'une routine utile ;
- une procédure est utile si elle est nommée dans les instructions d'une routine utile ;
- un type est utile si il est instancié par les instructions d'une routine utile.

4.2 Trucs et astuces

Supprimer une construction inutile n'est pas toujours élémentaire : par exemple, un type qui n'est jamais instancié est inutile, mais il peut apparaître comme type d'une propriété d'un type structure lui aussi inutile. Aussi, supprimer un type inutile peut entraîner des erreurs de compilation.

De plus, le calcul d'utilité a été récemment implémenté dans GALGAS, et un *faux positif* est possible.

On peut donc commencer par supprimer procédures et fonctions inutiles (attention si vous appelez les fonctions par l'intermédiaire d'un objet de type `@function`, voir section **??** page **??**), c'est en général plus simple que la suppression d'un type.

Une astuce consiste à renommer la construction calculée comme inutile puis à recompiler le projet : si il n'y a pas d'erreur, cette construction peut être supprimée sans dommage.

4.3 Cas particulier : l'appel indirect des fonctions

Le calcul d'utilité ne rend utile que les fonctions appelées directement à partir des instructions des routines utiles. L'appel indirect via des objets de type `@function` (page **??**) n'est pas pris en compte par le calcul d'utilité. En conséquence, les fonctions appelées uniquement via des objets de type `@function` (page **??**) sont calculées inutiles.

Il suffit d'ajouter l'attribut `%usefull` à ces fonctions pour forcer leur utilité (section **??** page **??**).

Chapitre 5

Diagrammes syntaxiques des grammaires en TeX

Le compilateur GALGAS implémente l'option `--emit-syntax-diagrams` qui permet d'obtenir les diagrammes syntaxiques de chaque grammaire de votre langage. Ceux-ci sont décrits en LaTeX en utilisant le paquetage `tikz`.

Il n'y a pas de miracle, les diagrammes syntaxiques peuvent être tronqués parce qu'ils débordent dans la marge droite ou dans le bas de la page : d'ailleurs vouloir exploiter ces diagrammes peut être l'occasion de revisiter la forme des règles de production.

Note. La compilation LaTeX des diagrammes syntaxiques est très lente !

Dans tout ce chapitre, nous appliquons cette démarche au langage LOGO, défini à la section **??** page **??**.

5.1 Mise en œuvre

La mise en œuvre est très simple : il suffit d'ajouter l'option indiquée ci-dessus lors de la compilation de votre projet :

```
galgas --emit-syntax-diagrams chezmoi/logo/+logo.galgasProject
```

Les fichiers LaTeXproduits sont rangés dans le répertoire `chezmoi/build/tex`. Deux fichiers sont produits pour chaque grammaire implémentée par votre projet ; pour le projet LOGO, la grammaire définie s'appelle `logo_grammar` (section **??** page **??**), ces fichiers sont :

- `chezmoi/build/logo_grammar.document.tex` (section **??** page **??**) ;
- `chezmoi/build/logo_grammar.tex` (section **??** page **??**).

5.2 Le document `logo_grammar.document.tex`

Le fichier `chezmoi/build/logo_grammar.document.tex` contient un document LaTeX di-rectement compilable qui vous permet d'obtenir immédiatement un document PDF contenant les diagrammes syntaxiques de votre langage ; il inclut le fichier `chezmoi/build/lo- go_grammar.tex` qui contient les diagrammes syntaxiques.

Ce fichier sert d'exemple de configuration de `tikz` et de l'affichage des diagrammes. Pour donner une chance aux règles de production de s'afficher complètement, le format du document est *paysage A3*. Ce fichier contient plusieurs définitions de commandes qui permettent de paramétrer l'affichage des diagrammes, et qui sont donc appelées dans `chezmoi/build/logo_grammar.tex` .

```
\newcommand\nonTerminalSection[2]{\section{Nonterminal \texttt{#1}}\label{nt:#2}}
```

Cette commande est émise avant chaque non terminal. La définition ci-dessus définit une *section*, et une étiquette qui permet d'établir des hyper-liens sur les non terminaux. Le premier argument est le nom du non terminal, le second est son numéro, utilisé pour les hyper-liens.

```
\newcommand\ruleSubsection[3]{\subsection{Component \texttt{#1}, in file \texttt{#2}, line #3}}
```

Cette commande est émise avant chaque diagramme syntaxique d'un non terminal. La définition ci-dessus définit une *sous-section*, et une étiquette qui permet d'établir des hyper-liens sur les non terminaux. Les trois arguments sont : le nom du composant syntaxique qui contient la règle, le fichier dans lequel la règle apparaît, et le numéro de ligne dans ce fichier.

Définir une sous-section par règle n'est pas forcément souhaitable, on peut vouloir obtenir la liste des règles, sans aucun texte intermédiaire. Pour cela, il suffit d'écrire :

```
\newcommand\ruleSubsection[3]{}
```

```
\newcommand\ruleMatrixColumnSeparation{3mm}
```

Définit l'espacement horizontal entre deux colonnes dans les diagrammes syntaxiques.

```
\newcommand\ruleMatrixRowSeparation{3mm}
```

Définit l'espacement vertical entre deux rangées dans les diagrammes syntaxiques.

```
\newcommand\nonTerminalSymbol[2]{\hyperref[nt:#2]{#1}}
```

Cette commande est émise pour chaque occurrence d'un non terminal dans un diagramme syntaxique. Le premier argument est son nom, le second son numéro. Le numéro permet de définir l'hyper-lien vers la définition du non terminal.

```
\newcommand\startSymbol[2]{The start symbol is \hyperref[nt:#2]{#1}.}
```

Commande émise une fois, pour définir le texte qui annonce l'axiome de la grammaire. Les deux arguments sont le nom de l'axiome et son numéro, qui sert à établir un hyper-lien vers sa définition.

```
\newcommand\nonTerminalSummaryStart{This is the alphabetical list of non terminal : }
```

C'est le texte introductif de la table des non-terminaux.

```
\newcommand\nonTerminalSummary[2]{\hyperref[nt:#2]{#1}}
```

Commande émise à chaque occurrence d'un terminal dans la table. Les deux arguments sont le nom de l'axiome et son numéro, qui sert à établir un hyper-liens vers sa définition.

```
\newcommand\nonTerminalSummarySeparator{, }
```

Commande émise pour séparer deux non terminaux consécutifs dans la table.

```
\newcommand\nonTerminalSummaryEnd{.\\}
```

Commande émise une fois, pour terminer la table des non terminaux.

5.3 Le fichier `logo_grammar.tex`

La présentation adoptée dans ce fichier est :

- l'axiome de la grammaire (émission de la commande `\startSymbol`);
- la table des non terminaux (émission des commandes `\nonTerminalSummaryStart`, `\nonTerminalSummary`, `\nonTerminalSummarySeparator`, `\nonTerminalSummaryEnd`);
- pour chaque non terminal :
  - son annonce par la commande `\nonTerminalSection`;
  - pour chaque règle de production de ce non terminal :
    - son annonce par la commande `\ruleSubsection`;
    - son diagramme syntaxique par un environnement `tikzpicture`.

Chapitre 6

Formatage pour LaTeX

Si vous utilisez LaTeXpour écrire la documentation de votre compilateur, vous êtes confronté sans doute au problème de la présentation des programmes sources. En effet, les paquetages classiques pour ce type de problème, comme par exemple `listings`, peuvent être trop rigides pour des règles lexicales particulières d'un langage.

Par exemple, en GALGAS, les constantes entières acceptent le caractère `_`, comme dans `123_456`. Elles peuvent être préfixées par `0x`, et postfixées par `S`, `LS` pour indiquer leur type : `0x123_456S`, ou encore `0x_123_456_LS`. Le paquetage `listings` ne peut pas être paramétré pour afficher correctement les constantes entières de GALGAS.

Comment faire ? Développer des commandes LaTeXparticulières pour faire ce travail. Elles s'appuient sur un mode particulier des compilateurs engendrés par GALGAS, qui permet de traduire un fichier source en un code compatible LaTeX. C'est de cette façon que le code GALGAS est présenté dans ce document. Si les fichiers `.tex` sont codés en UTF-8, alors les caractères accentués peuvent être utilisés sans restriction, comme des caractères comme `æ` ou `Œ` (voir par exemple le *getter* `unicodeToLower` *du type* `@char` *(page* **??***)*).

Dans la suite, nous allons progressivement présenter la démarche pour formatter un code source :

- d'abord comment configurer votre compilateur pour qu'il engendre du code LaTeX ;
- comment afficher ce code en utilisant le paquetage `filecontents` ;
- une amélioration de la solution précédente en définissant un environnement particulier (utilise le paquetage `verbatim`) ;
- définition d'une commande permettant d'afficher du code en ligne, appelable comme la commande `\verb` (utilise le paquetage `verbatim`).

Dans tout ce chapitre, nous appliquons cette démarche au langage LOGO, défini à la section **??** page **??**.

6.1 Configuration de votre compilateur

6.1.1 option `--mode=latex`

Tout compilateur engendré par GALGAS possède un mode d'exécution particulier, le mode *latex*. Il est activé par l'option `--mode=latex`.

Dans ce mode, seule l'analyse lexicale est effectuée, aussi le fichier source doit être *lexicalement correct*, mais n'a pas besoin d'être ni *syntaxiquement correct*, ni *sémantiquement correct*.

Le fichier de sortie a pour nom le fichier d'entrée postfixé par l'extension `.tex`. Il contient le texte source formatté pour LaTeX.

Par exemple, si le fichier d'entrée est `test.logo` et contient :

```
ROUTINE trace
BEGIN
  FORWARD 50;
  ROTATE 90;
END
```

En appelant le compilateur LOGO par la commande `logo --mode=latex test.logo`, le fichier `test.logo.tex` est engendré et contient :

```
\keywordsStyle{R{}O{}U{}T{}I{}N{}E{}}\hspace*{.6em}t{}r{}a{}c{}e{} \\
\keywordsStyle{B{}E{}G{}I{}N{}} \\
\hspace*{.6em}\hspace*{.6em}\keywordsStyle{F{}O{}R{}W{}A{}R{}D{}}\hspace*{.6em}
                                    \integerStyle{5{}0{}}\delimitersStyle{;{}} \\
\hspace*{.6em}\hspace*{.6em}\keywordsStyle{R{}O{}T{}A{}T{}E{}}\hspace*{.6em}
                                    \integerStyle{9{}0{}}\delimitersStyle{;{}} \\
\keywordsStyle{E{}N{}D{}}
```

Pour l'afficher, il suffit de définir les commandes `\keywordsStyle`, `\integerStyle` et `\delimitersStyle` [1], et de placer ce texte dans un environnement où une police à échappement fixe est activée :

```
\newcommand\keywordsStyle[1]{\textcolor{blue}{\textbf{#1}}}
\newcommand\delimitersStyle[1]{\textcolor{brown}{\textbf{#1}}}
\newcommand\integerStyle[1]{\textcolor{brown}{#1}}
\texttt{
\keywordsStyle{R{}O{}U{}T{}I{}N{}E{}}\hspace*{.6em}t{}r{}a{}c{}e{} \\
\keywordsStyle{B{}E{}G{}I{}N{}} \\
\hspace*{.6em}\hspace*{.6em}\keywordsStyle{F{}O{}R{}W{}A{}R{}D{}}\hspace*{.6em}
                                \integerStyle{5{}0{}}\delimitersStyle{;{}} \\
\hspace*{.6em}\hspace*{.6em}\keywordsStyle{R{}O{}T{}A{}T{}E{}}\hspace*{.6em}
                                \integerStyle{9{}0{}}\delimitersStyle{;{}} \\
```

[1] Aucune commande n'est définie pour les identificateurs, car l'analyseur lexical ne définit pas de style pour ceux-ci (voir section **??** page **??**).

```
\keywordsStyle{E{}N{}D{}}
}
```

On obtient ainsi (noter l'identation de la première ligne) :

```
 ROUTINE trace
BEGIN
  FORWARD 50;
  ROTATE 90;
END
```

6.1.2 option `--mode:suffixe=latex`

Si vous documentez plusieurs compilateurs, vous pouvez avoir une collision de nom de style. Une variante de l'option `--mode=latex` est de préciser un suffixe : `--mode:suffixe=latex`. Le `suffixe` doit être un nom uniquement constitué de lettres (minuscules ou majuscules). Ce suffixe est ajouté aux noms de style. En appelant le compilateur LOGO par la commande `logo --mode=latex:Logo test.logo`, le fichier `test.logo.tex` est engendré et contient :

```
\keywordsStyleLogo{R{}O{}U{}T{}I{}N{}E{}}\hspace*{.6em}t{}r{}a{}c{}e{} \\
\keywordsStyleLogo{B{}E{}G{}I{}N{}} \\
\hspace*{.6em}\hspace*{.6em}\keywordsStyleLogo{F{}O{}R{}W{}A{}R{}D{}}\hspace*{.6em}
                            \integerStyleLogo{5{}0{}}\delimitersStyleLogo{;{}} \\
\hspace*{.6em}\hspace*{.6em}\keywordsStyleLogo{R{}O{}T{}A{}T{}E{}}\hspace*{.6em}
                            \integerStyleLogo{9{}0{}}\delimitersStyleLogo{;{}} \\
\keywordsStyleLogo{E{}N{}D{}}
```

6.1.3 Formatages complémentaires

Il est possible de formatter l'affichage du code en utilisant des paquetages standard. Ci-après sont présentées deux possibilités avec les paquetages `lineo` et `mdframed`.

6.1.3.1 Formatage avec le paquetage `lineno`

Le paquetage `lineno` permet de numéroter les lignes sources :

```
\resetlinenumber
\begin{linenumbers}
\ttfamily
...
\end{linenumbers}
```

Et on obtient :

```
ROUTINE trace
BEGIN
  FORWARD 50;
  ROTATE 90;
END
```

6.1.3.2 Formatage avec le paquetage `mdframed`

Le paquetage `mdframed` permet (entre autres) d'afficher un trait vertical dans la marge gauche. Pour cela, il faut d'abord le configurer en créant un evironnement, ici `siderules` :

```
\newmdenv[
  topline=false,
  bottomline=false,
  rightline=false,
  linecolor=red!25,
  linewidth=2pt
]{siderules}
```

En utilisant l'environnement `siderules` :

```
\begin{siderules}
\ttfamily
...
\end{siderules}
```

On obtient :

```
ROUTINE trace
BEGIN
  FORWARD 50;
  ROTATE 90;
END
```

6.1.4 Comment s'effectue la traduction en LaTeX

La traduction s'effectue comme suit :

- à chaque `style` défini dans l'analyseur lexical correspond une commande LaTeXparti-culière : par exemple à `keywordsStyle` correspond `\keywordsStyle` [2] (section **??** page **??**) ;
- si une erreur lexicale est détectée, une commande `\lexicalError` [3] est insérée ;

[2] Si un suffixe est précisé (`--mode:suffixe=latex`), alors ce suffixe est ajouté à la commande `\keywordsStyle`.

[3] Si un suffixe est précisé (`--mode:suffixe=latex`), alors ce suffixe est ajouté à la commande `\lexicalError`.

Caractère source	Formattage pour LaTeX
'>'	\textgreater{}
'<'	\textless{}
'∼'	$\sim$
'∧'	$\wedge$
'&'	\&
'|'	\textbar{}
'%'	\%
'#'	\#
'$'	\$
' '	\hspace*{.6em}
'\n'	\newline\n
'{'	\{
'}'	\}
'_'	_
'\'	\textbackslash{}
'''	\textquotesingle{}
'"'	\textquotedbl{}
Autre caractère : 'c'	c{}

Tableau 6.1 – *Échappement et substitution des caractères pour formattage LaTeX*

- les caractères possédant une signification particulière en LaTeXsont échappés ou substitués selon le tableau **??**;
- après tout caractère non échappé ni substitué est ajoutée la séquence `{}`.

Vous devez donc créer une commande particulière pour chaque style, plus éventuellement la commande `\lexicalError` pour afficher l'occurrence des erreurs lexicales. Vous pouvez choisir de ne pas définir la commande `\lexicalError`, auquel cas la compilation LaTeXéchouera en présence d'erreur lexicale; mais si elle réussit, vous êtes sûr qu'il y a aucune erreur lexicale.

6.1.5 Fonctionnement de l'option `--mode=latex`

L'option `--mode=latex` utilise les noms de style définis dans l'analyseur lexical LOGO. Par exemple, l'extrait suivant indique que le style `integerStyle` est attaché au terminal `$integer$` :

```
style integerStyle -> "Integer Constants"
$integer$ !uint32value style integerStyle ...
```

À chaque style, correspond une commande LaTeXobtenue en préfixant le nom du style par un anti-slash `\`[4]. Si aucun style n'est défini par un terminal particulier, il est affiché avec le style par défaut : c'est le cas des identificateurs LOGO dans les listings précédents.

[4] Les chiffres et le caractère de soulignement `_` sont interdits dans les noms de style.

Noter que l'affichage des commentaires nécessite l'utilisation conjointe d'un style particulier et de l'instruction lexicale `drop` (section **??** page **??**); pour le langage LOGO :

```
style commentStyle -> "Comments"
$comment$ style commentStyle ...
rule '#' {
  repeat
  while '\u0001' -> '\u0009' | '\u000B' | '\u000C' | '\u000E' -> '\uFFFD'~:
  end
  drop $comment$
}
```

6.2 Affichage via le paquetage `filecontents`

Insérer un texte en effectuant un copié/collé comme suggéré à la section précédente est très laborieux! Le paquetage `filecontents` va permettre de simplifier l'écriture en utilisant l'environnement `filecontents*` :

```
\begin{filecontents*}{temp.logo}
ROUTINE trace
BEGIN
  FORWARD 50;
  ROTATE 90;
END
\end{filecontents*}
\immediate\write18{logo --mode=latex temp.logo}
\noindent{\ttfamily\input{temp.logo.tex}}
```

L'environnement `filecontents*` écrit son contenu dans le fichier `temp.logo` du répertoire courant. La commande `\immediate\write18` [5] permet de lancer la commande shell `logo --mode=latex temp.logo` [6], qui a pour résultat d'écrire le fichier formatté `temp.logo.tex` dans le répertoire courant. Il suffit donc de l'inclure grâce à la commande `\input` en sélectionnant une police à échappement fixe (`\ttfamily`). `\noindent` permet d'éliminer l'indentation de la première ligne.

Cette deuxième approche est plus satisfaisante car on peut faire figurer le texte source LOGO directement dans le fichier LaTeX, mais nous allons voir dans la section suivante une meilleure solution.

6.3 Environnement d'affichage formatté

Dans cette section, on va voir comment nous allons définir un environnement `logocode` qui permettra d'entrer et de formatter implicitement un texte LOGO :

[5] Penser à ajouter l'option `-shell-escape` lors de la compilation LaTeX.

[6] Le répertoire vers l'exécutable `logo` doit faire partie des chemins définis par la variable `$PATH` du shell.

```
\begin{logocode}
ROUTINE trace
BEGIN
  FORWARD 50;
  ROTATE 90;
END
\end{logocode}
```

Ce qui permettra d'obtenir :

```
ROUTINE trace
BEGIN
  FORWARD 50;
  ROTATE 90;
END
```

6.3.1 Package `verbatim`

Pour cela, nous avons besoin du paquetage `verbatim`. Il est conseillé d'inclure ce paquetage juste après la déclaration `\documentclass` :

```
\documentclass [...] {...}
\usepackage{verbatim}
...
```

6.3.2 Définition de l'environnement

La définition de l'environnement `logocode` est la suivante :

```
\newwrite\tempfile
\makeatletter
\newenvironment{logocode}{%
  \begingroup
  \@bsphack
  \immediate\openout\tempfile=temp.logo%
  \let\do\@makeother\dospecials
  \catcode`\^^M\active
  \verbatim@startline
  \verbatim@addtoline
  \verbatim@finish
  \def\verbatim@processline{\immediate\write\tempfile{\the\verbatim@line}}%
  \verbatim@start
}{
  \immediate\closeout\tempfile
  \@esphack
  \endgroup
```

```
18    \immediate\write18{logo --mode=latex temp.logo}
19    {\noindent\ttfamily\input{temp.logo.tex}}
20  }
21  \makeatother
```

Quelques commentaires :

- ligne 1, la commande `\newwrite\tempfile` est nécessaire pour l'écriture de fichier ; elle doit figurer une seule fois dans le texte source, si vous définissez plusieurs environnements d'affichage, veillez à ne pas la dupliquer ;
- ligne 3, le nom d'environnement (en bleu) est défini : bien entendu, vous pouvez changer ce nom pour l'adapter au nom de votre compilateur ;
- ligne 8, attention, après la commande `\catcode`, c'est un accent `` ` `` » ;
- ligne 19, l'affichage de la ligne traduite est effectuée ; à cet endroit, nous pouvez utiliser toutes les commandes de formattage, comme par exemple les paquetages `lineno` et `mdframed` cités plus haut.

Par exemple, à la place de la ligne 19, on peut utiliser l'environnement `siderules` (paquetage `mdframed`) et écrire :

```
\noindent\begin{siderules}\ttfamily\input{temp.logo.tex}\end{siderules}
```

6.4 Affichage du code en ligne

Pour afficher du code en ligne, on va définir une commande `\logo` qui s'utilise comme la commande verbatim en ligne `\verb` ; si on écrit :

```
Les mots réservés de LOGO sont \logo+BEGIN+, \logo+END+, ..., les délimiteurs
sont \logo+;+ et \logo+.+.
```

Le délimiteur utilisé ici est `+`, mais, comme pour `\verb`, tout caractère peut être utilisé, à condition qu'il n'apparaisse pas dans la chaîne à formatter. On obtient donc :

Les mots réservés de LOGO sont `BEGIN`, `END`, ..., les délimiteurs sont `;` et `.`.

Comme pour l'affichage d'un listing, nous avons besoin du paquetage `verbatim`. Rappelons qu'il est conseillé d'inclure ce paquetage juste après la déclaration `\documentclass` :

```
\documentclass [...] {...}
\usepackage{verbatim}
...
```

La définition de commande `\logo` est la suivante :

```
1  \newwrite\tempfile
2  \makeatletter
3  \newcommand*\logo{%
4    \@bsphack%
5    \begingroup%
6    \let\do\@makeother\dospecials%
7    \let\do\do@noligs\verbatim@nolig@list%
8    \catcode`\^^M=15\relax%
9    \@vobeyspaces%
10   \@logo{\temporary}%
11 }%
12 \newcommand\@logo[2]{%
13   \catcode`-=12\relax%
14   \catcode`<=12\relax%
15   \catcode`>=12\relax%
16   \catcode`,=12\relax%
17   \catcode`'=12\relax%
18   \catcode``=12\relax%
19   \catcode`#2\active%
20   \catcode`~\active%
21   \lccode`\~`#2\relax%
22   \lowercase{%
23     \begingroup%
24     \def\@tempa##1~{%
25       \expandafter\endgroup%
26       \expandafter\DeclareRobustCommand%
27       \expandafter*%
28       \expandafter#1%
29       \expandafter{@tempa}%
30       \@esphack%
31       \immediate\openout\tempfile=temp.logo%
32       \immediate\write\tempfile{##1}%
33       \immediate\closeout\tempfile%
34       \immediate\write18{logo --mode=latex temp.logo}%
35       \colorbox{gray!6}{\ttfamily\input{temp.logo.tex}\unskip}%
36     }%
37   }%
38   \ifnum`#2=`\~\else\@makeother\~\fi%
39   \expandafter\endgroup%
40   \@tempa%
41 }%
42 \makeatother
```

Commentaires :

- ligne 1, la commande `\newwrite\tempfile` est nécessaire pour l'écriture de fichier ; elle doit figurer une seule fois dans le texte source, si vous définissez plusieurs environnements d'affichage, veillez à ne pas la dupliquer ;
- ligne 8, 13 à 21 et 38 : attention, c'est un accent aigu `` ` `` ;
- ligne 3, 10 et 12 : le nom `logo` apparaît trois fois (en bleu pour être repéré plus facilement) : si vous changez le nom de la commande, veillez à en remplacer toutes les occurrences ;

- une difficulté est d'assurer que la commande n'insère aucune espace supplémentaire : c'est pour cela que toutes les lignes se terminent par `%` [7] ;
- enfin le plus intéressant : ligne 31, le fichier `temp.logo` est ouvert en écriture ;
- ligne 32, le contenu de la commande est écrit dans ce fichier ;
- ligne 33, le fichier est fermé ;
- ligne 34, le compilateur est appelé pour effectuer la traduction en LaTeX ; **attention**, cette commande est un argument de `\lowercase` [8] (ligne 22), si bien que tous les caractères sont passés en minuscules : ainsi, si on écrit `logo --mode=latex:LOGO temp.logo` , c'est la commande `logo --mode=latex:logo temp` qui est exécutée ;
- ligne 35, le code traduit est affiché ; comme la commande `\input` (ligne 35) insère toujours une espace après elle, on la supprime par `\unskip` .

Noter bien que la ligne 35 est une commande générale d'affichage : ici on a choisi un fond gris, et une police à échappement fixe.

Enfin, la commande `\logo` ne peut pas être utilisée dans les notes en bas de page (commande `\footnote`), ni en argument d'une macro.

[7] En fait, uniquement certaines lignes doivent être obligatoirement terminées par `%` ; pour simplifier, on applique cette terminaison à toutes.

[8] Aucune idée de son rôle, mais si on supprime `\lowercase` , la compilation LaTeXéchoue.

Chapitre 7

Traduction dirigée par la syntaxe

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

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

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

7.1 Le programme d'exemple

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

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

L'analyseur syntaxique est le suivant :

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

La grammaire :

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

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

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

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

7.2 Activer la traduction dirigée par la syntaxe

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

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

Pour le composant `syntax` :

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

Et pour la grammaire :

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

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

7.3 Obtenir la chaîne traduite

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

```
grammar expGrammar in inSourceFile
```

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

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

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

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

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

7.4 Modifier l'instruction d'appel de terminal

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

```
$idf$ ?*
```

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

- les séparateurs qui précèdent le terminal ;

[2] Il existe des variantes pour exprimer l'obtention de la chaîne traduite, voir la description de l'instruction grammaire à la section **??** page **??**.

- le terminal lui-même.

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

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

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

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

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

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

On va prendre un exemple pour illustrer cette construction : produite une chaîne dont les identificateurs et les séparateurs qui les précèdent auront disparus. On modifie le composant `syntax` comme suit (il existe une expression plus simple de l'instruction `$idf$ ?* :> ?@string s ?@string t`, puisque `s` et `t` ne sont pas utilisés : c'est `$idf$ ?* :> ?* ?*`, décrite à la section **??** page **??**) :

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

[3] Il existe d'autres variantes de cet opérateur, voir la description de l'*instruction d'appel de terminal* à la section **??** page **??**.

```
}
```

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

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

7.5 Insérer du texte : instruction send

L'instruction `send` a la syntaxe suivante (l'instruction `send` est décrite à la section **??** page **??**) :

```
send exp
```

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

Par exemple, supposons que l'on veuille transformer les parenthèses en accolades ; on écrit le composant `syntax` comme suit (là encore, il existe une forme plus concise de l'instruction `$($ :> ?@string s ?@string t`, puisque `t` est inutilisé : c'est `$($ :> ?@string s ?*`, décrite à la section **??** page **??**) :

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

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

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

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

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

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

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

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

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

```
<expression>
```

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

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

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

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

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

On utilise souvent cette construction pour ne pas transmettre la chaîne obtenue par la dérivation d'un non terminal ; par exemple, si on ne veut pas transmettre les expressions entre parenthèses, on modifie la dernière règle `facteur` en (ou encore : `<expression> :> ?*`) :

```
syntax expSyntax {
  ...
  rule <facteur> {
    $($
```

```
      <expression> :> ?@string e
      $)$
   }
}
```

II

Composants

Chapitre 8

Le composant `project`

Le composant `project` permet de paramétrer un projet GALGAS. Il doit être placé dans un fichier source particulier, d'extension « `.galgasProject` ». Sont déclarés dans un fichier projet :

- la version du projet (dans l'en-tête : section **??** page **??**) ;
- le nom des exécutables engendrés (dans l'en-tête : section **??** page **??**) ;
- les cibles de compilation : section **??** page **??** ;
- les fichiers sources : section **??** page **??**.

Voici un exemple de composant projet :

```
project (1:2:3) -> "logo" {
#--- Targets
  %makefile-macosx
  %makefile-unix
  %makefile-x86linux32-on-macosx
  %makefile-x86linux64-on-macosx
  %makefile-win32-on-macosx
  %applicationBundleBase : "fr.what"
  %codeblocks-windows

#--- Source files
  "galgas-sources/logo-lexique.galgas"
  "galgas-sources/logo-options.galgas"
  "galgas-sources/logo-semantics.galgas"
  "galgas-sources/logo-syntax.galgas"
```

```
   "galgas-sources/logo-grammar.galgas"
   "galgas-sources/logo-cocoa.galgas"
   "galgas-sources/logo-program.galgas"
}
```

8.1 En-tête du fichier projet

L'en-tête d'un projet définit deux informations :

- la version du projet ;
- le nom des exécutables engendrés.

8.1.1 Version du projet

```
project (1:2:3) -> "logo" {
   ...
}
```

La version du projet apparaît sous la forme d'un triplet qui suit le mot-clé `project` : `1:2:3` dans le code ci-dessus. C'est ce triplet (sous la forme `1.2.3`) qui apparaît lorsque l'on invoque l'option `--version` sur l'utilitaire ligne de commande engendré. Dans le code, cette information peut être obtenu par le *constructeur* `projectVersionString` *du type* `@application` *– page* **??**.

8.1.2 Nom des exécutables engendrés

```
project (1:2:3) -> "logo" {
   ...
}
```

Le nom des exécutables engendrés est fixé par la chaîne de caractères qui apparaît dans l'en-tête : `logo` dans l'exemple ci-dessus. Les exécutables compilés en mode *release* portent directement ce nom, ceux compilés en mode *debug* portent ce nom augmenté du suffixe « `-debug` » : `logo-debug`.

8.2 Cibles de compilation

GALGAS peut engendrer des cibles de compilation pour Mac, Linux et Windows. Les outils engendrés sont des *utilitaires en ligne de commande*, sauf sur Mac où une application Cocoa peut être engendrée.

8.2.1 Cibles pour Linux

Deux choix sont possibles :

- `Code::Blocks` ;
- compilation en ligne de commande.

8.2.1.1 `Code::Blocks` pour Linux

L'option `%codeblocks-linux32` engendre une cible qui peut être compilée sur Linux 32 bits, et `%codeblocks-linux64` une cible compilable sur Linux 64 bits, en utilisant `Code::Blocks`[1].

```
project (0:0:1) -> "logo" {
  %codeblocks-linux32
  ...
}
```

8.2.1.2 Compilation en ligne de commande pour Linux

La déclaration `%makefile-unix` engendre une cible qui peut être compilée indifféremment sur Linux ou sur Mac. L'exécutable engendré est un exécutable 32 bits sur un Linux 32 bits, et un 64 bits sur un Linux 64 bits.

```
project (0:0:1) -> "logo" {
  %makefile-unix
  ...
}
```

8.2.2 Cibles pour Mac

Comme GALGAS est développé sur Mac, c'est pour cette plateforme que l'on trouve le plus grand nombre de cibles :

- application Cocoa ;
- compilation en ligne de commande ;
- cross-compilation pour Win32 ;
- cross-compilation pour Linux32 ;
- cross-compilation pour Linux64.

[1] http://www.codeblocks.org

8.2.2.1 Application Cocoa

Cette cible est l'objet du chapitre **??** à partir de la page **??**.

8.2.2.2 Compilation en ligne de commande pour Mac

La déclaration `%makefile-macosx` engendre une cible pour obtenir un exécutable en ligne de commande sur Mac. Note : on peut aussi utiliser `%makefile-unix`.

```
project (0:0:1) -> "logo" {
  %makefile-macosx
  ...
}
```

8.2.2.3 Cross-compilation en ligne de commande pour Win32

La déclaration `%makefile-win32-on-macosx` engendre une cible pour obtenir sur Mac un exécutable en ligne de commande pour `Win32`. À la première cross-compilation, le cross-compilateur est téléchargé à partir du site `rts-software` et placé dans le répertoire `~/galgas-tools-for-cross-compilation`.

```
project (0:0:1) -> "logo" {
  %makefile-win32-on-macosx
  ...
}
```

8.2.2.4 Cross-compilation en ligne de commande pour Linux32

La déclaration `%makefile-x86linux32-on-macosx` engendre une cible pour obtenir sur Mac un exécutable en ligne de commande pour Linux 32 bits sur x86. À la première cross-compilation, le cross-compilateur est téléchargé à partir du site `rts-software` et placé dans `~/galgas-tools-for-cross-compilation`.

```
project (0:0:1) -> "logo" {
  %makefile-x86linux32-on-macosx
  ...
}
```

8.2.2.5 Cross-compilation en ligne de commande pour Linux64

La déclaration `%makefile-x86linux64-on-macosx` engendre une cible pour obtenir sur Mac un exécutable en ligne de commande pour Linux 64 bits sur x86. À la première cross-compilation, le cross-compilateur est téléchargé à partir du site `rts-software` et placé dans `~/galgas-tools-for-cross-compilation`.

```
project (0:0:1) -> "logo" {
  %makefile-x86linux64-on-macosx
  ...
}
```

8.2.3 Cible pour Windows : CodeBlocks

Sur Windows, la compilation C++ du projet engendré s'effectue avec Code::Blocks[2]. La cible est engendrée par la déclaration `%codeblocks-windows`.

```
project (0:0:1) -> "logo" {
  %codeblocks-windows
  ...
}
```

8.3 Déclaration %quietOutputByDefault

À partir de la version 3.1.4, GALGAS et les exécutables engendrés par GALGAS sont verbeux par défaut, c'est-à-dire que leur exécution affiche sur le terminal de nombreuses informations sur le déroulement de l'exécution, comme par exemple la mise à jour ou la création de fichiers. L'option de la ligne de commande *quiet* (section **??** page **??**) permet d'inhiber l'émission de ces messages.

On peut inverser ce comportement en faisant figurer `%quietOutputByDefault` parmi les déclarations du fichier projet :

```
project (0:0:1) -> "logo" {
  ...
  %quietOutputByDefault
  ...
}
```

Dans ce cas, l'exécutable engendré par GALGAS est silencieux par défaut, et bavard grâce à l'option de la ligne de commande *verbose* (section **??** page **??**).

En résumé :

- par défaut, sans l'option `%quietOutputByDefault` parmi les déclarations du fichier projet, l'exécutable est bavard par défaut, et l'option de la ligne de commande *quiet* permet de le rendre silencieux ; l'option de la ligne de commande *verbose* n'existe pas ;
- si l'option `%quietOutputByDefault` est présente parmi les déclarations du fichier projet, l'exécutable est silencieux par défaut, et l'option de la ligne de commande *verbose* permet de le rendre bavard ; l'option de la ligne de commande *quiet* n'existe pas.

[2] http://www.codeblocks.org

Une conséquence est que ni la présence de l'option *quiet* ni la présence de l'option *verbose* ne peuvent être testées par la construction `[option nom_composant_option.nom_option nom_info]` (voir section **??** page **??**). Il faut utiliser le *constructeur* `verboseOutput` *du type* `@application` *– page* **??**.

8.4 Déclaration des fichiers sources du projet

Deux types de fichiers sources peuvent être déclarés :

- des fichiers sources GALGAS ;
- des fichiers sources C++.

Un fichier source est déclaré sous la forme d'une chaîne de caractères qui définit son chemin :

- le chemin est absolu si il commence par un « / » ;
- sinon il est relatif au répertoire qui contient le fichier projet ;
- l'extension du chemin définit le type : « `.galgas` » pour un source GALGAS, « `.cpp` » pour un source C++.

Les sources GALGAS déclarés sont inclus dans la compilation GALGAS. L'ordre dans lequel apparaissent ces fichiers n'a pas d'importance sémantique, il définit simplement l'ordre dans lesquels les analyses lexicale et syntaxique sont effectuées.

Les sources C++ déclarés sont ignorés par la compilation GALGAS, et sont simplement ajoutés à la liste des fichiers C++ à compiler.

Chapitre 9

Projet Xcode et application Cocoa

Vous pouvez demander à GALGAS d'engendrer un projet Xcode, qui contiendra :

- le compilateur en version *release* sous la forme d'un utilitaire en ligne de commande ;
- le compilateur en version *debug* sous la forme d'un utilitaire en ligne de commande ;
- une application Cocoa permettant d'appeler les deux utilitaires.

9.1 Paramétrage du projet GALGAS

Pour engendrer un projet Xcode, il y a deux attributs obligatoires, et un attribut optionnel :

- un premier attribut obligatoire qui définit le OSX SDK et la version système cible (section **??** page **??**) ;
- un second attribut obligatoire `%applicationBundleBase` (section **??** page **??**) ;
- un attribut optionnel `%macCodeSign`, qui définit comment est signé l'application OS X engendrée (section **??** page **??**).

9.1.1 Attribut définissant le OSX SDK et la version système cible

Pour engendrer un projet Xcode, il vous suffit d'ajouter une déclaration telle que `%MacOS` dans votre fichier projet (d'extension `.galgasProject`, voir chapitre **??** à partir de la page **??**). Par exemple :

```
project (0:0:1) -> "logo" {
  %applicationBundleBase : "fr.what"
  ...
```

9.1.2 Attribut `%applicationBundleBase`

Il y a un second attribut obligatoire à ajouter dans le projet GALGAS : `%applicationBundleBase`. Celle-ci fixe le *Bundle Identifier* de l'application Cocoa. À la chaîne définie dans l'option (ici `"fr.what"`) est ajouté le nom du projet (défini dans l'en-tête, ici `"logo"`), précédé par un point : le *Bundle Identifier* est donc `fr.what.logo`.

9.1.3 Attribut `%macCodeSign`

Par défaut, l'application engendrée par Xcode n'est pas signée. Cela signifie qu'elle tournera sur le système et la machine où elle a été compilée, mais peut-être pas sur un autre système et / ou une autre machine.

L'attribut `%macCodeSign` permet de préciser comment Xcode va signer l'application.

Dans la version actuelle de GALGAS, l'attribut `%macCodeSign` permet de signer l'application avec votre compte `Mac Developer`, ou un certificat défini dans le *Trousseau d'accès*, par exemple un certificat auto-signé.

L'attribut `%macCodeSign` doit être associé à une chaîne de caractères, qui comprend deux éléments séparés par un « : » :

```
  %macCodeSign = "MacDeveloper:ZW8HY75J3X"
```

ou

```
  %macCodeSign = "Certificate:John Egg Smith"
```

Le premier correspond au certificat associé à votre compte `Mac Developer`, le second à un certificat détenu dans le *Trousseau d'accès* (« *Key Chain* »).

Manifestement, Apple recommande de signer l'application par votre compte `Mac Developer` : l'interface de Xcode 8 ne permet pas de signer une application avec un certificat auto-signé, mais accepte un projet contenant cette signature.

Les chaînes `ZW8HY75J3X` et `John Egg Smith` sont juste des exemples, il faut que vous utilisiez des valeurs valides.

Pour obtenir la chaîne associée à votre compte `Mac Developer`, vous avez plusieurs possibilités :

- utiliser l'utilitaire `certtool` (section **??** page **??**) ;
- utiliser l'application *Trousseaux d'accès* (section **??** page **??**) ;
- signer l'application dans le projet Xcode engendré, et observer sa description dans un éditeur de texte : section **??** page **??**.

La section **??** page **??** montre comment définir un certificat auto-signé.

Une fois que l'attribut `%macCodeSign` aura été défini, vous pourrez effectuer la compilation GALGAS de votre projet. Compiler le projet Xcode engendrera une application signée. On pourra le vérifier, comme expliqué à la section **??** page **??**.

9.1.3.1 Utilitaire `certtool`

Sur Mac OS X, l'utilitaire `certtool` permet de manipuler les certificats. L'option `y` permet de les afficher. Entrer dans le terminal :

```
certtool y
```

L'exécution de la commande affiche sur le terminal une grande quantité de lignes dans lesquelles il faut rechercher le certificat correspondant à votre compte `Mac Developer`. On peut faciliter ce travail en redirigeant la sortie de la commande dans un fichier :

```
certtool y > certificats.txt
```

Rechercher dans la sortie de la commande la chaîne `Mac Developer`. Vous la trouvez associée avec l'adresse électronique de votre compte développeur (ici `john@smith.nowhere`) :

```
Subject Name    :
Other name      : 9ZED6TWL8M
Common Name     : Mac Developer: john@smith.nowhere (Q933RG93DL)
OrgUnit         : ZW8HY75J3X
Org             : JOHN SMITH
Country         : US
```

La chaîne recherchée apparaît associée à l'entrée `OrgUnit`.

9.1.3.2 Application *Trousseaux d'accès*

L'application *Trousseaux d'accès* (« Keychain Access ») permet de retrouver les certificats ; ouvrir l'application, sélectionner dans la barre latérale `Mes certificats`, et, éventuellement, entrer `Mac Developer` dans la zone recherche. Le certificat recherché doit apparaître comme l'indique la figure **??**. Double-cliquer pour en avoir le détail : figure **??**. La chaîne recherchée apparaît associée à l'entrée `Unité d'organisation`.

9.1.3.3 Signature dans Xcode

On peut utiler Xcode pour retrouver la chaîne recherchée. Pour cela, on va modifier le projet Xcode engendré pour signer l'application engendrée.

Évidemment, toute modification de Xcode pourra être écrasée par une recompilation GALGAS du projet : on n'utilise donc la modification du projet juste pour découvrir la chaîne recherchée.

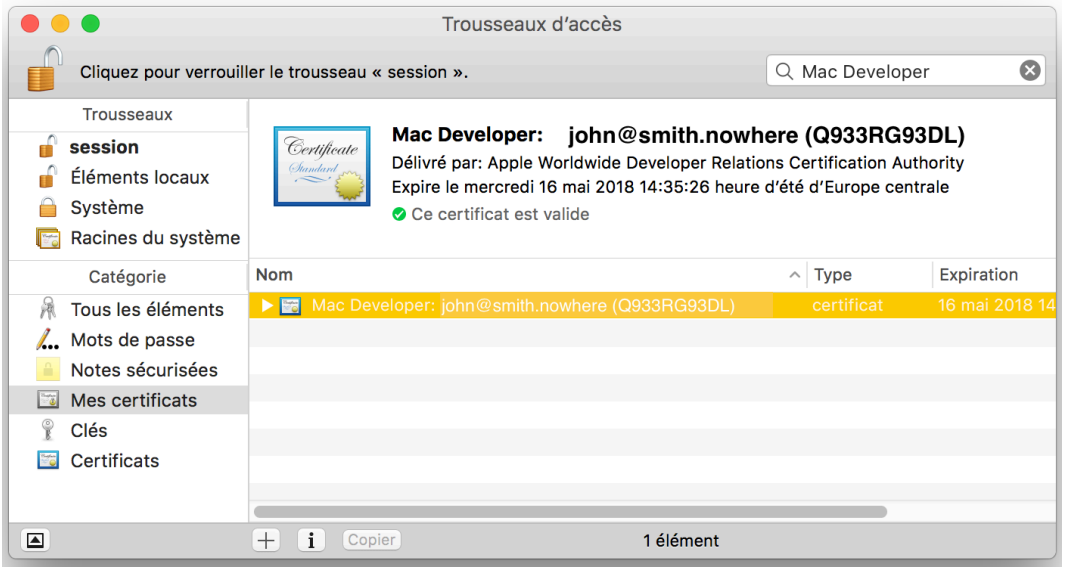


Figure 9.1 – *Application* Trousseaux d'accès

Figure 9.2 – *Détail du certificat*

Ouvrir donc le projet Xcode, sélectionner `logo` dans la barre latérale gauche, puis la cible (dans « TARGETS ») `Cocoa logo`, puis l'onglet `General`, et repérer le bloc `Signing` : figure **??**. Cliquez alors sur « Enable Development Signing », de façon à aboutir à la figure **??**.

Maintenant, fermer Xcode. Le fichier projet Xcode `logo.xcodeproj` est en fait un répertoire. Pour afficher son contenu, effectuer un clic secondaire sur l'icône de `logo.xcodeproj`, et sélectionner dans le menu contextuel qui apparaît l'item « *Afficher le contenu du paquet* » (figure **??**.a) ; la fenêtre qui apparaît (figure **??**.b) montre le contenu de paquet. Le fichier qui nous intéresse est `project.pbxproj`. Ce fichier est un fichier texte seul, ouvrez-le avec un éditeur de texte. Repérer la ligne définissant le paramètre

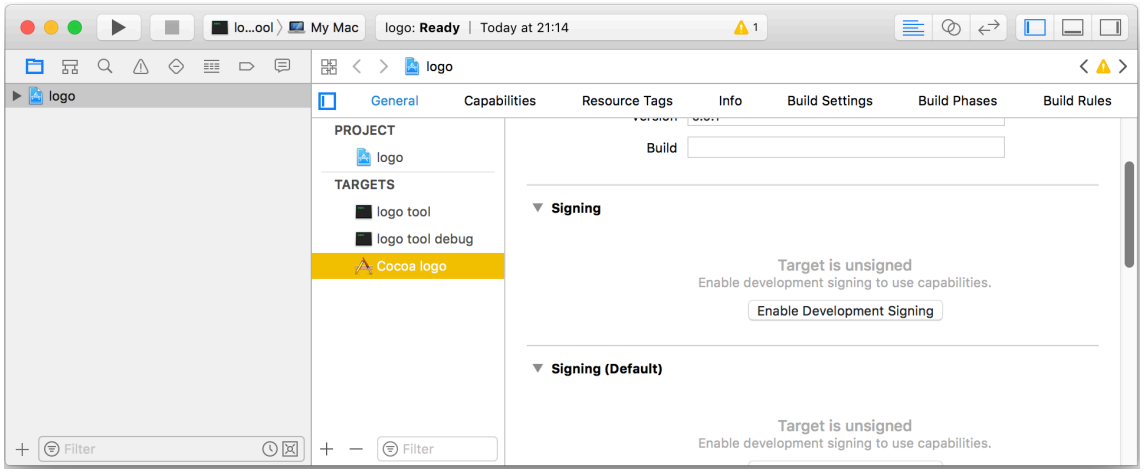


Figure 9.3 – *Projet Xcode sans signature*

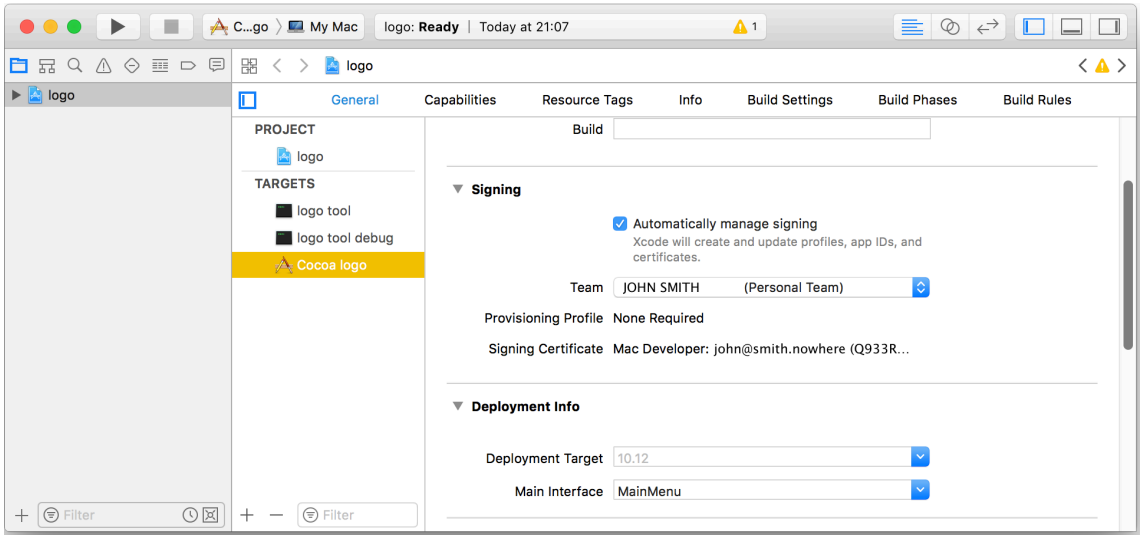


Figure 9.4 – *Signature à partir du projet Xcode*

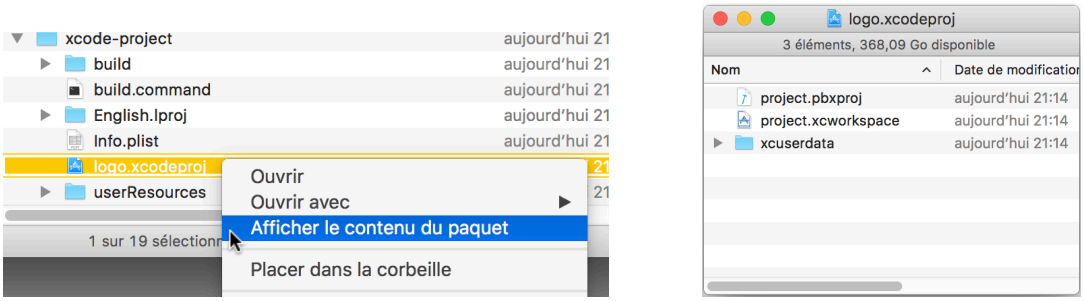
`DEVELOPMENT_TEAM` :

```
DEVELOPMENT_TEAM = ZW8HY75J3X;
```

La valeur associée au paramètre `DEVELOPMENT_TEAM` est la chaîne recherchée : `ZW8HY75J3X`.

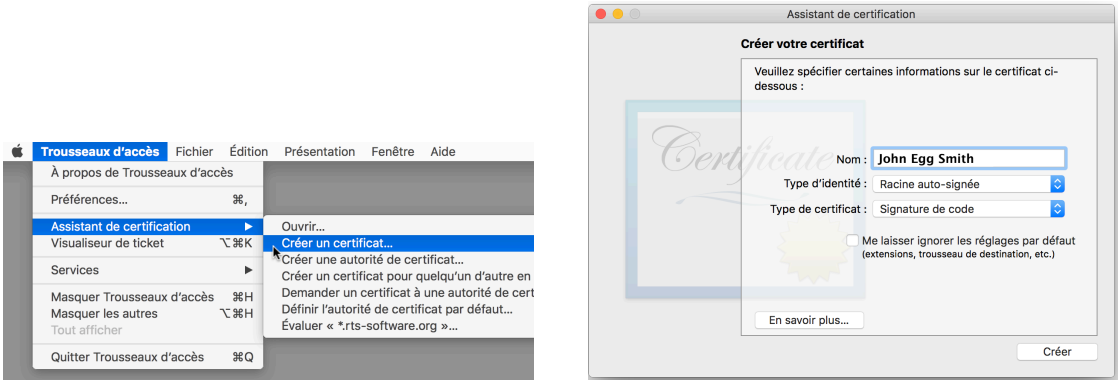
9.1.3.4 Définir un certificat auto-signé

Pour définir un certificat auto-signé, appler l'application « *Trousseaux d'accès* » (« *Keychain Access* »). Ensuite, sélectionner dans le menu de l'application `Assistant de certification` -> `Créer un certificat…`, comme indiqué à la figure **??**.a. Ensuite, dans la fenêtre qui apparaît (figure **??**.b), entrer le nom que vous allez donner au certificat (ici « John Egg Smith »), sélectionner pour le `Type d'identité` « Racine auto-signée », et pour `Type de certificat` « Signature de code ». Terminer la création en cliquant sur `Créer`.



(a) Afficher le contenu du paquet **(b)** Contenu du paquet

Figure 9.5 – *Afficher dans le Finder le contenu du paquet projet Xcode*



(a) Menu de création d'un certificat **(b)** Création d'un certificat auto-signé

Figure 9.6 – *Certificat auto-signé*

Le nom que vous avez donné au certificat est important, c'est lui que vous allez utilisez pour l'attribut `%macCodeSign` (respecter absolument la casse et les espaces) :

```
%macCodeSign = "Certificate:John Egg Smith"
```

9.1.4 Vérifier la signature d'une application

Pour vérifier la signature d'une application, on peut utiliser l'outil `spctl`. Dans le terminal, exécutez :

```
spctl -a -t exec -vv chemin.app
```

Où `chemin.app` est le chemin vers l'application que la compilation du projet Xcode a créé.

Si l'application a été signée par le compte `Mac Developer`, l'exécution de la commande affiche :

```
chemin.app: accepted
override=security disabled
origin=Mac Developer: john@smith.nowhere (Q933RG93DL)
```

Si l'application a été signée par un certificat auto-signé, l'exécution de la commande affiche :

Fichier ou répertoire	Rôle
`build.command`	Effectue la compilation Xcode, appelable via une commande *Shell*
`Info.plist`	Informations pour l'application Cocoa
`English.lproj`	Informations pour l'application Cocoa
`userResources`	Permet d'associer des icônes aux fichiers sources de votre compilateur, ainsi qu'à l'application Cocoa engendrée (voir section **??** page **??**)

Tableau 9.1 – *Fichiers et répertoires relatifs au projet Xcode*

```
chemin.app: accepted
override=security disabled
origin=John Egg Smith
```

Si l'application n'a pas été signée, l'exécution de la commande :

```
chemin.app: accepted
source=no usable signature
override=security disabled
```

9.2 Projet Xcode engendré

Quand le projet GALGAS est compilé, un répertoire `xcode-project` directory est créé, et contient :

- le fichier projet `Xcode` ;
- un fichier `build.command` ;
- un fichier `Info.plist` ;
- un répertoire `English.lproj` ;
- un répertoire `userResources` .

Le rôle de chacun est précisé par le tableau **??**. Ne pas modifier ces fichiers et répertoires à la main, une compilation GALGAS supprimerait vos changements. La seule exception est le contenu du répertoire `userResources` qui n'est pas modifié par les compilations GALGAS.

9.3 Définir des icônes pour votre application Cocoa

Vous pouvez définir :

- une icône pour l'application Cocoa ;
- une icône particulière pour chaque type de fichier source.

Le nom de chaque fichier d'icône fixe son rôle :

- pour l'application Cocoa, le fichier d'icône doit s'appeler `application_icns.icns` ;
- pour chaque type de fichier source, le nom est basé sur l'extension du fichier : si celui-ci est par exemple `.logo`, le fichier d'icônes doit s'appeler `logo_icns.icns`.

Ces fichiers d'icônes doivent être placés dans le répertoire `userResources`, et il faut ensuite refaire une compilation GALGAS pour que ces fichiers soient ajoutés au projet `Xcode`.

En résumé :

1. concevoir les fichiers d'icônes, en fixant leur nom comme indiqué ci-dessus ;
2. placer ces icônes dans le répertoire `userResources` ;
3. effectuer une compilation GALGAS : celle-ci met à jour le projet `Xcode`, en ajoutant les fichiers d'icônes au *target* Cocoa ;
4. recompiler le *target* Cocoa du projet `Xcode` : les icônes sont prises en compte.

9.4 Indexation des fichiers sources

Vous pouvez configurer votre projet GALGAS pour que l'application Cocoa engendrée établisse une indexation et des références croisées : un `cmd-click` affiche un menu contextuel. Cette indexation est basée sur l'analyse syntaxique. C'est ce qui a été fait pour l'application `CocoaGalgas` (figure **??** page **??**). On voit dans le menu contextuel trois classes d'index : `Class Definition`, `Class Reference as Superclass` et `Abstract Category Method Definition` ; au dessous, les références croisées correspondantes.

Pour configurer votre projet, vous avez à modifier le composant *lexique*, le composant *syntax*, le composant *grammar*, et la règle d'analyse du fichier source. Les cinq modifications sont présentées successivement ci-après, en prenant comme exemple le langage LOGO (section **??** page **??**).

9.4.1 En tête du composant `lexique`

Il faut modifier l'en-tête, en ajoutant la déclaration `indexing in` :

```
lexique logo_lexique indexing in "INDEXING" {
  ...
```

La chaîne `"INDEXING"` définit le nom du répertoire qui contient les fichiers cache de l'indexation. Ce répertoire est relatif au répertoire qui contient le fichier source.

Note : si vous effectuez maintenant la compilation GALGAS, vous obtiendrez une erreur sur la définition de la grammaire, indiquant qu'elle doit aussi indiquer la prise en compte de l'indexation.

