ESCOBAR Quentin

GIRARD Dorian

Rapport projet de compilation :

MiniC

1. Introduction :

L’objectif de notre projet est de développer un compilateur pour un sous langage du C nommé MiniC. Le compilateur doit générer un graphe à partir d’un code MiniC. Pour cela nous utilisons le langage DOT pour créer une représentation intermédiaire afin de la transformer par la suite en graphe sous format pdf.

Pour réaliser ce projet nous avons dû apprendre les outils tels que LEX et YACC pour l’analyse syntaxique et sémantique et DOT pour la génération de graphes.

Nous avons séparé la création de notre compilateur en plusieurs étapes.

Tout d’abord, la partie d’analyse lexical en utilisant un fichier LEX ou l’on va définir les mots clés réservés utilisés par notre langage ainsi que les actions à effectuer lorsque LEX rencontrera ce mot.

Dans un second temps nous avons utilisé YACC pour réaliser l’analyse syntaxique qui va vérifier que le code en entrée soit bien conforme à la grammaire de notre langage.

Enfin, de nouveau avec l’aide de YACC nous avons réalisé la partie d’analyse sémantique tout en construisant en mémoire la représentation intermédiaire en DOT.

1. Organisation du travail :

Nous avons commencé le projet en travaillant ensemble sur le fichier LEX jusqu’à l’analyse syntaxique.

Après avoir créer nos premières structures pour faire remonter les informations nous avons séparé les tâches de travail en deux groupes :

1. Dorian s'est consacré à la génération des arbres abstraits en mémoire et à la génération du code intermédiaire en DOT.

1. Quentin s'est concentré sur l’analyse sémantique, afin que le code généré corresponde aux exigences du langage. Comme la vérification de types lors d’une déclaration, la portée des variables, etc.

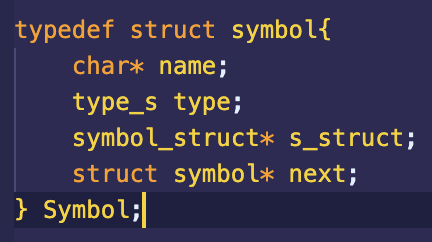
Cependant nous nous sommes aidés lorsqu’il y avait des soucis que l’on arrivait pas a résoudre et avons continué de communiquer tout au long du projet surtout sur la partie sémantique.

Concernant l’aide extérieur, nous avons utilisé la documentation de LEX et YACC ainsi que des forums pour la création d’un fichier MakeFile, ou lorsque nous étions bloqués sur une erreur ou que nous n’arrivions pas à comprendre quelque chose.

1. Structure de données utilisées

En ce qui concerne la partie de l’analyse sémantique, nous avons décidé d’utiliser une table de symboles pour pouvoir stocker les informations nécessaires sur les déclarations de variables, fonctions et de tableaux.

Pour l'implémenter nous avons utilisé une structure de liste chaînée.



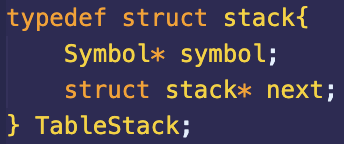
Chaque élément de la liste chaînée est un symbol qui possède un nom ou un identificateur, un type parmi la liste suivante : fonction, variable ou tableau



et une structure qui correspond au type du symbole. Par exemple, si le symbole est une fonction alors la structure va contenir le type de la fonction, et son nombre d’arguments.

Et enfin la structure du symbole contient aussi le prochain élément de la liste chaînée.

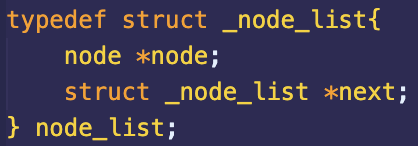
Aussi, nous avons dû ajouter une autre structure pour pouvoir gérer les imbrications de bloc d’instructions. Cette structure est une pile et chaque élément de cette dernière est une table de symboles comme vu précédemment.



Un élément de la pile contient le premier symbole de la liste chaînée et le prochain élément de la pile.

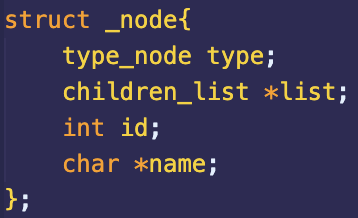
Et enfin nous avons ajouté une variable globale *“top”* au programme correspondant au sommet de la pile. Cette variable nous permet d'accéder au sommet de la pile depuis n’importe quelle endroit du code pour pouvoir ajouter ou supprimer un symbole.

