RÉPUBLIQUE DU SÉNÉGAL

UNIVERSITÉ CHEIKH ANTA DIOP DE DAKAR

ÉCOLE SUPÉRIEURE POLYTECHNIQUE

DÉPARTEMENT GÉNIE INFORMATIQUE

LANGAGE C

SUJET :

Analyse et évaluation d'expressions arithmétiques

Présenté

Rawane DIOUF

Mouhamed Lamine FAYE

Adolphe Amadou GALLAND

Cheikh Ahmadou Bamba SALL

heikh Ahmed Tidiane THIANDOUM

Encadrant

Pr Ibrahima Fall

Année universitaire : 2023-2024

Table des matières

Introduction

Le présent rapport détaille le travail effectué dans le cadre du projet de programmation en langage C intitulé "Analyse et évaluation d'expressions arithmétiques". Ce projet a été réalisé dans le cadre du cursus universitaire à l'École Supérieure Polytechnique de l'Université Cheikh Anta Diop de Dakar, pendant l'année universitaire 2023-2024, par le groupe constitué de :

— Rawane Diouf

— Mouhamed Lamine Faye

— Adolphe Amadou Galland

— Cheikh Ahmadou Bamba Sall

— Cheikh Ahmed Tidiane Thiandoum

I Présentation du projet

1 Objectif

L'objectif de ce projet était de développer un programme en langage C capable d'analyser la syntaxe des expressions arithmétiques fournies par l'utilisateur et, si la syntaxe est correcte, d'évaluer leur valeur. Le programme devait également être capable de gérer les erreurs de syntaxe de manière appropriée.

2 Description du Problème

Le problème consiste à écrire un programme qui prend en entrée une expression arithmétique, vérifie sa syntaxe selon une grammaire définie et évalue son résultat si la syntaxe est correcte. Le programme doit pouvoir traiter des opérations d'addition, de

soustraction, de multiplication et de division, ainsi que des parenthèses pour définir la priorité des opérations.

3 Grammaire BNF

La syntaxe des expressions arithmétiques est définie par une grammaire Backus-Naur (BNF), qui comprend des symboles non terminaux (catégories grammaticales), des symboles terminaux (symboles apparaissant tels quels dans les textes analysés), un symbole de départ et des règles de dérivation.

expression	→	*terme opérateur-additif expression* | *terme*
Terme	→	*facteur opérateur-multiplicatif terme* | *facteur*
Facteur	→	*nombre* | '(' *expression* ')'
nombre	→	*chiffre nombre* | *chiffre*
Chiffre	→	'0' | '1' | ... | '9'
opérateur-additif	→	'+' | '−'
opérateur-multiplicatif	→	'∗' | '/'

FIGURE 1 – Grammaire des expressions arithmétiques

II Méthodologie de Développement

Le développement du programme s'est déroulé en trois étapes :

1 Analyse Syntaxique

Dans cette phase, un analyseur syntaxique a été développé pour vérifier si une expression donnée est syntaxiquement correcte en fonction de la grammaire BNF fournie.

Dans le cadre de ce projet, nous avons adopté une approche descendante récursive pour réaliser l'analyse syntaxique des expressions arithmétiques fournies par l'utilisateur.

1.1 Définition des Fonctions

Pour chaque non-terminal de la grammaire BNF, nous avons défini une fonction correspondante dans notre programme.
Par exemple, les prototypes ci-dessous

```
int expression ();
int term ();
int factor ();
int number ();
```

ont été définies pour analyser respectivement les expressions, les termes, les facteurs, et les nombres.

Pour les chiffres, les opérateurs additifs et opérateurs multiplicatifs, nous avons eu recours aux macro-fonctions par des soucis de clarté du code, de réutilisabilité, et de facilité de maintenance.

Les voici :

```
#define is_multiplicative_operator(c) ((c == '*') || (c == '/'))
#define is_additive_operator(c) ((c == '+') || (c == '-'))
#define is_digit(c) ( \
(c == '0') || (c == '1') || (c == '2') || (c == '3') || \
(c == '4') || (c == '5') || (c == '6') || (c == '7') || \
(c == '8') || (c == '9') \
)
#define is_start_factor(c) ((c == '('))
#define is_end_factor(c) ((c == ')'))
```

Ces derniers nous permettent de déterminer respectivement s'il s'agit d'un opérateur

multiplicatif, d'un opérateur additif, d'un chiffre, d'une parenthèse ouvrante et d'une parenthèse fermante.

