Documentation

Cette documentation est à l'usage des utilisateurs susceptibles de réaliser un exercice.

Généralités

3 zones de codes

- **options :** on y indique des variables définissant le mode d'exécution de l'exercice.
- **initialisation :** on y initisalise les variables aléatoires qui rendent l'exercice unique.
- **code :** on y place le code servant à l'exécution de l'exercice.

Commentaires

Une ligne vide est ignorée. Tout ce qui suit `#` est considéré comme commentaire

Code à balises

On utilise une langage fait de balises, comme en html.

- Certains balises fonctionnent par paire : `<bloc>...</bloc>`
- D'autres balises sont auto-fermantes : `<bloc/>`
- En général, il est possible d'indiquer une option avec l'ouverture du bloc : `<bloc:info/>` ou `<bloc:info>...</bloc/>`

Les options

Dans le cadre des options, on ne déclarera que des blocs `<option>`

```
<option:nom>
key1 => label1
key2 => label2
</option>
```

- `nom` est le nom de l'option et sera accessible ensuite avec `@nom`
- `key1` est une clé et doit être numérique. C'est elle qui sera stockée en BDD quand l'utilisateur voudra indiquer l'option choisie.
- `label1` est l'étiquette correspondante. Elle sera affichée dans le menu de chois des options.

Remarque : On refuse les noms d'options commençant par `_`.

Important : La première valeur de l'option est toujours la valeur par défaut.

Initialisation et code

Initialisation

L'exécution d'un exercice peut se faire dans deux contextes :

- il s'agit d'une nouvelle exécution, il faut donc initialiser les paramètres utiles.
- Il s'agit de l'exécution d'un exercice préalablement initialisé et dont les paramètres ont été stocké en BDD.

L'initialisation ne s'exécute que dans le premier cas. Ainsi, réexécuter un exercice précédemment sauvegardé en BDD devrait toujours donner le même résultat.

Affectation

C'est l'essentiel du travail d'initialisation : calculer une valeur pour les paramètres constitutifs de l'exercice. Pour l'initialisation, ces calculs utilisent souvent de l'aléatoire.

Pendant l'exécution de l'exercice, on aura également besoin de faire des calculs par exemple pour calculer les réponses attendues.

Nommage des variables

- On annonce les variables en les faisant précéder par `@`.
- L'affectation d'une variable se fait avec un signe `=`
- Les variables dont le nom commence par `_`, par exemple `@_x`, définies dans le bloc d'initialisation ne seront pas sauvegardées et seront donc perdues au moment de l'exécution de l'exercice. Elles peuvent servir de variable intermédiaire pour faciliter la rédaction de l'init.

Important : *On interdit les noms de variables commençant par* `__`.

Formule ordinaire

On peut affecter une variable avec une formule de calcul ordinaire qu'il suffit d'écrire.

Exemple :

```
@_x = 3
@y = 3+2*@_x^2
```

Évaluation d'une pile

Si l'expression à évaluer est entourée de `<P: >`, alors elle est comprise comme une pile. L'expression est alors découpées selon les espaces.

Les éléments de la pile subissent un prétraitement :

- Si un bloc est reconnu de la forme `@name` ou `@name.sub` ou `@name[index]` alors il est remplacé par la valeur correspondante. Cela pourrait donc être un objet.
- Si une chaîne contient des éléments de la forme `@name` ou `@name.sub` ou `@name[index]` alors ils sont substitués mais alors sous forme textuelle et avec des parenthèses !

Par exemple si on a `3*@b+2` et que `@b` contient `x+4` alors on obtiendra `3*(x+4)`

On suite on passe à l'exécution de la pile :

- Tout bloc non chaîne de caractère est considéré comme opérande est placé dans la pile des opérandes.
- Tout bloc de la forme `"module.fonction"` est considéré comme une fonction.
    - Cette fonction sera cherchée parmi les modules disponibles. Il faudra donc qu'elle existe bien.
    - Cette fonction a une certaine **arité**. Elle prendra donc les opérandes dans la pile des opérandes. Il faut qu'il y en ait assez.
    - Attention à l'ordre : l'opérande le plus haut sur la pile sera celui donné en dernier à la fonction.
    - Le résultat de l'exécution est placé sur la pile des opérandes.

