Hakim AOUDIA
Richard ARNAOUT

2022 - 2023
Licence 3 Informatique

Projet d'Analyse syntaxique

Objectif:

Le but est de construire un analyseur lexical (flex) et un analyseur syntaxique (bison) en respectant les règles de grammaire afin de vérifier si un langage de programmation contient des erreurs de syntaxe. Le langage est dérivé du langage C qui se nomme TPC.

Afin de pouvoir implémenter ce langage dans notre programme nous avons commencé par faire en sorte que notre langage accepte certains mots de la grammaire (tout comme le langage C) des entiers **int** , des **char** et des **void**. Le but est de pouvoir créer un langage de programmation conforme à la compilation.

Première phase :

Dans cette phase du projet nous devions avant tout créer l'analyseur lexical avec le langage Flex afin de récupérer les mots appartenant à la grammaire du langage TPC. Il fallait faire attention à ce que notre analyseur lexical n'accepte que les règles qu'on lui demande. on a implémenté un compteur appelé **lineno** en l'incrémentant de un a chaque saut de ligne afin qu'il puisse renvoyer la ligne à laquelle se trouverait une erreur de syntaxe du code.
A chaque règle définie , on renvoie **yytext**, ce qui signifie que cette grammaire appartient au langage que nous sommes en train de créer. Ensuite on a créé un autre compteur appelé **colno** permettant de compter chaque caractère par ligne, ceci va nous permettre de détecter la position exacte de l'erreur générée dans le code. Une fois un saut de ligne rencontré on le réinitialise à 0. on oublie pas d'incrémenter à chaque caractère en fonction du nombre de caractères présents.
Par exemple le mot "int" à 3 caractères on va incrémenter **colno** de 3.

Eviter les commentaires :

En s'inspirant de nos précédents TP dont le TP2 pour éviter de lire les lexème contenus dans les commentaires aussi bien avec la syntaxe **/* */** que **//** nous avons ajouté des règles **/* <COM>** avec **BEGIN COM** et **BEGIN INITIAL** pour que le programme puisse ignorer les commentaires mais aussi ignorer les fonctions qu'il pourrait reconnaître

dans le commentaire ainsi que les double slash (//). Nous avons bien fait attention à respecter les tabulations, les sauts de lignes, les opérateurs, les espaces (en trop) , les noms de variables et fonctions (accompagnés de la parenthèse et de l'accolade) ainsi que les chiffres et les underscore pour que le programme soit le plus optimal possible.

Difficultées rencontrées pour l'analyseur lexical :

Au début nous avons mis les "/*" entre crochets comme ceci [/*]. Nous avions oublié que cela signifie '/' ou '*'. Nous avons corrigé cela après avoir cherché notre erreur dans le reste du code.

La partie de l'analyseur lexical s'est avérée plutôt simple comme première étape. il a fallu juste ajouter les mots de grammaire que notre programme doit accepter. Nous avons eu quelques avertissements sur nos variables **lineno**(int) et **yytext** (char *) qui étaient dans le programme C de l'analyseur syntaxique (permet de compiler flex et renvoie **yytext** si la grammaire est respectée) en changeant le type par "**extern**". Lorsque la partie de l'analyseur lexical à été bien faite. Place à l'analyseur syntaxique qui va permettre de suivre l'ordre du langage pour que la syntaxe soit correcte.

De l'analyseur Lexical à Analyseur syntaxique :

Ayant reçu une partie du fichier Bison nous avons commencé à lire et comprendre comment celui-ci fonctionnait. On nous demande de créer une règle pour pouvoir initialiser une variable locale dans une fonction, on a alors commencé à faire deux règles **DeclVarsGlobale** et **DeclVarsLocale**.

On a surtout travaillé sur **DeclVarsLocale** car **DeclVarsGlobale** a été donné dans le fichier d'origine.

Au tout début nous utilisions **Declarateurs** dans la règle **DeclVarsLocale** mais nous avons eu de nombreux warnings. Donc pour régler ce problème nous avons créé une autre règle **InitVarsLocale**. Dans cette règle **InitVarsLocale** on a listé tous les cas possibles.

