Rendu de projet de compilation

Martin D'Escrienne Yann et Tognetti Yohann

Introduction et résumé du projet :

Au cours du semestre nous avons appris de nombreuses notions sur la compilation et les compilateurs qui nous ont permis de réaliser à bien ce projet.

Nous avons néanmoins pris quelques libertés, en expérimentant quelques fois nous même avec des structures qui n'ont pas été vues en cours.

Les test fournis passent (au niveau lexicographique, syntaxique, sémantique et traduction du backend). Sauf quelques exceptions (voir plus bas).

L'analyse syntaxique se base sur la grammaire donnée avec quelques modifications pour répondre aux attentes de la consigne.

L'analyse sémantique gère de nombreux cas, bloque de nombreuses erreurs et génère des warnings. Nous avons essayé de nous rapprocher le plus possible du compilateur *gcc* et de ses erreurs/warnings (les erreurs ne sont pas forcément les mêmes mots pour mots).

Notez que notre analyse sémantique ne permet de prédéfinitions de fonctions.

La traduction du backend est faite comme demandé se basant sur l'exemple donné et les réponses sur le forum. L'utilisation des variables temporaire n'est néanmoins pas optimale.

Les fonctions qui furent utile pour les actions sémantiques se trouvent dans le fichier *code.c*

Tests qui ne passent pas :

- Pointeur.c : En effet, la fonction *malloc* n'est pas déclarée (ni en *extern*), notre compilateur génère donc une erreur. De plus le « (*i)++ » ne sera pas accepter par la grammaire.

```
######################################################## pointeur.c
EXTERN INT IDENTIFIER:printd (  INT IDENTIFIER:i  )  ;

INT IDENTIFIER:main (  )  {
   INT
add stage
  * IDENTIFIER:i ;
   INT  * IDENTIFIER:j ;


 malloc : not defined
SEMANTIC ERROR
   IDENTIFIER:i = IDENTIFIER:malloc
```

```
1 extern int printd( int i );
2
3 int main() {
4    int *i;
5    int *j;
6
7    i=malloc(sizeof(int));
8    j=malloc(sizeof(int));
9
.0   *i=4;
.1
.2   printd(*i);
.3
.4   *j=6;
.5   printd(*j);
.6
.7   *j=*j+(*i)++;
.8   printd(*j);
.9   return 0;
!0 }
```

- Expr.c : il y'a l'opération « << » qui n'est pas demandé dans le projet, il y a donc une erreur syntaxique.

```c
extern int printd( int i );

int main() {
  int i;
  int j;
  int k;
  i = 45000;
  j = -123;
  k = 43;
  printd(((i+j)*k/100+j*k*i-j<<k)/(k-j>>2));
  return 0;
}
```

Note : si ces tests sont corrigés, ils passent alors parfaitement.

Répartition du travail :

Pour la répartition du travail, celle-ci vu équitable, le travail se faisait souvent à deux par Visio, en alternant celui qui « écrivait le code » (difficulté de travailler sur le même fichier en même temps). Le fait de n'être que deux dans le groupe a permis une bonne cohésion et un travail optimal.

Certaines parties furent faite séparément, l'un codant les fonctions dans le *code.c* l'autre les utilisant dans le *structfe.y*.

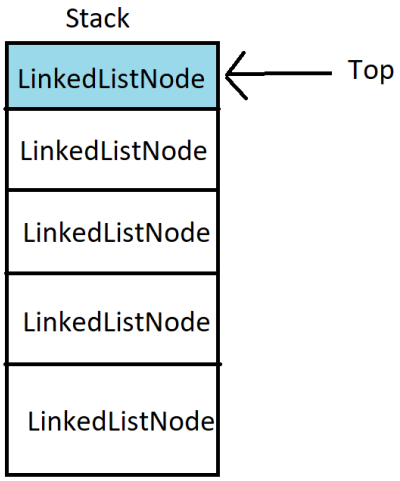
Structures de données :