Affectation d'un tableau

On peut affecter un tableau simple (pas un tableau de tableau) avec la syntaxe habituelle `[...]`.

```
@t = [1, 2, 3]
@v = [@a, @c-2, 17]
```

Commandes plus avancées pour des tableaux

Affectation `<:n>`

- `@a <:3>= 5` va ainsi créer un tableau `[5, 5, 5]`.
- On dispose de la variable `@__i` qui représente l'index. Ainsi :

  `@a <:3>= 5 + @__i` va affecter `[5, 6, 7]`
- On peut également faire des calculs sur un tableau :

  `@b <:3>= @a[]^2` va affecter `[25, 36, 49]`.

Attention, l'utilisation dans l'expression de `@a[]` lèvera une erreur...

- si on a pas précisé `<:3>` ou une autre taille avant le `=` pour indiquer que l'on fait un calcul sur tableau,
- si la taille indiquée excède la taille de `@a`.

Avec une pile

On peut aussi travailler avec une taille non connue à l'exécution. Ceci par exemple est autorisé :

```
@n = <P:3 10 Alea.entier>
@a <:@n>= @__i^2
```

Ce qui créera une tableau de taille aléatoire.

On peut naturellement utiliser ces affectations dans le cas d'une pile.

```
@n = <P:3 10 Alea.entier>
@a <:@n>= <P:0 100 Alea.entier>
```

Le répéteur est un tableau `<:@a>`

Ce qui crée un tableau d'entiers aléatoires de taille aléatoire.

Le répéteur peut être un tableau. Par exemple :

```
@a <:5>= @__i + 10
@b <:@a>= @__v^2 + @__i + 1
```

Ainsi le tableau `@b` est obtenu en parcourant les items de `@a`.

On dispose du raccourci `@__v` désignant l'item de `@a` en cours de lecture et `@__i` désignant l'indice.

Ici, `@a = [10, 11, 12, 13, 14]` et donc on prend chaque item, on l'élève au carré, on lui ajoute son indice et on ajoute `1` ce qui donne `@b = [101, 123, 147, 173, 201]`.

Affectation push

On peut également définir un tableau par ajouts successifs :

```
@a[] = 3
@a[] = 12
```

Crée un tableau contenant les valeurs `3` et `12`.

Alea entier

On dispose de la fonction `Alea.entier` mais cela nécessite de passer par la pile. Ainsi, quand on veut créer un nombre entier aléatoire entre `0` et `5` compris par exemple, il faut écrire :

```
@x = <P: 0 5 Alea.entier >
```

Cela peut sembler un peu lourd. Alors on a la possibilité d'utiliser une variable réservée : `@__a._6` qui crée un nombre entier aléatoire de `0` à `6` **exclus**.

```
@x = @__a._6
```

Naturellement on peut choisir n'importe quel nombre entier strictement positif.

Attention : si un alea est plusieurs fois présent dans le même calcul, ce ne sera pas le même !

```
@x = @__a._6 + @__a._10 + @__a._6
```

Le premier et le dernier nombre aléatoire, choisi dans $[0;6[$, ne sont pas les mêmes !

Comme les mêmes besoins reviennent constamment, il existe divers formats de nombres aléatoires :

- `@__a._10` ou `@__a.i10`, alea entier de 0 à 10 exclu,
- `@__a.I10`, alea entier de 1 à 10 inclu,
- `@__a.f10`, alea flottant de 0 à 10 exclu
- `@__a.s10`, alea entier de -10 à 10 exclus (de -9 à 9)

- `@__a.S10`, alea entier de -10 à 10, inclus et sans le 0.
- `@__a.vxyz`, le `v` signale qu'une lettre sera choisi aléatoirement parmi `xyz`. Liste des variable au choix, par ex `@__a.vtAMnK`... Permet de varier les énoncés et de ne pas toujours utiliser `x` comme variable.

Infini

On peut gérer une quantité infini avec `infinity`.

Contrôle de flux

Dans le bloc d'**initialisation** ou le bloc de **code** de l'exercice, le concepteur a le droit à quelques structures de contrôle de flux, c'est à dire des `if`, `else`... Ces structures supposent la possibilité d'énoncer des **conditions logiques**.

Conditions

Comparaison

Il est possible de comparer des quantités en utilisant `==`, `!=`, `<`, `<=`, `>`, `>=`.