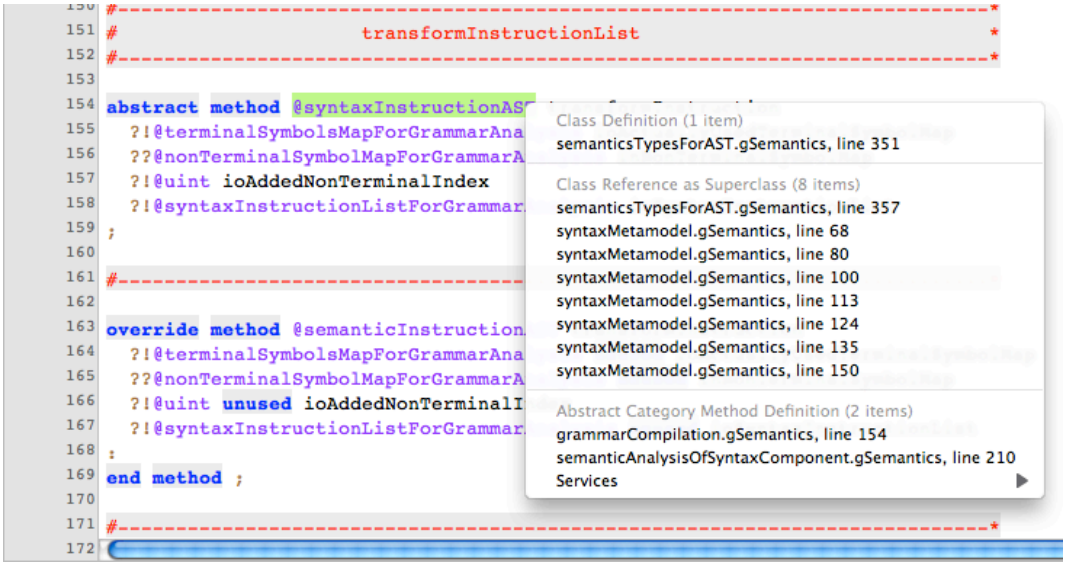


Figure 9.7 – *Indexation et références croisées dans l'application CocoaGalgas*

9.4.2 En tête du composant `grammar`

Il suffit de préfixer par `indexing` l'en-tête du composant `grammar` :

```
indexing grammar logo_grammar ... {
  ...
```

Note : maintenant, la compilation GALGAS s'effectue sans erreur.

9.4.3 Règle d'analyse des fichiers sources

La règle d'analyse des fichiers source doit mentionner dans l'en-tête la grammaire utilisée pour l'analyse (pour l'exemple du langage LOGO, c'est le rôle de la troisième ligne `grammar logo_grammar`).

```
case . "logo"
message "a source text file with the .logo extension"
grammar logo_grammar
?sourceFilePath:@lstring inSourceFile {
  grammar logo_grammar in inSourceFile
}
```

Quand le mode d'exécution (absence de l'option `--mode`) est le mode par défaut, les instructions de la règle sont exécutées. Ci-dessus, la seule instruction est l'instruction `grammar logo_grammar in inSourceFile` (ligne 5).

Quand le mode d'exécution (présence de l'option `--mode`) n'est pas le mode par défaut, les instructions de la règle ne sont pas exécutées, et les opérations sont guidées par la grammaire indiquée ligne 3. Dans le cas de l'indexation, l'exécution construit l'indexation du fichier source.

9.4.4 Déclaration des classes d'index

La déclaration des classes d'index s'effectue dans l'analyseur lexical. Dans la cadre du langage d'exemple LOGO, on veut simplement indéxer les routines, plus précisément l'endroit de leur définition, et les endroits où elles sont appelées. On définit donc deux classes d'index `routineDefinition` et `routineCall`. À chaque déclaration est associée une chaîne de caractères, qui sera le titre affiché dans le menu contextuel.

```
lexique logo_lexique indexing in "INDEXING" {
  ...
indexing routineDefinition : "Routine Definition"
  ...
indexing routineCall : "Routine call"
  ...
```

Ces définitions peuvent être placées à tout endroit dans la définition de l'analyseur lexical.

9.4.5 Définition des entrées indexées

L'analyseur syntaxique va être complété de façon à définir les symboles qui seront indéxés. Plus précisement, c'est l'instruction d'analyse de symbole terminal qui est modifiée.

Considérons d'abord la déclaration de routine. La règle de l'analyseur syntaxique qui définit cette analyse est :

```
rule <routine_definition> {
  $ROUTINE$
  $identifier$ ?let routineName
  $BEGIN$
  <instruction_list>
  $END$
}
```

Le nom de la routine est défini par l'instruction `$identifier$ ?let routineName` : on la modifie alors de façon à signifier que l'indentificateur doit être indéxé comme une définition de routine :

```
rule <routine_definition> {
  $ROUTINE$
  $identifier$ ?let routineName indexing routineDefinition
  $BEGIN$
  <instruction_list>
  $END$
}
```

Maintenant, l'instruction d'appel de routine :

```
rule <instruction> {
  select
    $CALL$
    $identifier$ ?let @lstring routineName
    $;$
  or
    ...
  end
}
```

On modifie de manière analogue l'instruction `$identifier$ ?let @lstring routineName` :

```
rule <instruction> {
  select
    $CALL$
    $identifier$ ?let @lstring routineName indexing routineCall
    $;$
  or
    ...
  end
}
```

9.4.6 Compilation et essai

Les modifications sont terminées. Vous pouvez recompiler votre projet (compilation GALGAS puis compilation de la cible Cocoa du projet `Xcode`). La figure **??** montre le résultat obtenu en effectuant un `cmd-click` sur le nom de la routine.

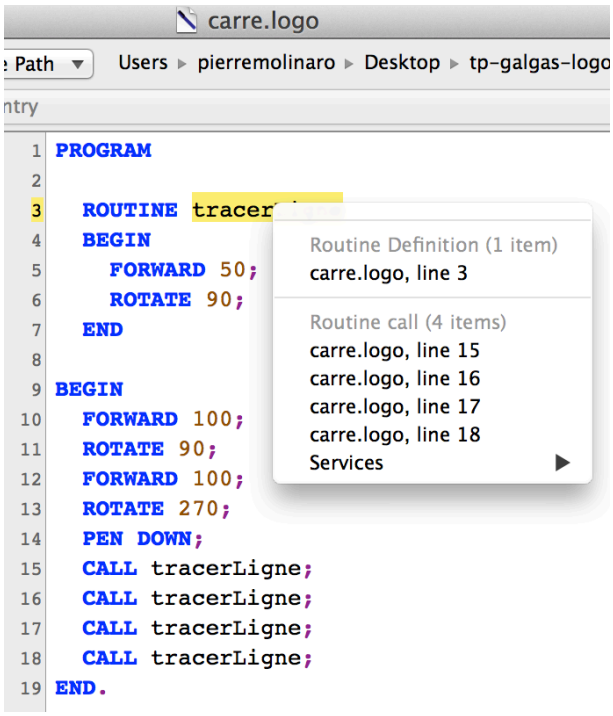


Figure 9.8 – *Exemple d'indexation en LOGO*

Chapitre 10

Le composant `lexique`

Le rôle d'un analyseur lexical est de grouper les caractères de la chaîne d'entrée en *symboles terminaux*, ou encore *terminaux*, en écartant les séparateurs comment les espaces ou les commentaires.

En GALGAS, un analyseur lexical est défini par un composant `lexique`. Les composants `syntax`, qui définissent un ensemble de règles de production, font référence à un composant `lexique`.

10.1 Définition d'un composant `lexique`

En GALGAS, un composant `lexique` a la structure suivante :

```
lexique nom {
  declarations
}
```

Le `nom` est le nom donné au composant; il est utilisé pour référencer le composant `lexique` dans un composant `syntax`.

Un composant `lexique` peut contenir les déclarations suivantes :

- déclaration d'attribut lexical;
- déclaration d'un symbole terminal;
- déclaration d'une liste de symboles terminaux;
- déclaration d'un message d'erreur lexical;
- déclaration d'un style;
- déclaration de règles d'analyse.

Un *attribut lexical* contient la valeur associée à un symbol terminal : par exemple, la valeur entière d'une constante entière, la valeur chaîne de caractères d'un identificateur ou d'une constante chaîne de caractères, ...

In GALGAS, all terminal symbols must be declared either by a //single terminal symbol declaration//, either by a //list of terminal symbols declaration//. This defines the set of defined terminal symbols of your grammar.

Lexical error messages need also to be explicitly declared by //lexical error message declaration//.

A //style declaration// declares a style identifier, for defining automatic coloring in a text editor. Currently, coloring is only available for Mac OS X Cocoa applications.

The order of declarations is not significant, but any entity must be declared before being used.

==== Lexical Rules Overview ==== The //lexical rules// define the executable part of a lexical component. Every lexical rule define //matching strings// that are are tested against substring from current location in input string. A matching string has a one character or more.

10.2 Comment opère un analyseur lexical

You can consider the lexical analyzer as an autonomous thread which analyzes the input string and which sends the sequence of the terminal symbols to the parser. Of course, for efficiency, the lexical analyzer is actually a parser subroutine.

The flowchart of a GALGAS lexical analyzer execution is :

how_works_a_lexical_analyzer.png

When the input string is loaded from source file, a "NUL" character is appended as End Of String (eos) mark.

During execution, the lexical analyzer maintains a //current location// that designates the next character of the input string to be analyzed. Initially, current location points out the first character of the input string.

The lexical analyzer loops until the end of input string is reached. At the beginning of every loop, lexical attributes are reset to their default value.

Then, the first lexical rule matching expressions are tested against substring at current location in input string : * on match success, the first lexical rule is executed ; usually, this execution sends a terminal symbol to the parser ; however, in some cases as parsing a delimitor or a comment, no terminal symbol is sent ; * on match failure, the lexical analyzer tries to find a match with the second lexical rule, and so on.

If no lexical rule matches, the character at current location is tested against eos character. On match success, the lexical analyzer sends once a predefined terminal symbol (denoted by "

") to the parser, for telling it the end of input string is reached. On match failure, the //unknow character// lexical error is raised. The character at current location is discarded, that is the current location points out the next character of the input string.

10.3 Ambiguïtés lexicales

GALGAS does not currently check that the set of lexical rules is unambiguous. So, if the set is unambiguous, the rule order is not significant ; if two or more rules introduce an ambiguity, the first defined one is used.

10.4 Un exemple

This is very simple scanner, from "galgas/samples/notSLRgrammar.ggs" :

|"**lexique** my_scanner_for_not_SLR_grammar :
#— Identifiers

id
error **message** **rule** 'a'-> 'z'| 'A'-> 'Z' :
send
id
;
end **rule** ;
#— Delimitors
list delimitorsList error **message** **rule** **list** delimitorsList ;
#— Separators
rule '\1'-> **end** **rule** ;
end **lexique** ;"|

This `lexique` component defines the following set of terminal symbols : "
id
" (explicitly declared), "
$=$
" and "
$*$
" (declared by "delimitorsList" list.

The first rule sends the "
id
" terminal symbol each time a lower case or upper case character is found. The second rule names the "delimitorsList" list and sends the "
$=$
" or "
$*$
" terminal symbol each time the corresponding character is found. The last rule discards silently the space character and any control character.

Note that this scanner considers identifiers of only one character : "ab" is scanned as two consecutive

identifiers.

===== Finding Sample Code =====

You can find examples of `lexique` components in : * "galgas/sample/alt_sample.ggs" file ; this is a very basic scanner that handles one-letter identifier and four delimitors ; * "galgas/sample/arith_expression.ggs" file (for scanning literal integers) ; * "galgas/sample/test_LR1_grammar.ggs" file gives an example of a small scanner for "toy" parser ; * "galgas/galgas/galgas_sources/galgas_scanner.ggs" file : this is the actual scanner of the GALGAS language, and scans identifiers, keywords, delimiters, literal integers, literal characters, literal character strings, galgas type names (the '@' character followed by a sequence of letters), comments, ...

10.5 Déclarations lexicales

10.5.1 Déclaration d'un symbole terminal

The //single terminal symbol declaration// declares a name used for naming a terminal symbol. This declaration just performs declaration, not scanning. For sending this terminal symbol to the parser, it must be named in a "send" lexical instruction within a lexical rule.

The declaration associates to the terminal symbol a possibly empty list of lexical attributes and a syntax error message (not a //lexical// error message), defined by a character string.

First example :

|"$literal_integer$ error **message**

This declaration names no lexical attribute. Consequently, when the lexical send instruction "send $literal_integer$;" will be called from a lexical rule, only the terminal symbol will be sent to the parser, but not the literal integer value. The parser has no way to get the actual value : all integer values share the same terminal symbol. It is sufficient for a pure parser, however a real compiler needs the actual value.

Second example :

|"@uint unsignedValueAttribute ;
$literal_integer$!unsignedValueAttribute error **message**

In this declaration, the "unsignedValueAttribute" attribute is named in the terminal symbol declaration. So, when the lexical send instruction "send $literal_integer$;" will be called from a lexical rule, the terminal symbol will be sent to the parser together with the unsigned value of the "unsignedValueAttribute" attribute, enabling the semantic instructions to catch it.

10.5.2 Déclaration d'une liste de symboles terminaux

The //list of terminal symbol declaration// associates to a name a list of terminal symbols with a generic syntax error message. It is typically used for declaring the keywords and the delimiters.

An example of key words declaration :

| "**list** keywordList error **message**

The declared terminal symbols are : "if", "$then$", "$else$". The actual syntax error message is built from generic error message by replacing "

An other example is a delimitor list declaration :

|"**list** delimitorList error **message**

Actual scanning of a delimitor is done by a "**rule** **list**" lexical instruction.

10.5.3 Déclaration d'un attribut terminal

Lexical attributes carry values associated with terminal symbol. GALGAS handles string, unsigned, character, float lexical attributes. Every lexical attribute needs to be declared and its declaration names a GALGAS type name.

The following table summerizes the attributes features and type notation :

| ASCII String | "@string" | "| ASCII Character | "@char" | "| 32-bit Unsigned Integer | "@uint" | "0" | "uint32" || 32-bit Signed Integer | "@sint" | "0" | "sint32" || Float | "@double" | "0.0" | "double" |

In GALGAS, type names are identifiers prefixed by a "@" character.

An "@string", "@char", "@uint", "@sint", "@double" lexical attribute carry a string, character, unsigned, signed, double value.

In a "**syntax**" component, information that defines the location of the scanned terminal symbol in the input string is added to attribute value : so an "@string" object in the lexique component corresponds to an "@lstring" object in the syntax component. Location information is used by the parser and the semantic instructions for building syntax and semantic error messages that indicates //where// the error is located.

The //default value// is the one used at the beginning of every scanning loop for resetting lexical attribute.

The //corresponding C type// is useful if you want to write your own lexical actions (in C++). Please note that this correspondance is **only** available for lexical actions, and not for semantic action. The "C_String" type is a C++ class that handles mutable character strings, without being worried about memory management. It is declared in the "libpm/strings/C_string.h" file. The "uint32" type is the 32-bit unsigned integer type, and the "sint32" type is the 32-bit signed integer type.

10.5.4 Déclaration d'un message d'erreur lexicale

The //lexical error message declaration// associates a name to a string. These error messages are used in lexical actions, and define the message that are displayed when a lexical error occurs.

| "**message** decimalNumberTooLarge :

10.6 Règles lexicales

There are two kinds of //lexical rules// : - the //list lexical rule// ; - the //single lexical rule//.

10.6.1 Règle s'appuyant sur une liste

This is the simpliest form : it just names a previously defined list of terminal symbols ; for example :

|"**rule** **list** delimitorList ;"|

//Matching expressions// are the set of strings defined by the list. This rule tries to find a substring from input string at current location that matches a terminal symbol string defined in the list, sorted by decreasing length (so longest strings are tested first). On match success, //executing the rule// consists of sending the corresponding terminal symbol.

This kind of rule is typically used for scanning for a delimitor.

10.6.2 Simple règle

A //single lexical rule// has the following form :

|"**rule** //matching_expression// :
//lexical_instructions//
end **rule** ;"|

The //matching expression// defines a set of matching strings, that are tested against the substring from input string at current location. On match, the //lexical instructions// are executed.

A matching expression can be : - a one-character string (for example, "'a'" matches the "a" character) ; - an union of one-character strings, defined by a character subrange (for example, "'a'-> 'z'" matches a lower case letter) ; - a one or more characters string (for example, "- an union of above (for example : "'A'-> 'Z'| 'a'-> 'z'" matches a lower or upper case letter).

On match success, the current location is moved to designate the character after the matching string.

10.7 Instructions lexicales

10.7.1 Instruction lexicale `select`

The //lexical select instruction// is the following :

|"**select**
when //matching_expression_1_in_select// : //lexical_instructions_1//
when //matching_expression_2_in_select// : //lexical_instructions_2//
...

default //default_lexical_instructions//
end **select**;"|

A //lexical select instruction// has one or more "**when**" branches.

//matching expression_1_in_select//, //matching expression_2_in_select// conform to the defined above //matching_expression//.

This instruction tries to match the different //matching expressions// until a matching success is found. In such case, the corresponding //lexical instructions// are executed. If all matching fail, the //default lexical instructions// are executed.

10.7.2 Instruction lexicale `repeat`

The //lexical repeat instruction// is the following :

|"**repeat**
//lexical_instructions_0//
while //matching_expression_1_in_repeat// : //lexical_instructions_1//
while //matching_expression_2_in_repeat// : //lexical_instructions_2//
...
end **repeat**;"|

A //lexical while instruction// has one or more "**while**" branches.

//matching expression_1_in_repeat//, //matching expression_2_in_repeat// can be : - an expression conform to the defined above //matching_expression// ; - the " //string//" construct : the match succeeds when the //string// **is not** the current string ; - the " //string1//, //string2//, ..." construct : the match succeeds when neither of //string1//, //string2//, ... are the current string.

This instruction first executes the //lexical instructions 0//. Then, it tries to match the different //matching expressions// until a matching success is found. In such case, the corresponding //lexical instructions// are executed, then the instruction is executed again (from //lexical instructions 0//). If all matching fail, execution of this instruction is complete (excution goes on the next instruction).

10.7.3 Appel d'une action lexicale

The //lexical action call instruction// calls a C++ defined method for performing computation and checking on lexical attributes. Its syntax is the following :

|"lexical_action_name (parameter, ...);"|

or

|"lexical_action_name (parameter, ...) error message_name, ...;"|

A lexical action is designated by its name. It accepts one or more parameters, and zero, one or more messages names.

A parameter is : - either a lexical attribute, - either a lexical function call ; - either the joker character "'*'" that represents the character at current location.

A lexical action can be predefined or defined by the user. Predefined lexical actions are actually methods of "C_Lexique" class (the generated scanner is a class that inherits from this class). User defined lexical actions must be implemented as methods of the generated scanner class.

**Note that no parameter type checking, no error message count checking is performed by GALGAS. ** A parameter type error or a message count error is detected at C++ compilation stage.

10.7.4 Appel d'une fonction lexicale

The //lexical function call// calls a C++ defined method for performing computation on lexical attributes. It can only appear as parameter of a lexical action call or a parameter of an other lexical function call. Its syntax is the following :

|"lexical_function_name (parameter, ...) ;"|

A lexical function is designated by its name. It accepts one or more parameters.

A lexical function parameter is : - either a lexical attribute, - either a lexical function call ; - either the joker character "'*'" that represents the character at current location.

A lexical function can be predefined or defined by the user. Predefined lexical actions are actually methods of "C_Lexique" class (the generated scanner is a class that inherits from this class). User defined lexical functions must be implemented as methods of the generated scanner class.

**Note that no parameter type checking is performed by GALGAS. ** A parameter type error is detected at C++ compilation stage.

10.7.5 Instruction lexicale `error`

The //lexical error instruction// raises a lexical error. Its syntax is :

|"error message_name ;"|

The //message name// is the name of a previously declared lexical error message.

10.7.6 Instruction lexicale `send`

The //lexical send instruction// sends a terminal symbol to the parser. It has several forms :

=== First Form ===

|"send terminal_symbol ;"|

This instruction sends inconditionnaly the //terminal symbol// to the parser.

=== Second Form ===

|"send search //attribute_name// in //lexical_list// default terminal_symbol;"|

This instruction first search for //attribute name// value in the //lexical list//. If found, the corresponding terminal symbol is sent to the parser. If not found, the default //terminal symbol// is sent.

Several consecutive "search" are accepted, allowing sequential searching in different lists :

|"send search //attribute_name_1// in //lexical_list_1// default search //attribute_name_2// in //lexical_list_2// default terminal_symbol;"|

10.7.7 Instruction lexicale `drop`

|Available in GALGAS 1.5.6 and later.|

The //lexical drop instruction// does not send any terminal symbol to the parser. It is only significant for lexical coloring (see [[#coloring_comments|coloring comments]]).

This instruction names a terminal symbol : |"**drop** //terminal_symbol//;"|

10.7.8 Instruction lexicale `tag`

|Available in GALGAS 1.5.6 and later.|

This instruction declares a new //tag identifier//.

|"**tag** //tag_identifier//;"|

A "**tag**" instruction records a location in the scanned file. The only way to use the declared tag identifier is the [[#lexical_rewind_instruction|lexical rewind instruction]].

10.7.9 Instruction lexicale `rewind`

|Available in GALGAS 1.5.6 and later.|

|"**rewind** //tag_identifier// send //terminal_symbol//;"|

This instruction rewinds the scanned location from the tag identifier value, and sends the terminal symbol to the parser.

10.8 Routines lexicales prédéfinies

Lexical routine calls are instructions. Lexical function calls can appear as actual output parameters of routine calls and function calls. GALGAS predefines several lexical routines and several lexical functions (listed below).

A lexical routine accepts : * zero, one or more input/output or input formal arguments ; * zero, one or more error messages.

Running the `--print-predefined-lexical-actions` command line option lists all predefined routines and functions prototype.

10.8.1 Routine codePointToUnicode

```
codePointToUnicode ?@string inCodePointString
                    !@string outString
```

10.8.2 convertBinaryStringIntoBigInt

```
convertBinaryStringIntoBigInt ?@string inString
                               !@bigint outBigInt
                              error inCharacterIsNotBinaryDigitError
```

10.8.3 convertDecimalStringIntoBigInt

```
convertDecimalStringIntoBigInt ?@string inString
                                !@bigint outBigInt
                               error inCharacterIsNotDecimalDigitError
```

10.8.4 convertDecimalStringIntoSInt

```
convertDecimalStringIntoSInt ?@string inString
                              !@sint outSignedNumber
                             error inNumberTooLargeError,
                                   inCharacterIsNotDecimalDigitError
```

10.8.5 Routine convertDecimalStringIntoSInt64

```
convertDecimalStringIntoSInt64 ?@string inString
                                !@sint64 outSignedNumber
                               error inNumberTooLargeError,
                                     inCharacterIsNotDecimalDigitError
```

10.8.6 Routine convertDecimalStringIntoUInt

```
convertDecimalStringIntoUInt ?@string inString
                              !@uint outUnsignedNumber
                             error inNumberTooLargeError,
```

```
                                        inCharacterIsNotDecimalDigitError
```

10.8.7 Routine convertDecimalStringIntoUInt64

```
convertDecimalStringIntoUInt64 ?@string inString
                                  !@uint64 outUnsignedNumber
                                  error inNumberTooLargeError,
                                        inCharacterIsNotDecimalDigitError
```

10.8.8 convertHexStringIntoBigInt

```
convertHexStringIntoBigInt ?@string inString
                             !@bigint outSignedNumber
                             error inCharacterIsNotHexDigitError
```

10.8.9 Routine convertHTMLSequenceToUnicodeCharacter

```
convertHTMLSequenceToUnicodeCharacter ?@string inString
                                         !@char outUnicodeCharacter
                                         error inUnassignedHTMLSequenceError
```

10.8.10 Routine convertHexStringIntoSInt

```
convertHexStringIntoSInt ?@string inString
                           !@sint outSignedNumber
                           error inNumberTooLargeError,
                                 inCharacterIsNotHexDigitError
```

10.8.11 Routine convertHexStringIntoSInt64

```
convertHexStringIntoSInt64 ?@string inString
                             !@sint64 outSignedNumber
                             error inNumberTooLargeError,
                                   inCharacterIsNotHexDigitError
```

10.8.12 Routine convertHexStringIntoUInt

```
convertHexStringIntoUInt ?@string inString
                           !@uint outUnsignedNumber
                           error inNumberTooLargeError,
                                 inCharacterIsNotHexDigitError
```

10.8.13 Routine convertHexStringIntoUInt64

```
convertHexStringIntoUInt64 ?@string inString
                             !@uint64 outUnsignedNumber
                             error inNumberTooLargeError,
                                   inCharacterIsNotHexDigitError
```

10.8.14 Routine convertStringToDouble

```
convertStringToDouble ?@string inString
                        !@double outDouble
                        error inConversionError
```

This action tries to convert the string value of the first argument into a double value. On success, the resulting double is set to the second argument. The conversion error message is displayed on conversion error.

10.8.15 Routine convertUInt64ToSInt64

```
convertUInt64ToSInt64 !@uint64 inUnsignedNumber
                        !@sint64 outSignedNumber
                        error inNumberTooLargeError
```

If the unsigned value of the "inUnsignedNumber" argument is greater than "2⁶³-1", the error is raised. Otherwise, the value is assigned to the "ioSignedNumber" argument.

10.8.16 Routine convertUIntToSInt

```
convertUIntToSInt !@uint inUnsignedNumber
                    !@sint outSignedNumber
                    error inNumberTooLargeError
```

If the unsigned value of the "inUnsignedNumber" argument is greater than "2³¹-1", the error is raised. Otherwise, the value is assigned to the "ioSignedNumber" argument.

10.8.17 Routine convertUnsignedNumberToUnicodeChar

```
convertUnsignedNumberToUnicodeChar ?@uint inUnsignedNumber
                                     !@char outUnicodeCharacter
                                    error inUnassignedUnicodeValueError
```

10.8.18 Routine enterBinaryDigitIntoBigInt

```
enterBinaryDigitIntoBigInt ?@char inCharacter
                             ?!@bigint ioBigInt
                            error inCharacterIsNotBinDigitError
```

10.8.19 Routine enterBinDigitIntoUInt64

```
enterBinDigitIntoUInt64 ?@char inCharacter
                          ?!@uint64 ioUnsignedNumber
                          error inNumberTooLargeError,
                                inCharacterIsNotBinDigitError
```

10.8.20 Routine enterBinDigitIntoUInt64

```
enterBinDigitIntoUInt64 ?@char inCharacter
                          ?!@uint64 ioUnsignedNumber
                          error inNumberTooLargeError,
                                inCharacterIsNotBinDigitError
```

10.8.21 Routine enterCharacterIntoCharacter

```
enterCharacterIntoCharacter ?!@char ioCharacter
                              ?@char inCharacter
```

This routine performs "ioCharacter = inCharacter" assignment.

10.8.22 Routine enterCharacterIntoString

```
enterCharacterIntoString ?!@string ioString
                           ?@char inCharacter
```

Appends the character value of the second argument to the string value of the first argument. The resulting string is set to the first argument.

10.8.23 Routine enterDecimalDigitIntoBigInt

```
enterDecimalDigitIntoBigInt ?@char inCharacter
                            ?!@bigint ioBigInt
                            error inCharacterIsNotDecimalDigitError
```

10.8.24 Routine enterDigitIntoASCIIcharacter

```
enterDigitIntoASCIIcharacter ?!@char ioASCIICharacter
                             !@char inDecimalDigitCharacter
                             error inErrorCodeGreaterThan255,
                                   inErrorNotDecimalDigitCharacter
```

Build an ASCII character from its decimal definition.

First, the character value of the "inDecimalDigitCharacter" argument is tested to be a valid decimal digit, that is in one range "['0', '9']". On failure, the "inErrorNotDecimalDigitCharacter" error message is displayed. On success, the unsigned value of the "ioASCIICharacter" argument is multiplied by ten, and is added the decimal value corresponding to second argument. If the result is lower or equal to "2⁸-1", it is set to the "ioASCIICharacter" argument. Otherwise, the "inErrorCodeGreaterThan255" error is raised.

Note : this lexical action treats characters as unsigned values.

10.8.25 Routine enterDigitIntoUInt

```
enterDigitIntoUInt !@char inDecimalDigitCharacter
                   ?!@uint ioUnsignedNumber
                   error inNumberTooLargeError,
                         inCharacterIsNotDecimalDigitError
```

First, the value of "inDecimalDigitCharacter" argument is tested to be in the range "['0', '9']". On failure, the "inCharacterIsNotDecimalDigitError" error message is displayed. On success, the unsigned value of the first argument is multiplied by ten, and is added the decimal value corresponding to the "ioUnsignedNumber" argument. If the result is lower or equal to "2³²-1", it is set to the "ioUnsignedNumber" argument. Otherwise, the "inNumberTooLargeError" error is raised.

10.8.26 Routine enterDigitIntoUInt64

```
enterDigitIntoUInt64 !@char inDecimalDigitCharacter
                     ?!@uint64 ioUnsignedNumber
                     error inNumberTooLargeError,
                           inCharacterIsNotDecimalDigitError
```

First, the value of "inDecimalDigitCharacter" argument is tested to be in the range "['0', '9']". On failure, the "inCharacterIsNotDecimalDigitError" error message is displayed. On success, the unsigned value of the first argument is multiplied by ten, and is added the decimal value corresponding to the "ioUnsignedNumber" argument. If the result is lower or equal to "2⁶⁴-1", it is set to the "ioUnsignedNumber" argument. Otherwise, the "inNumberTooLargeError" error is raised.

10.8.27 Routine `enterHexDigitIntoASCIIcharacter`

```
enterHexDigitIntoASCIIcharacter ?!@char ioASCIICharacter
                                  !@char inHexDigitCharacter
                                  error inErrorCodeGreaterThan255,
                                        inErrorNotHexDigitCharacter
```

Build an ASCII character from its hexadecimal definition.

First, the character value of the "inHexDigitCharacter" argument is tested to be a valid hexadecimal digit, that is in one of the ranges "['0', '9']", "['a', 'f']", "['A', 'F']". On failure, the "inErrorNotHexDigitCharacter" error message is displayed. On success, the unsigned value of the first argument is multiplied by sixteen, and is added the hexadecimal value corresponding to "ioASCIICharacter" argument. If the result is lower or equal to "2⁸-1", it is set to the "ioASCIICharacter" argument. Otherwise, the "inErrorCodeGreaterThan255" error is raised.

Note : this lexical action treats characters as unsigned values.

10.8.28 Routine `enterHexDigitIntoBigInt`

```
enterHexDigitIntoBigInt ?@char inCharacter
                          ?!@bigint ioBigInt
                          error inCharacterIsNotHexDigitError
```

10.8.29 Routine `enterHexDigitIntoUInt`

```
enterHexDigitIntoUInt !@char inHexDigitCharacter
                       ?!@uint ioUnsignedNumber
                       error inNumberTooLargeError,
                             inCharacterIsNotHexDigitError
```

First, the character value of the "inHexDigitCharacter" argument is tested to be a valid hexadecimal digit, that in one of the the ranges "['0', '9']", "['a', 'f']", "['A', 'F']". On failure, the "inCharacterIsNotHexDigitError" error message is displayed. On success, the unsigned value of the "ioUnsignedNumber" argument is multiplied by sixteen, and is added the hexadecimal value corresponding to second argument. If the result is lower or equal to "2³²-1", it is set to the "ioUnsignedNumber" argument. Otherwise, the first error is raised.

10.8.30 Routine enterHexDigitIntoUInt64

```
enterHexDigitIntoUInt64 !@char inHexDigitCharacter
                          ?!@uint64 ioUnsignedNumber
                          error inNumberTooLargeError,
                                inCharacterIsNotHexDigitError
```

First, the character value of the "inHexDigitCharacter" argument is tested to be a valid hexadecimal digit, that in one of the the ranges "['0', '9']", "['a', 'f']", "['A', 'F']". On failure, the "inCharacterIsNotHexDigitError" error message is displayed. On success, the unsigned value of the "ioUnsignedNumber" argument is multiplied by sixteen, and is added the hexadecimal value corresponding to second argument. If the result is lower or equal to "2⁶⁴-1", it is set to the "ioUnsignedNumber" argument. Otherwise, the first error is raised.

10.8.31 Routine enterOctDigitIntoUInt

```
enterOctDigitIntoUInt !@char inString
                        ?!@uint ioUnsignedNumber
                        error inNumberTooLargeError,
                              inCharacterIsNotOctDigitError
```

10.8.32 Routine enterOctDigitIntoUInt64

```
enterOctDigitIntoUInt64 !@char inString
                          ?!@uint64 ioUnsignedNumber
                          error inNumberTooLargeError,
                                inCharacterIsNotOctDigitError
```

10.8.33 Routine multiplyUInt

```
multiplyUInt !@uint inUnsignedNumber
              ?!@uint ioUnsignedNumber
              error inResultTooLargeError
```

Multiply the "ioUnsignedNumber" value by "inUnsignedNumber" value. Detection of overflow is performed.

10.8.34 Routine multiplyUInt64

```
multiplyUInt64 !@uint inUnsignedNumber
                ?!@uint64 ioUnsignedNumber
```

```
                error inResultTooLargeError
```

Multiply the "ioUnsignedNumber" value by "inUnsignedNumber" value. Detection of overflow is performed.

10.8.35 Routine `negateSInt`

```
negateSInt ?!@sint ioNumber
           error inNumberTooLargeError
```

10.8.36 Routine `negateSInt64`

```
negateSInt64 ?!@sint64 ioNumber
             error inNumberTooLargeError
```

10.8.37 Routine `resetString`

```
resetString ?!@string ioString
```

10.9 Fonctions lexicales prédéfinies

A lexical function accepts : * zero, one or more input formal arguments.

Running the `--print-predefined-lexical-actions` command line option lists all predefined routines and functions prototype.

10.9.1 Fonction `toLower`

```
toLower ?@char inCharacter -> @char
```

If the character value of the argument is an upper case letter, this function returns the corresponding lower case letter. Otherwise, it returns the unchanged character value of the argument.

10.9.2 Fonction `toUpper`

```
toUpper ?@char inCharacter -> @char
```

If the character value of the argument is an lower case letter, this function returns the corresponding upper case letter. Otherwise, it returns the unchanged character value of the argument.

10.10 Définir vos propres actions et fonctions lexicales

You can define your own lexical actions and functions in C++ and make them available to called by lexical action call instructions.

10.10.1 Où ?

You must define your lexical actions and functions as a method of the C++ class generated by compilation of the `lexique` component. You need to modify the generated code, adding method prototype declaration in class declaration.

So that the method declaration that you added is not deleted at the time of a future compilation, define it in user zone 2 of the generated header file. For more details, see [[generated_files |file generation process page]].

For implementing your method, you can insert it in user zone 2 of the generated implementation file (for more details, see [[generated_files |file generation process page]]). Alternatively, you can implement it in any other file, provided you include the needed header files.

10.10.2 Correspondance entre les appels d'actions GALGAS et C++

This table gives the correspondance between lexical argument types and C++ types. **Note this correspondance is only available for lexical arguments**.

|"? @string" |"**const** C_String &"| |"?! @string" |"C_String &"| |"? @char" |"**const** **char**"| |"?! @char" |"**char** &"| |"? @uint" |"**const** uint32"| |"?! @uint" |"uint32 &"| |"? @sint" |"**const** sint32"| |"?! @sint" |"sint32 &"| |"? @double" |"**const** **double**"| |"?! @double" |"**double** &"|

"?" means the formal argument has input passing mode : it cannot be modified by the lexical action. "?!" means the formal argument has in/out passing mode : its value is got from the caller, can modified by the lexical action and is returned to the caller.

An error message argument corresponds to the C++ type "**const** **char** *".

In C++ generated code, the method call instruction generated by lexical action call names the lexical action name, prefixed by "scanner_routine_".

For example, consider the "convertStringToDouble" lexical action described below. This corresponds to the following method prototype :

"**void** scanner_routine_convertStringToDouble (**const** C_String &, **double** &, **const char** *);" ==== Defining Action and Function Prototype ====

The prototype must conform to the rules presented in the [[#Correspondance between Lexical Action Calls and C++ Called Methods|above]] section.

10.11 Exemples d'analyseurs lexicaux

10.11.1 Analyser des identificateurs

```
|"@string identifierString;
$identifier$!identifierString error **message** **rule** **repeat**
enterCharacterIntoString!?identifierString!*;
**while** **end** **repeat**;
send $identifier$;
**end** **rule**;"|
```

```
|"@string identifierString;
$identifier$!identifierString error **message** **rule** **repeat**
enterCharacterIntoString!?identifierString!toLower (!*);
**while** **end** **repeat**;
send $identifier$;
**end** **rule**;"|
```

10.11.2 Analyser des identificateurs et des mots-clés

```
|"@string identifierString;

$identifier$!identifierString error **message**
**list** keywordList error **message**
**rule** **repeat**
enterCharacterIntoString!?identifierString!*;
**while** **end** **repeat**;
send search identifierString in keywordList default $identifier$;
**end** **rule**;"|
```

10.11.3 Analyser des délimiteurs

```
|"**list** galgasDelimitorsList **error message**
**rule list** galgasDelimitorsList;"|
```

10.11.4 Analyser des séparateurs

```
|"**rule** **end rule**;"|
```

10.11.5 Analyser des commentaires

```
|"**rule** '#' :
**repeat**
**while** **end repeat**;
**end rule**;"|
```

10.11.6 Analyser des entiers décimaux non signés

```
|"$unsigned_literal_integer$!ulongValue **error message** $signed_literal_integer$!longValue error **mes-
sage**
**message** decimalNumberTooLarge :
**message** internalError :
**rule** enterDigitIntoUlong!?ulongValue!* error decimalNumberTooLarge, internalError;
**repeat**
**while** enterDigitIntoUlong!?ulongValue!* error decimalNumberTooLarge, internalError;
**while** **end repeat**;
**select**
**when** convertUlongToLong!?longValue!ulongValue send $signed_literal_integer$;
default
send $unsigned_literal_integer$;
**end select**;
**end rule**;"|
```

10.11.7 Analyser des entiers hexadécimaux non signés

10.11.8 Analyser des constantes caractère

```
|"$literal_char$! charValue **error message**
**message** incorrectCharConstant :
**message** ASCIIcodeTooLargeError :
**rule** **select**
**when** **select**
**when** enterCharacterIntoCharacter!?charValue!**when** enterCharacterIntoCharacter!?charValue!**when**
enterCharacterIntoCharacter!?charValue!**when** enterCharacterIntoCharacter!?charValue!**when** en-
terCharacterIntoCharacter!?charValue!**when** enterCharacterIntoCharacter!?charValue!**when** en-
terCharacterIntoCharacter!?charValue!**when** enterCharacterIntoCharacter!?charValue!**when** **re-
peat**
enterHexDigitIntoASCIIcharacter!?charValue!* error ASCIIcodeTooLargeError, internalError;
**while** **end repeat**;
default
error incorrectCharConstant;
```

```
**end select**;
**when** enterCharacterIntoCharacter!?charValue!*;
default
error incorrectCharConstant;
**end select**;
**select**
**when** send $literal_char$;
default
error incorrectCharConstant;
**end select**;
**end rule**;"|
```

10.11.9 Analyser des constantes chaîne de caractères

10.11.10 Analyser des constantes flottantes

```
|"$literal_double$!floatValue!tokenString **error message**
$.$ **error message**
**message** floatNumberConversionError :
**rule** **select**
**when**  enterCharacterIntoString!?tokenString!enterCharacterIntoString!?tokenString!enterCharacte-
rIntoString!?tokenString!*;
**repeat**
**while** enterCharacterIntoString!?tokenString!*;
**while** **end repeat**;
convertStringToDouble!tokenString!?floatValue error floatNumberConversionError;
send $literal_double$;
default
send $.$;
**end select**;
**end rule**;"|
```

10.12 *Back tracking* avec les instructions `tag` et `rewind`

|Available in GALGAS 1.5.6 and later.|

The "**tag**" and "**rewind**" instructions can be used for performing back tracking.

The first example is the way the non terminal symbols are scanned in GALGAS 1.5.6 (and later).

A non terminal is composed of a single '<' character, followed by a letter, zero, one or more letters, digits or underscore characters, is ended by a single '>' character. For example "<abcdef>" is a valid non terminal. However, "<abcdef >" is //not// a valid non terminal (because of the space before the final '>' character) : it

is considered as a '<' delimitor, followed by the "abcdef" identifier and by the '>' delimitor.

In the file "galgas/galgas_sources/galgas_scanner.ggs", the three delimitors befgging with a '<' character and the non terminal symbols are scanned by the following code :

```
"$<$ **error message** "the '<' delimitor" **style** delimitersStyle ;"
"""$non_terminal_symbol$ ! tokenString **error message** "a non terminal symbol <...>" **style** non-
TerminalStyle ;"


"**rule** '<' :"
" **tag** onlyInfDelimiter ;"
" **select**"
" **when** '=' :"
" send " **when** '<' :"
" send " **when** " **repeat**"
" enterCharacterIntoString !?tokenString !* ;"
" **while** " **end repeat** ;"
" **select**"
" **when** '>' :"
" send $non_terminal_symbol$ ;"
" default"
" **rewind** onlyInfDelimiter send $<$ ;"
" **end select** ;"
" default"
" send $<$ ;"
" **end select** ;"
"**end rule** ;"
```

The "**tag**" instruction records a scanning location. When the final '>' character is not found, the scanner is rewinded at the character following the '<' character, and the "$<$" terminal is sent. On next scanning, an identifier (or a key word) will be found.

The second examples shows how to scan for integer constants, float constants, and array bounds in Pascal : * an integer constant is a (non empty) sequence of digits ; * a float constant is a (non empty) sequence of digits, following by a dot and a (possibly empty) sequence of digits ; * an array bound is an integer constant, followed by the '..' delimitor (two dots) and an integer constant.

The problem is that "1..2" should not be scanned as a float constant, a single dot delimitor, and an integer constant.

This can be achieved by the following code :

```
"**rule** " **repeat**"
" **while** " **end repeat** ;"
" **tag** endOfIntegerConstant ;"
" **select**"
```

```
" **when** " **select**"
" **when** " **rewind** endOfIntegerConstant send $integer_constant$;"
" **when** " **repeat**"
" **while** " **end repeat**;"
" send $float_constant$;"
" default"
" send $float_constant$;"
" **end select**;"
" default"
" send $integer_constant$;"
" **end select**;"
"**end rule**;"
```

10.13 Ajouter la coloration lexicale (sur Mac uniquement)

With GALGAS, you can easily embbed your compiler in a GUI application (currently available only for Mac OS X). This application has a built-in text editor, from which you can modify, save and compile source file. With //style declarations//, you can add automatic coloring in the built-in text editor.

A //style declaration// associates a message to a style identifier. For example :

```
|"**style** keywordsStyle ->
```

The associated message is used in application preferences window as a comment of each color selection item.

A //style declaration// does not link a style identifier to any terminal symbol. You need to add this information to //single terminal symbol declaration// and //list of terminal symbols declaration// by naming the style identifier after the syntax error message :

```
|"$literal_integer$ error **message**
|"**list** delimitorList error **message**
```

10.13.1 Exemple : les styles de l'analyseur lexical GALGAS

As an example, you can take a look on GALGAS scanner, in "galgas/galgas_sources/galgas_scanner.ggs" file. The style declarations are the following :

```
|"**style** keywordsStyle ->
```

You can search for the occurrence of style identifiers, to see how they are used.

In Cocoa GALGAS application, the Color tab of the Preferences window lists all style comments, each of them being associated to a "NSColorWell" for color selection :

cocoa_galgas_color_styles.png

Note that no default color is defined in style declaration. Until you define yourself a color from Preference window, it defaults to black color.

10.13.2 Appliquer un style aux commentaires

|Available in GALGAS 1.5.6 and later.|

In GALGAS 1.5.6 and later, you can define a color for comments. Proceed as follows : - declare a new terminal symbol, for example "$comment$" ; - declare a style for this new terminal symbol ; - when a comment is scanned, use the "**drop**" instruction for naming the new terminal symbol (instead of the usual "send" instruction).

The "**drop**" instruction is only significant for syntax coloring.

For example, GALGAS comments are defined in "galgas/galgas_sources/galgas_scanner.ggs" in this way :

```
"**style** commentStyle -> "Comments :" ;"
"..."
"$comment$ error **message** "**rule** " **repeat**"
" **while** " **end repeat** ;"
" **drop** $comment$ ;"
"**end rule** ;"
```

Chapitre 11

Écrire un composant gammaire

11.1 GALGAS and Context-Free Grammars

11.2 Analyse en plusieurs phases

Chapitre 12

Graphic User Interface Component

Chapitre 13

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 **??** page **??**.

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"
}
```

13.1 Déclaration d'une option

La déclaration d'une option présente la 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 **??** page **??**) ;
- `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 caractères qui contient une description de l'option, qui sera affichée par l'option `--help` de votre compilateur.

13.2 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 `-S` 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`.

13.3 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 `-M=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 `-M=0`, comme l'option `--max-iterations-count=0` n'a aucun effet.

13.4 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, par exemple `abc def`, écrivez : `"-F=abc def"` ou `"--file-name=abc def"`.

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 14

Règle d’analyse de fichier source

III

Le système de types

Chapitre 15

Présentation du système de types

GALGAS définit :

- des types de base, définis en dur dans le langage (section **??** page **??**) ;
- des constructions permettant de construire de nouveaux types (section **??** page **??**).

15.1 Types de base

Les types de base sont :

- `@application` (page **??**), accès aux informations relatives à l'application ;
- `@bigint` (page **??**), entiers de taille illimitée ;
- `@binaryset` (page **??**), fonctions booléennes, implementées par des *Binary Decision Diagrams* ;
- `@bool` (page **??**), les booléens ;
- `@char` (page **??**), les caractères Unicode ;
- `@data` (page **??**), les séquences d'octets ;
- `@double` (page **??**), les nombres flottants correspondant au type `double` du C ;
- `@filewrapper` (page **??**), dont les objets permettent d'explorer les *filewrappers* ;
- `@location` (page **??**), objets dont la valeur désigne un texte source et un indice dans ce texte source ;
- `@object` (page **??**), dont une instance peut encapsuler toute valeur ;
- `@sint` (page **??**), entiers 32 bits signés ;

- `@sint64` (page **??**), entiers 64 bits signés ;
- `@string` (page **??**), chaînes de caractères Unicode ;
- `@stringset` (page **??**), ensembles de chaînes de caractères Unicode ;
- `@timer` (page **??**) ;
- `@type` (page **??**), dont une instance représente un type ;
- `@uint` (page **??**), entiers 32 bits non signés ;
- `@uint64` (page **??**), entiers 64 bits non signés.

15.2 Constructions de nouveaux types

Les nouveaux types qui peuvent être construits sont :

- des *types de listes*, chapitre **??** à partir de la page **??** ;
- des *types de listes ordonnées*, chapitre **??** à partir de la page **??** ;
- des *types de tableaux*, chapitre **??** à partir de la page **??** ;
- des *types de classes*, chapitre **??** à partir de la page **??** ;
- des *types énumérés*, chapitre **??** à partir de la page **??** ;
- des *types de graphes*, chapitre **??** à partir de la page **??** ;
- des *types de tables*, chapitre **??** à partir de la page **??** ;
- des *types de dictionnaires*, chapitre **??** à partir de la page **??** ;
- des *types de structures*, chapitre **??** à partir de la page **??** ;
- des *types externes*, chapitre **??** à partir de la page **??**.

15.3 Types prédéfinis

Les types prédéfinis sont :

- les types de base (section **??** page **??**) ;
- les types de structure suivants :
  - `@lbool` (page **??**) ;
  - `@lbigint` (page **??**) ;

  - `@lchar` (page **??**);
  - `@ldouble` (page **??**);
  - `@lsint` (page **??**);
  - `@lsint64` (page **??**);
  - `@lstring` (page **??**);
  - `@luint` (page **??**);
  - `@luint64` (page **??**);
  - `@range` (page **??**);
- les types de listes suivants :
  - `@2stringlist` (page **??**);
  - `@2lstringlist` (page **??**);
  - `@bigintlist` (page **??**);
  - `@functionlist` (page **??**);
  - `@lbigintlist` (page **??**);
  - `@lstringlist` (page **??**);
  - `@luintlist` (page **??**);
  - `@objectlist` (page **??**);
  - `@stringlist` (page **??**);
  - `@typelist` (page **??**);
  - `@uintlist` (page **??**);
  - `@uint64list` (page **??**).

15.4 Opérations définies pour tous les types

Tout type implémente implicitement :

- l'opérateur `==` ;
- l'opérateur `!=` ;
- le *getter* `description` ;
- le *getter* `dynamicType` ;
- le *getter* `object` .

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

15.4.1 L'opérateur ==

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

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

15.4.2 L'opérateur !=

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

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

15.4.3 Le getter `description`

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

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

15.4.4 Le getter `dynamicType`

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

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

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

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

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

Par exemple, en déclarant :

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

Et avec la séquence d'instructions suivante :

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

```
t = [a dynamicType]
log t # Affiche @B, type statique de a : @A
```

15.4.5 Le getter `object`

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

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

Chapitre 16

Le type @application

Le type `@application` ne définit que des constructeurs et des procédures de type qui permettent d'obtenir des informations sur le programme courant et son exécution.

16.1 Numéros de version

16.1.1 Constructeur `galgasVersionString`

```
constructor @application galgasVersionString -> @string
```

Ce constructeur renvoie la version du compilateur GALGAS qui a engendré cet exécutable. Pour le compilateur correspondant à cette documentation, la chaîne renvoyée est `"GALGASBETAVERSION"` :

```
let s = @application.galgasVersionString # "GALGASBETAVERSION"
```

16.1.2 Constructeur `projectVersionString`

```
constructor @application projectVersionString -> @string
```

Ce constructeur renvoie la version du projet GALGAS dont la compilation fournit cet exécutable. C'est l'information qui apparaît après le mot réservé `project` (voir section **??** page **??**), en utilisant le point «.» comme séparateur. Par exemple, si l'en-tête du projet est :

```
project (1:2:3) -> "logo" {
  ...
}
```

La chaîne renvoyée est `"1.2.3"` :

```
let s = @application.projectVersionString # "1.2.3"
```

16.2 Arguments de la ligne de commande

16.2.1 Constructeur `commandLineArgumentCount`

```
constructor @application commandLineArgumentCount -> @uint
```

Ce constructeur renvoie le nombre d'arguments de la ligne de commande.

16.2.2 Constructeur `commandLineArgumentAtIndex`

```
constructor @application commandLineArgumentAtIndex ?@uint inIndex
  -> @string
```

Ce constructeur renvoie l'argument d'indice `inIndex` de la ligne de commande. Les arguments sont indexés à partir de zéro, aussi la valeur de `inIndex` doit être strictement inférieur à la valeur retournée par `@application.commandLineArgumentCount`. Une erreur d'exécution est déclenchée dans le cas contraire.

À titre d'exemple, voici comment imprimer tous les arguments de la ligne de commande :

```
for idx in 0 ..< @application.commandLineArgumentCount do
  message "Argument " + idx + ": '"
    + @application.commandLineArgumentAtIndex {!idx} + "'\n"
end
```

16.3 Options booléennes de la ligne de commande

16.3.1 Constructeur `boolOptionNameList`

```
constructor @application boolOptionNameList -> @2stringlist
```

Ce constructeur renvoie la liste des options booléennes définie par l'application, que ces options soient nommées dans la ligne de commande ou non. Chaque option est définie par un couple, son nom de domaine et son identificateur. À titre d'exemple, voici comment imprimer la liste des options booléennes :

```
for (domain identifier) in @application.boolOptionNameList do
```

```
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
end
```

16.3.2 Constructeur boolOptionCommentString

```
constructor @application boolOptionCommentString
    ?@string inDomainName
    ?@string inOptionIdentifier -> @string
```

Ce constructeur renvoie la chaîne de commentaires associée à l'option booléenne spécifiée par son nom de domaine et son identificateur. Par exemple :

```
for (domain identifier) in @application.boolOptionNameList do
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
  message "Comment: '"
    + @application.boolOptionCommentString {!domain !identifier} + "'\n"
end
```

Une erreur d'exécution est déclenchée si l'option n'existe pas, et la chaîne renvoyée n'est pas construite.

16.3.3 Constructeur boolOptionInvocationCharacter

```
constructor @application boolOptionInvocationCharacter
    ?@string inDomainName
    ?@string inOptionIdentifier -> @char
```

Ce constructeur renvoie le caractère d'activation associé à l'option booléenne spécifiée par son nom de domaine et son identificateur.

Le caractère d'activation est le caractère qui, précédé de « - » permet l'activation de l'option sur la ligne de commande. Si l'option ne définit pas de caractère d'activation, la valeur renvoyée est NUL.

Par exemple :

```
for (domain identifier) in @application.boolOptionNameList do
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
  message "Invocation character: '"
    + @application.boolOptionInvocationCharacter {!domain !identifier} + "'\n"
end
```

Une erreur d'exécution est déclenchée si l'option n'existe pas, et le caractère renvoyé n'est pas construit.

16.3.4 Constructeur boolOptionInvocationString

```
constructor @application boolOptionInvocationString
    ?@string inDomainName
    ?@string inOptionIdentifier -> @string
```

Ce constructeur renvoie la chaîne d'activation associée à l'option booléenne spécifiée par son nom de domaine et son identificateur.

La chaîne d'activation est la chaîne qui, précédée de « - - » permet l'activation de l'option sur la ligne de commande. Si l'option ne définit pas de chaîne d'activation, la valeur renvoyée est la chaîne vide.

Par exemple :

```
for (domain identifier) in @application.boolOptionNameList do
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
  message "Invocation string: '"
    + @application.boolOptionInvocationString {!domain !identifier} + "'\n"
end
```

Une erreur d'exécution est déclenchée si l'option n'existe pas, et la chaîne renvoyée n'est pas construite.

16.3.5 Constructeur `boolOptionValue`

```
constructor @application boolOptionValue
    ?@string inDomainName
    ?@string inOptionIdentifier -> @bool
```

Ce constructeur renvoie la valeur associée à l'option booléenne spécifiée par son nom de domaine et son identificateur. Si l'option n'existe pas, le résultat n'est pas construit.

16.3.6 Procédure de type `setBoolOptionValue`

```
proc @application setBoolOptionValue
    ?@string inDomainName
    ?@string inOptionIdentifier
    ?@bool inValue
```

Ce procédure de type affecte la valeur de `inValue` à l'option booléenne spécifiée par son nom de domaine et son identificateur. Si l'option n'existe pas, cette fonction est sans effet.

16.4 Options entières de la ligne de commande

16.4.1 Constructeur `uintOptionNameList`

```
constructor @application uintOptionNameList -> @2stringlist
```

Ce constructeur renvoie la liste des options entières définie par l'application, que ces options soient nommées dans la ligne de commande ou non. Chaque option est définie par un couple, son nom de domaine et son identificateur. Par d'exemple :

```
for (domain identifier) in @application.uintOptionNameList do
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
end
```

16.4.2 Constructeur `uintOptionCommentString`

```
constructor @application uintOptionCommentString
    ?@string inDomainName
    ?@string inOptionIdentifier -> @string
```

Ce constructeur renvoie la chaîne de commentaires associée à l'option entière spécifiée par son nom de domaine et son identificateur. Par exemple :

```
for (domain identifier) in @application.uintOptionNameList do
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
  message "Comment: '"
    + @application.uintOptionCommentString {!domain !identifier} + "'\n"
end
```

Une erreur d'exécution est déclenchée si l'option n'existe pas, et la chaîne renvoyée n'est pas construite.

16.4.3 Constructeur `uintOptionInvocationCharacter`

```
constructor @application uintOptionInvocationCharacter
    ?@string inDomainName
    ?@string inOptionIdentifier -> @char
```

Ce constructeur renvoie le caractère d'activation associé à l'option entière spécifiée par son nom de domaine et son identificateur.

Le caractère d'activation est le caractère qui, précédé de « - » permet l'activation de l'option sur la ligne de commande. Si l'option ne définit pas de caractère d'activation, la valeur renvoyée est `NUL`.

Par exemple :

```
for (domain identifier) in @application.uintOptionNameList do
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
  message "Invocation character: '"
    + @application.uintOptionInvocationCharacter {!domain !identifier} + "'\n"
end
```

Une erreur d'exécution est déclenchée si l'option n'existe pas, et le caractère renvoyé n'est pas construit.

16.4.4 Constructeur `uintOptionInvocationString`

```
constructor @application uintOptionInvocationString
    ?@string inDomainName
    ?@string inOptionIdentifier -> @string
```

Ce constructeur renvoie la chaîne d'activation associée à l'option entière spécifiée par son nom de domaine et son identificateur.

La chaîne d'activation est la chaîne qui, précédée de « - - » permet l'activation de l'option sur la ligne de commande. Si l'option ne définit pas de chaîne d'activation, la valeur renvoyée est la chaîne vide.

Par exemple :

```
for (domain identifier) in @application.uintOptionNameList do
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
  message "Invocation string: '"
    + @application.uintOptionInvocationString {!domain !identifier} + "'\n"
end
```

Une erreur d'exécution est déclenchée si l'option n'existe pas, et la chaîne renvoyée n'est pas construite.

16.4.5 Constructeur `uintOptionValue`

