المدرسة الحسنية للأشغال العمومية
ECOLE HASSANIA DES TRAVAUX PUBLICS

وزارة التجهيز والماء
ⵜⴰⵎⴰⵡⴰⵙⵜ ⵏ ⵜⵣⵔⴼⵜ ⴷ ⵡⴰⵎⴰⵢ
MINISTERE DE L'EQUIPEMENT ET DE L'EAU

RAPPORT DU PROJET :

MISE EN PLACE D'UN COMPILATEUR EN C++ :

Réalisé par :

AJALE Saad
AOUANET Bahaeddine
BENHADDU Lahcen
ERRAZI FatimaZahra
FRIHA Oussama

Encadré par :

MR. KANDALI KHALID

Table des matières

Introduction

Dans le domaine de l'informatique, les compilateurs jouent un rôle fondamental en permettant la transformation des langages de programmation de haut niveau en un code compréhensible par les machines. Ils représentent un pilier essentiel pour l'exécution des programmes, la portabilité des applications et l'optimisation des performances.
Ce projet s'inscrit dans le cadre d'une exploration approfondie de la conception et de l'implémentation d'un compilateur. L'objectif principal est de créer un outil capable de traduire un langage de programmation défini en un code exécutable, en respectant les standards des étapes de compilation : l'analyse lexicale, l'analyse syntaxique, l'analyse sémantique et la génération de code.
En utilisant le langage C++, reconnu pour sa performance et sa proximité avec le matériel, ce projet vise non seulement à approfondir les connaissances sur les compilateurs, mais également à mettre en pratique des notions avancées en algorithmique, structures de données et gestion des erreurs.

Chapitre I

Présentation Générale du Projet

1. Contexte et motivation
2. Objectifs du projet
3. Présentation du langage cible du compilateur

1. Contexte et motivation

Les compilateurs sont essentiels pour traduire les langages de programmation en code machine, rendant les programmes exécutables et portables. Ce projet vise à concevoir un compilateur simple, implémenté en C++, afin de comprendre les étapes clés de la compilation (analyse lexicale, syntaxique, sémantique, etc.), tout en développant des compétences en algorithmique et en programmation avancée. En explorant les interactions entre matériel et logiciel, ce travail allie théorie et pratique pour maîtriser les bases de la compilation et produire un outil fonctionnel.

2. Objectifs du projet

L'objectif principal de ce projet est de concevoir et implémenter un compilateur fonctionnel en C++. Les objectifs spécifiques incluent :

1. **Compréhension des principes de la compilation :** Maîtriser les étapes fondamentales (analyse lexicale, syntaxique, sémantique, optimisation et génération de code).
2. **Création d'un langage cible** : Définir les règles syntaxiques et sémantiques d'un langage de programmation simplifié.
3. **Développement technique** : Utiliser C++ pour implémenter les différentes étapes du compilateur et gérer efficacement la mémoire, les structures de données et les algorithmes.
4. **Gestion des erreurs** : Concevoir un système robuste de détection et de gestion des erreurs pour assurer la fiabilité du compilateur.
5. **Production de livrables** : Générer un code machine ou intermédiaire exécutable, avec une documentation détaillée du processus et des résultats.
6. **Renforcement des compétences** : Améliorer les compétences en conception logicielle, en résolution de problèmes et en développement C++.

3. Présentation du langage cible du compilateur

Le langage cible du compilateur est C++, choisi pour sa performance élevée, sa syntaxe claire et sa capacité à gérer des structures complexes. Il est largement utilisé dans les domaines critiques grâce à son support des paradigmes multiples, sa gestion fine de la mémoire et ses structures de contrôle puissantes. Ce projet vise à concevoir un compilateur capable de transformer un sous-ensemble de programmes en C++ en code machine exécutable, tout en mettant en œuvre les étapes fondamentales de la compilation.

Qualités de C++ :

- **Performance élevée :** Idéal pour les applications exigeantes.
- **Flexibilité :** Supporte plusieurs paradigmes (procédural, orienté objet, générique).
- **Gestion mémoire :** Contrôle précis grâce aux pointeurs et références.
- **Portabilité :** Compatible avec diverses plateformes.
- **Richesse :** Dispose de bibliothèques standard étendues et de vastes ressources.
- **Efficacité :** Permet une optimisation fine du code.

