Université Gustave Eiffel

Projet d’Analyse syntaxique

Licence d’informatique

—2024-2025—

Auteurs :

- CHIBANE Boualem.
- BENVENISTE Alexandre.

Introduction

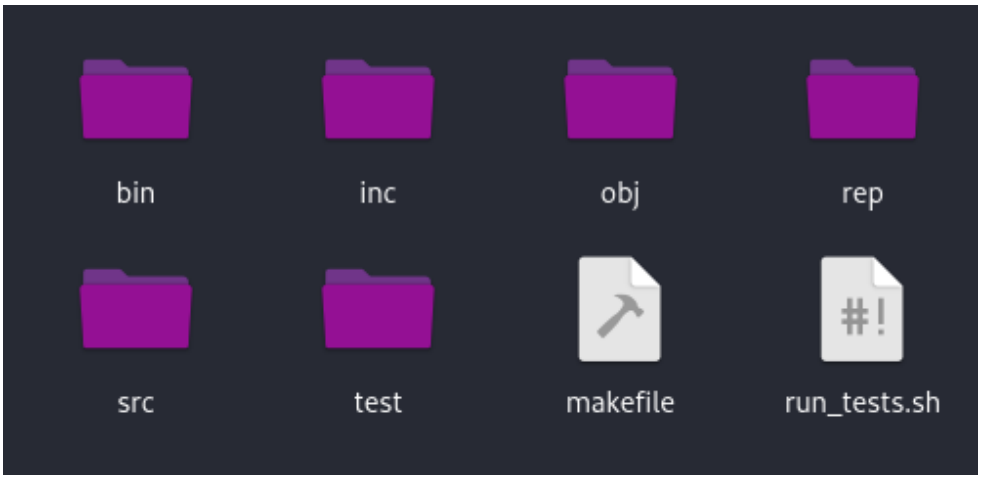
Ce projet consiste à développer un analyseur syntaxique pour le langage "tpc", un sous-ensemble du langage C, en utilisant les outils **Flex** et **Bison**. L'objectif est de vérifier la validité lexicale et syntaxique des programmes écrits dans ce langage, de générer un arbre syntaxique abstrait (AST) pour les programmes corrects, et de signaler précisément les erreurs pour les programmes incorrects.

Le projet inclut également un script d’automatisation pour exécuter des tests sur des programmes corrects et incorrects, et retourne des rapports d’analyse.

Objectifs

- Vérification lexicale et syntaxique: Détecter les erreurs dans les programmes.
- Construction de l'AST : Représenter la structure des programmes corrects sous forme d’un arbre syntaxique abstrait.
- Automatisation des tests : Fournir des outils pour tester et valider le fonctionnement de l’analyseur.

Structure du projet :



- ➢ bin : le fichier exécutable
- ➢ obj : les fichiers .o (lex.yy.o, y.tab.o ,tree.o …)
- ➢ src : les fichiers tree.c|h tpc.lex et tps-2024-2025.y
- ➢ inc : les fichiers .h (tree.h)
- ➢ rep: le rapport du projet
- ➢ test :
  - good : tous les fichiers tests corrects
  - syn-err : tous les fichiers tests incorrects

- ➢ makefile
- ➢ run tests.sh : le fichier script.

Conception :

1. Analyse lexicale (Flex)

L'analyse lexicale est réalisée à l'aide de **Flex**. Les règles définissent les différents tokens reconnus dans le langage tpc :

- **Identifiants (`IDENT`)**
- **Nombres (`NUM`)** :
- **Caractères (`CHARACTER`)**
- **Gestion des retours à la ligne** (lineno++)

2. Analyse syntaxique (Bison)

L'analyse syntaxique est définie à l'aide de **Bison**. la grammaire dans notre projet respecte celle donné sur elearning.

