TP 3 : Analyse sémantique (1) — Vérification du typage

Description du travail à réaliser Dans ce TP, vous devrez compléter le code java fourni, en implémentant la vérification du typage comme vu en cours.

Remarque : Il vous est possible d'adapter le code fourni pour y intégrer vos réalisations précédentes, ou inversement d'utiliser le code fourni pour enrichir vos travaux du TP2. Si vos extensions à `plk` comprennent au moins les expressions de base et les fonctions (déclarations et appels), vous pouvez donc le garder, sinon étendez-le ou prenez celui fourni.

Dans cette aventure, vous n'êtes pas seul...

Sur Celene, une archive pour le TP3 vous est fournie, qui contient l'architecture de base du code attendu. Vous y trouverez l'équipement suivant :

- Un fichier pour la grammaire (*Soft Compiler Hub* : `sch.g4`), correspondant peu ou prou à ce qui était attendu au TP2. (sans les boucles for, les incrémentations, les déclarations/affectations, qui étaient demandées en dernière partie. Vous pouvez vous en passer, mais il est recommandé de les encoder pour augmenter l'expressivité du langage).

  Notez que dans ce langage, un programme est une suite d'instructions et de déclarations de fonctions.[1] Pour plus de simplicité, toutes les fonctions ont un type de retour (et il n'y a pas de type void). Ainsi, le résultat d'un appel de fonction est toujours une expression et jamais une instruction.

- Un package "ast" contenant la structure de base des classes java nécessaires à la constructions de l'arbre :

  - Toutes les classes de l'AST dont vous aurez besoin, y compris pour la gestion des fonctions, ainsi que l'interface Visiteur. Les types primitifs sont isolés, pour permettre ultérieurement la gestion de types composés (tableaux, tuples,...).
  - Une classe `Position` qui vous permettra de générer des messages d'erreur précis. La position est un champ apparaissant dans chaque `Node` correspondant au numéro de ligne et de caractère dans le code source.

[1] Pour la génération de code, on demandera à ce qu'une des fonctions s'appelle `main`, ce sera le point d'entrée d'exécution.

  - L'`AstBuild` adapté, avec en plus la gestion des positions.
  - Un **visiteur générique** (`BaseVisitor.java`. C'est une classe abstraite qui implémente l'interface visiteur, et dont la fonction est de rappeler n'importe quel visiteur concret sur tous les nœuds de l'AST. Il permet de simplifier l'écriture des visiteurs quand ils ne concernent que certains nœuds; mais vous pouvez également vous en passer en remplacement l'héritage du `BaseVisitor` par une implémentation du `Visitor` si vous préférez.

- Un package "support" avec une classe pour la gestion des erreurs. Chaque visiteur aura une instance de cette classe pour y insérer les erreurs éventuelles. Cela permet de vérifier après chaque étape si une erreur a été détectée, et d'arrêter le programme avant d'essayer de passer aux étapes suivantes. Typiquement, si la construction de la table détecte un problème, le compilateur n'essaiera pas de faire la vérification des types (qui consiste en un second visiteur).

- Un package "semantic" contenant le nécessaire pour la mise en place des visiteurs de la vérification de types.

  - Une classe `MethodSig` qui contient les informations de type associée à une déclaration de fonction (cf cours).
  - Une classe `SymbolTable` contenant les méthodes nécessaires à l'implémentation de la table des symboles telle que vue en cours. Elle est à compléter.
  - Une classe `TableBuilder`. C'est le visiteur qui crée la table à partir de l'AST. Les seules nœeuds qui intéressent ce visiteur sont ceux correspondant aux déclarations et aux blocs. Le visiteur générique se charge de le diffuser sur tout l'arbre.

    Les trois méthodes de visite sont à compléter.
  - Une classe `TypeChecker`. C'est le visiteur qui effectue la vérification de typage sur l'ensemble de l'AST.

    Vous devez coder les méthodes de visite.

  Le code fourni contient des indications en commentaire

- Les scripts de compilation et de nettoyage, comme au TP précédent.

- Une classe `Main` (lecture sur l'entrée standard)

- Un fichier de test adapté à la grammaire fournie; créez d'autres tests pour illustrer en particulier la bonne gestion des erreurs de typage.

1 Première mission : Échauffement

1.1 Test et affichage

Dans un premier temps, assurez-vous que le code compile et s'exécute sans erreur avec le test fourni ('./run.sh ¡ [chemin du fichier test]').

Pour vous assurer d'avoir bien intégré le patron de conception Visiteur, si ce n'est pas déjà fait (TP2), écrivez un visiteur pour afficher le contenu de l'AST de façon lisible avec l'indentution (bonus : coloration syntaxique).

Intégrez l'appel de ce visiteur au Main pour vérifier que votre visiteur parcourt l'AST comme voulu.

1.2 (Facultatif) Encodage

Si ce n'est pas déjà fait, ajoutez à la grammaire les constructions syntaxiques pour `for, x++, x--, int x=1,x+=y,...` et toutes celles que vous voudriez encoder sans modifier l'AST.

2 Deuxième mission : Table des symboles

Remplissez les classes `SymbolTable` et `TableBuilder`, et vérifiez avec des tests que la table construite et les erreurs produites sont bien telles qu'attendues [2].

3 Troisième mission : Vérification du typage

3.1 Vérification du bon typage des expressions

Remplissez la classe `TypeChecker` et produisez quelques tests nécessaires à la vérification de son comportement (notamment, précision des erreurs remontées par le visiteur lorsqu'il rencontre des expressions mal typées).

3.2 Vérifications plus avancées

Enrichissez votre vérificateur de type pour qu'il puisse détecter une erreur quand :

- Une fonction ne retourne pas toujours quelque chose (si elle retourne quelque chose dans une branche conditionnelle, elle doit retourner quelque chose aussi lorsqu'elle passe par une autre branche)
- Une instruction `inst` est ignatteignable, car elle se trouve dans une fonction où un `return` est forcément appelé lors de l'exécution avant que `inst` ne soit rencontré.

[2] Conseil : commencez avec des tests plus simple que celui fourni, en commençant par exemple avec un seul bloc, puis en augmentant progressivement les différentes constructions et imbrications de blocs, d'expressions...

Une autre extension possible : rajoutez en plus des erreurs, des avertissements (`warnings`) pour indiquer à l'utilisateur des éléments de code potentiellement problématiques (je fais appel à votre imagination et aux types d'avertissements de ce genre que vous auriez pu rencontrer).