1.2 Approche Descendante

Nous avons suivi une approche descendante, ce qui signifie que nous avons commencé l'analyse en partant du symbole de départ, qui est expression, et avons procédé récursivement vers le bas en fonction des règles de dérivation de la grammaire BNF.

1.3 Lecture Caractère par Caractère

Notre programme lit les expressions caractère par caractère, en maintenant en mémoire un seul caractère à la fois. Nous avons veillé à ce que la propriété suivante soit toujours vraie : la valeur du caractère lu (`read_character`) correspond au premier caractère non encore examiné.

1.4 Appel de Fonctions Récursives

Chaque fois qu'un non-terminal apparaît dans le membre droit d'une règle de dérivation, notre programme appelle la fonction correspondante pour analyser ce non-terminal. Par exemple, dans la règle expression $\rightarrow$ terme opérateur-additif expression | terme, le programme appelle les fonctions `term()` et `expression()` de manière récursive.

2 Évaluation de l'Expression

Une fois que la syntaxe a été validée, le programme évalue la valeur de l'expression arithmétique en suivant les règles de priorité des opérateurs.

2.1 Lecture de l'expression

L'évaluation commence par la fonction `parser()`, qui lit le premier caractère de l'expression à évaluer à l'aide de `read_next_character()`.

Cette fonction vérifie ensuite si l'expression se termine immédiatement avec un point (`.`) en appelant `is_stop_session_character()`.

2.2 Évaluation de l'expression

Si l'expression ne se termine pas immédiatement, `parser()` appelle `expression()` pour évaluer l'expression arithmétique.

Dans `expression()`, on commence par évaluer le premier terme de l'expression en appelant `term()` et en stockant le résultat dans `result`.

Ensuite, on vérifie si l'expression commence par un opérateur additif (comme + ou -). Si c'est le cas, on crée une pile d'expressions (`Expression_stack`) pour stocker les opérateurs et les termes.

On parcourt ensuite l'expression en itérant sur les opérateurs additifs. Pour chaque opérateur, on lit le prochain caractère, évalue le prochain terme en appelant `term()` et ajoute l'opérateur et le terme à la pile d'expressions.

Après avoir parcouru toute l'expression et stocké les opérateurs et les termes dans la pile, on évalue l'expression en appelant `evaluate_expression()`, qui prend en entrée la pile d'expressions.

`evaluate_expression()` traite la pile d'expressions en dépopant les opérateurs et les termes un par un et en effectuant les opérations correspondantes jusqu'à ce que la pile soit vide. Le résultat final est stocké dans `result`.

2.3 Évaluation des termes

Dans `term()`, on commence par évaluer le premier facteur du terme en appelant `factor()` et en stockant le résultat dans `result`.

Ensuite, on vérifie si le terme contient des opérateurs multiplicatifs (comme * ou /). Si c'est le cas, on crée une pile de termes (`Term_stack`) pour stocker les opérateurs et les facteurs.

On parcourt ensuite le terme en itérant sur les opérateurs multiplicatifs. Pour chaque opérateur, on lit le prochain caractère, évalue le prochain facteur en appelant `factor()` et ajoute l'opérateur et le facteur à la pile de termes.

Après avoir parcouru tout le terme et stocké les opérateurs et les facteurs dans la pile, on évalue le terme en appelant `evaluate_term()`, qui prend en entrée la pile de termes.

`evaluate_term()` traite la pile de termes de la même manière que `evaluate_expression()` traite la pile d'expressions pour calculer le résultat final.

2.4 Évaluation des facteurs

Dans `factor()`, on évalue le facteur en vérifiant d'abord s'il s'agit d'un chiffre en appelant `is_digit()` ou s'il s'agit d'une parenthèse ouvrante. Si c'est un chiffre, on appelle `number()` pour convertir la séquence de chiffres en nombre. Si c'est une parenthèse ouvrante, on évalue l'expression contenue entre les parenthèses en appelant récursivement `expression()`.

Si le caractère lu ne correspond à aucun de ces cas, une erreur est générée en appelant `abort_process()`.

2.5 Conversion des nombres

Dans `number()`, on lit la séquence de caractères représentant le nombre en vérifiant si chaque caractère est un chiffre en appelant `is_digit()`.

Une fois que tous les chiffres sont lus, la séquence de caractères est convertie en nombre à l'aide de `str_to_number()`.