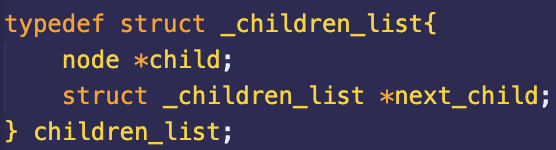
Ensuite pour la partie représentation intermédiaire en mémoire, nous avons implémenter une liste chaînée d'arbres abstraits.



Chaque élément de la liste possède le premier nœud de l’arbre abstrait de la fonction et le prochain élément de la liste.

Un nœud possède un type qui permet ensuite de définir sa couleur et sa forme dans le schéma du fichier DOT. Un nœud a aussi un id, un nom et une liste de nœuds fils. Ces derniers seront reliés au nœud parent dans le schéma.





Cette structure nous permet de créer et parcourir des arbres abstraits de manière efficace pour l’affichage du graphe dans le fichier.

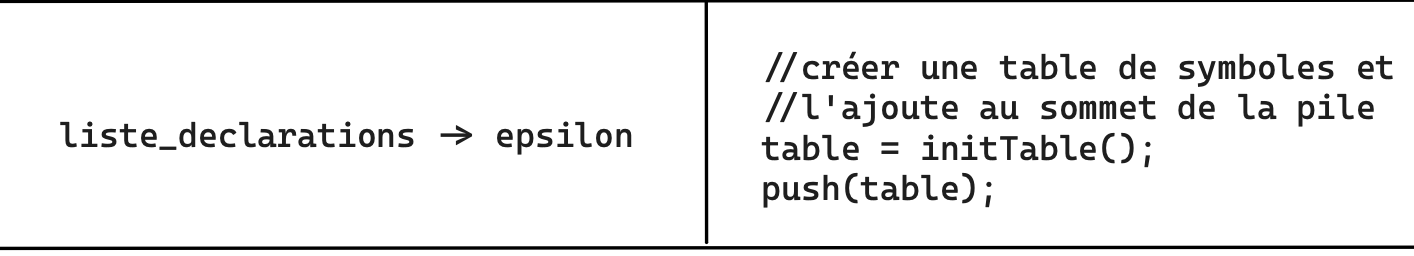
1. Schémas de traduction de code

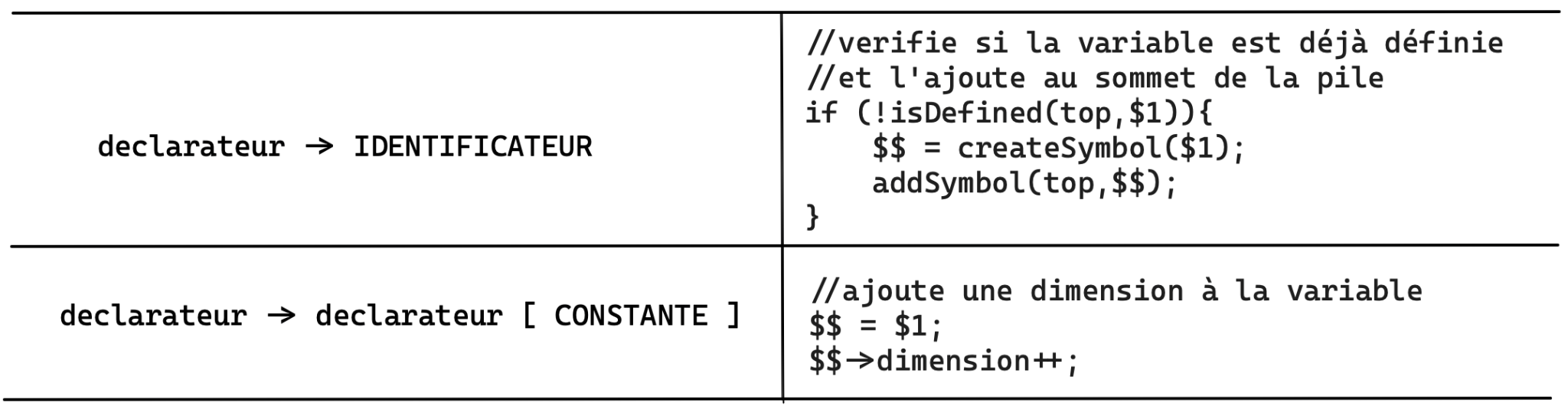
Tout d'abord dans le fichier LEX nous avons eu des soucis pour correctement gérer les token identificateur et constante mais aussi pour gérer les commentaires. Pour gérer les commentaires nous avons décidé d’utiliser un regex pour les identifier et les ignorer. Pour gérer les token identificateur et constante il a fallu modifier l’union de Yacc et aussi ajouter la fonction strdup() pour les chaînes de caractères dans le Lex pour que le compilateur ne plante pas.