```
constructor @application uintOptionValue
    ?@string inDomainName
    ?@string inOptionIdentifier -> @uint
```

Ce constructeur renvoie la valeur associée à l'option entière spécifiée par son nom de domaine et son identificateur. Si l'option n'existe pas, le résultat n'est pas construit.

16.4.6 Procédure de type `setUIntOptionValue`

```
proc @application setUIntOptionValue
    ?@string inDomainName
    ?@string inOptionIdentifier
    ?@uint inValue
```

Ce procédure de type affecte la valeur de `inValue` à l'option entière spécifiée par son nom de domaine et son identificateur. Si l'option n'existe pas, cette fonction est sans effet.

16.5 Options chaînes de caractères de la ligne de commande

16.5.1 Constructeur stringOptionNameList

```
constructor @application stringOptionNameList -> @2stringlist
```

Ce constructeur renvoie la liste des options chaînes de caractères définie par l'application, que ces options soient nommées dans la ligne de commande ou non. Chaque option est définie par un couple, son nom de domaine et son identificateur. Par d'exemple :

```
for (domain identifier) in @application.stringOptionNameList do
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
end
```

16.5.2 Constructeur stringOptionCommentString

```
constructor @application stringOptionCommentString
    ?@string inDomainName
    ?@string inOptionIdentifier -> @string
```

Ce constructeur renvoie la chaîne de commentaires associée à l'option chaîne de caractères spécifiée par son nom de domaine et son identificateur. Par exemple :

```
for (domain identifier) in @application.stringOptionNameList do
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
  message "Comment: '"
    + @application.stringOptionCommentString {!domain !identifier} + "'\n"
end
```

Une erreur d'exécution est déclenchée si l'option n'existe pas, et la chaîne renvoyée n'est pas construite.

16.5.3 Constructeur stringOptionInvocationCharacter

```
constructor @application stringOptionInvocationCharacter
    ?@string inDomainName
    ?@string inOptionIdentifier -> @char
```

Ce constructeur renvoie le caractère d'activation associé à l'option chaîne de caractères spécifiée par son nom de domaine et son identificateur.

Le caractère d'activation est le caractère qui, précédé de « - » permet l'activation de l'option sur la ligne de commande. Si l'option ne définit pas de caractère d'activation, la valeur renvoyée est NUL.

Par exemple :

```
for (domain identifier) in @application.stringOptionNameList do
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
  message "Invocation character: '"
    + @application.uintOptionInvocationCharacter {!domain !identifier} + "'\n"
end
```

Une erreur d'exécution est déclenchée si l'option n'existe pas, et le caractère renvoyé n'est pas construit.

16.5.4 Constructeur `stringOptionInvocationString`

```
constructor @application stringOptionInvocationString
    ?@string inDomainName
    ?@string inOptionIdentifier -> @string
```

Ce constructeur renvoie la chaîne d'activation associée à l'option chaîne de caractères spécifiée par son nom de domaine et son identificateur.

La chaîne d'activation est la chaîne qui, précédée de « - - » permet l'activation de l'option sur la ligne de commande. Si l'option ne définit pas de chaîne d'activation, la valeur renvoyée est la chaîne vide.

Par exemple :

```
for (domain identifier) in @application.stringOptionNameList do
  message "Domain: '" + domain + "', identifier: '" + identifier + "'\n"
  message "Invocation string: '"
    + @application.stringOptionInvocationString {!domain !identifier} + "'\n"
end
```

Une erreur d'exécution est déclenchée si l'option n'existe pas, et la chaîne renvoyée n'est pas construite.

16.5.5 Constructeur `stringOptionValue`

```
constructor @application stringOptionValue
    ?@string inDomainName
    ?@string inOptionIdentifier -> @string
```

Ce constructeur renvoie la valeur associée à l'option chaîne de caractères spécifiée par son nom de domaine et son identificateur. Si l'option n'existe pas, le résultat n'est pas construit.

16.5.6 Procédure de type `setStringOptionValue`

```
proc @application setStringOptionValue
    ?@string inDomainName
    ?@string inOptionIdentifier
```

```
    ?@string inValue
```

Cette procédure de type affecte la valeur de `inValue` à l'option chaîne de caractères spécifiée par son nom de domaine et son identificateur. Si l'option n'existe pas, cette fonction est sans effet.

16.6 Constructeur `system`

```
constructor @application system -> @string
```

Ce constructeur permet de savoir sur quel type de système l'application tourne en renvoyant la chaîne :

- `"unix"` sur Unix, par exemple OSX ou Linux ;
- `"windows"` sur Windows.

16.7 Procédure de type `exit`

```
proc @application exit ?@uint inErrorCode
```

L'exécution de cette procédure de type avorte immédiatement l'exécution (la fonction C `exit` est appelée). L'argument est le code d'erreur associé. Si il n'est pas construit, la valeur 1 est utilisée.

16.8 Constructeur `verboseOutput`

```
constructor @application verboseOutput -> @bool
```

Ce constructeur permet de savoir si l'indicateur de sortie verbeuse est activé ou non.

La sortie verbeuse est controllée par les options de la ligne de commande *quiet* et *verbose* (section **??** page **??**) ; leur présence dans le compilateur engendré dépend de la présence de la déclaration `%quietOutputByDefault` parmi les déclarations du fichier projet (section **??** page **??**).

Les deux options de la ligne de commande *quiet* et *verbose* s'excluent et ne peuvent pas être appelées par la construction `[option nom_composant_option.nom_option nom_info]` (voir section **??** page **??**) : c'est ce constructeur, qui s'adapte à la configuration du compilateur, qu'il faut appeler.

Par exemple :

```
if @application.verboseOutput then
  # impressions de la sortie verbeuse
end
```

16.9 Instrospection des composants lexique

16.9.1 Constructeur `keywordIdentifierSet`

```
constructor @application keywordIdentifierSet -> @stringset
```

Ce constructeur renvoie l'ensemble des identificateurs des listes de mots réservés définies dans les composants lexiques du projet. Un identificateur est composé du nom du lexique, suivi de « : », et du nom de la liste des mots réservés.

Si par exemple un projet définit le composant lexique suivant :

```
lexique monLexique {
  ...
  list mots1 ... { ... }
  ...
  list mots2 ... { ... }
  ...
}
```

Alors :

```
let theList = @application.keywordIdentifierSet
log theList # "monLexique:mots1", "monLexique:mots2"
```

16.9.2 Constructeur `keywordListForIdentifier`

```
constructor @application keywordListForIdentifier
  ?@string inIdentifier
  -> @stringlist
```

Ce constructeur renvoie le contenu de la liste désignée par `inIdentifier`. Si `inIdentifier` n'est pas une des valeurs renvoyées par le *constructeur `keywordIdentifierSet` du type `@application` – page* **??**, la liste retournée est vide.

Si par exemple un projet définit le composant lexique suivant :

```
lexique monLexique {
  ...
  list mots ... { "a", "b", "c" }
  ...
}
```

Alors :

```
let theList = @application.keywordListForIdentifier {!"monLexique:mots"}
log theList # "a", "b", "c"
```

Chapitre 17

Le type `@bigint`

Le type `@bigint` définit les entiers signés d'une taille quelconque, seulement limitée par la mémoire disponible. Ce type est simplement une interface des entiers de la librairie GMP[1].

17.1 Constante littérale

Utiliser le suffixe «`G`» pour définir une constante littérale de type `@bigint` :

```
@bigint a = 1234567890_1234567890_1234567890_G
message [a string] + "\n" # 1234567890123456789012345678 90
```

Vous pouvez utiliser le caractère de soulignement «`_`» pour séparer les chiffres.

Avec le préfixe «`0x`», vous pouvez écrire les nombres en héxadécimal :

```
@bigint a = 0x123456789ABCDEF0_123456789abcdefG
message [a hexString] + "\n" # 0x123456789ABCDEF0_123456789ABCDEF
```

Les lettres minuscules «`a`» à «`f`» et majuscules «`A`» à «`F`» sont utilisées pour définir les constantes entières en héxadécimal.

17.2 Construction

Le type `@bigint` ne définit que deux constructeurs :

- *constructeur `zero` du type `@bigint` – page* **??**;

[1] http://www.gmplib.org.

- *constructeur* `default` *du type* `@bigint` *– page* **??**.

Ces deux constructeurs renvoient un `@bigint` initilisé à 0.

Pour construire un `@bigint`, vous pouvez aussi utiliser les *getters* suivants :

- *getter* `bigint` *du type* `@bool` *(page* **??***)* ;
- *getter* `bigint` *du type* `@sint` *(page* **??***)* ;
- *getter* `bigint` *du type* `@sint64` *(page* **??***)* ;
- *getter* `bigint` *du type* `@uint` *(page* **??***)* ;
- *getter* `bigint` *du type* `@uint64` *(page* **??***)* .

17.2.1 Constructeur `zero`

```
constructor @bigint zero -> @bigint
```

Le constructeur `zero` renvoie un `@bigint` initialisé à zéro :

```
@bigint a = .zero
message [a string] + "\n" # 0
```

17.2.2 Constructeur `default`

```
constructor @bigint default -> @bigint
```

Le constructeur `default`, comme le constructeur `zero`, renvoie un `@bigint` initialisé à zéro :

```
@bigint a = .default
message [a string] + "\n" # 0
```

17.3 Comparaison

Le type `@bigint` implémente les six opérateurs de comparaison `==`, `!=`, `<`, `<=`, `>` et `>=`. Il implémente aussi les *getters* `isZero` et `sign` qui permettent de comparer un `@bigint` avec zéro.

17.3.1 Opérateurs infixés de comparaison

```
operator @bigint == @bigint -> @bool
operator @bigint != @bigint -> @bool
operator @bigint >= @bigint -> @bool
operator @bigint >  @bigint -> @bool
operator @bigint <= @bigint -> @bool
operator @bigint <  @bigint -> @bool
```

17.3.2 Getter `isZero`

```
getter isZero -> @bool
```

Ce *getter* renvoie `true` si valeur du récepteur est nulle, et `false` dans le cas contraire.

```
message [[0G isZero] ocString] + "\n" # YES
message [[1G isZero] ocString] + "\n" # NO
```

17.3.3 Getter `sign`

```
getter sign -> @sint
```

Ce *getter* renvoie :

- `-1S` si la valeur du récepteur est strictement négative ;
- `0S` si la valeur du récepteur est nulle ;
- `1S` si la valeur du récepteur est strictement positive.

```
message [[0G sign] >= 0S ocString] + "\n" # YES
message [[1G sign] < 0S ocString] + "\n" # NO
```

17.4 Conversions

Les *getters* suivants permettent de convertir un `@bigint` dans un type entier usuel :

- *getter* `sint` *du type* `@bigint` *(page* **??***)* ;
- *getter* `sint64` *du type* `@bigint` *(page* **??***)* ;
- *getter* `uint` *du type* `@bigint` *(page* **??***)* ;
- *getter* `uint64` *du type* `@bigint` *(page* **??***)* .

Ils échouent si le récepteur ne peut pas être converti sans perte. On peut utiliser les *getters* suivants pour vérifier préalablement si une conversion est possible :

- *getter* bitCountForSignedRepresentation *du type* @bigint *(page* **??***)* ;
- *getter* bitCountForUnsignedRepresentation *du type* @bigint *(page* **??***)* ;
- *getter* fitsInSInt *du type* @bigint *(page* **??***)* ;
- *getter* fitsInSInt64 *du type* @bigint *(page* **??***)* ;
- *getter* fitsInUInt *du type* @bigint *(page* **??***)* ;
- *getter* fitsInUInt64 *du type* @bigint *(page* **??***)* .

17.4.1 Getter bitCountForSignedRepresentation

```
getter bitCountForSignedRepresentation -> @uint
```

Ce *getter* permet de connaître le nombre de bits nécessaires pour écrire la valeur du récepteur dans la représentation binaire *complément à deux*.

```
message [[0G bitCountForSignedRepresentation] string] + "\n" # 1
message [[1G bitCountForSignedRepresentation] string] + "\n" # 2
message [[-1G bitCountForSignedRepresentation] string] + "\n" # 1
message [[0x8000G bitCountForSignedRepresentation] string] + "\n" # 17
message [[-0x8000G bitCountForSignedRepresentation] string] + "\n" # 16
```

Pour connaître le nombre d'octets nécessaires pour représenter la valeur du récepteur dans la représentation binaire *complément à deux*, on calcule :

```
([bigint bitCountForSignedRepresentation] - 1) / 8 + 1
```

Et pour le nombre de mots de 32 bits :

```
([bigint bitCountForSignedRepresentation] - 1) / 32 + 1
```

17.4.2 Getter bitCountForUnsignedRepresentation

```
getter bitCountForUnsignedRepresentation -> @uint
```

Ce *getter* permet de connaître le nombre de bits nécessaires pour écrire la valeur absolue du récepteur dans la représentation binaire *naturelle*.

```
message [[0G bitCountForUnsignedRepresentation] string] + "\n" # 1
message [[1G bitCountForUnsignedRepresentation] string] + "\n" # 1
message [[-1G bitCountForUnsignedRepresentation] string] + "\n" # 1
message [[0x8000G bitCountForUnsignedRepresentation] string] + "\n" # 16
message [[-0x8000G bitCountForUnsignedRepresentation] string] + "\n" # 16
```

Comme c'est la valeur absolue qui est prise en compte, le signe n'intervient pas.

Pour connaître le nombre d'octets nécessaires pour représenter la valeur absolue du récepteur dans la représentation binaire *naturelle*, on calcule :

```
([bigint bitCountForUnsignedRepresentation] - 1) / 8 + 1
```

Et pour le nombre de mots de 32 bits :

```
([bigint bitCountForUnsignedRepresentation] - 1) / 32 + 1
```

17.4.3 Getter `fitsInSInt`

```
getter fitsInSInt -> @bool
```

Ce *getter* permet de savoir si le récepteur peut être converti en `@sint`. Pour effectuer la conversion, utilisez le *getter* `sint` *du type* `@bigint` *(page* **??***)*.

```
message [[0x1234_5678G fitsInSInt] ocString] + "\n" # YES
message [[0x7FFF_FFFFG fitsInSInt] ocString] + "\n" # YES
message [[0x8000_0000G fitsInSInt] ocString] + "\n" # NO
message [[-0x8000_0000G fitsInSInt] ocString] + "\n" # YES
message [[-0x8000_0001G fitsInSInt] ocString] + "\n" # NO
```

17.4.4 Getter `fitsInSInt64`

```
getter fitsInSInt64 -> @bool
```

Ce *getter* permet de savoir si le récepteur peut être converti en `@sint64`. Pour effectuer la conversion, utilisez le *getter* `sint64` *du type* `@bigint` *(page* **??***)*.

```
message [[0x1234_5678_9ABC_DEF0G fitsInSInt64] ocString] + "\n" # YES
message [[0x7FFF_FFFF_FFFF_FFFFG fitsInSInt64] ocString] + "\n" # YES
message [[0x8000_0000_0000_0000G fitsInSInt64] ocString] + "\n" # NO
message [[-0x8000_0000_0000_0000G fitsInSInt64] ocString] + "\n" # YES
message [[-0x8000_0000_0000_0001G fitsInSInt64] ocString] + "\n" # NO
```

17.4.5 Getter `fitsInUInt`

```
getter fitsInUInt -> @bool
```

Ce *getter* permet de savoir si le récepteur peut être converti en `@uint`. Pour effectuer la conversion, utilisez le *getter* `uint` *du type* `@bigint` *(page* **??***)*.

```
message [[0x1234_5678G fitsInUInt] ocString] + "\n" # YES
message [[0x1234_5678_9G fitsInUInt] ocString] + "\n" # NO
message [[-1G fitsInUInt] ocString] + "\n" # NO
```

17.4.6 Getter fitsInUInt64

```
getter fitsInUInt64 -> @bool
```

Ce *getter* permet de savoir si le récepteur peut être converti en `@uint64`. Pour effectuer la conversion, utilisez le *getter* `uint64` *du type* `@bigint` *(page* **??***)*.

```
message [[0x1234_5678_9ABC_DEF0G fitsInUInt64] ocString] + "\n" # YES
message [[0x1234_5678_9ABC_DEF0_1G fitsInUInt64] ocString] + "\n" # NO
message [[-1G fitsInUInt64] ocString] + "\n" # NO
```

17.4.7 Getter sint

```
getter sint -> @sint
```

Ce *getter* permet de convertir le récepteur en `@sint`. Si la conversion n'est pas possible, un message d'erreur est affiché et la valeur renvoyée n'est pas construite. On peut tester si la conversion est possible en appelant le *getter* `fitsInSInt` *du type* `@bigint` *(page* **??***)*.

```
message [[-0x1234_5678G sint] hexString] + "\n" # 0xEDCBA988
```

17.4.8 Getter sint64

```
getter sint64 -> @sint64
```

Ce *getter* permet de convertir le récepteur en `@sint64`. Si la conversion n'est pas possible, un message d'erreur est affiché et la valeur renvoyée n'est pas construite. On peut tester si la conversion est possible en appelant le *getter* `fitsInSInt64` *du type* `@bigint` *(page* **??***)*.

```
message [[-0x1234_5678_9ABC_DEF0G sint64] hexString] + "\n" # 0xEDCBA98765432110
```

17.4.9 Getter uint

```
getter uint -> @uint
```

Ce *getter* permet de convertir le récepteur en `@uint`. Si la conversion n'est pas possible, un message d'erreur est affiché et la valeur renvoyée n'est pas construite. On peut tester si la conversion est possible en appelant le *getter* `fitsInUInt` *du type* `@bigint` *(page* **??***)*.

```
message [[0x1234_5678G uint] hexString] + "\n" # 0x12345678
```

17.4.10 Getter `uint64`

```
getter uint64 -> @uint64
```

Ce *getter* permet de convertir le récepteur en `@uint64`. Si la conversion n'est pas possible, un message d'erreur est affiché et la valeur renvoyée n'est pas construite. On peut tester si la conversion est possible en appelant le *getter* `fitsInUInt64` *du type* `@bigint` *(page* **??***)*.

```
message [[0x1234_5678_9ABC_DEFG uint64] hexString] + "\n" # 0x123456789ABCDEF
```

17.5 Conversions en chaîne de caractères

Plusieurs *getters* sont disponibles pour convertir un `bigint` en `@string` :

- *getter* `string` *du type* `@bigint` *(page* **??***)* ;
- *getter* `spacedString` *du type* `@bigint` *(page* **??***)* ;
- *getter* `hexString` *du type* `@bigint` *(page* **??***)* ;
- *getter* `xString` *du type* `@bigint` *(page* **??***)*.

17.5.1 Getter `string`

```
getter string -> @string
```

Ce getter renvoie la valeur du récepteur sous la forme d'une chaîne de caractères décimaux (de `0` à `9`). Si cette valeur est négative, le premier caractère est un signe `-`. Par exemple :

```
@bigint a = -1234567890_1234567890_1234567890_G
message [a string] + "\n" # -1234567890123456789012345678 90
```

17.5.2 Getter `spacedString`

```
getter spacedString ?@uint inSeparation -> @string
```

Ce getter renvoie la valeur du récepteur sous la forme d'une chaîne de caractères décimaux (de `0` à `9`). Si cette valeur est négative, le premier caractère est un signe `-`. Un espace est inséré entre `inSeparation` caractères consécutifs. Si la valeur du récepteur est négative, aucun espace n'est ajouté après le signe « - ». Par exemple :

```
message [123_456_789_012_345_678G spacedString !3] + "\n"
# "123 456 789 012 345 678"
message [-123_456_789_012_345_678G spacedString !3] + "\n"
# "-123 456 789 012 345 678"
```

17.5.3 Getter hexString

```
getter hexString -> @string
```

Ce getter renvoie la valeur du récepteur sous la forme d'une chaîne de caractères héxadécimaux (`0` à `9` , `A` à `F`). La valeur retournée est préfixée par «`0x`», qui est placé après un éventuel signe « - ». Exemple :

```
@bigint a = -1234567890_1234567890_1234567890_G
message [a hexString] + "\n" # -0x18EE90FF6C373E0EE4E3F0AD2
```

17.5.4 Getter hexStringSeparatedBy

```
getter hexStringSeparatedBy ?@char inSeparator ?@uint inGroup -> @string
```

Returns the an hexadecimal string representation of the receiver value, prefixed by the string `0x`. Groups of `inGroup` digits are separated by the `inSeparator` character.

If `inGroup` is equal to zero, a run-time error is raised.

For example :

```
let s = [0x123456789ABCDEF0G hexStringSeparatedBy !'_' !4] # 0x1234_5678_9ABC_DEF0
```

17.5.5 Getter xString

```
getter xString -> @string
```

Ce getter renvoie la valeur du récepteur sous la forme d'une chaîne de caractères héxadécimaux (`0` à `9` , `A` à `F`). Si cette valeur est négative, le premier caractère est un signe `-` . Il n'y a pas de préfixe «`0x`». Exemple :

```
@bigint a = -1234567890_1234567890_1234567890_G
message [a xString] + "\n" # -18EE90FF6C373E0EE4E3F0AD2
```

17.6 Extraction

Six *getters* d'extraction sont définis. Ils permettent d'obtenir la valeur d'un `@bigint` sous la forme d'un `@uintlist` ou d'un `@uint64list` . Ces getters sont :

- *getter* `extract8ForUnsignedRepresentation` *du type* `@bigint` *(page* **??***)* ;
- *getter* `extract8ForSignedRepresentation` *du type* `@bigint` *(page* **??***)* ;
- *getter* `extract32ForUnsignedRepresentation` *du type* `@bigint` *(page* **??***)* ;
- *getter* `extract32ForSignedRepresentation` *du type* `@bigint` *(page* **??***)* ;
- *getter* `extract64ForUnsignedRepresentation` *du type* `@bigint` *(page* **??***)* ;
- *getter* `extract64ForSignedRepresentation` *du type* `@bigint` *(page* **??***)* .

Les *getters* « `extract8...` » fournissent des mots de 8 bits, « `extract32...` » des mots de 32 bits et « `extract64...` » des mots de 64 bits. Les *getters* « `...Unsigned...` » extraient la valeur absolue du nombre, et retournent une représentation *binaire naturelle*. Les *getters* « `...Signed...` » extraient la valeur du nombre en tenant compte de son signe, et retournent une représentation *complément à deux*.

17.6.1 Getter `extract8ForUnsignedRepresentation`

```
getter extract8ForUnsignedRepresentation -> @uintlist
```

Ce *getter* permet d'obtenir la représentation binaire *naturelle* de la valeur absolue du récepteur sous la forme d'un `@uintlist`, dont la valeur de chaque élément est comprise entre 0 et 255. L'octet de poids faible est à l'indice 0, et l'octet de poids fort au dernier indice. Suivant le sens de parcours de la liste, on peut construire une représentation *little endian* ou *big endian*.

```
# Parcours dans le sens des indices croissants : little endian
@uintlist a = [0xFF_EEDD_CCBB_AA99_8877_6655_4433_2211G
  extract8ForUnsignedRepresentation
]
var s = ""
for (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x11 0x22 0x33 0x44 . . . 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF
# Parcours dans le sens des indices décroissants : big endian
s = ""
for > (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0xFF 0xEE 0xDD 0xCC 0xBB 0xAA . . . 0x44 0x33 0x22 0x11
```

Si le récepteur est nul, le vecteur retourné comprend un seul élément de valeur 0.

```
@uintlist a = [0G extract8ForUnsignedRepresentation]
var s = ""
for (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x0
```

17.6.2 Getter extract8ForSignedRepresentation

```
getter extract8ForSignedRepresentation -> @uintlist
```

Ce *getter* permet d'obtenir la représentation binaire *complément à deux* de la valeur du récepteur sous la forme d'un `@uintlist`, dont la valeur de chaque élément est comprise entre 0 et 255. L'octet de poids faible est à l'indice 0, et l'octet de poids fort au dernier indice. Suivant le sens de parcours de la liste, on peut construire une représentation *little endian* ou *big endian*.

Si la valeur du récepteur est positive, alors son bit de poids fort est zéro. Ce bit est le bit le plus significatif du dernier élément de la liste renvoyée. Dans l'exemple ci-dessus, c'est la valeur `0xFF_EEDD_..._2211G` qui est utilisée, comme pour le premier exemple du *getter* `extract8ForUnsignedRepresentation`. Comme le bit de poids fort de ce nombre est 1, l'extraction en *signé* retourne un élément de plus que l'extraction en *non signé*, élément dont la valeur est 0.

```
# Parcours dans le sens des indices croissants : little endian
@uintlist a = [0xFF_EEDD_CCBB_AA99_8877_6655_4433_2211G
  extract8ForSignedRepresentation
]
var s = ""
for (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x11 0x22 0x33 0x44 . . . 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF 0x00
# Parcours dans le sens des indices décroissants : big endian
s = ""
for > (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x00 0xFF 0xEE 0xDD 0xCC 0xBB 0xAA . . . 0x44 0x33 0x22 0x11
```

Un nombre négatif est représenté sous la forme de son complément à deux, son bit de poids fort est toujours un 1 :

```
# Parcours dans le sens des indices croissants : little endian
@uintlist a = [-0x4433_2211G extract8ForSignedRepresentation]
var s = ""
for (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0xEF 0xDD 0xCC 0xBB
# Parcours dans le sens des indices décroissants : big endian
s = ""
for > (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0xBB 0xCC 0xDD 0xEF
```

17.6.3 Getter `extract32ForUnsignedRepresentation`

```
getter extract32ForUnsignedRepresentation -> @uintlist
```

Ce *getter* permet d'obtenir la représentation binaire *naturelle* de la valeur absolue du récepteur sous la forme d'un `@uintlist`. Le mot de poids faible est à l'indice 0, et le mot de poids fort au dernier indice. Suivant le sens de parcours de la liste, on peut construire une représentation *little endian* ou *big endian*.

```
@uintlist a = [0xFF_EEDD_CCBB_AA99_8877_6655_4433_2211G
  extract32ForUnsignedRepresentation
]
# Parcours dans le sens des indices croissants : little endian
var s = ""
for (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x44332211 0x88776655 0xCCBBAA99 0x00FFEEDD
# Parcours dans le sens des indices décroissants : big endian
s = ""
for > (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x00FFEEDD 0xCCBBAA99 0x88776655 0x44332211
```

Si le récepteur est nul, le vecteur retourné comprend un seul élément de valeur 0.

```
@uintlist a = [0G extract32ForUnsignedRepresentation]
var s = ""
for (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x0
```

17.6.4 Getter extract32ForSignedRepresentation

```
getter extract32ForSignedRepresentation -> @uintlist
```

Ce *getter* permet d'obtenir la représentation binaire *complément à deux* de la valeur du récepteur sous la forme d'un `@uintlist`. L'octet de poids faible est à l'indice 0, et l'octet de poids fort au dernier indice. Suivant le sens de parcours de la liste, on peut construire une représentation *little endian* ou *big endian*.

Si la valeur du récepteur est positive, alors son bit de poids fort est zéro. Ce bit est le bit le plus significatif du dernier élément de la liste renvoyée.

```
let @uintlist a = [0xFF_EEDD_CCBB_AA99_8877_6655_4433_2211G
  extract32ForSignedRepresentation
]
# Parcours dans le sens des indices croissants : little endian
var s = ""
for (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x44332211 0x88776655 0xCCBBAA99 0x00FFEEDD
# Parcours dans le sens des indices décroissants : big endian
s = ""
for > (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x00FFEEDD 0xCCBBAA99 0x88776655 0x44332211
```

Un nombre négatif est représenté sous la forme de son complément à deux, son bit de poids fort est toujours un 1 :

```
let @uintlist a = [-0x55_4433_2211G extract32ForSignedRepresentation]
# Parcours dans le sens des indices croissants : little endian
var s = ""
for (n) in a
```

```
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0xBBCCDDEF 0xFFFFFFAA
# Parcours dans le sens des indices décroissants : big endian
s = ""
for > (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0xFFFFFFAA 0xBBCCDDEF
```

17.6.5 Getter `extract64ForUnsignedRepresentation`

```
getter extract64ForUnsignedRepresentation -> @uint64list
```

Ce *getter* permet d'obtenir la représentation binaire *naturelle* de la valeur absolue du récepteur sous la forme d'un `@uint64list`. Le mot de poids faible est à l'indice 0, et le mot de poids fort au dernier indice. Suivant le sens de parcours de la liste, on peut construire une représentation *little endian* ou *big endian*.

```
let @uint64list a = [0xFF_EEDD_CCBB_AA99_8877_6655_4433_2211G
  extract64ForUnsignedRepresentation
]
# Parcours dans le sens des indices croissants : little endian
var s = ""
for (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x8877665544332211 0xFFEEDDCCBBAA99
# Parcours dans le sens des indices décroissants : big endian
s = ""
for > (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0xFFEEDDCCBBAA99 0x8877665544332211
```

Si le récepteur est nul, le vecteur retourné comprend un seul élément de valeur 0.

```
@uint64list a = [0G extract64ForUnsignedRepresentation]
var s = ""
for (n) in a
```

```
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x0
```

17.6.6 Getter extract64ForSignedRepresentation

```
getter extract64ForSignedRepresentation -> @uint64list
```

Ce *getter* permet d'obtenir la représentation binaire *complément à deux* de la valeur du récepteur sous la forme d'un `@uintlist`. L'octet de poids faible est à l'indice 0, et l'octet de poids fort au dernier indice. Suivant le sens de parcours de la liste, on peut construire une représentation *little endian* ou *big endian*.

Si la valeur du récepteur est positive, alors son bit de poids fort est zéro. Ce bit est le bit le plus significatif du dernier élément de la liste renvoyée.

```
let @uint64list a = [0xFF_EEDD_CCBB_AA99_8877_6655_4433_2211G
  extract64ForSignedRepresentation
]
# Parcours dans le sens des indices croissants : little endian
var s = ""
for (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0x8877665544332211 0xFFEEDDCCBBAA99
# Parcours dans le sens des indices décroissants : big endian
s = ""
for > (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0xFFEEDDCCBBAA99 0x8877665544332211
```

Un nombre négatif est représenté sous la forme de son complément à deux, son bit de poids fort est toujours un 1 :

```
let @uint64list a = [-0x55_4433_2211G extract64ForSignedRepresentation]
# Parcours dans le sens des indices croissants : little endian
var s = ""
for (n) in a
  do s += [n hexString]
  between s += " "
end
```

```
message s + "\n" # 0xFFFFFFAABBCCDDEF
# Parcours dans le sens des indices décroissants : big endian
s = ""
for > (n) in a
  do s += [n hexString]
  between s += " "
end
message s + "\n" # 0xFFFFFFAABBCCDDEF
```

17.7 Arithmétique

17.7.1 Opérateurs + et - préfixés

```
operator + @bigint -> @bigint
operator - @bigint -> @bigint
```

L'opérateur « - » préfixé effectue la négation de l'expression qui le suit. L'opérateur « + » préfixé n'a aucun effet, il retourne la valeur de l'expression.

```
@bigint a = +1234567890_1234567890_1234567890_G
message [a string] + "\n" # 12345678901234567890123456789O
```

17.7.2 Getter abs

```
getter abs -> @bigint
```

Le *getter* `abs` retourne la valeur absolue.

```
@bigint a = [-1234567890_1234567890_1234567890_G abs]
message [a string] + "\n" # 12345678901234567890123456789O
```

17.7.3 Addition et soustraction

```
operator @bigint + @bigint -> @bigint
operator @bigint - @bigint -> @bigint
```

Les opérateurs « + » et « - » infixés effectuent respectivement la somme et la différence de leurs opérandes. Comme la taille des `@bigint` est non limitée, aucun débordement n'a lieu.

17.7.4 Incrémentation et décrémentation

```
operator @bigint ++
operator @bigint --
```

Le type `@bigint` accepte les opérateurs d'incrémentation `++` et de décrémentation `--`. Aucun débordement n'a lieu.

17.7.5 Multiplication

```
operator @bigint * @bigint -> @bigint
```

L'opérateur `*` infixé effectue le produit de ses opérandes. Comme la taille des `@bigint` est non limitée, aucun débordement n'a lieu.

17.8 Division

La division d'un entier n par un diviseur d retourne un quotient q et un reste r :

$$n = q * d + r, \text{ avec } 0 \leqslant \mid r \mid < \mid d \mid$$

Trois opérations différentes sont possibles, suivant que l'on veuille obtenir un quotient arrondi :

- *vers* $+\infty$, et r a un signe opposé à d ;
- *vers* $-\infty$, et r a le même signe que d ;
- *vers zéro*, et r a le même signe que n.

En C, les opérateurs de division (« / »), et de calcul du reste (« % ») utilisent un quotient arrondi *vers zéro*. L'opérateur de décalage à droite (« >> ») de n bits renvoie le quotient arrondi vers *vers* $-\infty$ de la division par 2^n. En GALGAS, les opérateurs correspondants sur les types `@uint` , `@sint` , `@uint64` et `@sint64` sont conformes à ce comportement.

Le type `@bigint` obéit aux mêmes règles :

- les opérateurs `/` et `mod` infixés effectuent la division qui calcule le quotient arrondi *vers zéro* ;
- l'opérateur `>>` infixé calcule le quotient arrondi *vers* $-\infty$ de la division par 2^n ;

De plus, trois méthodes sont disponibles, qui retournent quotient et reste de la division :

- la méthode `divideBy` retourne le le quotient arrondi *vers zéro* et le reste correspondant ;
- la méthode `floorDivideBy` retourne le le quotient arrondi *vers* $-\infty$ et le reste correspondant ;
- la méthode `ceilDivideBy` retourne le le quotient arrondi *vers* $+\infty$ et le reste correspondant.

17.8.1 Opérateur « / » infixé

```
operator @bigint / @bigint -> @bigint
```

Il effectue la division entière de l'expression de gauche par l'expression de droite et renvoie le quotient. Si l'expression de gauche est nulle, alors un message d'erreur est affiché et le résultat n'est pas construit.

```
message [(-7S) / 2S string] + "\n" # -3
message [(-7G) / 2G string] + "\n" # -3
message [(-7S) / (-2S) string] + "\n" # 3
message [(-7G) / (-2G) string] + "\n" # 3
message [7S / (-2S) string] + "\n" # -3
message [7G / (-2G) string] + "\n" # -3
```

17.8.2 Opérateur « mod » infixé

```
operator @bigint mod @bigint -> @bigint
```

Il renvoie le reste de la division entière de l'expression de gauche par l'expression de droite, telle que décrite au dessus. Si cette dernière est nulle, alors un message d'erreur est affiché et le résultat n'est pas construit.

```
message [9876543210G mod 1234567890G string] + "\n" # 90
message [(-9876543210G) mod 1234567890G string] + "\n" # -90
message [(-9876543210G) mod (-1234567890G) string] + "\n"  # -90
message [9876543210G mod (-1234567890G) string] + "\n"  # 90
message [2000S mod 183S string] + "\n" # 170
message [(-2000S) mod 183S string] + "\n" # -170
message [(-2000S) mod (-183S) string] + "\n" # -170
message [2000S mod (-183S) string] + "\n" # 170
```

17.8.3 Méthode `divideBy`

```
method @bigint divideBy ?@bigint inDivisor
                        !@bigint outQuotient
                        !@bigint outRemainder
```

Elle effectue la division dont le quotient arrondi *vers zéro*, c'est-à-dire elle combine les opérateurs « `/` » et « `mod` » en une seule opération pour retourner quotient et reste.

```
@bigint quotient
@bigint remainder
[9876543210_9876543210G divideBy
  !1234567890G
```

```
  ?quotient:quotient
  ?remainder:remainder
]
message [quotient string] + " " + remainder + "\n" # 80000000737 8280
[-9876543210_9876543210G divideBy
  !1234567890G
  ?quotient:quotient
  ?remainder:remainder
]
message [quotient string] + " " + remainder + "\n" # -80000000737 -8280
[-9876543210_9876543210G divideBy
  !-1234567890G
  ?quotient:quotient
  ?remainder:remainder
]
message [quotient string] + " " + remainder + "\n" # 80000000737 -8280
[9876543210_9876543210G divideBy
  !-1234567890G
  ?quotient:quotient
  ?remainder:remainder
]
message [quotient string] + " " + remainder + "\n" # -80000000737 8280
```

17.8.4 Méthode `floorDivideBy`

```
method @bigint floorDivideBy ?@bigint inDivisor
                             !@bigint outQuotient
                             !@bigint outRemainder
```

Elle effectue toujours la division dont le quotient arrondi *vers* $-\infty$.

```
@bigint quotient
@bigint remainder
[9876543210_9876543210G floorDivideBy
  !1234567890G
  ?quotient:quotient
  ?remainder:remainder
]
message [quotient string] + " " + remainder + "\n" # 80000000737 8280
[-9876543210_9876543210G floorDivideBy
  !1234567890G
  ?quotient:quotient
  ?remainder:remainder
```

```
]
message [quotient string] + " " + remainder + "\n" # -80000000738 1234559610
[-9876543210_9876543210G floorDivideBy
  !-1234567890G
  ?quotient:quotient
  ?remainder:remainder
]
message [quotient string] + " " + remainder + "\n" # 80000000737 -8280
[9876543210_9876543210G floorDivideBy
  !-1234567890G
  ?quotient:quotient
  ?remainder:remainder
]
message [quotient string] + " " + remainder + "\n" # -80000000738 -1234559610
```

17.8.5 Méthode `ceilDivideBy`

```
method @bigint ceilDivideBy ?@bigint inDivisor
                            !@bigint outQuotient
                            !@bigint outRemainder
```

Elle effectue toujours la division dont le quotient arrondi *vers* $+\infty$.

```
@bigint quotient
@bigint remainder
[9876543210_9876543210G ceilDivideBy
  !1234567890G
  ?quotient:quotient
  ?remainder:remainder
]
message [quotient string] + " " + remainder + "\n" # 80000000738 -1234559610
[-9876543210_9876543210G ceilDivideBy
  !1234567890G
  ?quotient:quotient
  ?remainder:remainder
]
message [quotient string] + " " + remainder + "\n" # -80000000737 -8280
[-9876543210_9876543210G ceilDivideBy
  !-1234567890G
  ?quotient:quotient
  ?remainder:remainder
]
message [quotient string] + " " + remainder + "\n" # 80000000738 1234559610
```

```
[9876543210_9876543210G ceilDivideBy
  !-1234567890G
  ?quotient:quotient
  ?remainder:remainder
]
message [quotient string] + " " + remainder + "\n" # -80000000737 8280
```

17.9 Décalages

17.9.1 Opérateur <<

```
operator @bigint << @uint -> @bigint
```

L'opérateur « `<<` » infixé effectue un décalage à gauche. L'expression de droite est toujours un `@uint`. Un décalage à gauche de n bits est sémantiquement équivalent à une multiplication par 2^n, que le nombre auquel s'applique le décalage soit signé ou non. C'est la sémantique des décalages à gauche des types `@sint` et `@sint64` :

```
message [0x1234567890G << 12 hexString] + "\n" # 0x1234567890000
message [(-0x1234567890G) << 12 hexString] + "\n" # -0x1234567890000
message [2000S << 2 string] + "\n" # 8000
message [(-2000S) << 2 string] + "\n" # -8000
```

17.9.2 Opérateur >>

```
operator @bigint >> @uint -> @bigint
```

L'opérateur « `>>` » infixé effectue un décalage à droite. L'expression de droite est toujours un `@uint` :

```
message [0x1234567890G >> 12 hexString] + "\n" # 0x1234567
message [(-0x1234567890G) >> 12 hexString] + "\n" # -0x1234567
message [2000S >> 2 string] + "\n" # 500
message [(-2000S) >> 2 string] + "\n" # -500
```

Un décalage à droite de n bits d'un nombre posifif ou négatif est sémantiquement équivalent au quotient *par défaut* d'une division par 2^n, c'est-à-dire que le reste est toujours positif ou nul.

Quelques exemples de décalage à droite de nombres positifs :

```
message [9G >> 1 string] + "\n" # 4
message [9S >> 1 string] + "\n" # 4
message [7G >> 1 string] + "\n" # 3
message [7S >> 1 string] + "\n" # 3
```

```
message [3G >> 1 string] + "\n" # 1
message [3S >> 1 string] + "\n" # 1
message [1G >> 1 string] + "\n" # 0
message [1S >> 1 string] + "\n" # 0
```

Et pour des nombres négatifs :

```
message [-9G >> 1 string] + "\n" # -5
message [-9S >> 1 string] + "\n" # -5
message [-7G >> 1 string] + "\n" # -4
message [-7S >> 1 string] + "\n" # -4
message [-3G >> 1 string] + "\n" # -2
message [-3S >> 1 string] + "\n" # -2
message [-1G >> 1 string] + "\n" # -1
message [-1S >> 1 string] + "\n" # -1
```

Dans tous les cas, la sémantique du décalage à droite du type `@bigint` est la même que celles des types `@sint` et `@sint64`.

17.10 Opérations logiques

Le type `@bigint` implémente les opérations logiques `&` (*et logique*), `|` (*ou logique*), `^` (*ou exclusif logique*) et `~` (*négation logique*). Si les opérandes sont positifs ou nuls, le comportement de ces opérateurs est celui attendu. Pour comprendre le comportement avec des opérandes négatifs, ou de signe contraire, il faut considérer que la représentation des `@bigint` est la suivante :

- la valeur d'un nombre positif ou nul est préfixée par une infinité de zéros ;
- la valeur d'un nombre strictement négatif est préfixée par une infinité de uns.

Par exemple :

- `0x1234` est représenté par `0x...01234` ;
- `-0x1234` est représenté par `0x...FEDCC`.

17.10.1 Opérateur & infixé

```
operator @bigint & @bigint -> @bigint
```

L'opérateur `&` infixé réalise un « *et logique* » entre ses opérandes. Le résultat est positif ou nul dès qu'un des deux opérandes est positif.

```
message [0x1234G & 0x4321G hexString] + "\n" # 0x220
message [-0x1234G & 0x4321G hexString] + "\n" # 0x4100
message [-0x80G & 0xFFG hexString] + "\n" # 0x80
```

Considérons le deuxième exemple et voyons comment le résultat est obtenu :

Premier opérande	0x...FEDCC	représentation théorique de -0x1234
Second opérande	0x...04321	représentation théorique de 0x4321
Résultat	0x...04100	représentation théorique de 0x4100

17.10.2 Opérateur | infixé

```
operator @bigint | @bigint -> @bigint
```

L'opérateur | infixé réalise un « *ou logique* » entre ses opérandes. Le résultat est négatif dès qu'un des deux opérandes est négatif.

```
message [0x1234G | 0x4321G hexString] + "\n" # 0x5335
message [-0x1234G | 0x4321G hexString] + "\n" # -0x1013
message [-0x80G | 0xFFG hexString] + "\n" # -0x1
```

Considérons le deuxième exemple et voyons comment le résultat est obtenu :

Premier opérande	0x...FEDCC	représentation théorique de -0x1234
Second opérande	0x...04321	représentation théorique de 0x4321
Résultat	0x...FEFED	représentation théorique de -0x1013

17.10.3 Opérateur ∧ infixé

```
operator @bigint ∧ @bigint -> @bigint
```

L'opérateur ∧ infixé réalise un « *ou exclusif logique* » entre ses opérandes. Le résultat est négatif quand les deux opérandes sont de signe contraire, et positif si ils sont de même signe.

```
message [0x1234G ∧ 0x4321G hexString] + "\n" # 0x5115
message [-0x1234G ∧ 0x4321G hexString] + "\n" # -0x5113
message [-0x80G ∧ 0xFFG hexString] + "\n" # -0x81
message [-0x80G ∧ -0xFFG hexString] + "\n" # 0x81
```

Considérons le deuxième exemple et voyons comment le résultat est obtenu :

Premier opérande	0x...FEDCC	représentation théorique de -0x1234
Second opérande	0x...04321	représentation théorique de 0x4321
Résultat	0x...FAEED	représentation théorique de -0x5113

17.10.4 Opérateur ∼ préfixé

```
operator ∼ @bigint -> @bigint
```

L'opérateur `∼` préfixé réalise la complémentation logique de son opérande. Le résultat est négatif si l'opérande est positif ou nul, et positif si il est négatif.

```
message [∼  0x1234G hexString] + "\n" # -0x1235
message [∼ -0x1234G hexString] + "\n" # 0x1233
```

Considérons le second exemple et voyons comment le résultat est obtenu :

Opérande	`0x…FEDCC`	représentation théorique de `-0x1234`
Résultat	`0x…01233`	représentation théorique de `0x1233`

17.11 Manipulation de bits

Les constructions suivantes permettent d'accéder à un bit particulier de la représentation signée en *complément à deux* de la valeur d'un `@bitint` .

Pour comprendre le comportement avec un récepteur négatif, il faut considérer, comme pour les opérateurs logiques, que la représentation des `@bigint` est la suivante :

- la valeur d'un nombre positif ou nul est préfixée par une infinité de zéros ;
- la valeur d'un nombre strictement négatif est préfixée par une infinité de uns.

Par exemple :

- `0x1234` est représenté par `0x…01234` ;
- `-0x1234` est représenté par `0x…FEDCC`.

17.11.1 Getter `bitAtIndex`

```
getter bitAtIndex ?@uint inIndex -> @bool
```

Ce *getter* permet d'obtenir la valeur d'un bit particulier de la représentation signée en *complément à deux* du récepteur. À partir d'un certain rang, la valeur obtenue pour un nombre positif est toujours `false` , et pour un nombre négatif toujours `true` .

```
message [[0x1234G bitAtIndex !7] ocString] + "\n" # NO
message [[0x1234G bitAtIndex !5] ocString] + "\n" # YES
message [[0x1234G bitAtIndex !25] ocString] + "\n" # NO
message [[-0x1234G bitAtIndex !7] ocString] + "\n" # YES
message [[-0x1234G bitAtIndex !5] ocString] + "\n" # NO
```

```
message [[-0x1234G bitAtIndex !25] ocString] + "\n" # YES
```

17.11.2 Setter setBitAtIndex

```
setter @bigint setBitAtIndex ?@bool inValue ?@uint inIndex
```

Ce *setter* permet de mettre à zéro ou à un bit particulier de la représentation signée en *complément à deux* du récepteur. Noter que cette opération ne change jamais le signe d'un nombre.

```
var a = 0x1234G
[!?a setBitAtIndex !true !14]
message [a hexString] + "\n" # 0x5234
[!?a setBitAtIndex !true !40]
message [a hexString] + "\n" # 0x10000005234
a = -0x1234G
[!?a setBitAtIndex !false !14]
message [a hexString] + "\n" # -0x5234
[!?a setBitAtIndex !false !40] # -0x10000005234
message [a hexString] + "\n"
```

Considérons le dernier exemple et voyons comment le résultat est obtenu :

Récepteur	0x...FFFF_FFFF_EDCC	représentation théorique de -0x1234
Valeur de 2^{40}	0x...0100_0000_0000	représentation théorique de 2^{40}
Valeur de $\sim 2^{40}$	0x...FEFF_FFFF_FFFF	représentation théorique de $\sim 2^{40}$
Résultat	0x...FEFF_FFFF_EDCC	représentation théorique de -0x10000005234

Le résultat est un *et logique* entre la valeur du récepteur et $\sim 2^{40}$.

17.11.3 Setter complementBitAtIndex

```
setter @bigint complementBitAtIndex ?@uint inIndex
```

Ce *setter* permet de complémenter un bit particulier de la représentation signée en *complément à deux* du récepteur. Noter que cette opération ne change jamais le signe d'un nombre.

```
var a = 0x1234G
[!?a complementBitAtIndex !14]
message [a hexString] + "\n" # 0x5234
a = -0x1234G
[!?a complementBitAtIndex !40]
message [a hexString] + "\n" # -0x10000005234
```

Considérons le dernier exemple et voyons comment le résultat est obtenu :

Récepteur	`0x...FFFF_FFFF_EDCC`	représentation théorique de `-0x1234`
Résultat	`0x...FEFF_FFFF_EDCC`	représentation théorique de `-0x10000005234`

Chapitre 18

Le type `@binaryset`

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

18.1 Constructeurs

18.1.1 Constructeur `binarySetWithBit`

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

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

Exemple :

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

18.1.2 Constructeur `binarySetWithEqualComparison`

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

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

Ce constructeur retourne un binary set qui encode la relation *a* == *b*, où *a* est encodé à partir du bit d'indice

inLeftFirstIndex jusqu'au bit d'indice *inLeftFirstIndex* + *inBitCount* - *1*, et *b* est encodé à partir du bit d'indice bit *inRightFirstIndex* jusqu'au bit d'indice *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

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

18.1.3 Constructeur `binarySetWithEqualToConstant`

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

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

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

Exemple :

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

18.1.4 Constructeur `binarySetWithGreaterOrEqualComparison`

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

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

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

Exemple :

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

18.1.5 Constructeur `binarySetWithGreaterOrEqualToConstant`

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

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

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

18.1.6 Constructeur `binarySetWithLowerOrEqualComparison`

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

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

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

Exemple :

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

18.1.7 Constructeur `binarySetWithLowerOrEqualToConstant`

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

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

The constructor returns a binary set that encodes the *a* <= *cst* relation, where *a* est encodé à partir du bit d'indice *inBitIndex* jusqu'au bit d'indice *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by the *inConstant* argument.

18.1.8 Constructeur `binarySetWithNotEqualComparison`

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

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

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

Exemple :

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

18.1.9 Constructeur `binarySetWithNotEqualToConstant`

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

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

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

18.1.10 Constructeur `binarySetWithPredicateString`

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

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

The *inBitString* argument string encodes a predicate string, such as those returned by *getter* `predicateStringValue` *du type* `@binaryset` *(page* **??***)* .

The *inBitString* argument string characters should have one of the five following values :

- `'0'` : a bit set to zero ;
- `'1'` : a bit set to one ;

- `'X'` : a don't care bit ;
- `' '` : a separator (non significant character) ;
- `'|'` : the boolean *or* operation (in infix notation).

Exemple : An empty predicate string (or a string containing only spaces) provides an empty binary set :

```
@binaryset s = .binarySetWithPredicateString !" "]
@bool b = = .s isEmptySet]; # b is true
```

A predicate string containing only 'X' characters (at least one) provides an full binary set :

```
@binaryset s = .binarySetWithPredicateString !" X X"] # Spaces are non significant
@bool b = [s isFullSet]; # b is true
```

A predicate string can encode a binary value (MSB first) :

```
@binaryset s [binarySetWithPredicateString !"1100"] # 12 in decimal
log s # Affiche <@binaryset: 1100>
```

You can use the boolean '|' operator for providing an or'ed values :

```
@binaryset s [binarySetWithPredicateString !" 1100 | 1101"]
log s # Affiche <@binaryset: 110X>
```

You can use you can use don't care bits and '|' operator together :

```
@binaryset s [binarySetWithPredicateString !"11X00X0 | 111XXX"]
log s # Affiche <@binaryset: 1100X0, 111XXX>
```

18.1.11 Constructeur `binarySetWithStrictGreaterComparison`

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

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

The constructor returns a binary set that encodes the *a* > *b* relation, where *a* est encodé à partir du bit d'indice *inLeftFirstIndex* jusqu'au bit d'indice *inLeftFirstIndex* + *inBitCount* - *1*, and *b* est encodé à partir du bit d'indice *inRightFirstIndex* to *inRightFirstIndex* + *inBitCount* - *1*.

Exemple :

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

18.1.12 Constructeur `binarySetWithStrictGreaterThanConstant`

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

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

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

18.1.13 Constructeur `binarySetWithStrictLowerComparison`

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

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

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

Exemple :

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

18.1.14 Constructeur `binarySetWithStrictLowerThanConstant`

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

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

The constructor returns a binary set that encodes the *a* < *cst* relation, where *a* est encodé à partir du bit d'indice *inBitIndex* jusqu'au bit d'indice *inBitIndex* + *inBitCount* - *1*, and *cst* is defined by the *inConstant* argument.

18.1.15 Constructeur emptyBinarySet

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

Retourne un empty `@binaryset` object.

18.1.16 Constructeur fullBinarySet

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

Returns a full `@binaryset` object.

18.2 Getters

18.2.1 Getter accessibleStates

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

Returns the set of accessible states from an initial state set. It computes the set of accessible states from the *inInitialStateSet* state set using the accessibility relation encoded by the receiver.

Exemple :

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

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

18.2.2 Getter binarySetByTranslatingFromIndex

```
getter binarySetByTranslatingFromIndex ?@uint inFirstIndex ?@uint inTranslation -> @string
```

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

18.2.3 Getter `compressedValueCount`

```
getter compressedValueCount -> @uint64
```

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

18.2.4 Getter `compressedStringValueList`

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

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

18.2.5 Getter `containsValue`

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

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

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

```
log b # Affiche <@bool:true>
```

18.2.6 Getter equalTo

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

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

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

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

18.2.7 Getter existOnBitIndex

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

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

18.2.8 Getter existsOnBitRange

```
getter existsOnBitRange ?@uint inFirstBitIndex ?@uint inBitCount -> @bool
```

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.

Exemple :

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

18.2.9 Getter existOnBitIndexAndBeyond

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

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

18.2.10 Getter forAllOnBitIndex

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

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

18.2.11 Getter `forAllOnBitIndexAndBeyond`

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

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

18.2.12 Getter `greaterOrEqualTo`

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

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

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

18.2.13 Getter `isEmpty`

```
getter isEmpty -> @bool
```

Returns a `@bool` value that indicates whether the receiver's value is the empty set : `true` if receiver's value is the empty set, and `false` otherwise.

18.2.14 Getter `isFull`

```
getter isFull -> @bool
```

Returns a `@bool` value that indicates whether the receiver's value is the full set : `true` if receiver's value is the full set, and `false` otherwise.

18.2.15 Getter `ITE`

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

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

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

18.2.16 Getter lowerOrEqualTo

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

Returns the binary set computed by applying the *lower or equal* operator on the receiver's value and the *inOperand* argument. `[a lowerOrEqualTo !b]` is `((~x) |y)`.

18.2.17 Getter notEqualTo

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

Returns the binary set computed by applying the *not equal* operator on the receiver's value and the *inOperand* argument. `[a notEqualTo !b]` is `(x ^ y)`.

18.2.18 Getter predicateStringValue

```
getter predicateStringValue -> @string
```

Returns a string representation of the receiver's value. The returned string is compatible with the *constructeur `binarySetWithPredicateString` du type `@binaryset` – page* **??**.

18.2.19 Getter strictGreaterThan

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

Returns the binary set computed by applying the *strict greater* operator on the receiver's value and the *inOperand* argument. `[a strictGreaterThan !b]` is `(x & ~y)`.

18.2.20 Getter strictLowerThan

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

Returns the binary set computed by applying the *strict lower* operator on the receiver's value and the *inOperand* argument. `[a strictLowerThan !b]` is `(~x & y)`.

18.2.21 Getter stringValueList

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

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

18.2.22 Getter `stringValueListWithNameList`

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

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

18.2.23 Getter `swap021`

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

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

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

18.2.24 Getter `swap01`

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

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

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

18.2.25 Getter `swap102`

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

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

This getter considers that the receiver encodes an *(x, y, z)* relation, where *x* is defined by bits index *0* to

inBitCount1 - 1, *y* is defined by bits index *inBitCount1* to *inBitCount1 + inBitCount2 - 1* and *z* is defined by bits index *inBitCount1 + inBitCount2* to *inBitCount1 + inBitCount2 + inBitCount3 - 1*.

18.2.26 Getter swap120

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

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

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

18.2.27 Getter swap201

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

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

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

18.2.28 Getter swap210

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

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

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

18.2.29 Getter `transitiveClosure`

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

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

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

18.2.30 Getter `transposedBy`

```
getter transposedBy ?@uintlist inVector -> @binaryset
```

Retourne la valeur transposée du récepteur. L'argument `inVector` spécifie comment la transposition s'opère : la valeur à l'indice i est l'indice de destination du bit i dans le *binaryset* renvoyé.

1er exemple. Si on veut échanger les bits 0 et 1, on écrit :

```
let vector = @uintlist {!1, !0}
let result = [myBinarySet transposedBy !vector]
```

2e exemple.

