UNIVERSITÉ CÔTE D'AZUR

LICENCE INFORMATIQUE

PROJET DE COMPILATION

Dylann BATISSE
Thibault JUNIN

TABLE DES MATIÈRES

Ce présent document est le rapport du projet de compilation que nous devions réaliser dans le cadre du cours de compilation en Licence 3.

Le projet était de réaliser un compilateur d'un sous langage du C, appelé miniC, qui génère un code intermédiaire DOT qui avec l'outil graphviz sera transformé en graphe et cela en deux passes. La 1ère passe étant le parsing et la détection d'erreurs du programme et la 2ème passe est la génération du fichier dot en parcourant l'arbre en mémoire.

Au cours de ce projet, nous avons réussi à réaliser le compilateur demandé et à générer le graphe, nous n'avons pas mis en évidence de problème avec celui-ci à ce jour.

ORGANISATION DU PROJET

Afin de réaliser un projet d'une telle envergure, nous avions besoin d'une bonne organisation. Pour cela, nous avons centralisé notre code à l'aide de Github. L'utilisation de Github nous a permis de travailler en continue sur plusieurs branches pour tester différente structure de données tout en gardant un historique des modifications.

Nous avons également utilisé l'outil Live Share de Visual Studio Code qui nous a permis de travailler en simultané tout en nous permettant de discuter sur le projet.

REPARTITION DU TRAVAIL

Nous avons travaillé l'entièreté de ce projet ensemble, à l'aide des outils que nous avons cité plus haut, nous partagions notre code avec Live Share, et nous étions en communication sur Discord. Durant ces sessions de travail, nous avons partagé nos idées, essayé en direct nos modifications, ce qui fait que nous avons touché à toutes les parties du projet, sans une répartition ferme et défini des tâches. Cependant, Dylann s'est plus particulièrement affairé de la réalisation du Yacc, quant à Thibault, il s'est occupé de la génération du fichier DOT.

AIDES UTILISEES

Ce projet de compilation étant compliqué mais pas difficile pour autant, nous avons mis à profit un certain nombre de ressources trouvées sur internet, notamment sur le site d'IBM[1], StackOverflow, Github[2] et le Dragon Book[3].

[1] https://developer.ibm.com/technologies/systems/tutorials/au-lexyacc/

[2] https://github.com/Kadle11/Mini-Python-Compiler-in-Lex-and-Yacc

[3] « Compilers, Principles, Techniques, & Tools. Second Edition », Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman.

FONCTIONNALITES DU COMPILATEUR

A ce jour, où nous écrivons le rapport, nous n'avons pas été en mesure d'identifier les parties du compilateur ne fonctionnant pas, l'analyse lexicale, syntaxique et sémantique ainsi que la génération du fichier dot se réalisent sans problème.

Toutes les erreurs lexical et syntaxiques sont gérés et reconnait bien le langage miniC, l'analyse sémantique ne détecte pas toutes les erreurs possibles mais en détecte tout de même une grande partie.

ANALYSE LEXICALE

Tout d'abord, pour la partie lex, nous avons commencé par identifier les différentes unités lexicales qui allaient nous être utiles au projet, une fois cette première identification faite, nous avons associé les unités lexicales aux token yacc. Dès lors cette étape réalisée, nous avons nettoyé le lex en supprimant toutes les unités lexicales que nous avons jugées non nécessaires et dont nous n'avions pas attribué de token.

ANALYSE SYNTAXIQUE

Une fois l'analyseur lexical fonctionnel nous nous sommes attaqués à l'analyse syntaxique. C'est-à-dire être sûr que le programme soit correctement formé. Il fallait s'assurer que le parseur soit opérationnel (analyse lexico-syntaxique) donc gérer toutes les erreurs de grammaire. Après cela nous avons commencé à introduire des routines sémantiques petit à petit pour visualiser les résultats des actions sémantiques, par exemple les valeurs de $1 ou $3 en fonction des %type que nous donnions avec la structure %union de yacc.

ANALYSE SEMANTIQUE

Le parseur étant fini, les actions sémantiques peuvent commencer à être développées. C'est une phase d'analyse où l'on doit établir une signification par rapport aux sens des éléments que l'on a détectés auparavant dans le lex. Nous appelons ça une traduction dirigée par la syntaxe et dans notre cas, notre grammaire n'implique que des attributs synthétisés aussi appelé grammaire S-attribuée. Pour chaque grammaire et chaque production nous avons des actions sémantiques associées qui, dans ce cas présent, vont nous permettre de créer l'arbre en mémoire tout au long de l'analyse de bas en haut (ascendante).

