CS444 - Projet Compilation Documentation

Les messages d'erreurs

Pour la gestion des erreurs nous avons à disposition ErreurContext.java, ErreurInterneVerif.java et ErreurReglesTypage.java.

Pour la suite nous avons préferé gerer les erreurs avec la classe enum ErreurContext et non par la classe ErreurReglesTypage. En effet, cela nous permet de différencier les erreurs levées et d'afficher des messages spécifiques. D'autre part, si la passe 1 a été correctement faite, il ne devrait pas avoir d'ErreurInterneVerif.

Voici les erreurs contextuelles ainsi que leur description et le message affiché :

- ErreurVariableRedeclaree :
Description : Lors de la déclaration des variables, si une variable a le meme nom qu'une variable déjà déclarée
Message : Erreur contextuelle : Variable non déclarée
Variable [nom_variable] non déclarée ... ligne [num_ligne]

- ErreurVariableNonDeclaree
Description : Si on utilise une variable qui n'a pas été déclarée au préalable
Message : Erreur contextuelle : Variable non déclarée
Variable [nom_variable] non déclarée ... ligne [num_ligne]

- ErreurTypageNonCompatible
Description : Lors d'une affectation, une operation binaire ou une operation unaire et que le type attendu n'est pas le bon.
Message : Erreur contextuelle : Opération non compatible :
Opération Affect : Types [type1] et [type2] incompatibles ... ligne [num_ligne]

- ErreurTypeIndefini
Description : Lors d'une déclaration d'une variable, on lui donne un type indefini
Message : Erreur contextuelle : Typeindefini
Le type[type_indefini] est indefini dans l'environnement ... ligne [num_ligne]

- ErreurIdentNomReserve
Description : Lors d'une utilisation d'un mot reservé
Message : Erreur contextuelle : Identificateur déclaré avec un nom réservé
[mot_reservé] est un nom réservé ... ligne [num_ligne]

- ErreurNonRepertoriee
Description : Aucune des erreurs précédentes, normalement on utilise jamais cette erreur.
Message : Erreur contextuelle : Erreur non repertoriee

Architecture passe 2

– Décoration de l'arbre

Nous avons décoré l'arbre avec la méthode setDecor(). Decor est une classe qui contient des informations sur les noeuds de l'arbre. Un Decor contient 3 attributs : - Defn qui permet d'ajouter des informations sur les identificateurs - Type qui spécifie le type d'un noeud - Infocode qui servira pour la passe 3

Ainsi on rajoute les informations en fonction du noeud qu'on regarde. ex : pour une constante entière, on rajoute à l'arbre un Decor contenant un DefnConstInteger qui contient la valeur de la constante pour un intervalle on ajoute à l'arbre un Decor de Type.Interval qui contient les bornes de l'intervalle Et ainsi de suite...

Dans le cas des identificateurs on définit un Defn qui est une structure spécifique. Elle contient : - sa nature : ex : NatureDefn.Var pour une variable et NatureDefn.Type pour un type - son type - son genre si c'est une variable prédéfinie dans l'environnement

On utilise l'environnement pour sauvegarder les informations contextuelles sur les variables et les types. Dans un premier temps on l'initialise avec les types prédéfinis (integer, boolean, real) ainsi que les mots réservés (integer, boolean, real). Ensuite on liera chaque nouveau Defn correspondant à une variable à la chaîne de caractères qui représente son nom. Lorsqu'on rencontrera des variables on cherchera d'abord si son nom est déjà lié à un Defn. Si c'est le cas (la variable a déjà été déclarée) on récupère ce Defn ainsi que les informations qu'il contient à propos de la variable en question. Sinon on créé un nouveau Defn et on le lie au nom de la variable rencontrée. Ces deux opérations sont mutuellement exclusives en fonction de l'endroit de l'arbre où on se trouve : - dans IDENT_DECL on retourne une erreur si la variable est déjà présente dans l'environnement : on ne peut pas redéclarer une variable existante - dans IDENT_UTIL on retourne une erreur si la variable n'est pas présente dans l'environnement : on ne peut pas utiliser une variable non déclarée

Ainsi on peut modéliser le contexte de façon à respecter les conditions du langage.

– ReglesTypage

Une fois qu'on a décoré l'arbre de façon exhaustive, on va vérifier que les opérations qu'on effectue sont valides au sens du typage. On utilise 3 fonctions pour vérifier cela : - affectCompatible qui vérifie que l'affectation d'une grandeur à une variable est valide - unaireCompatible qui vérifie que les opérations unaires (PlusUnaire, MoinsUnaire, Non) sont valides - binaireCompatible qui vérifie que les opérations binaires (Plus, Mult, Et, Ou, ...) sont valides

Ces fonctions retournent 3 structures : ResultatAffectCompatible, ResultatUnaireCompatible et ResultatBinaireCompatible. Ces trois structures ont des arrtibuts en commun : - Ok : permet de savoir si l'opération testée est valide ou non - Conv1 : vrai si l'opération requiert le rajout d'un noeud Conversion sur son fils1 - Conv2 : vrai si l'opération requiert le rajout d'un noeud Conversion sur son fils2

On va tester ensuite ces champs pour déterminer les opérations à effectuer. Si le champ Ok est false, on retourne une erreur. Si les champs Conv1 ou Conv2 sont vrais on rajoute un noeud Conversion à l'endroit correspondant.

– Noeud Conversion

Pour rajouter le noeud Conversion, on détache dans un premier temps le fils correspondant du noeud courant de l'arbre. Ensuite on créé le noeud Conversion avec Arbre.creation1 et on lui attache le fils qu'on a détaché au préalable. Ainsi on a inséré un noeud Conversion entre le noeud courant et le fils qui le requiert.

La méthodologie de test

L'objectif principale de cette base de test est de tester notre compilateur sur des régles sémantiques décrites dans le fichier context.txt.

Pour ce faire, nous avons respecter certaines conditions afin de couvrir la majeur partie de règles.

— Bien décomposer les problèmes en écrivant des méthodes COURTES.
— Factoriser les éléments communs (Eviter les tests testant les mêmes régles sémantiques)
— Documenter chaque test à l'aide d'un commentaire avant le code testé

On écrira des programmes JCas valides et invalides sémantiquement.

–> Dans le cas valide :
Le programme s'éxecute correctement, on vérifie que l'arbre est correctement décoré.

–> Dans le cas invalide :
- On vérifie que le message d'erreur est pertinent.

Une fois le fichier *context.txt* assimilé, nous avons réalisé une liste de tests exhaustive suivant chacune des régles définies.

Pour chaques régles, nous avons écrits des tests valides sémantiquement, et des tests invalides afin de couvrir le plus de cas possibles.