Florian DAMBRINE
Tudor LUCHIANCENCO

GI04

Génération d'un parseur LOGO

Utilisation de ANTLR & Java

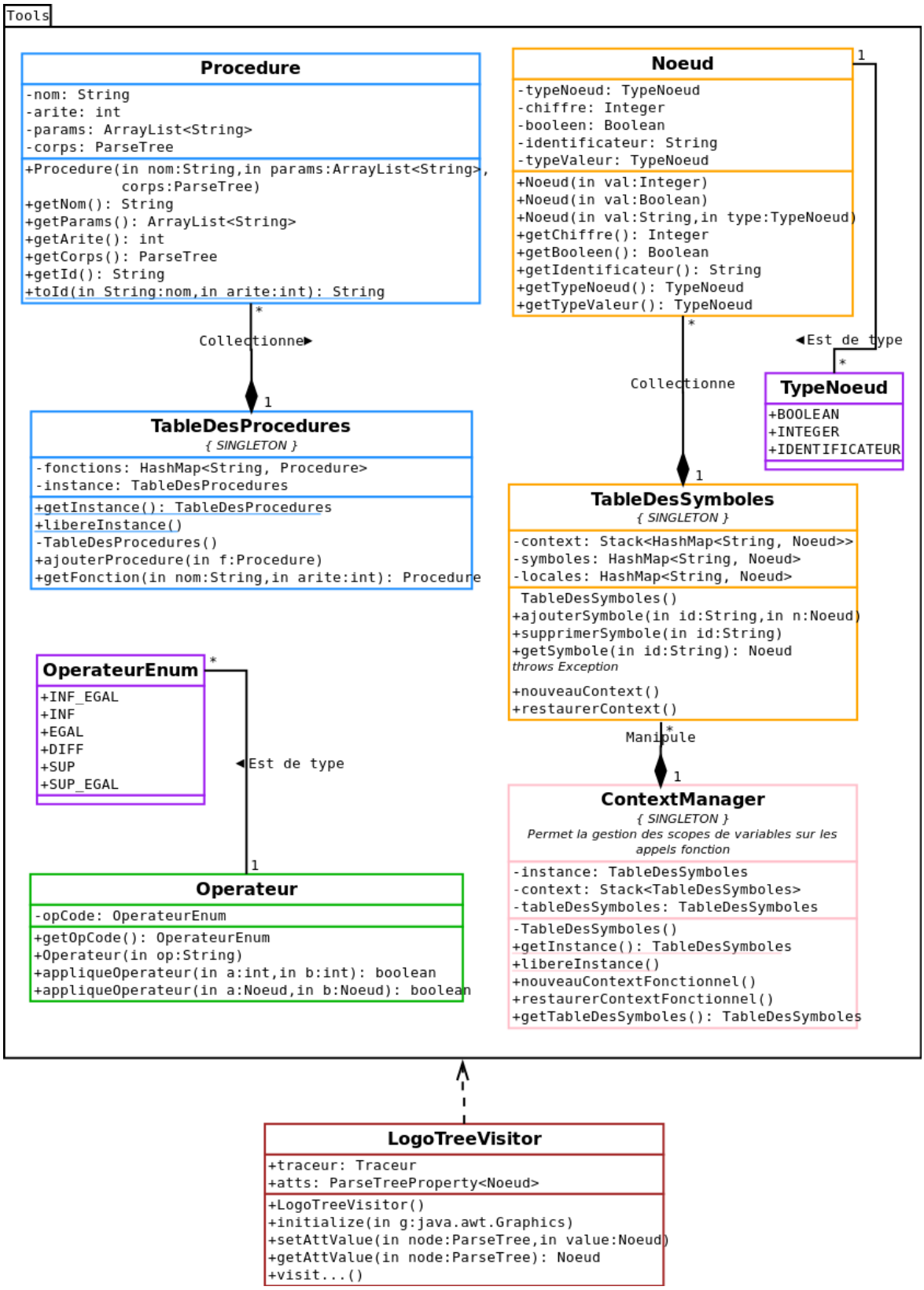
Responsable : Claude MOULIN

Date : 13/06/2013

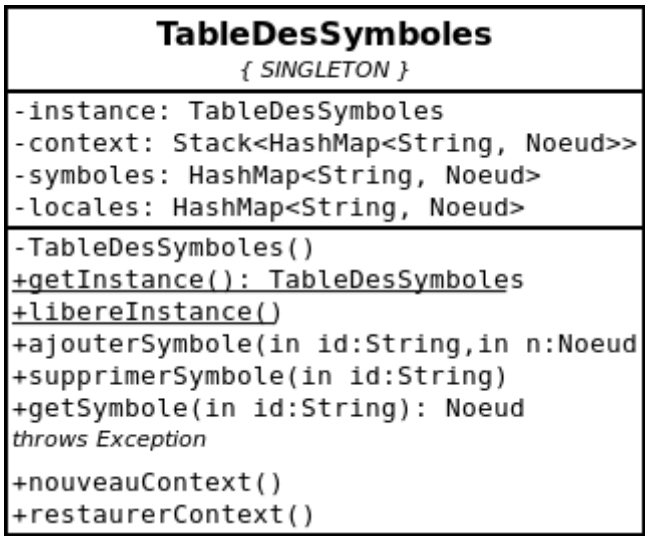
Table des matières

1. Généralités : fonctionnement de l'interpréteur

1.1. Structure générale de l'interpréteur



1.2. Structure de la table des symboles



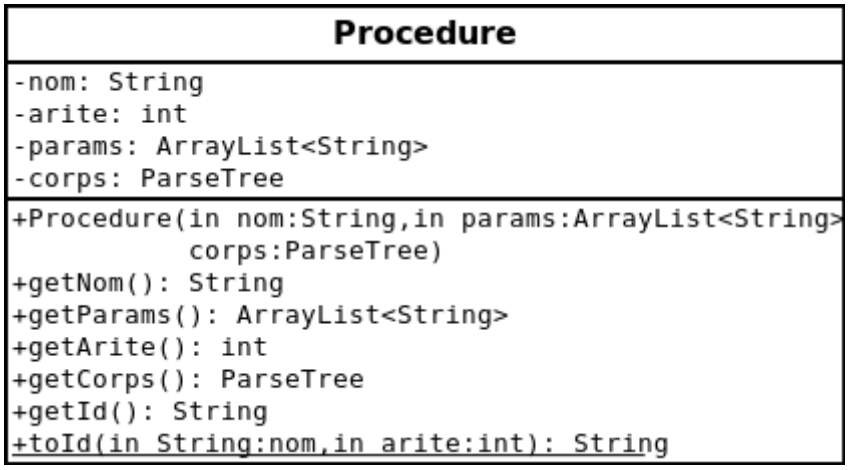
La table des symboles implémente le design pattern singleton afin d'assurer son unicité dans le programme. Cela facilite également la récupération de cet objet en utilisant la méthode **getInstance()**.

La table des symbole contient deux **HashMap**, l'un stocke les variables globales et l'autre les variables locales (respectivement nommés **symboles** et **locales**). La classe contient également une **pile** de **HashMap** nommée **context** qui permet de mémoriser les variables locales lors d'un appel fonction, d'une entrée dans une boucle ou une instruction conditionnelle.

2. Les fonctions et procédures

2.1. Interprétation de la déclaration d'une procédure ou fonction

Les procédures sont gérés à travers l'objet **Procedure**.