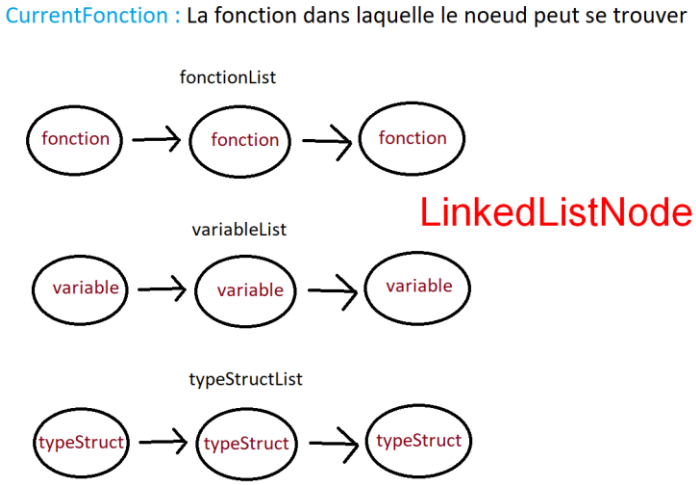
Partie frontend :

Pour la partie frontend, pour la « table des symboles » nous avons utilisé une pile qui empile des nœuds « LinkedListNode » qui retient les variables, les fonctions et structures déclarés du bloc courant.

Les variables, structures et fonctions sont des structures misent dans des listes chainées.

Cela nous sert pour l'analyse sémantique (re-declaration, appel de variables inexistantes…)





Pour les opération binaire, unaire, les comparaison… nous avons utilisé une structure Type qui nous permet de gérer les comparaison et erreurs de typages efficacement. Cette structure remonte lors des actions sémantiques jusqu'à la fin de l'expression.

```
typedef struct _Type
{
    int isUnary;
    int isPtr;
    int isExtern;
    int isFunction;
    struct _FunctionType* functionType;
    UnaryType unaryType;
    struct _TypeStruct* typeStruct;
} Type;
```

le type est un pointeur *

booléens pour savoir quel sorte de type est actuellement affecté

Type : fonctions

int ou void

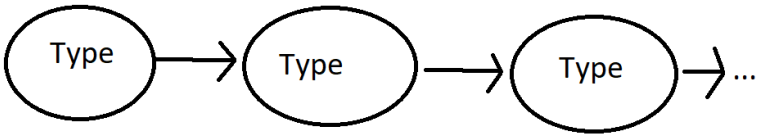
Type : Structures

A gauche, un schéma de comment est faite la structure FonctionType qui est la moins triviale.

FonctionType :

- returnType : le type du retour de la fonction

- parameterType : les types des parametres de la fonction



Pour l'envoie des données entre actions sémantique et différents états de la grammaire lors des déclarations de fonctions, structure et variables, nous avons utilisé la structure « Transit ».

Elle permet également de récupérer le type qui remonte de l'analyse ascendante dans la variable ou la fonction.

