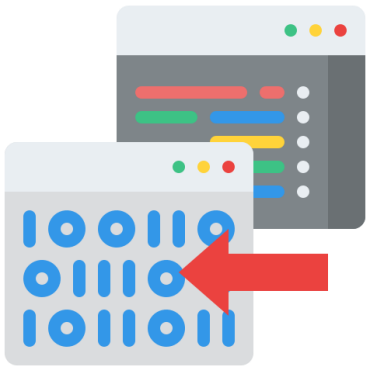
Université Abdelmalek Essaadi

Ecole Nationale des Sciences Appliquées Al-Hoceima

Théorie De Langages Et Compilation

Mini projet compilation

Première Année Génie Informatique



Réalisé par : *AMINE ISSAM*
AMMARA ABDERRAHMANE

Encadré par : Pr. M. Oulhadj

Année Universitaire : 2023/2024

Table des matières

1)Introduction:

Le langage de programmation est un outil puissant pour exprimer des instructions à une machine de manière structurée et compréhensible. Dans le cadre de ce mini-projet de compilation, nous avons étudié la théorie des langages et compilateurs en mettant en pratique les concepts appris à travers le développement d'un compilateur simple pour un langage spécifique.

Ce rapport présente notre travail sur la conception et l'implémentation d'un compilateur pour un langage de programmation simplifié. Nous commençons par définir les éléments du langage en utilisant des expressions régulières et des règles de grammaire. Ensuite, nous décrivons l'implémentation d'une table de symboles pour gérer les variables du programme. Nous abordons également l'implémentation des analyseurs lexical et syntaxique à l'aide de Flex et Bison. Enfin, nous fournissons un compilateur fonctionnel capable de lire un fichier source, d'effectuer l'analyse lexicale et syntaxique, et d'afficher le résultat de l'exécution du programme.

Ce rapport vise à documenter notre compréhension et notre application des concepts de compilation dans un projet concret, tout en mettant en évidence les défis rencontrés et les solutions adoptées.

2) Définitions Et Expressions Régulières :

Pour reconnaître les mots du langage donné, nous devons définir des expressions régulières pour chaque composant du langage. Voici les définitions et expressions régulières correspondantes :

1. Mots-clés :

- reel : reel
- entier : entier
- affiche : affiche
- lire : lire

2. Opérateurs :

- = : =
- + : \+
- : \
- - : \-
- / : /

3. Identificateurs :

- [a-z] : Une lettre minuscule.

4. Nombres :

- Entiers : [0-9]+
- Réels : [0-9]+\.[0-9]+

5. Chaine de caractères :

- "[^"]" : Texte entre guillemets doubles.

6. Séparateurs :

- [\t\n;] : Un espace, une tabulation, une nouvelle ligne ou un point-virgule.

Ces expressions régulières permettent d'identifier chaque élément du langage selon sa syntaxe spécifique. Elles seront utilisées dans un analyseur lexical pour découper le code source

en tokens, qui sont ensuite utilisés dans l'analyse syntaxique pour comprendre la structure du programme.

3) Les Règles De La Grammaire :

Pour définir les règles de la grammaire du langage donné, nous devons spécifier comment les différents éléments du langage peuvent être combinés pour former des constructions valides. Voici les règles de grammaire pour le langage :

1. Programme :

- Un programme est une séquence d'instructions.

Programme -> Instruction*

Cela signifie qu'un programme peut contenir zéro, une ou plusieurs instructions

2. Instruction :

- Une instruction peut être une déclaration, une affectation, un affichage ou une lecture.

Instruction -> Declaration | Affectation | Affichage | Lecture

3. Déclaration :

- Une déclaration commence par les mots-clés **reel** ou **entier**, suivis d'un identificateur et se termine par un point-virgule

Declaration -> 'reel' Identificateur ';' | 'entier' Identificateur ';'

4. Affectation :

- Une affectation commence par un identificateur, suivi du symbole =, suivi d'une expression et se termine par un point-virgule.

```
Affectation -> Identificateur '=' Expression ';'
```

5. Affichage :

- L'affichage commence par le mot-clé affiche, suivi d'une chaîne de caractères entre guillemets doubles ou d'un identificateur et se termine par un point-virgule.

```
Affichage -> 'affiche' (Chaine | Identificateur) ';'
```

6. Lecture :

- Une lecture commence par le mot-clé lire, suivi d'un identificateur et se termine par un point-virgule.

```
Lecture -> 'lire' Identificateur ';'
```

7. Expression :

- Une expression peut être un nombre, un identificateur ou une opération arithmétique entre deux expressions

```
Expression -> Nombre | Identificateur | Expression ('+' | '-' | '*' | '/') Expression
```

8. Identificateur :

- Un identificateur est une lettre minuscule.

```
Identificateur -> [a-z]
```

9. Nombre :

- Un nombre peut être un entier ou un réel.

