Table des figures

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

```
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 macros fonctions par des soucis de clarté du code, de réutilisabilité, et de facilité de maintenance.
Macros fonction de reconnaissances des symboles non-terminaux :

```
#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 == ')'))
```

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. [TO COMPLET]

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();
    //[PERE] ERREUR
    if (process_id == -1) {
        //TODO: catch fork error
    }
    //[FILS]
    else if (process_id == 0) {
        while (true) {
            printf("A toi :");
            parser();
            clear_buffer();
        }
    }
    //[PERE]
    else {
        int status;
        waitpid(process_id, &status, 0);
        if (status == 0) {
            SESSION = false;
        }
    }
}
```

III Structure du Programme

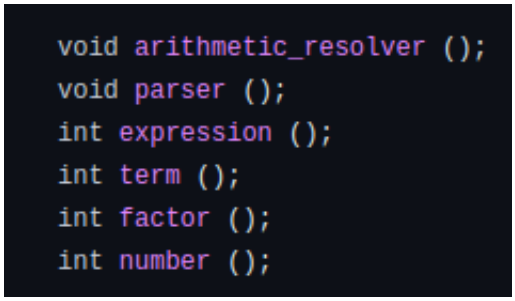
Le programme développé suit une architecture bien organisée pour faciliter la clarté du code son évolutivité. Cette architecture est répartie dans trois répertoires principaux :

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.



```
void arithmetic_resolver ();
void parser ();
int expression ();
int term ();
int factor ();
int number ();
```

`syntax_analysis_expression_evaluation.h`

```c
typedef struct Term_stack {
    int value;
    struct Term_stack *next_term;
    char operator;
} Term_stack ;

int evaluate_term (Term_stack *term_stack);
Term_stack *create_term_stack();
int is_empty_term_stack (Term_stack *term_stack);
Term_stack *clear_term_stack (Term_stack *term_stack);
Term_stack *add_term(Term_stack *term_stack, char operator, int value);
void print_term_stack(Term_stack *term_stack);
```

term_struct.h

```c
typedef struct Expression_stack {
    char operator;
    int value;
    struct Expression_stack *next_expression;
} Expression_stack;

int evaluate_expression (Expression_stack *exp);
Expression_stack *create_expression_stack();
int is_empty_expression_stack (Expression_stack *exp);
Expression_stack *clear_expression_stack (Expression_stack *exp);
Expression_stack *add_expression(Expression_stack *exp, char operator,  int value);
void print_expression_stack(Expression_stack *exp);
```

expression_struct.h

```c
extern bool SESSION;
void start_session();
void stop_session();
void abort_process(int status_code, char* message);
```

session.h

```c
extern char read_character;
void clear_buffer ();
void read_next_character ();
void init_read_character();
void print_error_message (char *prefix, char *message);
void print_result (int value);
int str_to_number (char *str, int len);
```

utils.h

3 src : Les Fichiers Sources

Le répertoire src contient tous les fichiers sources du programme.

```c
int main() {
    arithmetic_resolver();
    return 0;
}
```

main.c

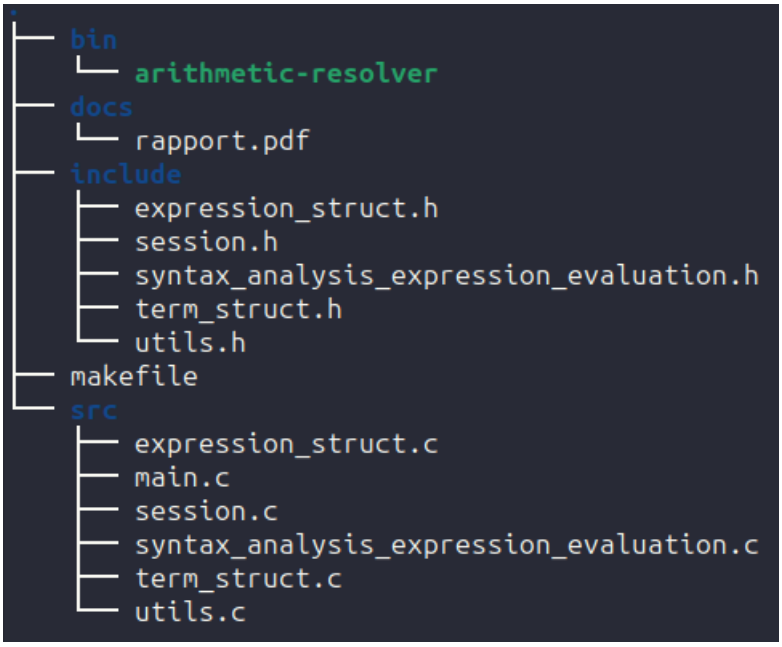


FIGURE 2 – La structure des répertoires du projet sous forme d'arbre

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.