```
  let b = @binaryset.binarySetWithStrictGreaterComparison {!0 !2 !4}
    & @binaryset.binarySetWithEqualToConstant {!2 !2 !0}
  log b # <@binaryset: 000001, 00001X, 01001X, 100011>
  let vr = @uintlist {!0, !1, !4, !5, !2, !3}
  let r = [b transposedBy !vr]
  log r # <@binaryset: 000001, 00001X, 00011X, 001011>
  let vs = @uintlist {!4, !5, !0, !1, !2, !3}
  let s = [b transposedBy !vs]
  log s # <@binaryset: 010000, 100X00, 110X00, 111000>
```

La constante `b` encode la relation $A > B$, où A est encodé par les bits 0 et 1, et B par les bits 4 et 5. Les bits 2 et 3 sont fixés à 0. Dans le résultat `r`, A est encodé par les bits 0 et 1 (inchangés), B par les bits 2 et 3, et maintenant les bits 4 et 5 sont fixés à 0. Dans le résultat `s`, A est encodé par les bits 4 et 5, B par les bits 2 et 3, et les bits 0 et 1 sont fixés à 0.

18.2.31 Getter `uint64ValueList`

```
getter uint64ValueList ?@uint inBitCount -> @uint64list
```

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

18.2.32 Getter valueCount

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

Returns in an `@uint64` object the number of different values encoded by receiver, considering it uses *inBitCount* bits. No overflow test in performed.

18.3 Logical Operators

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

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

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

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

∼	Negation, Complementation

This operator Retourne un `@binaryset` object.

18.4 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 *getter* `equalTo` *du type* `@binaryset` *(page* **??***)* and *getter* `notEqualTo` *du type* `@binaryset` *(page* **??***)* that return a `@binaryset` object.

18.5 Shift Operators

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

<<	Left Shift
>>	Right Shift

Exemple :

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

Chapitre 19

Le type `@bool`

Le type `@bool` est le type booléen. Les deux mots réservés `true` et `false` sont du type `@bool` type, et dénote les valeurs *vari* et *faux*. Le seul constructeur du `@bool` type est le constructeur `default`, qui initialise un booléen à `false`.

19.1 Conversion en chaîne de caractères

19.1.1 Getter `cString`

```
getter cString -> @string
```

Retourne la chaîne `"true"` si le booléen est vrai, et la chaîne `"false"` dans le cas contraire.

19.1.2 Getter `ocString`

```
getter ocString -> @string
```

Retourne la chaîne `"YES"` si le booléen est vrai, et la chaîne `"NO"` dans le cas contraire.

19.2 Conversion en entier

19.2.1 Getter `bigint`

```
getter bigint -> @bigint
```

Retourne l'entier `1G` si le booléen est vrai, et l'entier `0G` dans le cas contraire.

```
  message [[false bigint] string] + "\n" # 0
  message [[true bigint] string] + "\n" # 1
```

19.2.2 Getter `sint`

```
getter sint -> @sint
```

Retourne l'entier `1S` si le booléen est vrai, et l'entier `0S` dans le cas contraire.

19.2.3 Getter `sint64`

```
getter sint64 -> @sint64
```

Retourne l'entier `1LS` si le booléen est vrai, et l'entier `0LS` dans le cas contraire.

19.2.4 Getter `uint`

```
getter uint -> @uint
```

Retourne l'entier `1` si le booléen est vrai, et l'entier `0` dans le cas contraire.

19.2.5 Getter `uint64`

```
getter uint64 -> @uint64
```

Retourne l'entier `1L` si le booléen est vrai, et l'entier `0L` dans le cas contraire.

19.3 Opérateurs logiques

```
operator @bool & @bool -> @bool
operator @bool | @bool -> @bool
operator @bool ^ @bool -> @bool
operator not @bool -> @bool
```

Le type `@bool` accepte les trois opérateurs suivants

- l'opérateur `&` infixé qui effectue un *et logique* ;

- l'opérateur `|` infixé qui effectue un *ou logique* ;
- l'opérateur `^` infixé qui effectue un *ou exclusif logique* ;
- l'opérateur `not` infixe qui effectue la*négation logique*.

19.4 Comparaison

Le type `@bool` implémente les six opérateurs de comparaison `==` , `!=` , `<` , `<=` , `>` et `>=` , avec `false < true` .

Chapitre 20

Le type `boolset`

Le mot-clé `boolset` permet de définir des types d'ensembles d'indicateurs booléens. Un tel objet a une sémantique de valeur.

La syntaxe de définition d'un type ensemble d'indicateurs booléens est de la forme :

```
boolset @MonEnsemble {
  # Liste de déclaration d'indicateurs, par exemple :
  indicateur0,
  indicateur1,
  indicateur2
}
```

Il n'est pas possible de définir du code dans cette déclaration : la seule possibilité est de le définir dans des extensions (chapitre **??** à partir de la page **??**).

Le nom des indicateurs doivent être différents des noms suivants : `all`, `description`, `dynamicType`, `none`, `object`. L'implémentation actuelle limite à 64 le nombre d'indicateurs qui peuvent être définis.

Pour initialiser un `boolset`, on utilise un des constructeurs définis :

```
var x = @MonEnsemble.indicateur1
log x # LOGGING x: <boolset @MonEnsemble: indicateur1>
```

Si on veut un ensemble ayant plusieurs indicateurs à vrai, on utilise l'opérateur `|`, qui effectue l'union de ses opérandes :

```
var x = @MonEnsemble.indicateur1 | @MonEnsemble.indicateur2
log x # LOGGING x: <boolset @MonEnsemble: indicateur1 indicateur2>
```

L'inférence de type permet d'éliminer les annotations de type non nécessaires :

```
var x = @MonEnsemble.indicateur1 | .indicateur2
```

Ou encore :

```
@MonEnsemble x = .indicateur1 | .indicateur2
```

Pour tester la valeur d'un indicateur, on utilise le *getter* du même nom :

```
@bool b = [x indicateur2]
```

20.1 Constructeurs

Un constructeur est défini pour chaque indicateur (section **??** page **??**).

Trois constructeurs particuliers sont implicitement définis pour tout ensemble de booléens :

- le constructeur `none` (section **??** page **??**);
- le constructeur `all` (section **??** page **??**);
- le constructeur `default` (section **??** page **??**).

20.1.1 Constructeur ayant le nom d'un indicateur

Ce constructeur définit un ensemble dont le seul booléen portant le nom de l'indicateur est vrai, les autres sont faux.

```
var x = @MonEnsemble.indicateur1
log x # LOGGING x: <boolset @MonEnsemble: indicateur1>
```

Si le contexte le permet, l'annotation de type peut être omis lors de l'appel du constructeur :

```
@MonEnsemble x = .none
```

20.1.2 Constructeur none

Ce constructeur définit un ensemble dont tous les booléens sont faux.

```
var x = @MonEnsemble.none
log x # LOGGING x: <boolset @MonEnsemble:>
```

Si le contexte le permet, l'annotation de type peut être omis lors de l'appel du constructeur :

```
@MonEnsemble x = .none
```

20.1.3 Constructeur `all`

Ce constructeur définit un ensemble dont tous les booléens sont vrais.

```
var x = @MonEnsemble.all
log x # LOGGING x: <boolset @MonEnsemble: indicateur0 indicateur1 indicateur2>
```

Si le contexte le permet, l'annotation de type peut être omis lors de l'appel du constructeur :

```
@MonEnsemble x = .all
```

20.1.4 Constructeur `default`

Le constructeur `default` est défini implicitement, et a la même signification que le constructeur none.

```
var x = @MonEnsemble.default
log x # LOGGING x: <boolset @MonEnsemble:>
```

20.2 Getters

Un *getter* est défini pour chaque indicateur (section **??** page **??**) : il permet de tester un indicateur.

Deux getters particuliers sont implicitement définis pour tout ensemble de booléens :

- le getter none (section **??** page **??**);
- le getter `all` (section **??** page **??**).

20.2.1 Getter ayant le nom d'un indicateur

Ce getter permet d'obtenir la valeur de l'indicateur nommé.

```
var x = @MonEnsemble.indicateur1
var b = [x indicateur1]
log b # LOGGING b: <@bool:true>
b = [x indicateur2]
log b # LOGGING b: <@bool:false>
```

20.2.2 Getter none

Ce getter renvoie `true` si tous les indicateurs sont faux.

Expression	Signification
`a & b`	Intersection : ensemble des indicateurs appartenant à `a` et à `b`.
`a | b`	Union : ensemble des indicateurs appartenant à `a` ou à `b`.
`a ∧ b`	Exclusion : ensemble des indicateurs appartenant soit à `a`, soit à `b`.
`a - b`	Différence : ensemble des indicateurs appartenant à `a` et n'appartenant pas à `b`.
`a == b`	Test d'égalité
`a != b`	Test d'inégalité

Tableau 20.1 – *Opérateurs infixes des types `boolset`*

Expression	Signification
`~ a`	Complémentation : est équivalent à `.all - a`.

Tableau 20.2 – *Opérateur préfixe des types `boolset`*

```
var x = @MonEnsemble.none
var b = [x none]
log b # LOGGING b: <@bool:true>
```

20.2.3 Getter `all`

Ce getter renvoie `true` si tous les indicateurs sont vrais.

```
var x = @MonEnsemble.all
var b = [x all]
log b # LOGGING b: <@bool:true>
```

20.3 Opérateurs infixes

Les opérateurs infixes du tableau **??** sont définis pour tout `boolset`.

20.4 Opérateur préfixe

Un seul opérateur préfixe est défini pour tout `boolset` (tableau **??**).

Chapitre 21

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 :

```
var myChar = '\u03A0' # The 'SIGMA' character
```

Note : an unassigned point code (as `'\FFFF'`) raises a compile-time error.

Character Constant	Meaning
`'\Uabcdxyzt'`	An Unicode Character

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

```
var myChar = '\U00010170' # 'GREEK ACROPHONIC NAXIAN FIVE HUNDRED' character
```

Note : an unassigned point code (as `'\U0000FFFF'`) raises a compile-time error.

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

21.1 Constructors

21.1.1 Constructeur `replacementCharacter`

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

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

21.1.2 Constructeur `unicodeCharacterFromRawKeyboard`

```
constructor unicodeCharacterFromRawKeyboard -> @char
```

Retourne un objet `@char` obtenu en lisant le clavier. Tout caractère Unicode entré est retourné immédiatement.

Note. Ce constructeur n'est pas implémenté pour Windows. L'appel engendre l'erreur « *@char unicodeCharacterFromRawKeyboard constructor is not implemented for Windows* », et renvoie une valeur poison.

21.1.3 Constructeur `unicodeCharacterWithUnsigned`

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

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

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 *getter* `isUnicodeValueAssigned` *du type* `@uint` *(page* **??***)* .

21.2 Getters

21.2.1 Getter `isalnum`

```
getter isalnum -> @bool
```

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

21.2.2 Getter `isalpha`

```
getter isalpha -> @bool
```

Returns an `@bool` value indicating whether the receiver'value represents an ASCII letter : `true` if the receiver'value represents an ASCII letter (between `'A'` and `'Z'` , or between `'a'` and `'z'`), and `false` otherwise.

21.2.3 Getter `iscntrl`

```
getter iscntrl -> @bool
```

Returns an `@bool` value indicating whether the receiver'value represents an ASCII control character : `true` if the receiver'value represents an ASCII control character (strictly before the *SPACE* character), and `false` otherwise.

21.2.4 Getter `isdigit`

```
getter isdigit -> @bool
```

Returns an `@bool` value indicating whether the receiver'value represents an ASCII digit : `true` if the receiver'value represents an ASCII digit (between `'0'` and `'9'`), and `false` otherwise.

21.2.5 Getter `islower`

```
getter islower -> @bool
```

Returns an `@bool` value indicating whether the receiver'value represents an ASCII lowercase ASCII letter : `true` if the receiver'value represents an ASCII lowercase letter (between `'a'` and `'z'`), and `false` otherwise.

21.2.6 Getter `isUnicodeCommand`

```
getter isUnicodeCommand -> @bool
```

Returns an `@bool` value indicating whether the receiver'value represents an Unicode command : `true` if the receiver'value represents an Unicode command, and `false` otherwise.

21.2.7 Getter `isUnicodeLetter`

```
getter isUnicodeLetter -> @bool
```

Returns an `@bool` value indicating whether the receiver'value represents an Unicode letter : `true` if the receiver'value represents an Unicode letter, and `false` otherwise.

21.2.8 Getter `isUnicodeMark`

```
getter isUnicodeMark -> @bool
```

Returns an `@bool` value indicating whether the receiver'value represents an Unicode mark character : `true` if the receiver'value represents an Unicode mark character, and `false` otherwise.

21.2.9 Getter `isUnicodePunctuation`

```
getter isUnicodePunctuation -> @bool
```

Returns an `@bool` value indicating whether the receiver'value represents an Unicode punctuation character : `true` if the receiver'value represents an Unicode punctuation character, and `false` otherwise.

21.2.10 Getter `isUnicodeSeparator`

```
getter isUnicodeSeparator -> @bool
```

Returns an `@bool` value indicating whether the receiver'value represents an Unicode separator character : `true` if the receiver'value represents an Unicode separator character, and `false` otherwise.

21.2.11 Getter `isUnicodeSymbol`

```
getter isUnicodeSymbol -> @bool
```

Returns an `@bool` value indicating whether the receiver'value represents an Unicode symbol character : `true` if the receiver'value represents an Unicode symbol character, and `false` otherwise.

21.2.12 Getter `isupper`

```
getter isupper -> @bool
```

Returns an `@bool` value indicating whether the receiver'value represents an ASCII uppercase ASCII letter : `true` if the receiver'value represents an ASCII uppercase letter (between `'A'` and `'Z'`, and `false` otherwise.

21.2.13 Getter `string`

```
getter string -> @string
```

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

21.2.14 Getter `uint`

```
getter uint -> @uint
```

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

21.2.15 Getter `unicodeName`

```
getter unicodeName -> @string
```

Returns the unicode name of the receiver's value : for an decimal string representation of the receiver's value, see the *getter* `hexString` *du type* `@uint` *(page* **??***)*; for a decimal string representation of the receiver's value, see the *getter* `string` *du type* `@uint` *(page* **??***)* .

Exemple :

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

21.2.16 Getter unicodeToLower

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

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

Exemple :

```
['Æ' unicodeToLower] # returns 'æ'
['æ' unicodeToLower] # returns 'æ'
```

21.2.17 Getter unicodeToUpper

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

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

Exemple :

```
['Æ' unicodeToUpper] # returns 'Æ'
['æ' unicodeToUpper] # returns 'Æ'
```

21.2.18 Getter utf8Length

```
getter utf8Length -> @uint
```

Returns the number of bytes of the UTF-8 representation of the receiver, that is :

- 1 for code points lower than `0x80`;
- 2 for code points greater or equal than `0x80` and lower than `0x800`;
- 3 for code points greater or equal than `0x800` and lower than `0x10000`;
- 4 for code points greater or equal than `0x10000`.

21.3 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 22

Le type `@data`

Le type `@data` est un buffer d'octets. Il peut être utilisé pour lire et écrire des fichiers binaires.

22.1 Constructeurs

22.1.1 Constructeur `dataWithContentsOfFile`

```
constructor dataWithContentsOfFile ?@string inFilePath -> @data
```

Ce constructeur instancie un objet `@data` avec le contenu du fichier désigné par `inFilePath`. Si le fichier n'existe pas, une erreur d'exécution est déclenchée et le constructeur renvoie une valeur poison.

22.1.2 Constructeur `emptyData`

```
constructor emptyData -> @data
```

Ce constructeur instancie un objet `@data` vide.

22.2 Getters

22.2.1 Getter `count`

```
getter count -> @uint
```

Ce *getter* renvoie le nombre d'octets du récepteur.

22.2.2 Getter `cStringRepresentation`

```
getter cStringRepresentation -> @string
```

Ce *getter* renvoie la valeur du récepteur sous la forme d'une liste d'octets séparés par des virgules. Chaque octet est écrit en décimal. Toutes les 16 valeurs, un retour-chariot est inséré.

22.2.3 Getter `length`

```
getter length -> @uint # Obsolète, utiliser count
```

Ce *getter* renvoie le nombre d'octets du récepteur.

22.3 Méthodes

22.3.1 Méthode `writeToExecutableFile`

```
method writeToExecutableFile ?@string inFilePath
```

Cette méthode écrit le contenu du récepteur dans le fichier désigné par `inFilePath`, et rend ce fichier exécutable.

22.3.2 Méthode `writeToFile`

```
method writeToFile ?@string inFilePath
```

Cette méthode écrit le contenu du récepteur dans le fichier désigné par `inFilePath`.

22.3.3 Méthode `writeToFileWhenDifferentContents`

```
method writeToFileWhenDifferentContents
  ?@string inFilePath
  !@bool outFileModified
```

Cette méthode écrit le contenu du récepteur dans le fichier désigné par `inFilePath`, uniquement si la valeur du récepteur est différente du contenu du fihier. La variable `outFileModified` est retournée à l'appelant, et permet de savoir si le fichier a été modifié ou non.

22.4 Setters

22.4.1 Setter `appendByte`

```
setter appendByte ?@uint inValue
```

Ce *setter* ajoute la valeur de `inValue` à la fin du récepteur. Comme un objet de `@data` est un tableau d'octets, `inValue` doit être compris entre 0 et 255. Si il est supérieur à 255, une erreur d'exécution est déclenchée.

22.4.2 Setter appendData

```
setter appendData ?@data inData
```

Ce *setter* ajoute la valeur de `inData` à la fin du récepteur.

22.4.3 Setter appendShortBE

```
setter appendShortBE ?@uint inValue
```

Pour ce *setter*, `inValue` doit être compris entre 0 et $2^{16} - 1$, c'est-à-dire réprésentable par un entier non signé sur deux octets. Si ce n'est pas le cas, une erreur d'exécution est déclenchée. Si c'est le cas, deux octets sont ajoutés à la fin du récepteur, d'abord l'octet de poids fort, puis l'octet de poids faible.

22.4.4 Setter appendShortLE

```
setter appendShortLE ?@uint inValue
```

Pour ce *setter*, `inValue` doit être compris entre 0 et $2^{16} - 1$, c'est-à-dire réprésentable par un entier non signé sur deux octets. Si ce n'est pas le cas, une erreur d'exécution est déclenchée. Si c'est le cas, deux octets sont ajoutés à la fin du récepteur, d'abord l'octet de poids faible, puis l'octet de poids fort.

22.4.5 Setter appendUIntBE

```
setter appendUIntBE ?@uint inValue
```

Ce *setter* ajoute la valeur de `inValue` à la fin du récepteur, sous la forme de quatre octets, en commençant par l'octet de poids fort.

22.4.6 Setter appendUIntLE

```
setter appendUIntLE ?@uint inValue
```

Ce *setter* ajoute la valeur de `inValue` à la fin du récepteur, sous la forme de quatre octets, en commençant par l'octet de poids faible.

22.4.7 Setter `appendUTF8String`

```
setter appendUTF8String ?@string inValue
```

Ce *setter* ajoute la valeur de `inValue` à la fin du récepteur, sous la forme d'une chaîne de caractères UTF-8, y compris le zéro final.

22.5 Énumération des valeurs

Un objet de type `@data` est énumérable par une instruction `for` (section **??** page **??**).

Chapitre 23

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 (.)

23.1 Constructor

23.1.1 Constructeur doubleWithBinaryImage

```
constructor doubleWithBinaryImage ?@uint inValue -> @double
```

Returns a double object from the binary image of the argument.

23.1.2 Constructeur pi

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

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

23.2 Getters

23.2.1 Getter binaryImage

```
getter binaryImage -> @uint64
```

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

23.2.2 Getter cos

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

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

23.2.3 Getter sin

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

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

23.2.4 Getter sint

```
getter sint -> @sint
```

Returns the receiver's value in an `@sint` (page **??**) (32-bit signed integer) object : if receiver's value is outside `@sint` bounds, a runtime error is raised.

23.2.5 Getter sint64

```
getter sint64 -> @sint64
```

Returns the receiver's value in an `@sint64` (page **??**) (64-bit signed integer) object : if receiver's value is outside `@sint64` bounds, a runtime error is raised.

23.2.6 Getter string

```
getter string -> @string
```

Returns a decimal string representation of the receiver's value (this getter never fails).

23.2.7 Getter tan

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

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

23.2.8 Getter uint

```
getter uint -> @uint
```

Returns the receiver's value in an @uint (page **??**) (32-bit unsigned integer) object : if receiver's value is outside @uint bounds, a runtime error is raised.

23.2.9 Getter uint64

```
getter uint64 -> @uint64
```

Returns the receiver's value in an @uint64 (page **??**) (64-bit unsigned integer) object : if receiver's value is outside @uint64 bounds, a runtime error is raised.

23.3 Arithmétique

23.3.1 Opérateurs infixés

Le type @double accepte les opérateurs arithmétiques infixés suivants :

- + , addition ;
- - , soustraction ;
- * , multiplication ;
- / , division, une erreur d'exécution est déclenchée si le diviseur est nul ;
- mod , calcul du reste, une erreur d'exécution est déclenchée si le diviseur est nul ;
- &/ , division, qui retourne zéro si le diviseur est nul.

Ces opérateurs exigent que les deux opérandes soient des objets du même type @double .

23.3.2 Opérateurs préfixés

Le type @double accepte les opérateurs arithmétiques préfixés suivants :

- + , qui retourne simplement la valeur de l'opérande ;
- - , négation arithmétique.

La valeur renvoyée est du même type @double .

23.3.3 Instructions

Le type `@double` accepte les instructions arithmétiques suivantes :

- `+=` , addition ;
- `-=` , soustraction ;
- `*=` , multiplication ;
- `/=` , division.

`x+=y` est équivalent à `x=x+y` ; `x-=y` est équivalent à `x=x-y` . La variable cible `x` , comme l'expression source `y` doivent être du même type `@double` .

23.4 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 24

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 **??** à partir de la page **??**).

24.1 Constructor

24.2 Setter

24.2.1 Setter `setCurrentDirectory`

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

24.3 Getters

24.3.1 Getter `allTextFilePathes`

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

24.3.2 Getter `allDirectoryPathes`

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

24.3.3 Getter currentDirectory

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

24.3.4 Getter allFilePathesWithExtension

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

24.3.5 Getter directoryExistsAtPath

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

24.3.6 Getter fileExistsAtPath

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

24.3.7 Getter textFileContentsAtPath

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

24.3.8 Getter binaryFileContentsAtPath

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

24.3.9 Getter absolutePathForPath

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

Chapitre 25

Le type `@location`

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

25.1 Constructeurs

25.1.1 Constructeur `here`

```
constructor here -> @location
```

Le constructeur `here` crée un objet de type `@location` qui désigne le dernier *token* analysé. Ainsi, si l'on écrit :

```
  $token$
  ...
  let currentLocation = @location.here
```

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

25.1.2 Constructeur `next`

```
constructor next -> @location
```

Le constructeur `next` crée un objet de type `@location` qui désigne le prochain *token* analysé. Ainsi, si l'on écrit :

```
let currentLocation = @location.next
...
$token$
...
```

La position capturée est le token correspondant à `$token$`. Si `next` est appelé alors que la chaîne source est complètement analysée, la position capturée est le dernier caractère du texte source.

25.1.3 Constructeur `nowhere`

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

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

The returned object responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)*.

25.2 Getters

25.2.1 Getter `column`

```
getter column -> @uint # Obsolete, use endColumn
```

Returns an `@uint` value containing the column of the receiver's value; this getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)*.

25.2.2 Getter `endColumn`

```
getter endColumn -> @uint
```

Returns an `@uint` value containing the column of the receiver's end location; this getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)*.

25.2.3 Getter `endLine`

```
getter endLine -> @uint
```

Returns an `@uint` value containing the end line of the receiver's end location. This getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)*
.

25.2.4 Getter endLocationIndex

```
getter endLocationIndex -> @uint
```

Returns an `@uint` value containing the the offset from the the end of the source of the location defined by receiver's value. This getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)* .

25.2.5 Getter endLocationString

```
getter endLocationString -> @string
```

Returns an `@string` object that contains a string representation of the end location defined by receiver's value. This getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)* .

25.2.6 Getter isNowhere

```
getter isNowhere -> @bool
```

Returns an `@bool` value indicating whether the receiver'value points out a source location or does not. This getter returns `true` if the receiver's value does not point out an actual location in a text source (i.e. it has been constructed using the nowhere constructor), and `false` if the receiver's value points out an actual location in a text source (i.e. it has been constructed using the `here` keyword.

25.2.7 Getter line

```
getter line -> @uint  # Obsolete, use endLine
```

Returns an `@uint` value containing the end line of the receiver's value. This getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)* .

25.2.8 Getter locationIndex

```
getter locationIndex -> @uint # Obsolete, use endLocationIndex
```

Returns an `@uint` value containing the the offset from the the beginning of the source of the location defined by receiver's value. This getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)* .

25.2.9 Getter locationString

```
getter locationString -> @string # Obsolete use endLocationString
```

Returns an `@string` object that contains a string representation of the end location defined by receiver's value. This getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)* .

25.2.10 Getter `startColumn`

```
getter startColumn -> @uint
```

Returns an `@uint` value containing the column of the receiver's start location; this getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)* .

25.2.11 Getter `startLine`

```
getter startLine -> @uint
```

Returns an `@uint` value containing the start line of the receiver's value. This getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)* .

25.2.12 Getter `startLocationIndex`

```
getter startLocationIndex -> @uint
```

Returns an `@uint` value containing the the offset from the the end of the source of the location defined by receiver's value. This getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)* .

25.2.13 Getter `startLocationString`

```
getter startLocationString -> @string
```

Returns an `@string` object that contains a string representation of the start location defined by receiver's value. This getter raises a run-time error if the receiver's value responds `true` to the *getter* `isNowhere` *du type* `@location` *(page* **??***)* .

25.2.14 Getter `union`

```
getter union ?@location inOther -> @location
```

L'objet courant et l'argument doivent concerner le même source. Sinon, une erreur d'exécution est déclenchée. Cette fonction retourne l'union des intervalles définis par l'objet courant et l'argument.

Chapitre 26

Le type `@function`

Le type `@function` permet de faire l'inventaire des fonctions définies dans votre projet GALGAS et de les appeler de manière indirecte. Un objet de type `@function` est une référence à une fonction du projet GALGAS, et permet de l'appeler de manière indirecte.

Pour faire l'inventaire des fonctions : *constructeur `functionList` du type `@function` – page* **??**.

Pour savoir si une fonction d'un certain nom existe : *constructeur `isFunctionDefined` du type `@function` – page* **??**.

Pour instancier un objet `@function` qui référence une fonction : *constructeur `functionWithName` du type `@function` – page* **??**, ou exploiter la liste retournée par le *constructeur `functionList` du type `@function` – page* **??**.

Pour connaître le type des arguments et le type retourné par une fonction : *getter* `formalParameterTypeList` *du type* `@function` *(page* **??***)* et *getter* `resultType` *du type* `@function` *(page* **??***)* .

Pour appeler une fonction : *getter* `invoke` *du type* `@function` *(page* **??***)* .

26.1 Constructeurs

26.1.1 Constructeur `functionList`

```
constructor functionList -> @functionlist
```

Ce constructeur renvoie la liste de toute les fonctions définies dans le projet GALGAS.

26.1.2 Constructeur `functionWithName`

```
constructor functionWithName ?let @string inFunctionName -> @function
```

Ce constructeur renvoie un objet de type `@function` permettant d'appeler de manière indirecte la fonction dont le nom est `inFunctionName`. Si il n'y a pas de fonction de ce nom, une erreur d'exécution est déclenchée, et une valeur *poison* est retournée. Pour savoir si une fonction existe, utiliser le *constructeur `isFunctionDefined` du type `@function` – page* **??**.

26.1.3 Constructeur `isFunctionDefined`

```
constructor isFunctionDefined ?let @string inFunctionName -> @bool
```

Ce constructeur permet de savoir si une fonction dont le nom est `inFunctionName` existe.

26.2 Getters

26.2.1 Getter `formalParameterTypeList`

```
getter formalParameterTypeList -> @typelist
```

Ce *getter* renvoie la liste des types des arguments formels de la fonction désignée par le récepteur. Une fonction n'admet que des arguments formels en entrée, aussi le mode de passage est connu et n'est pas renvoyé par ce *getter*.

26.2.2 Getter `invoke`

```
getter invoke ?@objectlist inParameters
              ?@location inErrorLocation -> @object
```

Ce *getter* appelle la fonction désignée par le récepteur avec la liste de paramètres effectifs `inParameters`. La valeur renvoyée par ce *getter* est la valeur renvoyée par la fonction appelée. Si liste de paramètres effectifs `inParameters` est invalide (nombre incorrect d'éléments, type des arguments ne correspondant pas), une erreur d'exécution est déclenchée, en signalant la position de l'erreur grâce à `inErrorLocation`.

26.2.3 Getter `resultType`

```
getter resultType -> @type
```

Ce *getter* renvoie le type de la valeur retournée par la fonction désignée par le récepteur.

Chapitre 27

Le type `@object`

Chapitre 28

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.

28.1 Constructors

28.1.1 Constructeur `min`

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

Returns an `@sint` object that the minimum value of the 32-bit signed range (-2^{31}).

28.1.2 Constructeur `max`

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

Returns an `@sint` object that the maximum value of the 32-bit signed range ($2^{31} - 1$).

28.2 Getters

28.2.1 Getter bigint

```
getter bigint -> @bigint
```

Ce *getter* permet de convertir un `@sint` en `@bigint`. Comme la plage des valeurs des `bigint` n'est limitée que par la mémoire disponible, il n'échoue jamais.

```
  message [[-1234S bigint] string] + "\n" # -1234
```

28.2.2 Getter double

```
getter double -> @double
```

Returns the receiver's value converted in a `@double` object. As a 32-bit integer value can always be converted in a `@double` value, this getter never fails.

28.2.3 Getter hexStringSeparatedBy

```
getter hexStringSeparatedBy ?@char inSeparator ?@uint inGroup -> @string
```

Returns the an hexadecimal string representation of the receiver value, prefixed by the string `0x`. Groups of `inGroup` digits are separated by the `inSeparator` character.

If `inGroup` is equal to zero, a run-time error is raised.

For example :

```
let s = [0x12345678S hexStringSeparatedBy !'_' !2] # 0x12_34_56_78
```

28.2.4 Getter sint64

```
getter sint64 -> @sint64
```

Returns the receiver's value in an `@sint64` (page **??**) (64-bit signed integer) object. As a 32-bit signed value can always be converted in a 64-bit signed value, this getter never fails.

This getter is the only way to convert an `@sint` (page **??**) object into an `@sint64` (page **??**) object.

28.2.5 Getter string

```
getter string -> @string
```

Returns a decimal string representation of the receiver's value. For an hexadecimal string representation of the receiver's value, see *getter* `hexString` *du type* `@uint` *(page* **??***)* and *getter* `xString` *du type* `@uint` *(page* **??***)* .

28.2.6 Getter `uint`

```
getter uint -> @uint
```

Returns the receiver's value in an `@uint` (page **??**) (32-bit unsigned integer) object. An error is raised is receiver's value is negative.

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

28.2.7 Getter `uint64`

```
getter uint64 -> @uint64
```

Returns the receiver's value in an `@uint64` (page **??**) (64-bit unsigned integer) object. An error is raised is receiver's value is negative.

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

28.3 Arithmétique

28.3.1 Opérateurs infixés

Le type `@sint` accepte les opérateurs arithmétiques infixés suivants :

- `+` , addition, une erreur d'exécution est déclenchée en cas de débordement ;
- `-` , soustraction, une erreur d'exécution est déclenchée en cas de débordement ;
- `*` , multiplication, une erreur d'exécution est déclenchée en cas de débordement ;
- `/` , division, une erreur d'exécution est déclenchée si le diviseur est nul ;
- `mod` , calcul du reste, une erreur d'exécution est déclenchée si le diviseur est nul ;
- `&+` , addition, le résultat étant silencieusement tronqué sur 32 bits ;
- `&-` , soustraction, le résultat étant silencieusement tronqué sur 32 bits ;
- `&*` , multiplication, le résultat étant silencieusement tronqué sur 32 bits ;
- `&/` , division, qui retourne zéro si le diviseur est nul.

Ces opérateurs exigent que les deux opérandes soient des objets du même type `@sint` .

28.3.2 Opérateurs préfixés

Le type `@sint` accepte les opérateurs arithmétiques préfixés suivants :

- `+`, qui retourne simplement la valeur de l'opérande ;
- `-`, négation arithmétique, une erreur d'exécution est déclenchée si l'opérande est égal à -2^{31} ;
- `&-`, négation arithmétique, sans détection de débordement : la négation de -2^{31} est -2^{31}.

La valeur renvoyée est du même type `@sint`.

28.3.3 Instructions

Le type `@sint` accepte les instructions arithmétiques suivantes :

- `+=`, addition, une erreur d'exécution est déclenchée en cas de débordement ;
- `-=`, soustraction, une erreur d'exécution est déclenchée en cas de débordement ;
- `*=`, multiplication, une erreur d'exécution est déclenchée en cas de débordement ;
- `/=`, division, une erreur d'exécution est déclenchée en cas division par zéro ;
- `++`, incrémentation, une erreur d'exécution est déclenchée en cas de débordement ;
- `--`, décrémentation, une erreur d'exécution est déclenchée en cas de débordement ;
- `&++`, incrémentation, le résultat étant silencieusement tronqué sur 32 bits ;
- `&--`, décrémentation, le résultat étant silencieusement tronqué sur 32 bits.

`x+=y` est équivalent à `x=x+y` ; `x-=y` est équivalent à `x=x-y`. La variable cible `x`, comme l'expression source `y` doivent être du même type `@sint`.

Incrémentation et décrémentation sont des instructions, et ne peuvent pas apparaître des expressions.

```
@sint n = ... ; n ++ # Incrémentation
```

```
@sint n = ... ; n -- # Décrémentation
```

28.4 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.

28.5 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.

28.6 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 29

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.

29.1 Constructors

29.1.1 Constructeur `min`

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

Returns an `@sint64` object that the minimum value of the 64-bit signed range (-2^{63}).

29.1.2 Constructeur `max`

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

Returns an `@sint64` object that the maximum value of the 64-bit signed range ($2^{63} - 1$).

29.2 Getters

29.2.1 Getter bigint

Ce *getter* permet de convertir un `@sint64` en `@bigint`. Comme la plage des valeurs des `bigint` n'est limitée que par la mémoire disponible, il n'échoue jamais.

```
message [[-1234LS bigint] string] + "\n" # -1234
```

29.2.2 Getter double

```
getter double -> @double
```

Returns the receiver's value converted in a `@double` object. As a 64-bit integer value can always be converted in a `@double` value, this getter never fails.

29.2.3 Getter hexStringSeparatedBy

```
getter hexStringSeparatedBy ?@char inSeparator ?@uint inGroup -> @string
```

Returns the an hexadecimal string representation of the receiver value, prefixed by the string `0x`. Groups of `inGroup` digits are separated by the `inSeparator` character.

If `inGroup` is equal to zero, a run-time error is raised.

For example :

```
let s = [0x123456789ABCDEF0LS hexStringSeparatedBy !'_' !3] # 0x1_234_567_89A_BCD_EF0
```

29.2.4 Getter sint

```
getter sint -> @sint
```

Returns the receiver's value in an `@sint` (page **??**) (32-bit signed integer) object. An error is raised is receiver's value is lower than -2^{31} or greater than $2^{31} - 1$.

This getter is the only way to convert an `@sint64` (page **??**) object into an `@sint` (page **??**) object.

29.2.5 Getter string

```
getter string -> @string
```

Returns a decimal string representation of the receiver's value. This getter never fails.

29.2.6 Getter `uint`

```
getter uint -> @uint
```

Returns the receiver's value in an `@uint` (page **??**) (32-bit unsigned integer) object. An error is raised is receiver's value is negative or greater than $2^{32} - 1$.

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

29.2.7 Getter `uint64`

```
getter uint64 -> @uint64
```

Returns the receiver's value in an `@uint64` (page **??**) (64-bit unsigned integer) object. This getter raises a run-time error if the receiver's value is negative.

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

29.3 Arithmétique

29.3.1 Opérateurs infixés

Le type `@sint64` accepte les opérateurs arithmétiques infixés suivants :

- `+`, addition, une erreur d'exécution est déclenchée en cas de débordement ;
- `-`, soustraction, une erreur d'exécution est déclenchée en cas de débordement ;
- `*`, multiplication, une erreur d'exécution est déclenchée en cas de débordement ;
- `/`, division, une erreur d'exécution est déclenchée si le diviseur est nul ;
- `mod`, calcul du reste, une erreur d'exécution est déclenchée si le diviseur est nul ;
- `&+`, addition, le résultat étant silencieusement tronqué sur 64 bits ;
- `&-`, soustraction, le résultat étant silencieusement tronqué sur 64 bits ;
- `&*`, multiplication, le résultat étant silencieusement tronqué sur 64 bits ;
- `&/`, division, qui retourne zéro si le diviseur est nul.

Ces opérateurs exigent que les deux opérandes soient des objets du même type `@sint64`.

29.3.2 Opérateurs préfixés

Le type `@sint64` accepte les opérateurs arithmétiques préfixés suivants :

- `+`, qui retourne simplement la valeur de l'opérande ;
- `-`, négation arithmétique, une erreur d'exécution est déclenchée si l'opérande est égal à -2^{63} ;
- `&-`, négation arithmétique, sans détection de débordement : la négation de -2^{63} est -2^{63}.

La valeur renvoyée est du même type `@sint64`.

29.3.3 Instructions

Le type `@sint64` accepte les instructions arithmétiques suivantes :

- `+=`, addition, une erreur d'exécution est déclenchée en cas de débordement ;
- `-=`, soustraction, une erreur d'exécution est déclenchée en cas de débordement ;
- `*=`, multiplication, une erreur d'exécution est déclenchée en cas de débordement ;
- `/=`, division, une erreur d'exécution est déclenchée en cas division par zéro ;
- `++`, incrémentation, une erreur d'exécution est déclenchée en cas de débordement ;
- `--`, décrémentation, une erreur d'exécution est déclenchée en cas de débordement ;
- `&++`, incrémentation, le résultat étant silencieusement tronqué sur 64 bits ;
- `&--`, décrémentation, le résultat étant silencieusement tronqué sur 64 bits.

`x+=y` est équivalent à `x=x+y` ; `x-=y` est équivalent à `x=x-y`. La variable cible `x`, comme l'expression source `y` doivent être du même type `@sint64`.

Incrémentation et décrémentation sont des instructions, et ne peuvent pas apparaître des expressions.

```
@sint64 n = ... ; n ++ # Incrémentation
```

```
@sint64 n = ... ; n -- # Décrémentation
```

29.4 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.

29.5 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.

29.6 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 30

Le type `@string`

Le type `@string` définit les chaînes de caractères Unicode.

30.1 Chaînes de caractères littérales

En GALGAS, les chaînes de caractères littérales sont délimitées par des caractères «"», par exemple : `"a string"`. Une chaîne de caractères littérale est un objet constant de type `@string`, si bien que l'on peut lui appliquer méthodes et *getters* : `["ae" uppercaseString]` retourne la chaîne `"AE"`.

30.2 Constructeurs

30.2.1 Constructeur `componentsJoinedByString`

```
constructor componentsJoinedByString
    ?@stringlist inComponents
    ?@string inSeparator -> @string
```

Retourne la chaîne de caractéres obtenue en concaténant tous les éléments de `inComponents` en insérant une copie de `inSeparator` entre deux éléments consécutifs.

```
let aList = @stringlist {!"A", !"B", !"C"}
let s = @string.componentsJoinedByString {!aList !"-"} # "A-B-C"
```

30.2.2 Constructeur CppChar

```
constructor CppChar ?@char inChar -> @string
```

Retourne la chaîne de caractéres constitué du caractère `inChar` précédé et suivi par un caractère « " ».

```
let s = @string.CppChar {!'A'} # "A"
```

30.2.3 Constructeur CppLineComment

```
constructor CppLineComment -> @string
```

Retourne une chaîne de caractères constitué de :

- deux caractères « / » ;
- suivi de 117 caractères « - » ;
- suivi d'un caractère « * » ;
- et terminée par un retour à la ligne.

30.2.4 Constructeur CppTitleComment

```
constructor CppTitleComment ?@string inString -> @string
```

Retourne une chaîne de caractères constitué de cinq lignes de commentaires C++ :

- une ligne obtenue par appel du *constructeur `CppLineComment` du type `@string` – page* **??** ;
- une ligne obtenue par appel du *constructeur `CppSpaceComment` du type `@string` – page* **??** ;
- une ligne de commentaire contenant `inString` centré ;
- une ligne obtenue par appel du *constructeur `CppSpaceComment` du type `@string` – page* **??** ;
- une ligne obtenue par appel du *constructeur `CppLineComment` du type `@string` – page* **??**.

30.2.5 Constructeur CppSpaceComment

```
constructor CppSpaceComment -> @string
```

Retourne une chaîne de caractères constitué de :

- deux caractères « / » ;

- suivi de 117 caractères *espace* ;
- suivi d'un caractère « * » ;
- et terminée par un retour à la ligne.

Ce constructeur permet d'écrire des commentaires encadrés dans le code C++ engendré.

30.2.6 Constructeur `default`

```
constructor default -> @string
```

Retourne la chaîne vide (voir section **??** page **??**).

30.2.7 Constructeur `homeDirectory`

```
constructor homeDirectory -> @string
```

Retourne une chaîne de caractères contenant le chemin absolu vers le répertoire *home* de l'utilisateur. Fonctionne sous Unix et Windows.

30.2.8 Constructeur `newWithStdIn`

```
constructor newWithStdIn -> @string
```

Bloque l'exécution en attente de saisie d'une ligne sur le terminal. La saisie du retour-chariot relance l'exécution. La chaîne saisie (y compris le retour-chariot qui la termine) est renvoyée par le constructeur.

30.2.9 Constructeur `retrieveAndResetTemplateString`

```
constructor retrieveAndResetTemplateString -> @string
```

Ce constructeur est utilisé pour la génération de *templates*.

30.2.10 Constructeur `separatorString`

```
constructor separatorString -> @string
```

Ce constructeur renvoie la chaîne de caractères formant le séparateur entre le token courant et le suivant (y compris les commentaires). Son utilisation principale est de permettre d'implémenter un mécanisme permettant de vérifier qu'une instruction se termine par une fin de ligne, ou un `;`.

Par exemple, considérons l'analyse d'une liste d'instructions :

```
repeat
while
  <instruction>
end
```

Dans le code ci-dessus, rien n'oblige à séparer les instructions par une fin de ligne. On impose l'occurrence d'une fin de ligne (ou plusieurs) ou d'un `;` en écrivant :

```
repeat
while
  <instruction>
  select
   $;$
  or
    let s = @string.separatorString
    if not [s containsCharacter !'\n'] then
      error .here
       : "instruction should be terminated by an end of line or a ';'"
    end
  end
end
```

30.2.11 Constructeur `stringByRepeatingString`

```
constructor stringByRepeatingString
  ?@string inString
  ?@uint inCount
  -> @string
```

Ce constructeur retourne la chaîne de caractères constituée d'une séquence de `inCount` chaînes `inString`.

30.2.12 Constructeur `stringWithContentsOfFile`

```
constructor stringWithContentsOfFile ?@string inFilePath -> @string
```

Ce constructeur lit le fichier texte désigné par le chemin relatif ou absolu `inFilePath` et retourne sont contenu. Une erreur d'exécution est déclenché si le fichier ne peut pas être lu.

30.2.13 Constructeur `stringWithCurrentDateTime`

```
constructor stringWithCurrentDateTime -> @string
```

Ce constructeur retourne une chaîne de caractères contenant la date et l'heure courante.

Par exemple :

```
let s = @string.stringWithCurrentDateTime # "Wed Jan  6 20:08:33 2016"
```

30.2.14 Constructeur `stringWithCurrentDirectory`

```
constructor stringWithCurrentDirectory -> @string
```

Ce constructeur retourne une chaîne de caractères contenant le chemin absolu du répertoire courant.

30.2.15 Constructeur `stringWithEnvironmentVariable`

```
constructor stringWithEnvironmentVariable
  ?@string inVariableName
  -> @string
```

Ce constructeur retourne la valeur associée à la variable d'environnement `inVariableName`. Une erreur d'exécution est déclenchée si la variable d'environnement n'est pas définie. L'existence d'une variable d'environnement peut être testée par le *getter* `doesEnvironmentVariableExist` *du type* `@string` *(page* **??***)* .

30.2.16 Constructeur `stringWithEnvironmentVariableOrEmpty`

```
constructor stringWithEnvironmentVariableOrEmpty
  ?@string inVariableName
  -> @string
```

Ce constructeur retourne la valeur associée à la variable d'environnement `inVariableName`. Si la variable d'environnement n'est pas définie, la chaîne vide est retournée et aucune erreur n'est déclenchée.

30.2.17 Constructeur `stringWithSequenceOfCharacters`

```
constructor stringWithSequenceOfCharacters
  ?@char inChar
  ?@uint inCount
  -> @string
```

Ce constructeur retourne la chaîne de caractères constituée d'une séquence de `inCount` caractères `inChar`. Pour répéter une chaîne de caractères, voir le *constructeur* `stringByRepeatingString` *du type* `@string` –

page **??**.

30.2.18 Constructeur stringWithSourceFilePath

```
constructor stringWithSourceFilePath -> @string
```

Ce constructeur retourne le chemin absolu du fichier source en cours d'analyse.

30.2.19 Constructeur stringWithSymbolicLinkContents

```
constructor stringWithSymbolicLinkContents ?@string inPath -> @string
```

30.3 Getters

30.3.1 Getter absolutePathFromPath

```
getter @string absolutePathFromPath ?@string inPath -> @string
```

Si la valeur du récepteur est un chemin absolu, cette valeur est retournée et `inPath` est inutilisé.

Si la valeur du récepteur est un chemin relatif, cette valeur est retournée préfixée par `inPath` .

30.3.2 Getter assemblerRepresentation

```
getter @string assemblerRepresentation -> @string
```

Retourne la valeur du récepteur sous la forme d'une chaîne de caractères construite en traduisant chaque caractère :

- une lettre ASCII (minuscule ou majuscule) est inchangée ;
- un chiffre décimal est inchangé ;
- les caractères « . », « - » et « $ » sont inchangés ;
- tout autre caractère est traduit en son point de code en hexadécimal, précédé et suivi par un caractère « _ ».

La chaîne obtenue est un identificateur C ou C++ valide si le récepteur commence par une lettre ASCII ou un caractère de soulignement « _ ».

Par exemple :

```
let x = ["$Z2.3" assemblerRepresentation] # "$Z2.3"
let y = [":?" assemblerRepresentation] # "_3A__3F_"
let y = ["_é" assemblerRepresentation] # "_5F__E8_"
```

Voir aussi le *getter* `identifierRepresentation` *du type* `@string` *(page* **??***)* qui retourne toujours un identificateur C ou C++ valide et le *getter* `nameRepresentation` *du type* `@string` *(page* **??***)* .

Pour reconstituer la chaîne d'origine, appeler le *getter* `decodedStringFromRepresentation` *du type* `@string` *(page* **??***)* .

30.3.3 Getter `capacity`

```
getter @string capacity -> @uint
```

Retourne le nombre de caractères alloués pour stocker la valeur du récepteur.

30.3.4 Getter `characterAtIndex`

```
getter @string characterAtIndex ?@uint inIndex -> @char
```

Retourne le caractère situé à l'indice `inIndex` de la valeur du récepteur. Le premier caractère a pour indice 0. Si `inIndex` est supérieur au égal à la longueur de la valeur du récepteur, une erreur d'exécution est déclenchée.

30.3.5 Getter `commandWithArguments`

```
getter @string commandWithArguments ?@stringlist inArguments -> @sint
```

Exécute la commande *shell* dont le nom est la valeur du récepteur, avec les arguments désignés par la valeur de `inArguments`. La sortie de la commande est affichée sur la console. Quand la commande est terminée, sa valeur de sortie est retournée.

Contrairement au *getter* `system` *du type* `@string` *(page* **??***)* , des espaces sont acceptés dans le nom de la commande et dans les arguments.

```
let r = ["cp" commandWithArguments !{!"fichierA.txt", !"fichierB.txt"}]
if r == 0S then
  # Ok, pas d'erreur
else
```

```
  # Erreur
end
```

30.3.6 Getter componentsSeparatedByString

```
getter @string componentsSeparatedByString ?@string inSeparator -> @stringlist
```

Retourne une liste des sous-chaînes de la valeur du récepteur qui a été divisée par `inSeparator`.

```
let b = ["a--b--c--" componentsSeparatedByString !"--"]
# "a", "b", "c", ""
```

30.3.7 Getter containsCharacter

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

Retourne `true` si le récepteur contient le caractère `inCharacter`, et `false` dans le cas contraire.

```
let b = ["abcdef" containsCharacter !'c'] # true
```

30.3.8 Getter containsCharacterInRange

```
getter @string containsCharacterInRange
  ?@char inFirstCharacter
  ?@char inLastCharacter
  -> @bool
```

Retourne `true` si le récepteur contient un ou plusieurs caractère dont le point de code est supérieur ou égal à celui de `inLastCharacter` et inférieur ou égal à celui de `inFirstCharacter`, et `false` dans le cas contraire. En conséquence, si le point de code Unicode de `inFirstCharacter` doit être strictement supérieur au point de de code de `inLastCharacter`, la valeur renvoyée est toujours `false`.

