Le langage de commande Gibiane
Description informelle
Revision: 1.23

Christian Queinnec
Université Paris 6
LIP6
Mél: `Christian.Queinnec@acm.org`

Résumé

Gibiane est un langage inventé et utilisé depuis une quinzaine d'années par le Commissariat à l'Énergie Atomique (CEA). Gibiane permet la mise en œuvre de bibliothèques de codes écrits en Fortran, C, etc. Les points d'entrée de ces bibliothèques sont considérés comme des opérateurs dont l'enchaînement est assuré, interactivement ou pas, par Gibiane. Ce rapport technique (correspondant à la première étape du contrat CEA/SAV 24025/CH) décrit ce langage de commande.

Table des matières

1 Généralités

GIBIANE est un langage de commande permettant d'enchaîner des calculs usant d'opérateurs définis par des bibliothèques de sous-programmes spécialisés souvent écrites en Fortran ou en C. À cet effet, il procure une ossature algorithmique classique, une structure de donnée modifiable : la table, enfin, une mise en œuvre syntaxiquement peu contrainte.

Le langage GIBIANE a été initialement conçu pour procurer de très larges possibilités de redéfinition. Ce document décrit la forme initiale, prédéfinie, de ce langage. Il couvre sa syntaxe concrète et décrit de façon informelle la sémantique de ses instructions prédéfinies.

2 Conventions

La syntaxe concrète sera exprimée sous forme de productions. On y distingue les `terminaux` des *non-terminaux*. Les productions comportent un non-terminal suivi du signe ⇒ et d'une définition pouvant comporter des alternatives préfixées par |. Les répétitions quelconques, éventuellement vides, sont notées par une étoile suffixée en exposant, les répétitions non vides par une croix suffixée en exposant.

Lorsque dans le texte, le verbe "devoir" est employé, il signifie qu'une erreur est signalée si la condition évoquée n'est pas remplie. Par exemple, "l'argument de `quitter` doit être une valeur unique de type boucle" signifie qu'une erreur sera signalée si l'expression qui suit l'identificateur réservé `quitter` retourne moins ou plus d'une valeur ou encore que cette unique valeur n'est pas de type boucle. Le comportement d'un programme GIBIANE, après signalisation d'une erreur, n'est pas spécifié et peut donc être quelconque.

3 Valeurs

GIBIANE a comme types de base prédéfinis : les entiers, les réels, les chaînes de caractères, les booléens, les procédures, les boucles, les tables et les types. Les noms de ces types apparaissent en table 1. À l'exception des tables, toutes ces valeurs sont immuables c'est-à-dire qu'elles ne peuvent subir de modifications.

Toutes les valeurs sans exception peuvent être utilisées comme argument de procédure, retournées en résultat, affectées à une variable ou stockées dans une table.

`entier`	`reel`
`chaine`	`logique`
`procedure`	`boucle`
`table`	`type`

Table 1: Noms des types prédéfinis de GIBIANE

À l'exception des types, des tables, des boucles et des procédures qui ne peuvent être obtenues que par calcul, les autres valeurs peuvent apparaître, citées, dans des textes de programmes grâce à des syntaxes prédéfinies. Les types de base prédéfinis sont accessibles comme valeur de variables prédéfinies.

3.1 Entiers

Syntaxe concrète :

entier ⇒ *signe entier-non-signé*
  | *entier-non-signé*
signe ⇒ `+` | `-`
entier-non-signé ⇒ *chiffre*$^+$
chiffre ⇒ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Les entiers sont exprimés en base décimale et sont éventuellement préfixés d'une signe `+` ou `-`. L'intervalle de variation de ces entiers peut être quelconque mais toute implantation doit au moins procurer l'intervalle $-2^{15}+1\ldots 2^{15}-1$.

3.2 Réels

Syntaxe concrète :

réel ⇒ *signe réel-non-signé*
  | *réel-non-signé*
réel-non-signé ⇒ *réel-simple*
  | *réel-simple marque-exposant entier*
réel-simple ⇒ *chiffre*$^+$. *chiffre**
  | *chiffre** . *chiffre*$^+$
marque-exposant ⇒ `e` | `E` | `d` | `D`

Les réels peuvent être préfixés par un signe `+` ou `-` suivis d'une suite de chiffres décimaux éventuellement vide, suivie d'un point suivi d'une suite de chiffres décimaux éventuellement vide. Ces deux séquences de chiffres décimaux ne peuvent être simultanément vides : un point décimal est nécessairement précédé ou suivi d'un chiffre décimal. Cette description peut être suffixée par un exposant décimal débutant par une des lettres `e`, `E`, `d` ou `D` et immédiatement suivi d'un entier éventuellement signé.