Par exemple :

```
int a;
int a, b, c;
int a = 3;
int a = 3, b = 2;
int a = a + b;
int a = 3, b, c, d = 2;
etc..
```

Nous avons compris qu'il fallait faire des règles avec des non terminaux récursifs gauche afin d'arriver à des terminaux permettant la structuration du langage.

Dans un fichier test, nous avons mis **plusieurs sortes d'exemples** de fichiers permettant de tester si le programme s'exécute sans message d'erreur. "**good**" est le dossier qui contient des fichiers tpc sans message d'erreur. Le dossier **"syn-error"** permet de voir par exemple si on a une faute de grammaire que l'analyseur ne reconnaît pas et renvoie qu'il y a une erreur.

Nous avons essayé de voir quels sont les cas qu'il faut différencier. En testant de nombreuses fois des règles, nous sommes arrivés à un résultat qui avait l'air de fonctionner.

Car à ce moment là on a fait nos tests mais nous n'étions jamais sur, c'est à dire que si nos tests étaient faux ou alors incomplets, malgré tout le travail fourni **on aurait pu être faussé** par nos propres testes. Alors, pour éviter d'oublier une syntaxe ou de se tromper sur une syntaxe on pas mal testé plusieurs cas (en mettant des noms de variables suivi de virgules ainsi que leur affectation, en évaluant les opérateurs , les affectations, les opérations avec le LEFT VALUE, etc). Puis nous sommes arrivés à un résultat satisfaisant. Cette étape nous a pris un petit peu plus de temps (moins d'une semaine).

Jusqu'à la phase intermédiaire

Nous avons été dans les temps pour finir la phase intermédiaire, la qualité de travail que l'on a fourni pour ce rendu à été au rendez-vous. Il ne nous restait plus que quelques jours avant le dépôt. Nous avons fini la dernière étape qui était de rajouter les options qui nous permettaient par la suite d'avoir plusieurs choix lors de l'exécution du programme. Pour cela nous avons utilisé la bibliothèque conseillé pour le projet **getopt** qui ressemble au switch du C. Cela va nous permettre de choisir le type d'affichage souhaité de notre programme(affichage de l'arbre abstrait qui sera à construire pour le rendu final avec "-t ou —tree" , ou affichage d'une description de l'interface utilisateur avec -h –help)

Deuxième phase : Arbre abstrait

Pour cette phase nous devions créer un arbre abstrait à l'aide des fonctions donnés dans le fichiers tree.c.

Avant de créer l'arbre abstrait du projet, on s'est pas mal entraîné sur le TP4 pour s'aider à reproduire le même arbre sur le projet.

On a créé des Token par la suite pour bien trier notre arbre abstrait, on a rangé ces token par "famille".

Des type node pour permettre de produire des noeuds à PROG DeclVars DeclFoncts Declarateurs DeclFonct EnTeteFonct Corps etc..

Ensuite un token <Identifier> IDENT pour permettre de lire le nom de variable par exemple un int a;

Puis des tokken IF ELSE WHILE RETURN OR AND VOID EQ ADDSUB TYPE ORDER CHARACTER NUM DIVSTAR IDENT.

le but est de montrer quel token appartiennent à quelle variable, ils peuvent avoir plusieurs valeur possibles (ex: addsub soit + soit -)

CHARACTER considéré comme un String parce que affiche les guillemets ‘ et aussi pour pouvoir ‘\n’ ‘\t’ ou ‘x’

Par ailleurs, on avait toujours un avertissement à la compilation, on a demandé au professeur comment la corriger, la réponse était en amphi. Nous avons ajouté cette règle dans notre bison afin de la retirer → -%expect 1 et c’est réglé.

Ensuite pour donner à ces noms de token leur vrai nom de variable dans le programme nous avons créé un %union permettant l’identification de chaque type de token qu’on a fait.