3 Gestion de la session

Du fait que le language C ne dispose pas de gestionnaire d'exception, lors de la rencontre d'erreur d'exécution critique, comme lors du traitement de l'instruction 0/0, le programme est tué. Alors nous avons pensé à la programmation multiprocessus.
Notre algorithme est le suivant : à l'exécution, le processus principal va créer un sous-processus et ce dernier se charge de traiter les instructions (analyses et évaluations des expressions arithmétiques). À la rencontre d'erreur critiques, c'est ce dernier qui sera tué et alors le processus principal décidera s'il faut ou non créer un nouveau sous-processus.

```
while (SESSION) {
    pid_t process_id = fork(); // Crée un nouveau processus
    /*
      Executé par le "pére" en cas d'erreur
      lors de la création d'un nouveau processus
    */
    if (process_id == -1) {
        //TODO: catch fork error
    }
    //Executé par le processus "fils"
    else if (process_id == 0) {
        while (true) {
            printf("A toi :");
            parser(); // Analyse l'entrée de l'utilisateur
            clear_buffer(); // Vide le buffer d'entrée
        }
    }
    //Executé par le processus "pére"
    else {
```

```
        int status;
        waitpid(process_id, &status, 0); // Attend la fin du processus fils
        if (status == 0) {
            SESSION = false; // Si le code d'état est 0, arrête la session
        }
    }
}
```

III Structure du Programme

Le programme développé suit une architecture bien organisée pour faciliter la clarté du code son évolutivité. Cette architecture est répartie dans trois répertoires principaux :

1 *bin* : Les Exécutables

Ce répertoire contient les exécutables générés après la compilation du programme.

2 *includes* : Les Entêtes

Ce répertoire contient les fichiers d'en-tête (.h) nécessaires pour déclarer les prototypes de fonctions, les structures de données et d'autres éléments utilisés dans les fichiers sources.

```c
// syntax_analysis_expression_evaluation.h
#ifndef SYNTAX_ANALYSIS_EXPRESSION_EVAL_H_INCLUDED
#define SYNTAX_ANALYSIS_EXPRESSION_EVAL_H_INCLUDED
#include <stdbool.h>
#include "structs/term_struct.h"
#include "structs/expression_struct.h"

// Prototypes des fonctions
void arithmetic_resolver (); // Résout l'expression arithmétique
void parser (); // Analyse l'expression
int expression (); // Évalue une expression
int term (); // Évalue un terme
int factor (); // Évalue un facteur
int number (); // Évalue un nombre

#endif
```

syntax_analysis_expression_evaluation.h

```c
// term_struct.h
#ifndef TERM_STRUCT_H_INCLUDED
#define TERM_STRUCT_H_INCLUDED
#include <stdlib.h>

// Définition de la structure Term_stack
typedef struct Term_stack {
    int value; // Valeur du terme
    struct Term_stack *next_term; // Pointeur vers le terme suivant dans la pile
    char operator; // Opérateur du terme
} Term_stack ;

// Prototypes des fonctions
int evaluate_term (Term_stack *term_stack); // Évalue un terme
Term_stack *create_term_stack(); // Crée une nouvelle pile de termes
int is_empty_term_stack (Term_stack *term_stack); // Vérifie si la pile de termes est vide
Term_stack *clear_term_stack (Term_stack *term_stack); // Vide la pile de termes
Term_stack *add_term(Term_stack *term_stack, char operator, int value); // Ajoute un terme à la pile

#endif
```

term_struct.h

```c
// expression_struct.h
#ifndef EXPRESSION_STRUCT_H_INCLUDED
#define EXPRESSION_STRUCT_H_INCLUDED
#include <stdlib.h>

// Définition de la structure Expression_stack
typedef struct Expression_stack {
    char operator; // Opérateur de l'expression
    int value; // Valeur de l'expression
    struct Expression_stack *next_expression; // Pointeur vers l'expression suivante dans la pile
} Expression_stack;

// Prototypes des fonctions
int evaluate_expression (Expression_stack *exp); // Évalue une expression
Expression_stack *create_expression_stack(); // Crée une nouvelle pile d'expressions
int is_empty_expression_stack (Expression_stack *exp); // Vérifie si la pile d'expressions est vide
Expression_stack *clear_expression_stack (Expression_stack *exp); // Vide la pile d'expressions
Expression_stack *add_expression(Expression_stack *exp, char operator,  int value); // Ajoute une expression à la pile

#endif
```