Exemples :

`3.14` `-0.0` `-2.78D-6`	*sont des réels légaux*
`3.` `.14` `3.D2` `.2e5`	*sont non moins légaux*
-0D0 *.D0* *-e3*	*sont illégaux*

Le respect d'un standard comme l'IEEE 754 n'est pas exigé.

3.3 Chaînes

Syntaxe concrète :

chaîne ⇒ ' *caractères-sauf-apostrophe-sauf-si-doublée* '

Les chaînes de caractères débutent par une apostrophe et s'achèvent de même. Les apostrophes internes doivent être redoublées pour être prises en compte à l'intérieur de la chaîne de caractères. Une chaîne peut s'étendre sur plusieurs lignes.

Exemples :

`'chaines'`   `'L''aile'`   `''`   `''''`

Les chaînes de caractères ne peuvent être modifiées ni en taille, ni en contenu. Une implantation peut limiter la taille des chaînes de caractères citées ou calculées.

3.4 Logiques

Les booléens Vrai et Faux sont utilisables comme valeurs des identificateurs réservés `vrai` et `faux`.

3.5 Procédures

Les procédures sont créées au moyen de l'instruction prédéfinie `debproc`. Lorsqu'invoquée, une procédure peut prendre des arguments et retourner des valeurs.

Exemples :

```
debproc fact;          // En supposant que ==, * et - soient définis correctement
  argument i*entier;
  si ( i == 1 );
    resproc 1;
  sinon;
    resproc ( i * (fact (i - 1)) );
  finsi;
finproc;
```

3.6 Boucles

Les boucles sont créées au moyen de l'instruction prédéfinie `repeter`. Une boucle peut être réitérée ou quittée, pendant sa durée de vie, grâce aux instructions prédéfinies `iterer` et `quitter`. Une boucle est associée à un indice de boucle qui compte le nombre de fois que la boucle a été itérée. Cet indice est régulièrement incrémenté en début de boucle et sa valeur initiale est zéro. Cet indice peut être obtenu par l'expression prédéfinie `indice`.

Exemples :

```
repeter fact;       // calcule n! dans la variable r
  si ( n < (indice fact) );
    quitter fact;
  sinon;
    r = r * (indice fact);
fin fact;
```

3.7 Types

À chacun des noms des types prédéfinis, apparaissant en table 1, est associée une valeur de type `type`. Cette valeur peut figurer dans l'instruction prédéfinie `argument` (cf. 5.5.1) pour spécifier le type de la valeur souhaitée.

Tous les types prédéfinis peuvent être obtenus comme valeur des identificateurs réservés de même nom. L'environnement initial peut apporter de nouveaux types prédéfinis.

3.8 Tables

Les tables sont des aggrégats extensibles de valeurs quelconques indexées par des valeurs quelconques. Les tables sont créées par l'expression prédéfinie `creer`. Les positions indexées des tables peuvent être référencées en lecture ou en écriture.

La comparaison des index s'effectue comme suit : la comparaison entre entiers, réels et booléens est naturelle. Deux chaînes de caractères sont égales si elles ont même taille et même contenu. Une procédure, une boucle, un type ou une table n'est égal qu'à lui-même. Une implantation peut limiter la taille des tables.

4 Programme

Un programme Gibiane est formé d'une suite d'instructions. Une instruction est une suite d'éléments syntaxiques terminée par un point-virgule. Ces éléments syntaxiques peuvent être des variables, des expressions, etc. À un niveau plus fin, ces éléments syntaxiques sont formés de lexèmes qui correspondent aux valeurs immédiates (nombres ou chaînes de caractères), aux identificateurs ou encore, suivant les contextes, à quelques signes de ponctuation qui sont les parenthèses, le point-virgule et le point d'exclamation.

Les lexèmes sont séparés, tant que de besoin, par des séparateurs qui sont des espaces, des tabulations, des retours en tête de ligne, des sauts à la ligne ou à la page, des commentaires et la fin de fichier. Les lexèmes s'étendent au maximum à droite, ou, en d'autres termes, lorsqu'un lexème a pour préfixe un autre lexème, ce premier est préféré.

Les commentaires débutent soit par une étoile en première colonne, soit par deux barres de fraction contiguës et, dans tous les cas, s'achèvent en fin de ligne.

Syntaxe concrète :

identificateur ⇒ *caractère-spécial*$^+$
 | *caractère-normal*$^+$ *caractère-suite**
caractère-normal ⇒ *lettre* | *chiffre* | `_` | `:`
caractère-suite ⇒ `'` | `+` | `-`
caractère-spécial ⇒ `<` | `>` | `=` | `*` | `/` | `|` | `%` | `&` | `^` | `~` | `+` | `-`

Un identificateur est une suite de caractères contigus qui ne peut contenir ni séparateur, ni signe de ponctuation. Un identificateur est soit formé d'une suite de caractères spéciaux (à l'exclusion de la suite formée du seul signe `=`), soit formé d'une suite de caractères tels que lettres, chiffres et quelques autres caractères mais ne pouvant débuter par une apostrophe.