Nombre -> Entier | Reel

10. **Entier :**

- Un entier est une séquence de chiffres.

Entier -> [0-9]+

11. **Réel :**

- Un réel est une séquence de chiffres suivie d'un point décimal et d'une autre séquence de chiffres.

Reel -> [0-9]+\.[0-9]

12. **Chaine de caractères :**

- Une chaîne de caractères est une séquence de zéro ou plusieurs caractères entre guillemets doubles.

Chaine -> "[^"]"

4)Implémentation De La Table De Symbole :

C'est une implémentation en C++ définissant la structure pour définir une variable , et le pointeur vers la tête qui représente la table des symboles avec les différentes méthodes demandées (ajouter_variable,chercher,modifier).

```cpp
#include <iostream>
#include <string>

using namespace std;

// Structure pour représenter une variable
struct Variable {
    char id; // nom de l'identificateur (variable), un seul caractère
    int type; // 1 pour les entiers, 2 pour les réels
    int intValue; // la valeur de la variable si il s'agit d'un entier
    float floatValue; // la valeur de la variable si il s'agit d'un réel
    Variable* suivant; // pointeur vers l'élément suivant dans la liste
};

// Pointeur vers la tête de la liste chaînée
Variable* tete = nullptr;
```

a) Fonction Ajouter-Variable :

```cpp
// Fonction pour ajouter une variable à la liste
void ajouter_variable(char id, int type, int intValue, float floatValue) {
    Variable* nouvelleVariable = new Variable;
    nouvelleVariable->id = id;
    nouvelleVariable->type = type;
    nouvelleVariable->intValue = intValue;
    nouvelleVariable->floatValue = floatValue;
    nouvelleVariable->suivant = nullptr;

    if (tete == nullptr) {
        tete = nouvelleVariable;
    } else {
        Variable* courant = tete;
        while (courant->suivant != nullptr) {
            courant = courant->suivant;
        }
        courant->suivant = nouvelleVariable;
    }
}
```

b) Fonction Rechercher :

```cpp
// Fonction pour rechercher une variable dans la liste
Variable* rechercher(char id) {
    Variable* courant = tete;
    while (courant != nullptr) {
        if (courant->id == id) {
            return courant;
        }
        courant = courant->suivant;
    }
    return nullptr;
}
```

c) Fonction Modifier :

```cpp
// Fonction pour modifier la valeur d'une variable dans la liste
void modifier(char id, int intValue, float floatValue) {
    Variable* variable = rechercher(id);
    if (variable != nullptr) {
        variable->intValue = intValue;
        variable->floatValue = floatValue;
    } else {
        cout << "Variable non trouvée." << endl;
    }
}
```

5)Implémentation Des Analyseurs Lexical

Ci-dessus est le fichier lexer.l:

lexer.l

```
%{
#include <stdio.h>
#include <stdlib.h> // Pour atoi

#define REEL 1
#define ENTIER 2
#define AFFICHE 3
#define LIRE 4
#define IDENTIFIER 5
#define INTEGER 6
#define REAL 7
#define ASSIGN 8
#define PLUS 9
#define MULTIPLY 10
#define MINUS 11
#define DIVIDE 12
#define SEMICOLON 13
#define STRING_LITERAL 14

extern int yylineno;
void yyerror(const char *s);
%}

%option noyywrap

DIGIT       [0-9]
LETTER      [a-z]

%%

"reel"          { printf("REEL\n"); return REEL; }
"entier"        { printf("ENTIER\n"); return ENTIER; }
"affiche"       { printf("AFFICHE\n"); return AFFICHE; }
"lire"          { printf("LIRE\n"); return LIRE; }
{LETTER}+{DIGIT}*       { printf("IDENTIFIER\n"); return IDENTIFIER; }
{DIGIT}+        { printf("INTEGER: %d\n", atoi(yytext)); return INTEGER; }
{DIGIT}+"."{DIGIT}+     { printf("REAL: %s\n", yytext); return REAL; }
"="             { printf("ASSIGN\n"); return ASSIGN; }
"+"             { printf("PLUS\n"); return PLUS; }
"*"             { printf("MULTIPLY\n"); return MULTIPLY; }
"-"             { printf("MINUS\n"); return MINUS; }
"/"             { printf("DIVIDE\n"); return DIVIDE; }
";"             { printf("SEMICOLON\n"); return SEMICOLON; }
\"[^\"]*\"      { printf("STRING_LITERAL: %s\n", yytext); return STRING_LITERAL; }

[ \t\n]+        { /* skip whitespace */ }
.               { printf("Unknown character: %s\n", yytext); }
%%
```

