**Rapport de notre compilateur MiniC Semy BENADY Pierre CONSTANTIN**

Nous avons réalisé ce projet en binôme et nous avons globalement réussi à créer un parseur qui fonctionne et à générer la plupart des fichiers correctement de notre point de vue. Nous reparlerons des erreurs rencontrées vers la fin du rapport.

**Structure générale du fichier miniC.y**

Nous avons décidé de structurer ce fichier comme suit :

* La déclaration de nos structures, fonctions et des variables globales.
* La déclaration de la grammaire et des tokens / types.
* Les règles de productions (syntaxe + code C).
* Lancement du parsing et génération du dot.

**Déclaration de nos structures, fonctions, variables globales**

Nous travaillons avec quatre structures assez simples : un tableau d’informations sur les déclarateurs, une structure d’informations sur un symbole, une structure portée qui répertorie une liste chaînée de portées et enfin une structure représentant un nœud.

Ces quatre structures ont été choisis pour nous permettre de facilement analyser la portée d’une déclaration afin de l’associer à un symbole qui lui même si l’expression est conforme sera généré en nœud afin d’afficher notre arbre en fichier dot.

Le nœud est classique : il permet de généré un nœud d’un graphic en dot à partir des informations sur sa couleur, sa forme, son nom... Il nous permet une génération centrée sur cet arbre.

La structure symbole elle nous permet de stocker dans une table de symbole (que l’on ne va pas hacher) chaque variable avec le nombre de déclarations (gestion d’erreur), son type, son nom, savoir s’il s’agit d’un tableau, sa dimension...

Passons aux choix sur les fonctions.

Nous avons tout d’abord choisis de gérer les types de retour des fonctions imbriquées grâce à une pile stack\_f\_type. De même pour la portée, notre reach\_stack est choisie. Nous choisissons dynamiquement de notre portée grâce à enter\_bloc et end\_bloc. Nous vérifions et réinitialise à chaque entrée si la fonction/variable est déjà déclarée avec is\_declared.

**Déclaration de la grammaire**

Ce n’est pas très original car nous avons choisi de garder celle que vous nous aviez donné!

**Règles de production syntaxique**

Nous allons voir les principales règles de production :

programme → liste\_declarations liste\_fonctions { print\_func($2); }

--------------------------------------    
| <liste\_fonctions>.code |

|print\_fonction(liste\_fonctions)|   
 --------------------------------------

Simplement : on affiche les fonctions qui sont dans liste\_fonctions!

declaration → type liste\_declarateurs ';' { $1, $2.name, $2.dim;}

liste\_declarateurs → .... {add\_symbol($3.name, "int", 0, 0);}

--------------------------------------    
| <liste\_declarateurs>.code | → on ajoute chaque déclarateurs dans la liste

| declaration.infos=$1, $2.infos |

| liste\_declarateurs =ajoutersymbols($)| → on ajoute chaque symbole dans    
 ------------------------------------------------------- notre table de symboles en précision s’il est un int ou un tab.

fonction → type IDENTIFICATEUR '(' liste\_parms ')' { add\_symbol\_f($2, $1, $4.nb\_params); enter\_f($1); }

mon\_bloc { verif\_return = 0;$$ = make\_node\_f($2, $1, $4, $7); leave\_f(); if (strcmp($1, "void") != 0 && verif\_return == 0) {printf("Erreur: fonction %s sans return\n", $2); } };

--------------------------------------    
| <liste\_parms>.code | → on enregistre la fonction et mémorise

| | le type de retour dans une pile.

| <mon\_bloc>.code | → on crée notre noeud en rentrant dans

| verif return && makenode | “mon\_bloc”, on fait une vérif ------------------------------------------------------- sémantique et on sort du bloc si on    
 a bien un retour.

On se retrouve avec cette construction :

fonction

│

├── type IDENT (...) ← déclaration

│ └── add\_symbol\_f(...) ← ajout table symboles

│ └── enter\_f(...) ← empile type de retour

│

├── mon\_bloc ← bloc de la fonction

│ ├── start\_bloc() ← nouvelle portée

│ ├── liste\_instructions ← contenu

│ ├── make\_node\_bloc(...) ← nœud AST du bloc

│ └── end\_bloc() ← sortie de portée

│

├── make\_node\_f(...) ← fonction complète

└── leave\_f() + verif return ← sortie sémantique

Ainsi le code se construit entière comme montré ci-dessus: on génère des arbres fils et on descend petit dans l’arbre au fur et à mesure de découvrir des fonctions. 

**Génération du dot**

Il s’agit de la partie où l’on s’est inspiré de ces deux projets :

<https://github.com/Alsroth/Compilateur_miniC_DOT>

<https://github.com/MonsieurCo/Projet-Compilation-Compilateur-Mini-C>

De plus nous avons communiqué avec ces élèves ainsi que d’anciens des années précédentes via le discord universitaire (merci notamment à Alecu Luca et Gicchero Baptiste). Nous avons pu comprendre et faire les choix des fonctions à la fin de notre miniC.y grâce à cela. Cela a été une difficulté pour nous car nous n’avions pas bien compris le principe des fichiers dot malgré leur simplicité.

On a donc définit notre topologie comme un fichier dot classique a [label="Foo" color=red]; ce qu’on a essayé de réécrire à chaque fois. On met une forme différente s’il s’agit d’un bloc, d’une fonction... Et on connecte chaque pères et fils ensembles.

**Erreurs dans le miniC.y**

Il y a tout de même des erreurs dans notre code :

La gestion du break et du switch ne marchent pas si nous ne retrouvons pas de return. Des erreurs par manque de temps aussi comme des shift reduce pour la gestion des tableaux notamment.

Certains tests ne génèrent donc pas de code : break, switch, tab et tab multi.