Exemples :

`&&`	`->`	`x'`	`H2O`	`4u`	`2e0f`	`Cu--`	*sont légaux*
`*`	`/`	`+`	`<=>`	`**`	`0x34`	`Ag+'`	*sont légaux*
'x	*a.b*	*a+b*	=				*sont illégaux*

À l'exception des instructions prédéfinies qui s'évaluent de manière particulière, les instructions sont toujours évaluées en séquence. À l'exception des expressions prédéfinies qui s'évaluent de manière particulière, les expressions sont toujours évaluées de la gauche vers la droite.

4.1 Variables

Gibiane nomme les valeurs manipulées à l'aide de variables. Les variables sont représentées par des identificateurs. La casse des identificateurs n'est pas significative : `fOO` et `FoO` sont deux lexèmes représentant le même identificateur. Une implantation peut limiter la taille des identificateurs.

Toutes les variables peuvent a priori être la cible d'une affectation. Une variable peut être initialisée (c'est-à-dire avoir une valeur) ou être non initialisée. Une variable doit être initialisée afin d'être évaluée sans erreur.

Il existe deux sortes de variables en Gibiane : les variables globales, définies en dehors de toute procédure, et les variables locales qui n'ont d'existence que pendant l'invocation d'une procédure. Les variables mentionnées par une instruction `argument` sont locales tout comme le sont les variables subissant une affectation au sein d'une procédure. Les instructions prédéfinies, `argument`, `repeter` et `debproc`, sont les seules à affecter la (ou les) variable(s) sur lesquelles elles portent.

Les variables locales ont une portée dynamique, elles sont visibles de la procédure où elles sont créées ainsi que des procédures appelées par cette même procédure. Lorsqu'une procédure possèdant des variables locales est invoquée, ces variables locales sont créées et masquent les variables locales ou globales de même nom. Une variable locale a initialement pour valeur la valeur de la variable qu'elle masque mais elle reste non initialisée si la variable qu'elle masque est non initialisée. Une variable locale, sujette à l'instruction `argument`, peut devenir non initialisée si l'argument demandé manque.

Exemples :

```
pi = 3.14; // la variable pi est globale.
           // il n'y a pas de variable globale t.

debproc f;
  argument fun;
  t = pi;  // la variable locale t prend la valeur dynamique
           // de la variable locale pi
  pi = 1;  // la variable locale pi vaut maintenant 1
  resproc (fun t);
finproc;

debproc g;
  argument t*entier;
  resproc (t * pi);  // pi est une variable libre donc dynamique
finproc;

g 2;      // retourne 6.28
f g;      // retourne 3, la variable globale pi vaut toujours 3.14.
```

4.2 Identificateurs réservés

Certains identificateurs sont réservés : ils permettent de spécifier les expressions prédéfinies (répétition, définition de procédure, alternative, ...) ainsi que les constantes logiques Vrai et Faux. Les identificateurs réservés ne peuvent être employés comme noms de variable. Les identificateurs figurant en table 2 sont réservés. La casse des identificateurs réservés n'est pas significative. Ainsi `SI`, `si`, `Si` et `sI` désignent-ils tous l'alternative.

Les constantes logiques sont les valeurs des identificateurs réservés `vrai` et `faux` qui ne sont pas des variables et donc ne peuvent affectées.

si	sinon	finsi	vrai	faux
repeter	fin	quitter	iterer	indice
debproc	finproc	argument	resproc	existe
creer	evaluer	type		

Table 2: Identificateurs réservés de Gibiane (version française)

5 Instructions

Une instruction peut, en Gibiane, être une affectation, une commande ou une instruction prédéfinie. Les instructions se terminent cependant toutes par un point-virgule. La syntaxe des instructions repose sur la définition des expressions et des lieux affectables.

5.1 Lieu

Syntaxe concrète :

lieu ⇒ *lieu-simple*
  | *variable* `*` *expression-simple*
lieu-simple ⇒ *variable*
  | *expression-simple* `!` *expression-simple*

Les lieux affectables, ou lieux, décrivent les endroits où peuvent être stockées des valeurs. Un lieu affectable est soit une variable, soit une variable dont le type est restreint, soit une position indexée de table. Une variable dont le type est restreint est suivie d'une expression qui doit retourner un type. Les positions indexées de tables sont calculées au moyen de deux sous-expresssions — celle de gauche qui sera la première à être évaluée, doit conduire à une table — celle de droite, qui sera évaluée en second, doit retourner une unique valeur qui servira d'index dans la table précédemment obtenue. Tout index est possible dans toute table et constitue un lieu affectable.