Chapitre II

Principes et Notions de base

1. Les principes de base d’un compilateur
2. Étapes de compilation
3. Outils et technologies existants pour la construction de compilateurs
4. Comparaison des approches classiques et modernes

1. Les principes de base d’un compilateur

Un compilateur est un programme qui permet de traduire un code source écrit dans un langage de programmation (appelé langage source) en un autre langage, souvent en code machine ou en bytecode, pour que le programme puisse être exécuté sur un ordinateur. Il repose sur plusieurs principes de base, dont l’analyse lexicale qui consiste à découper le code source en unités appelées tokens. Ensuite, l’analyse syntaxique vérifie que la structure du code respecte la grammaire du langage. L’analyse sémantique intervient pour vérifier la validité logique et s’assurer que le programme a un sens cohérent. Le compilateur génère ensuite un code intermédiaire qui est indépendant de la machine avant de procéder à des optimisations, afin de rendre le programme plus rapide ou plus compact. Enfin, il génère le code machine ou bytecode exécutable qui pourra être exécuté sur la machine cible.

2. Étapes de compilation

Les étapes de la compilation passent par plusieurs phases :

1. **Prétraitement** : Cette étape prépare le code avant la compilation. Par exemple, elle inclut des fichiers externes dans le code ou remplace des macros définies (comme #define en C) par des valeurs spécifiques. Cela permet de simplifier le code avant de le traiter davantage dans les étapes suivantes.
2. **Analyse lexicale** : Le compilateur découpe le code en petites unités appelées tokens. Les tokens sont des éléments de base du programme, comme des mots-clés (if), des identifiants (nom d'une variable), ou des opérateurs (+, -). Cela aide le compilateur à comprendre les différentes parties du code.
3. **Analyse syntaxique** : À ce stade, le compilateur organise les tokens en un arbre syntaxique, qui structure le code en fonction des règles de la grammaire du langage de programmation. L'arbre syntaxique montre comment les éléments du code sont liés entre eux, permettant ainsi de vérifier si le code suit une structure correcte.
4. **Analyse sémantique** : Cette étape vérifie la logique du programme. Le compilateur s'assure que les types de données sont correctement utilisés (par exemple, qu'on ne tente pas d'additionner une chaîne de caractères avec un nombre) et que les opérations sont valides.

5.Génération de code intermédiaire : Après validation, le compilateur génère un code intermédiaire. Ce code est indépendant du matériel spécifique (processeur ou système d'exploitation), ce qui permet de l'optimiser avant de devenir un programme exécutable. Cela facilite la portabilité du code sur différentes plateformes.

6.Optimisation : Dans cette dernière étape, le compilateur améliore le code pour le rendre plus rapide et plus efficace. Par exemple, il peut supprimer des calculs redondants ou réorganiser certaines instructions pour mieux utiliser les ressources matérielles (comme la mémoire ou le processeur).

3. Outils et technologies existants pour la construction de compilateurs

Ce tableau représente des outils et technologies existants pour la construction de compilateurs en C++ :

Outil/Technologie	Description
Flex	Générateur d'analyseur lexical qui découpe le code en tokens (mots-clés, identifiants, opérateurs).
Bison	Générateur d'analyseur syntaxique pour créer un arbre de syntaxe à partir des tokens.
LLVM	Infrastructure de compilation modulaire, permettant l'analyse, l'optimisation et la génération de code machine.
Clang	Front-end de LLVM pour C++ qui analyse le code source et génère du code intermédiaire LLVM.
GCC (GNU Compiler Collection)	Suite d'outils incluant un compilateur C++ (G++) largement utilisé pour la compilation multiplateforme.
G++	Compilateur C++ de la suite GCC, qui compile le code source C++ en un programme exécutable.
Intel C++ Compiler (ICC)	Compilateur optimisé pour les processeurs Intel, améliorant les performances des applications C++.

