Yanis Mzyene
Walid Amara

Rapport : Analyseur Syntaxique

1. Introduction

Le but de ce projet est de concevoir et d'implémenter un analyseur syntaxique pour un petit langage de programmation appelé **TPC**, en utilisant les outils **Flex** pour l'analyse lexicale et **Bison** pour l'analyse syntaxique.

---

2. Définition du Langage Source

Un programme écrit en TPC est composé d'une suite de déclarations globales et de fonctions.
Les fonctions peuvent contenir des déclarations de variables locales ainsi qu'une suite d'instructions.

Les types de base du langage sont :

- `int` : entier signé
- `char` : caractère

Le mot-clé `void` est utilisé pour indiquer qu'une fonction ne retourne aucune valeur ou ne possède aucun paramètre.

Des variables globales peuvent être déclarées avant les fonctions, et des variables locales peuvent être déclarées à l'intérieur des fonctions.

---

3. Lexèmes du Langage

Les principaux lexèmes reconnus par l'analyseur lexical sont :

- **Identificateurs** : composés de lettres, chiffres et du caractère `_`, avec distinction entre majuscules et minuscules.
- **Constantes numériques** : suites de chiffres.
- **Caractères littéraux** : délimités par des apostrophes (`'`), avec gestion des caractères spéciaux (`\n`, `\t`, `\'`).
- **Commentaires** :

    - commentaires multi-lignes (`/* ... */`)
    - commentaires sur une ligne (`// ...`)
- **Opérateurs** :
  `=`, `+`, `-`, `*`, `/`, `%`, `!`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `&&`, `||`
- **Délimiteurs** :
  `;`, `,`, `(`, `)`, `{`, `}`

Les mots-clés tels que `if`, `else`, `while`, `return`, `struct`, etc., sont réservés et ne peuvent pas être utilisés comme identificateurs.

Toute séquence ne correspondant pas à un lexème valide provoque une erreur lexicale.

---

4. Grammaire du Langage TPC

La grammaire du langage TPC est définie à l'aide de Bison.
Elle permet de structurer correctement :

- les déclarations de variables,
- les déclarations de fonctions,
- les instructions (affectations, expressions, conditions, boucles, etc.).

Chaque règle de la grammaire contribue à la construction de l'arbre syntaxique abstrait, où les opérateurs sont représentés par des nœuds internes et leurs opérandes par des fils.

---

5. Extension du Langage : Types Structures

Une extension importante du langage a consisté à ajouter la gestion des **types structures**, inspirée du langage C.

5.1 Déclaration des Structures

Les structures doivent être déclarées avec le mot-clé `struct`, suivi d'un identificateur et d'un ensemble de champs.
Chaque structure doit contenir au moins un champ.

Exemple :

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

Une structure peut contenir un champ dont le type est une autre structure, à condition que celle-ci ait été déclarée auparavant.

5.2 Utilisation des Structures

Une fois déclarée, une structure peut être utilisée pour :

- déclarer des variables,
- déclarer des paramètres de fonctions,
- définir le type de retour d'une fonction.

Exemple :

struct aircraft my_plane, my_shuttle;

Les variables de type structure peuvent être utilisées dans des affectations et des expressions, y compris l'accès à leurs champs avec l'opérateur `.`.

5.3 Contraintes Supplémentaires

Afin de simplifier l'implémentation, certaines restrictions ont été imposées :

- Interdiction de déclarer des structures globales après les fonctions.
- Interdiction de déclarer des structures locales après des instructions.
- Interdiction de définir une structure et une variable de ce type dans la même déclaration.
- Interdiction des `typedef`.
- Obligation de nommer chaque structure lors de sa déclaration.
- Interdiction des structures anonymes ou imbriquées directement dans d'autres structures.

---

6. Organisation du Projet

Le projet respecte l'arborescence imposée :

- `src` : fichiers sources (Flex, Bison, C)
- `bin` : contient le binaire `tpcas`
- `obj` : fichiers objets intermédiaires
- `rep` : contient le rapport du projet
- `test` :

    - `good` : programmes TPC corrects
    - `syn-err` : programmes contenant des erreurs syntaxiques

Un `makefile` est présent à la racine pour automatiser la compilation.

---

7. Interface Utilisateur

L'analyseur syntaxique est lancé via la commande :

./tpcas [OPTIONS]

ou avec redirection :

./tpcas < fichier.tpc

Options disponibles :

- `-t` ou `--tree` : affiche l'arbre syntaxique abstrait
- `-h` ou `--help` : affiche l'aide

Valeurs de retour :

- `0` : aucune erreur lexicale ou syntaxique
- `1` : erreur lexicale ou syntaxique détectée
- `≥ 2` : erreur de ligne de commande ou autre erreur interne

---

8. Détails de l'Implémentation

8.1 Analyse Lexicale (Flex)

Flex est utilisé pour reconnaître les lexèmes du langage.
Les commentaires sont ignorés correctement, et les erreurs lexicales sont signalées avec le numéro de ligne et la position dans la ligne.

8.2 Analyse Syntaxique (Bison)

Bison permet de vérifier la structure syntaxique du programme et de construire l'arbre abstrait.

Chaque règle de la grammaire crée un nœud correspondant dans l'AST à l'aide du module `tree.c`.

---

9. Problèmes Rencontrés et Solutions

9.1 Gestion des Structures

L'ajout des structures a nécessité des modifications importantes dans la grammaire, notamment pour respecter les contraintes imposées (ordre des déclarations, structures imbriquées, portée).

9.2 Construction de l'Arbre Abstrait

La représentation correcte des accès aux champs (`struct.variable`) a demandé une attention particulière afin que l'arbre abstrait respecte la structure attendue.

9.3 Gestion des Erreurs

Un soin particulier a été apporté à l'affichage des messages d'erreur, afin d'indiquer précisément la ligne et la colonne où l'erreur est détectée.

---

10. Conclusion

Ce projet a permis de mettre en pratique les concepts d'analyse lexicale et syntaxique à l'aide de Flex et Bison.
L'analyseur développé est capable de reconnaître un langage proche du C, incluant désormais la gestion des structures, de détecter les erreurs et de produire un arbre syntaxique abstrait conforme aux spécifications.

Ce travail constitue une base solide pour un futur projet de compilation plus avancé, notamment pour l'analyse sémantique et la génération de code.