Par exemple : `@a <= 3`

Opérateurs logiques

On peut structurer la condition logique avec les opérateurs `and` et `or`.

Le `and` est prioritaire.

En cas d'opération complexe, il conviendra de faire usage de `{...}` en guise de parenthèses.

Par exemple, cette condition est autorisée :

```
{@a == 0 or @a == 2} and @b = 10
```

some et all

Dans le cas d'un tableau, il est possible de chercher si la totalité des éléments vérifient une condition ou si certains la vérifient. On utilisera alors les opérateurs `all` et `some`.

Par exemple, si `@a = [1, 4, 5, 10]` on pourra faire une comparaison pour savoir si un des éléments est égal à 5 ou encore si tous les éléments sont égaux à 5.

- Au moins un élément égal à 5

  ```
  some {@a==5}
  ```

- Tous les éléments égaux à 5

  ```
  all {@a==5}
  ```

- Au moins un élément égal à 5 et tous inférieurs à 10

  ```
  some {@a == 5} and all {@a < 10}
  ```

- Au moins un entre 5 et 10

```
some {@a > 5 and @a < 10}
```

if

structure de `if` classique.

```
<if @a == @b>
...
<elif @a != 12>
...
<else>
...
<endif>
```

Les `if` peuvent être imbriqués.

On peut utiliser `</if>` pour fermer le bloc.

needed

Indique une contrainte.

Si elle n'est pas respectée pendant l'initialisation, celle-ci redémarre automatiquement pour un nouvel essai.

L'idée est d'initialiser aléatoirement et de « relancer les dés » jusqu'à obtenir des valeurs convenables.

```
<needed @xA != @xB or @yA != @yB >
```

Donc, un échec de `needed` interrompt la tentative d'initialisation et en relance une autre à partir de zéro. Pour éviter une situation bloquante, au bout d'une centaine d'essai, la tentative d'initialisation est abandonnée et un message d'erreur est envoyé.

Noter que, bien que `needed` soit un bloc singleton, il ne faut pas le fermer par `/>`.

until

Ne peut être placé que dans le bloc init et ne peut contenir que des affectations.

```
<until @a > 0>
@a = @__a._10 - 3
</until>
```

Répète les affectations **jusqu'à** ce que la condition soit vraie. La condition n'est évaluée qu'à la fin. Pertinent quand on veut intialiser une valeur aléatoire en respectant une condition sans pour autant tout redémarrer comme avec needed.

Pour éviter une situation bloquante, `until` n'est répété au plus que 100x.

Ainsi, si le professeur indique une condition trop contraignante ou impossible, la tentative d'exécution échoue et un message est affiché.

Les blocs constituant un exercice

Ces blocs ne seront présents que dans la zone de code, pas dans l'initialisation.

Forme générale

Déclaration

Un bloc est déclaré de la façon suivante :

```
<nombloc:titre>
...
</nombloc>
```

Certains type de blocs auront un comportement particulier comme les blocs `form`, `texte`, `input`, `graph`, ...

Un bloc peut contenir des sous blocs. Par exemple un bloc `form` peut contenir des blocs `input` ou `text`. Un bloc `graph` peut contenir des blocs `point` ou `function`...

Paramètres

Un bloc peut également recevoir des paramètres.

```
<nomparam:value/>
```

Bien noter le `/` final qui indique un bloc autofermant. Ce format de balise indique que l'on ajoute un paramètre au bloc.

Certains paramètres sont essentiels au fonctionnement. Par exemple :

```
<input:x>
<tag:$x$/>
<solution:3>
</input>
```

On renseigne ici que le bloc `<input>` aura un paramètre `solution` égal à 3 et une étiquette `$x$` qui sert à l'affichage.

Paramètres comme tableaux

Il est possible de traiter ces paramètres en tableaux :

```
<param[]:1/>
<param[]:7/>
```

Ainsi on a un paramètre de nom `param` qui est le tableau `[1, 7]`.

Toutefois, si on ajoute plusieurs valeurs dans un même param, celui-ci sera toujours compris comme un tableau.

```
<param:1/>
<param:7/>
```

Dans ce cas, le `7` n'écrase pas le `1`. La première ligne produit `param = 1` et la seconde ligne met à jour `param = [1, 7]`.