Un node sur une structure Node*, un entier , un caractère et un identifiant pour chaque variable dans le programme. Bien évidemment on a enrichi également notre tree.c pour les chaînes de caractères utilisées dans le programme dans une liste statique de chaînes de caractères.

Construction de l’arbre abstrait

Une fois que l’on a accompli cette tâche qui était plutôt longue mais assez simple, nous sommes arrivées à l'étape qui demandait beaucoup de rigueur et de ne surtout pas se tromper. La construction de l’arbre avec les fonctions fournies , makeNode(), addChild() , addSibling(), printTree() et deleteTree(). Ces fonctions ont été très faciles à utiliser pour nous car nous avons eu une démonstration de ces dernières au CM. La difficulté à été de savoir quoi mettre au bon endroit pour que l’arbre ait une forme assez convenable. Une fois ceci terminé on a obtenu un **arbre de dérivation**. c’est à ce moment la que la tâche s’est avérée assez compliquée car il fallait le **rendre abstrait** , c’est à dire le plus simplifié possible, avec le plus de terminaux et le moins de non terminaux, nous avons utilisé comme outils cette règle du cours “$$ = $1” et quelques fonctions de réductions comme addsibling pour simplifier au maximum les non terminaux appelés en trop . Cela à été la tâche la plus longue et difficile pour nous à comprendre

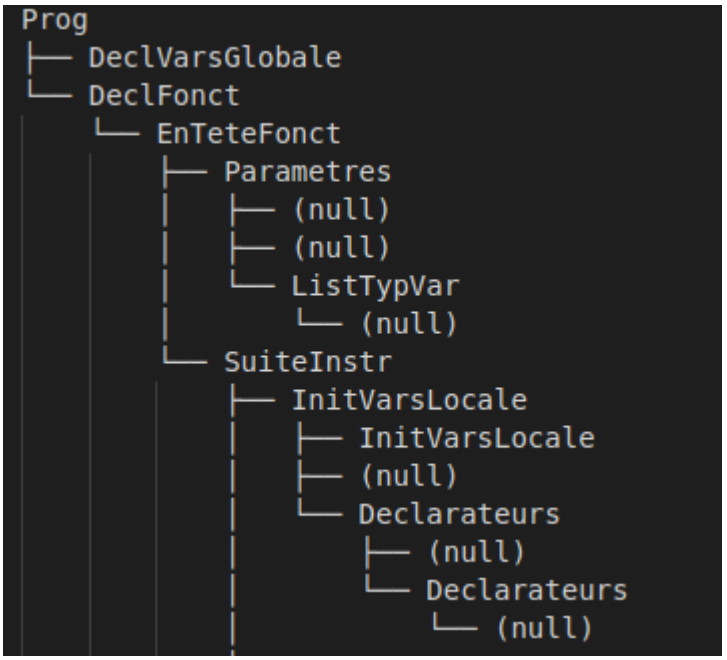
On a utiliser l’exemple du cours pour tester notre arbre:

```c
int main(void){
    int a, b;
    while(b != 0){
        if (a > b){
           a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

On s’est aidé de cet exemple pour corriger notre arbre. mettre les branches fils à la bonne place.

Difficultés rencontrées lors de la construction de l’arbre:

Voici la première version de notre arbre :



Il y avait des null surement car nous y avions pas encore déclarer dans les enum les terminaux et non terminaux

Alors au début nous avons d'abord ajouter dans les fichiers tree.c et tree.h les non terminaux et terminaux comme ceci :

```
static const char *StringFromLabel[] = {
  "Prog",
  "DeclVarsGlobale",
  "DeclVarsLocale",
  "InitVarsLocale",
  "Declarateurs",
  "DeclFoncts",
  "DeclFonct",
  "EnTeteFonct",
  "Parametres",
  "ListTypVar",
  "Corps",
  "SuiteInstr",
  "Instr",
  "Exp",
  "TB",
  "FB",
  "M",
  "E",
  "T",
  "F",
  "LValue",
  "Arguments",
  "ListExp",
  "Type",
  "Ident",
  "Void",
  "If",
  "Else",
  "While",
  "Return",
  "Or",
  "And",
  "Eq",
```

Pareil dans le fichier.h sauf qu'on écrit dans le enum de label_t.