Nous avons créé des fonctions pour interagir avec la pile et ses structures, ce qui nous a permis de gérer les différentes opérations et vérifications nécessaires pour respecter les règles du langage MiniC. Cependant, nous avons eu des soucis pour parcourir la pile au début et avons dû créer des fonctions de debug pour afficher son contenu récursivement.

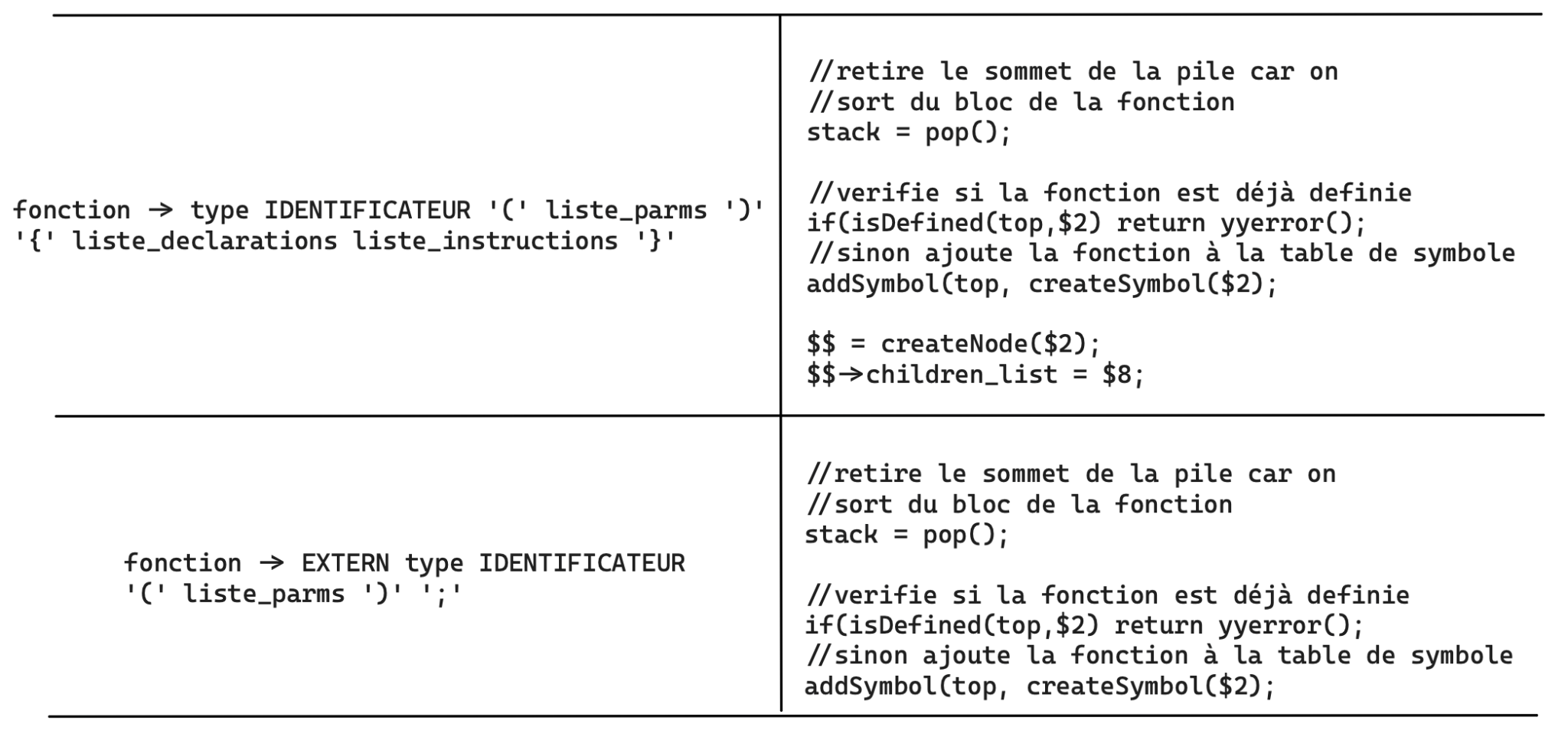
Cela nous a permis de mieux comprendre nos erreurs et de corriger les problèmes de parcours.