6)Fournir un compilateur :

Avant de commencer, on créer un fichier main.c où on va:

1. Inclure les bibliothèques nécessaires : Dans votre cas, vous incluez **<stdio.h>** pour la gestion des entrées/sorties, **<stdlib.h>** pour certaines fonctions utilitaires, et éventuellement **<string.h>** si vous avez besoin de manipuler des chaînes de caractères.

2. Déclarer les fonctions **yylex()** et **yyin** : **yylex()** est la fonction générée par Flex qui effectue l'analyse lexicale, et **yyin** est un pointeur de fichier qui sera utilisé pour lire à partir du fichier source.
3. Écrire la fonction **main()** : C'est la fonction principale qui sera exécutée lorsque vous lancez votre programme. Dans cette fonction, vous :
    - Vérifiez que le bon nombre d'arguments est passé en ligne de commande.
    - Ouvrez le fichier source spécifié en argument en lecture.
    - Affectez le descripteur de fichier **yyin** au fichier source.
    - Appelez la fonction **yylex()** pour démarrer l'analyse lexicale.
    - Fermez le fichier source après l'analyse.

En bref, le fichier **main.c** (ou **main.cpp**) coordonne l'exécution de votre analyseur lexical Flex et gère la lecture du fichier source à analyser.

main.c > ...

```c
#include <stdio.h>
#include <stdlib.h> // Pour atoi
#include <string.h> // Pour strcpy

extern int yylex(void);
extern FILE *yyin; // Déclaration de la variable yyin

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <source file>\n", argv[0]);
        return 1;
    }

    // Ouverture du fichier source
    FILE *source = fopen(argv[1], "r");
    if (!source) {
        perror("Error opening source file");
        return 1;
    }

    // Assignation du fichier source à yyin
    yyin = source;

    // Exécution de l'analyseur lexical
    int token;
    while ((token = yylex())) {
        // Do nothing. Printing happens in the rules.
    }

    // Fermeture du fichier source
    fclose(source);

    return 0;
}
```

On ouvre le cmd dans notre dossier où se trouve le fichier lexer.l et le fichier source.txt (où on a collé le code qui se trouve au début de l'exercice) et puis on tappe la commande **« flex lexer.l »**:

1. La commande invoque Flex, un générateur de scanners lexical. Flex prend en entrée un fichier source contenant des règles de tokenization (tel que **lexer.l**) et génère un fichier en C ou en C++ qui constitue l'analyseur lexical correspondant.
2. Dans notre cas, **lexer.l** contient les règles pour l'analyse lexicale du langage spécifié. Flex utilisera ces règles pour

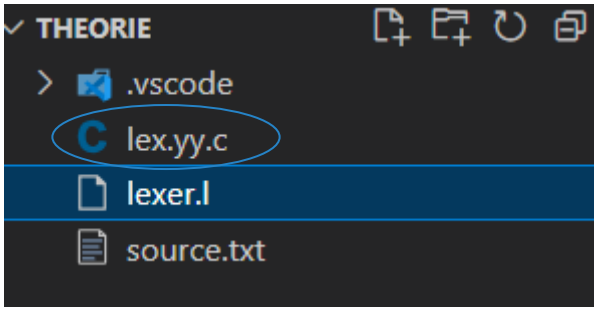
générer un analyseur lexical en C ou en C++ (selon les options utilisées).

