INPT – ENSEEIHT
Département IMA

Anthony Foulfoin
1ère année
Groupe B

PROJET C
Calcul d'expressions arithmétiques

Table des matières

1 Présentation générale

1.1 Présentation de la notation polonaise suffixée

La notation polonaise post fixée permet de s'affranchir des parenthèses en plaçant les opérateurs immédiatement après les opérandes. Cette notation permet d'évaluer une expression arithmétique plus simplement, grâce à l'utilisation d'une pile.

Par exemple, l'expression *a+(b*c)* s'écrira *abc*+*, et l'expression *a-b-c* s'écrira *ab-c-*.

Voir la partie 2.2.2 pour plus d'explications sur l'algorithme de conversion permettant d'obtenir une expression en notation polonaise à partir d'une notation infixe.

1.2 Fournitures

Voici la liste des fichiers fournis pour la réalisation du projet :

- le module *read_item*, composé des fichiers *read_item.h*, *read_item.o* et *type_item.h*. Il permet la lecture au clavier des items composant une expression.

- une partie du module *linear_struct*, l'interface *linear_struct.h* et la définition du type dans *linear_struct.c*, mettant en œuvre une structure de type pile.

1.3 But du projet

Le but de ce projet est de réaliser en langage C, un programme calculant la valeur d'une expression arithmétique, en utilisant la notation polonaise suffixée.

Il faudra pour cela :

- compléter le module *linear_struct ;*
- construire l'application principale en utilisant les modules *read_item* et *linear_struct* ;
- écrire un fichier *makefile* permettant de mettre à jour l'application.

2 Spécifications

2.1 Fonctionnalités du programme

L'interface du programme devra proposer successivement et indéfiniment :

1. La saisie d'une expression arithmétique.

2. La saisie des valeurs des identificateurs présents dans l'expression saisie.

À la suite de cette saisie, le programme affichera la valeur de l'expression calculée en fonction des valeurs des identificateurs.

Dans le cas où une erreur de saisie est détectée lors de la lecture de l'expression, le programme invitera l'utilisateur à la ressaisir.

De même lors de la saisie d'une valeur erronée pour un identificateur, le programme redemandera la saisie de cette valeur.

2.2 Spécifications détaillées

2.2.1 Syntaxe des expressions

Les expressions seront formées des items suivants :

- des constantes entières sur un ou plusieurs caractères ;
- des 4 opérateurs binaires +, -, *, / ;
- des identificateurs que l'on supposera composés d'un seul caractère ;
- des parenthèses.

Les opérateurs sont associatifs à gauche. Les priorités de + et – sont égales mais inférieures à celles de * et /, elles-mêmes de priorités égales.

2.2.2 Algorithme de conversion

L'algorithme de conversion nécessite :

- une pile, vide au départ, pour stocker temporairement certains items ;
- une pile pour ranger l'expression en notation polonaise.

Pour préparer la deuxième phase d'évaluation, on stockera dans une liste les identificateurs de l'expression. Un même identificateur peut figurer plusieurs fois dans une expression.

Au fur et à mesure de la lecture des items, l'algorithme procède de la manière suivante :

- les opérandes (entiers ou identificateurs) sont insérés dans la pile dans leur ordre d'arrivée ;

- les parenthèses ouvrantes sont empilées ;

- une parenthèse fermante entraîne le dépilement de tous les éléments de la pile jusqu'à la rencontre d'une parenthèse ouvrante. Les éléments dépilés sont insérés dans la pile, dans leur ordre de dépilement. La parenthèse ouvrante est retirée de la pile. Si aucune parenthèse ouvrante n'est rencontrée, c'est la preuve que l'expression est mal parenthèsée.

- un opérateur provoque le dépilement, s'il y en a, de tous les opérateurs de priorité supérieure ou égale présents dans la pile jusqu'à une parenthèse ouvrante ou le fond de la pile. Ils sont insérés dans la pile, dans leur ordre de dépilement. Pour simplifier l'algorithme la parenthèse ouvrante peut-être considérée comme un opérateur de priorité plus faible que celle de tous les autres. Ensuite l'opérateur est empilé.