Notre compilateur est capable de gérer les déclarations de variables, de tableaux et de fonctions en les ajoutant dans la table de symboles au sommet et en vérifiant si ces variables sont déjà déclarées. Voici les schémas simplifiés de traduction de code :





le *‘déclarateur’* est du type *Symbol.*



le ‘fonction’ est du type *Node*.

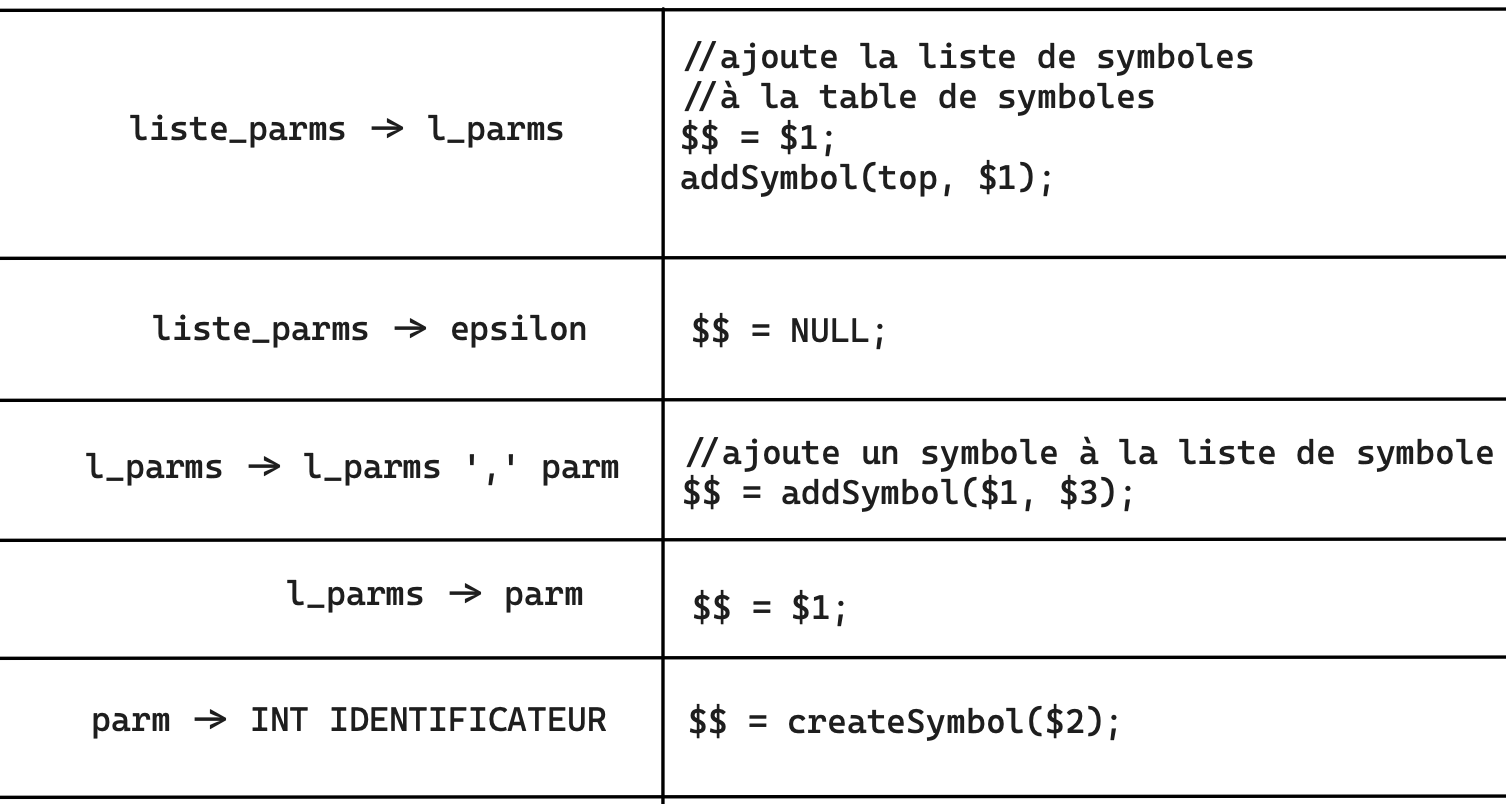
Nous avons rencontré un problème avec cette implémentation car le compilateur affichait une erreur si une fonction était récursive. Ceci était dû au fait que la fonction n’était pas encore déclarée au moment de la vérification. On a alors ajouté une routine sémantique entre *‘liste\_declarations’* et ‘*liste\_instructions’* qui permet d’ajouter la fonction dans table des symboles au sommet de la pile. Ainsi on peut maintenant reconnaître des fonctions récursives.

