Une fois l'A.S.T. construit, l'objectif a été de créer un ensemble de méthodes visant à interpréter ce dernier et donc visant à interpréter la commande initialement entrée par l'utilisateur.

Il a été décidé, au même titre que pour la génération de l'arbre et par conséquent en tenant compte du nombre d'instructions à implémenter, de gérer l'ensemble de l'interprétation dans une classe : `InterpreterArbre` (package `arbre`).

Nous commencerons par détailler le cheminement (récursif) d'une interprétation (cas général, abstrait). Puis, nous donnerons ensuite des exemples de cette succession d'étapes grâce à des cas concrets.

Cheminement d'une interprétation

Tout commence dans la classe `Arbre`, où l'on trouve une méthode `interpreterArbre`. Cette méthode va renvoyer sous forme d'une chaine de caractères, le résultat d'une interprétation. Ce retour est notamment (et exclusivement, dans le cadre de ce projet) utile pour l'utilisation de la commande `return`, sur laquelle nous reviendrons. Afin d'obtenir ce résultat, il est fait un appel à la méthode `interpreterArbreSyntaxique`, que l'on retrouve (déclarée en `static`) dans la classe `InterpreterArbre` que nous introduisions précédemment.

Nous nous trouvons désormais dans la classe `InterpreterArbre`, où est déclarée la méthode `interpreterArbreSyntaxique`. Cette dernière ne fait (presque) qu'une seule chose : interpréter le nœud racine de l'arbre. Tout s'explique de par la structure en arbre avec laquelle on travaille. En effet, le véritable avantage lorsque l'on travaille avec ce type de structure, c'est l'utilisation de la **récursivité**. De ce fait, la méthode `interpreterNoeud` est récursive, et nous allons maintenant nous préoccuper de cette dernière.

Interpréter un nœud consiste à effectuer une action qui est fonction du contenu de ce nœud (autrement dit, fonction de l'instruction qu'il contient). La méthode `interpreterNoeud` n'est donc qu'une passerelle, de laquelle on va appeler une méthode spécifique en fonction de chaque instruction rencontrée. Cette méthode sera donc appelée dès qu'un nœud contiendra une instruction. Notre projet permettra ainsi de traiter tous les cas suivants :

- ✓ *assignation* (d'une valeur à une variable)
- ✓ *condition* (structure « if »)
- ✓ *boucle* (structure « while »)
- ✓ *return* (retour d'une expression évaluée)
- ✓ *séquences de commandes* (séparées par des « ; »)
- ✓ *let* (déclaration d'une variable locale ou aliasing local)

On retrouvera, pour chacun de ces différents cas, une méthode d'interprétation. Par exemple, pour le cas d'une assignation, on trouvera une méthode `interpreterAssignation`. Cette dernière prendra systématiquement en paramètre un nœud qui contiendra le symbole de l'instruction correspondante («`:=` »), grâce à la méthode `interpreterNoeud`.

Nous allons maintenant nous intéresser au corps de ces différentes méthodes d'interprétation. Notons dans un premier temps un point essentiel : elles auront toutes recourt une ou plusieurs fois à la méthode `interpreterNoeud`. Et c'est donc dans ces appels que se créée la récursivité dont on parlait précédemment. Ensuite, ces méthodes auront aussi besoin d'un moyen pour évaluer un nœud, autrement dit trouver sa valeur. Un nœud, dans ces cas-là, peut donc soit directement contenir une valeur (arithmétique ou booléenne), soit contenir une variable, soit une expression (qui pourra être arithmétique ou booléenne également). C'est ici qu'intervient la méthode `trouverValeur`. Elle prend un nœud en paramètre et va renvoyer la valeur de ce nœud (ou la valeur

« qui se cache derrière ce nœud » si c'est une variable ou une expression). Prenons le cas simple d'une assignation, les étapes seront les suivantes : trouver le nom de la variable à laquelle on souhaite assigner une valeur, trouver la valeur que l'on souhaite assigner, et ajouter cette assignation à la mémoire (cf. paragraphe suivant). Grâce aux méthodes précédemment décrites, on dispose de tous les outils pour interpréter une assignation correctement. On ne détaillera pas ici les algorithmes des autres interprétations, l'essentiel à comprendre étant dans ce qui a été décrit jusque-là, et dans ce qui le sera juste après.

Venons-en à la mémoire de notre interpréteur. Bien évidemment, il nous a fallu trouver un moyen pour sauvegarder les valeurs assignées à nos variables (afin de construire notre environnement). Pour cela, nous avons créé un objet `Memoire` qui contiendra ces informations. Il est unique pour chaque exécution de notre programme. Il permettra ainsi d'associer une valeur à une variable, qu'elle soit globale ou locale (temporaire).

Intervient ici la notion de variable temporaire. Cette fonctionnalité a été implémentée à cause de l'instruction let. En effet, en l'utilisant, on peut définir une variable qui sera locale, c'est-à-dire que l'on pourra la modifier dans le `let`, mais qu'une fois en dehors de ce dernier, les modifications ne seront plus effectives (on revient à la mémoire de niveau précédent, puisque l'on peut avoir plusieurs let successifs). Ainsi, en plus d'une mémoire « principale » (globale), on peut disposer de une ou plusieurs mémoires dites « temporaires » qui seront créées à chaque utilisation d'un `let`, puis supprimer à la fin de ces derniers. Ainsi, lorsque l'on cherchera la valeur en mémoire d'une variable, on prendra soin de toujours vérifier dans un premier temps dans la dernière mémoire temporaire créée, puis ensuite dans l'avant-dernière, et ainsi de suite. Grâce à ce mécanisme, nous pouvons manipuler des variables à chaque instant sans se poser systématiquement la question de leur situation dans la mémoire.

Exemples d'interprétations

Exemples d'une assignation :

⇨ Commande :

```
>>> e := 1
```

Arbre :

```
:=
      e
      1
```

⇨ Interprétation :

Aucun affichage, mais dans la mémoire, on associe à `e` la valeur 1.

Exemple d'une structure « if » puis d'assignation :

⇨ Commande :

```
>>> if e > 0 then c := e + 1 else d := 2 - 1
```

⇨ Interprétation :

Aucun affichage, on a évalué la condition « `e > 0` » à `true`, et donc interpréter l'assignation après le `then`, ce qui revient à affecté 2 à `c`.

Exemple de séparation de commandes, de let, d'assignation et de return :

⇨ Commande :

```
>>> let a in a := 1 ; return a end ; return 5 > 4
```

⇨ Arbre :

```
;
      let
            a
            ;
                  :=
                        a
                        1
                  return
                        a
      return
            >
                  5
                  4
```

⇨ Interprétation :

On sépare le let et l'instruction après le « ; » (possible grâce au end du let), puis on interprète le let, qui affecte dans une mémoire temporaire la valeur 1 à a. Puis, on détruit cette mémoire temporaire (a reprend sa valeur initiale, ici, aucune), et on affiche l'évaluation de l'expression « 5 > 4 » qui est vrai.

⇨ Affichage :

```
1
true
```