Université Gustave Eiffel

Projet d'Analyse syntaxique

Licence d'informatique

—2025-2026—

Le but du projet est d'écrire un analyseur syntaxique en utilisant les outils `flex` et `bison`.

Le langage source est un petit langage de programmation appelé TPC, qui est presque un sous-ensemble du langage C. Quand le fichier d'entrée en TPC ne contient pas d'erreur lexicale ni syntaxique, votre analyseur syntaxique devra le traduire en un arbre abstrait. Vous pouvez utiliser le module `tree.c` fourni pour les TD et l'adapter à votre projet. L'analyseur devra pouvoir afficher l'arbre abstrait sur la sortie standard, mais on ne demande pas qu'il sauvegarde l'arbre abstrait dans un fichier.

Le projet est à faire en binôme ou seul. Vous pouvez vous choisir un binôme librement dans n'importe quel groupe. Si vous n'en avez pas trouvé, vous pouvez contacter Eric Laporte.

Pour vous situer dans l'avancement de votre projet, nous vous fournirons un bac à sable avec lequel vous lancerez automatiquement votre projet sur vos jeux d'essais, et vous obtiendrez un score qui évaluera si votre projet détecte les erreurs lexicales et syntaxiques.

La date limite de rendu est le mardi 30 décembre 2025 à 23h55.

1 Définition informelle du langage source

Un programme TPC est une suite de fonctions. Chaque fonction est constituée de déclarations de variables (locales à la fonction), et d'une suite d'instructions. Il peut y avoir des variables de portée globale. Elles sont alors déclarées avant les fonctions.

Les types de base du langage sont `int` (entier signé) et `char`. Le mot-clé `void` est utilisé pour indiquer qu'une fonction ne fournit pas de résultat ou n'a pas d'arguments.

2 Définition des lexèmes

Les identificateurs sont constitués d'une lettre ou d'un symbole souligné ("_"), suivi éventuellement de lettres, chiffres, symbole souligné. Il y a distinction entre majuscule et minuscule. Les mots-clés comme `if`, `else`, `return`, etc., doivent être écrits en minuscules. Ils sont reconnus par l'analyseur lexical et ne peuvent pas être utilisés comme identificateurs.

Les lexèmes pour les constantes numériques sont des suites de chiffres.

Les caractères littéraux dans le programme (1) sont délimités par le symbole `'`, comme en C. Dans les caractères littéraux, la barre oblique inverse ("\") est utilisée pour déspécialiser `'` et pour spécialiser `n` et `t` : `\n` et `\t` sont le saut de ligne et la tabulation.

Les commentaires peuvent être délimités soit par `/*` et `*/`, soit par `//` et la fin de la ligne. Les délimiteurs `/*` et `*/` fonctionnent un peu comme des parenthèses, mais à un seul niveau : même s'il y a eu un `/*` supplémentaire à l'intérieur du commentaire, le premier `*/` termine le commentaire.

Les autres lexèmes sont :

`=`	: opérateur d'affectation
`+`	: addition ou plus unaire
`-`	: soustraction ou moins unaire
`*`	: multiplication
`/` et `%`	: division et reste de la division entière
`!`	: négation booléenne
`==`, `!=`, `<`, `>`, `<=`, `>=`	: les opérateurs de comparaison
`&&`, `||`	: les opérateurs booléens
`;` et `,`	: le point-virgule et la virgule
`(`, `)`, `{` et `}`	: les parenthèses et les accolades

(1). Un caractère littéral est une constante de type caractère définie comme le code ASCII d'un caractère, par exemple `'@'`.

Chacun de ces lexèmes sera identifié par l'analyse lexicale qui devra produire une erreur pour tout élément ne faisant pas partie du lexique du langage.

3 Terminaux de la grammaire

Dans ce qui suit,

- **CHARACTER** et **NUM** désignent respectivement un caractère littéral et une constante numérique ;
- **IDENT** désigne un identificateur ;
- **TYPE** désigne un nom de type simple qui peut être `int` ou `char` ;
- **EQ** désigne les opérateurs d'égalité (’`==`’) et d'inégalité (’`!=`’) ;
- **ORDER** désigne les opérateurs de comparaison ’`<`’, ’`<=`’, ’`>`’ et ’`>=`’ ;
- **ADDSUB** désigne les opérateurs ’`+`’ et ’`-`’ (binaire ou unaire) ;
- **DIVSTAR** désigne les opérateurs ’`*`’, ’`/`’ et ’`%`’ ;
- **OR** et **AND** désignent les deux opérateurs booléens ’`||`’ et ’`&&`’.
- Les mots-clés sont notés par des terminaux qui leur sont identiques à la casse près.

L'instruction nulle est ’`;`’.

4 Grammaire du langage TPC

Voir le fichier `tpc-2025-2026.y`.