```
Prog
├── DeclVarsGlobale
│   ├── Type
│   └── Ident
└── DeclFoncts
    └── DeclFonct
        ├── EnTeteFonct
        │   ├── Type
        │   ├── Ident
        │   └── Parametres
        │       └── Void
        └── Corps
            ├── DeclVarsLocale
            │   ├── DeclVarsLocale
            │   ├── Type
            │   └── InitVarsLocale
            │       ├── Ident
            │       └── InitVarsLocale
            │           └── Ident
```

En corrigeant on se retrouve avec un arbre de ce type. Le problème ici est par exemple DeclVarsLocale qui est initialisé sans enfants dans certains cas. Il ne sert donc à rien.

On a pu corrigé avec $$ = NULL;

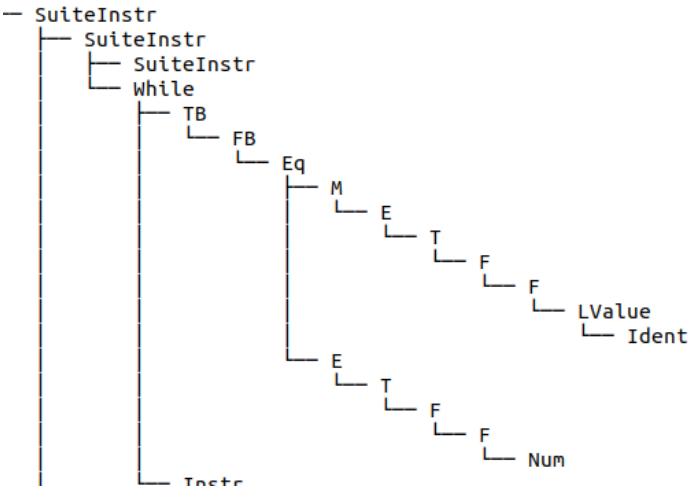
On a aussi eu ce problème qu'on peut voir dans l'image de droite. Enfaite tout les nœuds était créer même ceux "inutiles". Enfaite faisait comme ceci pour créer un noeud :

```
TB        { $$ = makeNode(TB); }
```

Pour corriger cela on a alors remplacer par:

```
TB              { $$ = $1; }
```

On fait pareil pour TB, FB, M, E, T, F.



On a alors cherché à remplacer les IDENT par le nom des variables, les NUM par le bon entier et pareillement pour le TYPE, ADDSUB etc. Notre première version ressemblait à cela

```
%union {
   Node* node;

   char ident[64];
   char type[5];
   char operateur;
   char character[4];
   int num;
```

```
}
```

Une variable pour chaque terminal.
Dans notre fichier tree.c on les initialise dès qu'on crée un node.

```
Node *makeNode(label_t label) {
 Node *node = malloc(sizeof(Node));
 if (!node) {
   printf("Run out of memory\n");
   exit(1);
 }
 node->label = label;
 node-> firstChild = node->nextSibling = NULL;
 node->lineno=lineno;

 strcpy(node->ident, "");       // Ident
 strcpy(node->type, "");        // Type
 node->operateur = ' ';         // Operateur
 strcpy(node->character, "");   // Character
 node->num = 0;                 // Num

 return node;
}
```

Nous avions fait cela pour "faciliter" (en réalité c'est l'inverse ce n'est pas pratique) l'affichage dans la fonction printTree ou nous faisions cela :

```
 if(strcmp(node->ident, "") != 0){              // Ident
   printf("Ident : %s", node->ident);
 } else if(strcmp(node->type, "") != 0){       // Type
   printf("Type : %s", node->type);
 } else if(node->operateur != ' ') {           // Operateur
   printf("Operator : %c", node->operateur);
 } else if(strcmp(node->character, "") != 0) { // Character
   printf("Character : %s", node->character);
 } else if(node->num != 0) {                   // Num
```