Un paramètre peut aussi être un texte avec formatage. Par exemple `<param:image de {@x:}/>`

Formatage

Dans une zone de texte, on est susceptible d'ajouter une formule. On peut alors écrire un bloc de formatage `{expr:}`. Ce bloc indique que l'on veut formater expression selon un certain formatage. Bien sûr on peut vouloir afficher le contenu d'une variable : `{@x:}`. Il existe divers formages.

- `{@x:}` pas de formatage particulier
- `{@x:$}` formaté en latex. Attention, ce formatage se contente de produire le code latex.

  Si on souhaite en plus que ce code soit identifié comme du latex et soit rendu en tant que tel, il faudra ajouter les `$`, donc écrire `${@x:$}`
- `{@x:f}` ou `{@x:3f}` pour un affichage approximé. Si on ne précise pas de nombre de chiffres après la virgules, alors le nombre est écrit en entier.

> Attention : nerdamer qui est utilisé pour les calculs remplace automatiquement les nombres décimaux en fractions. Ainsi, si on écrit `2.1`, on aura `21/10`. Cela peut être gênant dans une expression. Le format `float` permet alors de convertir les nombres fractionnaires en approximations à virgule.

- `{@x:f$}` pour float et latex. En effet, comme précisé dans la note ci-dessus, nerdamer convertit les nombres décimaux en fractions. Alors on est ennuyé si on veut un affichage avec nombres décimaux et en rendu latex. Ce format est là pour cela. On peut préciser un niveau d'approx, par exemple `{@x:3f$}`.

Supposons que l'on ait produit un trinome de forme $a \cdot x^2 + b \cdot x + c$ et que l'on ait stocké dans des variables `@a`, `@b`, `@c` les valeurs des coefficients, générés aléatoirement pour les besoins de l'exercice.

On pourrait croire qu'il suffit d'écrire `$@a x^2 + @b x + @c$` pour obtenir une représentation correcte du trinome.

Mais cette méthode est mauvaise car `@a` pourrait être égal à `1` et `@b` pourrait être nul ou négatif... C'est pour cette raison qu'il convient d'utiliser le formatage. L'expression sera convenablement interprétée par nerdamer qui produira une représentation adéquat. On pourra donc noter : `${@a x^2 + @b x + @c:$}$`

Utilisation de `expand`

Une difficulté est que nerdamer voudra probablement travailler la forme de l'objet numérique. Avec un trinome, il voudra automatiquement le développer. Que faire si nous désirons une forme développée ?

On peut utiliser la fonction `expand`. Avec l'exemple précédent, on pourra par exemple écrire :

`${expand(@a*x^2 + @b*x + @c):$}$`

L'expression est analysée de sorte que si `@b == 0`, le terme en `x` n'apparaîtra pas. `expand` force un affichage développé.

Bloc de texte

```
<texte:header>
...
</texte>
```

Le plus simple, sert à afficher du texte. `header` permet d'indiquer un titre mais n'est pas requis.

Au lieu de `texte` on dispose de quelques variantes :

- `texte` ou `text`, bloc ordinaire
- `warning` pour un formatage d'alerte
- `info` pour un formatage d'info
- `help` ou `aide` pour un bloc d'aide qui se replie. Il faut appuyer le bouton pour afficher l'aide.

Bloc formulaire

```
<form:header>
...
</form>
```

Ce bloc est essentiel pour le déroulement des exercices car c'est lui qui prend en charge les questions / réponses.

En effet, quand l'exercice s'affiche, il affiche les blocs dans l'ordre et quandd il arrive à un formulaire il s'arrête en attendant que l'utilisateur réponde aux questions. Quand l'utilisateur a répondu, l'affichage se met à jour et l'exercicce se poursuit.

S'il s'agit d'un exercice qui a déjà été exécuté et que l'on rejoue, les réponses sont déjà données. Le formulaire le détecte et affiche directement la suite.

Un formulaire contiendra donc des input divers et des blocs de texte.

Bloc input

Il s'agit de l'input de base.

```
<input:name>
...
</input>
```

Un champ input se traduira par une zone de saisie dans laquelle on peut entrer du texte. Les paramètres sont ici très importants, il faut détailler.