- Après le dernier item lu, la pile est entièrement dépilée, et les items dépilés sont insérés dans la pile, dans leur ordre de dépilement. Dans cette phase la pile ne doit pas contenir de parenthèse ouvrante, sinon c'est la preuve que l'expression était mal parenthèsée.

2.2.3 Évaluation d'une expression polonaise suffixée

À l'issue de la première phase, on dispose d'une pile contenant l'expression en notation polonaise, et d'une liste des identificateurs. La liste va permettre de saisir au clavier les valeurs de ces identificateurs et de les y enregistrer. L'expression polonaise située dans la pile est dans l'ordre inverse de ce qui est nécessaire pour la suite. Il faudra donc la retourner, en l'empilant dans une nouvelle pile. Le calcul de la valeur d'une expression polonaise suffixée, de type entier, nécessite une pile d'entiers pour stocker les résultats intermédiaires. L'algorithme est le suivant :

- On prélève les items de l'expression polonaise stockée préalablement dans la pile.
- On empile les entiers et les valeurs entières des identificateurs.
- Un opérateur binaire s'applique sur les deux derniers éléments empilés, que l'on dépile, et le résultat est empilé.
- À la fin de l'expression, la pile ne contient plus qu'une valeur, qui est le résultat.

Par exemple pour l'expression ab24 + * si a vaut 3 et si b vaut 2, les états successifs de la pile seront [3], [3, 2], [3, 2, 24], [3, 26], [78].

2.3 Contraintes supplémentaires

En plus des contraintes définies par le cahier des charges, plusieurs autres contraintes ont été définies pour la réalisation de ce projet :

- La conception du projet devra se faire par module, chaque module réalisant une tâche qui lui est propre, et pour laquelle ce dernier est spécialisé.

- Chaque fonction devra effectuer des tâches (plus ou moins) élémentaires. Une fonction réalisant plusieurs tâches complexes sera alors divisée en autant de sous-fonctions. La réutilisation de code sera alors optimale, et les tests unitaires plus efficaces.

- L'utilisation des modules définis devra être la plus simple possible, de telle manière que l'application finale n'ait plus qu'à « sous-traiter » son problème aux différents modules compétents.

- Les fonctions devront dans la mesure du possible ne comporter qu'un seul et unique *return*, sauf si la fonction concernée est élémentaire. Les fonctions comporteront ainsi un seul point d'entrée et un seul point de sortie.

- Aucun message, aucun code quel qu'il soit, ne devra être codé « en dur » au sein de l'application. Ceux-ci devront être définis par le biais de constantes. La modification d'une constante entrainera alors la mise à jour de l'ensemble du code source.

3 Conception

3.1 Structure générale de l'application

Afin de satisfaire des soucis de modularité, de clarté, et de réutilisation du code, il a été choisi de développer l'application autour de plusieurs modules, chaque module étant « spécialisé » dans la gestion d'un ensemble de tâches observant un but commun. Au sein de chaque module, chaque tâche fait l'objet d'une fonction qui lui est spécifique. Les tâches trop complexes pour être réalisées en une seule fonction sont divisées en plusieurs sous-fonctions.

3.1.1 Les principaux modules

L'application est divisée 3 principaux modules :

- Le module de lecture des expressions (*read_item*).

- Un module générique *linear_struct* permettant de créer les piles nécessaires aux deux algorithmes.

- L'application principale.

3.1.2 L'application principale

L'application principale est elle-même divisée en 6 modules, chaque module jouant un rôle qui lui est spécifique.

Trois modules principaux :

- Le module *convertisseur*, qui permet de convertir une expression en notation infixe en une expression polonaise suffixée.

- Le module *calcul_arith*, qui permet d'évaluer une expression arithmétique en notation polonaise.