5.2 Expression

Syntaxe concrète :

expression-simple ⇒ *entier* | *réel* | *chaîne*
  | `vrai` | `faux` | *lieu-simple*
  | `(` *expression* `)`
expression ⇒ *expression-simple*
  | *commande*
  | *affectation*
  | *expression-prédéfinie*

Les expressions peuvent être des valeurs immédiates (entiers, réels ou chaînes), les identificateurs réservés `vrai` ou `faux`, des lieux simples (des variables ou des positions indexées de tables), des commandes (c'est-à-dire des invocations de procédure), des affectations ou, enfin, des expressions prédéfinies ou parenthésées.

Lorsqu'évaluée, une expression retourne une séquence de valeurs éventuellement vide. La plupart des expressions retournent une unique valeur, c'est le cas des valeurs immédiates et des identificateurs réservés. L'évaluation d'un lieu doit mener à un endroit où peut être stockée une valeur.

Une commande est une suite d'expressions qui, après évaluation, conduira, le plus souvent, à l'invocation d'une procédure sur des arguments. Une commande peut retourner un nombre quelconque de valeurs. Une affectation est également une expression, elle retourne les valeurs affectées. Les expressions parenthésées permettent de lever certaines ambiguïtés et retournent ce que retourne l'expression entre parenthèses. Quelques expressions prédéfinies sont disponibles comme `existe` qui seront décrites plus loin.

5.2.1 Affectation

Syntaxe concrète :

affectation ⇒ *lieu*$^+$ `=` *expression-simple*$^+$

Une affectation permet de calculer puis de ranger des valeurs dans des variables ou dans des positions indexées de tables.

L'évaluation d'une affectation s'effectue de la gauche vers la droite comme suit. Les lieux sont tout d'abord évalués de la gauche vers la droite. Les expressions sont alors évaluées séquentiellement de la gauche vers la droite ; les valeurs qu'elles retournent sont conservées dans leur ordre de production. L'affectation proprement dite est alors réalisée. Le lieu obtenu le plus à gauche est examiné. Si ce lieu est une variable dont le type n'est pas restreint ou une position dans une table, alors la première des valeurs est stockée dans ce lieu. Si ce lieu est une variable dont le type est restreint alors il doit exister une valeur compatible qui y sera stockée. C'est la plus à gauche des valeurs compatibles qui sera élue. Lorsque le lieu est affecté, la valeur stockée est retirée des valeurs à affecter et le processus recommence avec le lieu suivant. Il doit y avoir autant de valeurs à affecter que de lieux.

Une affectation, utilisée comme expression, retourne comme valeurs les valeurs qui étaient à stocker dans l'ordre où elles furent stockées.

Exemples :

```
i j = 2 3;                                    i vaut 2 et j vaut 3
            tab est supposée être une table
i j tab!3 = j i 4;          i vaut 3, j vaut 2, tab contient 4 à l'index 3
i i = 2 3;                                           i vaut toujours 3
i tab!(i = 4) = 3 i;     i vaut toujours 3 et tab contient 4 en position 4
            la procédure - supposée exister
tab!j = tab;                         tab se contient elle-même à l'index 2
i*entier tab!j = tab i;                               aucun changement
i tab!j!i = tab!i (i - 2);             i vaut 4, tab contient 1 à l'index 3
tab!'i' = tab!(i - 1);   tab contient 1 à l'index i qui est une chaîne de caractères
j k = (i*entier j = 'ab' 2);                             i et j valent 2
z = 2 3;    // Erreur: trop de valeurs
z1 z2 = 1;  // Erreur: pas assez de valeurs
```

5.2.2 Commande

Syntaxe concrète :

commande $\Rightarrow$ *expression-simple*$^+$

Une commande correspond à l'invocation d'une procédure.

L'évaluation d'une commande s'effectue ainsi : les expressions sont évaluées séquentiellement de la gauche vers la droite. Les valeurs retournées par ces expressions sont alors prêtes à servir d'arguments. Ces valeurs sont alors scrutées de la gauche vers la droite pour trouver une valeur de type procédure. Si une telle valeur n'est point trouvée, les valeurs obtenues forment le résultat final de la commande.

Si une procédure est trouvée, elle est alors invoquée et prendra comme arguments les valeurs restantes sans changement d'ordre. Les valeurs retournées par cette invocation ainsi que les arguments non consommés seront considérés comme formant, dans cet ordre, une nouvelle commande à évaluer. La commande s'achève lorsqu'il ne reste plus de procédure parmi les arguments. Il se peut, par contre, qu'il reste des arguments non consommés qui deviendront des résultats supplémentaires.

Exemples :

```
             // en supposant que + et * existent et soient binaires
1 + 2 * 3;   // retourne (1+2)*3
```

```
1 2 3 + *;   // retourne aussi (1+2)*3
1 + 2 3;     // retourne 3 et 3

debproc enumerer;
  argument i*entier;
  if ( i > 0 );
    resproc (enumerer (i - 1)) i;
  finsi;
finproc;

a b c = 3 enumerer;      // affecte les trois valeurs 1, 2 et 3.

debproc redire;
  argument i*entier truc;
  repeter B i;
    resproc truc;
  fin B;
finproc;

debproc fact;
  argument n*entier;
  resproc (enumerer n) (redire * (n-1));
finproc;
```

5.2.3 Expression prédéfinie

Plusieurs expressions prédéfinies sont disponibles.