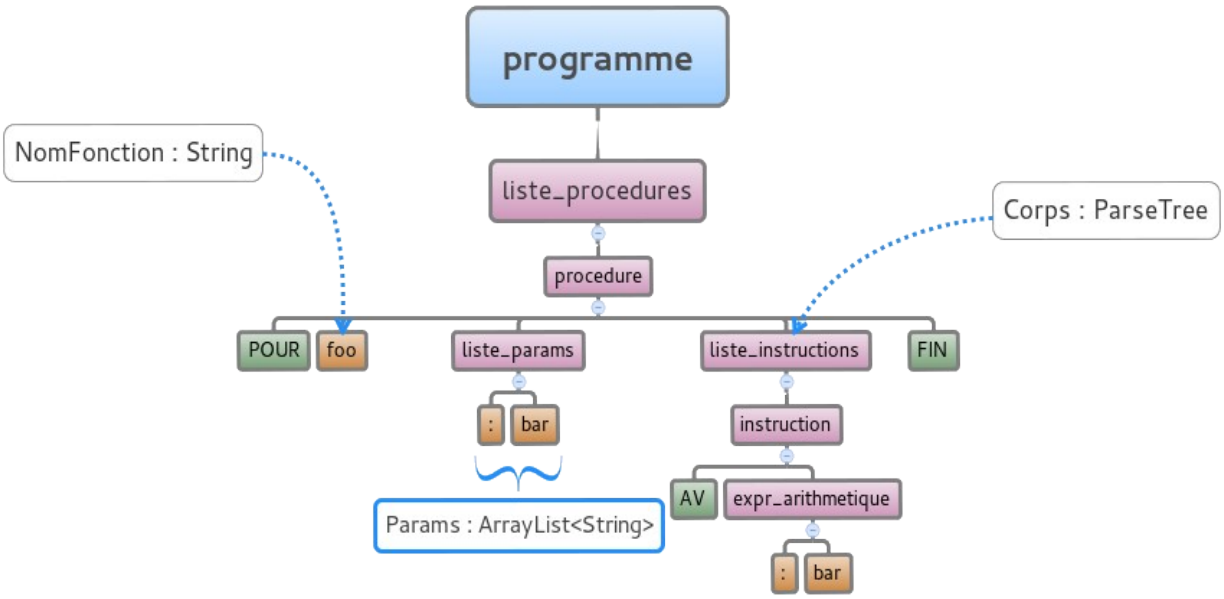


La classe **Procedure** permet de mémoriser le nom de la procédure, le nombre de paramètres, la liste des paramètres ainsi que le corps de la procédures.

Voyons sur cette exemple comment l'arbre est utilisé pour créer un objet **Procedure**

Programme LOGO :

```
POUR foo :bar
   AV :bar
FIN
```



Voici la méthodes JAVA permettant d'interpréter la déclaration d'une procédure :

```
@Override
public Integer visitProcedure(ProcedureContext ctx) {
    //Création de l'objet procédure

    String nomFonction = ctx.ID().getText();

    ArrayList<String> params = new ArrayList<String>();

    for(int i = 0; i < ctx.liste_params().getChildCount(); i++){
        if(!ctx.liste_params().getChild(i).getText().matches(":")){
            System.out.println(ctx.liste_params().getChild(i).getText());
            params.add(ctx.liste_params().getChild(i).getText());
        }
    }

    Procedure f = new Procedure(nomFonction, params, ctx.liste_instructions());

    TableDesProcedures.getInstance().ajouterProcedure(f);

    return 0;
}
```