```
let b = ["abcdef" containsCharacterInRange !'c' !'d'] # true
let c = ["abcdef" containsCharacterInRange !'x' !'z'] # false
```

30.3.9 Getter count

```
getter @string count -> @uint
```

Retourne le nombre de caractères UTF-32 du récepteur. Si le récepteur n'est pas une chaîne ASCII, ce getter ne donne pas le nombre d'octets de sa représentation en UTF-8 : pour cela utiliser le *getter* `utf8Length` *du type* `@string` *(page* **??***)*.

30.3.10 Getter `cStringRepresentation`

```
getter @string cStringRepresentation -> @string
```

Retourne la valeur du récepteur sous la forme d'une chaîne UTF-8, c'est-à-dire qu'un caractère « " » a été inséré au début et à la fin.

Le caractère « " » est échappé en « \" », le caractère « \ » est échappé en « \\ ». Le caractère de code ASCII `0x0D` (« *Carriage Return* ») est écrit sous la forme d'un anti slash suivi du caractère `"n"`.

30.3.11 Getter `currentColumn`

```
getter @string currentColumn -> @uint
```

Retourne l'indice de la colonne, c'est-à-dire :

- si le récepteur ne contient pas de retour à la ligne, le nombre de caractères du récepteur ;
- si le récepteur contient des retours à la ligne, le nombre de caractères du récepteur qui suivent la dernière occurrence d'un retour à la ligne.

30.3.12 Getter `decimalSignedBigInt`

```
getter @string decimalSignedBigInt -> @bigint
```

Retourne la valeur du récepteur convertie en entier signé. La valeur du récepteur doit donc ne contenir que des chiffres décimaux, éventuellement précédés par un « + » ou un « - ». Si ce n'est pas le cas, une erreur d'exécution est déclenchée.

La valeur du récepteur peut être testée en appelant le *getter* `isDecimalSignedBigInt` *du type* `@string` *(page* **??***)* .

30.3.13 Getter `decimalSignedNumber`

```
getter @string decimalSignedNumber -> @sint
```

Retourne la valeur du récepteur convertie en entier signé 32 bits. La valeur du récepteur doit donc ne contenir que des chiffres décimaux, éventuellement précédés par un « + » ou un « - ». Si ce n'est pas le cas, une erreur d'exécution est déclenchée.

La valeur du récepteur peut être testée en appelant le *getter* `isDecimalSignedNumber` *du type* `@string` *(page* **??***)* .

30.3.14 Getter decimalSigned64Number

```
getter @string decimalSigned64Number -> @sint64
```

Retourne la valeur du récepteur convertie en entier signé 64 bits. La valeur du récepteur doit donc ne contenir que des chiffres décimaux, éventuellement précédés par un « +» ou un « -». Si ce n'est pas le cas, une erreur d'exécution est déclenchée.

La valeur du récepteur peut être testée en appelant le *getter* `isDecimalSigned64Number` *du type* `@string` *(page* **??***)* .

30.3.15 Getter decimalUnsignedNumber

```
getter @string decimalUnsignedNumber -> @uint
```

Retourne la valeur du récepteur convertie en entier non signé 32 bits. La valeur du récepteur doit donc ne contenir que des chiffres décimaux. Si ce n'est pas le cas, une erreur d'exécution est déclenchée.

La valeur du récepteur peut être testée en appelant le *getter* `isDecimalUnsignedNumber` *du type* `@string` *(page* **??***)* .

30.3.16 Getter decimalUnsigned64Number

```
getter @string decimalUnsigned64Number -> @uint64
```

Retourne la valeur du récepteur convertie en entier non signé 64 bits. La valeur du récepteur doit donc ne contenir que des chiffres décimaux. Si ce n'est pas le cas, une erreur d'exécution est déclenchée.

La valeur du récepteur peut être testée en appelant le *getter* `isDecimalUnsigned64Number` *du type* `@string` *(page* **??***)* .

30.3.17 Getter decodedStringFromRepresentation

```
getter @string decodedStringFromRepresentation -> @string
```

Ce getter suppose que le récepteur est une chaîne de caractères résultat de l'appel du *getter* `assemblerRepresentation` *du type* `@string` *(page* **??***)* , du *getter* `identifierRepresentation` *du type* `@string` *(page* **??***)* ou du *getter* `nameRepresentation` *du type* `@string` *(page* **??***)* , et retourne la chaîne d'origine.

Par exemple :

```
let s = ["chaîne accentuée" identifierRepresentation]
log s # LOGGING s: <@string:"cha_EE_ne_20_accentu_E9_e">
let y = [s decodedStringFromRepresentation]
```

```
log y #LOGGING y: <@string:"chaîne accentuée">
```

Une erreur est déclenchée à l'exécution si le réception n'est pas une chaîne valide, et la valeur retournée n'est pas construite.

30.3.18 Getter `directories`

```
getter @string directories ?@bool inRecursiveSearch -> @stringlist
```

Retourne la liste des sous-répertoires du répertoire désigné par la valeur du récepteur. Si le paramètre `inRecursiveSearch` est vrai, une recherche récursive dans les sous répertoires est effectuée.

30.3.19 Getter `directoriesWithExtensions`

```
getter @string directoriesWithExtensions
    ?@bool inRecursiveSearch
    ?@stringlist inExtensionList -> @stringlist
```

Retourne la liste des sous-répertoires du répertoire désigné par la valeur du récepteur, en ne retenant que les répertoires dont l'extension appartient à liste `inExtensionList`. Si le paramètre `inRecursiveSearch` est vrai, une recherche récursive dans les sous répertoires est effectuée.

30.3.20 Getter `directoryExists`

```
getter @string directoryExists -> @bool
```

Retourne `true` si la valeur du récepteur désigne un répertoire existant, et `false` dans le cas contraire.

30.3.21 Getter `doesEnvironmentVariableExist`

```
getter @string doesEnvironmentVariableExist -> @bool
```

Retourne `true` si la valeur du récepteur nomme une variable d'environnement existante, et `false` dans le cas contraire.

30.3.22 Getter `doubleNumber`

```
getter @string doubleNumber -> @double
```

Retourne la valeur du récepteur convertie en entier non signé. La valeur du récepteur doit donc ne contenir que des chiffres décimaux. Si ce n'est pas le cas, une erreur d'exécution est déclenchée.

La valeur du récepteur peut être testée en appelant le *getter* `isDoubleNumber` *du type* `@string` *(page* **??***)* .

30.3.23 Getter `fileExists`

```
getter @string fileExists -> @bool
```

Retourne `true` si la valeur du récepteur désigne un fichier existant, et `false` dans le cas contraire.

30.3.24 Getter `fileNameRepresentation`

```
getter @string fileNameRepresentation -> @string
```

Retourne la valeur du récepteur sous la forme d'une chaîne de caractères qui sera toujours un nom de fichier valide :

- une lettre ASCII minuscule est inchangée ;
- un chiffre décimal est inchangé ;
- tout autre caractère est traduit en son point de code en hexadécimal, précédé et suivi par un caractère « - ».

Par exemple :

```
let x = ["Z23" fileNameRepresentation] # "-5A-23"
let y = [":?" fileNameRepresentation] # "-3A--3F-"
```

En particulier, les lettres majuscules sont remplacées ; c'est indispensable pour les systèmes de fichiers qui sont insensibles à la casse, cela permet d'obtenir des noms de fichiers différents à partir de noms ne différant que par la casse :

```
let x = ["exemple" fileNameRepresentation] # "exemple"
let y = ["Exemple" fileNameRepresentation] # "-45-xemple"
```

30.3.25 Getter `firstCharacterOrNul`

```
getter @string firstCharacterOrNul -> @char
```

Si la longueur de la valeur du récepteur est non nulle, retourne son premier caractère, sinon le caractère NUL.

30.3.26 Getter `here`

Caractère	Codage en HTML
&	&amp;
"	&quot;
<	&lt;
>	&gt;

Tableau 30.1 – *Codage des caractères, getter* `HTMLRepresentation` *du type* `@string`

```
getter @string here -> @lstring
```

Retourne un `@lstring` dont le champ `string` est la valeur du récepteur, et dont le champ `location` désigne la position courante de l'analyse. L'expression `[s here]` est équivalente à `@lstring.new{!s !.here}`.

30.3.27 Getter `hiddenCommandWithArguments`

```
getter @string hiddenCommandWithArguments ?@stringlist inArguments -> @string
```

Exécute la commande *shell* dont le nom est la valeur du récepteur, avec les arguments désignés par la valeur de `inArguments`. Quand la commande est terminée, la sortie de la commande est retournée.

Contrairement au *getter* popen *du type* `@string` *(page* **??***)* , des espaces sont acceptés dans le nom de la commande et dans les arguments.

30.3.28 Getter `hiddenFiles`

```
getter @string hiddenFiles ?@bool inRecursiveSearch -> @stringlist
```

Retourne la liste des fichiers cachés du répertoire désigné par la valeur du récepteur. Si le paramètre `inRecursiveSearch` est vrai, une recherche récursive dans les sous répertoires est effectuée.

30.3.29 Getter `HTMLRepresentation`

```
getter @string HTMLRepresentation -> @string
```

Retourne la valeur du récepteur encodée pour former une chaîne HTML valide. Les caractères « & », « " », « < » et « > » sont modifiés selon le tableau **??** page **??**.

30.3.30 Getter `identifierRepresentation`

```
getter @string identifierRepresentation -> @string
```

Retourne la valeur du récepteur sous la forme d'une chaîne de caractères qui sera toujours un identificateur C ou C++ valide :

- une lettre ASCII (minuscule ou majuscule) est inchangée ;
- tout autre caractère est traduit en son point de code en hexadécimal, précédé et suivi par un caractère « _ ».

Par exemple :

```
let x = ["Z23" identifierRepresentation] # "Z_32__33_"
let y = [":?" identifierRepresentation] # "_3A__3F_"
```

Voir aussi le *getter* `nameRepresentation` *du type* `@string` *(page* **??***)* qui laisse inchangé un chiffre décimal, et le *getter* `assemblerRepresentation` *du type* `@string` *(page* **??***)* .

Pour reconstituer la chaîne d'origine, appeler le *getter* `decodedStringFromRepresentation` *du type* `@string` *(page* **??***)* .

30.3.31 Getter `isDecimalSignedBigInt`

```
getter @string isDecimalSignedBigInt -> @bool
```

Ce *getter* permet de savoir si le récepteur représente un entier signé, c'est-à-dire si le *getter* `decimalSignedBigInt` *du type* `@string` *(page* **??***)* peut être appelé sans qu'une erreur d'exécution se déclenche.

30.3.32 Getter `isDecimalSignedNumber`

```
getter @string isDecimalSignedNumber -> @bool
```

Ce *getter* permet de savoir si le récepteur représente un entier signé sur 32 bits, c'est-à-dire si le *getter* `decimalSignedNumber` *du type* `@string` *(page* **??***)* peut être appelé sans qu'une erreur d'exécution se déclenche.

30.3.33 Getter `isDecimalSigned64Number`

```
getter @string isDecimalSigned64Number -> @bool
```

Ce *getter* permet de savoir si le récepteur représente un entier signé sur 64 bits, c'est-à-dire si le *getter* `decimalSigned64Number` *du type* `@string` *(page* **??***)* peut être appelé sans qu'une erreur d'exécution se déclenche.

30.3.34 Getter `isDecimalUnsignedNumber`

```
getter @string isDecimalUnsignedNumber -> @bool
```

Ce *getter* permet de savoir si le récepteur représente un entier non signé sur 32 bits, c'est-à-dire si le *getter* `decimalUnsignedNumber` *du type* `@string` *(page* **??***)* peut être appelé sans qu'une erreur d'exécution se déclenche.

30.3.35 Getter `isDecimalUnsigned64Number`

```
getter @string isDecimalUnsigned64Number -> @bool
```

Ce *getter* permet de savoir si le récepteur représente un entier non signé sur 64 bits, c'est-à-dire si le *getter* `decimalUnsigned64Number` *du type* `@string` *(page* **??***)* peut être appelé sans qu'une erreur d'exécution se déclenche.

30.3.36 Getter `isDoubleNumber`

```
getter @string isDoubleNumber -> @bool
```

Ce *getter* permet de savoir si le récepteur représente un nombre flottant, c'est-à-dire si le *getter* `doubleNumber` *du type* `@string` *(page* **??***)* peut être appelé sans qu'une erreur d'exécution se déclenche.

30.3.37 Getter `isSymbolicLink`

```
getter @string isSymbolicLink -> @bool
```

Retourne `true` si la valeur du récepteur désigne un lien symbolique existant, et `false` dans le cas contraire.

30.3.38 Getter `lastCharacter`

```
getter @string lastCharacter -> @char
```

Retourne le dernier caractère du récepteur. Si celui-ci est vide, une erreur d'exécution est déclenchée.

30.3.39 Getter `lastPathComponent`

```
getter @string lastPathComponent -> @string
```

Retourne le récepteur ne contient pas de caractère « / », il est retourné inchangé. Sinon, ce *getter* retourne la sous-chaîne de caractères qui suit la dernière occurrence du caractère « / ».

30.3.40 Getter `leftSubString`

```
getter @string leftSubString ?@uint inLength -> @string
```

Retourne la sous-chaîne constituée des `inLength` derniers caractères du récepteur. Si celui-ci contient moins de `inLength` caractères, une erreur d'exécution est déclenchée.

30.3.41 Getter `length`

```
getter @string length -> @uint # Obsolete, utiliser count
```

Retourne le nombre de caractères UTF-32 du récepteur. Si le récepteur n'est pas une chaîne ASCII, ce getter ne donne pas le nombre d'octets de sa représentation en UTF-8 : pour cela utiliser le *getter* `utf8Length` *du type* `@string` *(page* **??***)* .

30.3.42 Getter `lowercaseString`

```
getter @string lowercaseString -> @string
```

Retourne la valeur du récepteur, dans laquelle toutes les lettres majuscules sont transformées en minuscules.

Par exemple :

```
let x = ["AbcD" lowercaseString] # "abcd"
let y = ["ÊÆ" lowercaseString] # "êæ"
```

30.3.43 Getter `md5`

```
getter @string md5 -> @string
```

Retourne la somme de contrôle MD5 du récepteur sous la forme d'une chaîne de 32 caractères hexadécimaux.

Par exemple :

```
let x = ["Hello" md5] # "8B1A9953C4611296A827ABF8C47804D7"
```

30.3.44 Getter `nameRepresentation`

```
getter @string nameRepresentation -> @string
```

Retourne la valeur du récepteur sous la forme d'une chaîne de caractères construite en traduisant chaque caractère :

- une lettre ASCII (minuscule ou majuscule) est inchangée ;
- un chiffre décimal est inchangé ;
- tout autre caractère est traduit en son point de code en hexadécimal, précédé et suivi par un caractère « _ ».

La chaîne obtenue est un identificateur C ou C++ valide si le récepteur commence par une lettre ASCII ou un caractère de soulignement « _ ».

Par exemple :

```
let x = ["Z23" nameRepresentation] # "Z23"
let y = [":?" nameRepresentation] # "_3A__3F_"
let y = ["_é" nameRepresentation] # "_5F__E8_"
```

Voir aussi le *getter* `identifierRepresentation` *du type* `@string` *(page* **??***)* qui retourne toujours un identificateur C ou C++ valide, et le *getter* `assemblerRepresentation` *du type* `@string` *(page* **??***)* .

Pour reconstituer la chaîne d'origine, appeler le *getter* `decodedStringFromRepresentation` *du type* `@string` *(page* **??***)* .

30.3.45 Getter `nativePathWithUnixPath`

```
getter @string nativePathWithUnixPath -> @string
```

Sous Unix, ce *getter* retourne la valeur du récepteur. Sous Windows, il retourne la valeur du récepteur encodé *à la Windows*.

Par exemple, sous Windows :

```
let x = ["/C/Program Files/f" nativePathWithUnixPath] # "C:\Program Files\f"
```

30.3.46 Getter `nowhere`

```
getter @string nowhere -> @lstring
```

Retourne un `@lstring` dont le champ `string` est la valeur du réception, et dont le champ `location` est vide.

30.3.47 Getter pathExtension

```
getter @string pathExtension -> @string
```

Si le récepteur ne contient pas de caractère « . », la chaîne vide est retournée. Sinon, ce *getter* retourne la sous-chaîne de caractères qui suit la dernière occurrence du caractère « . ».

30.3.48 Getter popen

```
getter @string popen -> @string
```

Ce *getter* exécute la commande Shell exprimée par la valeur du récepteur. La sortie de cette commande est accumulée et retournée par ce *getter* lorsque la commande est terminée.

30.3.49 Getter range

```
getter @string range -> @range
```

Retourne un objet de type @range (page **??**) dont le champ `start` est 0 et le champ `length` est égal au nombre de caractères du récepteur.

30.3.50 Getter regularFiles

```
getter @string regularFiles ?@bool inRecursiveSearch -> @stringlist
```

Retourne la liste des fichiers non cachés du répertoire désigné par la valeur du récepteur. Si le paramètre `inRecursiveSearch` est vrai, une recherche récursive dans les sous répertoires est effectuée.

30.3.51 Getter regularFilesWithExtensions

```
getter @string regularFilesWithExtensions
    ?@bool inRecursiveSearch
    ?@stringlist inExtensionList -> @stringlist
```

Retourne la liste des fichiers non cachés du répertoire désigné par la valeur du récepteur, en ne retenant que les fichiers dont l'extension est nommée dans `inExtensionList`. Si le paramètre `inRecursiveSearch` est vrai, une recherche récursive dans les sous répertoires est effectuée.

30.3.52 Getter relativePathFromPath

```
getter @string relativePathFromPath ?@string inPath -> @string
```

Retourne le chemin relatif du récepteur à partir du chemin `inPath`.

30.3.53 Getter `reversedString`

```
getter @string reversedString -> @string
```

Retourne la valeur renversée du récepteur.

```
let x = ["abcde" reversedString] # "edcba"
```

30.3.54 Getter `rightSubString`

```
getter @string rightSubString ?@uint inLength -> @string
```

Retourne la sous-chaîne constituée des `inLength` premiers caractères du récepteur. Si celui-ci contient moins de `inLength` caractères, une erreur d'exécution est déclenchée.

30.3.55 Getter `stringByCapitalizingFirstCharacter`

```
getter @string stringByCapitalizingFirstCharacter -> @string
```

Retourne la valeur du récepteur dans laquelle le premier caractère, si il est une lettre, a été mis en majuscule.

30.3.56 Getter `stringByDeletingLastPathComponent`

```
getter @string stringByDeletingLastPathComponent -> @string
```

Si le récepteur ne contient pas de caractère « / », la chaîne vide est retournée. Sinon, ce *getter* retourne la sous-chaîne de caractères qui précède la dernière occurrence du caractère « / ».

30.3.57 Getter `stringByDeletingPathExtension`

```
getter @string stringByDeletingPathExtension -> @string
```

Si le récepteur ne contient pas de caractère « . », la valeur du récepteur est retournée. Sinon, ce *getter* retourne la sous-chaîne de caractères qui précède la dernière occurrence du caractère « . ».

30.3.58 Getter stringByLeftAndRightPadding

```
getter @string stringByLeftAndRightPadding
    ?@uint inPaddedStringLength
    ?@char inPaddingChar -> @string
```

Si la longueur de la valeur du récepteur est supérieure ou égal à `inPaddedStringLength`, la valeur du récepteur est retournée. Sinon, ce *getter* retourne la valeur du récepteur précédée et suivie d'un nombre égal de caractères `inPaddingChar`, de façon à atteindre la longueur `inPaddedStringLength`. Si le nombre de caractères à ajouter est impair, le nombre de caractères ajoutés est supérieur d'une unité au nombre de caractères insérés au début.

30.3.59 Getter stringByLeftPadding

```
getter @string stringByLeftPadding
    ?@uint inPaddedStringLength
    ?@char inPaddingChar -> @string
```

Si la longueur de la valeur du récepteur est supérieure ou égal à `inPaddedStringLength`, la valeur du récepteur est retournée. Sinon, ce *getter* retourne la valeur du récepteur suivie d'un nombre de caractères `inPaddingChar`, de façon à atteindre la longueur `inPaddedStringLength`.

30.3.60 Getter stringByRemovingCharacterAtIndex

```
getter @string stringByRemovingCharacterAtIndex ?@uint inIndex -> @string
```

Retourne la valeur du récepteur, amputée du caractère situé à l'indice `inIndex`. Une erreur d'exécution est déclenchée si `inIndex` est supérieur ou égal au nombre de caractères du récepteur.

30.3.61 Getter stringByReplacingStringByString

```
getter @string stringByReplacingStringByString
      ?@string inSearchedString
      ?@string inReplacementString  -> @string
```

Retourne la valeur du récepteur, dans laquelle chaque occurrence de `inSearchedString` est remplacée par `inReplacementString`.

30.3.62 Getter stringByRightPadding

```
getter @string stringByRightPadding
    ?@uint inPaddedStringLength
    ?@char inPaddingChar -> @string
```

Si la longueur de la valeur du récepteur est supérieure ou égal à `inPaddedStringLength`, la valeur du récepteur est retournée. Sinon, ce *getter* retourne la valeur du récepteur précédée d'un nombre de caractères `inPaddingChar`, de façon à atteindre la longueur `inPaddedStringLength`.

30.3.63 Getter `stringByStandardizingPath`

```
getter @string stringByStandardizingPath -> @string
```

Retourne la valeur du récepteur, dans laquelle les séquences « `./` » et « `../` » sont supprimées.

30.3.64 Getter `stringByTrimmingWhiteSpaces`

```
getter @string stringByTrimmingWhiteSpaces -> @string
```

Retourne la valeur du récepteur dans laquelles les espaces en tête et en fin ont été supprimés.

30.3.65 Getter `subString`

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

Retourne la sous-chaîne de `inLength` caractères extraite de la valeur du récepteur à partir de l'indice `inStart`.

30.3.66 Getter `subStringFromIndex`

```
getter @string subStringFromIndex ?@uint inIndex -> @string
```

Retourne la sous-chaîne de caractères extraite de la valeur du récepteur à partir de l'indice `inIndex`. Si `inIndex` est supérieur ou égal au nombre de caractères du récepteur, la chaîne vide est retournée.

30.3.67 Getter `system`

```
getter @string system -> @sint
```

Exécute la commande *shell* avec la valeur du récepteur en argument. La valeur retournée par cet appel est la valeur retournée par ce *getter*. On peut exécuter plusieurs commandes séquentiellement en les séparant

par un point-virgule. La sortie de la commande est affichée sur la console.

Contrairement au *getter* `commandWithArguments` *du type* `@string` *(page* **??***)* , les espaces dans les arguments doivent être explicitement échappés.

30.3.68 Getter `unixPathWithNativePath`

```
getter @string unixPathWithNativePath -> @string
```

La valeur du récepteur est un chemin valide pour la plateforme courante.

Sous Windows, ce *getter* retourne la valeur du récepteur sous la forme d'un chemin Unix. Sous Unix, la valeur du récepteur est renvoyée.

30.3.69 Getter `uppercaseString`

```
getter @string uppercaseString -> @string
```

Retourne la valeur du récepteur, dans laquelle toutes les lettres minuscules sont transformées en majuscules.

Par exemple :

```
let x = ["AbcD" uppercaseString] # "ABCD"
let y = ["êæ" uppercaseString] # "ÊÆ"
```

30.3.70 Getter `utf32Representation`

```
getter @string utf32Representation -> @string
```

30.3.71 Getter `utf8Length`

```
getter @string utf8Length -> @uint
```

Retourne le nombre d'octets de la représentation UTF-8 de la valeur du récepteur. Si le récepteur n'est pas une chaîne ASCII, ce nombre diffère de la valeur retournée par le *getter* `length` *du type* `@string` *(page* **??***)* :

```
var s1 = "Toto"
var nUTF32 = [s length] # 4
var nUTF8  = [s utf8Length] # 4
s1 = "Tâche"
nUTF32 = [s length] # 5
```

```
  nUTF8   = [s utf8Length] # 6
```

30.3.72 Getter `utf8Representation`

```
getter @string utf8Representation -> @string
```

Retourne la valeur du récepteur sous la forme d'une chaîne UTF-8, c'est-à-dire qu'un caractère « " » a été inséré au début et à la fin.

Les caractères non-ASCII sont échappés.

Le caractère « " » est échappé en « \" », le caractère « \ » est échappé en « \\ ».

30.3.73 Getter `utf8RepresentationEscapingQuestionMark`

```
getter @string utf8RepresentationEscapingQuestionMark -> @string
```

Retourne la valeur du récepteur sous la forme d'une chaîne UTF-8, c'est-à-dire qu'un caractère « " » a été inséré au début et à la fin.

Les caractères non-ASCII sont échappés. De plus, le caractère « " » est échappé en « \" », le caractère « \ » est échappé en « \\ », le caractère « ? » est échappé en « \? ».

30.3.74 Getter `utf8RepresentationEnclosedWithin`

```
getter @string utf8RepresentationEnclosedWithin ?@char inCharacter -> @string
```

Retourne la valeur du récepteur sous la forme d'une chaîne UTF-8, entourée par le caractère indiqué en argument.

Le caractère indiqué en argument est échappé en le préfixant par« \ ». Le caractère « \ » est échappé en « \\ ».

30.3.75 Getter `utf8RepresentationWithoutDelimiters`

```
getter @string utf8RepresentationWithoutDelimiters -> @string
```

Retourne la valeur du récepteur sous la forme d'une chaîne UTF-8 sans le délimiteur initial et ni le délimiteur terminal, c'est-à-dire qu'aucun caractère « " » n'est inséré au début et à la fin.

30.4 Méthodes

30.4.1 Méthode makeDirectory

```
method @string makeDirectory
```

Crée le répertoire désigné par la valeur du récepteur. Les répertoires intermédiaires sont créés si besoin. Si le répertoire existe déjà, aucune erreur n'est déclenchée.

30.4.2 Méthode makeDirectoryAndWriteToExecutableFile

```
method @string makeDirectoryAndWriteToExecutableFile ?@string inFilePath
```

Le récepteur contient la chaîne de caractères qui est écrit dans le fichier `inFilePath`. Les répertoires intermédiaires sont créés si besoin. Le fichier `inFilePath` est rendu exécutable.

30.4.3 Méthode makeDirectoryAndWriteToFile

```
method @string makeDirectoryAndWriteToFile ?@string inFilePath
```

Le récepteur contient la chaîne de caractères qui est écrit dans le fichier `inFilePath`. Les répertoires intermédiaires sont créés si besoin.

30.4.4 Méthode makeSymbolicLinkWithPath

```
method @string makeSymbolicLinkWithPath ?@string inFilePath
```

30.4.5 Méthode writeToExecutableFile

```
method @string writeToExecutableFile ?@string inFilePath
```

Le récepteur contient la chaîne de caractères qui est écrit dans le fichier `inFilePath`. Aucun répertoire intermédiaire n'est créé, c'est une erreur d'exécution si ils n'existent pas. Si il a été écrit avec succès, le fichier `inFilePath` est rendu exécutable.

30.4.6 Méthode writeToExecutableFileWhenDifferentContents

```
method @string writeToExecutableFileWhenDifferentContents
    ?@string inFilePath
    !@bool outFileModified
```

Le récepteur contient la chaîne de caractères qui est écrit dans le fichier `inFilePath`. Aucun répertoire intermédiaire n'est créé, c'est une erreur d'exécution si ils n'existent pas. Si il a été écrit avec succès, le fichier `inFilePath` est rendu exécutable. Le booléen `outFileModified` permet de savoir si le contenu original du fichier a été modifié.

30.4.7 Méthode `writeToFile`

```
method @string writeToFile ?@string inFilePath
```

Le récepteur contient la chaîne de caractères qui est écrit dans le fichier `inFilePath`. Aucun répertoire intermédiaire n'est créé, c'est une erreur d'exécution si ils n'existent pas.

30.4.8 Méthode `writeToFileWhenDifferentContents`

```
method @string @string writeToFileWhenDifferentContents
    ?@string inFilePath
    !@bool outFileModified
```

Le récepteur contient la chaîne de caractères qui est écrit dans le fichier `inFilePath`. Aucun répertoire intermédiaire n'est créé, c'est une erreur d'exécution si ils n'existent pas.Le booléen `outFileModified` permet de savoir si le contenu original du fichier a été modifié.

30.5 Setters

30.5.1 Setter `appendSpacesUntilColumn`

```
setter @string appendSpacesUntilColumn ?@uint inColumnIndex
```

Ce *setter* ajoute des espaces en fin de ligne jusqu'à atteindre la colonne `inColumnIndex`.

30.5.2 Setter `decIndentation`

```
setter @string decIndentation ?@uint inAmount
```

Lorsqu'un retour à la ligne est inséré, des espaces sont automatiquement insérés à la suite. Initialement, ce nombre est nul. Le *setter* `incIndentation` *du type* `@string` *(page* **??***)* permet de l'incrémenter, ce *setter* de le décrémenter. Ces deux *setters* permettent d'obtenir facilement des sorties où le texte est indenté.

30.5.3 Setter incIndentation

```
setter @string incIndentation ?@uint inAmount
```

Lorsqu'un retour à la ligne est inséré, des espaces sont automatiquement insérés à la suite. Initialement, ce nombre est nul. Ce *setter* permet de l'incrémenter, le *setter* `decIndentation` *du type* `@string` *(page* **??***)* de le décrémenter. Ces deux *setters* permettent d'obtenir facilement des sorties où le texte est indenté.

30.5.4 Setter insertCharacterAtIndex

```
setter @string insertCharacterAtIndex
    ?@char inChar
    ?@uint inIndex
```

Ce *setter* insère le caractère caractère `inChar` à l'indice `inIndex`. Si `inIndex` doit être inférieur ou égal au nombre de caractères du récepteur. Si `inIndex` est égal au nombre de caractères, `inChar` est inséré après le dernier caractère.

30.5.5 Setter setCapacity

```
setter @string setCapacity ?@uint inCapacity
```

Ce *setter* ajuste la zone mémoire allouée au buffer de la chaîne de caractères. Si `inCapacity` est strictement inférieur au nombre de caractères, ce *setter* n'a pas d'effet.

30.5.6 Setter removeCharacterAtIndex

```
setter @string removeCharacterAtIndex
    !@char outChar
    ?@uint inIndex
```

Ce *setter* retire de la chaîne le caractère à l'indice `inIndex`. Le caractère retiré est renvoyé dans `outChar`. `inIndex` doit être strictement inférieur au nombre de caractères ; sinon, une erreur d'exécution est déclenchée, et une valeur poison est retournée dans `outChar`.

30.5.7 Setter setCharacterAtIndex

```
setter @string setCharacterAtIndex
    ?@char inChar
    ?@uint inIndex
```

Ce *setter* remplace le caractère d'indice `inIndex` par le caractère `inChar`. Si `inIndex` est supérieur ou égal au nombre de caractères, une erreur d'exécution est déclenchée.

30.6 Procédures de type

30.6.1 Procédure de type `deleteFile`

```
proc @string deleteFile ?@string inFilePath
```

Supprime le fichier `inFilePath`. Une erreur d'exécution est déclenchée si le fichier n'existe pas.

30.6.2 Procédure de type `deleteFileIfExists`

```
proc @string deleteFileIfExists ?@string inFilePath
```

Supprime le fichier `inFilePath`. Aucune erreur d'exécution n'est déclenchée si le fichier n'existe pas.

30.6.3 Procédure de type `generateFile`

```
proc @string generateFile
    ?@string inStartPath
    ?@string inFileName
    ?@string inContents
```

Cette procédure commence par rechercher le fichier `inFileName` dans le répertoire `inStartPath`, et récursivement dans ses sous-répertoires.

Si le fichier existe, son contenu est remplacé par `inContents`.

Si il n'existe pas, il est créé dans le répertoire `inStartPath` avec le nom `inFileName`, et le contenu `inContents`.

30.6.4 Procédure de type `generateFileWithPattern`

```
proc @string generateFileWithPattern
    ?startPath:@string inStartPath
    ?fileName:@string inFileName
    ?lineComment:@string inLineCommentPrefix
    ?header:@string inHeader
    ?defaultUserZone1:@string inDefaultUserZone1
    ?generatedZone2:@string inGeneratedZone2
```

```
    ?defaultUserZone2:@string inDefaultUserZone2
    ?generatedZone3:@string inGeneratedZone3
    ?makeExecutable:@bool inMakeExecutable
```

30.6.5 Procédure de type `removeDirectoryRecursively`

```
proc @string removeDirectoryRecursively ?@string inDirectoryPath
```

Supprime le répertoire `inDirectoryPath`, après avoir supprimé tous ses fichiers et récursivement tous ses sous-répertoires.

30.6.6 Procédure de type `removeEmptyDirectory`

```
proc @string removeEmptyDirectory ?@string inDirectoryPath
```

Supprime le répertoire `inDirectoryPath`. Une erreur d'exécution est déclenchée si le répertoire n'est pas vide.

Chapitre 31

Le type `@stringset`

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

31.1 Constructors

31.1.1 Constructeur `emptySet`

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

Creates and returns an empty `@stringset` object.

31.1.2 Constructeur `setWithString`

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

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

31.2 Getters

31.2.1 Getter `anyString`

```
getter anyString -> @string
```

Retourne une des chaînes de caractères contenue dans le récepteur. Si le récepteur est vide, une erreur d'exécution est déclenchée.

31.2.2 Getter count

```
getter count -> @uint
```

Returns the number of strings in the set.

31.2.3 Getter hasKey

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

Returns a boolean value that indicates whether the value of *inString* argument is present in the set : `true` if the value of *inString* argument is present in the set, `false` otherwise.

31.2.4 Getter stringList

```
getter stringList -> @stringlist
```

Retourne la valeur du récepteur sous la forme d'une liste. L'ordre de la liste est l'ordre alphabétique.

31.3 Setter

31.3.1 Setter removeKey

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

Removes the value of *inString* argument from the receiver's value. If the receiver's value does not contain the value of *inString* argument, this setter leaves the receiver's value unchanged.

31.4 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 ;
```

31.5 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
```

31.6 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
```

31.7 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
```

31.8 Enumerating `@stringset` objects

The `for` 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 ;
```

31.9 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 32

Le type `@timer`

Le type `@timer` permet de mesurer des durées d'exécution de portions de code ; une utilisation typique est :

```
var @timer t = .start
  # instructions
message "Durée : " + [t string] + "\n" # Affiche la durée d'exécution des instructions
```

32.1 Constructeurs

Le type `@timer` accepte deux constructeurs :

- le contructeur `start` ;
- le constructeur `default` (section **??** page **??**), qui a le même effet que le constructeur `start` .

32.1.1 Constructeur `start`

```
constructor @timer start -> @timer
```

Appeler le constructeur `start` est la seule façon d'instancier un objet `@timer` . Le chronomètre est enclenché, c'est-à-dire qu'il compte la durée à partir de laquelle le constructeur `start` a été appelé.

32.2 Setters

Le type `@timer` accepte deux *setters* :

- le *setter* `resume` ;
- le *setter* `stop` .

32.2.1 Setter `resume`

```
setter @timer resume
```

Le *setter* `resume` redémarre le chronomètre si il est arrêté, et le réinitialise si il est en marche.

32.2.2 Setter `stop`

```
setter @timer stop
```

Le *setter* `stop` arrête le chronomètre. Si il est déjà arrêté, appeler ce *setter* n'a aucun effet.

32.3 Getters

Le type `@timer` accepte trois *getters* :

- le *getter* `isRunning` ;
- le *getter* `msFromStart` ;
- le *getter* `string` .

32.3.1 Getter `isRunning`

```
getter @timer isRunning -> @bool
```

Ce *getter* renvoie `true` si le récepteur décompte le temps, ou `false` si il a été arrêté par un appel au *setter* `stop` .

32.3.2 Getter `msFromStart`

```
getter @timer msFromStart -> @uint
```

La valeur obtenue par le *getter* `msFromStart` est la durée écoulée depuis son instanciation (par le constructeur `start`) ou depuis le dernier appel au *setter* `resume` . La durée est exprimée en millisecondes.

32.3.3 Getter `string`

```
getter @timer string -> @string
```

La valeur obtenue par le *getter* `string` est la durée écoulée depuis son instanciation (par le constructeur `start`) ou depuis le dernier appel au *setter* `resume`. La durée est exprimée sous la forme d'une chaîne de caractères.

Chapitre 33

Le type `@type`

Chapitre 34

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.

34.1 Constructors

34.1.1 Constructeur `errorCount`

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

Returns an `@uint` object that contains the number of errors. The returned value is the cumulative count of errors from the beginning of execution.

Exemple :

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

34.1.2 Constructeur `max`

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

Returns an `@uint` object that the maximum value of the 32-bit unsigned range ($2^{32} - 1$).

34.1.3 Constructeur random

```
constructor random -> @uint
```

Retourne une valeur aléatoire de type `@uint`. La procédure de type *procédure `setRandomSeed` du type `@uint` (page **??**)* permet d'en fixer la valeur initiale.

```
  let v = @uint.random
```

Note. Sur Unix, la valeur renvoyée est la valeur renvoyée par l'appel de la fonction `random` de la librairie `libc`. Sur Windows, c'est la fonction `rand` qui est appelée.

34.1.4 Constructeur valueWithMask

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

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

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 (0b1_1100)
```

34.1.5 Constructeur warningCount

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

Returns an `@uint` object that contains the number of warnings. The returned value is the cumulative count of warnings from the beginning of execution.

34.2 Procédure de type

34.2.1 Procédure de type setRandomSeed

```
proc @uint setRandomSeed ?@uint inSeed
```

Affecte la valeur initiale utilisée par le générateur de nombres aléatoires (voir le *constructeur `random` du type `@uint` – page **??***) Par exemple :

```
  [@uint setRandomSeed !0]
```

34.3 Getters

34.3.1 Getter `alphaString`

Ce *getter* permet de convertir un `@uint` en une chaîne de caractères, telle que l'ordre des entiers est conservé sur la chaîne obtenue.

La chaîne obtenue comporte exactement 7 lettres minuscules. C'est en fait une conversion en base 26, la lettre `a` ayant la valeur 0, et la lettre `z` la valeur 25.

```
message [0 alphaString] + "\n"          # aaaaaaa
message [12_345 alphaString] + "\n"     # aaaasgv
message [@uint.max alphaString] + "\n" # nxmrlxv
```

34.3.2 Getter `bigint`

Ce *getter* permet de convertir un `@uint` en `@bigint`. Comme la plage des valeurs des `bigint` n'est limitée que par la mémoire disponible, il n'échoue jamais.

```
message [[1234 bigint] string] + "\n" # 1234
```

34.3.3 Getter `double`

```
getter double -> @double
```

Returns the receiver's value converted in a `@double` object. As a 32-bit integer value can always be converted in a `@double` value, this getter never fails.

34.3.4 Getter `hexString`

```
getter hexString -> @string
```

Returns the an hexadecimal string representation of the receiver value, prefixed by the string `0x`. For getting an hexadecimal representation string without any prefix, see *getter* `xString` *du type* `@uint` *(page* **??***)*
.

34.3.5 Getter `hexStringSeparatedBy`

```
getter hexStringSeparatedBy ?@char inSeparator ?@uint inGroup -> @string
```

Returns the an hexadecimal string representation of the receiver value, prefixed by the string `0x`. Groups of `inGroup` digits are separated by the `inSeparator` character.

If `inGroup` is equal to zero, a run-time error is raised.

For example :

```
let s = [0x12345678 hexStringSeparatedBy !'_' !2] # 0x12_34_56_78
```

34.3.6 Getter isInRange

```
getter isInRange ?@range inRange -> @bool
```

Returns an `@bool` value indicating whether the receiver'value belongs to `inRange` range : 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.

34.3.7 Getter isUnicodeValueAssigned

```
getter isUnicodeValueAssigned -> @bool
```

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

34.3.8 Getter lsbIndex

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

Returns an `@uint` value of the index of the most significant bit of the receiver value. It raises a run-time error if the receiver value is zero.

Exemple :

```
@uint value = 192 ; # 192 is ...011000000 in binary
@uint x = [value lsbIndex] ; # x is equal to 7
```

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

34.3.9 Getter significantBitCount

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

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

34.3.10 Getter `sint`

```
getter sint -> @sint
```

Returns the receiver's value in an `@sint` (page **??**) (32-bit signed integer) object. An error is raised is receiver's value is greater than $2^{31} - 1$.

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

34.3.11 Getter `sint64`

```
getter sint64 -> @sint64
```

Returns the receiver's value in an `@sint64` (page **??**) (64-bit signed integer) object. As a 32-bit unsigned value can always be converted in a 64-bit signed value, this getter never fails.

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

34.3.12 Getter `string`

```
getter string -> @string
```

Returns a decimal string representation of the receiver's value. For an hexadecimal string representation of the receiver's value, see *getter* `hexString` *du type* `@uint` *(page* **??***)* and *getter* `xString` *du type* `@uint` *(page* **??***)* .

34.3.13 Getter `uint64`

```
getter uint64 -> @uint64
```

Returns the receiver's value in an `@uint64` (page **??**) (64-bit unsigned integer) object. As a 32-bit unsigned value can always be converted in a 64-bit unsigned value, this getter never fails.

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

34.3.14 Getter `xString`

```
getter xString -> @string
```

Returns an hexadecimal string representation of the receiver's value (without any prefix). For an decimal string representation of the receiver's value, see the *getter* `hexString` *du type* `@uint` *(page* **??***)*; for a decimal string representation of the receiver's value, see the *getter* `string` *du type* `@uint` *(page* **??***)* .

34.4 Arithmétique

34.4.1 Opérateurs infixés

Le type `@uint` accepte les opérateurs arithmétiques infixés suivants :

- `+`, addition, une erreur d'exécution est déclenchée en cas de débordement ;
- `-`, soustraction, une erreur d'exécution est déclenchée en cas de débordement ;
- `*`, multiplication, une erreur d'exécution est déclenchée en cas de débordement ;
- `/`, division, une erreur d'exécution est déclenchée si le diviseur est nul ;
- `mod`, calcul du reste, une erreur d'exécution est déclenchée si le diviseur est nul ;
- `&+`, addition, le résultat étant silencieusement tronqué sur 32 bits ;
- `&-`, soustraction, le résultat étant silencieusement tronqué sur 32 bits ;
- `&*`, multiplication, le résultat étant silencieusement tronqué sur 32 bits ;
- `&/`, division, qui retourne zéro si le diviseur est nul.

Ces opérateurs exigent que les deux opérandes soient des objets du même type `@uint`.

34.4.2 Opérateur préfixé

Le type `@uint` accepte un opérateur arithmétique préfixé :

- `+`, qui retourne simplement la valeur de l'opérande.

34.4.3 Instructions

Le type `@uint` accepte les deux instructions arithmétiques suivantes :

- `+=`, addition, une erreur d'exécution est déclenchée en cas de débordement ;
- `-=`, soustraction, une erreur d'exécution est déclenchée en cas de débordement ;
- `*=`, multiplication, une erreur d'exécution est déclenchée en cas de débordement ;
- `/=`, division, une erreur d'exécution est déclenchée en cas division par zéro ;
- `++`, incrémentation, une erreur d'exécution est déclenchée en cas de débordement ;
- `--`, décrémentation, une erreur d'exécution est déclenchée en cas de débordement ;
- `&++`, incrémentation, le résultat étant silencieusement tronqué sur 32 bits ;

- `&--` , décrémentation, le résultat étant silencieusement tronqué sur 32 bits.

`x+=y` est équivalent à `x=x+y` ; `x-=y` est équivalent à `x=x-y` . La variable cible `x` , comme l'expression source `y` doivent être du même type `@uint` .

Incrémentation et décrémentation sont des instructions, et ne peuvent pas apparaître des expressions.

```
@uint n = ... ; n ++ # Incrémentation
```

```
@uint n = ... ; n -- # Décrémentation
```

34.5 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.

34.6 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.

34.7 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 35

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.

35.1 Constructeurs

35.1.1 Constructeur `max`

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

Returns an `@uint64` object that the maximum value of the 64-bit unsigned range ($2^{64} - 1$).

35.1.2 Constructeur `uint64BaseValueWithCompressedBitString`

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

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

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>
```

35.1.3 Constructeur `uint64MaskWithCompressedBitString`

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

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'.

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> ;
```

35.1.4 Constructeur `uint64WithBitString`

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

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

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> ;
```

35.2 Getters

35.2.1 Getter alphaString

Ce *getter* permet de convertir un `@uint64` en une chaîne de caractères, telle que l'ordre des entiers est conservé sur la chaîne obtenue.

La chaîne obtenue comporte exactement 14 lettres minuscules. C'est en fait une conversion en base 26, la lettre `a` ayant la valeur 0, et la lettre `z` la valeur 25.

```
   message [0L alphaString] + "\n"           # aaaaaaaaaaaaaa
   message [12_345L alphaString] + "\n"      # aaaaaaaaaaasgv
   message [@uint64.max alphaString] + "\n" # hlhxczmxsyumqp
```

35.2.2 Getter bigint

Ce *getter* permet de convertir un `@uint64` en `@bigint`. Comme la plage des valeurs des `bigint` n'est limitée que par la mémoire disponible, il n'échoue jamais.

```
   message [[1234L bigint] string] + "\n" # 1234
```

35.2.3 Getter double

```
getter double -> @double
```

Returns the receiver's value converted in a `@double` object. As a 64-bit integer value can always be converted in a `@double` value, this getter never fails.

35.2.4 Getter hexString

```
getter hexString -> @string
```

Returns the an hexadecimal string representation of the receiver value, prefixed by the string `0x`. For getting an hexadecimal representation string with any prefix, see *getter* `xString` *du type* `@uint64` *(page* **??***)*
.

35.2.5 Getter hexStringSeparatedBy

```
getter hexStringSeparatedBy ?@char inSeparator ?@uint inGroup -> @string
```

Returns the an hexadecimal string representation of the receiver value, prefixed by the string `0x`. Groups of `inGroup` digits are separated by the `inSeparator` character.

If `inGroup` is equal to zero, a run-time error is raised.

For example :

```
let s = [0x123456789ABCDEF0L hexStringSeparatedBy !'_' !4] # 0x1234_5678_9ABC_DEF0
```

35.2.6 Getter sint

```
getter sint -> @sint
```

Returns the receiver's value in an `@sint` (page **??**) (32-bit signed integer) object. An error is raised is receiver's value is greater than $2^{31} - 1$. This getter is the only way to convert an `@uint64` (page **??**) object into an `@sint` (page **??**) object.

35.2.7 Getter sint64

```
getter sint64 -> @sint64
```

Returns the receiver's value in an `@sint64` (page **??**) (64-bit signed integer) object. An error is raised is receiver's value is greater than $2^{63} - 1$. This getter is the only way to convert an `@uint64` (page **??**) object into an `@sint64` (page **??**) object.

35.2.8 Getter string

```
getter string -> @string
```

Returns a decimal string representation of the receiver's value. For an hexadecimal string representation of the receiver's value, see *getter* `hexString` *du type* `@uint64` *(page* **??***)* and *getter* `xString` *du type* `@uint64` *(page* **??***)* .

35.2.9 Getter uint

```
getter uint -> @uint
```

Returns the receiver's value in an `@uint` (page **??**) (32-bit unsigned integer) object. An error is raised is receiver's value is greater than $2^{32} - 1$. This getter is the only way to convert an `@uint64` (page **??**) object into an `@uint` (page **??**) object.

35.2.10 Getter uintSlice

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

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

35.2.11 Getter xString

```
getter xString -> @string
```

Returns an hexadecimal string representation of the receiver's value (without any prefix). For an decimal string representation of the receiver's value, see the *getter* `hexString` *du type* `@uint64` *(page* **??***)*; for a decimal string representation of the receiver's value, see the *getter* `string` *du type* `@uint64` *(page* **??***)* .

35.3 Arithmétique

35.3.1 Opérateurs infixés

Le type `@uint64` accepte les opérateurs arithmétiques infixés suivants :

- `+` , addition, une erreur d'exécution est déclenchée en cas de débordement ;
- `-` , soustraction, une erreur d'exécution est déclenchée en cas de débordement ;
- `*` , multiplication, une erreur d'exécution est déclenchée en cas de débordement ;
- `/` , division, une erreur d'exécution est déclenchée si le diviseur est nul ;
- `mod` , calcul du reste, une erreur d'exécution est déclenchée si le diviseur est nul ;
- `&+` , addition, le résultat étant silencieusement tronqué sur 64 bits ;
- `&-` , soustraction, le résultat étant silencieusement tronqué sur 64 bits ;
- `&*` , multiplication, le résultat étant silencieusement tronqué sur 64 bits ;
- `&/` , division, qui retourne zéro si le diviseur est nul.

Ces opérateurs exigent que les deux opérandes soient des objets du même type `@uint64` .

35.3.2 Opérateur préfixé

Le type `@uint64` accepte un opérateur arithmétique préfixé :

- `+`, qui retourne simplement la valeur de l'opérande.

35.3.3 Instructions

Le type `@uint64` accepte les instructions arithmétiques suivantes :

- `+=`, addition, une erreur d'exécution est déclenchée en cas de débordement ;
- `-=`, soustraction, une erreur d'exécution est déclenchée en cas de débordement ;
- `*=`, multiplication, une erreur d'exécution est déclenchée en cas de débordement ;
- `/=`, division, une erreur d'exécution est déclenchée en cas division par zéro ;
- `++`, incrémentation, une erreur d'exécution est déclenchée en cas de débordement ;
- `--`, décrémentation, une erreur d'exécution est déclenchée en cas de débordement ;
- `&++`, incrémentation, le résultat étant silencieusement tronqué sur 64 bits ;
- `&--`, décrémentation, le résultat étant silencieusement tronqué sur 64 bits.

`x+=y` est équivalent à `x=x+y` ; `x-=y` est équivalent à `x=x-y`. La variable cible `x`, comme l'expression source `y` doivent être du même type `@uint64`.

Incrémentation et décrémentation sont des instructions, et ne peuvent pas apparaître des expressions.

```
@uint64 n = ... ; n ++ # Incrémentation
```

```
@uint64 n = ... ; n -- # Décrémentation
```

35.4 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.

35.5 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.

35.6 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 36

Le type `list`

36.1 Déclaration d'un type de liste

La déclaration d'un type `list` nomme toutes les propriétés des éléments de la liste :

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

36.2 Constructeurs

36.2.1 Le constructeur `emptyList`

Pour chaque liste, le constructeur `emptyList` est implicitement déclaré. Il retourne une liste vide :

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

36.2.2 Le constructeur `listWithValue`

Ce constructeur permet de construire directement une liste contenant un élément :

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

Using this constructor is equivalent to :

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

36.3 Adding elements

36.3.1 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
```

36.3.2 L'instruction +=

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

```
@MyList aList = ...
@MyList secondList = ...
aList += secondList
```

36.3.3 Le setter append

Le setter append permet d'ajouter un élément à la fin de la liste, à partir d'un objet du type *élément de la liste* implicitement déclaré.

```
@MyList aList = ...
@string aString = ...
@bool aBool = ...
@MyList-element élément = .new {!aString !aBool}
[!?aList append !élément]
```

36.3.4 The prependValue setter

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

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

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

36.3.5 Setter insertAtIndex

Le setter `insertAtIndex` permet d'insérer un nouvel élément à une position quelconque de la liste. Si le type `list` correspondant déclare n champs, l'appel du setter comprend $n + 1$ arguments :

- les n premiers correspondent aux valeurs des champs du nouvel élément inséré ;
- le dernier est l'indice d'insertion, une valeur de type `@uint` .

L'indice d'insertion peut varier entre 0 (insertion au début, comme le faisait le setter *prependValue*), et la longueur courante de la liste (insertion à la fin, comme le fait l'opérateur `+=` , section **??** page **??**). 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]
```

36.3.6 The concatenation operator

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

```
@MyList firstList = ..
@MyList secondList = ..
@MyList thirdList = firstList + secondList
```

36.4 Removing elements

36.4.1 Setter popFirst

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

```
@MyList aList = ...
@string aString
@bool aBool
```

```
[!?aList popFirst ?aString ?aBool]
```

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

36.4.2 Setter `popLast`

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

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

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

36.5 Methods

36.5.1 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.

36.5.2 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.

36.6 Getters

36.6.1 Le getter count

```
getter count -> @uint
```

Le *getter* `count` retourne le nombre d'éléments du récepteur.

36.6.2 Le getter length

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

Le *getter* `length` retourne le nombre d'éléments du récepteur. Obsolète : utiliser `count`.

36.6.3 Le getter range

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

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

36.6.4 Le getter subListFromIndex

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

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

36.6.5 Le getter subListToIndex

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

Ce *getter* retourne une liste comprenant les éléments du récepteur à partir de l'indice 0 jusqu'à l'indice `inIndex` compris. Si `inIndex` est supérieur ou égal au nombre d'éléments de la liste, une erreur d'exécution est déclenchée.

36.6.6 Le getter subListWithRange

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

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

36.7 Enumerating a list with a `for` instruction

The `for` 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
}
```

36.7.1 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.

```
for () in 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
```

36.7.2 Enumeration using the explicitly declared constants

The `for` 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.

```
for (kString kBool) in aList 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
```

36.7.3 Enumeration in the reverse order

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

```
for > (kString kBool) in aList do
  ...
end
```

36.8 Direct Access of an element attribute

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

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

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

36.8.1 Read Access

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

Si la propriété a été déclarée `private` , seules les *méthodes*, *getters* et *setters* de cette classe peuvent accéder cette propriété.

36.8.2 Write Access

By default, a setter is provided for performing a direct write access to an attribute at a given index.

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

For example :

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

Si la propriété a été déclarée `private`, seules les *méthodes*, *getters* et *setters* de cette classe peuvent accéder cette propriété.

36.8.3 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">
```

36.9 Types liste prédéfinis

36.9.1 Le type @2stringlist

Le type `@2stringlist` est implicitement déclaré de la façon suivante :

```
list @2stringlist {
  @string mValue0
  @string mValue1
}
```

36.9.2 Le type @2lstringlist

Le type `@2lstringlist` est implicitement déclaré de la façon suivante :

```
list @2lstringlist {
  @lstring mValue0
  @lstring mValue1
}
```

36.9.3 Le type @bigintlist

Le type `@bigintlist` est implicitement déclaré de la façon suivante :

```
list @bigintlist {
  @bigint mValue
}
```

36.9.4 Le type @functionlist

Le type `@functionlist` est implicitement déclaré de la façon suivante :

```
list @functionlist {
  @function mValue
}
```

36.9.5 Le type @lbigintlist

Le type `@lbigintlist` est implicitement déclaré de la façon suivante :

```
list @lbigintlist {
  @lbigint mValue
}
```

36.9.6 Le type `@luintlist`

Le type `@luintlist` est implicitement déclaré de la façon suivante :

```
list @luintlist {
  @luint mValue
}
```

36.9.7 Le type `@lstringlist`

Le type `@lstringlist` est implicitement déclaré de la façon suivante :

```
list @lstringlist {
  @lstring mValue
}
```

36.9.8 Le type `@objectlist`

Le type `@objectlist` est implicitement déclaré de la façon suivante :

```
list @objectlist {
  @object mValue
}
```

36.9.9 Le type `@stringlist`

Le type `@stringlist` est implicitement déclaré de la façon suivante :

```
list @stringlist {
  @string mValue
}
```

36.9.10 Le type `@typelist`

Le type `@typelist` est implicitement déclaré de la façon suivante :

```
list @typelist {
  @type mValue
}
```

36.9.11 Le type @uintlist

Le type `@uintlist` est implicitement déclaré de la façon suivante :

```
list @uintlist {
  @uint mValue
}
```

36.9.12 Le type @uint64list

Le type `@uint64list` est implicitement déclaré de la façon suivante :

```
list @uint64list {
  @uint64 mValue
}
```

Chapitre 37

Le type `sortedlist`

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

37.1 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 **??** page **??**) ;
- le constructeur `sortedListWithValue` qui construit une liste contenant un élément (section **??** page **??**) ;
- la construction `{...}` qui permet de construire explicitement une liste ordonnée (section **??** page **??**) ;

- l'opérateur `+=` pour ajouter un élément à une liste ordonnée (section **??** page **??**);
- l'opérateur `+=` pour ajouter tous les éléments d'une liste à une liste ordonnée (section **??** page **??**);
- l'opérateur `+` pour construire une liste ordonnée à partir de deux listes ordonnées (section **??** page **??**);
- le *getter* `length`, qui retourne le nombre d'éléments d'une liste (section **??** page **??**);
- le *setter* `popGreatest`, qui retourne les champs du plus grand élément d'une liste, et retire cet élément de cette liste (section **??** page **??**);
- le *setter* `popSmallest`, qui retourne les champs du plus grand élément d'une liste, et retire cet élément de cette liste (section **??** page **??**);
- la *méthode* `greatest`, qui retourne les champs du plus grand élément d'une liste sans la modifier (section **??** page **??**);
- la *méthode* `smallest`, qui retourne les champs du plus petit élément d'une liste sans la modifier (section **??** page **??**).

37.2 Constructeurs

37.2.1 Constructeur `emptySortedList`

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

```
@MaListeOrdonnee uneListe = @MaListeOrdonnee.emptySortedList
```

L'inférence de type permet de mentionner le type de liste une seule fois. On peut écrire :

```
var uneListe = @MaListeOrdonnee.emptySortedList
```

Ou bien

```
@MaListeOrdonnee uneListe = .emptySortedList
```

37.2.2 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 = @MaListeOrdonnee.sortedListWithValue {
  !'a' # Affecte au champ mCaractere
  !10  # Affecte au champ mEntier
}
```

Ici aussi, l'inférence de type permet de mentionner le type de liste une seule fois. On peut écrire :

```
var uneListe = @MaListeOrdonnee.sortedListWithValue {!'a' !10}
```

Ou bien

```
@MaListeOrdonnee uneListe = .sortedListWithValue {!'a' !10}
```

37.3 Opérateurs

37.3.1 Opérateur {...}

Cette construction permet de s'affranchir des constructeurs `emptySortedList` et `sortedListWithValue`. Pour initialiser une liste ordonnée vide, on peut écrire :

```
@MaListeOrdonnee uneListe = @MaListeOrdonnee {}
```

L'inférence de type permet de mentionner le type de liste une seule fois. On peut écrire :

```
var uneListe = @MaListeOrdonnee {}
```

Ou bien

```
@MaListeOrdonnee uneListe = {}
```

Pour initialiser une liste contenant un élément (en exploitant l'inférence de type) :

```
@MaListeOrdonnee uneListe = {!'a' !10}
```

On peut mentionner un nombre quelconque d'éléments, en les séparant par des virgules :

```
@MaListeOrdonnee uneListe = {!'a' !10, !'c' !5, !'b' !20}
```

37.3.2 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 = {}
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
```

37.3.3 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 ;
```

37.3.4 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 ;
```

37.4 Getter count

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

37.5 Getter length

Le getter `length` retourne un `@uint` contenant le nombre d'éléments de la liste ordonnée. Obsolète, utiliser `count`.

37.6 Setters

37.6.1 Setter popGreatest

Ce *setter* retourne les champs du plus grand élément de la liste ordonnée, et le retire. Si la liste est vide, un message d'erreur est affiché, et les variables destinées à recevoir les valeurs des champs sont placées dans l'état *invalide*. Par exemple :

```
@MaListeOrdonnee uneListe = ...
...
[!?uneListe popGreatest
  ?@char c
  ?@uint n
]
```

Si `uneListe` est vide, les variables `c` et `n` sont placées dans l'état *invalide*.

37.6.2 Setter `popSmallest`

Ce *setter* retourne les champs du plus petit élément de la liste ordonnée, et le retire. Si la liste est vide, un message d'erreur est affiché, et les variables destinées à recevoir les valeurs des champs sont placées dans l'état *invalide*. Par exemple :

```
@MaListeOrdonnee uneListe = ...
...
[!?uneListe popSmallest
  ?@char c
  ?@uint n
]
```

Si `uneListe` est vide, les variables `c` et `n` sont placées dans l'état *invalide*.

37.7 Méthodes

37.7.1 La méthode `greatest`

Cette méthode retourne les champs du plus grand élément de la liste ordonnée, sans le retirer. La liste n'est donc pas modifiée. Si elle est vide, un message d'erreur est affiché, et les variables destinées à recevoir les valeurs des champs sont placées dans l'état *invalide*. Par exemple :

```
@MaListeOrdonnee uneListe = ...
...
[uneListe greatest
  ?@char c
  ?@uint n
]
```

Si `uneListe` est vide, les variables `c` et `n` sont placées dans l'état *invalide*.

37.7.2 La méthode `smallest`

Cette méthode retourne les champs du plus petit élément de la liste ordonnée, sans le retirer. La liste n'est donc pas modifiée. Si elle est vide, un message d'erreur est affiché, et les variables destinées à recevoir les valeurs des champs sont placées dans l'état *invalide*. Par exemple :

```
@MaListeOrdonnee uneListe = ...
...
[uneListe smallest
  ?@char c
  ?@uint n
]
```

Si `uneListe` est vide, les variables `c` et `n` sont placées dans l'état *invalide*.

37.8 Énumération avec l'instruction `for`

L'instruction `for` (section **??** page **??**) permet d'énumérer les éléments d'une liste ordonnée, par ordre croissant ou décroissant.

Pour effectuer l'énumération par ordre croissant, écrire :

```
for () in uneListe do
  ...
end
```

Pour effectuer l'énumération par ordre décroissant, écrire :

```
for > () in uneListe do
  ...
end
```

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 = {}
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
var s = "" ;
```

```
for () in uneListe do
  s += [mCaractere string] + [mEntier string] + " "
end
message s + "\n" ; # Affiche "a1 b2 b1 c1 d1 f2 f1"
s = ""
for > () in uneListe do
  s += [mCaractere string] + [mEntier string] + " "
end
message s + "\n" # Affiche "f1 f2 d1 c1 b1 b2 a1"
```

Chapitre 38

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.

38.1 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] ;
```

38.2 Constructeur d'un type tableau

Le seul constructeur d'un type tableau est le constructeur `new`. Il a pour but de fixer les dimensions initiales du tableau (il pourra ensuite être redimensionné). Il comporte *dimension* arguments de type `@uint`, qui fixent la taille initiale de chaque axe. Par exemple :

```
@monTableau t [new !2 !3 !4] ;
```

Cette déclaration crée un tableau à $2 * 3 * 4$ éléments. Ces éléments sont par défaut *invalides*, c'est à dire que leur lecture par le getter `valueAtIndex` déclenche une *run-time error*. Pour être valide, un élément doit avoir été initialisé par un appel au setter `setValueAtIndex` .

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

38.3 Accès à un élément

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

38.3.1 Le getter `valueAtIndex`

Ce getter comporte *dimension* arguments de type `@uint` , qui précisent l'indice pour chaque axe. Les indices sont comptés à partir de zéro (comme en C).

Par exemple :

```
@string s = [t valueAtIndex !1 !2 !2] ;
```

Une *run-time error* est déclenchée si un indice dépasse sa borne correspondante, et la valeur retournée est *invalide*. Si les indices ont des valeurs correctes, l'élément est retourné ; si cet élément est invalide, une *run-time error* est déclenchée, et une valeur *invalide* est retournée.

38.3.2 Setter `setValueAtIndex`

Ce setter comporte (*dimension*+1) arguments :

- le premier argument est type `@TypeElement` , et contient la valeur à écrire ;
- les *dimension* suivants arguments sont de type `@uint` et précisent l'indice pour chaque axe.

Les indices sont comptés à partir de zéro (comme en C). Une *run-time error* est déclenchée si un indice dépasse sa borne correspondante, et le tableau est alors non modifié.

Par exemple :

```
@string s = ... ;
[!?t setValueAtIndex !s !1 !2 !2] ;
```

38.3.3 Setter `forceValueAtIndex`

Ce setter comporte (*dimension*+1) arguments :

- le premier argument est type `@TypeElement`, et contient la valeur à écrire ;
- les *dimension* suivants arguments sont de type `@uint` et précisent l'indice pour chaque axe.

Les indices sont comptés à partir de zéro (comme en C). Contrairement au setter `setValueAtIndex`, aucune *run-time error* n'est déclenchée si un indice dépasse sa borne correspondante : le tableau est d'abord agrandi, ce qui ajoute des éléments invalides, puis l'élément désigné par les indices est affecté.

Par exemple :

```
@string s = ... ;
[}?t forceValueAtIndex !s !5 !4 !4] ;
```

38.4 Validité d'un élément

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

38.4.1 Le getter `isValueValidAtIndex`

Ce getter comporte *dimension* arguments de type `@uint`, qui précisent l'indice pour chaque axe. Les indices sont comptés à partir de zéro (comme en C). Une *run-time error* est déclenchée si un indice dépasse sa borne correspondante, et la valeur retournée est *invalide*. Il renvoie une valeur de type `@bool`, suivant que l'élément est valide ou non.

Par exemple :

```
@bool b = [t isValueValidAtIndex !1 !2 !2] ;
```

38.4.2 Setter `invalidateValueAtIndex`

Ce setter comporte *dimension* arguments de type `@uint`, qui précisent l'indice pour chaque axe. Les indices sont comptés à partir de zéro (comme en C). Une *run-time error* est déclenchée si un indice dépasse sa borne correspondante. Il invalide l'élément correspondant, c'est dire qu'un appel au getter `valueAtIndex` pour lire cet élément déclenchera une *run-time error*.

Par exemple :

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

38.5 Contrôle des tailles des axes

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

permettent de modifier la taille d'un tableau.