- header : c'est l'identifiant de la réponse telle qu'elle sera sauvée dans la BDD. Il faut donc que tous les input en ait un différent. De plus, cet identifiant ne devrait pas rentrer en conflit avec ceux déjà déclaré avec `@` dans l'initialisation et les options.
- tag : ce qui sera affiché pour représenter la valeur demandée. Un affichage Tex est possible. Par exemple x_A.
- solution : la valeur attendue. Il est possible de fournir un tableau. Dans ce cas, la réponse sera considére

valide du moment qu'elle correspond à au moins un item de solution.
- tagSolution : on peut désirer forcer un affichage de solution. On peut le faire avec tagSolution. Voici un exemple d'usage :

```
@f = (x+5)*(x+7)
<form>
<input:d>
<tag:Développer ${@f:$}$/>
<solution:@f/>
<format:expand/>
<keyboard:square/>
<tagSolution:${expand(@f):$}$/>
</input>
</form>
```

Format

Le paramètre format permet d'indiquer le format attendu.

Voici les formats reconnus :

- numeric : on veut une expression dans laquelle il n'y a aucune variable et cette expression doit être développée un minimum. On n'acceptera pas `(1+2)/3`.
- round:2 : on veut un nombre arrondi, dans cet exemple à deux chiffres après la virgule.
- erreur:0.2 : demande un nombre. Sa valeur ne doit pas excéder une erreur de 0.2 dans l'exemple.
- empty : indique que l'on accepte l'ensemble vide. On peut alors répondre par `'vide'` ou `'∅'`
- infini : indique que l'on accepte un infin comme `'+inf'` ou `'-∞'`. Le signe est obligatoire même pour +
- expand : indique que l'on attend une expression développée.

Il ne serait pas logique de proposer à la fois numeric et round mais on peut demander empty et numeric par exemple.

On pourra donc utiliser : `<format:numeric/>` dans le cas où on veut permettre divers types de réponses :

```
<format:numeric/>
<format:empty/>
```

Les formats `infini` et `empty` auront également pour effet d'ajouter un bouton adapté au input.

Clavier

keyboard: permet d'ajouter un bouton pour des expressions mathématiques comme `sqrt`

- power
- sqrt
- square
- cube
- infini
- empty

Ainsi un bloc typique serait :

```
<input:x>
<tag:$x$/>
<solution:@gx/>
<format:numeric/>
<keyboard:sqrt/>
<keyboard:power/>
</input>
```

Dans ce cas on ajoute deux boutons, un pour racine et l'autre pour la puissance.

Si vous ajoutez un bloc d'aide dans un input, alors il y aura un bouton d'aide directement dans le input.

les keyboard `infini` et `empty` sont ajoutés automatiquement si on a précisé les format infini et empty.

Attention : n'essayez pas de mettre directement une formule comme solution. Le mieux est d'affecter au préalable une variable contenant la solution.

```
@sol = ...
<input:a>
...
<solution:@g/>
...
</input>
```

Bloc Radio

Utiliser pour une question à choix multiple. Une seule réponse possible.

Voici un exemple :

```
<radio:n>
0 => aucune solution
1 => une solution
2 => deux solutions
<solution:@gn/>
</radio>
```

Ici `@gn` qui aura été préalablement initialisé, contiendra une valeur parmi 0, 1, 2.

À l'affichage, les différentes possibilités sont mélangées, mais bien sûr, chaque réponse reste convenablement associée à sa clé.

Bloc Liste de choix

```
<choix>
1 => Le choix 1
2 => Le choix 2
3 => Le choix 3
</choix>
```

Ce bloc permet d'afficher une liste d'items associés à un pictogramme coloré dépendant de l'index indiqué. L'index doit être entre 1 et 7 inclus.

On dispose des paramètres :

- `shuffle:true` qui permet de mélanger à l'affichage (pas mélangé par défaut)
- `onlysquares:true` pour demander à n'avoir que des carrés. Par défaut true.

Les pictogrammes peuvent servir pour les daltoniens. On peut donc choisir de ne pas utiliser les pictogrammes.

Formulaire liste de choix

```
<formchoix:name>
1 => valeur 1
2 => valeur 2
1 => valeur 3
</formchoix>
```

Il s'agit d'un formulaire, donc d'un bloc attendant une validation. `name` est l'identifiant de la réponse.