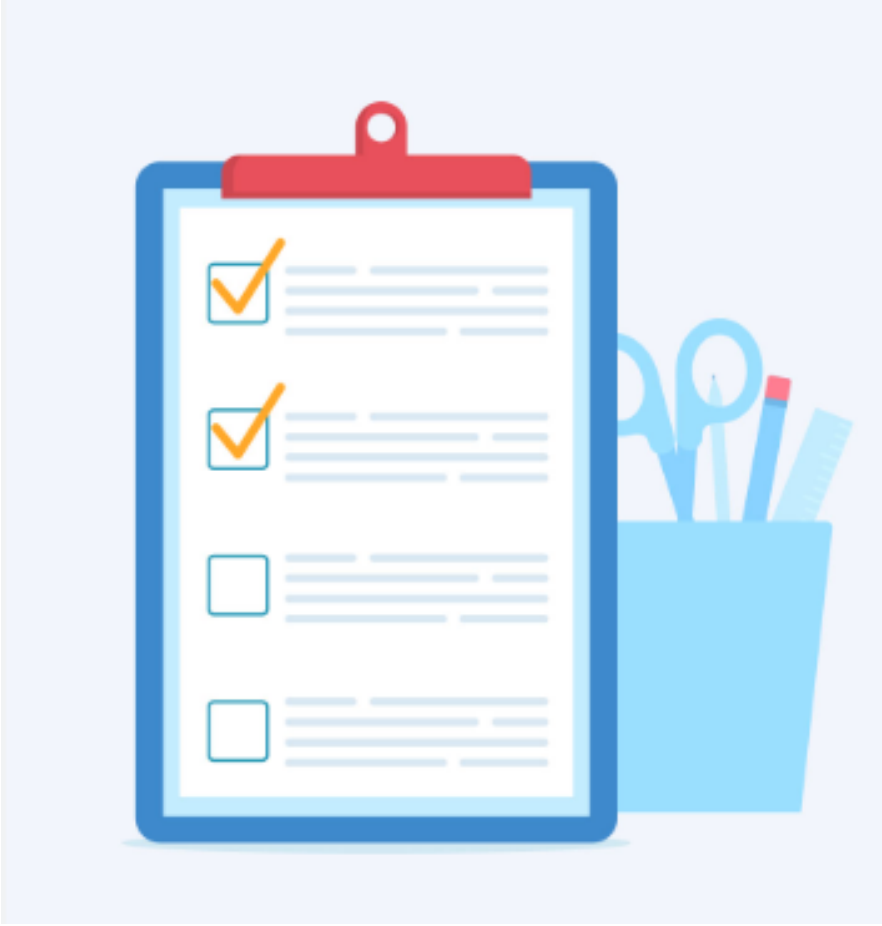
4. Comparaison des approches classiques et modernes

Les approches classiques de la construction de compilateurs reposent souvent sur des outils et des techniques établis, tels que les générateurs d'analyseurs lexicaux et syntaxiques comme **Lex** et **Yacc**. Ces outils suivent des processus séquentiels bien définis, avec une attention particulière à la gestion de la syntaxe et des erreurs. Cependant, les approches modernes se sont enrichies avec des concepts plus avancés comme l'analyse sémantique et l'optimisation de code, tirant parti d'outils comme **LLVM** et **Clang**, qui permettent une plus grande flexibilité et des performances accrues. Les compilateurs modernes sont également mieux équipés pour gérer des langages plus complexes et des optimisations de bas niveau grâce à leur architecture modulaire. De plus, les approches modernes intègrent des techniques d'analyse statique plus sophistiquées et de débogage, facilitant ainsi le développement de compilateurs pour des langages modernes et l'optimisation automatique du code. En résumé, les approches modernes se distinguent par leur capacité à optimiser le code de manière plus fine et leur adaptabilité à de nouveaux environnements de développement, tandis que les approches classiques restent solides et efficaces pour les langages et les environnements plus simples.

Chapitre III

Élaboration du cahier des charges

1. Fonctionnalités attendues
2. Contraintes
3. Livrables
4. Planning



L'objectif de ce projet est de développer un compilateur pour un langage de programmation simple. Le compilateur doit être capable d'analyser et de traduire du code source en effectuant des vérifications lexicales, syntaxiques et sémantiques. Ce projet inclut l'analyse d'expressions arithmétiques et logiques, ainsi que la gestion de structures comme les parenthèses, les accolades et les crochets.

Créer un compilateur fonctionnel pour un langage simple avec des fonctionnalités de base.
Permettre la tokenisation et l'analyse lexicale du code source.
Effectuer des vérifications syntaxiques et sémantiques, incluant la gestion des erreurs.
Proposer une interface de débogage pour analyser et corriger les erreurs.

1. Fonctionnalités attendues

1.1 Spécifications du langage :

Types de données :

Support des nombres (int, float), des chaînes de caractères (string), et des identifiants.

Instructions :

Déclarations de variables : int x = 10;
Assignations : x = x + 5;
Affichage des variables : print(x);
Structures conditionnelles : if, else, while.
Utilisation d'opérateurs binaires et de comparateurs : +, -, *, /, >, <, ==, !=.

Commentaires :

Commentaires en ligne : // Commentaire
Autres caractéristiques :
Gestion des parenthèses, accolades, et crochets.
Gestion des opérateurs logiques (&&, ||, !).

1. Fonctionnalités attendues

1.2 Étapes de compilation :

Analyse lexicale :

Le compilateur doit identifier les différents types de jetons : mots-clés, opérateurs, identifiants, numéros, commentaires.

Analyse syntaxique :

Vérification que le code respecte la grammaire du langage. Le programme doit être structuré correctement avec des parenthèses et des accolades équilibrées, des points-virgules après les instructions, etc.

Analyse sémantique :

Vérification des types de données, de la portée des variables, et des erreurs comme les opérateurs mal utilisés ou les affectations invalides.