3. Le fichier généré par Flex, par défaut nommé **lex.yy.c** pour C ou **lex.yy.cpp** pour C++, contiendra le code source de l'analyseur lexical basé sur les règles définies dans **lexer.l**.

```
C:\Users\boukl\Documents\theorie>flex lexer.l

C:\Users\boukl\Documents\theorie>
```

On remarque que le fichier lex.yy.c est désormais présent :



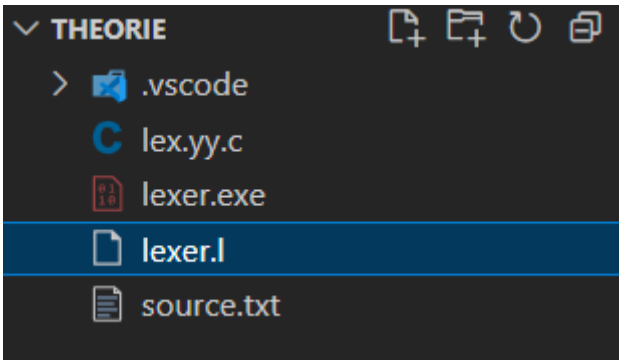
Apres, la commande **« gcc -o lexer main.c lex.yy.c » :**

La commande **gcc -o lexer main.c lex.yy.c** est une commande de compilation utilisée pour compiler votre programme C. Voici ce que chaque partie de la commande fait :

- **gcc**: C'est le compilateur GNU C, utilisé pour compiler des programmes écrits en langage C.
- **-o lexer**: Cette option spécifie le nom du fichier exécutable produit par la compilation. Dans ce cas, le fichier exécutable sera nommé **lexer**.
- **main.c**: C'est le fichier source principal de votre programme. Il contient le code que vous avez écrit pour lancer l'analyseur lexical et lire le fichier source.

- **lex.yy.c**: C'est le fichier généré par Flex contenant le code de l'analyseur lexical. Il contient les règles lexicales que vous avez définies dans votre fichier **lexer.l**, ainsi que d'autres fonctions et déclarations nécessaires pour exécuter l'analyse lexicale.

On remarque la production du fichier exécutable lexer.exe :



Maintenant on tappe la commande **« lexer source.txt »**

```
C:\Users\boukl\Documents\theorie>flex lexer.l

C:\Users\boukl\Documents\theorie>gcc -o lexer main.c lex.yy.c

C:\Users\boukl\Documents\theorie>lexer source.txt
REEL
IDENTIFIER
SEMICOLON
REEL
IDENTIFIER
SEMICOLON
REEL
IDENTIFIER
SEMICOLON
ENTIER
IDENTIFIER
SEMICOLON
ENTIER
IDENTIFIER
SEMICOLON
AFFICHE
Unknown character: Ô
Unknown character: Ç
Unknown character: £
IDENTIFIER
IDENTIFIER
INTEGER: 3
IDENTIFIER
IDENTIFIER
IDENTIFIER
IDENTIFIER
IDENTIFIER
IDENTIFIER
Unknown character: Ô
Unknown character: Ç
Unknown character: Ø
SEMICOLON
IDENTIFIER
ASSIGN
REAL: 3.9
SEMICOLON
IDENTIFIER
ASSIGN
REAL: 8.7
SEMICOLON
IDENTIFIER
ASSIGN
IDENTIFIER
MULTIPLY
IDENTIFIER
SEMICOLON
AFFICHE
Unknown character: Ô
Unknown character: Ç
Unknown character: £
IDENTIFIER
IDENTIFIER
IDENTIFIER
IDENTIFIER
IDENTIFIER
Unknown character: Ô
Unknown character: Ç
Unknown character: Ø
SEMICOLON
AFFICHE
IDENTIFIER
```

Donc comme prévu :

1. Elle exécuterait l'exécutable **lexer** que vous avez compilé précédemment.
2. L'exécutable **lexer** lirait le contenu du fichier **source.txt** passé en argument.
3. L'analyseur lexical intégré dans **lexer** scannerait le contenu de **source.txt** et générerait une séquence de jetons en fonction des règles définies dans votre fichier Flex (**lexer.l**).
4. Les jetons générés seraient imprimés ou affichés à l'écran ou stockés dans un autre fichier, en fonction de la manière dont vous avez programmé votre analyseur lexical.

7)Conclusion:

En conclusion, ce mini-projet de compilation nous a permis de mettre en pratique les concepts théoriques étudiés dans le domaine des langages de programmation et de la compilation. À travers la conception et l'implémentation d'un compilateur pour un langage simplifié, nous avons consolidé nos connaissances sur les étapes du processus de compilation, y compris l'analyse lexicale, syntaxique et sémantique.

Nous avons également acquis une expérience pratique dans l'utilisation d'outils tels que Flex et Bison pour la génération d'analyseurs lexical et syntaxique, ainsi que dans la manipulation des données de la table des symboles pour la gestion des variables du programme.

Ce projet nous a permis de mieux comprendre le fonctionnement interne des langages de programmation et des compilateurs, ce qui nous sera certainement bénéfique dans notre parcours d'études et notre future carrière dans le domaine de l'informatique.