```
        printf("Num : %d", node->num);
      } else {
        printf("%s", StringFromLabel[node->label]);
      }
```

Le problème est que nous initialisons et déclarons des variables pour rien car un nœud n'utilise que 1 variable. Il ne peut pas être un identifiant et un entier en même temps. De plus les comparaisons dans le printTree() dépendait de comment on déclarait nos variables dans le makeNode.

On a alors réfléchi a plutôt créer une structure **union** qui contiendrait toute nos variables comme ceci :

```
typedef union {
 char ident[64];
 char type[5];
 char operateur;
 char character[4];
 int num;
 char boolean[4];
 char compare[4];
} Data;
```

On avait alors créé une fonction pour chaque variable dans tree.c pour permettre d'initialiser un nœud plus facilement dans le fichier bison.

```
Node* makeIdentNode(label_t label, char* ident);
Node* makeTypeNode(label_t label, char* type);
Node* makeOperatorNode(label_t label, char operator);
Node* makeCharacterNode(label_t label, char* character);
Node* makeNumNode(label_t label, int num);
Node* makeBooleanNode(label_t label, char* boolean);
Node* makeCompareNode(label_t label, char* compare);
```

Cela ne servait à rien de différencier les variables "ident" et "type" par exemple car ce sont tous les deux des strings. Donc leurs initialisation, déclaration et même l'assignation était pareil.

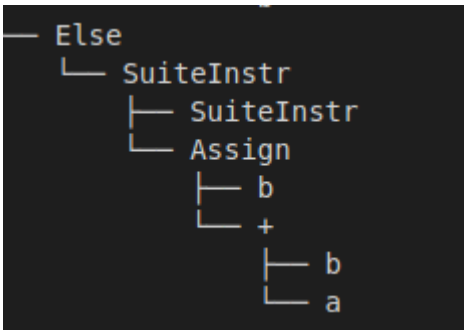
On a alors encore modifier notre code pour finalement resté sur cette version