Génération de code intermédiaire :

Générer un bytecode ou un autre format d'instruction intermédiaire.

Génération de code machine (optionnel) :

Traduction en instructions exécutables (si nécessaire pour le projet).

1. Fonctionnalités attendues

1.3 Débogage et gestion des erreurs :

Le compilateur doit inclure un débogueur pour afficher les erreurs lexicales, syntaxiques et sémantiques.
Les erreurs doivent être affichées avec des messages clairs et précis pour faciliter la correction.
Le débogueur doit aussi vérifier l'équilibre des parenthèses et la présence de points-virgules après chaque instruction.

1.4 Tests unitaires :

Des tests unitaires doivent être réalisés pour vérifier que le compilateur fonctionne comme prévu.
Les tests doivent inclure la vérification des différents types de jetons, ainsi que des cas d'erreur dans le code source.

1.5 Interface utilisateur :

Une interface pour saisir le code source du programme et afficher les résultats du débogage.
Affichage des tokens extraits et des erreurs détectées.

2. Contraintes

2.1 Techniques :

Le projet doit être développé en C++.
Utilisation de bibliothèques comme iostream, fstream, et cctype pour gérer l'entrée/sortie et la manipulation des caractères.
L'utilisation d'un débogueur intégré dans le compilateur pour l'analyse des erreurs est requise.

2.2 Performances :

Le compilateur doit être capable de traiter un programme contenant plusieurs milliers de lignes de code sans retards significatifs.

2.3 Documentation :

Le projet doit inclure une documentation détaillant la structure du compilateur, les étapes de compilation, ainsi que les tests unitaires effectués.

3. Livrables

Le code source complet du compilateur, bien commenté et documenté.
Un rapport décrivant la conception du compilateur et le processus de développement.
Les tests unitaires associés au projet.

4. Planning

Phase 1 (Analyse lexicale) : 1 semaine

Phase 2 (Analyse syntaxique) : 1 semaine

Phase 3 (Analyse sémantique) : 1 semaine

Phase 4 (Débogage et gestion des erreurs) : 1 semaine

Phase 5 (Génération de code intermédiaire) : 1 semaine

Phase 6 (Tests unitaires et documentation) : 1 semaine

Chapitre IV

Implementation en C++

Dans ce chapitre, nous allons explorer les étapes nécessaires à la création d'un compilateur en nous basant sur le code source fourni. Chaque phase du processus de compilation sera expliquée en détail, avec une mise en lumière des objectifs et des fonctionnalités associées. Ce processus inclut plusieurs étapes fondamentales, notamment la tokenisation, l'analyse syntaxique et l'analyse sémantique, qui seront mises en œuvre à travers des classes comme **Lexer** et **Debugeur**.

Nous suivrons les étapes suivantes :

Étape 1 : Importation des bibliothèques et définitions globales

Dans cette étape, nous importons les bibliothèques nécessaires et définissons les types de jetons (tokens) utilisés tout au long du processus.

Étape 2 : Classe UniteLexical pour la représentation des tokens

La classe UniteLexical permet de modéliser chaque unité lexicale avec son type et sa valeur, facilitant ainsi la manipulation des jetons lors des analyses.

Étape 3 : Classe Lexer pour l'analyse lexicale

Le rôle de la classe Lexer est de parcourir le code source afin d'extraire les unités lexicales (tokens). Elle identifie les mots-clés, les opérateurs, les identifiants, et d'autres éléments syntaxiques.

Étape 4 : Classe Debugeur pour les vérifications et l'analyse syntaxique

La classe Debugeur effectue les vérifications nécessaires sur la structure du code, détecte les erreurs de syntaxe et valide la cohérence des jetons extraits.

Étape 5 : Fonction main pour l'interaction utilisateur et la coordination

Cette étape centralise l'exécution du programme. Elle permet de recevoir le code source en entrée, d'exécuter les analyses via les classes précédentes, et d'afficher les résultats.

Étape 1 : Importation des bibliothèques et définitions globales

Code source :

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <cctype>
#include <stack>
using namespace std;


enum TokenType {
    BINARY_OP,
    BINARY_COMP,
    DOUBLE_AND,
    DOUBLE_OR,
    OPEN_ACCOLADE,
    CLOSE_ACCOLADE,
    OPEN_CROCHET,
    CLOSE_CROCHET,
    NEGATION,
    COMMA,
    AFFECT,
    POINT_VRG,
    KEY_WORD,
    CONST,
    ADRESS,
    IDENTIF,
    COMMENT,
    NUMBER,
    AUTRE
};
```

Étape 1 : Importation des bibliothèques et définitions globales

Dans cette première étape, nous importons les bibliothèques essentielles pour le programme et définissons les types de jetons (« tokens ») utilisés tout au long du processus de compilation.

Objectifs :

- **Importer les bibliothèques :** Gérer les opérations d'entrée/sortie, manipuler des chaînes, utiliser des conteneurs, etc.
- **Définir les jetons :** Poser les bases pour identifier les différents éléments syntaxiques du code source.

Étape 2 : Classe UniteLexical pour la représentation des tokens

Code source :