5.2.4 Test d'existence de position indexée dans une table

Syntaxe concrète :

expression-prédéfinie ⇒ `existe` *expression-simple expression-simple*

L'expression prédéfinie `existe` permet de tester si une table possède une position indexée. Elle retourne une unique valeur de type logique. L'expression prédéfinie `existe` possède un autre sens détaillé en section 5.2.5.

L'expression prédéfinie `existe` est formée de deux sous-expressions qui sont évaluées séquentiellement de la gauche vers la droite. L'expression de gauche doit retourner un unique résultat de type table. L'expression de droite doit retourner une unique valeur de type quelconque qui servira d'index. L'expression prédéfinie `existe` répond Vrai s'il existe dans la table une valeur ainsi indexée et répond Faux si ce n'est pas le cas.

Exemples :

```
t = creer table;
t!'un' = 1;
t!1 = ( existe t 'un' ); // stocke Vrai
```

5.2.5 Test d'initialisation de variable

Syntaxe concrète :

expression-prédéfinie ⇒ `existe` *variable*

L'expression prédéfinie `existe` retourne Vrai si la variable est initialisée et Faux autrement.

Si la variable est globale c'est que la variable n'a jamais été la cible d'une affectation.

Si la variable est locale et qu'elle masque une variable de même nom (locale ou globale) alors elle a pour valeur, à l'entrée de la procédure, la valeur qu'avait la variable masquée de même nom. Si la variable est locale et qu'elle ne masque aucune variable de même nom alors elle est non initialisée. Après l'instruction prédéfinie `argument`, une variable locale peut devenir non initialisée si elle requérait la présence d'un argument absent.

Exemples :

```
x = 10;

debproc somme;
  repeter B;
    argument i/entier;
    si ( existe i );
      x = x + i;
    sinon;
      resproc x;
      quitter B;
    finsi;
  fin B;
finproc;

somme 1 2 'trois' 4;  // retourne 17 et 'trois', x vaut toujours 10
```

5.2.6 Indice de boucle

Syntaxe concrète :

expression-prédéfinie ⇒ `indice` *expression-simple*

Il est possible de connaître le nombre de fois qu'une boucle a été itérée. Initialement cet indice est nul ; il est incrémenté à chaque début de boucle. La valeur de l'indice d'une boucle inactive est la valeur qu'il avait lorsque la boucle fut quittée.

L'expression située en argument de `indice` doit retourner une unique valeur de type boucle. L'expression entière produit un unique résultat de type entier.

Exemples :

```
debproc jusqua;
  argument n*entier;
  argument b;
  resproc b;
  si ( n < (indice b) );
```

```
    quitter b;
  finsi;
finproc;

debproc enumerer;
  argument n*entier;
  repeter Attendre;
    resproc (indice (jusqua n Attendre));
  fin Attendre;
finproc;

enumerer 5;   // retourne les entiers 1, 2, 3, 4 et 5. Le dernier
              // resproc en cours est annulé par quitter.
```

5.2.7 Création de table

Syntaxe concrète :

expression-prédéfinie ⇒ `creer` *expression-simple*

Cette expression prédéfinie retourne une table du type indiqué. Son argument doit retourner un unique type, sous-type de `table`.

Exemples :

```
debproc identite;    // La fonction identité
  argument a;
  resproc a;
finproc;

t = creer table;
t!'un' = 1;
t!(identite t!'un') = 2;
```

5.2.8 Test de type

Syntaxe concrète :

expression-prédéfinie ⇒ `type` *expression-simple*

Cette expression prédéfinie retourne le type de son argument. L'expression située en argument de `type` doit retourner une unique valeur de type quelconque. L'expression tout entière retourne un unique type.

Exemples :

```
debproc est_du_meme_type;
  argument x;
  argument y/(type x);
```

```
    resproc (existe y);
  finproc;
```

5.2.9 Évaluation dynamique

> **Syntaxe concrète :**
>
> *expression-prédéfinie* ⇒ `evaluer` *expression-simple*$^+$

Cette expression prédéfinie agit comme une procédure : ses arguments sont évalués et doivent produire des chaînes de caractères ou des nombres. Les nombres sont implicitement convertis en des chaînes. Toutes les chaînes obtenues sont alors concaténées, dans l'ordre et la chaîne résultante est évaluée, comme une expression, dans le contexte courant et retourne ce que retourne cette expression.