Les erreurs détectées par l'analyse sémantique sont : la portée des variables, re déclaration d'une fonction, nombres d'arguments d'une fonction, retour d'un int dans une fonction void non autorisée, accès à la dimension des tableaux, types différents dans une expression, fonction non déclarée si aucuns arguments passer en paramètre.

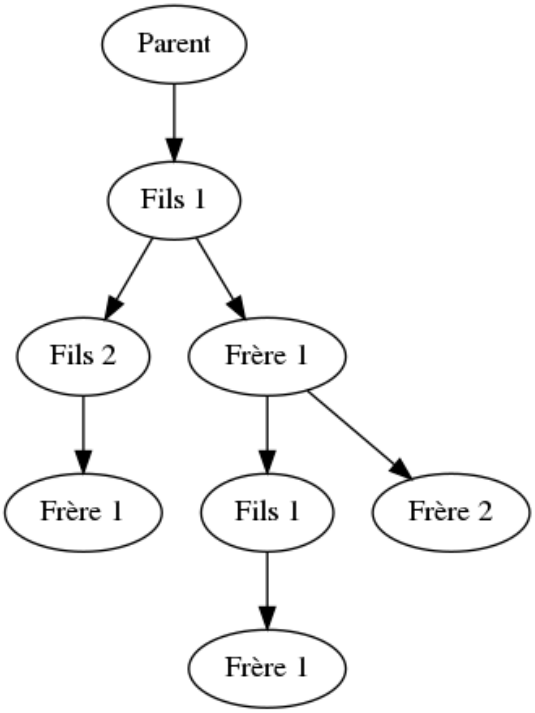
LES DIFFERENTES STRUCTURES DE DONNEES UTILISEES

Structures de donnée pour l'arbre abstrait

Le choix de la structure a été très difficile, car nous ne savions pas trop comment s'y prendre au début pour cela nous avons dû tester différentes structures de données jusqu'à avoir celle qui nous semble le plus facile à utiliser. En premier lieu nous avons décidé d'utiliser une simple table de hachage pour enregistrer tous les nœuds de l'arbre mais nous en avons vite conclu qu'il serait alors impossible d'enregistrer tout dans une seule table. Après s'être rendu compte de cette erreur nous sommes partis sur une structure de liste chaînée, dans celle-ci chaque nœud connaît ses frères et il connaît aussi ses enfants donc l'arbre est facilement accessible et parcourable.

```
typedef struct _node_t
{
    char *nom;
    type_t type;
    char *code;
    int is_func;
    int is_appel;
    struct _node_t *fils;
    struct _node_t *suivant;
} node_t;
```

1 Structure utilisée



2 Representation de la structure

Structure de donnée utilisée pour la table des symboles

Au début nous nous sommes orientés sur une table de hachage simple, similaire à celle initialement utilisé pour la structure de l'arbre, qui dans chaque case du tableau a un symbole associé avec des variables (nom, type, suivant). Au fur et à mesure nous nous sommes aperçus qu'il allait nous manquer des informations comme la portée des variables, savoir si elles sont locales ou global et le nombre de paramètres d'une fonction. Pour pallier ce manque d'informations nous avons utilisé 2 structures différentes, une pour les fonctions et une pour les variables.

```c
typedef struct _symbole_t
{
    char *nom;
    int scope;
    int tab_dimension;
    type_t type;
    struct _symbole_t *suivant;
} symbole_t;
```

2 Structure des variables.

```c
typedef struct _fonction_t
{
    type_t type;
    char *nom;
    liste_t *arguments;
    struct _symbole_t **local;
} fonction_t;
```

3 Structure des fonctions.

Grâce à ces structures, différentes erreurs sémantiques ont pu être détecté comme le montre cette capture d'écran ci-dessous.

```
-------------------------------------------------------
TEST : break.c
Semantic error : La variable a n'a pas encore été déclaré.

-------------------------------------------------------
TEST : compteur.c
Semantic error : test n'a pas encore été déclaré.
-------------------------------------------------------
TEST : cond.c
Semantic error : Mauvais nombre d'arguments lors l'appel de la fonction test

-------------------------------------------------------
TEST : div.c
 Dot file successfully generated to dot-output/div.c.dot.
 PDF successfully generated to pdf-output/div.c.pdf.
-------------------------------------------------------
TEST : expr.c
 Dot file successfully generated to dot-output/expr.c.dot.
 PDF successfully generated to pdf-output/expr.c.pdf.
-------------------------------------------------------
```