```
33  class UniteLexical {
34  private:
35      TokenType token_root;
36      string value;
37
38  public:
39      UniteLexical(TokenType token, const string &val) : token_root(token), value(val) {}
40
41      TokenType getTokenRoot() const {
42          return token_root;
43      }
44
45      string getValue() const {
46          return value;
47      }
48
49      void display() const {
50          cout << "Token: " << token_root << ", Value: " << value << endl;
51      }
52  };
```

Étape 2 : Classe UniteLexical pour la représentation des tokens

La classe **UniteLexical** permet de modéliser chaque unité lexicale avec son type et sa valeur, facilitant ainsi la manipulation des jetons lors des analyses.

Objectifs :

- **Stocker les informations sur les jetons :** Conserver leur type et leur valeur.
- **Faciliter l'affichage :** Offrir une méthode claire pour représenter les jetons en sortie.

Étape 3 : Classe Lexer pour l'analyse lexicale

Code source :

```
56  class Lexer {
57  private:
58      string m_src;
59      size_t index;
60      string buffer;
61      vector<UniteLexical> tokens;
62
63      const string liste_mots[8] = {"else", "then", "for", "while", "void", "int", "return", "if"};
64      const string binOp[4] = {"+", "-", "*", "/"};
65      const string binComp[6] = {">", "<", ">=", "<=", "==", "!="};
66
67  public:
68      Lexer(const string &src) : m_src(src), index(0), buffer("") {}
69
70      char pick() {
71          return (index < m_src.size()) ? m_src[index] : '\0';
72      }
73
74      void consume() {
75          if (index < m_src.size()) {
76              index++;
77          }
```

```
77          }
78      }
79
80      void clearBuffer() {
81          buffer.clear();
82      }
83
84      bool isWhite(char c) {
85          return (c == '\t' || c == '\n' || c == '\r');
86      }
87
88      bool chercher(const string liste[], size_t size, const string &buffer) {
89          for (size_t i = 0; i < size; ++i) {
90              if (buffer == liste[i]) {
91                  return true;
92              }
93          }
94          return false;
95      }
96
```

```
96
97      void handleComment() {
98          consume(); // Skip '/'
99          if (pick() == '/') {
100             consume();
101             while (pick() != '\n' && pick() != '\0') {
102                 buffer += pick();
103                 consume();
104             }
105             tokens.emplace_back(TokenType::COMMENT, buffer);
106             clearBuffer();
107         }
108     }
109
110     vector<UniteLexical> tokenization() {
111         while (index < m_src.size()) {
112             char current = pick();
113
114             while (isspace(current) || isWhite(current)) {
115                 consume();
116                 current = pick();
```

Étape 3 : Classe Lexer pour l'analyse lexicale

```
                consume();
                current = pick();
            }

            if (isdigit(current)) {
                clearBuffer();
                while (isdigit(current)) {
                    buffer += current;
                    consume();
                    current = pick();
                }
                tokens.emplace_back(TokenType::NUMBER, buffer);
            } else if (isalpha(current)) {
                clearBuffer();
                while (isalnum(current)) {
                    buffer += current;
                    consume();
                    current = pick();
                }
                if (chercher(liste_mots, 8, buffer)) {
                    tokens.emplace_back(TokenType::KEY_WORD, buffer);
                } else {
                    tokens.emplace_back(TokenType::IDENTIF, buffer);
```

```
                    tokens.emplace_back(TokenType::IDENTIF, buffer);
                }
            } else if (current == '/') {
                handleComment();
            } else {
                switch (current) {
                    case '{':
                        tokens.emplace_back(TokenType::OPEN_ACCOLADE, "{");
                        break;
                    case '}':
                        tokens.emplace_back(TokenType::CLOSE_ACCOLADE, "}");
                        break;
                    case '[':
                        tokens.emplace_back(TokenType::OPEN_CROCHET, "[");
                        break;
                    case ']':
                        tokens.emplace_back(TokenType::CLOSE_CROCHET, "]");
                        break;
                    case ',':
                        tokens.emplace_back(TokenType::COMMA, ",");
                        break;
                    case ';':
```