2.2. Gestion de la portée des variables dans les instructions conditionnelles

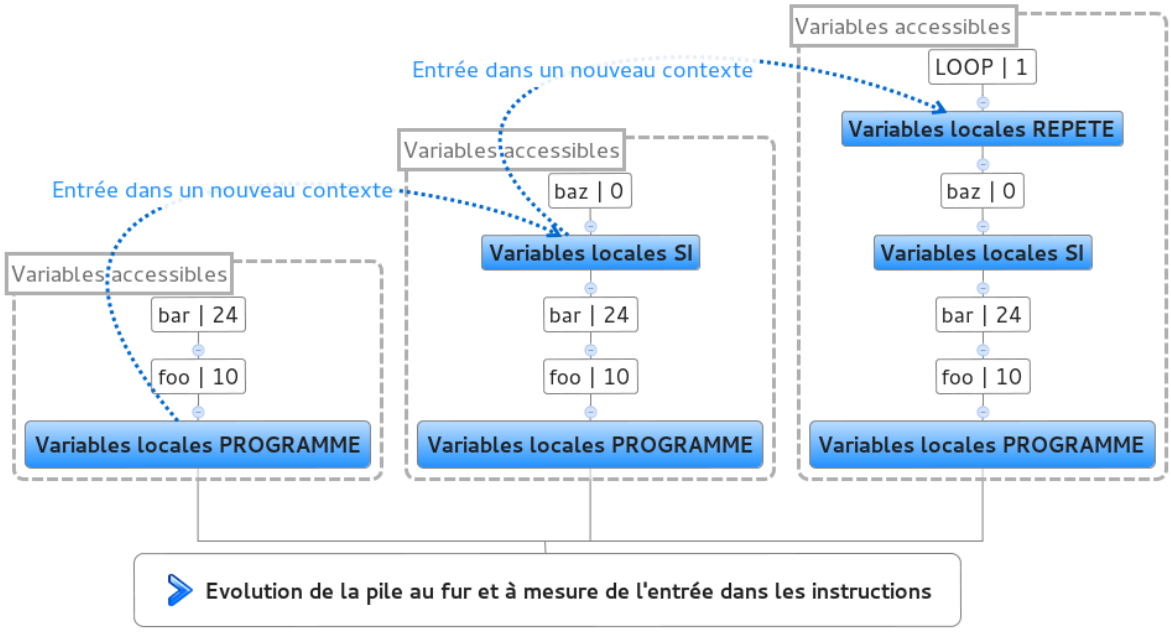
Nous avons fait le choix de considérer une variable déclarée dans un bloc comme locale à ce bloc. Ce choix nous permet, entre autre, d'utiliser indépendamment les variables **LOOP** dans une imbrication de boucles **REPETE**.

Pour garder une certaine cohérence, nous avons décidé d'étendre le concept de variables locales à un bloc de type **TANQUE**, **SI**, **SINON**.

Le principe de fonctionnement de la portée d'une variable fonctionne grâce à une pile qui permet d'empiler les **Maps** de variables locales lors de l'entrée dans un bloc.