38.5.1 Le getter `axisCount`

Ce getter sans argument renvoie un `@uint` qui contient le nombre d'axes d'un tableau. Comme ce nombre est fixé statiquement par la déclaration de type, la valeur retournée est toujours la même, pour toutes les objets d'un même type tableau.

Par exemple, pour la déclaration :

```
array @monTableau : @string [3] ;
```

Pour tous les objets de type `@monTableau`, l'appel au getter `axisCount` renvoie la valeur 3.

38.5.2 Le getter `sizeForAxis`

Ce getter présente un argument de type `@uint` qui est l'indice de l'axe interrogé. Les axes sont numérotés à partir de zéro, c'est à dire que le premier axe a l'indice 0, le deuxième l'indice 1, ... 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.

38.5.3 Le getter `rangeForAxis`

Ce getter présente un argument de type `@uint` qui est l'indice de l'axe interrogé. Les axes sont numérotés à partir de zéro, c'est à dire que le premier axe a l'indice 0, le deuxième l'indice 1, ... 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.

38.5.4 Setter `setSizeForAxis`

Ce setter permet de changer la taille d'un axe sans changer les tailles attribuées aux autres axes. Il présente deux arguments de type `@uint` :

- le premier est la nouvelle taille ;
- le second est l'indice de l'axe concerné.

Les axes sont numérotés à partir de zéro, c'est à dire que le premier axe a l'indice 0, le deuxième l'indice 1, ... Une *run-time error* est déclenchée si la valeur de l'argument est supérieure ou égale à la dimension du tableau, et le tableau n'est pas modifié.

Diminuer la taille d'un axe fait disparaître des éléments, qui sont alors perdus. Si la nouvelle taille est zéro, le tableau est vidé de tous ses éléments.

Augmenter la taille fait apparaître de nouveaux éléments, qui sont invalides par défaut. Il faudra alors explicitement les initialiser individuellement par un appel au setter `setValueAtIndex`.

38.5.5 Setter `setSize`

Ce setter permet de changer les tailles de tous les axes. Il présente `@uint` arguments de type `@uint` qui contiennent les nouvelles tailles de chaque axe.

Diminuer la taille d'un axe fait disparaître des éléments, qui sont alors perdus. Si une des nouvelles tailles est zéro, le tableau est vidé de tous ses éléments.

Augmenter une taille fait apparaître de nouveaux éléments, qui sont invalides par défaut. Il faudra alors explicitement les initialiser individuellement par un appel au setter `setValueAtIndex`.

38.6 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 39

Le type `valueclass`

Le type `valueclass` est un type de classe classique – il inclut l'*héritage simple* et les *classes abstraites* – mais a une sémantique de valeur (section **??** page **??**).

Il n'est pas possible de définir des méthodes dans une classe : on peut le faire uniquement via des *extensions* : chapitre **??** à partir de la page **??**.

39.1 Déclaration d'une classe

Voici différents exemples de déclaration de classes :

```
abstract valueclass @A {
  @uint mA
}
valueclass @B : @A {
  @string mB
}
valueclass @C : @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 propriétés. L'héritage multiple n'est pas implémenté en GALGAS.

Une classe qui hérite d'une autre peut être abstraite :

```
abstract valueclass @D : @C {
  ...
```

```
  }
```

Une classe non abstraite définit implicitement le constructeur `new`, et des *getters* pour lire les propriétés, et des *setters* pour les écrire. On ne peut pas définir explicitement d'autres constructeurs, *getters* ou *setters* à l'intérieur de la classe. Cependant, les extensions (chapitre **??** à partir de la page **??**) permettent de définir *getters*, *méthodes* et *setters* associés à une classe.

39.2 Sémantique de valeur

Une classe déclarée par `valueclass` a une *sémantique de valeur*, c'est-à-dire qu'une affectation entre instances provoque une copie. Prenons un exemple, en considérant la classe :

```
valueclass @classeSemantiqueDeValeur {
  @uint propriété
}
```

Et le fragment de code suivant :

```
@classeSemantiqueDeValeur a = .new {!10}
@classeSemantiqueDeValeur b = a # Copie
[!?a setPropriété !5]
message "Propriété de a " + [a propriété] + "\n" # Propriété de a 5
message "Propriété de b " + [b propriété] + "\n" # Propriété de b 10
```

Lors de l'affectation `b=a`, `b` reçoit une copie de la valeur de `a`, si bien que l'affectation ultérieure de la `propriété` de `a` n'affecte pas `b`.

39.3 Le constructeur new

Le constructeur `new` est implicitement défini pour toute classe non abstraite (c'est à dire les classes `@B` et `@C` de la section **??** page **??**). Ce constructeur présente un argument par propriété déclaré dans la classe instanciée et dans toutes ses super classes. L'ordre des arguments est celui obtenu en parcourant la hiérarchie de classes, en commençant par la classe de base. Par exemple on écrira :

```
@B b = @B.new {
  !0 # Propriété mA de @A
  !"Hello" # Propriété mB de @B
}
@C c = @C.new {
  !0 # Propriété mA de @A
  !"Hello" # Propriété mB de @B
  !@data.emptyData # Propriété mC de @C
}
```

Dans les exemples ci-dessus, les annotations de type apparaissent deux fois, à la déclaration de la variable et devant le constructeur `new`. Si ces deux annotations nomment le même type, l'une d'entre elles peut être omise. Par exemple :

```
@B b = .new {
  !0 # Propriété mA de @A
  !"Hello" # Propriété mB de @B
}
```

Ou bien :

```
var b = @B.new {
  !0 # Propriété mA de @A
  !"Hello" # Propriété mB de @B
}
```

Aucune des deux annotations ne peut être omise si elles nomment des types différents, comme lorsque l'on réalise une affectation polymorphique :

```
@A b = @B.new {
  !0 # Propriété mA de @A
  !"Hello" # Propriété mB de @B
}
```

39.4 Lecture d'une propriété

Par défaut, la lecture d'une propriété est activée par la définition implicite d'un *getter*, dont le nom est le nom de la propriété. Ainsi, pour une classe `@C` :

```
valueclass @C {
  @uint prop
}
```

Et pour une variable `c` de type `@C`, on peut écrire :

```
@uint v = [c prop]
```

À partir de la version 3.3.0, il est possible d'utiliser la notation pointée :

```
@uint v = c.prop
```

Si la propriété a été déclarée `private`, seules les *méthodes*, *getters* et *setters* de cette classe peuvent accéder cette propriété.

39.5 Écriture d'une propriété

Par défaut, une propriété est publique et un *setter* est engendré implicitement. Ce *setter* porte le nom `set<Propriété>`, c'est-à-dire le nom de la propriété avec sa première lettre en majuscule, précédé par `set`. Par exemple :

```
valueclass @C {
  @uint prop
}
```

Pour modifier la propriété `prop` d'un objet `c` instance de `@C`, on écrira :

```
[!?c setProp !12]
```

Si la propriété a été déclarée `private`, seules les *méthodes*, *getters* et *setters* de cette classe peuvent accéder cette propriété.

39.6 Conversions entre objets de classes différentes

Pour toute cette section, nous illustrons les constructions décrites en nous basant sur les trois classes suivantes :

```
valueclass @A {
  ...
}
valueclass @B : @A {
  ...
}
valueclass @C : @B {
  ...
}
```

Nous considérons trois variables `@a`, `b` et `c` respectivement de types `@A`, `@B` et `@C`.

39.6.1 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, ...)

39.6.2 Affectation polymorphique inverse

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 **??** page **??**) ;
- l'expression de test du type dynamique (section **??** page **??**) ;
- l'instruction `cast` (section **??** page **??**).

Chapitre 40

Le type `refclass`

Le type `refclass` est un type de classe classique – il inclut l'*héritage simple* et les *classes abstraites* – et a une sémantique de référence (section **??** page **??**).

Les instances d'une `refclass` sont des *pointeurs forts*, c'est-à-dire que le nombre de pointeurs forts qui désignent un objet est maintenu à jour en permanence. La déallocation est fait automatiquement lorsque le nombre de pointeurs forts tombe à zéro.

Pour chaque type `refclass @T` déclaré, est implicitement déclaré le type `@T-weak`, qui implémente un *pointeur faible* sur les objets de type `@T` (section **??** page **??**). Un pointeur faible n'est pas compté dans le comptage de références, et est mis automatiquement à `nil` lorsque l'objet disparaît.

Il n'est pas possible de définir des méthodes dans une classe : on peut le faire uniquement via des *extensions* : chapitre **??** à partir de la page **??**.

40.1 Déclaration d'une classe

Voici différents exemples de déclaration de classes :

```
abstract refclass @A {
  @uint mA
}
refclass @B : @A {
  @string mB
}
refclass @C : @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 propriétés. L'héritage multiple n'est pas implémenté en GALGAS.

Une classe qui hérite d'une autre peut être abstraite :

```
abstract refclass @D : @C {
  ...
 }
```

Une classe non abstraite définit implicitement le constructeur `new`, et des *getters* pour lire les propriétés, et des *setters* pour les écrire. On ne peut pas définir explicitement d'autres constructeurs, *getters* ou *setters* à l'intérieur de la classe. Cependant, les extensions (chapitre **??** à partir de la page **??**) permettent de définir *getters*, *méthodes* et *setters* associés à une classe.

40.2 Sémantique de référence

Une classe déclarée par `refclass` a une *sémantique de référence*, c'est-à-dire qu'une affectation entre instances provoque un partage de données :

```
shared refclass @classeSemantiqueDeReference {
  @uint propriété
}
```

Et l'exécution devient :

```
@classeSemantiqueDeReference a = .new {!10}
@classeSemantiqueDeReference b = a # Partage
[!?a setPropriété !5]
message "Propriété de a " + [a propriété] + "\n" # Propriété de a 5
message "Propriété de b " + [b propriété] + "\n" # Propriété de b 5
```

L'affectation `b=a` provoque un partage de données, `a` et `b` désigne le même objet : l'affectation de sa `propriété` via `a` est visible via `b`.

40.3 Le constructeur new

Le constructeur `new` est implicitement défini pour toute classe non abstraite (c'est à dire les classes `@B` et `@C` de la section **??** page **??**). Ce constructeur présente un argument par propriété déclaré dans la classe instanciée et dans toutes ses super classes. L'ordre des arguments est celui obtenu en parcourant la hiérarchie de classes, en commençant par la classe de base. Par exemple on écrira :

```
@B b = @B.new {
  !0 # Propriété mA de @A
  !"Hello" # Propriété mB de @B
```

```
}
@C c = @C.new {
  !0 # Propriété mA de @A
  !"Hello" # Propriété mB de @B
  !@data.emptyData # Propriété mC de @C
}
```

Dans les exemples ci-dessus, les annotations de type apparaissent deux fois, à la déclaration de la variable et devant le constructeur `new`. Si ces deux annotations nomment le même type, l'une d'entre elles peut être omise. Par exemple :

```
@B b = .new {
  !0 # Propriété mA de @A
  !"Hello" # Propriété mB de @B
}
```

Ou bien :

```
var b = @B.new {
  !0 # Propriété mA de @A
  !"Hello" # Propriété mB de @B
}
```

Aucune des deux annotations ne peut être omise si elles nomment des types différents, comme lorsque l'on réalise une affectation polymorphique :

```
@A b = @B.new {
  !0 # Propriété mA de @A
  !"Hello" # Propriété mB de @B
}
```

40.4 Lecture d'une propriété

Par défaut, la lecture d'une propriété est activée par la définition implicite d'un *getter*, dont le nom est le nom de la propriété. Ainsi, pour une classe `@C` :

```
refclass @C {
  @uint prop
}
```

Et pour une variable `c` de type `@C`, on peut écrire :

```
@uint v = [c prop]
```

À partir de la version 3.3.0, il est possible d'utiliser la notation pointée :

```
@uint v = c.prop
```

Si la propriété a été déclarée `private`, seules les *méthodes*, *getters* et *setters* de cette classe peuvent accéder cette propriété.

40.5 Écriture d'une propriété

Par défaut, une propriété est publique et un *setter* est engendré implicitement. Ce *setter* porte le nom `set<Propriété>`, c'est-à-dire le nom de la propriété avec sa première lettre en majuscule, précédé par `set`. Par exemple :

```
refclass @C {
  @uint prop
}
```

Pour modifier la propriété `prop` d'un objet `c` instance de `@C`, on écrira :

```
[!?c setProp !12]
```

Si la propriété a été déclarée `private`, seules les *méthodes*, *getters* et *setters* de cette classe peuvent accéder cette propriété.

40.6 Conversions entre objets de classes différentes

Pour toute cette section, nous illustrons les constructions décrites en nous basant sur les trois classes suivantes :

```
refclass @A {
  ...
}
refclass @B : @A {
  ...
}
refclass @C : @B {
  ...
}
```

Nous considérons trois variables `@a`, `b` et `c` respectivement de types `@A`, `@B` et `@C`.

40.6.1 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, ...)

40.6.2 Affectation polymorphique inverse

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 **??** page **??**);
- l'expression de test du type dynamique (section **??** page **??**);
- l'instruction `cast` (section **??** page **??**).

40.6.3 Comparaison

Les opérateurs de comparaison (`=` , `!=` , `<` , `<=` , `>` et `>=`) ne sont pas disponibles pour un pointeur fort. Les opérateurs `===` et `!==` permettent de tester si deux pointeurs forts désignent la même instance.

On peut donc écrire :

```
@A a = @A.new
var aa = a
if a === aa then
  message "same instance\n"
else
  message "different instances\n"
end
```

40.7 Pointeur faible

Il est possible de définir un *pointeur faible* sur une instance d'une `refclass` . Un pointeur faible ne change pas le comptage de références, et est mis automatiquement à `nil` lorsque l'objet disparaît.

Pour une classe est déclarée par `refclass @T` , le type de référence faible sur les instances de cette classe est `@T-weak` ; ce type est implicitement déclaré.

40.7.1 Constructeurs

Il y a deux constructeurs : `default` et `nil` . Ceux-ci ont le même effet, initialisé un pointeur faible à `nil`.

```
var weak1 = @A-weak.nil
@A-weak weak2 = .nil
var weak3 = @A-weak.default
```

40.7.2 Initialisation à partir d'une instance de `refclass @T`

Un objet de type `refclass @T-weak` peut être diectement initialisé à partir d'un objet de type `refclass @T` :

```
var a = @A.new
@A-weak weakA = a
```

Dès que l'objet `a` disparaît, `weakA` est mis à `nil`.

Cette initialisation est polymorphique : si `refclass @B` est une héritière de `refclass @A`, on peut écrire :

```
var b = @B.new
@A-weak weakA = b
```

Attention, si on écrit :

```
@A-weak weakA = @A.new
```

L'objet créé disparaît aussitôt (il n'a pas de pointeur fort qui le référencie) : `weakA` est mis à `nil` à la fin de l'exécution de l'instruction.

40.7.3 Extraction de l'objet

Si un pointeur faible n'est pas `nil`, l'opérateur `bang` permet d'obtenir un pointeur fort sur cet objet.

```
var a = @A.new
@A-weak weakA = a
var @A b = weakA.bang
```

Si le pointeur faible est `nil`, une erreur d'exécution est déclenchée.

40.7.4 Comparaison

Les opérateurs de comparaison (`=`, `!=`, `<`, `<=`, `>` et `>=`) ne sont pas disponibles pour un pointeur faible. Les opérateurs `===` et `!==` permettent de tester si deux pointeurs faibles désignent la même instance, ou si ils sont `nil` tous les deux. La valeur `nil` est toujours différente de la valeur qui désigne une instance.

On peut donc écrire :

```
@A-weak weakA = ...
if weakA === .nil then
  message "weakA is nil\n"
else
  message "weakA is not nil\n"
end
```

Note : comme l'adresse d'allocation peut varier d'une exécution à une autre, la relation d'ordre entre les objets n'est pas stable d'une exécution à l'autre.

40.7.5 Affectation conditionnelle

Trois formes d'affection conditionnelle sont possibles.

Affectation polymorphique inverse conditionnelle vers un pointeur fort, type explicite. Dans l'exemple suivant, `weakA` est un pointeur faible de type `@A-weak` qui désigne un objet de type `@B`. La condition teste la valeur de `weakA` : si elle est non nil et désigne un objet de type `@B` (ou d'une classe héritière de `@B`), l'affectation de `b` a lieu et la condition est vraie. La constante `b` a pour type `@B`. Sinon, la condition est fausse et la constante `b` est inaccessible.

```
let b = @B.new
@A-weak weakA = b
if let b = weakA as @B then
  log b # Affiche : LOGGING b: <@B: [@B]>
else
  message "weakA is nil\n"
end
```

Affectation polymorphique inverse conditionnelle vers un pointeur fort, type implicite. Dans l'exemple suivant, `weakA` est un pointeur faible de type `@A-weak` qui désigne un objet de type `@B`. La condition teste la valeur de `weakA` : si elle est non nil et désigne un objet de type `@A` (ou d'une classe héritière, comme `@B`), l'affectation de `a` a lieu et la condition est vraie. La constante `a` a pour type `@A`. Sinon, la condition est fausse et la constante `a` est inaccessible.

```
let b = @B.new
@A-weak weakA = b
if let a = weakA then
  log a # Affiche : LOGGING a: <@A: [@B]>
else
  message "weakA is nil\n"
end
```

Affectation polymorphique inverse conditionnelle vers un pointeur faible. Dans l'exemple suivant, `weakA` est un pointeur faible de type `@A-weak` qui désigne un objet de type `@B`. La condition teste la valeur de `weakA` : si elle est non nil et désigne un objet de type `@B` (ou d'une classe héritière de `@B`), l'affectation de `b` a lieu et la condition est vraie. La constante `b` a pour type `@B-weak`. Sinon, la condition est fausse

et la constante `b` est inaccessible.

```
let b = @B.new
@A-weak weakA = b
if let b = weakA as @B-weak then
  log b # Affiche : LOGGING b: <@B-weak:instance of @B>
else
  message "weakA is nil\n"
end
```

Chapitre 41

Le type enum

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

41.1 Déclaration

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

Par exemple :

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

Plusieurs types énumérés peuvent définir des constantes de même nom.

41.2 Instanciation

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

```
@feuTricolore feu = @feuTricolore.vert
```

L'annotation de type peut être omise :

```
@feuTricolore feu = .vert
```

```
var feu = @feuTricolore.vert
```

41.3 Comparaison

Un type enuméré accepte les six opérateurs de comparaison (`==` , `!=` , `<` , `<=` , `>` et `>`). L'ordre est celui de la déclaration, c'est-à-dire que :

```
   @feuTricolore.vert < @feuTricolore.orange < @feuTricolore.rouge
```

41.4 Tester une valeur

Il y a deux façons de tester une valeur d'un type énuméré. La première consiste à comparer avec une valeur obtenue par un constructeur, par exemple :

```
   if feu == @feuTricolore.orange then ...
```

La seconde possibilité est d'appeler les *getter* implicitement déclarés : pour chaque constante, un *getter* sans argument nommé `is<Constante>` (le préfixe `is` suivi du nom de la constante, dont le premier caractère est en majuscule) est déclaré ; il renvoie une valeur de type `@bool` qui est vrai si le récepteur a la valeur correspondante :

```
   if [feu isOrange] then ...
```

41.5 L'instruction `switch`

L'instruction `switch` (section **??** page **??**) est dédiée aux types énumérés. On écrit par exemple :

```
@feuTricolore feu = ...
switch feu
case vert : message "vert"
case orange : message "orange"
case rouge : message "rouge"
end
```

41.6 Valeurs associées

Il est possible d'associer des valeurs à chaque constante, ce qui permet d'alléger dans certains cas le code à écrire. Supposons par exemple que l'on ait dans un langage une construction optionnelle :

```
rule regleProduction {
  select
  or
    $option$
    $identifier$ ?let nomOption
  end
}
```

Comment construire l'arbre syntaxique abstrait ? Il y a en fait trois possibilités.

Première solution. La première consiste à considérer la chaîne vide comme significative de l'absence d'option :

```
rule regleProduction {
  @lstring nomOption
  select
    nomOption = ["" nowhere]
  or
    $option$
    $identifier$ ?nomOption
  end
}
```

Évidemment, cette solution est acceptable uniquement si l'information associée est simple, et si une valeur particulière peut être considéré comme l'absence d'option.

Deuxième solution. La deuxième solution fait appel à trois classes :

```
abstract class @abstractOption {}

class @noOption : @abstractOption {}

class @option : @abstractOption { @lstring mOptionName }
```

La construction de l'arbre est réalisée par :

```
rule regleProduction {
  @abstractOption optionAST
  select
    optionAST = @noOption.new
  or
    $option$
    $identifier$ ?let nomOption
    optionAST = @option.new {!nomOption}
  end
}
```

Cette solution, plus générale, est plus lourde à mettre en œuvre : trois classes, et analyser l'option nécessite

d'écrire un *getter* abstrait ou une méthode abstraite pour la classe abstraite de base `@abstractOption`, et les redéfinir dans les deux classes héritières `@noOption` et `@option`.

Troisième solution. La troisième et dernière solution consiste à écrire un type énuméré possédant des valeurs associées :

```
enum @option {
  case noOption
  case optionPresente (@lstring optionName)
}
```

À la constante `optionPresente` est associée une valeur de type `@lstring`, identifiée par le nom `optionName`. Ce nom est optionnel, on pourrait écrire `optionPresente (@lstring)`. La construction de l'arbre syntaxique est maintenant réalisée par :

```
rule regleProduction {
  @option optionAST
  select
    optionAST = @option.noOption
  or
    $option$
    $identifier$ ?let nomOption
    optionAST = @option.optionPresente {!optionName:nomOption}
  end
}
```

À la constante `optionPresente` correspond un constructeur de même nom, avec un argument qui correspond à la valeur associée `@lstring optionName`. Le nom `optionName` est utilisé comme sélecteur. Si on avait déclaré la valeur associée sans nom par `optionPresente (@lstring)`, alors l'appel du constructeur serait `@option.optionPresente {!nomOption}`.

Pour tester un type énuméré avec des valeurs associées, on peut appliquer les *getter* décrits à section **??** page **??**, mais on n'a pas accès aux valeurs associées.

Les six opérateurs de comparaison (`==`, `!=`, `<`, `<=`, `>` et `>`) sont définis sur des types énumérés avec des valeurs associées : l'ordre est celui de la déclaration des constantes, et, en cas d'égalité, les valeurs associées sont comparées les unes après les autres, dans leur ordre de déclaration.

Il n'y a qu'une façon d'extraire les valeurs associées, l'instruction `switch` :

```
switch optionAST
case noOption : ...
case optionPresente (@lstring nomOption) : ...
end
```

`nomOption` est une constante dont la portée s'étend jusqu'à la fin de la branche `case` courante.

41.7 Valeur par défaut

Un type énuméré n'a pas de valeur par défaut, c'est-à-dire qu'appeler le constructeur `default` engendre une erreur de compilation.

```
@feuTricolore feu = .default # Erreur de compilation
```

À partir de la version 3.3.3 de GALGAS, il est possible de définir une valeur par défaut, en la nommant dans une clause `default` à la fin de la déclaration du type énuméré :

```
enum @feuTricolore {
  case vert
  case orange
  case rouge
  default vert
}
```

La constante nommée dans la clause `default` ne doit pas avoir de valeur associée.

On peut donc maintenant écrire :

```
@feuTricolore feu = .default # Équivalent à .vert
```

Définir une valeur par défaut pour un type enuméré permet une structure qui l'utilise de posséder à son tour une valeur par défaut (section **??** page **??**).

Chapitre 42

Le type `graph`

Le type `graph` permet de faire des opérations sur les graphes orientés.

Chaque nœud est identifié par un nom qui est une chaîne de caractères (de type `@string`); à chaque nœud sont associées les informations :

- une liste de positions dans des textes sources (liste de `@location`);
- une information utilisateur dont le type est défini par la déclaration du type graphe.

Un arc est identifié par un couple de nœuds.

Un type `graph` se déclare comme suit :

```
graph @nom_du_type_graph (@nom_liste_information) {
}
```

Le nom `@nom_du_type_graph` est le nom donné au type. Le nom `@nom_liste_information` nomme un type qui spécifie l'information utilisateur associée à chaque nœud.

Attention, le type `@nom_liste_information` est un type *liste*, et l'information utilisateur a pour type l'élement associé, c'est à dire `@nom_liste_information-element`.

Par exemple, si l'on veut manipuler des graphes dont l'information associée est un entier `@uint`, on déclarera :

```
graph @monGraphe (@uintlist) {
}
```

Si l'information associée à chaque nœud est composée d'un entier et d'une chaîne de caractères, il faut déclarer un type liste particulier :

```
list @maListe {
  @uint monInfo1
  @string monInfo2
}
graph @monGraphe (@maListe) {
}
```

42.1 Constructeur `emptyGraph`

```
constructor emptyGraph -> @self
```

`emptyGraph` est le seul constructeur d'un graphe. Il instancie un graphe vide.

```
var gr = @monGraphe.emptyGraph
```

42.2 Construire un graphe

Trois setters permettent de construire un graphe :

- un *setter d'insertion* (section **??** page **??**), défini par l'utilisateur, réalise l'ajout d'un nœud ;
- le *setter* `addEdge` (section **??** page **??**), implicitement défini, réalise l'ajout d'un arc ;
- le *setter* `noteNode` (section **??** page **??**), implicitement défini, indique qu'un nœud doit être défini.

Ces trois setters peuvent être appelés dans un ordre quelconque. Il est possible d'entrer un arc alors que ni le nœud origine, ni le nœud destination ne sont définis. Il faudra simplement qu'ils le soient avant que les calculs soient entrepris sur le graphe.

42.2.1 Setter d'insertion

Pour pouvoir entrer un nœud, il faut déclarer explicitement un *setter d'insertion* :

```
graph @monGraphe (@maListe) {
  insert addNode error message "the '%K' node is already declared at %L"
}
```

`addNode` est le nom donné au *setter* d'insertion de nœud. Comme dans un graphe, un nœud est unique, la chaîne de caractères qui suit `error message` est le message d'erreur qui est affiché en cas de tentative d'insertion d'un nœud déjà existant. Dans cette chaîne, deux séquences particulières peuvent être utilisées :

- `%K`, qui est remplacée par le nom du nœud ;

- %L, qui est remplacée par la désignation de l'endroit dans le texte source où est déclaré le nœud déjà existant.

Un setter d'insertion présente que des arguments d'entrée :

- le premier est toujours de type `@lstring` ; la composante `@string` est le nom du nœud (qui est unique dans un graphe donné), et la composante `@location` est la position dans le texte source où est déclaré le nœud, et est ajoutée à la liste correspondante du nœud ;
- Les arguments suivants correspondent en nombre et en type aux champs du type liste qui définit les informations associées.

Ainsi, le setter d'insertion `addNode` du type de graphe `@monGraphe` possède trois arguments en entrée ;

- le premier de type `@lstring` ;
- le deuxième de type `@uint` , qui correspond au premier champ `monInfo1` du type liste `@maListe` ;
- le troisième de type `@string` , qui correspond au second champ `monInfo2` du type liste `@maListe` .

Il est par exemple appelé comme suit :

```
var @lstring lstr = ...
var gr = @monGraphe.emptyGraph
[!?gr addNode !lstr !0 !"xyz"]
```

42.2.2 Entrer un arc : setter addEdge

```
setter addEdge ?@lstring inSourceNode ?@lstring inTargetNode
```

Pour entrer un arc, appeler le setter prédéfini `addEdge` . Celui-ci possède deux arguments d'entrée de type `@lstring` :

- le premier spécifie le nœud origine de l'arc ; la composante `@string` est le nom du nœud origine, et la composante `@location` est ajoutée à la liste du nœud origine ;
- le second spécifie le nœud destination de l'arc ; la composante `@string` est le nom du nœud destination, et la composante `@location` est ajoutée à la liste du nœud destination.

42.2.3 setter noteNode

```
setter noteNode ?@lstring inNode
```

Le setter prédéfini `noteNode` permet d'indiquer qu'un nœud doit être défini : il possède un seul argument en entrée, de type `@lstring` , dont la composante `@string` désigne le nom du nœud, et dont la composante `@location` est ajoutée à la liste de ce nœud.

42.3 Enlever des arcs

Deux *setters* permettent d'enlever des arcs à un graphe :

- le *setter* `removeEdgesToNode` (section **??** page **??**) retire tous les arcs qui arrivent à un nœud ;
- le *setter* `removeEdgesToDominators` (section **??** page **??**) retire tous les arcs qui arrivent à un nœud *dominateur*.

42.3.1 Setter `removeEdgesToNode`

```
setter removeEdgesToNode ?@string inTargetNode
```

Ce *setter* ne présente qu'un seul argument, une chaîne qui désigne un nœud du graphe. Son exécution supprime tous les arcs dont le nœud destination est le nœud nommé en argument.

42.3.2 Setter `removeEdgesToDominators`

```
setter removeEdgesToDominators
```

Dans un graphe, un nœud d domine un autre nœud n si chaque chemin à partir du nœud d'entrée vers le nœud n doit passer par d [1].

Ce *setter* considère que les nœuds d'entrée sont les nœuds sans prédécesseur, et retire tous les arcs d'un nœud vers son dominateur.

42.4 Getters

42.4.1 Getter edges

```
getter edges -> @2stringlist
```

Ce getter retourne la liste des arcs.

42.4.2 Getter `graphviz`

```
getter graphviz -> @string
```

Ce getter retourne une chaîne de caractères contenant une description compatible *GraphViz*[2] du graphe.

[1] http://en.wikipedia.org/wiki/Dominator_(graph_theory)
[2] http://www.graphviz.org/

42.4.3 Getter `isNodeDefined`

```
getter isNodeDefined ?@string inNodeName -> @bool
```

Ce getter retourne `true` si le nœud `inNodeName` est défini, et `false` dans le cas contraire.

42.4.4 Getter `keyList`

```
getter keyList -> @stringlist
```

Ce getter retourne la liste des noms de nœuds, aussi bien les nœuds définis (c'est à dire les nœuds pour lesquels un setter d'insertion a été appelé, section **??** page **??**), que les nœuds indéfinis.

42.4.5 Getter `lkeyList`

```
getter lkeyList -> @stringlist
```

Ce getter retourne la liste des noms de nœuds, aussi bien les nœuds définis (c'est-à-dire les nœuds pour lesquels un setter d'insertion a été appelé, section **??** page **??**), que les nœuds indéfinis. Chaque nœud défini est accompagné de sa position de définition, les nœuds indéfinis sont accompagnés d'une position non définie (équivalente à celle obtenue par le constructeur `nowhere`).

42.4.6 Getter `locationForKey`

```
getter locationForKey ?@string inNodeName -> @location
```

Ce getter retourne la localisation de la définition du nœud `inNodeName`; si il n'est pas défini, une erreur d'exécution est déclenchée et une valeur *poison* est retournée.

42.4.7 Getter `nodeList`

```
getter nodeList -> @nom_liste_information
```

Ce *getter* retourne la liste des nœuds du graphe. L'ordre est imprévisible.

42.4.8 Getter `reversedGraph`

```
getter reversedGraph -> @T
```

Ce getter retourne un graphe de même type que le receiveur, mais dont les arcs sont inversés.

42.4.9 Getter `subgraphFromNodes`

```
getter subgraphFromNodes
  ?@lstringlist inStartNodes
  ?@stringset inNodesToExclude
  -> @self
```

Ce getter retourne le graphe défini par la partie accessible du receveur à partir des nœuds nommés dans `inStartNodes`, en excluant les nœuds nommés dans `inNodesToExclude`.

42.4.10 Getter `accessibleNodesFrom`

```
getter accessibleNodesFrom
  ?@lstringlist inStartNodes
  ?@stringset inNodesToExclude
  -> @lstringlist
```

Ce getter retourne la liste des nœuds accessibles à partir des nœuds nommés dans `inStartNodes`, en excluant les nœuds cités dans `inNodesToExclude`.

42.4.11 Getter `undefinedNodeCount`

```
getter undefinedNodeCount -> @uint
```

Ce getter retourne le nombre de nœuds indéfinis (c'est à dire les nœuds pour lesquels un setter d'insertion n'a été appelé, section **??** page **??**).

42.4.12 Getter `undefinedNodeKeyList`

```
getter undefinedNodeKeyList -> @stringlist
```

Ce getter retourne la liste des noms des nœuds indéfinis (c'est à dire les nœuds pour lesquels un setter d'insertion n'a été appelé, section **??** page **??**).

42.4.13 Getter `undefinedNodeReferenceList`

```
getter undefinedNodeReferenceList -> @lstringlist
```

Ce getter retourne la liste des références des nœuds indéfinis (c'est à dire les nœuds pour lesquels un setter d'insertion n'a été appelé, section **??** page **??**). Une référence à un nœud est un `@lstring` dont la chaîne est le nom du nœud, et la composante `@location` une position dans le texte source, définie par un appel à `addEdge` (section **??** page **??**) ou `noteNode` (section **??** page **??**).

42.5 Méthodes

42.5.1 Méthode `depthFirstTopologicalSort`

```
method depthFirstTopologicalSort
  !@nom_liste_information outSortedInformationList
  !@lstringlist outSortedKeyList
  !@nom_liste_information outUnsortedInformationList
  !@lstringlist outUnsortedKeyList
```

Cette méthode effectue un tri topologique du graphe. Tous les arguments sont en sortie :

- le premier argument `outSortedInformationList` est la liste triée des informations utilisateur liées aux nœuds ;
- le deuxième `outSortedKeyList` est la liste triée des noms de nœuds ;
- le troisième `outUnsortedInformationList` est la liste des informations utilisateur liées aux nœuds qui n'ont pas pu être triés ;
- le dernier `outUnsortedKeyList` est la liste des noms de nœuds qui n'ont pas pu être triés.

Si le tri échoue, aucun message d'erreur n'est émis ; il suffit de tester le nombre d'éléments du troisième ou du quatrième argument pour savoir si le tri a réussi.

Les deux premiers arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice. Le premier élément désigne un nœud qui n'a pas de prédecesseur, et le dernier un nœud qui n'a pas de successeur.

Les deux derniers arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice. L'ordre dans lequel les nœuds non triés apparaissent n'est pas défini.

Cette méthode diffère de `topologicalSort` (section **??** page **??**) par le fait que la liste triée est présentée en privilégiant un parcours en profondeur.

42.5.2 Méthode `circularities`

```
method circularities
  !@nom_liste_information outInformationList
  !@lstringlist outKeyList
```

Cette méthode renvoie la liste de tous les nœuds impliqués dans une circularité. Les deux arguments sont en sortie :

- le premier argument `outInformationList` est la liste des informations utilisateur liées aux nœuds sans prédécesseur ;
- le second `outKeyList` est la liste des noms de nœuds sans prédécesseur.

Les deux arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice.

42.5.3 Méthode `nodesWithNoPredecessor`

```
method nodesWithNoPredecessor
  !@nom_liste_information outInformationList
  !@lstringlist outKeyList
```

Cette méthode renvoie la liste de tous les nœuds sans prédecesseur. Les deux arguments sont en sortie :

- le premier argument `outInformationList` est la liste des informations utilisateur liées aux nœuds sans prédécesseur ;
- le second `outKeyList` est la liste des noms de nœuds sans prédécesseur.

Les deux arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice.

42.5.4 Méthode `nodesWithNoSucccessor`

```
method nodesWithNoSucccessor
  !@nom_liste_information outInformationList
  !@lstringlist outKeyList
```

Cette méthode renvoie la liste de tous les nœuds sans successeur. Les deux arguments sont en sortie :

- le premier argument `outInformationList` est la liste des informations utilisateur liées aux nœuds sans successeur ;
- le second `outKeyList` est la liste des noms de nœuds sans successeur.

Les deux arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice.

42.5.5 Méthode `topologicalSort`

```
method topologicalSort
  !@nom_liste_information outSortedInformationList
  !@lstringlist outSortedKeyList
  !@nom_liste_information outUnsortedInformationList
  !@lstringlist outUnsortedKeyList
```

Cette méthode effectue un tri topologique du graphe. Tous les arguments sont en sortie :

- le premier argument `outSortedInformationList` est la liste triée des informations utilisateur liées aux nœuds ;

- le deuxième `outSortedKeyList` est la liste triée des noms de nœuds ;
- le troisième `outUnsortedInformationList` est la liste des informations utilisateur liées aux nœuds qui n'ont pas pu être triés ;
- le dernier `outUnsortedKeyList` est la liste des noms de nœuds qui n'ont pas pu être triés.

Si le tri échoue, aucun message d'erreur n'est émis ; il suffit de tester le nombre d'éléments du troisième ou du quatrième argument pour savoir si le tri a réussi.

Les deux premiers arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice. Le premier élément désigne un nœud qui n'a pas de prédecesseur, et le dernier un nœud qui n'a pas de successeur.

Les deux derniers arguments renvoyés ont le même nombre d'éléments, et correspondent indice par indice. L'ordre dans lequel les nœuds non triés apparaissent n'est pas défini.

Cette méthode diffère de `depthFirstTopologicalSort` (section **??** page **??**) par le fait que l'ordre topologique n'est pas défini.

Chapitre 43

Le type `map`

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

43.1 Déclaration

La déclaration d'un type `map` nomme :

- des attributs associés au type table (section **??** page **??**) ;
- les *setters* d'insertion (section **??** page **??**) ;
- les *méthodes* de recherche (section **??** page **??**) ;
- les *setters* de retrait (section **??** page **??**).

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

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"
}
```

43.2 Constructeurs

Pour initialiser une table vide, il y a trois possibilités, sémantiquement identiques :

- la constante {} (section **??** page **??**) ;
- le constructeur `emptyMap` (section **??** page **??**) ;
- le constructeur `default` (section **??** page **??**).

43.2.1 Constante {}

Cette constante permet d'instancier une table vide. Exemple :

```
@MaTable uneTable = {}
```

Ou encore :

```
var uneTable = @MaTable {}
```

43.2.2 Constructeur `emptyMap`

Pour instancier une table vide, une autre possibilité est d'appeler le constructeur `emptyMap`. Exemple :

```
@MaTable uneTable = .emptyMap
```

Ou encore :

```
var uneTable = @MaTable.emptyMap
```

43.2.3 Constructeur `default`

Une table accepte le constructeur `default`. Exemple :

```
@MaTable uneTable = .default
```

Ou encore :

```
var uneTable = @MaTable.default
```

43.2.4 Constructeur `mapWithMapToOverride`

```
constructor mapWithMapToOverride ?@T inMapToOverride -> @T
```

Ce constructeur permet d'instancier une table vide, qui surcharge la table `inMapToOverride` citée en argument. Exemple :

```
@MaTable uneTable = .emptyMap
@MaTable autreTableTable = .mapWithMapToOverride {!uneTable}
```

43.3 Setters d'insertion

Une `map` peut déclarer zéro, un ou plusieurs *setters* d'insertion. Un *setter* d'insertion permet d'insérer une nouvelle entrée à une table. Une erreur est déclenchée en cas de tentative d'une clé déjà existante.

Un *setter* d'insertion est déclaré par :

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

L'identificateur `nom` donne un nom au *setter* d'insertion ; ce nom doit être unique parmi les *setters* d'insertion et de retrait. La chaîne de caractères `"message_erreur"` définit le message d'erreur qui est affiché en cas de tentative d'une clé déjà existante. Cette chaîne accepte deux séquences d'échappement :

- `%K`, qui est remplacée par la chaîne de caractères de la clé existante ;
- `%L`, qui est remplacée par la chaîne décrivant la position de la clé existante dans les fichiers source.

Un *setter* d'insertion est appelé dans une *instruction d'appel de setter*, comprenant tous ses arguments en sortie :

- le premier argument est une expression de type `@lstring` qui caractérise la clé à insérer ;
- ensuite, pour chaque attribut déclaré, une expression du type de cet attribut.

Par exemple :

```
@MaTable uneTable = {}
@lstring clef = ...
@string s = ...
@uint v = ...
[!?uneTable insertKey !clef !s !v]
```

43.4 Méthodes de recherche

Une `map` peut déclarer zéro, une ou plusieurs *méthodes* de recherche. Une *méthode* de recherche permet de rechercher une entrée d'une table, et retourne la valeur de ses attributs associés. Une erreur est déclenchée si la clé n'existe pas.

Une *méthode* de recherche est déclarée par :

```
search nom error message "message_erreur"
```

L'identificateur `nom` donne un nom à la *méthode* de recherche ; ce nom doit être unique parmi ces *méthodes*. La chaîne de caractères `"message_erreur"` définit le message d'erreur qui est affiché en cas de recherche d'une clé inexistante. Cette chaîne accepte une séquence d'échappement :

- %K, qui est remplacée par la chaîne de caractères de la clé inexistante recherchée.

Une *méthode* de recherche est appelée dans une *instruction d'appel de méthode* :

- le premier argument (sortie) est une expression de type `@lstring` qui caractérise la clé à rechercher ;
- ensuite, pour chaque attribut déclaré, un argument en entrée nommant une variable destinée à recevoir la valeur de l'attribut correspondant.

Par exemple :

```
@MaTable uneTable = {}
...
@lstring clef = ...
[uneTable searchKey !clef ?@string s ?@uint v]
```

43.5 Setters de retrait

Une `map` peut déclarer zéro, un ou plusieurs *setters* de retrait. Un *setter* de recherche permet de retirer une entrée d'une table, et retourne la valeur des attributs de la clé retirée. Une erreur est déclenchée si la clé n'existe pas.

Un *setter* de retrait est déclaré par :

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

L'identificateur `nom` donne un nom au *setter* de retrait ; ce nom doit être unique parmi les *setters* d'insertion et de retrait. La chaîne de caractères `"message_erreur"` définit le message d'erreur qui est affiché en cas de recherche d'une clé inexistante. Cette chaîne accepte une séquence d'échappement :

- %K, qui est remplacée par la chaîne de caractères de la clé inexistante à retirer.

Un *setter* de retrait est appelé dans une *instruction d'appel de setter* :

- le premier argument (sortie) est une expression de type `@lstring` qui caractérise la clé à retirer ;
- ensuite, pour chaque attribut déclaré, un argument en entrée nommant une variable destinée à recevoir la valeur de l'attribut correspondant de la clé retirée.

Par exemple :

```
@MaTable uneTable = {}
...
@lstring clef = ...
[!?uneTable removeKey !clef ?@string s ?@uint v]
```

43.6 Getters

43.6.1 Getter count

```
getter count -> @uint
```

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

43.6.2 Getter hasKey

```
getter hasKey ?@string inKey -> @bool
```

Le *getter* `hasKey` retourne un `@bool` qui est `true` si la clé `inKey` est dans la table de premier niveau du récepteur, `false` dans le cas contraire.

43.6.3 Getter keyList

```
getter keyList -> @lstringlist
```

Le *getter* `keyList` retourne la liste construite avec toutes les clés de la table de premier niveau du récepteur. L'ordre de la liste est l'ordre alphabétique croissant des clés.

43.6.4 Getter keySet

```
getter keySet -> @stringset
```

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

43.6.5 Getter locationForKey

```
getter locationForKey ?@string inKey -> @location
```

Le *getter* `locationForKey` retourne un `@location` qui contient l'information de position de la clé `inKey` dans la table de premier niveau du récepteur. Une erreur d'exécution est déclenchée si cette clé n'existe pas.

43.6.6 Getter `overriddenMap`

```
getter overriddenMap -> @T
```

Le *getter* `overriddenMap` retourne la table obtenue en amputant de la valeur du récepteur la table de premier niveau. Si le récepteur n'a pas de table surchargée, une erreur d'exécution est déclenchée.

43.7 Énumération

L'instruction `for` permet d'énumérer des objets de type `map` ; elle est décrite à la section **??** page **??**.

Chapitre 44

Le type `dict`

Un objet de type `dict` est un dictionnaire. Contrairement à un objet de type `map`, le type de la clé est déclaré explicitement et peut être d'un type quelconque.

Attention ! Le type de la clé d'un dictionnaire peut être un `@lstring`, comme la clé implicite d'une `map`. Toutefois, la sémantique est différente. Dans une `map`, c'est la composante `string` de la clé qui est prise en compte comme clé effective. La composante `location` n'est utilisée que pour le signalement d'erreur. Dans un `dict`, le type déclaré comme clé est intégralement pris en compte : si la clé d'un `dict` est un `@lstring`, la comparaison des clés prend en compte la composante `string` **et** la composante `location`.

44.1 Déclaration

```
dict @MonDictionnaire : @uint {
  @string mPremier
  @bool mSecond
}
```

La déclaration d'un type `dict` nomme :

- le nom du type `dict` (ici `@MonDictionnaire`) ;
- le type de la clé (ici `@uint`) ;
- les propriétés associées (ici `mPremier` et `mSecond`).

Les noms `key`, `object` et `desciption` sont interdits pour une propriété.

On peut déclarer un dictionnaire sans propriété, et ainsi gérer des *ensembles* :

```
dict @autreDictionnaire : @uint { }
```

44.2 Constructeurs

Pour initialiser un dictionnaire vide, il y a trois possibilités, sémantiquement identiques :

- la constante {} (section **??** page **??**);
- le constructeur `emptyDict` (section **??** page **??**);
- le constructeur `default` (section **??** page **??**).

44.2.1 Constante {}

Cette constante permet d'instancier un dictionnaire vide. Exemple :

```
@MonDictionnaire unDictionnaire = {}
```

Ou encore :

```
var unDictionnaire = @MonDictionnaire {}
```

44.2.2 Constructeur `emptyDict`

Pour instancier un dictionnaire vide, une autre possibilité est d'appeler le constructeur `emptyDict`. Exemple :

```
@MonDictionnaire unDictionnaire = .emptyDict
```

Ou encore :

```
var unDictionnaire = @MonDictionnaire.emptyDict
```

44.2.3 Constructeur `default`

Un dictionnaire accepte le constructeur `default`. Exemple :

```
@MonDictionnaire unDictionnaire = .default
```

Ou encore :

```
var unDictionnaire = @MonDictionnaire.default
```

44.3 Insertion

Un `dict` implémente implicitement l'opérateur `+=` qui permet d'insérer une nouvelle entrée à un dictionnaire. Ses arguments sont, dans l'ordre : un objet du type de la clé, puis un objet par propriété déclarée. Aucune erreur n'est déclenchée en cas de tentative d'insertion d'une clé déjà existante ; les valeurs des propriétés associées à la clé sont simplement remplacées.

Par exemple :

```
@MonDictionnaire unDictionnaire = {}
@uint clef = ...
@string s = ...
@uint v = ...
unDictionnaire += !clef !s !v
```

44.4 Recherche

Une `dict` déclare implicitement la *méthode* de recherche `searchKey` qui permet de rechercher une entrée d'un dictionnaire, et retourne la valeur de ses propriétés associées. Une erreur est déclenchée si la clé n'existe pas.

La *méthode* de recherche `searchKey` est appelée dans une *instruction d'appel de méthode* :

- le premier argument (sortie) est une expression de type de la clé qui caractérise la clé à rechercher ;
- ensuite, pour chaque propriété déclarée, un argument en entrée nommant une variable destinée à recevoir la valeur de la propriété correspondante.

Par exemple :

```
@MonDictionnaire unDictionnaire = {}
...
@lstring clef = ...
[unDictionnaire searchKey !clef ?@string s ?@uint v]
```

44.5 Retrait

Un `dict` déclare implicitement le *setter* de retrait `removeKey`. Il permet de retirer une entrée d'un dictionnaire, et retourne la valeur des propriétés associées à la clé retirée. Une erreur est déclenchée si la clé n'existe pas.

Le *setter* de retrait `removeKey` est appelé dans une *instruction d'appel de setter* :

- le premier argument (sortie) est une expression de type de la clé qui caractérise la clé à retirer ;

- ensuite, pour chaque propriété déclarée, un argument en entrée nommant une variable destinée à recevoir la valeur de la propriété correspondant de la clé retirée.

Par exemple :

```
@MonDictionnaire unDictionnaire = {}
...
@lstring clef = ...
[!?unDictionnaire removeKey !clef ?@string s ?@uint v]
```

44.6 Getters

44.6.1 Getter count

```
getter count -> @uint
```

Le *getter* `count` retourne un `@uint` qui contient le nombre d'entrées du dictionnaire.

44.6.2 Getter hasKey

Le *getter* `hasKey` retourne un `@bool` qui est `true` si la clé `inKey` est dans le dictionnaire, `false` dans le cas contraire.

44.7 Énumération

L'instruction `for` permet d'énumérer des objets de type `dict` ; elle est décrite à la section **??** page **??**.

Chapitre 45

Le type structure

Le mot-clé `struct` permet de définir des types de structure. Un objet de type structure a une sémantique de valeur.

La syntaxe de définition d'un type structure est de la forme :

```
struct @MaStructure {
  # Liste de déclaration de propriétés, par exemple :
  @uint mProp1
  @bool mProp2
}
```

Il n'est pas possible de définir du code dans une déclaration de structure : la seule possibilité est de le définir dans des `extension` (chapitre **??** à partir de la page **??**).

45.1 Constructeurs

45.1.1 Constructeur new

Tout type structure définit implicitement le constructeur `new`. Son appel comprend une valeur par propriété déclarée par le type structure.

Par exemple, pour la déclaration :

```
struct @maStructure {
  @uint mProp1
  @bool mProp2
}
```

L'appel du constructeur `new` est :

```
var aVariable = @maStructure.new {!123 !true}
```

Si le contexte le permet, l'annotation de type peut être omis lors de l'appel du constructeur :

```
@maStructure aVariable = .new {!123 !true}
```

À partir de la version 3.3.8, il est possible d'ajouter l'attribut `%selector` à la déclaration d'une propriété de structure. Le faire impose d'utiliser le sélecteur portant le nom de la propriété dans l'appel du constructeur `new`. Par exemple, si on déclare :

```
struct @maStructure {
  @uint mProp1 %selector
  @bool mProp2
}
```

Alors l'appel du constructeur `new` devient :

```
var aVariable = @maStructure.new {!mProp1:123 !true}
```

45.1.2 Constructeur `default`

Si chacune des propriétés accepte le constructeur par défaut, alors le type structure accepte le constructeur pas défaut. C'est le cas de la structure `@maStructure` définie au dessus : `@uint` accepte le constructeur par défaut (initialisation à `0`), ainsi que `@bool` (initialisation à `false`). Donc :

```
var aVariable = @maStructure.default
```

Initialise les propriétés de `aVariable` respectivement à `0` et `false`. On peut aussi écrire :

```
@maStructure aVariable = .default
```

45.2 Accès aux propriétés

La notation pointée `variable.propriété` permet d'accéder à une propriété d'une structure, aussi bien en lecture, en écriture et en lecture/écriture.

Exemple d'accès en lecture :

```
@uint v = aVariable.mProp1
```

Exemple d'accès en écriture :

```
aVariable.mProp1 = 10
```

Exemple d'accès en lecture/écriture :

```
aVariable.mProp1 ++
```

45.3 Getters

Un type structure définit un *getter* sans argument par propriété, qui permet d'accéder en lecture à cette propriété. Son nom est celui de la propriété. Par exemple, à la place de :

```
@uint v = aVariable.mProp1
```

On peut écrire :

```
@uint v = [aVariable mProp1]
```

45.4 Types structure prédéfinis

Plusieurs types préféfinis GALGAS sont des structures.

45.4.1 Le type `@lbigint`

```
struct @lbigint {
  @bigint bigint
  @location location
}
```

45.4.2 Le type `@lbool`

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

45.4.3 Le type `@lchar`

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

45.4.4 Le type `@ldouble`

```
struct @ldouble {
  @double double
```

```
  @location location
}
```

45.4.5 Le type @lsint

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

45.4.6 Le type @lsint64

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

45.4.7 Le type @lstring

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

45.4.8 Le type @luint

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

45.4.9 Le type @luint64

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

45.4.10 Le type @range

Le type `@range` définit les intervalles d'entiers non signés 32 bits (`@uint`).

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

La plupart des propriétés du type `@range` découle de cette définition (chapitre **??** à partir de la page **??**).

`@range.new {!a !b}` , où `a` et `b` sont des expressions de type `@uint` , représente :

- un intervalle vide si `b` est égal à zéro ;
- l'intervalle $[a, a + b - 1]$ si `b` est strictement positif.

45.4.10.1 Opérateurs ... et ..<

Deux opérateurs permettent de construire plus facilement des objets de type `@range` .

L'opérateur `...` permet de définir un intervalle fermé à partir de sa borne inférieure et de sa borne supérieure : si `a` et `b` sont des expressions de type `@uint` , l'expression `a ... b` est équivalente à la construction `@range.new {!a !b - a + 1}` . Une exception est levée si $b < a$.

L'opérateur `..<` permet de définir un intervalle ouvert à gauche à partir de sa borne inférieure et de sa borne supérieure : si `a` et `b` sont des expressions de type `@uint` , l'expression `a ..< b` est équivalente à `@range.new {!a !b - a}` . Une exception est levée si $b < a$.

45.4.10.2 Type @range et instruction for

On peut utiliser une expression de type `@range` avec l'instruction `for` :

```
for i in @range.new {!12 !5} do
  # i prend successivement les valeurs 12, 13, 14, 15, 16
end
```

Et, avec l'opérateur `...` :

```
for i in 12 ... 16 do
  # i prend successivement les valeurs 12, 13, 14, 15, 16
end
```

Et l'opérateur `..<` :

```
for i in 12 ..< 17 do
```

```
  # i prend successivement les valeurs 12, 13, 14, 15, 16
end
```

Si l'on veut parcourir l'énumération à partir de la dernière valeur, on utilise le modificateur `>` après le mot-clé `for` :

```
for > i in @range.new {!12 !5} do
  # i prend successivement les valeurs 16, 15, 14, 13, 12
end
```

```
for > i in 12 ... 16 do
  # i prend successivement les valeurs 16, 15, 14, 13, 12
end
```

```
for > i in 12 ..< 17 do
  # i prend successivement les valeurs 16, 15, 14, 13, 12
end
```

Chapitre 46

Le type `extern`

Un type `extern` est déclaré et spécifié en GALGAS, et implémenté par une classe C++. Ceci permet de définir des types qui seraient difficilement exprimables en GALGAS.

On va voir sur un exemple comment déclarer et implémenter :

- un type externe minimum ;
- un constructeur ;
- un *setter* ;
- une *méthode* ;
- un *getter* ;
- une *méthode* de classe.

L'exemple consiste à implémenter le type `@complex` qui représente les nombres complexes.

46.1 Type externe minimum

L'implémentation minimum ne sera pas opérationnelle, car elle ne comprendra pas de constructeur : on ne pourra donc pas instancier d'objet du type `@complex`. L'ajout de constructeur sera présenté à la section suivante. De même, cette implémentation minimum ne définira ni *setter*, ni *méthode*, ni *getter*.

46.1.1 Déclaration en GALGAS

La description minimum est la suivante :

```
extern @complex {
  "// No Predeclaration\n"
}{
  "  private : bool mIsValid ;\n"
  "  private : double mReal ;\n"
  "  private : double mImaginary ;\n"
}{
}
```

Cette description est divisée en trois parties, délimitées par les accolades `{` et `}`.

Première partie. Elle cite une séquence de chaînes de caractères, qui seront écrites telles quelles dans le fichier d'en-tête C++ engendré, juste avant la déclaration de la classe C++ ; on peut y placer là des pré-déclarations de classe, des inclusions de fichier, ... Pour le type `@complex`, aucune pré-déclaration n'est nécessaire, aussi on place un simple commentaire C++, de façon à le localiser dans le fichier d'en-tête C++ engendré.

46.1.2 Implémentation en C++

46.2 Constructeur

46.3 Setter

46.4 Méthode

46.5 Getter

46.6 Méthode de classe

Chapitre 47

Compléter le système de types

47.1 Ajouter une méthode , un *getter*, un *setter* ou un constructeur à un type prédéfini

Ajouter une méthode, un *getter*, un *setter* ou un constructeur à un type prédéfini s'effectue en quatre temps :

1. ajouter la méthode, le *getter*, le *setter* ou le constructeur dans GALGAS ;
2. reconstruire le fichier d'en-tête des types prédéfinis ;
3. implémenter la méthode, le *getter*, le *setter* ou le constructeur en C++ ;
4. mettre à jour la documentation LaTeX.

À titre d'exemple, nous allons montrer comment la méthode `makeDirectoryAndWriteToExecutableFile` de la classe `@string` a été ajoutée.

47.1.1 Ajouter la méthode dans GALGAS

Éditez le fichier GALGAS, en fonction du tableau **??**. Comme c'est une méthode que nous voulons ajouter, on édite le fichier `galgas-sources/semanticsInstanceMethods.galgas`.

Dans ce fichier, il y a une méthode pour chaque type prédéfini. Pour la classe `@string`, on a :

Opération	**Fichier**
Ajouter un constructeur	`galgas-sources/semanticsConstructors.galgas`
Ajouter un *getter*	`galgas-sources/semanticsGetters.galgas`
Ajouter un *setter*	`galgas-sources/semanticsSetters.galgas`
Ajouter une méthode	`galgas-sources/semanticsInstanceMethods.galgas`
Ajouter une méthode de type	`galgas-sources/semanticsTypeMethods.galgas`

Tableau 47.1 – *Fichier GALGAS à éditer pour compléter un type prédéfini*