Ce formulaire se présentera comme une liste de choix avec des pictogrammes colorés. Chaque item est un bouton. Quand on clique, on change le pictogramme et la couleur.

Voyons un exemple d'utilisation :

```
<choix>
1 => $\mathcal{N}$
2 => $\mathcal{Z}$
3 => $\mathcal{D}$
</choix>

<formchoix:a>
2 => -5
1 => 8
1 => 0
3 => 5,4
2 => -1
</formchoix>
```

Le premier bloc définit les couleurs (et donc indices) associées aux ensembles. On demande ensuite à l'élève de choisir. Il devra choisir 2 pour -5, 1 pour 8, etc.

On peut préciser le param `max:4/ pour indiquer que l'on autorise le choix à aller jusque 4 même si ce n'est pas une des réponses prévues.

Remarque : les indices peuvent se répéter dans `formchoix` car plusieurs items peuvent avoir la même réponse. Ici, 8 et 0 sont tous deux des entiers naturels.

On dispose là encore de paramètres :

- `shuffle:false` pour le mélange (mélangé par défaut)
- `onlysquares:true` pour ne pas utiliser les pictogrammes

Bloc Table

```
<table>
...
</table>
```

Permet d'afficher une table.

Il faudra lui fournir un paramètre `rows` qui est un tableau de tableaux.

On pourra lui fournir des tableaux `rowheaders` et `colheaders`.

Voici un exemple d'initialisation.

```
# init
# tableau a = [0, ..., 9]
@a <:10>= @__i
# tablau b = [a[i]*2 + alea]
@b <:@a>= <P:@__v 2 * 0 5 Alea.entier +>

# affichage de la table
<table>
<rowheaders[]:$x$/>
<rows[]:@a/>
<rowheaders[]:$f(x)$/>
<rows[]:@b/>
</table>
```

Bloc tkztab

Il s'agit d'un bloc pour tableau de variations ou de signe, nommé ainsi en relation avec le module tikz latex tkz-tab dont j'ai repris certaines notations.

```
<tkztab>
<color:red/>
<xlist:$1$,$a$,3,$+\infty$/>
<sign:$A$:1:,+,z,-,z,+,/>
<sign:$f(x)$:2:,-,d,+,z,-/>
<var:$f(x)$:3:-/1,+D-/$x+3$/$-\infty$,+/0,-/$-\infty$/>
</tkztab>
```

D'abord quelques paramètres :

- `xlist` est **obligatoire**. C'est un tableau ou un texte indiquant les valeurs des antécédents. On peut mettre des éléments latex qui seront transformés par katex. Naturellement, comme pour tous les params, on a le droit de placer des variables avec un formatage comme `${@a:$}$`.
- `color` pour choisir la couleur du tableau. On peut utiliser une couleur html standard comme `red` ou un numéro ce qui permet de faire le liens avec le bloc de choix.
- `tag` pour choisir l'entête de la première ligne. Par défaut, c'est `$x$`
- `espcl` pour choisir l'espacement entre les items, en pixels, dans la partie droite du tableau. 150 par

défaut.

- `lgt` pour choisir la largeur du bloc d'entête, 100 par défaut
- `pixelsperline` pour choisir la hauteur d'une unité verticale, 40 par défaut
- `headerHeight` pour choisir la hauteur de la ligne d'entête, exprimée en unité

Noter que le SVG s'ajuste à l'espace disponible en largeur.

Ensuite viennent les contenus à proprement dit.

Ligne tableau de signe

La forme est `<sign:$f(x)$:2:,-,d,+,z,-/>`

- `sign` indique qu'il s'agit d'une ligne tableau de signe
- `$f(x)$` est l'entête (obligatoire)
- `2` est la hauteur de la ligne, en unité (obligatoire, minimum 1)
- vient ensuite le contenu `,-,d,+,z,-,` qui respecte la syntaxe tkz-tab

La ligne sera tronquée ou complétée en fonction de la taille attendue avec le `xList` du parent. Si par exemple xList a 3 items, on aura donc 3 valeurs de x et deux intervalles, donc la ligne devrait avoir 5 éléments. Les éléments de rangs pairs correspondent aux valeurs de x et ceux de rang impair correspondent aux intervalles, donc aux signes.