Aussi nous avons eu un autre problème concernant la liste de paramètre d’une fonction. Le problème, qui était lié à la grammaire du YACC, ne permettait pas à l’analyseur syntaxique d’accepter une liste vide de paramètres. Pour résoudre ce soucis nous avons dû modifier la grammaire en utilisant ce qui était défini dans le fichier Grammaire-miniC.html en ajoutant trois nouvelles règles :

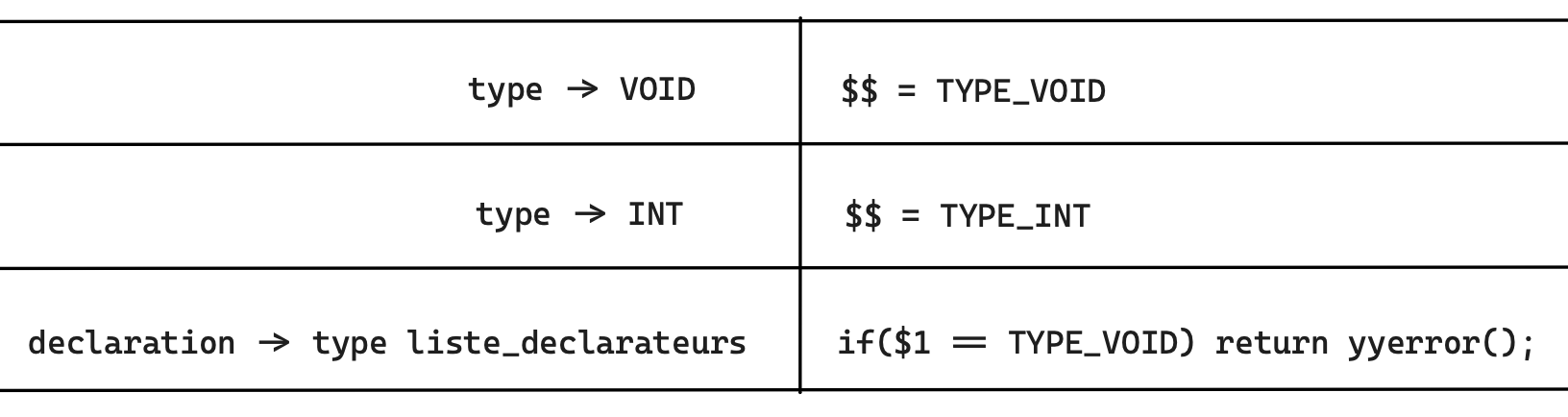
liste\_parms. -> l\_parms | 𝝴

l\_parms -> l\_params ‘ , ‘ parm | parm

parm -> INT IDENTIFICATEUR



Notre programme vérifie aussi le type des déclarations de variables. En effet, les variables de notre langage ne peuvent pas être du type VOID.