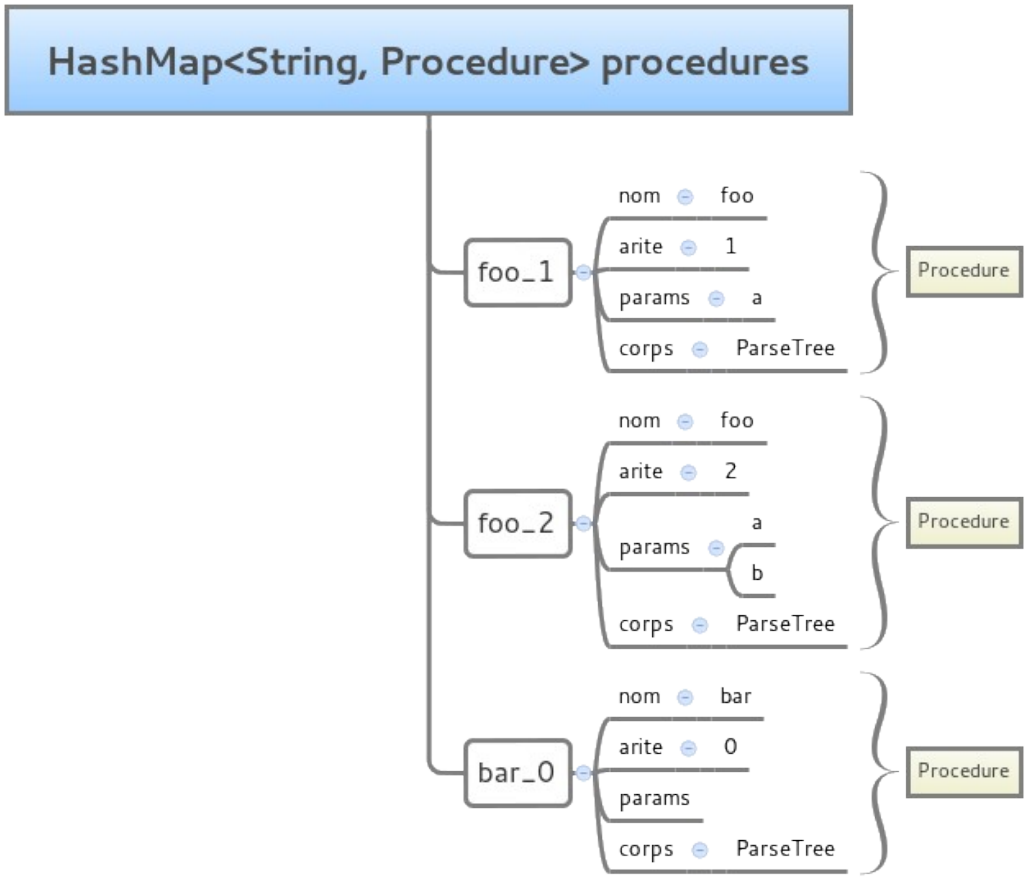
```
LOCALE "foo
DONNE "foo 10
LOCALE bar
DONNE "bar 24
SI true [
     LOCALE "baz
     DONNE "baz 0
     REPETE :foo [
          AV LOOP * :bar
     ]
]
```

Le schéma ci-dessous reflète l'état de la pile correspondant au programme ci-dessus:



2.3. Le stockage des procédures et des fonctions

De la même façon que les symboles sont stockés dans une **TableDesSymboles**, les procédures et fonctions sont stockées dans une **TableDesProcedures**. Celle-ci contient donc une **Map** permettant d'associer à un nom un objet **Procedure**.



Le schéma ci-dessus permet de représenter la façon dont la **Map** stocke les différentes procédures.

Notons que la clé qui permet d'accéder à la fonction est la concaténation du nom de la fonction avec son nombre de paramètres. La validité d'un appel de procédure est donc simplement contrôlé grâce à la méthode **containsKey(pattern)** de l'objet **Map** en lui passant pour pattern la concaténation du nom de la fonction avec son nombre de paramètres.

2.4. L'appel d'une procédure ou d'une fonction

Ce chapitre détaille les différentes étapes exécutés lors de l'appel d'une procédure.

2.4.1. L'appel

L'objet **TableDesProcedures** est rempli à travers le parsing des déclarations de procédures avant de rencontrer les premières appels fonctions.

Il est nécessaire de vérifier l'existence de la procédure avant son exécution. Rappelons qu'un appel de procédure est valide si :

- Son nom existe dans la table des procédures
- Le nombre de paramètres correspond au nom de la procédure

Nous avons vu précédemment que l'objet **TableDesProcedures** permettait de gérer ces problèmes en utilisant une clé particulière pour accéder à la **Map**.

2.4.2. La création d'un nouveau contexte

L'exécution d'une procédure ou d'une fonction nécessite de prendre des précautions sur la portée des variables. Voyons des exemples de codes et critiquons-les pour savoir quel politique adopter quant à la gestion de la portée des variables.

Exemple 1 :

```
POUR foo
   AV :bar
FIN

DONNE "bar 100
```

Dans ce premier exemple, La variable **bar** est déclarée dans la portée globale. Or les bonnes pratiques de programmation déconseillent fortement ce type de pratique sous peine d'observer des effets de bords et de voir ces algorithmes se comporter bizarrement.