Exemples :

```
a = '+ 3'; evaluer 'a =' 1 ' ' a;       // cad, a =1 + 3

debproc f;
  argument txta*chaine txtb*chaine;
  resproc (evaluer 'a ' txta '2 ''' textb);
finproc;

f 'txta = 1' '''';  // retourne 12 et la chaine vide.
                    // la variable globale a n'est pas modifiée
```

5.2.10 Expression parenthésée

Les parenthèses permettent de forcer une interprétation syntaxique particulière. Elles sont surtout intéressantes dans le cas d'affectation ou de commande utilisée comme sous-expression.

Une expression parenthésée s'évalue comme l'expression qu'elle contient et retourne les mêmes résultats qu'elle.

Exemples :

`i` *vaut 3*

`i tab!(i = 1) = i ((i = i + 1));` — `i` *vaut 1 et* `tab` *contient 4 à l'index 1*

`i = (1);` — *équivalent à* `i = 1;`

5.3 Expression comme instruction

> **Syntaxe concrète :**
>
> *instruction* ⇒ *expression* `;`

Toute expression peut-être considérée comme une instruction si suffixée par un point-virgule. Les valeurs produites par cette expression sont alors ignorées.

5.4 Instruction prédéfinie

Syntaxe concrète :

instruction ⇒ *alternative*
  | *répétition*
  | *définition-procédure*

Gibiane procure un certain nombre d'instructions prédéfinies. Ce sont l'alternative (`si`), la répétition (`repeter`) et la définition de procédure (`debproc`). Au sein de ces instructions sont, parfois, permises d'autres instructions prédéfinies spécialisées comme, par exemple, la reprise ou l'abandon d'une répétition, la récupération d'arguments supplémentaires ou le retour additionnel de résultats dans une définition de procédure.

Les instructions prédéfinies ont une syntaxe équilibrée, elles se terminent toutes par une instruction délimitant leur fin. Les instructions prédéfinies doivent être convenablement imbriquées.

5.4.1 Alternative

Syntaxe concrète :

alternative ⇒ `si` *expression-simple* ;
  *instruction**
  `sinon;`
  *instruction**
  `finsi;`
  | `si` *expression-simple* ;
  *instruction**
  `finsi;`

L'alternative est syntaxiquement composée de plusieurs instructions. L'alternant peut être omis.

En premier lieu, la condition (c'est-à-dire l'expression qui suit l'identificateur réservé `si`) est évaluée et doit conduire à une unique valeur de type logique. Si cette valeur est vraie alors la conséquence (c'est-à-dire les instructions qui suivent jusqu'au `sinon` ou `finsi` approprié) est évaluée et l'alternant ignoré. Si cette valeur est fausse alors la conséquence est ignorée et l'alternant (c'est-à-dire les instructions comprises entre `sinon` et `finsi`), si présent, est évalué. Le calcul continue alors en séquence.

5.4.2 Répétition

Syntaxe concrète :

répétition ⇒ `repeter` *variable* *expression-simple* ;
  *instruction**
  `fin` *variable* ;
  | `repeter` *variable* ;
  *instruction**
  `fin` *variable* ;

La répétition permet d'évaluer un certain nombre de fois une même séquence d'instructions. Deux sortes de répétitions existent : la répétition bornée et la répétition illimitée. La répétition bornée mentionne le nombre de répétitions à opérer. Le corps d'une répétition s'achève avec l'instruction `fin` suivie du même nom de variable que l'instruction `repeter` initiale. On distingue une répétition bornée d'une répétition illimitée par la présence d'une expression après la variable suivant l'identificateur réservé `repeter`.

En premier, est évaluée, si présente, l'expression bornant la répétition puis un objet de type boucle est créé et affecté à la variable mentionnée après l'identificateur réservé `repeter`. La valeur précédente de cette variable est alors perdue. Si une répétition apparaît dans une définition de procédure alors la variable est considérée comme affectée et, par conséquent, est locale à cette procédure.

Si le nombre de répétitions restant à effectuer est strictement plus grand que zéro, alors la boucle est active et le corps de la boucle est évalué et le nombre de répétitions restant à effectuer est décrémenté. Si le nombre de répétitions restant à effectuer est nul alors la boucle est achevée et l'évaluation reprend avec la première instruction suivant l'instruction `fin` correspondante. Lorsqu'une boucle est achevée, elle ne peut être reprise : la valeur de type boucle a alors achevé sa durée de vie utile, la boucle est dite alors inactive.

Il existe deux instructions prédéfinies opérant sur des objets de type boucle : `iterer` et `quitter`. Il existe également une expression prédéfinie opérant sur un objet de type boucle : `indice`.

5.4.3 Abandon d'itération en cours

Syntaxe concrète :

instruction ⇒ `iterer` *expression-simple* ;

L'instruction prédéfinie `iterer` permet d'abandonner l'itération en cours pour reprendre l'itération suivante ou, si la boucle est épuisée, de l'achever.