expression_struct.h

```c
// session.h
#ifndef SESSION_H_INCLUDED
#define SESSION_H_INCLUDED
#include <stdbool.h>
#include <unistd.h>
#include <sys/wait.h>

extern bool SESSION; // Déclaration de la variable globale SESSION

// Prototypes des fonctions
void start_session(); // Démarre une session
void stop_session(); // Arrête une session
void abort_process(int status_code, char* message); // Termine un processus avec un code d'état et un message

#endif
```

session.h

```c
// utils.h
#ifndef UTILS_H_INCLUDED
#define UTILS_H_INCLUDED
#include <stdio.h>
#include <ctype.h>

extern char read_character; // Déclaration de la variable globale read_character

// Prototypes des fonctions
void clear_buffer (); // Vide le buffer d'entrée
void read_next_character (); // Lit le prochain caractère non-blanc
void init_read_character(); // Initialise la lecture des caractères
void print_error_message (char *prefix, char *message); // Affiche un message d'erreur
void print_result (int value); // Affiche le résultat
int str_to_number (char *str, int len); // Convertit une chaîne de caractères en nombre

#endif
```

utils.h

3 *src* : Les Fichiers Sources

Le répertoire *src* contient tous les fichiers sources du programme.
Tout fichier d'entête dans le répertoire *includes* a son fichier source correspondant où se trouve la définition des fonctions déclarées. De plus, nous avons le fichier *main.c* qui contient la fonction principale.

```c
// main.c
#include "../includes/syntax_analysis_expression_evaluation.h"

// Fonction principale
int main() {
    arithmetic_resolver(); // Appelle la fonction qui résout l'expression arithmétique
    return 0; // Retourne 0 si le programme s'est exécuté correctement
}
```

main.c

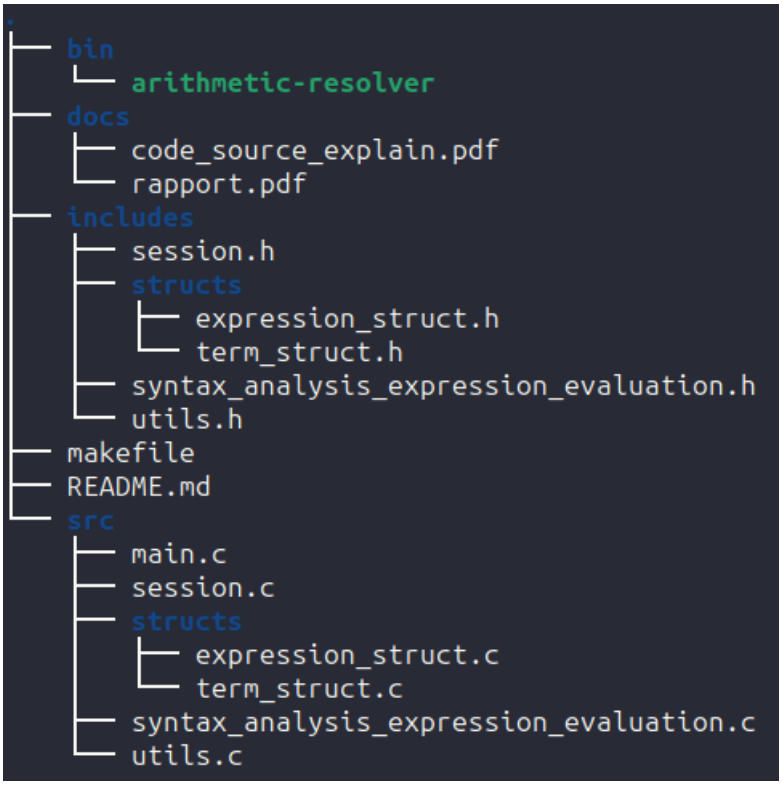

FIGURE 2 – La structure des répertoires du projet sous forme d'arbre

Conclusion

Ce projet de programmation en langage C a permis d'approfondir nos connaissances en matière de développement logiciel, en particulier dans la conception et l'implémentation d'un analyseur syntaxique et d'un évaluateur d'expressions arithmétiques.

Remerciements

Nous tenons à exprimer notre profonde gratitude envers notre enseignant, le Professeur Ibrahima Fall pour sa contribution précieuse à notre formation. Son expertise, son dévouement et son soutien constant ont été des éléments essentiels de notre apprentissage tout au long de ce semestre dans le cours de programmation en langage C.