```
override method @stringPredefinedTypeAST getInstanceMethodMap
  ?!@unifiedTypeMap ioUnifiedTypeMap
  !@instanceMethodMap outInstanceMethodMap
{
  outInstanceMethodMap = {}
  enterInstanceMethodWithInputArgument (
    !?outInstanceMethodMap
    !?ioUnifiedTypeMap
    !inputArgTypeName:"string"
    !inputArgName:"inFilePath"
    !methodName:"writeToFile"
    !true
  )
  ...
```

Pour ajouter la méthode `makeDirectoryAndWriteToExecutableFile` de la classe `@string`, on complète cette méthode par :

```
override method @stringPredefinedTypeAST getInstanceMethodMap
  ?!@unifiedTypeMap ioUnifiedTypeMap
  !@instanceMethodMap outInstanceMethodMap
{
  outInstanceMethodMap = {}
  ...
  enterInstanceMethodWithInputArgument (
    !?outInstanceMethodMap
    !?ioUnifiedTypeMap
    !inputArgTypeName:"string"
    !inputArgName:"inFilePath"
    !methodName:"makeDirectoryAndWriteToExecutableFile"
    !true
  )
}
```

47.1.2 Reconstruire le fichier d'en-tête des types prédéfinis

Le fichier `libpm/galgas2/predefined-types.h` contient la déclaration C++ de tous les types prédéfinis. Le fichier `libpm/galgas2/predefined-types.cpp` contient l'implémentation des constructions génériques des types prédéfinis. **Surtout n'éditez pas ces fichiers à la main !** On va utiliser GALGAS pour les reconstruire. Pour cela, appeler le script Shell `libpm/galgas2/-build-builtin-type-headers.command`. L'exécution de celui-ci recompile GALGAS, et engendre les nouvelles versions des fichiers `predefined-types.h` et `predefined-types.cpp`. Voici ce que l'on obtient :

```
Native Compiling for Mac OS X (debug): all-declarations-26.cpp
...
Native Compiling for Mac OS X (debug): check-gmp.cpp
Native Linking for Mac OS X (debug): galgas-debug
Done at +12s
Replaced '/Volumes/dev-svn/galgas/libpm/galgas2/predefined-types.h'.
No warning, no error.
[Displayed from file 'all-declarations-19.cpp' at line 952]
11800 memory blocks, 4158 arraies have been used.
7052 POD arraies have been used, 706 have been reallocated (509 with pointer change).
```

Ici, seul le fichier `predefined-types.h` a été modifié, le fichier `predefined-types.cpp` ne nécessitait pas de modification.

47.1.3 Implémenter la méthode en C++

Maintenant, effectuer la compilation C++ du projet GALGAS, soit avec Xcode, soit avec le *makefile* natif de votre choix. **Il est normal que cette compilation échoue, la méthode n'a pas encore été implémentée.**

Éditez le fichier `libpm/galgas2/GALGAS_string.cpp` et ajouter la méthode :

```
void GALGAS_string::
method_makeDirectoryAndWriteToExecutableFile (GALGAS_string inFilePath,
                                               C_Compiler * inCompiler
                                               COMMA_LOCATION_ARGS) const {
  if (isValid () && inFilePath.isValid ()) {
  //--- Make directory
    const C_String directory = inFilePath.mString.stringByDeletingLastPathComponent () ;
    bool ok = C_FileManager::makeDirectoryIfDoesNotExist (directory) ;
    if (! ok) {
      C_String message ;
      message << "cannot␣create␣'" << directory << "'␣directory" ;
      inCompiler->onTheFlyRunTimeError (message COMMA_THERE) ;
    }else{
      method_writeToExecutableFile (inFilePath, inCompiler COMMA_THERE) ;
    }
  }
```

```
}
```

Maintenant la compilation C++ de GALGAS s'effectue correctement. Mais ce n'est pas terminé !

47.1.4 Finaliser le nouveau compilateur GALGAS

L'exécutable GALGAS embarque le source de la libraire `libpm`. Or, à ce stade, c'est l'ancienne version qui est empbarquée. Lorsque l'on compile le projet `+galgas.galgasProject`, la librairie `libpm` est intégrée dans les sources C++ engendrés.

Il faut donc effectuer itérativement des cycles *compilation GALGAS – compilation C++* tant que la compilation GALGAS apporte des modifications du code C++ engendré.

IV

Sous-programmes

Chapitre 48

Sous-programmes

GALGAS définit les sous-programmes suivants :

- les *fonctions* (dans ce chapitre, section **??** page **??**) ;
- les *procédures* (dans ce chapitre, section **??** page **??**) ;
- les *méthodes* (section **??** page **??**) ;
- les *getters* (section **??** page **??**) ;
- les *setters* (section **??** page **??**).

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

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

48.1 Arguments formels et paramètres effectifs

GALGAS distingue trois sortes d'arguments formels :

- en entrée (section **??** page **??**) ;
- en entrée/sortie (section **??** page **??**) ;
- en sortie (section **??** page **??**).

Sous-programme	Construction	Référence
routine	Instruction d'appel de routine	section **??** page **??**
fonction	Appel de fonction (dans une expression)	section **??** page **??**
méthode	Instruction d'appel de méthode	section **??** page **??**
getter	Appel de getter (dans une expression)	section **??** page **??**
setter	Instruction d'appel de setter	section **??** page **??**

Tableau 48.1 – *Constructions d'appel de sous programme*

Argument formel en entrée	Remarque	Paramètre effectif en sortie
`?selector:@T variable`	Variable (modifiable)	`!selector:expression`
`?selector:@T unused variable`	Variable inutilisée	
`?selector:let @T constante`	Constante	
`?selector:let @T unused constante`	Constante inutilisée	

Tableau 48.2 – *Argument formel en entrée, paramètre effectif en sortie*

48.1.1 Argument formel en entrée

Le tableau **??** liste les différentes formes d'un argument formel en entrée. Le paramètre effectif correspondant est une expression précédée par `!`.

48.1.2 Argument formel en entrée/sortie

Le tableau **??** liste les différentes formes d'un argument formel en entrée. Le paramètre effectif correspondant est une *cible* précédée par `!?`. Une *cible* est soit une variable, soit l'accès à un champ d'une variable de type `struct`.

48.1.3 Argument formel en sortie

Le tableau **??** liste l'unique forme d'un argument formel en sortie. Le compilateur vérifie que les instructions du sous-programme fixent une valeur à chaque argument formel en sortie.Le paramètre effectif correspondant est une *cible* précédée par `?`. Une *cible* est soit une variable, soit l'accès à un champ d'une variable de type `struct`.

Argument formel en entrée/sortie	Paramètre effectif en sortie/entrée	Remarque
`?!selector:@T variable`	`!?selector:cible`	
`?!selector:@T unused variable`	`!?selector:*`	Variable anonyme
	`!?n*`	n variables anonymes

Tableau 48.3 – *Argument formel en entrée/sortie, paramètre effectif en sortie/entrée*

Argument formel en sortie	Paramètre effectif en entrée	Remarque
`!selector:@T variable`	`?selector:variable`	Affectation
	`?selector:@T variable`	Déclaration et affectation
	`?selector:let @T constante`	Déclaration et affectation
	`?selector:let @T unused constante`	Déclaration et affectation
	`?selector:let constante`	Déclaration et affectation
	`?selector:let unused constante`	Déclaration et affectation
	`?selector:*`	Une variable anonyme
	`?n*`	n variables anonymes

Tableau 48.4 – *Argument formel en sortie, paramètre effectif en entrée*

48.2 Liste d'arguments formels en entrée, en sortie, ou en entrée/-sortie

48.3 Liste de paramètres effectifs en entrée

48.4 Sélecteur

Il est possible d'associer un nom avec chaque symbole `?`, `?!`, `!` et `!?`. Ce nom est appelé *sélecteur*.

Un sélecteur commence par une lettre et est suivi par zéro, un ou plusieurs lettres ou chiffres, et termine obligatoirement par deux points `:`. Aucun espace n'est autorisé. Par exemple :

`?valeur:`

`!par2:`

Les sélecteurs peuvent être utilisés à chaque fois que le symbole `?`, `?!`, `!` et `!?` apparaît. Quand un argument formel est déclaré avec un sélecteur, alors le paramètre effectif doit nommer le même sélecteur.

Par exemple, si on considère une routine déclarée par :

```
proc aireRectangle
  ?longueur: @uint inA
  ?largeur: @uint inA
  !aire: @uint outAire
{
```

```
   ...
}
```

Alors son appel s'exprimera par :

```
aireRectangle (!longueur:2 !largeur:3 ?aire:let aire)
```

Chapitre 49

Fonctions et procédures

GALGAS définit les sous-programmes suivants :

- les *fonctions* (dans ce chapitre, section **??** page **??**) ;
- les *procédures* (dans ce chapitre, section **??** page **??**) ;
- les *méthodes* (section **??** page **??**) ;
- les *getters* (section **??** page **??**) ;
- les *setters* (section **??** page **??**).

Une *fonction* n'accepte que des arguments en entrée, et retourne une valeur. Elle est appelée dans une expression (section **??** page **??**).

Une *procédure* accepte des arguments en entrée, en sortie, en entrée/sortie. Elle est appelée dans une instruction (section **??** page **??**).

Une *méthode* accepte des arguments en entrée, en sortie, en entrée/sortie, et nomme un objet courant, qui est inchangé par l'exécution de la méthode. Une *méthode* est appelée dans une instruction (section **??** page **??**).

Un *getter* n'accepte que des arguments en entrée, retourne une valeur, et nomme un objet courant, qui est inchangé par l'exécution du getter. Il est appelé dans une expression (section **??** page **??**).

Un *setter* accepte des arguments en entrée, en sortie, en entrée/sortie, et nomme un objet courant, qui est modifié par l'exécution du setter. Un *setter* est appelé dans une instruction (section **??** page **??**).

49.1 Fonction

Une fonction GALGAS n'accepte que des arguments en entrée, et retourne une valeur. Elle est appelée dans une expression (section **??** page **??**).

49.1.1 déclaration d'une fonction

```
private # Optionnel
func nom_fonction liste_arguments_entree -> @T var_resultat {
  liste_instructions
}
```

Une fonction est désignée par `nom_fonction`. Ce nom est unique dans un projet GALGAS. La liste des paramètres d'entrée peut être vide (section **??** page **??**). La valeur renvoyée par l'exécution de la fonction est la valeur de `var_resultat` à l'issue de l'exécution de la `liste_instructions`. Aussi, l'exécution de la `liste_instructions` doit valuer `var_resultat`.

Exemple :

```
func produit ?@uint a ?@uint b -> @uint resultat {
  resultat = a * b
}
```

Mentionner la variable `resultat` est optionnel. Par défaut, en son absence, une variable nommée `result` est implictement déclarée :

```
func produit ?@uint a ?@uint b -> @uint {
  result = a * b
}
```

49.1.2 Fonction interne à un fichier

En préfixant la déclaration d'une fonction par `private`, on limite son appel aux expressions situées dans le même fichier que la déclaration.

49.1.3 Attribut %once

Une fonction sans argument accepte l'attribut `%once` :

```
func %once masque -> @uint {
  result = 1 << 16
}
```

L'attribut `%once` organise le cache du résultat : celui-ci est calculé lors du premier appel, est mémorisé internement, et est retourné directement lors des appels ultérieurs.

Une fonction `%once` peut être déclarée interne en la préfixant par `private`.

```
private func %once masque -> @uint {
  result = 1 << 16
}
```

49.1.4 Attribut `%usefull`

Une fonction accepte l'attribut `%usefull` :

```
func %usefull carré ?let @uint inX -> @uint {
  result = inX * inX
}
```

L'attribut `%usefull` déclare la fonction comme utile, c'est-à-dire qu'elle est considérée comme une construction racine du graphe d'utilité calculé par l'option `--check-usefulness` (chapitre **??** à partir de la page **??**). Ceci est nécessaire si la fonction n'est pas appelée directement, mais par l'intermédiaire d'un objet de type `@function` (page **??**).

Attributs `%once` et `%usefull` sont cumulables :

```
private func %once %usefull masque -> @uint {
  result = 1 << 16
}
```

49.2 Procédure

Une procédure GALGAS accepte des arguments en entrée, en sortie, en entrée/sortie. Elle est appelée dans une instruction (section **??** page **??**).

49.2.1 Déclaration d'une procédure

```
private # Optionnel
proc nom_procedure liste_arguments {
  liste_instructions
}
```

Une procédure est désignée par `nom_procedure`. Ce nom est unique dans un projet GALGAS. La liste des paramètres en entrée, en sortie ou en entrée/sortie est décrite à la section **??** page **??**).

Exemple :

```
proc produit ?@uint a ?@uint b !@uint resultat {
  resultat = a * b
```

```
}
```

49.2.2 Procédure interne à un fichier

En préfixant la déclaration d'une procédure par `private`, on limite son appel aux instructions situées dans le même fichier que la déclaration.

49.3 Fonction et routine externe

Il est possible de définir des fonctions ou des procédures *externes* à GALGAS, c'est-à-dire déclarées et appelables dans le code GALGAS, et définies en C++. Ceci permet d'écrire du code qui serait difficile ou impossible à écrire directement en GALGAS. Une autre possibilité est d'écrire un *type externe* (chapitre **??** à partir de la page **??**).

La définition d'une telle fonction ou procédure s'effectue en trois étapes :

- déclaration comme fonction ou procédure externe ;
- préparation du fichier C++ qui contiendra l'implémentation de la fonction ;
- implémentater la fonction C++.

49.3.1 Déclaration d'une fonction ou d'une procédure externe

Il suffit d'écrire l'en-tête de la fonction, précédée du mot réservé `extern`. Par exemple :

```
extern func maFonctionExterne ?@uint a ?@uint b -> @uint
```

Comme pour les fonctions GALGAS (section **??** page **??**), une fonction externe n'accepte que des arguments en entrée.

Il en est de même pour une procédure :

```
extern proc maProcédureExterne !@uint a ?!@uint b ?@uint64 c
```

Cette déclaration peut apparaître dans n'importe quel fichier d'extension `.galgas` : la portée de la déclaration est globale à tout le projet, donc une fonction ou une procédure externe peut être appelée à partir de n'importe quel fichier d'extension `.galgas` du projet.

Donc :

- si vous déclarez une fonction ou une procédure externe, sans jamais l'appeler en GALGAS, la compilation GALGAS puis la compilation C++ des codes engendrés s'effectuent sans erreur ;
- si vous déclarez une fonction ou une procédure externe et qu'elle est appelée à partir du code GALGAS, la compilation GALGAS s'effectue correctement, par contre l'édition de liens C++ indique une erreur de type *symbole indéfini*, ce qui est logique, la fonction étant en effet indéfinie.

49.3.2 Ajout d'un fichier source C++ au projet GALGAS

Créez un fichier C++ vide, et placez le dans un répertoire situé à la racine de votre pprojet GALGAS, c'est-à-dire dans le même répertoire dans le fichier projet GALGAS : par exemple `monRepertoire/monCode.cpp`.

Il faut ajouter dans le fichier projet GALGAS la prise en compte de ce fichier C++. Pour cela, éditez votre fichier d'extension `.galgasProject` et insérez la ligne commençant par `%tool-source` :

```
project (...) -> "..." {
  ...
  %tool-source : "monRepertoire/monCode.cpp"
  ...
}
```

La déclaration `%tool-source` indique le fichier C++ fait partie de la liste des fichiers C++ à compiler. Le chemin du fichier source C++ peut être soit un chemin absolu, soit, comme c'est le cas ici, un chemin relatif par rapport au répertoire qui contient le fichier projet GALGAS (d'extension `.galgasProject`).

Dans un fichier projet GALGAS, zéro, une ou plusieurs déclarations `%tool-source` peuvent apparaître.

À ce point d'avancement, le fichier C++ est vide, il peut être compilé. Si vous lancez une compilation GALGAS suivie d'une compilation C++, vous verrez apparaître `monCode.cpp` dans la liste des fichiers C++ compilés. Évidement, si il y a des fonctions ou procédures GALGAS externes non définies, vous aurez des erreurs d'édition des liens.

49.3.3 Écriture du fichier C++

49.3.3.1 Directive `#include`

Placée en tête du fichier, elle permet d'importer toutes les déclarations GALGAS d'un projet, y compris les prototypes des fonctions et procédures externes à GALGAS. Son libellé exact est :

```
#include "all-declarations.h"
```

Le fichier `all-declarations.h` est engendré par la compilation GALGAS et est situé dans le répertoire `build/output`.

49.3.3.2 Squelette de l'implémentation d'une fonction externe

Par exemple, on considère la fonction GALGAS externe :

```
extern func maFonctionExterne ?@uint a ?@uint b -> @uint
```

Le nom C++ de la fonction GALGAS `maFonctionExterne` est `function_maFonctionExterne`, c'est-à-dire que le nom GALGAS est préfixé par `function_`. Il y a d'autres transformations, si le nom de la fonction est contient des lettres non-ASCII, des chiffres, ou des caractères « _ ». En effet, le C++ n'accepte pas les lettres non ASCII dans les identificateurs, le compilateur GALGAS les remplacent par la séquence « _xx_ »,

où xx est le point de code du caractère, exprimé en hexadécimal. Pour ne pas introduire d'ambiguïté, le caractère « _ » est transformé de la même façon, il devient donc « _5F_ ». Pour des raisons historiques, il en est de même pour les chiffres décimaux.

Ainsi, une fonction nommée `hé_hé` en GALGAS est traduite en `function_h_E9__5F_h_E9_` en C++.

Pour écrire l'en-tête de la fonction, le plus simple est de rechercher son prototype dans les fichiers d'en-tête du répertoire `build/output`. Pour la fonction `maFonctionExterne`, celui-ci est :

```
class GALGAS_uint function_maFonctionExterne (class GALGAS_uint inArgument0,
                                              class GALGAS_uint inArgument1,
                                              class C_Compiler * inCompiler
                                              COMMA_LOCATION_ARGS) ;
```

Deux remarques :

- Les noms de types GALGAS (par exemple `@xyz`) sont préfixés par `GALGAS_` en C++, et les lettres non-ASCII, les chiffres, et les caractères « _ » sont remplacés par la séquence « _xx_ », où xx est le point de code du caractère, exprimé en hexadécimal (comme pour les noms de fonction, voir ci-dessus) ; ainsi, `@uint64` devient `GALGAS_uint_36__34_` ;
- il n'y a pas de virgule après « `inCompiler` », `COMMA_LOCATION_ARGS` est une macro (voir sa définition dans le fichier `build/libpm/utilities/M_SourceLocation.h`).

On peut donc écrire le squelette de la fonction C++ :

```
GALGAS_uint function_maFonctionExterne (GALGAS_uint inArgument0,
                                        GALGAS_uint inArgument1,
                                        C_Compiler * inCompiler
                                        COMMA_LOCATION_ARGS) {
}
```

Pour écrire le code de la fonction, il faut prendre en compte les *valeurs poison*. En GALGAS, il n'y a pas d'exception, et l'exécution continue en séquence, même si une erreur est détectée. Par exemple, une erreur se déclenche si on recherche une clé inexistante dans une table ; dans ce cas, il y a impossibilité de fournir des valeurs valides aux variables devant recevoir les informations associées à la clé. Autre exemple, si on essaie de diviser un entier par 0, il y a impossiblité de retourner un résultat valide. Dans tous ces cas, la valeur retournée est une *valeur poison*. Toute entité valuée GALGAS peut ainsi prendre deux états :

- *valide*, l'entité possède une valeur valide ;
- *poison*, l'entité ne possède pas de valeur valide.

Aussi toute routine – aussi bien les routines implémentées en dur dans GALGAS que les routines externes écrites directement en C++ – doit vérifier si les valeurs qu'elle reçoit sont valides. Si il n'est pas possible d'effectuer le calcul souhaité, il faut retourner une valeur poison. On pourra regarder avec profit comment sont implémentées les opérations sur les `@uint`, dans le fichier `build/libpm/galgas2/GALGAS_uint.cpp`.

Tout type GALGAS définit une méthode `isValid` qui permet de savoir si l'objet possède une valeur valide ou non. Le constructeur par défaut d'un objet crée une valeur poison, les constructeurs dédiés une valeur valide.

Par exemple, si l'on veut que `maFonctionExterne` retourne la somme des deux arguments, on écrira :

```
GALGAS_uint function_maFonctionExterne (GALGAS_uint inArgument0,
                                        GALGAS_uint inArgument1,
                                        C_Compiler * /* inCompiler */
                                        COMMA_UNUSED_LOCATION_ARGS) {
  GALGAS_uint result ; // Valeur poison
  if (inArgument0.isValid () && inArgument1.isValid ()) {
    result = GALGAS_uint (inArgument0.uintValue () + inArgument1.uintValue ()) ;
  }
  return result ;
}
```

Dès que l'un des deux arguments est une valeur poison, le résultat renvoyé est une valeur poison.

49.3.3.3 Squelette de l'implémentation d'une procédure externe

Par exemple, on considère la procédure GALGAS externe :

```
extern proc maProcédureExterne !@uint a ?!@uint b ?@uint64 c
```

Cette déclaration engendre le prototype suivant dans un des fichiers d'en-tête C++ (situé dans le répertoire `build/output`) :

```
void routine_maProc_E9_dureExterne (class GALGAS_uint & outArgument0,
                                    class GALGAS_uint & ioArgument1,
                                    class GALGAS_uint_36__34_ inArgument2,
                                    class C_Compiler * inCompiler
                                    COMMA_LOCATION_ARGS) ;
```

Remarques :

- le nom de la procédure GALGAS est préfixée par `routine_`, et, comme pour les noms de fonction, les lettres non-ASCII, les chiffres, et les caractères « _ » sont remplacés par la séquence « _xx_ », où xx est le point de code du caractère, exprimé en hexadécimal : ainsi, « `é` » devient « `_E9_` » ;
- un argument en sortie (comme `outArgument0`) est passé par référence ;
- un argument en entrée / sortie (comme `ioArgument1`) est passé par référence ;
- un argument en entrée (comme `inArgument2`) est passé par valeur (le constructeur de recopie est appelé).

Chapitre 50

Extensions

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

You can declare for any type :

- *category getters* ;
- *category methods* ;
- *category setters*.

Additional features are available for classes and are described in section **??** page **??**.

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

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

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

Within the category getter, method and setter instruction list, the `self` key word is allowed in any expression. It represents a copy of the current object. Of course, the current is lazily copied only when required.

The `self` key word is just a syntactic tag for representing a write or a read/write access to the current object. Using `self` is not allowed in category methods and category getters since they cannot modify the current object. Using `self` in category setters is explained in section **??** page **??**.

A category getter, method and setter can be declared in :

- a *semantics* component ;

- a *syntax* component;
- a *program* component.

A declared category getter, method and setter has a global scope, meaning it is available in the current component, and in any component that includes it directly or indirectly.

A type does not accept several category getters with the same name. During compilation of the project file, the project global checking mechanism detects such declarations and issues an error. Consequently, it is forbidden to declare a category getter with the same name than a predefined getter : the compiler issues an error on on a such declaration. The same rules apply on category methods and category setters.

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

50.1 Category getter

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

For example, the following code add a getter to the `uint64` that computes the square of its value :

```
getter @uint64 square -> @uint64 {
  result = self * self
}
```

This getter is called like a predefined getter :

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

You can add a category getter to a list :

```
getter @uintlist sum -> @uint {
  result = 0
  for self do
    result = result + mValue
  }
}
```

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

```
getter @uintlist countValuesGreaterThan
  ?let @uint inTestValue -> @uint
{
  result = 0
  for self do
```

```
    if mValue > inTestValue then
      result ++
    end if
  }
}
```

When used with a struct or class type, current object attributes values can be read by naming the attribute in an expression. For example, the **@lstring** (page **??**) has an attribute `string` whose type is `@string`. The following getter returns the value of this attribute, appended with the `" !"` string :

```
getter @lstring op -> @string {
  result = string . " !"
}
```

50.2 Category method

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

For example, the following code add a method to the `uint64` that computes the square of its value :

```
method @uint64 square !@uint64 {
  result = self * self
}
```

This getter is called like a predefined method :

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

You can add a category method to a list :

```
method @uintlist sum !@uint {
  result = 0
  for self do
    result = result + mValue
  }
}
```

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

```
method @uintlist countValuesGreaterThan
  ?let @uint inTestValue
  !@uint outResult
{
  outResult = 0
```

```
  for self do
    if mValue > inTestValue then
      outResult ++
    end if
  }
}
```

When used with a struct or class type, current object attributes values can be read by naming the attribute in an expression. For example, the `@lstring` (page **??**) 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 . " !"
}
```

50.3 Category setter

A category method is declared like a routine, but its header names a type and a setter name. As a routine, it accepts zero, one or more output, input/output, input and constant input formal parameters. Unlike a category method, a category setter can change the value of the current object.

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

```
setter @lstring appendInt ?let @uint inValue {
  string .= [inValue string]
}
```

The `self` key word is used as a syntactic tag for denoting a read/write or a write access on the current object. This key word is syntactically accepted in the following constructs :

1. the *setter call instruction* (section **??** page **??**);
2. the *concat instruction* (section **??** page **??**);
3. the *increment instruction* (section **??** page **??**);
4. the *decrement instruction* (section **??** page **??**);
5. the *assignment instruction* (section **??** page **??**).

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

```
setter @uint64list prependSquare ?let @uint64 inValue {
  [!?self prependValue !inValue * inValue]
}
```

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

```
setter @uintlist appendSquare ?let @uint inValue {
  self += !inValue * inValue
}
```

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

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

```
setter @string concatList ?let @stringlist inList {
  for inList do
    self .= mValue
  }
}
```

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

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

```
setter @uint increment {
  self ++
}
```

This construct is valid only for types that handle the `++` operator, such as `@uint` (page **??**), `uint64` , `@sint` (page **??**), `@sint64` (page **??**).

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

```
setter @uintlist removeOddValues {
  @uintlist listWithEvenValues [emptyList]
  for self do
    if (mValue & 1) == 0 then
      listWithEvenValues += !mValue
    end if
  }
  self = listWithEvenValues
}
```

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

50.4 Categories and classes

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

- *abstract* category getters, methods, setters ;
- *overriding* category getters, methods, setters ;
- *overriding abstract* category getters, methods, setters.

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

Examples of *abstract* category getters, methods, setters declarations :

```
abstract getter @aType getterName
  ?anOtherType aParameter
  -> @resultType

abstract method @aType methodName
  ?anOtherType aParameter
;

abstract setter @aType setterName
  ?anOtherType aParameter
;
```

Examples of *overriding* category getters, methods, setters declarations :

```
override getter @aType getterName
  ?anOtherType aParameter
  -> @resultType
{
  instructions
}

override method @aType methodName
  ?anOtherType aParameter
{
  instructions
}

override setter @aType setterName
  ?anOtherType aParameter
{
  instructions
}
```

Examples of *overriding abstract* category getters, methods, setters declarations :

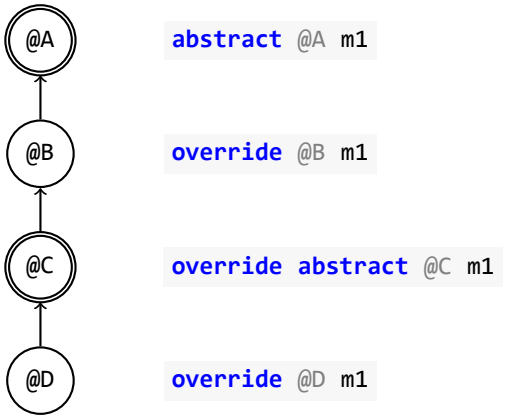


Figure 50.1 – *inheritance graph and categories*

```
override abstract getter @aType getterName
  ?@anOtherType aParameter
  -> @resultType

override abstract method @aType methodName
  ?anOtherType aParameter

override abstract setter @aType setterName
  ?anOtherType aParameter
```

Neither *abstract* category getters, methods, setters, neither *overriding abstract* category getters, methods, setters cannot be declared for concrete classes. Any kind of category getter, method, setter can be declared for abstract classes.

If an *abstract* category getter, method, setter, or an *overriding abstract* category getter, method, setter is declared for an abstract class, it should be also declared as *overriding* with the same name for every first concrete successor class.

A category getter, method, setter that has the same name as a category getter, method, setter declared for one of its super classes should be declared as *overriding*.

An abstract category getter, method, setter that has the same name as a category getter, method, setter declared for one of its super classes should be declared as *overriding abstract*.

The following example illustrates how theses rules should be applied. In the figure **??**, four classes are shown. An arrow links a class to its super class. The `@A` and `@C` classes are abstract. `m1` is a name for any getter, method or setter.

`m1` is declared as `abstract` for the `@A` class. It is allowed since `@A` is abstract. Consequently, the concrete `@B` class should override `m1` . The `@C` class is also abstract, and `m1` can be declared as `abstract` for this class. But as it has been also declared for one of theses super class, it should also declared as `override` . As `@D` is concrete, `m1` should be declared for this class with `override` tag.

V

Filewrappers et templates

Chapitre 51

Filewrappers

Un *filewrapper* permet d'embarquer dans le code engendré une arborescence de fichiers. Comme on va le voir dans la section suivante, la déclaration d'un *filewrapper* désigne un répertoire, qui va être exploré au moment de la compilation GALGAS de façon à embarquer dans le code engendré la copie de certains fichiers. Ces fichiers peuvent être de trois sortes :

- des fichiers *texte* ; ils sont sélectionnés par leur extension : la déclaration d'un *filewrapper* liste toutes les extensions des fichiers texte embarqués ;
- des fichiers *binaires* ; de même, ils sont sélectionnés par leur extension, et la déclaration d'un *filewrapper* liste toutes les extensions des fichiers binaires embarqués ;
- des *templates*, qui sont sélectionnés par leur nom ; ils sont analysés lors de leur lecture.

L'exploration des fichiers embarqués peut s'effectuer soit de manière statique, soit dynamique à l'aide d'un objet de `@filewrapper` (page **??**).

51.1 Déclararation d'un `filewrapper`

La déclaration d'un *filewrapper* 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 fichier ou ne définir aucun template.

VI

Instructions et expressions

Chapitre 52

Contrôle de l'accès aux variables et aux constantes

Le compilateur GALGAS effectue une surveillance très stricte des accès aux objets – constantes, variables et paramètres formels. Il signale ainsi par des *alertes* et des *erreurs* tout violation des règles d'accès.

On peut illustrer le résultat de cette surveillance par le fragment de code suivant :

```
var @uint x
if condition then
  x = 2
end
let y = x # Une erreur de compilation est déclenchée ici
```

Quelle serait la valeur de la variable `x` à l'issue de l'exécution de ce code ? Si `condition` est vrai, `x` vaut 2 ; sinon, `x` n'a pas de valeur.

Le compilateur GALGAS détecte cette situation et considère que la variable est initialisée si elle l'est par toutes les branches de l'instruction `if` . Dans le cas contraire, comme ci-dessus, elle est considérée comme non initialisée. Aussi sa lecture déclenche un message d'erreur. Pour que l'analyse sémantique ne détecte pas d'erreur, il faut donc que les deux branches affectent une valeur à `x` :

```
var @uint x
if condition then
  x = 2
else
  x = 4
end
let y = x # Ok
```

Pour contrôler le bon usage des variables et des constantes locales, le compilateur GALGAS associe pendant la compilation un automate d'états finis à chaque variable locale.

52.1 Variable locale

L'automate d'états finis associé à une variable locale est présenté à la figure **??**. C'est la forme la plus générale, les autres automates s'en déduisent.

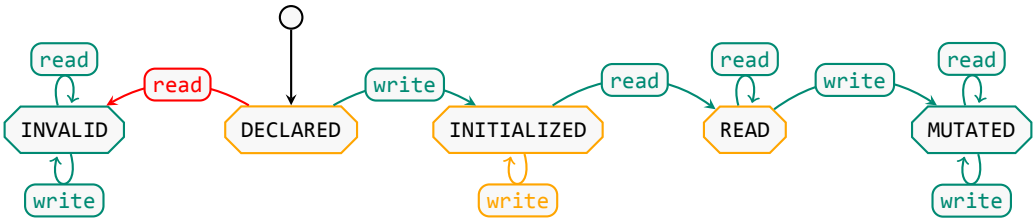


Figure 52.1 – *Automate des états d'une variable locale*

Les états sont les suivants :

- `INVALID`, une erreur d'accès a été détectée ;
- `DECLARED`, état d'une variable déclarée non initialisée ;
- `INITIALIZED`, état d'une variable déclarée et initialisée, mais jamais lue ;
- `READ`, état d'une variable déclarée, initialisée, lue au moins une fois, mais jamais modifiée ;
- `MUTATED`, état d'une variable déclarée, initialisée, et modifiée au moins une fois.

Un état initial est désigné par une flèche noire. La déclaration d'une variable locale place son automate dans l'état `DECLARED` :

```
var @uint x # État 'declared'
```

Il y a deux actions possibles :

- l'action `write`, qui exprime l'affectation d'une valeur à la variable ;
- l'action `read`, qui exprime la lecture de la valeur de la variable.

L'automate est *complet*, c'est-à-dire que les deux actions sont prises en compte dans tous les états.

Les transitions sont présentées en couleur, selon leur validité :

- *vert*, la transition est correcte et ne donne lieu à aucune émission de message d'alerte ou d'erreur ;
- *orange*, la transition est correcte et mais donne lieu à l'émission d'un message d'alerte ;
- *rouge*, la transition est incorrecte et donne lieu à l'émission d'un message d'erreur.

Ainsi, la lecture d'une variable dans l'état `DECLARED` est une erreur : en effet, la variable n'a pas de valeur. La transition correspondante est donc en rouge. Voici un exemple qui illustre cette situation :

```
var @uint x # État 'declared'
var y = x # Erreur, x n'a pas de valeur
```

L'écriture d'une variable qui est dans l'état `INITIALIZED` n'est pas une erreur, mais révèle une anomalie : la valeur écrasée n'a jamais été lue ; un message d'alerte est donc émis. Par exemple :

```
var @uint x = 2 # État 'initialized'
var x = 3 # Alerte, l'initialisation de x à 2 est inutile
```

Remarquons que les actions `read` et `write` sont acceptées silencieusement dans l'état `INVALID` : en effet, on arrive dans cet état après l'occurrence d'une erreur qui a été signalée à l'utilisateur, en acceptant silencieusement on n'émet pas de message d'erreur à chaque accès à la variable.

Enfin, l'état final de l'automate. À la fin de la portée de la variable, son automate est supprimé. Au moment de sa suppression, son état courant est considéré comme son état final. Trois situations peuvent survenir, qui sont reflètées la couleur du cadre de l'état :

- *verte*, l'état final est correct et ne donne lieu à aucune émission de message d'alerte ou d'erreur ;
- *orange*, l'état final est correct et mais donne lieu à l'émission d'un message d'alerte ;
- *rouge*, l'état final est incorrect et donne lieu à l'émission d'un message d'erreur.

Ainsi, l'état `READ` est un état final correct pour une variable locale. L'état `INVALIDE` est aussi silencieusement accepté, être dans cet état signifie qu'un message d'erreur a déjà été émis pour cette variable.

L'état `DECLARED` comme état final signifie que la variable a été déclarée, sans être initialisée : la variable est inutile, et un message d'alerte est émis.

L'état `INITIALIZED` comme état final signifie que la variable a été déclarée, initialisée, mais jamais lue : comme précédemment, la variable est inutile, et un message d'alerte est émis.

L'état `READ` comme état final signifie que la variable a été déclarée, initialisée, lue, mais jamais modifiée : c'est en fait une constante et un message d'alerte est émis, qui suggère de transformer la variable locale en constante locale.

52.2 Constante locale

L'automate d'états finis associé à une constante locale est présenté à la figure **??**. Une constante locale peut être écrite une seule fois, à partir de l'état `DECLARED`. En conséquence, une action `write` à partir des états `INITIALIZED` et `READ` est une erreur (elle apparaît en rouge) et redirige vers l'état `INVALID`. L'état `MUTATED` n'est plus accessible, et a été supprimé de la figure **??**.

Voici un exemple.

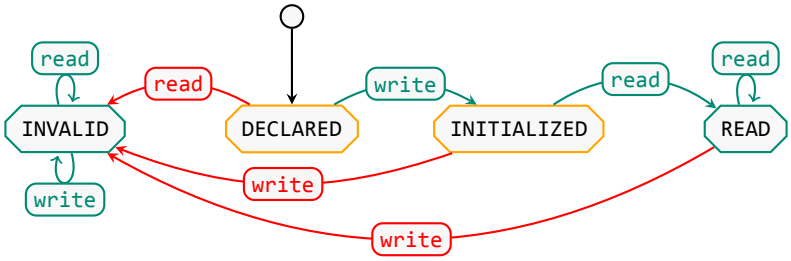


Figure 52.2 – *Automate des états d'une constante locale*

```
let @uint x = 2 # État 'initialized'
...
var y = x # Ok, l'état de 'x' est 'read'
```

On n'est pas obligé de fournir une valeur à la déclaration d'une constante. On peut ainsi écrire :

```
let @uint x # État 'declared'
...
x = 2 # État 'initialized'
...
var y = x # Ok, l'état de 'x' est 'read'
```

Chapitre 53

Expressions

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

53.1 Opérandes

53.1.1 Identificateur

53.1.2 `self`

Dans une expression, `self` représente une copie de l'objet courant. On ne peut donc utiliser `self` que dans une expression à l'intérieur d'une *méthode*, d'un *getter*, d'un *setter*, ou d'une extension (chapitre **??** à partir de la page **??**). Sont donc exclues les procédures et les fonctions.

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

Voici un exemple extrait de la section décrivant les *extensions de getter* (section **??** page **??**) :

```
getter @uint64 square -> @uint64 outResult {
  outResult = self * self
}
```

53.1.3 Expression de conversion polymorphique inverse

Cette construction est obsolète, et est remplacée avantageusement par l'*affectation conditionnelle* (section **??** page **??**).

La syntaxe de l'*expression de conversion polymorphique inverse* est :

```
expression as @T
```

Elle permet de renvoyer la valeur de `expression` sous la forme d'un objet de type statique `@T`. À l'exécution, la conversion échoue si le type dynamique de l'`expression` n'est pas `@T` ou une de ses classes héritières ; une erreur sémantique est alors déclenchée, et l'expression renvoie un objet *non construit*.

Pour tester le type dynamique de l'expression avant d'effectuer la conversion, utiliser la construction décrite à la section **??** page **??**. On peut aussi utiliser l'instruction `cast` (section **??** page **??**).

53.1.4 Test du type dynamique d'une expression

L'opérande `expression is conversion @T` teste le type dynamique de `expression` vis à vis du type `@T` :

- si `conversion` est `==`, la valeur renvoyée est `true` si le type dynamique de l'`expression` est exactement `@T`, et `false` dans le cas contraire ;
- si `conversion` est `>=`, la valeur renvoyée est `true` si le type dynamique de l'`expression` est `@T` ou une de ses classes héritières, et `false` dans le cas contraire ;
- si `conversion` est `>`, la valeur renvoyée est `true` si le type dynamique de l'`expression` n'est pas `@T` mais une de ses classes héritières, et `false` dans le cas contraire.

Alliée à la construction précédente, elle permet de lancer une conversion uniquement si elle est possible :

```
if expression is == @B then
  let @B cst = expression as @B
  ...
elsif expression is >= @C then
  let @C cst = expression as @C
  ...
else
  message "conversion impossible"
end
```

53.1.5 Parenthèses

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

53.1.6 `true` et `false`

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

53.1.7 Constante chaîne de caractères

53.1.8 Constante caractère

53.1.9 Constante entière

Une constante entière est une séquence de chiffres décimaux, éventuellement séparés par le caractère de soulignement _, et terminé par un suffixe. Ce suffixe détermine le type de la constante :

- pas de suffixe : `@uint` ;
- suffixe `S` : `@sint` ;
- suffixe `L` : `@uint64` ;
- suffixe `LS` : `@sint64` ;
- suffixe `G` : `@bigint` .

53.1.10 Constante flottante

53.1.11 Expression `if`

53.1.12 Appel de fonction

53.1.13 Appel de getter

53.1.14 Constructeur

L'appel d'un constructeur instancie un nouvel objet. Sa syntaxe est :

```
@T.constructeur {!exp0 !exp1 ...}
```

Par exemple :

```
@lstring ls = @lstring.new {!"" !@location.here{}}
@stringlist str = @lstringlist.emptyList {}
```

Deux simplifications syntaxiques sont proposées :

- si la liste des arguments est vide, les accolades peuvent être omises ;
- si le type peut être inféré, il peut être omis.

53.1.14.1 Suppression des accolades

Si la liste des arguments est vide, les accolades peuvent être omises.

```
@lstring ls = @lstring.new {!"" !@location.here}
@stringlist str = @lstringlist.emptyList
```

53.1.14.2 Inférence du type

Si le type peut être inféré, il peut être omis (remarquer que ceci est valable aussi pour `@location.here` qui peut être simplifié en `.here`.

```
@lstring ls = .new {!"" !.here}
@stringlist str = .emptyList
```

53.1.15 Constructeur par défaut

Pour la plupart des types, un constructeur par défaut est implicitement défini (voir la liste précise tableau **??** page **??**).

L'expression `@T.default` invoque le constructeur par défaut du type `@T` et renvoie un objet initialisé du type `@T`. Le type `@T` peut être inféré et l'appel du constructeur par défaut s'écrit simplement `.default`.

53.1.15.1 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 {
  !{}
  !{}
```

```
  !{}
  !{}
}
```

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`.

53.1.15.2 Les constructeurs par défaut pour chaque type

Le tableau **??** page **??** 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.

53.1.16 Valeur d'une option

Les options de la ligne de commande sont définies dans un composant `option` (chapitre **??** à partir de la page **??**). 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 **??**.

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

```
option mesOptions {
  @bool extractOption : 'S', "asm" -> "Extract assembly code"
}
```

Alors :

Type	Constructeur par défaut
`abstract class @T`	*Pas de constructeur par défaut*
`@application`	*Pas de constructeur par défaut*
`array @T`	*Pas de constructeur par défaut*
`@bigint`	Équivalent au constructeur `zero`, initialisation à `0G`
`@bool`	Initialisation à `false`
`@boolset @T`	Équivalent au constructeur `.none` (section **??** page **??**)
`@char`	Initialisation au caractère `NULL` : `'\0'`
`class @T`	Oui, si tous les attributs possèdent un constructeur par défaut
`@data`	Équivalent au constructeur `emptyData`
`@double`	Initialisation à `0.0`
`enum @T`	Voir section **??** page **??**
`@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
`@timer`	Équivalent au constructeur `start`
`@type`	*Pas de constructeur par défaut*
`@uint`	Initialisation à `0`
`@uint64`	Initialisation à `0L`

Tableau 53.1 – *Constructeur par défaut pour chaque type*

- `[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"`.

Noter qu'à partir de la version 3.1.4, les options *quiet* et *verbose* (section **??** page **??**) ne peuvent pas être appelées par cette construction : il faut utiliser le *constructeur `verboseOutput` du type `@application`* – *page* **??**.

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

Priorité	Opérateur	Commentaire	Référence
0 (plus faible)	`|`	« ou » logique	section **??** page **??**
	`||`	« ou », évaluation en court-circuit	section **??** page **??**
	`^`	« ou exclusif » logique	section **??** page **??**
1	`&`	« et » logique	section **??** page **??**
	`&&`	« et », évaluation en court-circuit	section **??** page **??**
2	`==` , `!=`	Test d'identité	section **??** page **??**
	`<` , `<=`	Comparaison	section **??** page **??**
	`>` , `>=`	Comparaison	section **??** page **??**
3	`<<` , `>>`	Décalage	section **??** page **??**
	`+` , `&+`	Addition, concaténation	section **??** page **??**
	`-` , `&-`	Soustraction	section **??** page **??**
4	`*` , `&*`	Multiplication	section **??** page **??**
	`/` , `&/`	Division	section **??** page **??**
	`mod`	Modulo	section **??** page **??**
5	`-` , `&-`	Négation arithmétique	section **??** page **??**
6	`not`	Complémentation booléenne	section **??** page **??**
7	`~`	Complémentation bit-à-bit	section **??** page **??**
8 (plus forte)	`.`	Accès à un champ d'une structure	section **??** page **??**

Tableau 53.3 – *Priorité des opérateurs*

53.2 Opérateurs

53.2.1 Priorité des opérateurs

La priorité des opérateurs est définie dans le tableau **??**. 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`.

53.2.2 Logique

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

53.2.3 Logique, évaluation en court-circuit

`||` et `&&`

53.2.4 Complémentation bit-à-bit

`~`

53.2.5 Comparaison

53.2.6 Décalage

`<<` et `>>`

53.2.7 Arithmétique

GALGAS propose des opérateurs arithmétiques qui détectent les dépassements de capacité : `+`, `-`, `*`. Une erreur est déclenchée à chaque débordement.

L'opérateur de division `/` déclenche une erreur lors d'une division par zéro. Pour les types signés, une erreur est aussi déclenchée par la division de `@sint.min` par `-1S`, et la division de `@sint64.min` par `-1LS`.

Ces opérateurs ne détectent pas un dépassement de capacité : `&+`, `&-`, `&*`, et fonctionnent donc comme les opérateurs arithmétiques du C.

L'opérateur de division `&/` ne déclenche aucune erreur lors d'une division par zéro, la valeur renvoyée est zéro. La division de `@sint.min` par `-1S` retourne `@sint.min`, et celle de `@sint64.min` par `-1LS` retourne `@sint64.min`.

L'opérateur `mod` déclenche une erreur en cas de division par zéro.

La négation arithmétique (opérateur `-` unaire) est uniquement définie sur les types entiers signés, et détecte les débordements provoqués par `- @sint.min` et `- @sint64.min`.

Il existe un second opérateur de négation arithmétique (`&-`), lui aussi uniquement défini sur les types entiers signés, qui ne détecte aucun débordement : `&- @sint.min` renvoie `@sint.min` et `&- @sint64.min` renvoie `@sint64.min`.

On peut tester l'absence de débordement grâce aux *getters* `canAdd`, `canSubstract`, `canMultiply` et `canDivide`, définis pour les quatre types `@uint`, `@uint64`, `@sint` et `@sint64`. Si ces *getters* renvoient `true`, alors les opérations correspondantes s'effectueront sans débordement. Par exemple :

```
@uint a = ...
@uint b = ...
if [a canAdd !b] then
  @uint r = a + b # Pas de débordement
else
  # L'addition provoquerait un débordement
end
```

53.2.8 Accès à un champ d'une structure

Chapitre 54

Instructions sémantiques

54.1 Rôle du point-virgule « ; »

Le point-virgule n'est pas un terminateur d'instruction. Il représente une instruction vide. Aussi, il peut être utilisé en nombre quelconque entre deux instructions consécutives. Ainsi, on peut écrire :

```
instruction1  instruction2
```

Ou encore :

```
instruction1 ; instruction2
```

```
instruction1 ;;;; instruction2
```

54.2 Instruction de déclaration de variable

Une déclaration de variable peut citer une `expression` qui lui fournit sa valeur initiale. Dans le cas contraire, la variable est *non initialisée* jusqu'à ce qu'une valeur lui soit affectée.

Les deux formes de déclaration sont donc :

- déclaration d'une variable *non initialisée* : « `var @type variable` »;
- déclaration d'une variable *initialisée* : « `var @type variable = expression` ».

54.2.1 Déclaration «`var @type variable`»

La forme générale de déclaration d'une variable *non initialisée* est :

```
var @type variable
```

54.2.2 Déclaration «var @type variable = expression»

La forme générale de déclaration d'une variable *initialisée* est :

```
var @type variable = expression
```

On peut omettre le type, à la condition que l'expression puisse fournir l'information de type. Par exemple, dans l'écriture suivante :

```
var s = "Hello" # Type implicite @string
```

Prenons un autre exemple, celui de l'initialisation d'une liste :

```
var @stringlist sl = @stringlist {}
```

Le type est annoté de façon redondante, puisqu'il apparaît à la fois dans le membre de gauche et dans l'expression ; l'une des deux annotations peut être omise :

```
var sl = @stringlist {}
```

Ou :

```
var @stringlist sl = {}
```

Par contre, omettre les deux annotations ne permet pas de déduire le type de la variable ; c'est donc une erreur détectée par le compilateur :

```
var sl = {} # Erreur : le type est indéterminé
```

Un dernier exemple, avec un constructeur :

```
var @location sl = @location.here
```

On peut écrire :

```
var sl = @location.here
```

Ou :

```
var @location sl = .here
```

Mais en aucun cas :

```
var sl = .here # Erreur : le type est indéterminé
```

54.3 Instruction de déclaration de constante

Une déclaration de constante peut citer une `expression` qui lui fournit sa valeur initiale. Dans le cas contraire, la variable est *non initialisée* jusqu'à ce qu'une valeur lui soit affectée.

Les deux formes de déclaration sont donc :

- déclaration d'une constante *initialisée* : « `let @type constante = expression` ».
- déclaration d'une constante *non initialisée* : « `let @type constante` » ;

54.3.1 Déclaration d'une constante initialisée

Le mot clé `let` caractérise une déclaration de constante. L'annotation de type peut être omis si le type peut être déduit de l'expression, comme pour l'instruction de déclaration de variable (section **??** page **??**). On peut donc reprendre les exemples de la section précédente :

```
let @string s = "Hello"
```

L'expression est une chaîne de caractères, dont le type est `@string`. On peut donc omettre l'annotation de type dans l'instruction :

```
let s = "Hello"
```

Prenons un autre exemple, celui de l'initialisation d'une liste :

```
let @stringlist sl = @stringlist {}
```

Le type est annoté de façon redondante, puisqu'il apparaît à la fois dans le membre de gauche et dans l'expression ; l'une des deux annotations peut être omise :

```
let sl = @stringlist {}
```

Ou :

```
let @stringlist sl = {}
```

Par contre, omettre les deux annotations ne permet pas de déduire le type de la constante, c'est donc une erreur détectée par le compilateur :

```
let sl = {} # Erreur : le type est indéterminé
```

Un dernier exemple, avec un constructeur :

```
let @location sl = @location.here
```

On peut écrire :

```
let sl = @location.here
```

Ou :

```
let @location sl = .here
```

Mais en aucun cas :

```
let sl = .here # Erreur : le type est indéterminé
```

54.3.2 Déclaration d'une constante non initialisée

Il est possible de déclarer une constante sans fournir de valeur initiale. Une seule affectation est alors permise.

```
let @string s
...
s = "Hello"
...
s = "World" # Erreur sémantique, une constante ne peut pas être modifiée
```

L'intérêt est de pouvoir initialiser la constante dans chaque branche d'une instruction sélective :

```
let @string s
if ... then
  s = "Hello 1"
else
  s = "Hello 2"
end
```

54.4 L'instruction d'affectation

L'instruction d'affectation peut prendre plusieurs formes. La plus courante est :

```
variable = expression
```

Si `variable` est une instance de structure (chapitre **??** à partir de la page **??**), on peut directement en modifier un champ en utilisant l'opérateur `.` :

```
variable.champ = expression
```

Si `champ` est lui-même une structure, on peut accéder à l'un de ses champs (et ainsi de suite) :

```
variable.champ.champ1 = expression
```

54.5 L'instruction `cast`

L'instruction `cast` permet simplement d'exprimer de manière élégante une série de tests de conversions polymorphiques inverses. Sa syntaxe est :

```
cast expression
case conversion @T1 nom1 :
  ...
```

```
case conversion @T2 nom2 :
  ...
else
  ...
end
```

L'instruction accepte une ou plusieurs branches `case`, et zéro ou une branche `else`. `conversion` est soit `==`, soit `>=`. `nom1` et `nom2` sont des constantes dont le type est le type nommé dans la branche `case` qui la déclare, et dont la portée est limitée à cette branche `case`.

Lors de l'exécution, le type dynamique de `expression` est comparé successivement aux types (`@T1`, `@T2`) nommés dans les branches `case` ; dès que ce type dynamique est :

- exactement la classe `@T` (`conversion` est `==`),
- la classe `@T` ou de l'une de ses classes héritières (`conversion` est `>=`),
- une classe héritière de la classe `@T`, mais pas la classe `@T` (`conversion` est `>`),

alors la constante prend la valeur de `expression` et les instructions de la branche correspondante sont exécutées.

Si toutes les comparaisons échouent, la branche `else` est exécutée (si elle existe). La forme typique de cette instruction est donc :

```
cast expression
case >= @B v1 :
  ...
case >= @C v2 :
  ...
else
  message "conversion impossible"
end
```

Note : si la variable `nom1` ou `nom2` n'est pas utilisée dans la branche correspondante, une alerte est émise. Pour la supprimer, ne pas mentionner la variable en écrivant `case >= @T :`.

54.6 L'instruction d'ajout += d'un élément à une collection

L'instruction `+=` présente deux syntaxes :

- le membre de droite est une expression : `cible += expression` ; cette instruction est décrite section **??** page **??** ;
- le membre de droite est une liste d'expressions : `cible += !expression1 ... !expressionN` ; cette instruction est décrite ci-dessous.

La `cible` est une variable ou un champ de structure.

Cette instruction s'applique aux types suivants :

- `@stringset` (section **??** page **??**);
- `list @T` (section **??** page **??**);
- `sortedlist @T` (section **??** page **??**);
- `map @T` (section **??** page **??**).

54.6.1 Instruction d'ajout et le type `@stringset`

Sous la forme `cible += expression`, l'instruction permet de concaténer d'effectuer l'union des ensembles de chaînes :

```
var strset1 = @stringset {!"a", !"b"} # Valeur : "a", "b"
var strset2 = @stringset {!"b", !"c"} # Valeur : "b", "c"
strset1 += strset2 # strset1 vaut "a", "b", "c"
```

La forme `cible += !expression1 ... !expressionN` permet d'ajouter une chaîne à l'ensemble :

```
var strset1 = @stringset {!"a", !"b"} # Valeur : "a", "b"
strset1 += !"c" # strset1 vaut "a", "b", "c"
strset1 += !"b" # strset1 vaut "a", "b", "c"
```

54.6.2 Instruction d'ajout et les listes

Sous la forme `cible += expression`, l'instruction effectue une concaténation de listes : `cible` et `expression` doivent avoir le même type `list @T`, et l'`expression` est ajoutée à la fin de la `cible`.