```
                        break;
                    case ';':
                        tokens.emplace_back(TokenType::POINT_VRG, ";");
                        break;
                    case '=':
                        consume();
                        if (pick() == '=') {
                            tokens.emplace_back(TokenType::BINARY_COMP, "==");
                            consume();
                        } else {
                            tokens.emplace_back(TokenType::AFFECT, "=");
                        }
                        break;
                    case '&':
                        consume();
                        if (pick() == '&') {
                            tokens.emplace_back(TokenType::DOUBLE_AND, "&&");
                            consume();
                        }
                        break;
                    case '|':
                        consume();
                        if (pick() == '|') {
```

Étape 3 : Classe Lexer pour l’analyse lexicale

```
                            tokens.emplace_back(TokenType::DOUBLE_OR, "||");
                            consume();
                        }
                        break;
                    case '!':
                        consume();
                        if (pick() == '=') {
                            tokens.emplace_back(TokenType::BINARY_COMP, "!=");
                            consume();
                        } else {
                            tokens.emplace_back(TokenType::NEGATION, "!");
                        }
                        break;
                    default:
                        tokens.emplace_back(TokenType::AUTRE, string(1, current));
                        break;
                }
                consume();
            }
        }
        return tokens;
    }
};
```

Le rôle de la classe **Lexer** est de parcourir le code source afin d’extraire les unités lexicales (« tokens »). Elle identifie les mots-clés, les opérateurs, les identifiants, et d'autres éléments syntaxiques.

Étape 3 : Classe Lexer pour l’analyse lexicale

La classe **Lexer** implémente un analyseur lexical dans un compilateur ou un interpréteur. Son rôle est de lire une chaîne de texte source et de la découper en "tokens", qui sont des unités lexicales représentant des éléments du langage source, tels que des mots-clés, des identifiants, des opérateurs, des délimiteurs, etc.

Attributs de la classe :

- **m_src :** Contient le code source à analyser, passé en paramètre au constructeur.
- **index** : Garde la trace de la position actuelle dans le texte source.
- **buffer :** Utilisé pour accumuler des caractères lors de l'analyse des tokens.
- **tokens** : Un vecteur contenant les tokens extraits du code source.
- **liste_mots :** Un tableau de chaînes de caractères représentant les mots-clés du langage.
- **binOp** et bin**Comp** : Ces tableaux contiennent des opérateurs binaires (+, -, *, /) et des opérateurs de comparaison (>, <, >=, <=, ==, !=).

Étape 3 : Classe Lexer pour l’analyse lexicale

Méthodes principales :

- **Lexer(const string &src)** : Constructeur qui initialise le lexer avec le texte source à analyser, ainsi que l'index à 0 et le buffer à une chaîne vide.
- **pick() :** Retourne le caractère à la position actuelle (index) dans le texte source, ou un caractère nul ('\0') si l'on a atteint la fin du texte.
- **consume() :** Avance l'index pour analyser le caractère suivant.
- **clearBuffer() :** Vide le buffer, utile lorsqu'on a fini de lire un token et qu'on doit le préparer pour un autre.
- **isWhite() :** Vérifie si un caractère est un espace, une tabulation, ou une nouvelle ligne, et peut ainsi être ignoré dans l'analyse.
- **chercher()** : Recherche si un buffer donné (un mot ou une chaîne) fait partie d'une liste de chaînes spécifiées (comme les mots-clés ou les opérateurs).
- **handleComment()** : Gère les commentaires dans le code source. S'il rencontre un caractère '/', il vérifie s'il s'agit d'un commentaire (avec '//') et l'ignore jusqu'à la fin de la ligne. Ce commentaire est ensuite ajouté comme un token de type **COMMENT.**
- **tokenization() :** Méthode principale qui effectue l'analyse lexicale. Elle parcourt le texte source et extrait les différents tokens.

Étape 3 : Classe Lexer pour l’analyse lexicale

Les étapes principales de l'analyse lexicale sont :

- **Espaces et commentaires :** Si un espace ou un commentaire est trouvé, il est ignoré.
- **Nombres :** Si un chiffre est trouvé, il est traité comme un token de type **NUMBER.**
- **Identifiants et mots-clés :** Si un caractère alphabétique est trouvé, un mot est accumulé dans le buffer. Si ce mot est un mot-clé (comme int, return, etc.), il est ajouté comme un token de type **KEY_WORD.** Sinon, il est ajouté comme un token d'identifiant (**IDENTIF**).
- Opérateurs et symboles : Si un caractère correspond à un opérateur ou à un symbole (comme {, }, [,], +, =, etc.), il est transformé en un token spécifique. Par exemple, les opérateurs de comparaison comme == ou != sont gérés en tant que token **BINARY_COMP.**

Objectifs :

La méthode **tokenization()** décompose le code source en tokens pour faciliter l'analyse syntaxique. Elle ignore les espaces et les commentaires, mais capture ces derniers séparément. Les nombres et identifiants sont traités comme des tokens distincts pour les différencier facilement. Enfin, les opérateurs arithmétiques et de comparaison sont reconnus comme des tokens essentiels pour les calculs et comparaisons.

Étape 4 : Classe Debugeur pour les vérifications et l'analyse syntaxique

Code source :

