Mathilde Needham LDD3 - Magistère Informatique

RAPPORT DE DM: KAWA

Compilation

Tous les éléments principaux du langage requis (opérations arithmétiques, gestion des variables, instructions, classes et attributs, méthodes et héritage) ont été implémentés dans les différents fichiers de mon projet. Pour chacun de ces aspects, des fichiers de test dédiés ont été ajoutés afin de valider leur bon fonctionnement.

I. Difficultés rencontrées

Au début du projet, il m'a fallu du temps pour bien distinguer les responsabilités des fichiers *typechecker.ml* et *interpreter.ml*, afin que chacun remplisse sa fonction spécifique sans redondance.
De plus, lorsque j'ai commencé à travailler sur les extensions et améliorations du langage, il a été nécessaire de générer le fichier *conflicts* pour m'assurer qu'aucun conflit n'était introduit par l'implémentation du parseur.
Enfin, comme je l'expliquerai plus en détail ci-dessous, j'ai ajouté un environnement local en paramètre à toutes les fonctions du typechecker et de l'interpréteur. Cela m'a pris un certain temps avant de parvenir au comportement attendu et à un résultat satisfaisant.

II. Extensions

A. Variables locales

Tout d'abord, j'ai introduit la possibilité de déclarer des variables locales dans différentes parties du programme, et pas uniquement à l'intérieur des méthodes. Cela permet, par exemple, de déclarer des variables locales directement dans le main. Ces variables sont alors placées dans un environnement différent de celui contenant les variables globales, déclarées tout en haut du fichier.

De plus, une variable déclarée à l'intérieur d'un bloc conditionnel, tel qu'un if, devient inaccessible en dehors de la portée de ce bloc. Ainsi, la visibilité des variables est mieux gérée, respectant les portées définies. Cette fonctionnalité est vérifiée dans le fichier de test *var3_visibilite.kwa* .

Cet ajout apporte une particularité à mon code. J'ai introduit une nouvelle instruction dans mon fichier *kawa.ml*, **NewVar**(string * typ), qui permet d'ajouter une nouvelle variable locale à l'environnement des variables locales. Par ailleurs, j'ai supprimé le champ *locales* du type *method_def*, car il n'était plus nécessaire. En effet, chaque méthode, en tant que partie indépendante du code, dispose désormais de son propre environnement local, ce qui rend ce champ redondant.

B. InstanceOf

L'extension **InstanceOf** ajoute une nouvelle règle à la grammaire de notre langage, introduisant ainsi une expression supplémentaire. L'expression **InstanceOf**, de la forme (expr*typ), renvoie un booléen indiquant si expr est de type typ.
Le typechecker ne réalise aucune vérification spécifique sur expr ou typ. C'est au niveau de l'interpréteur que la comparaison entre les types est effectuée.

Dans les cas où typ = void ou l'évaluation de expr renvoie une valeur nulle, l'expression InstanceOf renvoie systématiquement le booléen false.

C. Super

Cette extension permet d'appeler une méthode telle qu'elle est définie dans la classe parente de la classe courante.
Dans le fichier *typechecker.ml*, on vérifie d'abord que le contexte d'exécution correspond bien au code d'une méthode appartenant à une classe (c'est-à-dire qu'une variable `this`, représentant l'objet courant, est définie dans l'environnement de variables locales). On s'assure également que la classe courante possède une classe parente, à laquelle le token **super** peut faire référence.
Dans le fichier *interpreter.ml,* l'expression **super** est évaluée de la même manière que l'expression **this**. Toutefois, lors de l'évaluation d'un appel de méthode, une expression **MethCall** de type (expr * string * (expr list)), si la première expression est **super**, on modifie la recherche : au lieu de chercher le nom de la méthode dans la liste des méthodes de la classe courante, on effectue cette recherche dans la liste des méthodes de la classe parente de la classe courante.

D. Déclarations en séries et initialisation en une ligne

Ces extensions vont venir changer le fichier *kawaparser.mly*. J'ai décidé lors de l'implémentation de ces deux extensions que les lignes :

- var int x,y,z;
- var int x= o;
- attribute bool b;

seraient reconnues dans mon parser mais que les lignes :

- var int x, y, z = 0;
- attribute bool b = true;

ne seraient pas reconnues.
Il n'est pas possible de déclarer et d'initialiser plusieurs variables en une seule ligne. De même, les attributs d'une classe peuvent être déclarés en série, mais ils ne peuvent pas être initialisés sur une seule ligne. En effet, chaque attribut est propre à une instance de la classe, c'est-à-dire à un objet spécifique. Initialiser un attribut directement dans la définition de la classe n'aurait donc pas de sens, car les valeurs des attributs doivent être définies pour chaque instance individuelle.

E. Attribut final

J'ai ajouté la possibilité dans une classe de déclarer un attribut comme étant final, c'est à dire que celui n'est pas modifiable en dehors du constructeur de cette classe.
<u>Choix d'implémentation :</u>
Pour cela j'ai modifie le champ *attributes* du type *class_def* dans kawa.ml, qui est passé du type (string*typ) list a ((string*typ)*bool) list. En gardant le côté tuple de cette liste j'ai pu facilement changer les endroits qui appeler cette liste en appelant simplement la liste qui regroupe le premier élément de chaque tuple qui la forme.
Cette extension de demander pas de modifier le fonctionnement de ou d'ajouter des cas à gérer dans le *typechecker.ml*, en effet qu'un attribut soit final ou non, son type reste le même.
Dans le fichier *interpreter.ml,* j'ai rajouté une variable en **is_constructor** de type

(bool*string) qui est initialisé à (false*” “) pour indiquer que nous ne sommes pas dans un constructeur. Lorsqu’on rentre dans un constructeur (le cas **NewCstr** de la fonction eval), on va sauvegarder la variable courante puis changer cette variable à (true*class_name) o`u class_name est le nom de la classe du constructeur dans lequel on se trouve.
Dans le cas Set de la fonction exec, on vérifie si notre variable est finale, et si oui si on est dans un cas ou l’on a le droit de l’initialiser.
De plus, si une classe fille hérite d’un attribut final de sa classe parent, elle pourra tout de même l’initialiser dans son constructeur, car lorsque je vérifie les attributs d’une classe j’appelle ma fonction attributes qui va chercher récursivement tous les attributs de cette classe plus ceux qu’elle hérite de ses classes parents.