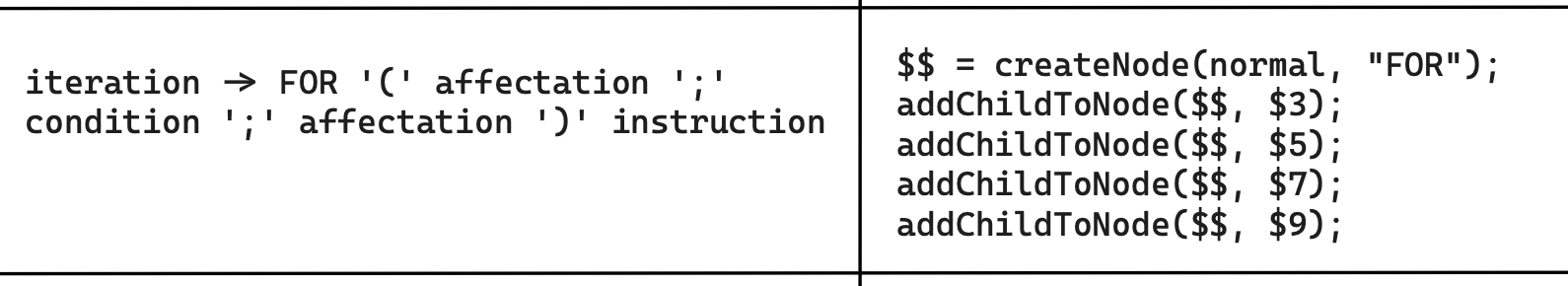


le ‘type’ est une énumération nommée *type\_t.*

Notre compilateur est aussi capable de générer un code intermédiaire en DOT. Pour cela nous construisons des arbres abstraits tout au long de l’analyse syntaxique. Dans notre programme nous utilisons principalement deux fonctions pour implémenter cette structure.

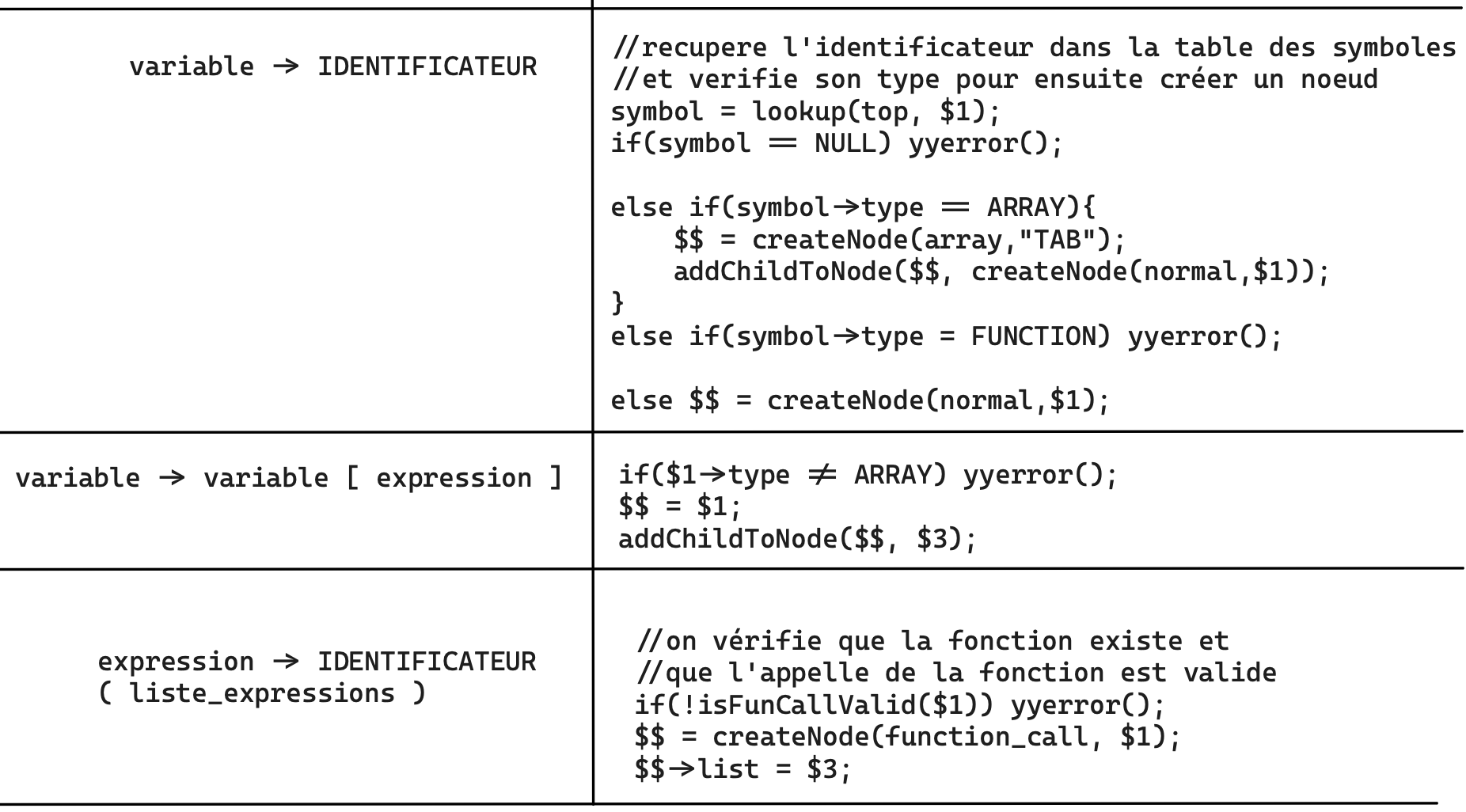
* createNode(type, nom);   
  Cette fonction crée et renvoie un nœud.
* addChildToNode(node1, node2);  
  Cette fonction ajoute un noeud à un autre à la fin de la liste de noeuds