```
typedef union {
 char string[64];
 char character;
 int num;
} Data;
```

Cela permet de seulement avoir 3 fonctions pour créer un noeud

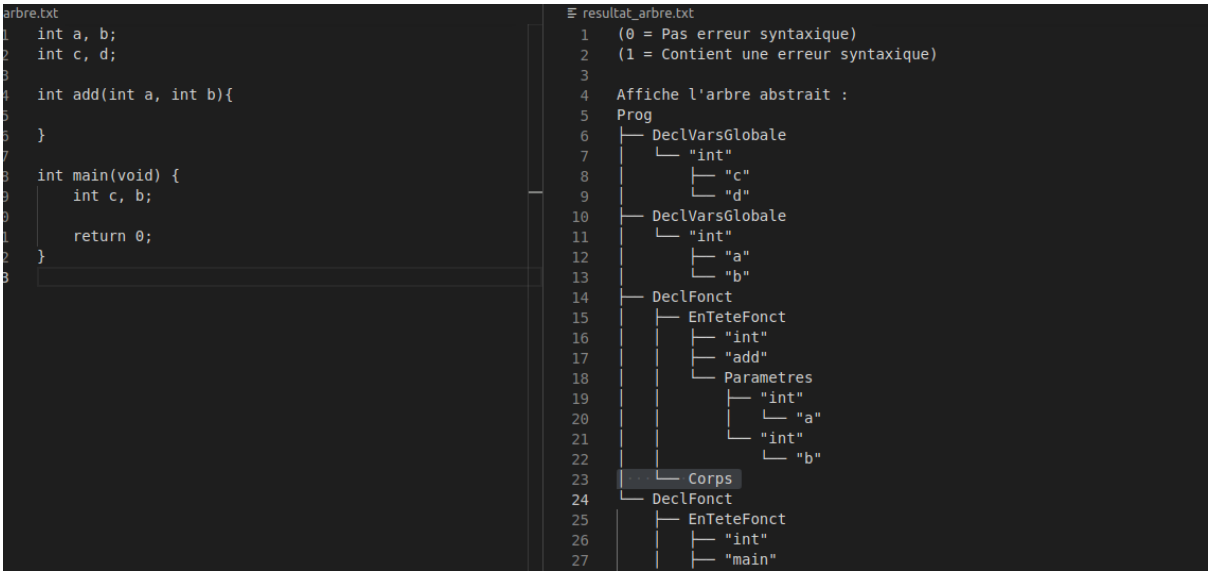
```
Node *makeDataStringNode(label_t label, char* data);
Node *makeDataCharNode(label_t label, char data);
Node *makeNumNode(label_t label, int num);
```

Et on a aussi changé la fonction printTree() avec un switch plutôt que de tester si la valeur d'une variable était nulle ou pas comme précédemment.

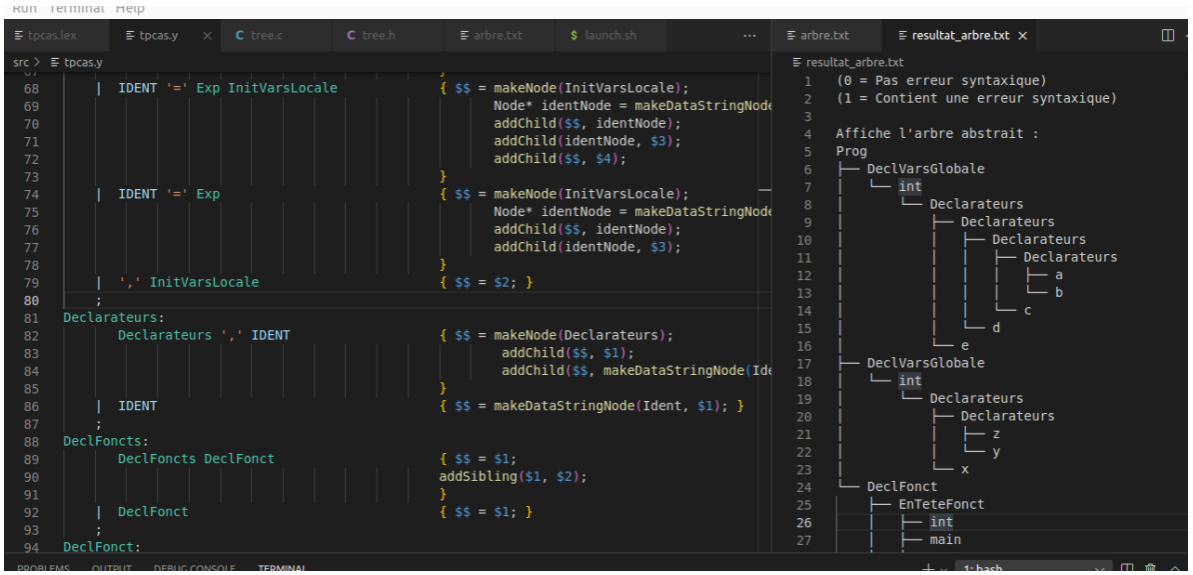


On avait des cas ou des nœuds n'avaient pas de fils. Pour éviter cela on fait cela dans tout les règles vide : $$ = NULL;

Nous avons fait quelques dessins pour comprendre comment mettre correctement nos non terminaux par ordre afin qu'ils puissent s'afficher correctement dans l'arbre.



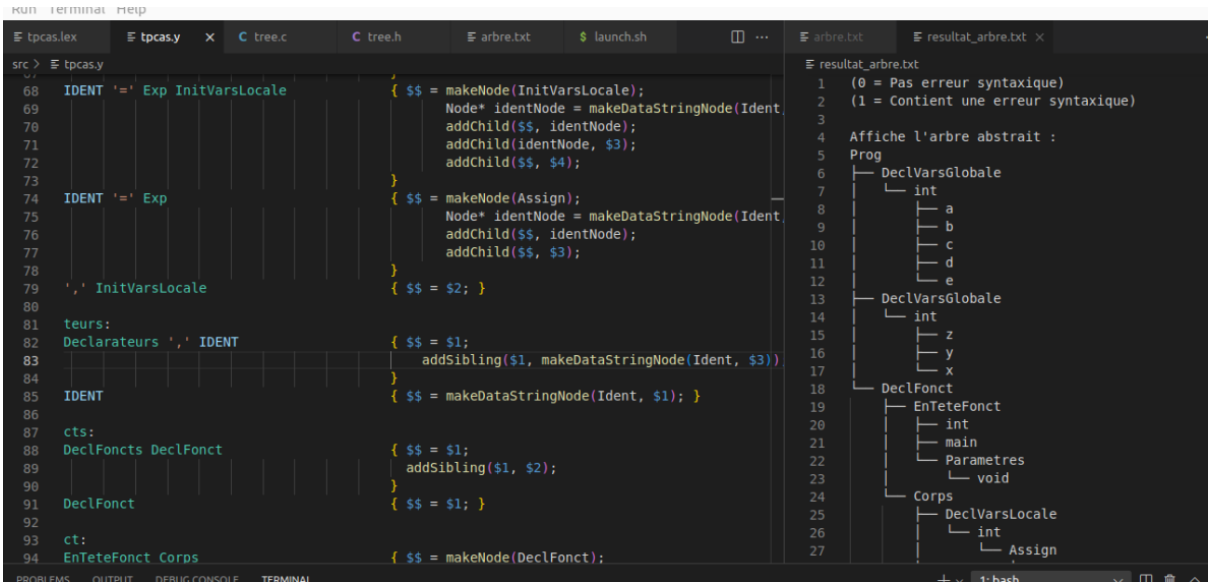
Voici un arbre abstrait à peu près correct durant notre réflexion. A cette on avait demandé à un professeur si c'était correcte de relié les DeclVarsGlobale et DeclFonct a Prog. Il nous a dit d'utiliser plutôt un père DeclVarsGlobales et un autre DeclFonct pour stocker d'un côté les déclarations globales et de l'autre les fonctions.