3. Arbre syntaxique abstrait

L'AST est construit à l'aide de nœuds représentant les éléments syntaxiques du programme :

Fonctions:

```
- Node *makeNode(char * label);
- void addSibling(Node *node, Node *sibling);
- void addChild(Node *parent, Node *child);
- void deleteTree(Node*node);
- void printTree(Node *node);
```

-

Exemple: Entrée:

-

```
1  int main(void) {
2      int a;
3      a = 2;
4      return a;
5  }
```

AST:

```
Program
└── Functions
    └── Function
        ├── Head
        │   ├── int
        │   ├── main
        │   └── Parameter
        │       └── void
        └── Body
            ├── Vars
            │   └── int
            │       └── a
            └── Suite_instr
                ├── Eq
                │   ├── a
                │   └── 2
                └── Return
                    └── a
```

3. Gestion des erreurs

L'analyseur détecte et signale les erreurs lexicales et syntaxiques, en affichant le numéro de ligne et une description de l'erreur. Exemple :

```
syntax error
Erreur pendant l'analyse à la ligne 6, code de retour : 1
```

Tests :

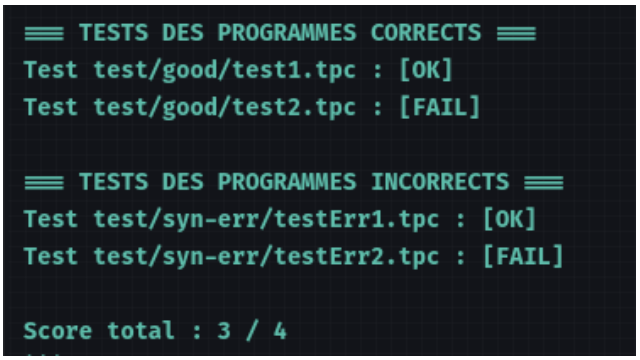
Organisation des tests:

Les tests sont divisés en deux catégories :

1. Tests corrects : Programmes valides dans test/good .
2. Tests incorrects : Programmes contenant des erreurs dans test/syn-err .

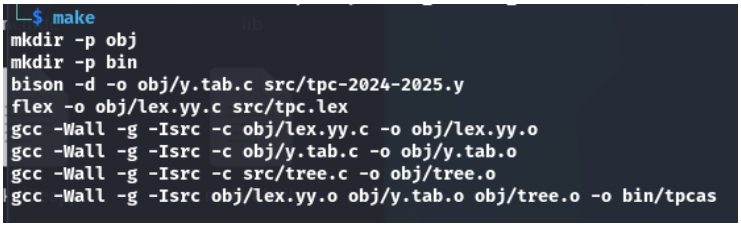
Script d'exécution des tests

Le script run_tests.sh automatise l'exécution des tests et génère un rapport contenant les résultats et un score global. Exemple de sortie :



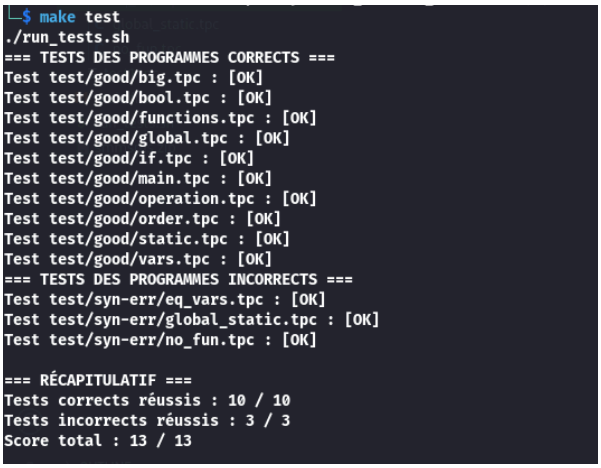
Compilation du programme :

Pour compiler le programme, on se positionne dans le répertoire du projet et utiliser la commande: - “make”



lancement des testes :

Pour utiliser le script qui évalue nos fichiers de tests, on utilise la commande : - “make test”



Voici le bilan du script, on peut voir qu’il y a 2 blocs.

Bloc des tests de programmes corrects :

Affichent tous les tests du dossier good et montre pour chaque test si celui-ci est valide.