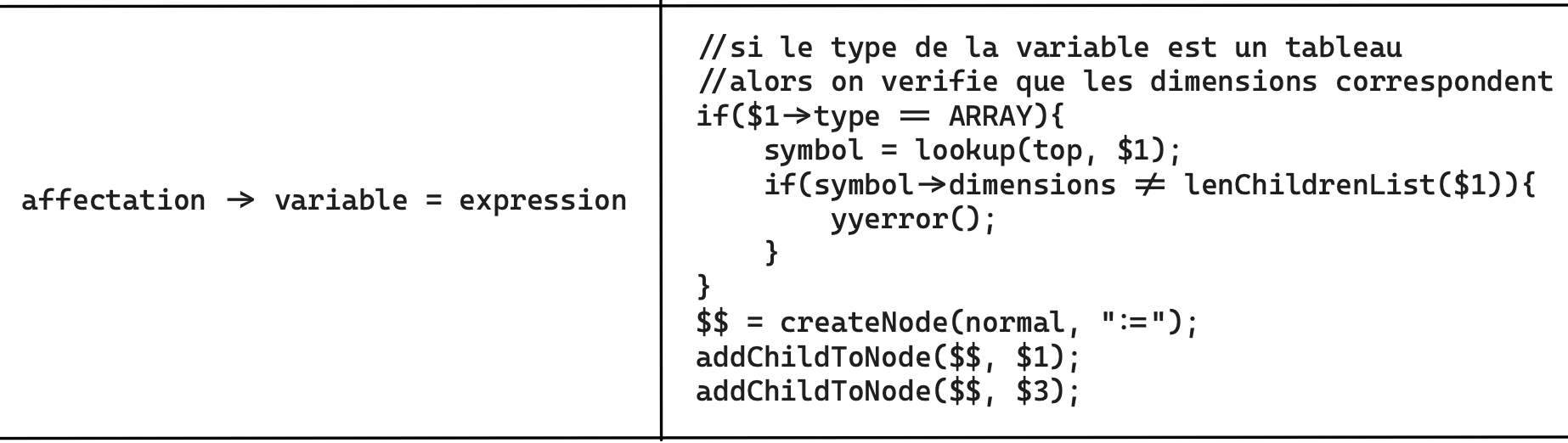
Par exemple le programme permet gérer les boucles for avec le code ci-dessous :



Cependant la traduction de la structure *‘switch’* nous a posé des difficultés car les *‘cases’* et les instructions qui les suivaient étaient séparées dans le graphe en sortie. Pour cela nous avons dû créer une fonction qui déplace les nœuds des instructions dans les listes de fils des nœuds correspondant aux *‘cases’*. Cette fonction vérifie aussi la sémantique du switch. En effet, dans notre langage, on ne peut pas avoir plusieurs instructions *‘default’* dans un même switch. Aussi on ne peut pas avoir plusieurs instructions *‘cases’* avec la même valeur.

De plus, notre compilateur peut vérifier la validité d’une expression. En effet, notre programme vérifie si une variable n’est pas appelée en tant que tableau ou fonction et inversement. Aussi, il vérifie qu’un tableau est bien appelé avec le bon nombre de dimensions et pour les fonctions qu’elles sont bien appelées avec le bon nombre de paramètres et que leurs types de retour est bien égal à *‘int’* dans une expression.





De plus, nous avons fait face à un autre problème lors de l'implémentation des expressions dans notre programme. Le problème était que les priorités des opérateurs n’étaient pas respectées lors de l’analyse syntaxique. Pour résoudre ce problème nous avons modifié la grammaire de sorte à ce que les règles de priorité de YACC puissent être appliquées.