- Le « pseudo-module » *main* qui « assemble les briques » en utilisant les différents modules définis, afin de mettre au point l'application finale.

Trois modules « annexes » utilisés comme outils afin de mettre en œuvre les trois modules précédents :

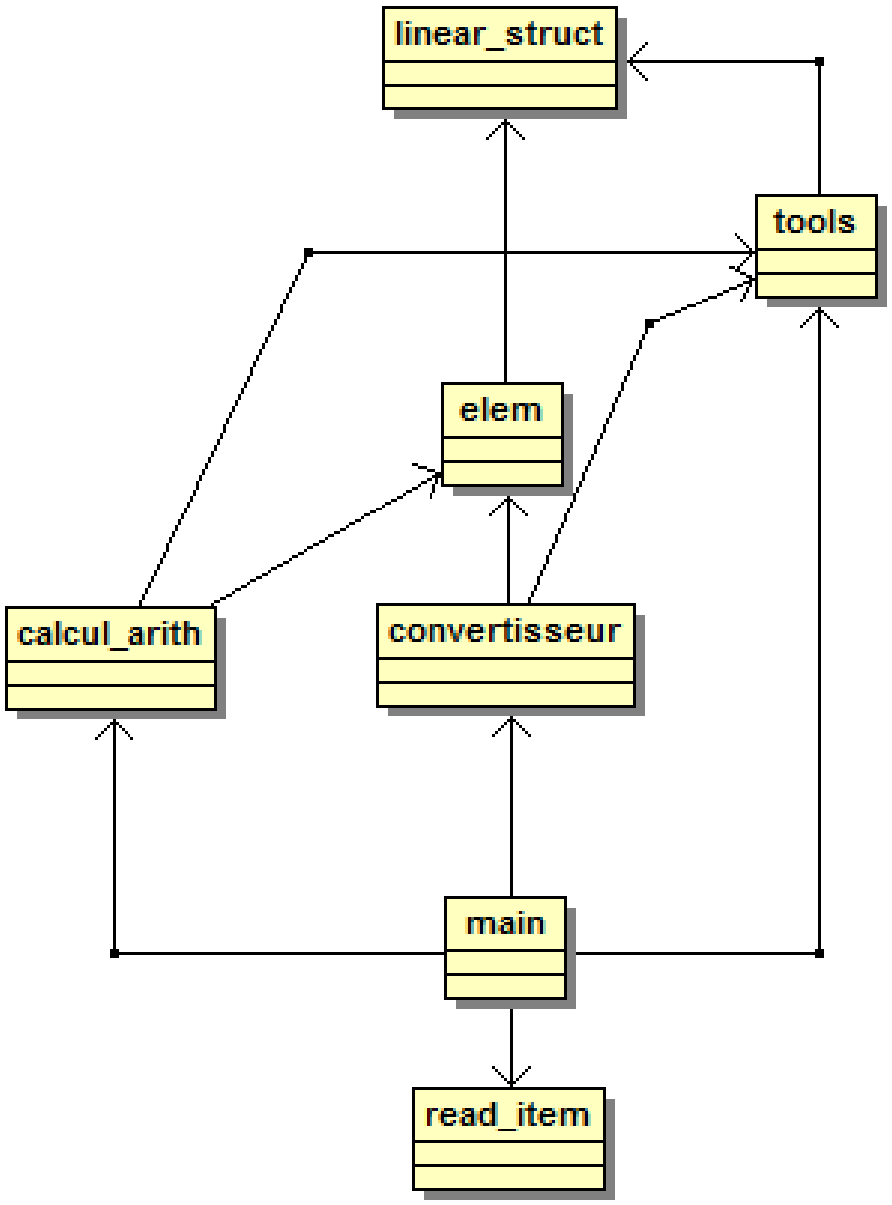
- Le module *elem* qui définit la structure manipulée par *convertisseur* et *calcul_arith* afin de mémoriser des informations dans les piles (*linear_struct*), ainsi que toutes les méthodes nécessaires à la manipulation de ces structures. La structure définie est ainsi destinée à être stockée dans les *linear_struct*.