L'expression est évaluée et doit conduire à une unique valeur de type boucle. Cette boucle doit de plus être active. Si le nombre de répétitions restant à effectuer est strictement plus grand que zéro, alors la boucle reste active, l'exécution reprend au début du corps de la boucle, le nombre de répétitions restant à effectuer est décrémenté et l'indice de boucle associé est incrémenté. Si le nombre de répétitions restant à effectuer est nul alors la boucle s'achève et l'exécution reprend avec la première instruction qui suit l'instruction `fin` correspondante.

Exemples :

```
debproc sortir;
  argument t*table limite*entier;
  boucle*boucle = t!'retour';
  si ( ( indice boucle ) > limite );
    quitter boucle;
  finsi;
finproc;

tab = creer table;
repeter B;
  tab!'retour' = B;
  sortir 33 tab;
fin B;
```

5.4.4 Abandon total de boucle

Syntaxe concrète :

instruction ⇒ `quitter` *expression-simple* ;

L'instruction prédéfinie `quitter` permet d'abandonner totalement une répétition en cours.

L'expression est évaluée et doit conduire à une unique valeur de type boucle. Cette boucle doit de plus être active. Le nombre de répétitions restant à effectuer est mis à zéro, la boucle devient inactive et l'exécution reprend avec la première instruction qui suit l'instruction `fin` correspondante.

5.5 Définition de procédure

Syntaxe concrète :

définition-procédure ⇒ `debproc` *variable* ;
instruction*
`finproc` ;

Les procédures sont définies grâce à l'instruction `debproc`. Une procédure est une succession d'instructions paramétrées par des arguments qui peuvent être en nombre quelconque. Le corps de la procédure débute après l'instruction `debproc` et s'achève avec l'instruction `finproc`. Les définitions de procédures ne peuvent être imbriquées.

Une définition de procédure conduit à la création d'une valeur de type procédure qui sera affectée à la variable mentionnée après l'identificateur réservé `debproc`. La valeur précédente de cette variable, s'il y en avait une, est alors perdue. La procédure une fois définie, c'est-à-dire la variable une fois affectée, l'évaluation continue avec l'instruction qui suit l'instruction `finproc`.

Lorsqu'une procédure est invoquée au sein d'une commande, elle récupère ses arguments au moyen de l'instruction prédéfinie `argument` et spécifie ses résultats au moyen de l'instruction prédéfinie `resproc`. Une procédure retourne l'ensemble ordonné de ses résultats à son appelant lorsque l'évaluation de son corps s'achève. La procédure peut s'achever sans avoir consommé tous ses arguments.

Exemples :

```
debproc mange_entiers;
  repeter B;
    argument i/entier;
    si (existe i);
      iterer B;
    sinon;
      quitter B;
    finsi;
  fin B;
finproc;
```

5.5.1 Récupération d'argument

Syntaxe concrète :

instruction ⇒ `argument` *argument*$^+$;

argument ⇒ *variable*
  | *variable* / *expression-simple*
  | *variable* * *expression-simple*

L'instruction prédéfinie `argument` permet à une procédure de récupérer dynamiquement des arguments s'ils sont disponibles. Ces valeurs sont obtenues par affectation sur des variables locales.

L'évaluation d'une instruction prédéfinie `argument` procède comme suit. Tout d'abord, les expressions sont évaluées de la gauche vers la droite. Chacune de ces expressions doit produire un unique type (du type `type`).

Lorsqu'une procédure est invoquée, une suite de valeurs (ses arguments) est associée à cette invocation. La procédure peut récupérer ces valeurs afin d'adapter son comportement. Ces valeurs sont choisies parmi les arguments de la gauche vers la droite en fonction de leur type (à titre exceptionnel, lorsqu'un réel est demandé, la valeur choisie est l'entier ou le réel qui est le plus à gauche parmi les arguments). Lorsqu'un argument est récupéré, il est retiré de la séquence des arguments en attente.

Lorsqu'une telle valeur est trouvée, elle est affectée à la variable qui la requérait. Si aucune valeur n'est du bon type et qu'une étoile sépare le nom de la variable de son type, alors une erreur est signalée. Si aucune valeur n'est du bon type et qu'une barre de fraction sépare le nom de la variable de son type, alors la variable est considérée comme non initialisée et ne peut être lue sans qu'une erreur soit signalée. L'expression prédéfinie `existe` permet de tester si une variable est initialisée ou pas. Si le nom de la variable n'est pas suivi d'un type alors le premier des arguments est affecté à la variable.