5 Travail demandé

5.1 Extension du langage, développement logiciel et rédaction du rapport

Modifiez la liste de lexèmes et la grammaire du langage de façon à autoriser les types structures. La déclaration des types structures doit être conforme au langage C et chaque structure doit avoir au moins un champ. Une structure doit pouvoir avoir un champ de type structure :

```
struct aircraft {
   int height, width;
   struct engine motor;
   };
```

Une fois un type structure déclaré, on doit pouvoir utiliser la syntaxe `struct my_structure` (le mot-clé `struct` suivi de l'identificateur) pour déclarer des variables, des paramètres, des champs de structures, et des types de retour de fonctions :

```
struct aircraft my_plane, my_shuttle, my_satellite;
```

Pour simplifier par rapport au langage C, on demande de respecter les contraintes suppémentaires suivantes :

- On interdit de définir une variable de type structure en même temps que l'on définit le type structure. Il faut obligatoirement séparer les deux définitions, comme ci-dessus.
- On interdit les `typedef`.
- On interdit de déclarer un type structure sans lui donner un nom en même temps, dans la même déclaration.
- On interdit de déclarer le nom d'un type structure sans déclarer ses champs en même temps.
- On interdit de déclarer un nouveau type structure à l'intérieur de la déclaration d'un type structure. Pour déclarer des types structures imbriqués, il faut donc obligatoirement avoir déclaré la structure interne avant de déclarer l'externe.

Vous pouvez faire des modifications supplémentaires à la grammaire, mais elles ne doivent pas affecter le langage engendré, parce que cela pourrait compliquer le projet de compilation au 2e semestre.

Écrivez un analyseur syntaxique de ce langage en utilisant `flex` pour l'analyse lexicale et `bison` pour l'analyse syntaxique.

Décrivez dans un rapport vos choix et les difficultés que vous avez rencontrées.

5.2 Organisation

Nous vous demandons de respecter l'organisation suivante. *Pour évaluer vos projets, nous supposerons que vous l'aurez fait.* Le répertoire que vous déposerez doit s'appeler `ProjetASL3_NOM1_NOM2`, contenir à la racine un makefile nommé `makefile` et au moins les 5 sous-répertoires suivants :

- `src` pour les fichiers sources écrits par les humains,
- `bin` pour le fichier binaire (votre analyseur doit être nommé `tpcas`),
- `obj` pour les fichiers intermédiaires entre les sources et le binaire,
- `rep` pour votre rapport,
- `test` pour les jeux d'essais, avec deux sous-répertoires :
  - `good`
  - `syn-err`.

Votre analyseur doit avoir l'interface utilisateur suivante.

- Ligne de commande :
  - on doit au moins pouvoir lancer le compilateur par `./tpcas [OPTIONS]` (dans ce cas, la bonne façon de donner le fichier d'entrée TPC en ligne de commande est de faire une redirection par `<`) ; *pour évaluer vos projets, nous lançons des tests automatiques qui utilisent cette commande* ;
  - vous pouvez aussi implémenter, en plus, la ligne de commande `./tpcas [OPTIONS] FILE.tpc`
- Options [2] : au moins
  `-t`, `--tree` affiche l'arbre abstrait sur la sortie standard
  `-h`, `--help` affiche une description de l'interface utilisateur et termine l'exécution
- Valeur de retour :
  - 0 si le programme source ne contient aucune erreur lexicale ni syntaxique
  - 1 s'il contient une erreur lexicale ou syntaxique [3]
  - 2 ou plus pour les autres sortes d'erreurs : ligne de commande, fonctionnalité non implémentée, mémoire insuffisante...

  Quand nous évaluerons votre projet, nos tests automatiques enregistreront chaque valeur de retour, et le score de votre compilateur dépendra du nombre de fois où il renvoie les bonnes valeurs.

5.3 Autres consignes

Messages d'erreur Les messages d'erreur lexicale ou syntaxique donneront le numéro de ligne, et le numéro dans la ligne d'un caractère proche de l'erreur.

Arbres abstraits Dans l'arbre abstrait produit par votre analyseur syntaxique, chaque opérateur devra être représenté par un nœud interne de l'arbre, et ses opérandes par les fils de ce nœud. Chaque liste devra être transformée de façon à ce que tous les éléments de la liste soient représentés par les fils d'un même nœud interne, et ces fils devront apparaitre dans le même ordre que dans le fichier d'entrée. Aucun nœud ne pourra représenter un des caractères `, ; ( ) { } [ ]` ni être étiqueté par un de ces caractères.

Tests Écrivez deux jeux de tests, un pour les programmes TPC lexicalement et syntaxiquement corrects, et un autre pour les programmes incorrects. Implémentez un script de déploiement des tests, qui produit un rapport unique donnant les résultats de tous les tests et un score global. (Nous utilisons nos propres jeux d'essais pour l'évaluation.)

Quand nous évaluerons votre projet, une partie de la note viendra de nos tests automatiques qui enregistreront chaque valeur de retour, et votre compilateur gagnera des points s'il renvoie les bonnes valeurs.

Nommage et dépôt Déposez votre projet sur la plateforme elearning dans la zone prévue à cet effet, sous la forme d'une archive tar compressée de nom "ProjetASL3_NOM1_NOM2.tar.gz", qui, au désarchivage, crée un répertoire "ProjetASL3_NOM1_NOM2" contenant le projet. L'utilisation du bac à sable ne dépose pas votre projet et ne vous dispense pas de le déposer vous-même.

(2). Pour analyser la ligne de commande vous pouvez utiliser la fonction `getopt()`.
(3). On ne demande pas que l'analyseur puisse redémarrer après erreur, mais si vous implémentez cette fonctionnalité, l'analyseur doit renvoyer 1 s'il a redémarré après une erreur.