```
| IDENT '=' Exp InitVarsLocale          { $$ = makeNode(InitVarsLocale);
                                          Node* identNode = makeDataStringNode
                                          addChild($$, identNode);
                                          addChild(identNode, $3);
                                          addChild($$, $4);
                                        }
| IDENT '=' Exp                         { $$ = makeNode(InitVarsLocale);
                                          Node* identNode = makeDataStringNode
                                          addChild($$, identNode);
                                          addChild(identNode, $3);
                                        }
| ',' InitVarsLocale                    { $$ = $2; }
;
Declarateurs:
    Declarateurs ',' IDENT              { $$ = makeNode(Declarateurs);
                                           addChild($$, $1);
                                           addChild($$, makeDataStringNode(Ide
                                        }
| IDENT                                 { $$ = makeDataStringNode(Ident, $1); }
;
DeclFoncts:
    DeclFoncts DeclFonct                { $$ = $1;
                                        addSibling($1, $2);
                                        }
| DeclFonct                             { $$ = $1; }
;
DeclFonct:
```

```
(0 = Pas erreur syntaxique)
(1 = Contient une erreur syntaxique)

Affiche l'arbre abstrait :
Prog
├── DeclVarsGlobale
│   └── int
│       └── Declarateurs
│           ├── Declarateurs
│           │   ├── Declarateurs
│           │   │   ├── Declarateurs
│           │   │   │   ├── a
│           │   │   │   └── b
│           │   │   └── c
│           │   └── d
│           └── e
├── DeclVarsGlobale
│   └── int
│       └── Declarateurs
│           ├── Declarateurs
│           │   ├── z
│           │   └── y
│           └── x
└── DeclFonct
    ├── EnTeteFonct
    │   ├── int
    │   ├── main
```