Exemples :

```
debproc filtrer_reels;
  repeter Extraire_reels;
    argument i/reel;
    si ( existe i );
      resproc i;
      iterer Extraire_reels;
    sinon;
      repeter Consommer_autres_arguments;
        argument chose;
        si ( existe chose );
          iterer Consommer_autres_arguments;
        sinon;
          quitter Extraire_reels;
        finsi;
      fin Consommer_autres_arguments;
    finsi;
  fin Extraire_reels;
finproc;

1 'deux' 3.4 filtrer_reels 5 'six' filtrer_reels; // retourne 1.0, 3.4 et 5.0

debproc instance_de;
  argument t*type;    // Ces deux lignes ne peuvent être
```

```
  argument x/t;       // fusionnées en une unique instruction.
  resproc (existe x);
finproc;

debproc somme_entiers;
  argument r*reel i*entier;
finproc;

somme_entiers 1 2.3;  // erreur: r sera lié à 1.0, il n'y a pas d'entier pour i
```

5.5.2 Fourniture de résultat

Syntaxe concrète :

instruction ⇒ `resproc` *expression-simple*$^+$;

L'instruction prédéfinie `resproc` permet d'ajouter des valeurs à la séquence de valeurs qui sera retournée par la procédure.

Les expressions qui suivent l'instruction `resproc` sont évaluées séquentiellement de la gauche vers la droite. Sitôt qu'une expression est évaluée, ses valeurs sont ajoutées, dans l'ordre, à la séquence des valeurs qui sera produite par la procédure.

Exemples :

```
debproc enum;
  argument i*entier;
  resproc i;
  if ( i > 0);
    resproc (enum (i - 1)) i;
  sinon;
    resproc;                  // aucun résultat
  finsi;
finproc;

a = (0 enum);    // a vaut 0
3 enum;          // retourne les valeurs 3, 2, 1, 0, 1, 2 et 3
```

L'instruction `finproc` force la procédure à s'achever et à retourner l'ensemble des valeurs accumulées à son appelant. Si aucune instruction `resproc` n'a eu lieu, la procédure ne retourne aucune valeur.

6 Environnement initial

L'environnement initial, c'est-à-dire l'ensemble des variables prédéfinies ainsi que leur valeur associée, n'est pas spécifié. En particulier, Gibiane ne procure aucune procédure prédéfinie sur les valeurs logiques, numériques ou de type chaîne de caractères. L'environnement initial peut également définir de nouveaux types de données.

7 Conclusions

Le langage, dont la description s'achève ici, est proche mais différent des langages qui portent actuellement le nom de GIBIANE. Les distinctions sont avant tout d'ordre syntaxique : définition des identificateurs, changement de nom de primitives, sélection dans les tables, etc. En revanche, d'autres distinctions sont d'ordre sémantique : expressions évaluables spécifiant des types, indice de boucles, règles de portée, etc.

Ce document définit GIBIANE, de façon plus précise que naguère, mais n'est toutefois pas une description formelle.

Références

[Chaa] Charras (T). – Développer dans castem2000: Esope. – sans numéro.

[Chab] Charras (T). – Développer dans castem2000: les objets. – sans numéro.

[Chac] Charras (T). – Développer dans castem2000: les opérateurs. – sans numéro.

[Cha94a] Charras (T). – *Castem2000 Gibiane.* – Rapport technique n° DMT/94-154, CEA, 1994.

[Cha94b] Charras (Th). – *CASTEM2000 GIBIANE.* – Rapport technique n° DMT/94-154, CEA, 1994.

[CLRV89] Charras (T), Lautard (JJ), Robeau (MF) et Verpeaux (P). – Langage gibiane: définition. – 1989. sans numéro.

[Gro95] Groupe de travail. – *Rapport de synthèse du groupe architecture logicielle Elan, cahier des charges du langage de commande et du superviseur.* – Rapport technique n° DRN/CIS 95-020, CEA, 1995.

[GT] GT Logiciel jeunes spécialistes. – Rapport commun. – 07/02//1995.

[Lou94a] Loubiere (S). – *APROC Bibliothèque de procédures pour Apollo2.* – Rapport technique n° DMT/94-002, SERMA/LENR-1582, CEA, 1994.

[Lou94b] Loubiere (S). – *APROC Listing des procédures.* – Rapport technique n° DMT/94-024, SERMA/LNR-1583, CEA, 1994.

[RM94] Robeau (MF) et Moreau (F). – *Acropole, Guide utilisateur et manuel de référence.* – Rapport technique n° DMT/94/027, SERMA/LNR/1585, CEA, 1994.

[RV86] Robeau (MF) et Verpeaux (P). – *Langage de données Gibiane.* – Rapport technique n° DEMT 86/080, SERMA/LENR/86/768, CEA, 1986.

[Tou94] Toumi (I). – *Grands logiciels scientifiques, propositions pour une base de modélisation scientifique.* – Rapport technique n° DMT/94/638, SERMA/94/1695, CEA, 1994.

[Ver93] Verpeaux (Pierre). – *Esope (version 10.0), Gemat (version 10.0) Manuel d'utilisation.* – CEA, 1993.

Index