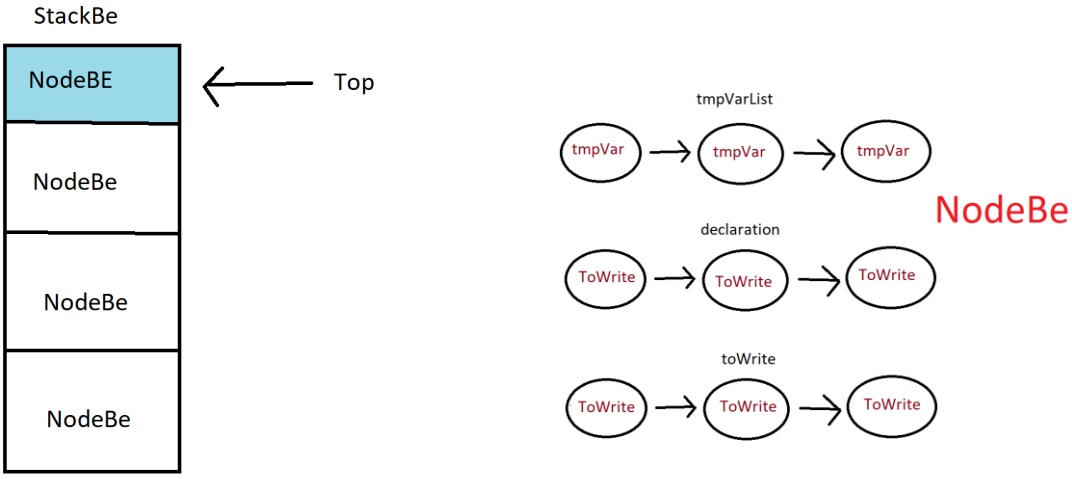
```
typedef struct _Transit
{
    Variable* variableD;
    Fonction* fonctionD;
    char* name;
    int isPtr;
} Transit;
```

Partie Backend :

Similairement à la partie frontend, la partie backend se base aussi sur une pile où les nœuds contiennent les déclaration, variables temporaires et les « contents » à écrire du bloc courant.

Un *content* est une structure de chaine de caractères à taille dynamique (réallouer).

Les *content* sont mit dans une liste chainée appelée *toWrite*, ils seront alors écrit les uns après les autres dans le fichier.



Les variables temporaires ont un type qui leur est propre (pour ne pas les utiliser pour tout et n'importe quoi).

Lorsque l'on get une variable temporaire elle n'est pas plus disponible (sauf si remise à disposition par une actions sémantique). Si aucune variable n'est disponible, une nouvelle est créée avec le type voulu.

```
typedef struct _StackBE
{
    NodeBE* top;
    Label label;
    int hasOr;
    int hasAnd;
    int hasEq;
    int hasNoEq;
    int hasSup;
    int hasSupEq;
    int hasInf;
    int hasInfEq;
} StackBE;
```

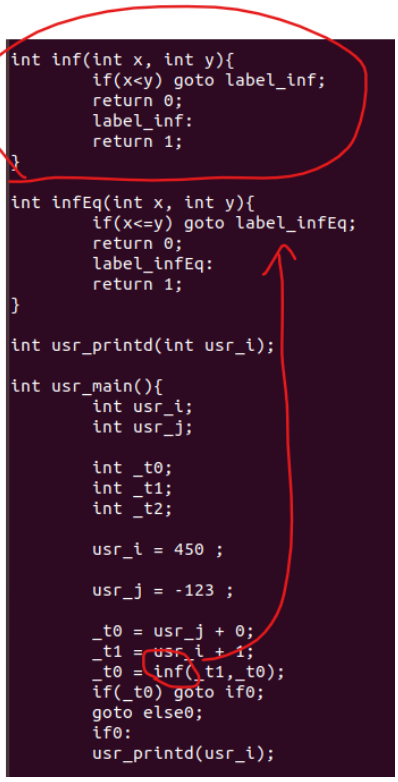
```
typedef struct _Label
{
    int numElse;
    int numIf;
    int numContinue;
    int numWhile;
    int numBody;
    int numTest;
    int numFor;

} Label;
```

La structure label, permet de générer des lables (else0,if0,while0) unique avec un numéro qui s'incrémente à chaque génération de labels.

Lors de l'appel d'operation de comparaison et de « && » et « || » une fonction qui réalise l'operation voulue est ajoutée au debut du programme backend.