CONCLUSION : Étant donné la consigne de l'énoncé, nous avons décider de faire échoué ce type de d'instruction. L'utilisation de variables globales est donc interdite dans les fonctions ou les procédures.

Exemple 2 ;

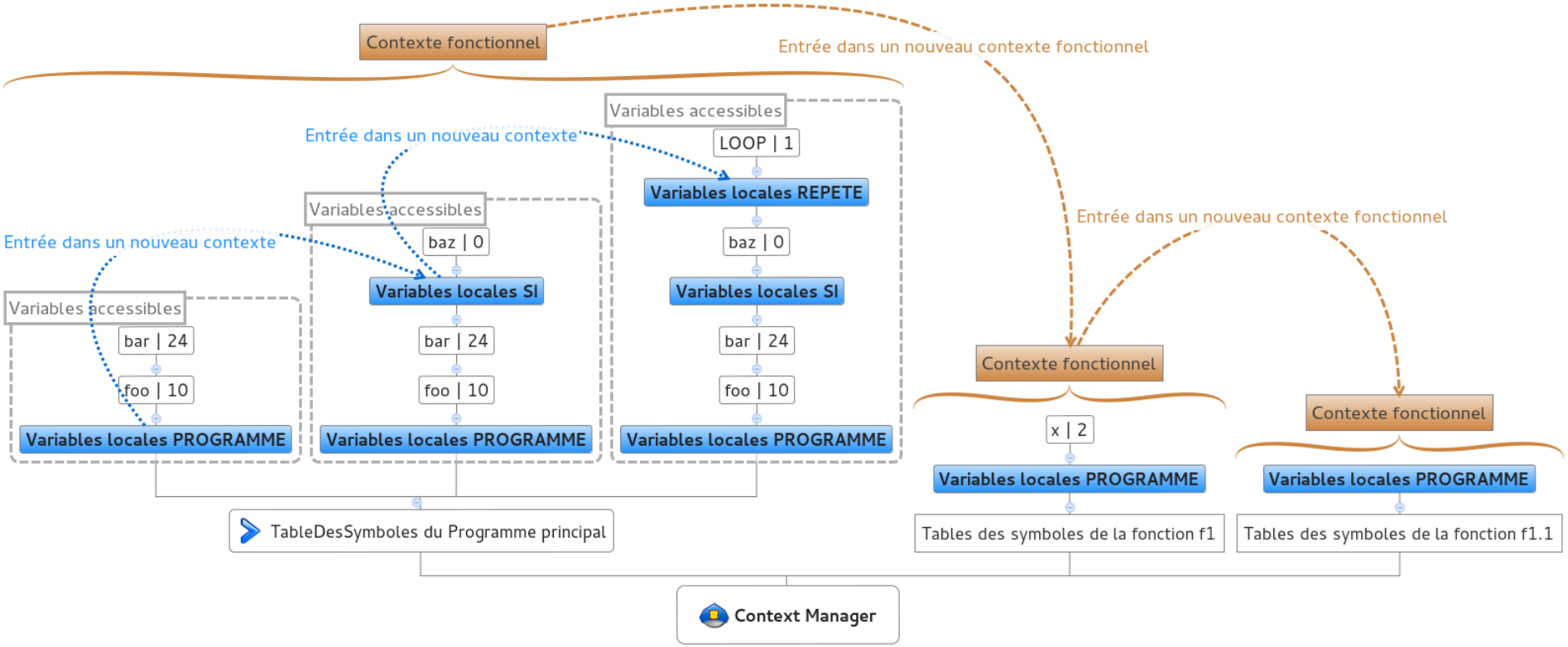
```
POUR foo
   AV :bar
FIN

LOCALE bar
DONNE "bar 100
```

Ce second exemple est complètement surprenant... Cependant, son écriture est syntaxiquement correcte, il faut donc envisager les cas d'erreurs à gérer.

CONCLUSION : Nous ne devons pas utiliser une variable locale dans une fonction qui serait définit dans le programme principal ou encore dans une fonction appelante.

