Compte-Rendu de Projet Compilation

HILARICUS Cyrille-Laure
LAFFORGUE Theo
GORIATCHEFF Bjorn
RICHARD Julie
Année 2016-2017

Responsable de cours :
Mr VOISIN

Sommaire

Stratégie d'implémentation

Analyse lexicale et syntaxique

— Analyseur lexical :
Nous avons un analyseur lexical complet nous permettant de traiter les commentaires et les messages.

— Grammaire :
Nous avons choisit une grammaire extrêmement détaillée de façon a pouvoir construire notre arbre syntaxique plus facilement.
Par soucis de simplicité nous ne traitons pas les *this* et les *super* au niveau syntaxique mais au niveau des vérifications contextuelles.
Il en va de même pour l'attribution des types effectifs des méthodes et variable qui se fera également au niveau contextuel (on récupère que des string depuis le code en lui même).

— L'arbre Syntaxique Abstrait :
Afin de pouvoir avoir un AST performant, nous avons décidé d'implémenter deux types de fonctions pour les AST :

  — Les fonctions de constructions de l'arbre, permettant de construire notre AST.
  — Les fonctions de gestion de l'arbre , permettant d'évaluer notre grammaire à différents niveaux.

Les structures C

Basés sur les différents exemples que nous avions à notre disposition nous avons donc décidé de découper notre "vue" du langage en 4 grande structures dérivées de l'union YYSTYPE :

— Structure de configuration d'une variable
— Structure de configuration d'une méthode
— Structure de configuration d'une classe
— Structure de configuration de l'environnement (structures chaînée permettant de gérer la portée).

Vérifications contextuelles/Environnement

Prise en compte de l'heritage,masquage : Pour cela nous avons décidé de créer une fonction qui verifie grâce au nom d'une variable/methode sa présence ou pas dans une classe fille. Cette fonction qui renvoie un booléen nous permet par la suite en cas de doublons (lorsqu'il s'agit d'heritage) de fusionner des listes correctes.

Génération de code

Nous avons décidé de débuter par une analyse des différents cas que nous aurons à traiter en génération de code.
Nous avons d'abord crée une fonction qui permettrait de créer les étiquettes nécessaires à la génération de code de conditions (dans le cas de ce langage pour la condition IF THEN ELSE).
Par la suite nous avons équipé toutes nos structures d'un champs supplémentaire de type char* nous permettant de stocker le code correspondant à ce type de structure.

Répartition des tâches

Nous avons réparti les tâches en tenant compte du niveau de complexité des tâches à réaliser mais aussi des différentes contraintes de tout en chacun. La répartition fut la suivante :

- Bjorn GORIATCHEFF :
  - Analyseur syntaxique : réalisation de la grammaire (première phase), adaptation des structures C au implémentées au préalable.
  - Environnement : Création de l'environnement.

- Cyrille-Laure HILARICUS :
  - Analyseur Lexical : réalisation complète de l'analyseur lexical et correctifs.
  - Vérification contextuelles : vérifications liées à l'existence des classes/méthodes
  - Environnement : Création de l'environnement, Mise en place des fonction de verifications liées à l'environnement et de la structure C liée en l'env.
  - Code Intermédiaire : Ebauche de code intermédiaire.

- Théo LAFFORGUE
  - Analyseur syntaxique : réalisation de la grammaire (phase complète) et adaptation de celle-ci en fonction de l'évolution de notre projet.
  - Réalisation des structures C liée à la réalisation de l'AST
  - Environnement : Mise en place de l'environnement.
  - Verifications contextuelles : cycle d'heritage,typage,encapsualtion,override,super.

- Julie RICHARD
  - Fonction print pour l'AST.
  - Début structures C

Bilan

Etat d'avancement

Nous nous avons donc clairement un interpréteur et non un compilateur. Celui-ci répond à l'ensemble des critères niveau syntaxique.
Nous avons pu traiter l'ensemble des vérifications contextuelles vérifiant :

- L'existence des méthodes, classes, blocs.
- Les cycles d'héritage.
- les Super.
- Les override.

Nous avons pu créer un environnement et l'enrichir au fur et à mesure que l'on descend dans l'arbre du programme. Cependant les vérifications de portée et de typage (vérification du type en tenant compte de l'environnement) restent *incomplètes*, ce qui par dépendance n'à par permis de continuer la génération de code.

Difficultés rencontrées

Nous avons rencontré de nombreuses difficultés lors de la gestion de l'environnement et donc de la portée, en effet nous avons eu du mal à formaliser en fonction de notre grammaire la prise en compte de l'héritage et donc la nécessité de ne pas prendre en compte les éventuels doublons liés à celui-ci dans l'environnement de nos classes.
D'un point du vu organisationnel, nous avons eu du mal à pouvoir avoir un retour de toute l'équipe sur le projet (absences régulières de certains membres, impossibilité de travailler pour certains car non munis d'ordinateur).

Malgré ces quelques problèmes d'ordre techniques/humains et de compréhension nous avons néanmoins eu une belle progression car chacun a pu s'enrichir des divers TP/TD réalisés le long du projet afin de nous familiariser avec les outils et les méthodes de travail.

Perspectives

Ayants assimilés la problématique de la gestion de code, nous aurions pu imaginé continuer ce projet afin de le mener à terme sans de grosses difficultés. En effet, une fois l'environnement totalement maîtrisé, nous pouvons donc créer les différentes fonctions de génération de code au différents niveaux de nombre programme (Génération de code pour lex expressions, Génération de code pour les instructions, Génération de code pour les blocs, Génération de code pour les méthodes ...). La génération de code dans ce cas peut s'assimiler a différents niveaux d'abstractions qu'il faut s'avoir interpréter.