[OK] -> le test est passé (pas d’erreur dans ce test correct)

[FAILED] -> le test a échoué (contient des erreurs)

Bloc des tests de programmes incorrects :

La même chose dans le dossier syn-err et montre pour chaque test si celui-ci est valide.

[OK] -> le test est passé (contient des erreurs)

[FAILED] -> le test a échoué (ne contient pas des erreurs)

Bloc récapitulatif:

Affiche le score (tests corrects, incorrects),Enfin le score total.

Execution du programme :

Pour utiliser le programme, on utilise la commande :

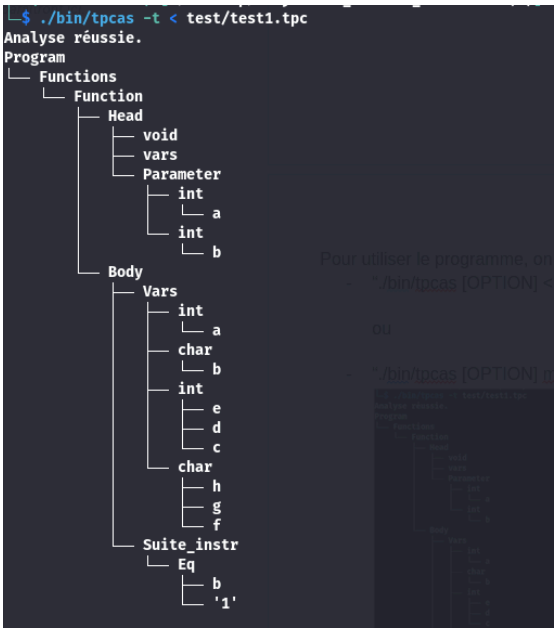
1. ./bin/tpcas [option] mon_fichier.tpc
2. ./bin/tpcas [option] < mon_fichier.tpc

Voici les différentes options:

➔ -t, --tree : permet d'afficher l'arbre abstrait du programme.

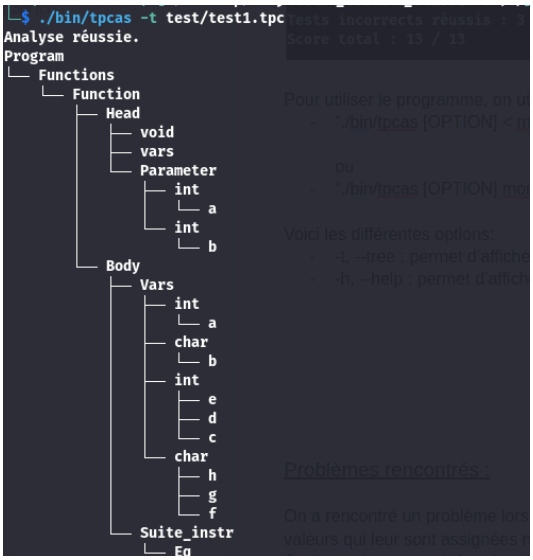
➔ -h, --help : permet d'afficher le manuel d'utilisation du programme.

- ./bin/tpcas [OPTION] < mon_fichier.tpc



ou

- ./bin/tpcas [OPTION] mon_fichier.tpc



-

Conclusion

Ce projet a permis de construire un analyseur syntaxique complet pour le langage tpc, capable de détecter les erreurs et de produire un arbre syntaxique abstrait pour les programmes corrects. L'utilisation de Flex et Bison a été essentielle pour répondre aux exigences du projet.

problèmes rencontrés:

On a rencontré un problème lors de l'affichage de l'arbre abstrait. Les variables utilisées et les valeurs qui leur sont assignées n'étaient pas affichées dans l'arbre, Puis on a eu l'idée d'utilisé la fonction sprintf pour formater ces derniers dans des chaines de caractère.
On à aussi eu un problème lors de la création de fichier de test qui n'était pas valide d'après le lexer, alors que celui-ci respecte la grammaire, s' ils étaient créés sur windows alors que sur linux le problème ne se produit pas avec le même contenu.