Exemple du « < »:



```
int inf(int x, int y){
        if(x<y) goto label_inf;
        return 0;
        label_inf:
        return 1;
}

int infEq(int x, int y){
        if(x<=y) goto label_infEq;
        return 0;
        label_infEq:
        return 1;
}

int usr_printd(int usr_i);

int usr_main(){
        int usr_i;
        int usr_j;

        int _t0;
        int _t1;
        int _t2;

        usr_i = 450 ;

        usr_j = -123 ;

        _t0 = usr_j + 0;
        _t1 = usr_i + 1;
        _t0 = inf(_t1,_t0);
        if(_t0) goto if0;
        goto else0;
        if0:
        usr_printd(usr_i);
```

Note : Si l'opération n'est pas appelée la fonction n'est pas définie au début du fichier.

Les déclarations de variables et de variables temporaires sont toutes misent au début du bloc qu'importe leur déclaration dans le frontend.

Pour les while, for, if et if else (l'ordre des images) nous n'avons pas suivis la structure du cours qui utilisais la négation, l'optimisation des goto est donc moindre :

```
while0:
_t0 = inf(usr_i,10);
if(_t0) goto body0;
goto continue0;
body0:
{

    usr_printd(usr_i);

    usr_i = usr_i + 2 ;

}

goto while0;
continue0:
```

```
usr_i = -10 ;

goto test0;
for0:
usr_printd(usr_i);

usr_i = usr_i + 1 ;

test0:
_t0 = infEq(usr_i,10);
if(_t0) goto for0;
```

```
_t1 = noEq(usr_p,0);
if(_t1) goto if0;
goto continue0;
if0:
_t0 = *usr_p;
usr_printd(_t0);

continue0:
```

```
_t0 = usr_j + 0;
_t1 = usr_i + 1;
_t0 = inf(_t1,_t0);
if(_t0) goto if0;
goto else0;
if0:
usr_printd(usr_i);

goto continue0;
else0:
usr_printd(usr_j);

continue0:
```

Information pour la compilation et le makefile :

Toutes les instructions et informations se trouve dans le lisezmoi.txt

Conclusion personnelle :

Yann Martin D'Escrienne :

Pour ma part ce projet m'a permis de concrétiser toutes les notions que l'on a apprises lors du cours de compilation. Même si nous n'avons pas suivis toutes les méthodes du cours comme l'arbre syntaxique ou les graphes de flot de contrôle.

Cela a été très instructifs même si le C nous à rappelé qu'il est plein de surprise et qu'il ne pardonne pas les étourderies. Le début fut assez difficile, mais à présent je maitrise et comprend beaucoup mieux Lex et yacc.

La plus grande difficulté rencontrée pour moi fut la gestion des chaines de caractères en C et les allocations… Où debugger une « core dumped » d'une structure mal allouée, ou une chaine de caractère qui ne voulais pas se concaténer pouvais prendre des heures. C'est pour cela que nous avons opté pour une chaine de caractère de taille dynamique.

Yohann Tognetti :

Ce projet a été plutôt difficile au début, après avoir fait pas mal de Java cette année qui est orienté objet et qui n'a pas le problème de gérer la mémoire avec des « malloc » et des « free ». Le langage C m'a paru beaucoup plus complexe qu'au paravent car après m'être habitué à la programmation orienté objet, la méthode de programmation qui est différente, a rendu la structuration difficile car les « struct » ne peuvent pas être gérés exactement comme des objets.

Ce projet m'a donc permis de revoir le C de manière plus poussé avec une approche différente. Les différentes erreurs « core dumped » m'ont permis de comprendre plus en détail la gestion de mémoire afin de réussir à me représenter les pointeurs.

Au niveau de Lex, les td ont déjà suffi à comprendre le fonctionnement avec la détection de paterne pour les convertir en TOKEN pour yacc (ou autre).

Au niveau de Yacc, cela m'a fait comprendre pas mal de notion vue en cours avec la construction de l'arbre ascendant ou l'on a dû effectuer différentes actions au niveau de la table des symbole (notre stack de table de symbole) ou encore la traduction du code en backend qui demande de bien comprendre l'arbre généré par yacc et de rajouter des éléments a celui-ci.