```
var liste1 = @stringlist {!"a", !"b"}
var liste2 = @stringlist {!"c", !"d"}
liste1 += liste2 # liste1 vaut "a" "b" "c" "d"
```

Sous la forme `cible += !expression1 ... !expressionN`, l'instruction ajoute un élément à la fin de `cible`. L'élément est défini par la liste des valeurs de ses champs.

Avec la liste :

```
list @maListe {
  @uint mProperty1
  @string mProperty2
}
```

On a :

```
var liste = @maListe {}
liste += !2 !"a" # liste vaut contient un élément 2, "a"
```

54.6.3 Instruction d'ajout et les listes triées

Sous la forme `cible += expression`, l'instruction effectue une concaténation de listes : `cible` et `expression` doivent avoir le même type `sortedlist @T`, et chaque élément de `expression` est ajouté à la `cible` en respectant l'ordre de tri.

Avec la liste triée :

```
sortedlist @maListeTriee {
  @uint mProperty1
  @string mProperty2
}{
  mProperty1 <
}
```

```
var liste1 = @maListeTriee {!3 !"a", !1 !"c"} # Valeur : (1, "c"), (3, "a")
var liste2 = @maListeTriee {!4 !"d", !2 !"b"} # Valeur : (2, "b"), (4, "d")
liste1 += liste2 # liste1 vaut (1, "c"), (2, "b"), (3, "a"), (4, "d")
```

Sous la forme `cible += !expression1 ... !expressionN`, l'instruction ajoute un élément à la fin de `cible`. L'élément est défini par la liste des valeurs de ses champs.

On a :

```
var liste = @maListeTriee {}
liste += !2 !"a" # Valeur : (2, "a")
liste += !1 !"b" # Valeur : (1, "b"), (2, "a")
liste += !3 !"c" # Valeur : (1, "b"), (2, "a"), (3, "c")
```

54.6.4 Instruction d'ajout et les tables

La forme `cible += expression` n'est pas prise en charge.

Sous la forme `cible += !expression1 ... !expressionN`, l'instruction ajoute un élément à la table `cible`. L'élément est défini par la clé et suivie de la liste des valeurs de ses champs.

Avec la table :

```
map @maTable {
  @uint mProperty1
  @string mProperty2
}
```

on a :

```
var table = @maTable {}
@lstring clef = ...
table += !clef !2 !"a"
```

54.7 Affectation combinée à une opération : +=, -=, *= et /=

L'instruction `+=` présente deux syntaxes :

- le membre de droite est une expression : `cible += expression` ; cette instruction est décrite ci-dessous ;
- le membre de droite est une liste d'expressions : `cible += !expression1 ... !expressionN` ; cette instruction est décrite section **??** page **??**.

La `cible` est une variable ou un champ de structure.

Les instructions `+=`, `-=`, `*=`, `/=` s'appliquent :

- aux types entiers `@sint` (section **??** page **??**, `@sint64` (section **??** page **??**, `@uint` (section **??** page **??** et `@uint64` (section **??** page **??** ;
- au type flottant `@double` (section **??** page **??**).

L'instruction `+=` s'applique aussi au type `@string` (section **??** page **??**).

D'une manière générale, x op= y est équivalent à x = x op y.

54.7.1 Instruction += et le type @string

Sous la forme `cible += expression`, l'instruction permet de concaténer des chaînes de caractères :

```
var s = "abc"
s += "def" # s vaut "abcdef"
```

54.8 Décrémentation -- et &--

L'instruction de décrémentation s'applique aux types `@sint` (page **??**), `@sint64` (page **??**), `@uint` (page **??**), `@uint64` (page **??**) et `@bigint` (page **??**) ; sa syntaxe est la suivante :

```
variable --
```

Les champs de structure peuvent être décrémentés :

```
variable.champ --
```

Ainsi qu'une propriété de l'objet courant :

```
self.champ --
```

Une erreur d'exécution est déclenchée en cas de dépassement de capacité.

L'opérateur `&--` effectue une décrémentation sans déclencher d'erreur en cas de dépassement de capacité :

```
variable &--
```

Les champs de structure peuvent être décrémentés sans déclencher d'erreur en cas de dépassement de capacité :

```
variable.champ &--
```

54.9 L'instruction `drop`

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

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

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

54.10 L'instruction `error`

L'instruction `error` permet de signaler une erreur à l'utilisateur. Elle est constituée de trois champs séparés par un double-point (`:`) :

```
error localisation : message_erreur : variable1, ..., variableN
```

Le champ `localisation` signale à l'utilisateur la position de l'erreur dans le texte source. C'est donc une expression de type `@location`, ou d'un type possédant un *getter* sans argument nommé `location` et renvoyant un objet de type `@location` : c'est le cas des types prédéfinis `@luint`, `@luint64`, `@lsint`, `@lsint64`, `@lbigint`, `@lbool`, `@lchar` et `@lstring`.

Le `message_erreur` est le message affiché à l'utilisateur : c'est donc une expression de type `@string`.

Le troisième champ liste les variables `variable1`, ..., `variableN` qui sont détruites du fait de l'erreur (section **??** page **??**).

Il y a un quatrième champ, optionnel, qui permet de transmettre à l'utilisateur des suggestions de corrections : il commence par le mot réservé `fixit` et est décrit à la section **??** page **??**.

Si il n'y a pas de variable à citer, l'instruction s'écrit :

```
error localisation : message_erreur
```

Par exemple :

```
$identifier$ ?@lstring nom
...
error nom.location : message_erreur
```

Comme `nom` est de type `lstring` (voir ci-dessus), on peut simplement écrire :

```
$identifier$ ?@lstring nom
...
error nom : message_erreur
```

54.10.1 Liste de variables détruites

Lister des variables qui ne peuvent pas être construites est indispensable dans certains cas. Examinons le code suivant (qui ne compile pas) :

```
$identifier$ ?@lstring nom
@unType résultat
if ok then
  résultat = ...
else
  error nom : message_erreur
end # Erreur : 'résultat' est valué par une branche
```

En effet, une des branches du `if` donne une valeur à `résultat`, et l'autre pas. Or, en cas d'erreur, on veut que `résultat` ne soit pas valué à l'exécution. On écrit alors le texte suivant (qui compile) :

```
$identifier$ ?@lstring nom
@unType résultat
if ok then
  résultat = ...
else
  error nom : message_erreur : résultat
end
```

Mentionner `résultat` à la fin de l'instruction `error` permet de faire croire au compilateur que `résultat` est valué.

54.10.2 Clause `fixit`

La clause `fixit` est optionnelle et permet de transmettre à l'utilisateur des suggestions de corrections : il peut apparaître à la fin d'une instruction `error` et d'une instruction `warning` (section **??** page **??**).

Si vous utilisez l'application Cocoa engendrée par GALGAS, la liste des suggestions apparaît au début du menu contextuel `cmd-clic` : il suffit donc d'effectuer un `cmd-clic` sur un token souligné par un trait rouge

(qui signale une erreur) ou orange (une alerte) pour voir apparaître la liste des suggestions. Les suggestions concernent le token désigné par l'expression `localisation` de l'erreur ou de l'alerte.

Sa syntaxe est la suivante :

```
fixit {
  remove
  replace expression
  after expression
  before expression
}
```

La clause `fixit` contient une séquence de suggestions ; il existe quatre types de suggestions :

- `remove` , qui suggère d'éliminer le token désigné ;
- `replace` , qui suggère de remplacer le token désigné par l'`expression` indiquée ;
- `after` , qui suggère d'insérer l'`expression` indiquée après le token désigné ;
- `before` , qui suggère d'insérer l'`expression` indiquée avant le token désigné.

Quatre types sont acceptés pour l'`expression` :

- `@string` , l'expression représente une suggestion unique ;
- `@stringlist` , l'expression représente une liste de suggestions, qui seront présentées à l'utilisateur dans l'ordre de la liste ;
- `@lstringlist` , l'expression représente une liste de suggestions, qui seront présentées à l'utilisateur dans l'ordre de la liste (les information de localisation sont ignorées) ;
- `@stringset` , l'expression représente une liste de suggestions, qui seront présentées à l'utilisateur dans l'ordre alphabétique des chaînes.

Ainsi, on peut écrire (et de même avec `after` et `before`) :

```
fixit { replace "toto" } # Suggère de remplacer le token par toto
fixit { replace "toto" replace "tata" } # Deux suggestions
fixit { replace @stringlist {!"toto" !"tata"} } # Identique au précédent
fixit { replace {!"toto" !"tata"} } # Le type @stringlist est inféré
```

L'`expression` n'est pas limitée aux constructions statiques, mais accepte des expressions calculées à l'exécution : la liste des suggestions peut donc être calculée dynamiquement.

La clause `fixit` contient une séquence de suggestions, qui peut être vide. Une séquence peut contenir plusieurs suggestions `replace` , `after` , `before` ; par contre, la suggestion `remove` ne peut apparaître qu'au plus une fois. Les suggestions sont présentées à l'utilisateur dans l'ordre où elles apparaissent dans la clause `fixit` . Il est parfaitement légal d'écrire par exemple :

```
fixit {
  replace "toto"
  after "tata"
  before "tutu"
  replace "titi"
  remove
}
```

La suite présente plusieurs exemples :

- un premier exemple `remove` sur un identificateur (section **??** page **??**) ;
- un exemple `remove` sur un mot réservé (section **??** page **??**) ;
- un exemple `replace` (section **??** page **??**) ;
- un second exemple `replace`, qui met en évidence un piège qui existe avec des chaînes de caractères (section **??** page **??**).

L'utilisation de `before` et de `after` est analogue à celui de `replace` : on se reportera aux exemples `replace` pour ceux-ci.

54.10.2.1 Premier exemple `fixit remove`

Supposons que votre langage contient une règle de production où un identificateur apparaît :

```
rule <règle> ... {
  ...
  $identifier$ ?let idf
  ...
}
```

Le langage change, l'identificateur n'est plus utile ; sans utiliser la clause `fixit`, on écrit alors :

```
rule <règle> ... {
  ...
  select
    $identifier$ ?let idf
    warning idf : "the identifier is no longer needed: remove it"
  or
  end
  ...

}
```

Et en utilisant la clause `fixit` :

```
rule <règle> ... {
  ...
  select
    $identifier$ ?let idf
    warning idf : "the identifier is no longer needed" fixit { remove }
  or
  end
  ...

}
```

Dans l'application Cocoa, le menu contextuel activé sur l'identificateur présentera la suggestion de supprimer l'identificateur.

54.10.2.2 Second exemple `fixit remove`

Reprenons l'exemple précédent, mais en considérant que c'est un mot réservé qui devient inutile ; on écrit :

```
rule <règle> ... {
  ...
  select
    $mot-réservé$
    warning .here : "this keyword is no longer needed" fixit { remove }
  or
  end
  ...

}
```

La différence est que l'on ne dispose pas d'une localisation explicite de l'alerte : on est obligé d'utiliser le *constructeur* `here` *du type* `@location` *– page* **??** ou le *constructeur* `next` *du type* `@location` *– page* **??**. Utiliser à bon escient ces constructeurs est indispensable pour bien localiser le signalement.

Dans l'exemple ci-dessus, on utilise `here`, qui désigne le token qui précède, c'est-à-dire `$mot-réservé$`. On peut utiliser `next`, qui désigne le token suivant, en écrivant :

```
rule <règle> ... {
  ...
  select
    let loc = @location.next
    $mot-réservé$
    warning loc : "this keyword is no longer needed" fixit { remove }
  or
  end
```

```
  ...

}
```

54.10.2.3 Exemple `fixit replace`

Supposons qu'un identificateur ne puisse prendre qu'un ensemble de valeurs possibles. C'est le rôle des écritures sémantiques de vérifier que la valeur est l'une de celles autorisées.

Ainsi, la règle syntaxique peut être :

```
rule <règle> ... {
  ...
  $identifier$ ?let valeurÀVérifier
  ...
}
```

Maintenant, la sémantique. Sans clause `fixit`, elle a l'allure suivante :

```
  let @stringset valeursPossibles = ...
  if [valeursPossibles hasKey !valeurÀVérifier] then
    ...
  else
    error valeurÀVérifier : "valeur invalide"
  end
}
```

On pourrait améliorer le message d'erreur ci-dessus en ajoutant la liste des valeurs possibles. Ajouter une clause `fixit replace` permet de le faire très simplement. Après le mot réservé `replace` est attendue une expression de type `@string` ou `@stringlist`, or la variable `valeursPossibles` est du type `@stringset` : on utilise donc le getter *getter* `stringList` *du type* `@stringset` *(page* **??***)* . D'où :

```
  let @stringset valeursPossibles = ...
  if [valeursPossibles hasKey !valeurÀVérifier] then
    ...
  else
    error valeurÀVérifier : "valeur invalide" fixit {
      replace [valeursPossibles stringList]
    }
  end
}
```

54.10.2.4 Exemple `fixit replace`, chaîne de caractères

Reprenons l'exemple précédent, mais en supposant `valeurÀVérifier` est obtenue à partir d'une chaîne de caractères.

```
rule <règle> ... {
  ...
  $literal_string$ ?let valeurÀVérifier
  ...
}
```

On pourrait que le fait que le terminal soit une chaîne plutôt qu'un identificateur ne change rien. Mais lorsque le terminal est une chaîne, la valeur `valeurÀVérifier` ne contient pas les « "» qui délimitent la constante chaîne de caractères. Si l'on fait le remplacement tel quel, les valeurs suggérées n'ont pas ces délimiteurs : lors du remplacement, ils sont perdus. Aussi, il faut explicitement les rajouter avant de soumettre les suggestions :

```
  let @stringset valeursPossibles = ...
  if [valeursPossibles hasKey !valeurÀVérifier] then
    ...
  else
    @stringlist suggestions = {}
    for (s) in valeursPossibles do
      suggestions += !"\"" + s + "\""
    end
    error valeurÀVérifier : "valeur invalide" fixit {
      replace suggestions
    }
  end
}
```

54.11 L'appel de procédure

Cette instruction permet d'exécuter une procédure. Si par exemple celle-ci est définie par :

```
proc maRoutine !@uint a ?!@string b {
  ...
}
```

L'instruction d'appel de cette routine est (il y a plusieurs variantes possibles pour le premier paramètre qui est en entrée) :

```
@string x = ...
maRoutine (?@uint y !?x)
```

Note : les parenthèses sont obligatoires, même si il n'y a aucun argment.

La correspondance entre arguments formels et paramètres effectifs est décrite à la section **??** page **??**.

54.12 L'instruction for

L'instruction `for` permet d'énumérer :

- une collection ;
- plusieurs collections de manière synchrone.

Pour énumérer une collection, la syntaxe est la suivante :

```
for enumeration_collection
while condition # Optionnel
before instructions_before  # Optionnel
do
  (nom_index) # Optionnel
  instructions_do
between instructions_between  # Optionnel
after instructions_after  # Optionnel
end
```

Énumérer plusieurs collections s'exprime en séparant les différentes énumérations par une virgule :

```
for enumeration_collection1, enumeration_collection2, ...
while condition # Optionnel
before instructions_before  # Optionnel
do
  (nom_index) # Optionnel
  instructions_do
between instructions_between  # Optionnel
after instructions_after  # Optionnel
end
```

54.12.1 Les quatre formes d'une énumération

Le tableau **??** liste les quatre façons d'exprimer l'énumération `enumeration_collection` .

54.12.2 Types énumérables et ordre d'énumération

Les types pouvant être énumérés sont listés dans le tableau **??**, ainsi que leur ordre d'énumération par défaut. Si le champ `sens` est vide, c'est l'ordre par défaut qui est adopté. Utiliser `>` fixe le sens inverse.

Énumération	Signification
`sens () in expression`	Utilisation de constantes définies implicitement qui représentent les champs de l'élément courant (section **??** page **??**).
`sens () prefixe in expression`	Utilisation de constantes préfixées définies implicitement qui représentent les champs de l'élément courant (section **??** page **??**).
`sens cst in expression`	Déclaration d'une constante représentant l'élément courant (section **??** page **??**).
`sens (cst1 cst2 ...) in expression`	Déclaration de constantes représentant les champs de l'élément courant (section **??** page **??**).

Tableau 54.1 – *Les quatre formes d'énumération de l'instruction `for`*

Type	Ordre d'énumération
`@data`	Ordre croissant des indices
`dict @T`	Ordre alphabétique des clés
`list @T`	Ordre croissant des indices
`map @T`	Ordre alphabétique des clés
`listmap @T`	Ordre alphabétique des clés
`sortedlist @T`	Ordre du tri
`@stringset`	Ordre alphabétique

Tableau 54.2 – *Types énumérables par l'instruction `for`*

54.12.3 Énumération « `() in expression` »

Des constantes correspondants à chaque champ de l'élément courant sont implicitement déclarées (tableau **??**).

Voici quelques exemples :

```
@data d = ...
for () in d do
  log data
end
```

```
@stringset v = ...
for () in v do
  log key # Affichage des clés dans l'ordre alphabétique
end
```

Avec la liste :

```
list @maListe {
  @uint mProperty1
```

Type	Constantes implicitement déclarées
`@data`	`data`, de type `@uint`
`dict @T`	`key` représente la clé, et à chaque champ de la table, correspond une constante de même nom.
`list @T`	À chaque champ de la liste, correspond une constante de même nom.
`map @T`	`lkey`, de type `@lstring`, qui représente la clé, et à chaque champ de la table, correspond une constante de même nom.
`listmap @T`	`key`, de type `@string`, qui représente la clé, et `mList`, qui représente la liste associée.
`sortedlist @T`	À chaque champ de la liste, correspond une constante de même nom.
`@stringset`	`key`, de type `@string`

Tableau 54.3 – *Constantes implicitement déclarées par «`() in expression`»*

```
  @string mProperty2
}
```

On peut écrire :

```
@maListe lst = ...
for () in lst do
  log mProperty1, mProperty2
end
```

Avec la table :

```
map @maTable {
  @uint mProperty1
  @string mProperty2
}
```

On peut écrire :

```
@maTable tab = ...
for () in tab do
  log lkey, mProperty1, mProperty2
end
```

54.12.4 Énumération «`() prefix in expression`»

Cette écriture est une extension de celle de la section précédente : `prefix` est utilisé pour préfixer le nom des constantes implicitement déclarées. En reprenant les exemples de la section précédente :

```
@data d = ...
for () xyz_ in d do
  log xyz_data
end
```

```
@stringset v = ...
for () pre in v do
  log prekey # Affichage des clés dans l'ordre alphabétique
end
```

```
@maListe lst = ...
for () lst in lst do
  log lstmProperty1, lstmProperty2
end
```

```
@maTable tab = ...
for () tb_ in tab do
  log tb_lkey, tb_mProperty1, tb_mProperty2
end
```

Utiliser un préfixe permet de lever les collisions des noms des constantes implicites quand on énumère des collections ayant des champs de même nom :

```
@maListe v1 = ...
@maListe v2 = ...
for () in v1, () in v2 do # Erreur !
 ...
end
```

Le compilateur GALGAS déclenche une erreur, car il y a ambiguïté sur la signification de `mProperty1` et de `mProperty2` à l'intérieur de la boucle : désigne-t'elle l'élément courant de `v1` ou l'élément courant de `v2` ?

Pour lever l'ambiguïté, on complète l'instruction en précisant un préfixe pour l'une des deux listes (par exemple la seconde) :

```
@maListe v1 = ...
@maListe v2 = ...
for () in v1, () l2_ in v2 do
  log mProperty1 # Désigne sans ambiguïté le champ de la première liste
  log l2_mProperty1 # Désigne sans ambiguïté le champ de la seconde liste
end
```

54.12.5 Énumération « `cst in expression` »

Dans cette forme, une seule constante est déclarée (`cst`), et son type est donné par le tableau **??**. Le type `@T-element` est implicitement déclaré avec la déclaration de la collection correspondante (liste, table), et est une structure : on accède donc à ses champs par l'opérateur `.` .

En reprenant les exemples de la section **??** page **??** :

Type de `expression`	Type implicite de la constante `cst`
`@data`	`@uint`
`dict @T`	`@T-element`
`list @T`	`@T-element`
`map @T`	`@T-element`
`listmap @T`	`@T-element`
`sortedlist @T`	`@T-element`
`@stringset`	`@string`

Tableau 54.4 – *Type de la constante dans «`cst in expression`»*

```
@data d = ...
for v in d do
  log v
end
```

```
@stringset v = ...
for s in v do
  log s
end
```

```
@maListe lst = ...
for x in lst do
  log x.mProperty1, x.mProperty2
end
```

```
@maTable tab = ...
for entry in tab do
  log entry.lkey, entry.mProperty1, entry.mProperty2
end
```

54.12.5.1 Type explicite

On peut annoter le nom de la constante en la faisant précéder par un nom de type. Le compilateur GALGAS vérifie alors l'identité entre le type explicitement déclaré, et le type implicitement déduit du type de l'expression enumérée. Il est possible de déclarer explicitement le type de la constante en écrivant l'énumération sous la forme :

```
@unType cst in expression
```

Les exemples précédents deviennent alors :

```
@data d = ...
for @uint v in d do
  log v
```

Type	Constantes à déclarer
`@data`	*Ce type n'est pas pris en charge par cette forme.*
`dict @T`	Une constante de type de la clé, suivi d'une constante pour chaque champ de la table, dans l'ordre de déclaration.
`list @T`	Une constante pour chaque champ, dans l'ordre de déclaration.
`map @T`	Une constante de type `@lstring`, qui représente la clé, suivi d'une constante pour chaque champ de la table, dans l'ordre de déclaration.
`listmap @T`	Une constante de type `@string`, qui représente la clé, et une constante qui représente la liste associée.
`sortedlist @T`	Une constante pour chaque champ, dans l'ordre de déclaration.
`@stringset`	*Ce type n'est pas pris en charge par cette forme.*

Tableau 54.5 – *Constantes à déclarer pour «`(cst1 cst2 ...) in expression`»*

```
end
```

```
@stringset v = ...
for @string s in v do
  log s
end
```

```
@maListe lst = ...
for @maListe-element x in lst do
  log x.mProperty1, x.mProperty2
end
```

```
@maListe tab = ...
for @maListe-element entry in tab do
  log entry.lkey, entry.mProperty1, entry.mProperty2
end
```

54.12.6 Énumération «`(cst1 cst2 ...) in expression`»

Le tableau **??** liste pour chaque type le nombre et la signification des constantes qui doivent être déclarées.

Prenons comme exemple le type liste suivant :

```
list @maListe {
  @uint mProperty1
  @string mProperty2
  @char mProperty3
  @bool mProperty4
}
```

L'énumération s'écrit :

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (p1 p2 p3 p4) in uneListe do
  log p1, p2, p3, p4
end
```

Plusieurs variantes sont possibles, et sont décrites ci-après.

54.12.6.1 Type explicite

Il est possible d'annoter chaque constante en précisant son type.

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (@uint p1 @string p2 @char p3 @bool p4) in uneListe do
  log p1, p2, p3, p4
end
```

Le compilateur vérifie alors que le type cité est égal au type déduit du type de l'expression énumérée.

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (@uint p1 @string p2 @char p3 @bool p4) in uneListe do
  log p1, p2, p3, p4
end
```

54.12.6.2 Joker

Si certaines constantes ne sont pas utiles, on peut les remplacer par un joker (`*`). Le nom du type ne doit alors pas figurer.

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (@uint p1 * * @bool p4) in uneListe do
  log p1, p4
end
```

Plusieurs jokers peuvent être rassemblés en mentionnant leur nombre d'occurrences.

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (@uint p1 2* @bool p4) in uneListe do
  log p1, p4
end
```

54.12.6.3 Points de suspension

Trois points consécutifs (`...`) peuvent être utilisés pour signifier que les dernières constantes ne sont pas utiles.

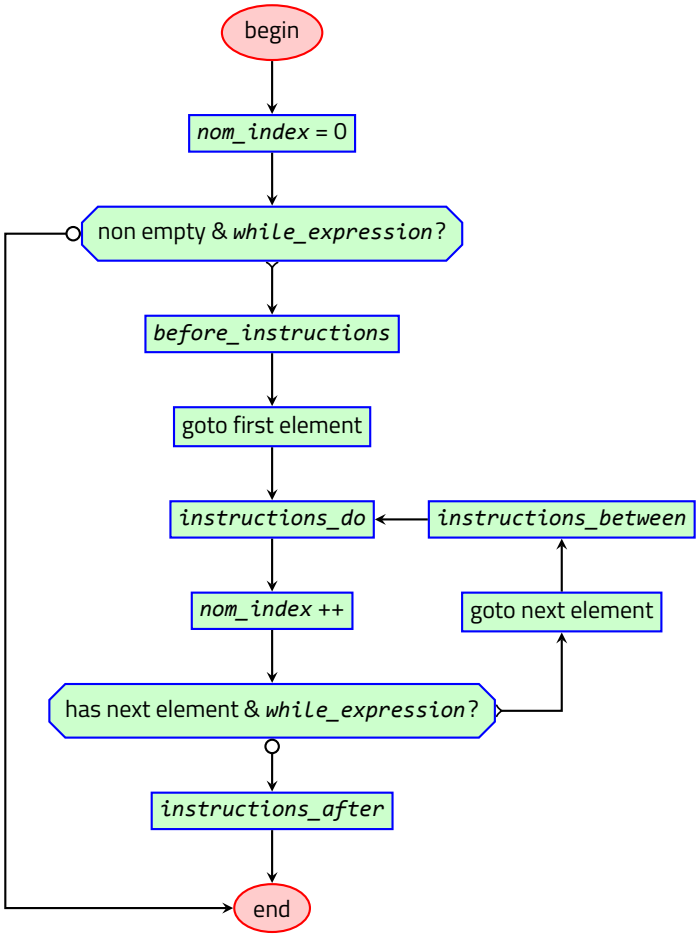


Figure 54.1 – *Organigramme d'exécution d'une instruction* `for`

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (@uint p1 ...) in uneListe do
  log p1
end
```

Et si aucune constante n'est utile, on écrit :

```
@maListe uneListe = {!1 !"a" !'b' !false}
for (...) in uneListe do

end
```

54.12.7 Organigramme d'exécution

L'organigramme illustrant l'exécution de l'instruction `for` est donné à la figure **??**. Si plusieurs collections sont énumérées, l'instructions se termine dès que la collection la moins nombreuse est complètement enumérée.

54.12.8 Champs optionnels

Plusieurs champs de l'instruction `for` sont optionnels.

`sens` . Ce champ peut prendre trois valeurs, et fixe l'ordre dans lequel les éléments sont énumérés :

- si le champ est vide, dans l'ordre indiqué par le tableau **??** ;
- `>` , dans l'ordre inverse à celui indiqué par le tableau **??**.

`(nom_index)` . Vous pouvez mentionner un identificateur entre parenthèses après le mot réservé `do` . Cet identificateur est le nom d'une constante qui a implicitement le type `@uint` et qui est initialisée à 0 avant toute exécution de la boucle, et incrémentée après chaque exécution des `instructions_do` , et avant l'exécution des `instructions_between` . Sa visibilité est la branche `do` .

`while expression` . L'énumération est exécutée tant que l' `expression` est vraie. L'absence de cette construction est équivalent à `while true` et permet d'énumérer toutes les valeurs.

`before instructions_before` . Ces instructions sont exécutées une seule fois, au début de l'exécution de l'instruction. Aucun accès aux objets énumérés n'est possible. Si l'énumération est vide, ces instructions ne sont pas exécutées.

`between instructions_between` . Ces instructions sont exécutées entre deux exécutions consécutives des `instructions_do` . Aucun accès aux objets énumérés n'est possible.

`after instructions_after` . Ces instructions sont exécutées une seule fois, à la fin de l'exécution de l'instruction. Aucun accès aux objets énumérés n'est possible. Si l'énumération est vide, ces instructions ne sont pas exécutées.

54.12.9 Modification de la collection

Au début de l'exécution de l'instruction `for` , les valeurs des collections enumérées sont capturées et mémorisées. L'énumération s'effectue sur ces valeurs mémorisées. Aussi, il est possible de modifier la collection en cours d'énumération sans que cela affecte l'exécution :

```
@stringlist v = {!"A", !"B", !"C"}
log v # "A", "B", "C"
for s in v do
  v += !s
end for
log v # "A", "B", "C", "A", "B", "C"
```

54.13 Incrémentation ++ et &++

L'instruction d'incrémentation s'applique aux types `@sint` (page **??**), `@sint64` (page **??**), `@uint` (page **??**) et `uint64` ; sa syntaxe est la suivante :

```
variable ++
```

Les champs de structure peuvent être incrémentés :

```
variable.champ ++
```

Ainsi qu'une propriété de l'objet courant :

```
self.champ ++
```

Une erreur d'exécution est déclenchée en cas de dépassement de capacité.

L'opérateur `&++` effectue une incrémentation sans déclencher d'erreur en cas de dépassement de capacité :

```
variable &++
```

Les champs de structure peuvent être incrémentés sans déclencher d'erreur en cas de dépassement de capacité :

```
variable.champ &++
```

54.14 L'instruction `if`

Dans sa forme la plus générale, l'instruction `if` a la syntaxe suivante :

```
if condition then
  instructions
elsif condition2 then
  instructions2
...
else
  instructions_else
end
```

Plus précisement, elle contient :

- zéro, une ou plusieurs branches `elsif` ;
- zéro ou une branche `else` .

Aucune branche `else` est équivalent à une branche `else` sans aucune instruction.

Les branches `elsif` sont simplement du sucre syntaxique : il est sémantiquement équivalent d'utiliser des instructions `if` imbriquées. Par exemple :

```
if condition then
  instructions
elsif condition2 then
```

```
  instructions2
else
  instructions_else
end
```

est équivalent à :

```
if condition then
  instructions
else
  if condition2 then
    instructions2
  else
    instructions_else
  end
end
```

Le langage permet plusieurs aux conditions notées `condition` et `condition2` dans les exemples ci-dessus :

- soit une simple expression ;
- soit une affectation conditionnelle ;
- soit une liste de simples expressions et d'affectations conditionnelles.

54.14.1 Simple expression

Le langage permet que le type des `condition` et `condition2` soit différent du type `@bool` (page **??**), sous certaines conditions. La règle complète est que le type des `condition` et `condition2` est :

- soit le type `@bool` ;
- soit un type *structure*, possèdant une propriété nommée `bool` , dont le type est `@bool` ;
- soit un type possédant un *getter* sans argument nommé `bool` et renvoyant une valeur de type `@bool` .

Voici un exemple illustrant le deuxième cas ; le type `@bool` (page **??**) est une structure possèdant une propriété nommée `bool` , dont le type est `@bool` . Aussi, écrire :

```
@lbool variable = ...
if variable then
  instructions
else
```

```
  else_instructions
end
```

est équivalent à :

```
@lbool variable = ...
if variable.bool then
  instructions
else
  else_instructions
end
```

Pour illustrer le troisième cas, on prend l'exemple de la classe suivante :

```
class @myClass { ... }

getter @myClass bool -> @bool outResult { ... }
```

Ainsi, on peut écrire :

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

Il est équivalent d'écrire :

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

54.14.2 L'affectation conditionnelle

L'affectation conditionnelle a la syntaxe suivante :

```
if let cible = expression as @T then
  then_instructions # cible est défini, du type @T
else
  else_instructions # cible n'est pas défini
end
```

Si l'`expression` est de type `@T`, alors la condition est vraie et les `then_instructions` sont exécutées. La constante `cible` est définie dans ces instructions, elle a pour type `@tT` et pour valeur celle de l'`expression`.

Si l'`expression` n'est pas de type `@T`, alors la condition est fausse et les `else_instructions` sont exécutées. La constante `cible` n'est pas définie dans ces instructions.

L'affectation conditionnelle permet de faire une affectation polymorphique inverse de manière propre, sans utiliser l'*expression de conversion polymorphique inverse* (`expression as @T`, section **??** page **??**).

54.14.3 Liste de simples expressions et d'affectations conditionnelles

La condition d'une instruction `if` peut être une liste simples expressions et d'affectations conditionnelles, le séparateur étant la virgule `,`. L'évaluation s'effectue de gauche à droite, sous la forme d'un court-circuit :

- dès qu'une condition ou affectation conditionnelle n'est pas vraie (c'est-à-dire fausse ou non construite), les éléments suivants ne sont pas évalués et la condition est fausse ;
- la condition est vraie si les évaluations de gauche à droite de toutes les simples expressions et affectations conditionnelles sont vraies.

Le résultat d'une affectation conditionnelle peut être utilisé dans les éléments suivants :

```
if let cible = exp as @T, let cible2 = [cible unGetter] as @U then
  then_instructions # cible et cible2 sont définis
else
  else_instructions # Ni cible ni cible2 ne sont définis
end
```

54.15 L'instruction `grammar`

L'instruction `grammar` permet d'exécuter l'analyse d'un texte par une grammaire. Le texte peut être contenu :

- dans un fichier (section **??** page **??**) ;
- dans une chaîne de caractères (section **??** page **??**).

54.15.1 Texte source dans un fichier

```
grammar
  nom_grammaire
  label_grammaire # Optionnel
  (liste_parametres)
  in expression # Chemin du fichier source
  traduction_dirigee_par_la_syntaxe # Optionnel
```

Le mot réservé `in` caractérise la localisation de la chaîne source dans un fichier.

- `nom_grammaire` est un identificateur nommant la grammaire, c'est le nom d'un composant grammaire du projet ;
- `label_grammaire` est optionnel, et permet d'exécuter une variante des règles de production (voir section **??** page **??**) ;
- `liste_parametres` est une liste de paramètres effectifs (en entrée, sortie, ou sortie/entrée), en accord avec la liste des arguments formels de l'axiome de la grammaire ;
- `expression` est une valeur de type `@lstring`, dont le champ `string` désigne un fichier source, par un chemin relatif ou absolu, et dont le champ `location` est la position de signalement d'erreur si le fichier source ne peut pas être lu ;
- `traduction_dirigee_par_la_syntaxe` est optionnel, et permet d'obtenir la chaîne source traduite lors d'une *traduction dirigée par la syntaxe* (voir section **??** page **??**).

Prenons l'exemple d'un composant grammaire :

```
grammar maGrammaire "SLR" {
  syntax ...
   <start_symbol> (?!@declarationAST ioDeclarations)
}
```

L'instruction grammaire s'écrit alors :

```
grammar maGrammaire (!?ioDeclarations) in fichierSource
```

Cette instruction est typiquement utilisé dans une règle d'analyse de fichier source (chapitre **??** à partir de la page **??**) :

```
case . "monExtension"
message "un fichier source"
grammar maGrammaire
?sourceFilePath: @lstring inSourceFile {
  var declaration = @declarationList {}
  grammar maGrammaire (!?ioDeclarations) in inSourceFile
  ...
}
```

54.15.2 Texte source dans une chaîne de caractères

```
grammar
  nom_grammaire
  label_grammaire # Optionnel
  (liste_parametres)
  on chaine_source : nom_associe
  traduction_dirigee_par_la_syntaxe # Optionnel
```

Le mot réservé `on` caractérise la localisation de la chaîne source dans une chaîne de caractères.

- `nom_grammaire` est un identificateur nommant la grammaire, c'est le nom d'un composant grammaire du projet ;
- `label_grammaire` est optionnel, et permet d'exécuter une variante des règles de production (voir section **??** page **??**) ;
- `liste_parametres` est une liste de paramètres effectifs (en entrée, sortie, ou sortie/entrée), en accord avec la liste des arguments formels de l'axiome de la grammaire ;
- `chaine_source` est une expression de type `@string`, qui contient directement la chaîne source à analyser ;
- `nom_associe` est une expression de type `@string`, qui est utilisé lorsque `@location.here` est appelé : `[@location.here file]` renvoie la valeur transmise dans `nom_associe` ;
- `traduction_dirigee_par_la_syntaxe` est optionnel, et permet d'obtenir la chaîne source traduite lors d'une *traduction dirigée par la syntaxe* (voir section **??** page **??**).

La chaîne `nom_associe` sert lorsque qu'une erreur est détectée lors de l'analyse de `chaine_source`. On a alors un message d'erreur dont la première ligne est :

`nom_associe:ligne:colonne:`

Ce message est alors similaire au message que l'on obtient lors que l'on analyse un fichier, où le message dont la première ligne est :

`nom_fichier_source:ligne:colonne:`

54.16 L'instruction `log`

L'instruction `log` permet d'afficher le détail de la valeur d'une variable, d'une constante ou d'une expression :

- pour une variable ou une constante, `log nom` ;
- pour une expression, `log "message" : expression` ;

Par exemple :

```
let x = 2
log x # Affiche LOGGING x: <@uint:2>
log "valeur" : x * 2 # Affiche LOGGING valeur: <@uint:4>
```

Plusieurs variables ou constantes peuvent être affichées par une même instruction `log`, en les séparant par une virgule :

```
let x = 2
log x, "valeur" : x * 2
```

54.17 L'instruction loop

L'instruction `loop` a la syntaxe suivante :

```
loop (variant_expression)
  instructions_1
while expression do
  instructions_2
end
```

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

Le `variant_expression` est une expression de type `@uint` qui assure que la boucle n'est pas sans fin : elle est calculée au début de l'exécution de l'instruction, et décrémentée après chaque itération. Si sa valeur atteint zéro, une erreur d'exécution est déclenchée.

L' `expression` est une expression de type `@bool` qui exprime la continuation de l'exécution de la boucle.

L'exécution de l'instruction `loop` est illustrée par l'organigramme de la figure **??** page **??**.

54.18 L'instruction d'appel de procédure

La syntaxe de l'appel d'une procédure est :

```
nom_procedure (liste_parametres_effectifs)
```

Les parenthèses sont obligatoires. La déclaration d'un procédure est présentée à la section **??** page **??**.

`nom_procedure` est le nom de la procédure, et `liste_parametres_effectifs` est la liste des paramètres effectifs de l'appel, en accord avec l'en-tête de la procédure.

Par exemple, la procedure suivante :

```
proc produit ?@uint a ?@uint b !@uint résultat {
  résultat = a * b
}
```

Peut être appelée par :

```
produit (!2 !3 ?@uint résultat)
```

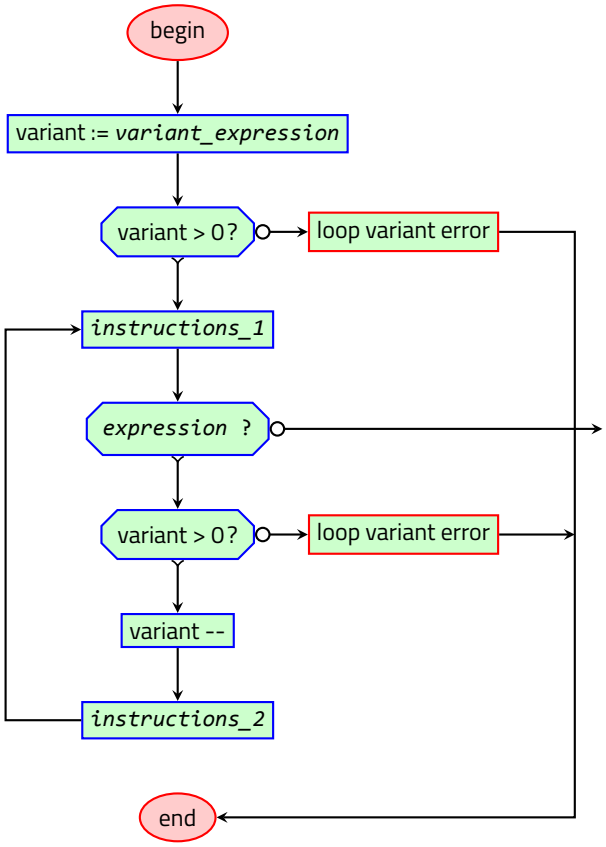


Figure 54.2 – *Organigramme d'exécution d'une instruction Loop*

54.19 L'instruction d'appel de méthode

En GALGAS, une *méthode* est un sous-programme qui s'applique à un objet, et qui ne modifie pas cet objet. La syntaxe de l'appel d'une méthode est :

```
[expression nom_methode liste_parametres_effectifs]
```

La valeur d'`expression` est l'objet sur lequel la méthode est appelée, `nom_methode` est le nom de la méthode, et `liste_parametres_effectifs` est la liste des paramètres effectifs, en accord avec l'en-tête de la méthode.

Avec l'option `-T`, un fichier HTML qui contient les caractéristiques de tous les types d'un projet est engendré dans le répertoire `build/helpers`. Ainsi, les en-têtes de toutes les méthodes sont listés. Par exemple, pour le type `@string`, la méthode `writeToExecutableFileWhenDifferentContents` est présentée comme suit :

```
method writeToExecutableFileWhenDifferentContents
  ?@string inFilePath
  !@bool outFileModified
```

Aussi, cette méthode peut être appelée par :

```
@string contents = ...
@string filePath = ...
[contents writeToExecutableFileWhenDifferentContents
  !filePath
  ?@bool fileChanged
]
```

54.20 L'instruction d'appel de procédure de classe

En GALGAS, une *procédure de classe* est est une procédure définie dans un type. Au contraire d'une méthode, elle ne s'applique pas à un objet. La syntaxe de l'appel d'une procédure de classe est :

```
[@T nom_procedure liste_parametres_effectifs]
```

`nom_procedure` est le nom de la procédure, et `liste_parametres_effectifs` est la liste des paramètres effectifs, en accord avec l'en-tête de la procédure.

Avec l'option `-T`, un fichier HTML qui contient les caractéristiques de tous les types d'un projet est engendré dans le répertoire `build/helpers`. Ainsi, les en-têtes de toutes les méthodes sont listés. Par exemple, pour le type `@string`, la procédure de classe `deleteFile` est présentée comme suit :

```
proc @string deleteFile ?@string inFilePath
```

Aussi, elle peut être appelée par :

```
@string filePath = ...
[@string deleteFile !filePath]
```

54.21 L'instruction d'appel de *setter*

En GALGAS, un *setter* est un sous-programme qui s'applique à un objet, et qui peut modifier cet objet. L'instruction d'appel accepte deux formes différentes.

54.21.1 Appel simple

La première syntaxe de l'appel d'un *setter* est :

```
[!?cible nom_setter liste_parametres_effectifs]
```

Le délimiteur `!?` devant `cible` permet de distinguer syntaxiquement un appel de *setter* d'un appel de méthode. `cible` désigne l'objet sur lequel le *setter* est appelé, et peut être une variable, ou le champ d'une variable (`variable.champ`), ou le champ d'un champ d'une variable (`variable.champ.champ2`),

... `nom_setter` est le nom du *setter*, et `liste_parametres_effectifs` est la liste des paramètres effectifs, en accord avec l'en-tête du *setter*.

Avec l'option `-T`, un fichier HTML qui contient les caractéristiques de tous les types d'un projet est engendré dans le répertoire `build/helpers`. Ainsi, les en-têtes de toutes les *setters* sont listées. Par exemple, pour le type `@string`, le *setter* `setCharacterAtIndex` est présenté comme suit :

```
setter setCharacterAtIndex
  ?@char inChar
  ?@uint inIndex
```

Aussi, ce *setter* peut être appelé par :

```
@string s = ...
[!?s setCharacterAtIndex !'a' !4]
```

54.21.2 Appel avec conversion de type

La seconde syntaxe de l'appel d'un *setter* est :

```
[!?cible as @T nom_setter liste_parametres_effectifs]
```

Statiquement, le type `@T` doit être un type héritier du type statique de `cible`.

À l'exécution, si le type dynamique est le type `@T` ou un de ses héritiers, l'instruction est exécutée. Sinon, une erreur d'exécution a lieu et le setter n'est pas appelé.

Par exemple, on considère les déclarations de classe et le setter suivant :

```
class @a { }

class @b : @a { }

setter @b aSetter { }
```

Si on écrit :

```
@a unObjet = ...
[!?unObjet aSetter] # Erreur sémantique
```

L'instruction `[!?unObjet aSetter]` donne lieu à une erreur sémantique, puisque la classe `@a` ne définit pas le setter `aSetter`.

L'écriture suivante est acceptée par le compilateur car la classe `@b` définit le setter `aSetter` :

```
@a unObjet = ...
[!?unObjet as @b aSetter] # Compilation ok
```

À l'exécution, le comportement dépend du type dynamique de `unObjet`. Si celui-ci est une instance de `@a`, une erreur d'exécution est déclenchée :

```
@a unObjet = @a.new
[!?unObjet as @b aSetter] # Compilation ok, erreur d'exécution
```

Par contre, si `unObject` est une instance de `@b`, l'appel du setter d'effectue :

```
@a unObjet = @b.new
[!?unObjet as @b aSetter] # Compilation ok, appel du setter
```

54.22 L'instruction `switch`

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

```
switch expression
case constante, constante, ... :
  liste_instructions
case constante, constante, ... :
  liste_instructions
...
end
```

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

Par exemple, avec la déclaration :

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

On peut écrire :

```
@feuTricolore feu = ...

switch feu
case vert, orange:
  ...
case rouge :
  ...
end
```

Si des constantes déclarées dans l'énumération ont des valeurs associées, alors les branches `case` nommant ces constantes doivent adopter une syntaxe particulière.

En prenant pour exemple une constante possédant deux valeurs associées, la forme la plus générale est :

```
switch expression
case constante (@type1 nom1 @type2 nom2) :
...
end
```

`nom1` et `nom2` sont des constantes qui reçoivent les valeurs associées.

Si on n'est pas intéressé par une valeur, on peut substituer `*` au nom :

```
switch expression
case constante (@type1 nom1 @type2 *) :
...
end
```

De même, l'annotation du type est optionnel : les types `@type1`, `@type2` peuvent être déduits de la déclaration de la valeur associée :

```
switch expression
case constante (nom1 *) :
...
end
```

Ainsi, si l'on n'est pas intéressé par les valeurs associées, on peut écrire :

```
switch expression
case constante (* *) :
...
end
```

Ou aussi :

```
switch expression
case constante (2*) :
...
end
```

Ou utiliser le mot clé `unused` (avec ou sans annotation de type) :

```
switch expression
case constante (@type1 unused nom1 unused nom2) :
...
end
```

Enfin, on peut mentionner dans la même branche `case` plusieurs constantes déclarant des valeurs associées, à la condition que ces valeurs associées soient de même nombre et de même type. Par exemple :

```
enum @erreur {
  case ok
  case erreur1 (@string unMessage)
  case erreur2 (@string autreMessage)
```

```
}
```

On peut écrire l'instruction `switch` correspondante :

```
@erreur erreur = ...

switch erreur
case ok:
  ...
case erreur1, erreur2 (@string m) :
  ...
end
```

54.23 L'instruction `warning`

L'instruction `warning` permet de signaler une alerte à l'utilisateur. Elle est constituée de deux champs séparés par un double-point (`:`) :

```
warning localisation : message_alerte
```

Le champ `localisation` signale à l'utilisateur la position de l'erreur dans le texte source. C'est donc une expression de type `@location`, ou d'un type possédant un *getter* sans argument nommé `location` et renvoyant un objet de type `@location` : c'est le cas dses types prédéfinis `@luint`, `@luint64`, `@lsint`, `@lsint64`, `@lbigint`, `@lbool`, `@lchar` et `@lstring`.

Le `message_alerte` est le message affiché à l'utilisateur : c'est donc une expression de type `@string`.

Il y a un troisième champ, optionnel, qui permet de transmettre à l'utilisateur des suggestions de corrections : il commence par le mot réservé `fixit` et est décrit à la section **??** page **??**.

Exemple d'instruction `warning` :

```
$identifier$ ?@lstring nom
...
warning nom.location : message_alerte
```

Comme `nom` est de type `@lstring` (voir ci-dessus), on peut simplement écrire :

```
$identifier$ ?@lstring nom
...
warning nom : message_alerte
```

54.24 L'instruction `with`

L'instruction `with` permet d'associer un test de recherche dans une table et l'accès aux champs correspondants si succès. Elle peut prendre quatre formes différentes, suivi que l'on veuille modifier la table ou

non, et suivant que l'on veut tolérer l'échec de la recherche ou non.

Première forme. Accès en lecture, tolérance de l'échec de la recherche (section **??** page **??**) :

```
with expression_cle prefixe_optionnel in expression_table
do
  liste_instructions_do
else
  liste_instructions_else # Optionnel
end
```

Deuxième forme. Accès en lecture, signalement d'erreur si échec de la recherche (section **??** page **??**) :

```
with expression_cle prefixe_optionnel in expression_table
error message methode_recherche
do
  liste_instructions_do
end
```

Troisième forme. Accès en lecture/écriture, tolérance de l'échec de la recherche (section **??** page **??**) :

```
with expression_cle prefixe_optionnel in !?cible_table
do
  liste_instructions_do
else
  liste_instructions_else # Optionnel
end
```

Quatrième forme. Accès en lecture/écriture, signalement d'erreur si échec de la recherche (section **??** page **??**) :

```
with expression_cle prefixe_optionnel in !?cible_table
error message methode_recherche
do
  liste_instructions_do
end
```

54.24.1 Accès en lecture tolérant l'échec de la recherche

```
with expression_cle prefixe_optionnel in expression_table
do
  liste_instructions_do
else
  liste_instructions_else # Optionnel
end
```

Où :

- `expression_cle` est une expression de type `@string` dont la valeur définit la clé ;

- `prefixe_optionnel` est soit vide, soit est constitué d'un double-point `:` suivi d'un identificateur qui préfixe les noms des champs dans la liste d'instructions `liste_instructions_do` ;
- `expression_table` est une expression qui désigne la table.

La clé désignée par la valeur de `expression_cle` est recherchée dans la table désignée par la valeur de `expression_table` :

- en cas d'échec, les instructions `liste_instructions_else` sont exécutées ;
- en cas de succès, ce sont les instructions `liste_instructions_do` qui sont exécutées ; les propriétés de l'élément recherché sont alors disponibles en lecture, sous leur nom éventuellement préfixé ; la clé est disponible sous le nom `lkey` éventuellement préfixé, et est de type `@lstring` .

Par exemple, on veut disposer d'une table possédant une propriété identifiant de manière unique la clé. On déclare :

```
map @maTable {
  @uint mIndex
  insert insertKey error message "entree deja presente"
}
```

Et on effectue des recherches / insertions de la façon suivante :

```
@uint idx
@lstring cle = ...
with cle.string in table do
  idx = mIndex
else
  idx = [table count]
  [!?table insertKey !cle !idx]
end
```

En utilisant un préfixe, le code devient :

```
@uint idx
@lstring cle = ...
with cle.string : xyz_ in table do
  idx = xyz_mIndex
else
  idx = [table count]
  [!?table insertKey !cle !idx]
end
```

54.24.2 Accès en lecture, signalement d'erreur si échec de la recherche

```
with expression_cle prefixe_optionnel in expression_table
error message methode_recherche
do
  liste_instructions_do
end
```

Où :

- `expression_cle` est une expression de type `@lstring` dont la valeur définit la clé ;
- `prefixe_optionnel` est soit vide, soit est constitué d'un double-point `:` suivi d'un identificateur qui préfixe les noms des champs dans la liste d'instructions `liste_instructions_do` ;
- `expression_table` est une expression qui désigne la table ;
- `methode_recherche` est une méthode de recherche de la table (c'est-à-dire déclarée par `search`) dont le message associé sera utilisé pour le signalement d'erreur.

La clé désignée par la valeur de `expression_cle` est recherchée dans la table désignée par la valeur de `expression_table` :

- en cas d'échec, une erreur est affichée, son message étant fourni par la méthode de recherche `methode_recherche`, et sa localisation par le champ `location` de l'`expression_cle` ;
- en cas de succès, ce sont les instructions `liste_instructions_do` qui sont exécutées ; les propriétés de l'élément recherché sont alors disponibles en lecture, sous leur nom éventuellement préfixé ; la clé est disponible sous le nom `lkey` éventuellement préfixé, et est de type `@lstring`.

Par exemple, on veut disposer d'une table qui implémente un *counted set*, c'est à dire que l'on associe à la clé son nombre de citations, et la doit avoir été entrée auparavant. On déclare :

```
map @maTable {
  @uint mOccurrenceCount
  search searchKey error message "entree %K absente"
}
```

Et on effectue des recherches de la façon suivante :

```
@lstring cle = ...
@uint occurenceCount
with cle in table error message searchKey do
  occurenceCount = mOccurrenceCount ++
end
```

En utilisant un préfixe, le code devient :

```
@lstring cle = ...
@uint occurenceCount
with cle : abc_ in table error message searchKey do
```

```
  occurenceCount = abc_mOccurrenceCount ++
end
```

54.24.3 Accès en lecture/écriture tolérant l'échec de la recherche

```
with expression_cle prefixe_optionnel in !?cible_table
do
  liste_instructions_do
else
  liste_instructions_else # Optionnel
end
```

Où :

- `expression_cle` est une expression de type `@string` dont la valeur définit la clé ;
- `prefixe_optionnel` est soit vide, soit est constitué d'un double-point `:` suivi d'un identificateur qui préfixe les noms des champs dans la liste d'instructions `liste_instructions_do` ;
- `cible_table` est de la forme `variable`, ou `variable.champ`, ou `variable.champ.champ2`, ... et désigne la table ; la cible est accédée en lecture/écriture.

La clé désignée par la valeur de `expression_cle` est recherchée dans la table désignée par la valeur de `cible_table` :

- en cas d'échec, les instructions `liste_instructions_else` sont exécutées ;
- en cas de succès, ce sont les instructions `liste_instructions_do` qui sont exécutées ; les propriétés de l'élément recherché sont alors disponibles en lecture / écriture, sous leur nom éventuellement préfixé ; la clé est disponible en lecture seule sous le nom `lkey` éventuellement préfixé, est de type `@lstring`.

Par exemple, on veut disposer d'une table qui implémente un *counted set*, c'est à dire que l'on associe à la clé son nombre de citations. On déclare :

```
map @maTable {
  @uint mOccurrenceCount
  insert insertKey error message "entree deja presente"
}
```

Et on effectue des recherches / insertions de la façon suivante :

```
@lstring cle = ...
with cle.string in !?table do
  mOccurrenceCount ++
else
  [!?table insertKey !cle !1]
```

```
end
```

En utilisant un préfixe, le code devient :

```
@lstring cle = ...
with cle.string : xyz_ in !?table do
  xyz_mOccurrenceCount ++
else
  [!?table insertKey !cle !1]
end
```

54.24.4 Accès en lecture/écriture, signalement d'erreur si échec de la recherche

```
with expression_cle prefixe_optionnel in !?cible_table
error message methode_recherche
do
  liste_instructions_do
end
```

Où :

- `expression_cle` est une expression de type `@lstring` dont la valeur définit la clé ;
- `prefixe_optionnel` est soit vide, soit est constitué d'un double-point `:` suivi d'un identificateur qui préfixe les noms des champs dans la liste d'instructions `liste_instructions_do` ;
- `expression_table` est une expression qui désigne la table ;
- `methode_recherche` est une méthode de recherche de la table (c'est-à-dire déclarée par `search`) dont le message associé sera utilisé pour le signalement d'erreur ;
- `cible_table` est de la forme `variable`, ou `variable.champ`, ou `variable.champ.champ2`, ... et désigne la table ; la cible est accédée en lecture/écriture.

La clé désignée par la valeur de `expression_cle` est recherchée dans la table désignée par la valeur de `expression_table` :

- en cas d'échec, une erreur est affichée, son message étant fourni par la méthode de recherche `methode_recherche`, et sa localisation par le champ `location` de l'`expression_cle` ;
- en cas de succès, ce sont les instructions `liste_instructions_do` qui sont exécutées ; les propriétés de l'élément recherché sont alors disponibles en lecture/écriture, sous leur nom éventuellement préfixé ; la clé est disponible en lecture seule sous le nom `lkey` éventuellement préfixé, et est de type `@lstring`.

Par exemple, on veut disposer d'une table qui implémente un *counted set*, c'est à dire que l'on associe à la clé son nombre de citations, et la clé doit avoir été entrée auparavant. On déclare :

```
map @maTable {
  @uint mOccurrenceCount
  search searchKey error message "entree %K absente"
}
```

Et on effectue des recherches de la façon suivante :

```
@lstring cle = ...
with cle in !?table error message searchKey do
  mOccurrenceCount ++
end
```

En utilisant un préfixe, le code devient :

```
@lstring cle = ...
with cle : abc_ in !?table error message searchKey do
  abc_mOccurrenceCount ++
end
```

Chapitre 55

Instructions syntaxiques

Les six instructions décrites dans ce chapitre ne peuvent être utilisées qu'à l'intérieur des règles de production, elles-mêmes obligatoirement placées dans un composant `syntax`.

55.1 Instruction d'appel de terminal

Cette instruction permet de vérifier l'occurrence d'un terminal. Sa syntaxe présente deux options :

```
$terminal$
parametres_entree # Optionnels
indexing nom_index # Optionnel
traduction_dirigee_par_la_syntaxe # Optionnel
```

Où :

- `$terminal$` est le nom du terminal à vérifier ; il doit être l'un des terminaux déclarés par le lexique importé par le composant syntaxique ;
- `parametres_entree` est une liste de zéro, un ou plusieurs paramètres effectifs en entrée, en accord avec la déclaration du `$terminal$` dans le lexique ; comment écrire une liste de paramètres en entrée est présenté à la section **??** page **??** ;
- `nom_index` est le nom d'un index, tel que déclaré dans le lexique ; cet index sert à peupler le menu contextual de cross référence en Cocoa (section **??** page **??**) ;
- `traduction_dirigee_par_la_syntaxe` permet de préciser les options de *traduction dirigée par la syntaxe* ; celle-ci est présentée en détail au chapitre **??** à partir de la page **??**.

55.2 Instruction d'appel de non terminal

55.3 Instruction `select`

La syntaxe de l'instruction **`select`** syntaxique[1] est la suivante :

```
A
select
  I0
or
  I1
or
  I2
end
B
```

L'instruction est présentée avec trois branches **`or`** : l'instruction doit comporter au moins deux branches.

Sa signification est la suivante : l'occurrence de chaque **`select`** syntaxique peut être remplacée par un nouvel non-terminal particulier, que l'on va nommer `T`. La séquence précédente devient donc :

```
A
<T>
B
```

Le non-terminal `T` se dérive de la façon suivante :

```
rule <T> { I0 }

rule <T> { I1 }

rule <T> { I2 }
```

55.4 Instruction `repeat`

La syntaxe de l'instruction **`repeat`** syntaxique[2] est la suivante :

```
A
repeat
  I0
while
```

[1] Ne pas confondre avec l'instruction de sélection lexicale, qui ne peut être utilisée que dans les analyseurs lexicaux (section **??** page **??**).

[2] Ne pas confondre avec l'instruction de répétition lexicale, qui ne peut être utilisée que dans les analyseurs lexicaux (section **??** page **??**).

```
  I1
while
  I2
end
B
```

L'instruction est présentée avec deux branches `while` : l'instruction doit comporter au moins une branche.

Sa signification est la suivante : l'occurrence de chaque `repeat` syntaxique peut être remplacée par un nouvel non-terminal particulier, que l'on va nommer `T`. La séquence précédente devient donc :

```
A
I0
<T>
B
```

Le non-terminal `T` se dérive de la façon suivante :

```
rule <T> { I1 I0 <T> }

rule <T> { I2 I0 <T> }

rule <T> {  }
```

55.5 Instruction parse

55.6 Instruction send

VII

Index