La grammaire de base ressemblait à ceci :

* *expression binary\_op expression*
* *binary\_op -> PLUS, …*

Nous l’avons alors remplacer par ceci :

* *expression PLUS expression*
* pareil pour tous les opérateurs binaires…

Enfin, le programme peut gérer les messages d’erreur et stopper le programme lorsqu’il en rencontre une grâce à l'implémentation de la fonction yyerror. Cette dernière nous permet d’analyser la syntaxe d’un fichier correctement et de nous indiquer la ligne où une erreur est survenue dans le cas contraire.

1. Fonctionnalités du compilateur
2. Partie lexicale

* Analyse du code en entrée
* Définition et retour des tokens du langage
* Neutralisation des commentaires /\* \*/
* Les constantes sont entières et exprimées en base 10.
* Les variables doivent commencer par une lettre.

1. Partie syntaxique :

* Analyse du code en fonction de la grammaire
* Toute variable doit être déclarée avant utilisation.
* Les variables ne peuvent pas porter le nom d’un mot clé réservé : extern, int, void, for, while, if, then,else, switch, case, default, break.
* Une fonction possède un identificateur qui est son nom, un type qui est soit int soit void. Les paramètres des fonctions sont des entiers. Les fonctions peuvent également être déclarées extern. Il ne peut pas y avoir de déclaration imbriquée de fonction.
* Les instructions considérées sont l’affectation, les instructions de contrôle et le return. Ce dernier peut retourner une valeur ou non.
* Les instructions de contrôle ont la même sémantique qu’en C, nous avons les instructions : if, else, switch, default, case, for, while, break.
* Les variables globales sont déclarées en début de programme, tandis que les variables locales sont déclarées au début d'un bloc. Elles sont de type entier ou tableau d’entiers à un nombre quelconque de dimensions.
* Les conditions sont toujours entourées de parenthèses. Les opérateurs de comparaison disponibles sont >, <, <=, >=, ==, !=. Les opérateurs booléens disponibles sont !, && et ||.
* Les expressions sont composées de valeurs numériques, de variables et d'opérateurs binaires tels que +, -, \*, /, <<, >>, &, |. Les appels de fonctions peuvent également faire partie des expressions arithmétiques.

1. Partie sémantique

* Les variables et les fonctions ne peuvent pas être définie plusieurs fois dans un même bloc
* Lors de l’appel à un tableau, on vérifie que le nombre de ses dimensions soit correct.
* Une instruction switch ne peut pas contenir plusieurs default et plusieurs cases avec la même valeur
* Lors de l’appel à une fonction, on vérifie que le nombre de paramètres est correcte
* Une variable ne peut pas avoir un type void
* Une fonction dont le type de retour est void ne peut pas être appelé dans une expression
* Une variable, un tableau et une fonction ne peuvent pas être appelée si ils ne sont pas définies
* Un identificateur déclaré en tant que variable (resp. fonction, tableau) ne peut pas être appelé en tant que tableau ou fonction (resp. variable)

1. Partie génération de code intermédiaire

Notre compilateur produit un graphe DOT correct pour les instructions suivantes :

* La déclaration d’une fonction
* Les séquences d’instructions
* L’instruction d’affectation
* Les expressions arithmétiques ou logiques
* Les références à un élément de tableau
* Les expressions conditionnelles
* L’instruction de retour d’une fonction
* L’instruction conditionnelle if-then et if-then-else
* L’instruction switch
* L’instruction de boucle for
* L’instruction de boucle while
* L’instruction break
* L’appel d’une fonction

1. Conclusion

La réalisation de ce projet nous a permis de mettre en pratique tout ce que nous avons vu pendant ce semestre. Nous avons pu mettre en pratique les différents concepts qu’on a vu en cours et en TD.

Nous avons dû apprendre et comprendre LEX et YACC. Aussi, il nous a fallu apprendre à implémenter des structures efficaces pour pouvoir gérer les informations de notre programme et pour générer un code intermédiaire.

Cela a été un projet demandant en recherche, avec beaucoup d’essais et d'échecs pour faire fonctionner correctement l’ensemble.

Cet exercice était tout de même agréable car nous avons pu créer le front-end d’un compilateur, c'est-à-dire de l’analyse du programme jusqu’à la génération d’un code intermédiaire et ainsi cela nous a permis de comprendre le fonctionnement d’un langage de programmation et de son exécution. Ce projet nous a aussi permis d’améliorer notre manière de travailler en équipe et notre organisation.

Enfin, avec du recul, la création d’un compilateur est une tâche longue et difficile, ce qui en fait un projet très important pour la suite de nos études et pour notre carrière professionnelle.