```
204
205 class Debugeur {
206 private:
207     vector<UniteLexical> tokens;
208
209     bool checkParenthesesBalance() {
210         stack<char> parentheses;
211         for (const auto &token : tokens) {
212             if (token.getValue() == "(" || token.getValue() == "[") {
213                 parentheses.push(token.getValue()[0]);
214             } else if (token.getValue() == ")" || token.getValue() == "]") {
215                 if (parentheses.empty() || (parentheses.top() == '(' && token.getValue() != ")") ||
216                     (parentheses.top() == '[' && token.getValue() != "]")) {
217                     return false;
218                 }
219                 parentheses.pop();
220             }
221         }
222         return parentheses.empty();
223     }
224
225     void verifierPointsVirgules() {
226         for (size_t i = 0; i < tokens.size(); ++i) {
227             if (tokens[i].getTokenRoot() == TokenType::AFFECT ||
228                 tokens[i].getTokenRoot() == TokenType::IDENTIF ||
229                 tokens[i].getTokenRoot() == TokenType::NUMBER) {
230                 if (i + 1 < tokens.size() && tokens[i + 1].getTokenRoot() != TokenType::POINT_VRG) {
```

```
230                 if (i + 1 < tokens.size() && tokens[i + 1].getTokenRoot() != TokenType::POINT_VRG) {
231                     cout << "Erreur : Point-virgule manquant apres '" << tokens[i].getValue() << "'" << endl;
232                 }
233             }
234         }
235     }
236
237     void verifierTypes() {
238         for (size_t i = 0; i < tokens.size(); ++i) {
239             if (tokens[i].getTokenRoot() == TokenType::AFFECT) {
240                 if (i > 0 && i + 1 < tokens.size()) {
241                     const UniteLexical &gauche = tokens[i - 1];
242                     const UniteLexical &droite = tokens[i + 1];
243
244                     if (gauche.getTokenRoot() == TokenType::IDENTIF && droite.getTokenRoot() != TokenType::NUMBER) {
245                         cout << "Erreur : Type incompatible entre '" << gauche.getValue()
246                              << "' et '" << droite.getValue() << "'" << endl;
247                     }
248                 }
249             }
250         }
251     }
252
253     void verifierParentheses() {
254         int compteurParentheses = 0;
```

```
254         int compteurParentheses = 0;
255         for (const auto &token : tokens) {
256             if (token.getValue() == "(") {
257                 compteurParentheses++;
258             } else if (token.getValue() == ")") {
259                 compteurParentheses--;
260             }
261
262             if (compteurParentheses < 0) {
263                 cout << "Erreur : Parenthèse fermante sans correspondance." << endl;
264                 return;
265             }
266         }
267
268         if (compteurParentheses > 0) {
269             cout << "Erreur : Parenthèse ouvrante sans correspondance." << endl;
270         }
271     }
272
273     void verifierMotsCles() {
274         bool dansFonction = false;
275         for (const auto &token : tokens) {
276             if (token.getValue() == "int" || token.getValue() == "void") {
277                 dansFonction = true;
278             } else if (token.getValue() == "return") {
```

Étape 4 : Classe Debugeur pour les vérifications et l'analyse syntaxique

Code source :

```
            } else if (token.getValue() == "return") {
                if (!dansFonction) {
                    cout << "Erreur : 'return' utilisé en dehors d'une fonction." << endl;
                }
            }
        }
    }

    void verifierCommentaires() {
        bool dansCommentaire = false;
        for (const auto &token : tokens) {
            if (token.getValue() == "/*") {
                dansCommentaire = true;
            } else if (dansCommentaire && token.getValue() == "*/") {
                dansCommentaire = false;
            }
        }

        if (dansCommentaire) {
            cout << "Erreur : commentaire non ferme ." << endl;
        }
    }

    void verifierOperateursInutilises() {
        for (size_t i = 0; i < tokens.size(); ++i) {
```