La classe **ContextManager** a été mise en place afin de gérer ces problématiques. Celle-ci contient une pile de **TableDesSymboles**, afin d'allouer un nouveau contexte « fonctionnel ou procédurale » lors de l'appel d'une fonction.

Le schéma page suivante permet d'intégrer l'objet **ContextManager** au schéma de gestion de la portée des variables présenté précédemment :



Pour résumer, une première pile permet de gérer la portée des variables locales lors de l'entrée dans des blocs d'instructions. C'est l'objet **TableDesSymboles** qui s'occupe de gérer cette pile**.**

La limitation de la portée des variables dans les fonctions est, quant à elle, gérée par le **ContextManager** qui grâce au même principe, isole les appels fonctions des variables existantes. Ces contextes sont bien évidemment détruits lorsque la procédure se termine.

2.4.3. Création des paramètres dans la table des symboles

Les valeurs des paramètres sont récupérées depuis l'arbre pour être ajoutées dans la table des symboles en tant que variables locales. Un mapping est réalisé entre l'ordre d'apparition des paramètres dans l'arbre et l'ordre d'apparition des paramètres dans la liste stockée dans la procédure. Les couples sont donc respectivement associés deux à deux.

2.4.4. Exécution du corps de la fonction

L'instruction **visitChildren(ctx)** est effectuée sur le nœud mémorisé dans l'objet **Procedure** pour exécuter le corps de la fonction.

2.4.5. Destruction du contexte

Les contextes de fonction, de variables locales sont détruits en sortie de fonction ou de procédure.

2.5. L'appel de procédures dans des procédures et procédures récursives

Étant donné que nous avons intégré l'objet **ContextManager** pour isoler les contextes de fonction, les appels de procédures deviennent indépendants et fonctionnent comme un appel fonction depuis le programme principal. Les appels de fonctions dans des fonctions ou la récursivité ne pose donc aucuns problèmes.

2.6. Les fonctions et le retour de paramètres

Pour permettre le retour de paramètres, nous utilisons la pile de variables locales de l'objet **TableDesSymboles**. L'instruction **RET** permet de pousser sur la pile la valeur de l'expression. Celle-ci sera récupérée par la fonction **visitX(ctx)** appelante, qui prendra soins de marquer le nœud de l'arbre avec ce résultat avant de détruire le contexte de fonction.

3. Les éléments non traités

Différents éléments du parser mériterais d'être approfondi :

- Réaliser un visiteur d'arbre permettant de valider l'arbre avant de l'exécuter (appels fonctions corrects, variables existantes, etc)
- Proposer des types de variables plus évolués (booléens, flottants, etc)
- Gérer les retours de fonctions avant la fin de la fonction
- Implémenter des instructions plus évoluées comme le **STOP**

4. Les vérifications sémantiques

4.1. Politique de gestion des erreurs

Les différents cas d'erreurs sont directement gérés dans le programme en choisissant la plus part du temps d'ignorer l'instruction qui a posé problème :

- Appel à une variable non déclarée
- Appel de procédure incorrect (nom ou nombre de paramètres)
- Appel à la variable LOOP en dehors de son contexte de boucle

Exemple d'un scénario « catastrophe » :

```
POUR foo:bar
   AV :bar
FIN

AV :toto //accès à une variable non déclarée
TD 90
TG LOOP //accès à LOOP en dehors du contexte
foo (1 2 3) //appel de foo incorrect
foo (100)
```

L'exécution de ce code prends en compte les différentes erreurs et n'exécute pas les différentes instructions invalides. Cependant il réalise quand même les instructions valides comme **TD 90** et **foo(100)**.

5. Conclusion

Ce mini projet nous aura permis d'aborder la façon dont un compilateur est réalisé. Nous aurons pris conscience, au cours des TPs, de la difficulté à gérer les types des variables, la valeur de retour des fonctions, et les contextes liés aux appels de procédures.