- un signe peut être `+` ou `-` (autre ignoré)
- en face d'un x, on peut avoir `z` pour zéro, `d` pour interdit et `t` pour une cloison neutre (autre ignoré)

Naturellement, comme pour tous les params, on a le droit de placer des variables avec un formatage comme `${@a:$}$`.

Ligne tableau de variation

Exemple `<var:$f(x)$:3:-/1,+D-/$x+3$/$-\infty$,+/0,-/$-\infty$/>`

- `var` signale que c'est une ligne de variations
- `$f(x)$` est l'entête (obligatoire)
- `3` est la hauteur en unités (obligaoire, minimum 3)
- vient ensuite le contenu `+D-/$x+3$/$-\infty$,+/0,-/$-\infty$` qui respecte la syntaxe tkz-tab.

Le nombres d'items de la ligne doit correspondre à la taille du `xList` parent. Il est complété ou tronqué au besoin.

Les items autorisés ont toujours la forme : `pos/tag/tag`. Les tags sont optionnels et selon les cas, une seul ou deux sont prises en compte.

Chaque fois, `pos` permet d'indiquer la forme. On a les cas de tkz-tab :

- `-`, cas normal, en bas
- `+`, cas normal, en haut
- `-D`, valeur interdite avec une valeur en bas sur la gauche
- `+D`, valeur interdite avec une valeur en haut sur la gauche

- `D-`, valeur interdite avec une valeur en bas sur la droite
- `D+`, valeur interdite avec une valeur en haut sur la droite
- `-D-`,
- `-D+`,
- `+D-`,
- `+D+`,
- `R`, position igorée

Naturellement, comme pour tous les params, on a le droit de placer des variables avec un formatage comme `${@a:$}$`.

Bloc Graph

Permet de tracer un graphique avec **jsxGraph**.

```
<graph>
...
</graph>
```

Le bloc a lui-même quelques paramètres :

- xmin: bord gauche, par défaut à -5
- xmax: bord droit, par défaut à +5
- ymin: bord bas, par défaut à -5
- ymax: bord haut, par défaut à +5
- zoom: autorise ou non le zoom. Par défaut à false.
- pan: autorise ou non le déplacement de la vue. Par défaut à false.
- axis: affiche ou non les axes. Par défaut à true.

Ensuite, on peut ajouter des sous-blocs pour les différents objets.

point

```
<point:name>
...
</point>
```

On dispose des paramètres :

- x
- y
- name: pour l'affichage d'un nom
- on: alors le point est un glider qui glisse sur un objet. Il faut choisir le nom d'un objet graphique défini préalablement.
- color: on peut choisir une couleur standard comme blue, red... ou un indice ce qui choisira parmi les couleurs des blocs choice

- size: pour la taille du point
- fixed: par défaut le point est mobile. On peut le fixer avec `fixed:true`

function

```
<function:name>
...
</function>
```

On dispose des paramètres :

- expression: l'expression de la fonction
- xmin: par défaut le bord gauche de la fenêtre
- xmax: par défaut le bord droit de la fenêtre
- color, idem que pour point
- strokeWidth: épaisseur du trait
- dash: pointillé. Par exemple `dash:2`

Voici deux exemples : Dans le premier on crée la courbe d'une fonction affine et un point mobile sur cette courbe.

```
<function:f>
<expression:3x+12/>
</function>
<point:A>
<on:f/>
</point>
```

Dans ce 2e exemple, on crée une expression aléatoire d'un polynome de degré 3.

La fonction `Alea.lagrangePolynome` tire `n+1` points au hasard, à coordonnées entières et dans le rectangle défini par xmin, ymin, xmax, ymax. Avec ces points elle calcule le polynome d'interpolation de Lagrange.

```
@p = <P:-2 -5 5 5 3 Alea.lagrangePolynome>
<graph>
  <xmin:-2/>
  <function:g>
    <expression:@p/>
  </function>
</graph>
```

Bloc shuffle

À l'exécution, mélange ses enfants et renvoie ces enfants. Peut servir par exemple si on veut produire des tableaux de variations et les mélanger.

```
<shuffle>
...
</shuffle>
```

Les fonctions disponibles

Module Alea

- `Alea.entier` reçoit `xmin` et `xmax` et renvoie un entier aléatoire entre `xmin` et `xmax` compris.
- `Alea.signe` renvoie `-1` ou `1` aléatoirement
- `Alea.lagrangePolynom` reçoit `xmin`, `ymin`, `xmax`, `ymax` et `n`. Renvoie un polynome de degré `n` interpolant `n+1` points de coordonnées entières pris dans le cadre.