```
            if (tokens[i].getTokenRoot() == TokenType::BINARY_OP) {
                if (i == 0 || i == tokens.size() - 1 ||
                    (tokens[i - 1].getTokenRoot() == TokenType::BINARY_OP ||
                     tokens[i + 1].getTokenRoot() == TokenType::BINARY_OP)) {
                    cout << "Erreur : Opérateur '" << tokens[i].getValue() << "' mal utilisé." << endl;
                }
            }
        }
    }

public:
    Debugeur(const vector<UniteLexical> &tok) : tokens(tok) {}

    void analyser() {
        if (!checkParenthesesBalance()) {
            cout << "Erreur : desequilibre dans les accolades ou crochets." << endl;
        }
        verifierPointsVirgules();
        verifierTypes();
        verifierParentheses();
        verifierMotsCles();
        verifierCommentaires();
        verifierOperateursInutilises();
    }
};
```

La classe **Debugeur** a pour but principal d'analyser un ensemble de tokens (issus d'un analyseur lexical) pour détecter des erreurs potentielles dans le code source avant son exécution ou compilation. Chaque méthode de cette classe effectue des vérifications spécifiques afin d'assurer la cohérence syntaxique et sémantique du code.

Étape 4 : Classe Debugeur pour les vérifications et l'analyse syntaxique

Objectifs :

Objectif global du débogueur :

Analyser les tokens pour détecter des erreurs syntaxiques et logiques, améliorer la qualité du code et éviter des comportements indésirables à l'exécution.

Objectifs des différentes sections :

Équilibre des accolades et crochets (checkParenthesesBalance) : Vérifie que toutes les accolades ({}, []) ouvrantes ont une correspondance fermante. Cela prévient des erreurs courantes dans la structure du code, comme des blocs mal définis.

Vérification des points-virgules (verifierPointsVirgules) : Assure que les déclarations importantes (affectations, identifiants, nombres) sont correctement terminées par un point-virgule. Cela garantit la lisibilité et la conformité avec les règles syntaxiques.

Validation des types (verifierTypes) : Vérifie que les affectations respectent les types attendus, par exemple, un identifiant ne peut pas être assigné à une valeur incompatible (exemple : int x = "texte";).

Vérification des parenthèses (verifierParentheses) : Détecte les parenthèses ouvrantes ou fermantes non appariées. Cela permet d'éviter des erreurs d'ordre logique dans les expressions mathématiques ou les conditions.

Étape 4 : Classe Debugeur pour les vérifications et l'analyse syntaxique

Vérification des mots-clés (verifierMotsCles) : Vérifie que les mots-clés comme return sont utilisés dans les bons contextes, notamment à l'intérieur des fonctions définies.

Validation des commentaires (verifierCommentaires) : Vérifie que tous les commentaires débutant par /* sont correctement fermés par */. Cela évite que des portions de code soient accidentellement ignorées.

Détection des opérateurs inutilisés (verifierOperateursInutilises) : Identifie les opérateurs binaires mal placés (par exemple, un opérateur sans opérande), ce qui permet d'éviter des erreurs de logique dans les calculs ou conditions.

Étape 5 : Fonction main pour l'interaction utilisateur et la coordination

Code source :

```
329  int main() {
330      // Demande de saisie du code source à l'utilisateur
331      string code;
332      cout << "Veuillez entrer le code source : " << endl;
333      getline(cin, code);
334
335      Lexer lexer(code);
336      vector<UniteLexical> tokens = lexer.tokenization();
337
338      cout << "\n=== Tokens ===" << endl;
339      for (const auto &token : tokens) {
340          token.display();
341      }
342
343      Debugeur debug(tokens);
344      debug.analyser();
345
346      return 0;
347  }
```

Étape 5 : Fonction main pour l'interaction utilisateur et la coordination

---

Cette dernière étape centralise l'exécution du programme. Elle permet de recevoir le code source en entrée, d'exécuter les analyses via les classes précédentes, et d'afficher les résultats.

Objectifs :

- **Recevoir le code source :** L'utilisateur entre le code à analyser.
- **Coordonner les analyses :** Lancer les phases de tokenisation et de débogage.
- **Afficher les résultats :** Permettre à l'utilisateur de voir les jetons extraits et les erreurs détectées.

Conclusion

Ce projet a permis de concevoir et d'implémenter un compilateur simple en C++, en mettant en œuvre un analyseur lexical et syntaxique capable de traiter le code source, d'extraire les tokens et de détecter certaines erreurs courantes. L’analyse lexicale permet de découper le code en unités significatives, tandis que l’analyse syntaxique vérifie la conformité du code aux règles de base du langage. En intégrant des mécanismes de vérification de type et en utilisant des structures de données appropriées, le projet offre une première approche solide de la compilation. Bien qu'encore limité aux aspects lexicaux et syntaxiques, ce projet constitue une excellente introduction aux concepts fondamentaux d'un compilateur, avec de nombreuses pistes d'amélioration possibles pour rendre l'outil plus robuste et fonctionnel.