Bergeot Loïc & Mathieu Julien

Projet d'Analyse syntaxique

Rapport de la création d'un Analyseur syntaxique pour le langage TPC

Décembre 2024

Table des matières

Introduction

Ce projet vise à créer un analyseur syntaxique pour le langage TPC qui est un langage minimaliste s'inspirant du langage C. L'objectif principal est de traduire un programme écrit en TPC en un arbre abstrait structuré, en respectant les règles syntaxiques définies.

Pour cela, nous avons utilisé les outils Flex et Bison. Flex effectue une analyse lexicale afin d'identifier les différents lexèmes et détecter les erreurs lexicales. Les données extraites vont ensuite être transmises à Bison, qui effectue l'analyse syntaxique et construit l'arbre abstrait en suivant la grammaire du langage TPC.

Ce projet inclut la définition et l'extension du langage TPC, l'implémentation de l'analyseur, et la mise en place de tests pour valider le comportement attendu. Ce document présente les choix techniques réalisés, les difficultés rencontrées, ainsi que les résultats obtenus.

Organisation

Lors des séances de TP, mon binôme et moi avons eu l'occasion de travailler ensemble et d'échanger sur le projet. En dehors des séances, nous avons pu échanger via Discord pour améliorer le code. Nous avons utilisé Git pour gérer les versions de notre code et de GitLab pour centraliser et partager nos fichiers.

I - Guide d'utilisation

A - Structure du projet

Le projet de est constitué de 5 dossiers et d'un makefile :

src/ : Contenant le fichier bison parser.y, le fichier flex lexer.lex, tree.h et tree.c pour la construction et l'affichage de l'arbre abstrait.

bin/ : Contenant l'exécutable nommé *tpcas*.

obj/ : Contenant les fichiers intermédiaires, les fichiers objets.

test/ : Contenant good/ et syn-err/ contenant des fichiers tests qui sont corrects et des tests qui provoquent une erreur syntaxique.

B - Utilisation du makefile

Le makefile comporte plusieurs règles simplifiant l'utilisation de ce projet :

make all

Compiler tous les fichiers et générer l'exécutable tpcas.

make clean

Supprimer tous les fichiers générés par le *make all* et ne garde donc que les fichiers présents dans src/ (tree.h, tree.c, parser.y et lexer.lex)

make test

Exécuter tous les tests et afficher un bilan du résultat de ces tests.

Pour obtenir l'arbre abstrait d'un programme il faut utiliser la commande :

bin/./tpcas < test/good/test01.tpc

II - Etapes d’implémentation

Etape 1 - Création du lexer

La première chose que nous avons fait a été de créer le lexer pour le langage TPC avec Flex. Son rôle est de transformer le code en une série de lexèmes, repérer les erreurs au passage, et garder des informations utiles comme les numéros de ligne et de caractère pour que le débogage soit plus simple.

A - Gestion des commentaires

- Deux modes distincts ont été définis pour les commentaires :
  les commentaires sur une seule ligne (`//`) et les commentaires multilignes (`/* */`).
- Nous avons utilisé les directives `BEGIN` de Flex pour alterner entre ces modes, avec une attention particulière pour gérer correctement les fins de lignes (`\n`) et incrémenter le numéro de ligne.

B - Support des caractères spéciaux dans les littéraux

- Pour les caractères littéraux,
  le traitement de séquences échappées (`\n`, `\t`, `\'`) a été ajouté avec une expression régulière dédiée (`'([^\\]|\\[nt\'])'`).
  Cela nous garantit que seuls les caractères valides sont reconnus, tout en respectant les règles du langage.

C - Suivi de position

- En plus du numéro de ligne (`lineno`), nous avons introduit une variable `cara` pour suivre la position relative dans la ligne. Cette information est utile pour les messages d'erreur détaillés et pour déboguer les erreurs lexicales.

D - Traitement des mots-clés et identificateurs

- Les mots-clés (`int`, `char`, `while`, etc...) sont identifiés séparément des identificateurs généraux. Pour cela, nous avons utilisé des règles spécifiques pour les mots-clés, précédant les expressions pour les identificateurs. Cela permet d'éviter tout conflit.

E - Extension pour le mot-clé `static`

- Par la suite, le mot static a été ajouté au langage TPC. Il a été traité comme un mot-clé standard. La gestion de sa position dans le flux d'entrée a également été intégrée pour garantir des messages d'erreur précis.