```
IDENT '=' Exp InitVarsLocale          { $$ = makeNode(InitVarsLocale);
                                        Node* identNode = makeDataStringNode(Ident
                                        addChild($$, identNode);
                                        addChild(identNode, $3);
                                        addChild($$, $4);
                                      }
IDENT '=' Exp                         { $$ = makeNode(Assign);
                                        Node* identNode = makeDataStringNode(Ident
                                        addChild($$, identNode);
                                        addChild($$, $3);
                                      }
',' InitVarsLocale                    { $$ = $2; }

teurs:
Declarateurs ',' IDENT                { $$ = $1;
                                        addSibling($1, makeDataStringNode(Ident, $3))
                                      }
IDENT                                 { $$ = makeDataStringNode(Ident, $1); }

cts:
DeclFoncts DeclFonct                  { $$ = $1;
                                        addSibling($1, $2);
                                      }
DeclFonct                             { $$ = $1; }

ct:
EnTeteFonct Corps                     { $$ = makeNode(DeclFonct);
```

```
(0 = Pas erreur syntaxique)
(1 = Contient une erreur syntaxique)

Affiche l'arbre abstrait :
Prog
├── DeclVarsGlobale
│   └── int
│       ├── a
│       ├── b
│       ├── c
│       ├── d
│       └── e
├── DeclVarsGlobale
│   └── int
│       ├── z
│       ├── y
│       └── x
└── DeclFonct
    ├── EnTeteFonct
    │   ├── int
    │   ├── main
    │   └── Parametres
    │       └── void
    └── Corps
        ├── DeclVarsLocale
        │   └── int
        │       └── Assign
```

Sur la première image ici on avait un erreur de structuration des branches avec les int a, b, c, d, e qui n'étaient pas alignés pareils. Nous avons corrigé cela sur la deuxième image.

Problème finales :

Dans ce projet nous avons 2 gros problème.

Le premier était l 'inversion des variable locale dans une fonction. Quand on dit inversion c'est-à- dire que dans l'arbre abstrait on commence à afficher les variables de fin au début.

Voici un exemple :

On peut voir ici que cela affiche c puis b puis a. On a alors simplement inversé 2 variables dans la règle bison **InitVarsLocale**. Cela est simple comme solution mais nous avons pris beaucoup de temps à trouver et à régler.

Le deuxième problème était de supprimer l'affichage des deux fils de **Corps** : **DeclVarsLocale** et **SuiteInstr**. Leurs affichage dans l'arbre ne servait à rien lorsque eux même n'avait pas de fils. Cependant dès qu'on créer Corps on ajouté deux fils.

Pour terminer, nous avons eu un souci d'affichage lorsque le corps du programme est vide, l'arbre affichait DeclvarLocal et SuiteInstr. On a pris beaucoup de temps à comprendre comment pouvoir les effacer mais une fois qu'ils étaient effacés le corps de la fonction ne s'affichait pas en entier ou donnait une segfault. Pour résoudre ce problème nous avons posé comme conditions que si DeclVarLocal->firstChild et SuiteInstr->firstChild étaient null on DeclVarLocal et SuiteInstr valent null sinon si le corps est rempli . on ajoute à corps en tant que fils ces derniers. et cela a marché au final avec un peu de persévérance.

Conclusion:

En conclusion, ce projet à été une expérience très enrichissante pour créer un tel analyseur syntaxique. Nous en sommes très satisfaits d'avoir vérifié plusieurs cas afin d'arriver à quelque chose qui puisse donner une exécution parfaite sans messages d'erreur. Nous sommes très contents de la qualité du travail fourni. mise à part la difficulté de structuration de l'arbre abstrait et quelques autres petits détails à corriger. Cet analyseur lexical et syntaxique s'est avéré très instructif à produire et nous sera utile pour le projet de compilation plus tard et si nous voulions créer également de nous même un langage qui pourra être par la suite un langage universel. mais bien sûr, il faudra fournir davantage de travail.