GENERATION DU DOT

Pour générer le code intermédiaire dot nous avons dû parcourir tout l'arbre qui a été au préalable enregistré dans des structures de données en mémoire lors de la première passe. Cet arbre est parcouru de façon récursive et les nodes sont générés grâce à leur propre adresse mémoire ce qui signifie donc que le nom de la node dans le fichier dot sera unique. Quant aux couleurs et formes ce sont juste des cas spécifiques qui sont vérifiés lors du parcours de l'arbre.

```
digraph mon_programme {
node_0x7fffcd67b9f0 [label="Programme"];
node_0x7fffcd678ef0 [label="main, int" shape=invtrapezium color=blue];
node_0x7fffcd67bab0 [label="BLOC"];
node_0x7fffcd679360 [label=":="];
node_0x7fffcd679240 [label="i"];
node_0x7fffcd679360 -> node_0x7fffcd679240
node_0x7fffcd6792e0 [label="45000"];
node_0x7fffcd679360 -> node_0x7fffcd6792e0
node_0x7fffcd67bab0 -> node_0x7fffcd679360
node_0x7fffcd679580 [label=":="];
node_0x7fffcd6793e0 [label="j"];
node_0x7fffcd679580 -> node_0x7fffcd6793e0
node_0x7fffcd679460 [label="-"];
node_0x7fffcd679500 [label="123"];
node_0x7fffcd679460 -> node_0x7fffcd679500
node_0x7fffcd679580 -> node_0x7fffcd679460
node_0x7fffcd67bab0 -> node_0x7fffcd679580
node_0x7fffcd679600 [label="printd" shape=septagon];
node_0x7fffcd679720 [label="+"];
node_0x7fffcd6796a0 [label="i"];
node_0x7fffcd679720 -> node_0x7fffcd6796a0
node_0x7fffcd6797c0 [label="j"];
node_0x7fffcd679720 -> node_0x7fffcd6797c0
node_0x7fffcd679600 -> node_0x7fffcd679720
node_0x7fffcd67bab0 -> node_0x7fffcd679600
node_0x7fffcd679860 [label="printd" shape=septagon];
node_0x7fffcd679980 [label="+"];
node_0x7fffcd679900 [label="45000"];
```

4 Exemple d'un extrait de génération d'un fichier dot.

DIFFICULTES RENCRONTRES

Nous avons rencontré pas mal de problèmes tout au long du projet notamment lors de l'analyse sémantique qui nous a beaucoup freiné car Lex et Yacc sont deux outils assez abstraits pour nous parce que nous n'y avons jamais touché auparavant. Au fur et à mesure nous arrivions à débloquer certains points en avançant petit à petit d'une grammaire à une autre tout en remontant la grammaire comme le fait Lex et Yacc de façon ascendante.

N'ayant pas fait énormément de C nous avons rencontré des problèmes au niveau de la gestion mémoire ce qui nous a fait perdre énormément de temps. En plus du C, Lex et Yacc ne nous aident pas vraiment non plus quant à savoir où se situent les erreurs. Pour cela nous avons même créé des fonctions de débogage qui nous donne la ligne et le caractère de l'endroit où on se situe dans le code. Cette méthode de traçage nous a permis d'avancer plus vite quand un problème était rencontré. L'outil Valgrind pour repérer les memory leak a également été utilisé.

Le fait que Lex et Yacc ait moins de documentation qu'un autre outil sur internet a rendu la chose un peu plus complexe car il fallait chercher dans des livres comme le Dragon Book qui est une source primordiale d'informations.

GRAMMAIRES MODIFIES

Modification de liste_expressions

Production de base pour **liste_expressions :**

```
liste_expressions   :
      liste_expressions ',' expression
   |
;
```

Cette production pourrait accepter liste d'expression de la forme ", expression, expression" mais dans notre langage nous ne voulons pas ça. Nous désirons que ça ressemble plutôt à "expression, expression, expression".

Modifications apportés :

```
liste_expressions   :
      create_expr_liste
   |
;
create_expr_liste :
      create_expr_liste ',' expression
   |  expression
;
```

Pour corriger cela nous avons donc créé une nouvelle production qui est create_expr_liste et va donc construire correctement l'expression souhaitée.