Difficultés rencontrées

1. Les commentaires imbriqués
    - Bien que les spécifications ne permettent pas de commentaires imbriqués, il a été difficile d'assurer qu'un `*/` mal placé ne cause pas d'erreurs inattendues. Cela a nécessité plusieurs tests pour valider le comportement.
2. L'optimisation des expressions régulières
    - Certaines expressions, notamment pour les caractères littéraux, ont nécessité un ajustement pour éviter des correspondances incorrectes. Par exemple, une séquence comme `'\\'` (une barre oblique inversée échappée) devait être correctement interprétée.
3. La gestion de la position avec `cara`
    - Le suivi de la position dans la ligne s'est avéré plus complexe que prévu, notamment lors de la gestion des espaces et des tabulations, qui affectent directement la valeur de `cara`.
4. L'ordre des règles entre mots-clés et identificateurs
    - L'ordre des règles dans Flex a été crucial. Les mots-clés devant être reconnus avant les identificateurs, une mauvaise organisation des règles entraînait des erreurs d'identification.

Résultats

Le lexer est actuellement capable de :

- Identifier correctement les lexèmes du langage TPC, y compris les nouveaux ajouts comme les variables locales `static`.
- Gérer les erreurs lexicales en indiquant avec précision la ligne et la position.
- Suivre efficacement les transitions entre différents états comme les modes de commentaires.

Etape 2 - Création du parser

L'objectif de ce parser est d'utiliser les lexèmes fournis par le lexer pour construire un arbre abstrait représentant le programme. Il peut repérer les erreurs de syntaxe (oubli d'un point virgule, mauvaise déclaration de fonction, mauvaise déclaration de variable ...).
Voici les principales étapes de son développement :

A - Construction de l'arbre abstrait

- L'arbre abstrait est construit en utilisant les fonctions définies dans `tree.c` et `tree.h`. Ces modules offrent des outils pour créer des nœuds, ajouter des enfants ou des frères, et parcourir l'arbre. Chaque règle grammaticale de Bison est donc associée à une structure d'arbre cohérente, ce qui permet une représentation claire et hiérarchique des programmes écrits en TPC.

B - Identification des erreurs

- Le parser identifie et signale les erreurs syntaxiques grâce à la fonction `yyerror()`. En cas de non-conformité avec la grammaire, cette fonction produit des messages détaillés contenant le type d'erreur (par exemple, "point-virgule manquant" ou "parenthèse non fermée"), le numéro de ligne (`lineno`) ou la position exacte du caractère problématique (`cara`).

C - Modification de la grammaire

- Modification de la grammaire pour ajouter les variables locales static. Pour cela nous avons ajouté plusieurs règles pour spécifier les variables locales pour que les variables static ne soient pas permises hors d'une fonction (variables static locales interdites).
- Nous avons également utilisé des directives comme `%nonassoc`, `%left`, et `%expect 1` pour gérer les priorités et réduire les conflits shift/reduce. Ces directives ont permis d'assurer un comportement prédictible, notamment dans les constructions conditionnelles imbriquées (`if-else`).

Etape 3 - Création du Makefile

Premièrement, le Makefile nous servait uniquement à compiler et produire notre exécutable.

Puis, il a été développé pour que nous puissions nettoyer les fichiers afin de pouvoir compiler correctement le programme.

En fin de projet, nous avons eu besoin d'une règle pour exécuter notre programme sur tous nos tests présents dans le répertoire test/. Le Makefile utilise des variables pour généraliser les chemins des fichiers (comme `SRC` et `OBJ`), afin de rendre la maintenance plus simple et efficace.

Etape 4 - Les tests

Au début, les tests ont été effectués à l'aide des fichiers d'exemple fournis par les enseignants. Par la suite, nous avons conçu nos propres scénarios pour couvrir les cas limites et les nouvelles fonctionnalités, comme la gestion des variables locales `static`.

Nous avons ensuite décidé d'automatiser l'exécution de tous les tests à l'aide de la règle make. Un script additionnel a été utilisé pour comparer les sorties produites avec les résultats attendus, nous fournissant ainsi un résumé des réussites et des échecs. Cela a permis un débogage rapide et itératif lors du développement.

III - Conclusion

Le parser, associé à son Makefile et aux tests automatisés, offre une solution robuste pour analyser les programmes en TPC. Il respecte la grammaire étendue, identifie efficacement les erreurs, et génère un arbre abstrait fidèle. L'approche par tests systématiques a permis de valider la plupart des scénarios rencontrés, rendant l'analyseur fiable et complet.