Module table

- `Table.indice` reçoit `val` et `tableau`. Renvoie l'indice de la première occurence de `val` dans `tableau`. `-1` si absent.
- `Table.indices` reçoit `val` et `tableau`. Renvoie un tableau contenant tous les indices des occurences de `val` dans `tableau`
- `Table.size` reçoit `tableau` et renvoie la taille du tableau.
- `Table.sum` renvoie la somme des éléments du tableau
- `Table.product` reçoit `tab1` et `tab2` qui doivent être de même taille et renvoie un tableau des `it1*it2`
- `Table.average` reçoit `tableau` et renvoie la moyenne des éléments du tableau. Erreur si tableau vide.
- `Table.average2` reçoit les tableaux `values` et `effectifs`, de même taille, et renvoie la moyenne.
- `Table.variance` reçoit `tableau` et renvoie la variance des éléments du tableau. Erreur si tableau vide.
- `Table.variance2` reçoit les tableaux `values` et `effectifs`, de même taille, et renvoie la variance.
- `Table.std` reçoit `tableau` et renvoie l'écart-type des éléments du tableau. Erreur si tableau vide.
- `Table.std2` reçoit les tableaux `values` et `effectifs`, de même taille, et renvoie l'écart-type.
- `Table.mediane` reçoit `tableau` et renvoie la médiane des éléments du tableau. Erreur si tableau vide.
- `Table.mediane2` reçoit les tableaux `values` et `effectifs`, de même taille, et renvoie la médiane.
- `Table.quantile` reçoit `tableau` et `q` et renvoie la quantile `q` des éléments du tableau. Erreur si tableau vide.
- `Table.quantile2` reçoit les tableaux `values` et `effectifs`, de même taille, et `q`, et renvoie la quantile.
- `Table.ECC2` reçoit les tableaux `values` et `effectifs` et une `valeur`. Renvoie l'effectif total de tous les individus dont la valeur est inférieure ou égale à `valeur`
- `Table.min` reçoit un tableau et renvoie la valeur minimum.
- `Table.max` reçoit un tableau et renvoie la valeur maximum.
- `Table.sortFreqs` reçoit un tableau et renvoie un tableau de tableau `t` avec `t[0]` contenant la liste des valeurs en ordre croissant et `t[1]` la liste des effectifs.
- `Table.filter` reçoit un tableau, un opérateur parmi `==, !=, <, <=, >, >=`, et une valeur. Renvoie le tableau des items satisfaisant le test.
- `Table.toBrut` reçoit les tableaux `valeurs` et `effectifs` et reconstitue la série brute, dans l'ordre.

Module Calc

- `*` fait une multiplication. Attention, en notation polonaise inversée donc par ex `<P:3 5 *>` pour faire 3×5
- `-`, `+`, `/` : idem
- `float` pour obtenir un float
- `abs` pour valeur absolue
- `sign` renvoie `-1` pour un négatif et `+1` pour un positif
- `solve` reçoit `left`, `right` (deux membres de l'équation) et `name` nom de la variable. Renvoie les solutions
- `sub` reçoit `expression`, `name` et `value` et renvoie l'expression où on substitué `name` pour `value`. Permet en particulier de calculer $f(x)$.
- `round` reçoit une valeur `x` et un `entier`. Arrondi à n digits.

Module Dist

- `Dist.binomial` reçoit `n` et `p` et simule un aléa $\mathcal{B}(n;p)$
- `Dist.binList` reçoit `count`, `n` et `p` et renvoie un tableau de `count` simulations de $\mathcal{B}(n;p)$
- `Dist.binCDF` reçoit `k`, `n` et `p` et renvoie la probabilité $p(X\leqsant k)$ avec X suivant $\mathcal{B}(n;p)$
- `Dist.binPDF` reçoit `k`, `n` et `p` et renvoie la probabilité $p(X=k)$ avec X suivant $\mathcal{B}(n;p)$