Modification de liste_param

Production de base pour **liste_param:**

```
liste_parms :
      liste_parms ',' parm
   |
;
```

Il s'agit du même cas que pour la liste d'expression. Nous pouvons avoir ", int a, int b" mais nous voulons plutôt quelque chose ressemblant à cela "int a, int b, int c".

```
create_liste_param :
      create_liste_param ',' parm
   |  parm
;
liste_parms :
      create_liste_param
   |
;
```

Une production intermédiaire est alors créer pour qu'on ait une liste non vide ou totalement vide si on le souhaite mais jamais avec un élément vide à l'intérieur.

Modificaition d'expression

Production de base pour **expression :**

```
expression  :
      '(' expression ')'
   |  expression binary_op expression %prec OP
   |  MOINS expression
   |  CONSTANTE
   |  variable
   |  IDENTIFICATEUR '(' liste_expressions ')'
;
```

Cette modification de la grammaire est un peu spéciale. Cette production devrait être juste, cependant en raison de l'associativité, nos expressions n'étaient pas dans l'ordre souhaiter. A ce jour, la cause du problème nous est inconnue.

```
expression :
   '(' expression ')'
   | expression PLUS expression
   | expression MOINS expression
   | expression DIV expression
   | expression MUL expression
   | expression RSHIFT expression
   | expression LSHIFT expression
   | expression BAND expression
   | expression BOR expression
   | MOINS expression %prec MOINS
   | CONSTANTE
   | variable
   | IDENTIFICATEUR '(' liste_expressions ')'
;
```

Pour régler le problème nous avons donc énuméré toutes les opérations binaires que nous disposions et la priorité du moins unaire conserver.

Modification de variable

Production de base pour **variable :**

```
variable   :
      IDENTIFICATEUR
   |  variable '[' expression ']'
;
```

Cette production seule pourrait suffire, cependant nous avons décidé de nous faciliter la vie lors de la construction des nœuds. Nous avons créé le nœud parent qui est « TAB » dans ce cas et on lui a associe son nom de variable ainsi que ses dimensions.

```
variable   :
      IDENTIFICATEUR
   |  tableau
;
tableau:
   IDENTIFICATEUR
   | tableau '[' expression ']'
;
```

Afin de facilité la génération des nœuds, nous avons recréé une production inspirée de variable, dédié à la gestion des tableaux.

Modification de « declarateur »

Production de base pour **declarateur :**

```
declarateur :
      IDENTIFICATEUR
   |  declarateur '[' CONSTANTE ']'
```

Même cas que pour la production « variable », nous avons décidé de simplifier la grammaire pour nos structures de données.

```
declarateur :
      IDENTIFICATEUR
   |  tableau_decl
;

tableau_decl:
   IDENTIFICATEUR
   | tableau_decl '[' expression ']'
;
```

Afin de facilité la génération des nœuds, nous avons recréé une production inspirée de declarateur, dédié à la gestion des tableaux.

Modification de selection

Production de base pour **selection:**

```
selection   :
      IF '(' condition ')' instruction %prec THEN
   |  IF '(' condition ')' instruction ELSE instruction
   |  SWITCH '(' expression ')' instruction
   |  CASE CONSTANTE ':' instruction
   |  DEFAULT ':' instruction
;
```

Cette production ne pourra pas correctement fonctionner pour le switch case.

```
selection   :
      IF '(' condition ')' instruction %prec THEN
   |  IF '(' condition ')' instruction ELSE instruction
   |  SWITCH '(' expression ')' instruction
   |  CASE CONSTANTE ':' liste_instructions selection
   |  DEFAULT ':' instruction
;
```

Pour régler le problème soulevé plus haut, nous avons changé instruction en **liste_instructions selection**. Du fait qu'il y a une liste d'instruction et que le prochain « case » sera délimitée par une liste d'instructions suivit d'une **selection**.

POUR CONCLURE…

Ce projet de compilation fut enrichissant en termes de connaissances notamment pour l'apprentissage du langage C mais aussi l'utilisation de Lex et Yacc qui sont des outils très puissants et qui nous ont permis la création de notre premier compilateur. C'est sûrement le projet le plus compliqué que l'on ait eu jusqu'à maintenant même si finalement le code ne paraît pas si complexe mais la compréhension du mécanisme quant à lui est beaucoup plus amphigourique.