- Le module *message* qui définit tous les messages et codes d'erreurs utilisés par l'ensemble des modules, ainsi qu'une méthode qui leur permet d'afficher un message d'erreur à l'utilisateur.

- Le module *tool* qui définit des outils supplémentaires mis à la disposition des autres modules, par exemple une méthode qui permet d'obtenir un entier de la part de l'utilisateur.

3.1.3 Structure générale

Voici un schéma représentant les différents modules de l'application et les principaux liens de dépendance qui les unissent.



3.2 Présentation des modules de l'application

Dans toutes les fonctions qui suivent, *pile1* est une pile temporaire servant à stocker des opérateurs, *pile2* est la pile dans laquelle figurera l'expression en notation polonaise à la fin du traitement. Pour plus de détails, voir la partie 2.2.2.

3.2.1 Le module read_item

Le module *read_item* fait partie des fournitures du projet. Celui-ci est composé des fichiers *read_item.o, read_item.h* et *type_item.h*.

3.2.1.1 Read_item.o

Ce fichier met à disposition une méthode *enum type_item read_item(char *op_ou_ident, int *valeur)* qui lit une expression au clavier et renvoie ses items un par un en les plaçant dans les variables *op_ou_ident* ou *valeur* selon qu'il s'agisse d'un caractère ou d'un entier. La fonction renvoie également un code indiquant le type de l'item lu.

3.2.1.2 Type_item.h

Les codes renvoyés par la fonction *read_item* sont définis dans le fichier *type_item.h*. Les différents codes possibles sont :

- *eol*, qui indique que tous les items de l'expression ont été lus ;
- *error*, qui indique qu'une erreur s'est produite pendant la lecture de l'item ;
- *operateur*, qui indique que l'item renvoyé représente un opérateur ;
- *entier*, qui indique que l'item renvoyé est un entier ;
- *ident*, qui indique que l'item renvoyé est un identificateur.

3.2.2 Le module linear_struct

Le module *linear_struct* fait partie des fournitures du projet. Celui-ci est composé des fichiers *linear_struct.c* et *linear_struct.h.* Le fichier *linear_struct.c* contient la définition du type associé aux piles, mais doit être complété suivant l'interface *linear_struct.c.*

3.2.2.1 Les types et constantes définis

Le fichier *linear_struct* définit une structure de type liste-chainée, utilisée ici en tant que pile. Cette structure est définie dans *linear_struct.h* et implémentée dans *linear_struct.*c.

Voici la définition de ce type :

```
typedef struct _chainon{
        ELT valeur;
        struct _chainon *suivant;
};*Linear_Struct;
```

Ce module définit également un type *Code_retour*, utilisé par certaines fonctions du module comme valeur de retour. Voici la définition de ce type :

```
typedef enum { OK, KO } Code_retour;
```

Enfin, ce module définit une constante *ELT*, dont la valeur est *void* *.

3.2.2.2 Les fonctions définies

Voici la liste de toutes les fonctions définies par le module. Celles-ci permettent de manipuler des structures de type *linear_struct* :

- *Code_retour tester_vide(Linear_Struct q)*. Cette fonction teste si une pile est vide ou non, et renvoie OK si la pile est vide, KO sinon.

- *Linear_Struct creer()*. Cette fonction permet de créer une pile vide.

- *void afficher(Linear_Struct q, void (*print_elem)(ELT e)).* Cette fonction permet d'afficher les éléments contenus dans la pile q.

- *ELT chercher(Linear_Struct q, ELT e, Code_retour (*cmp)(ELT e1,ELT e2)).* Cette fonction cherche et renvoie le premier élément d'une *linear_struct* égal à un élément donné 'e'.

- *Code_retour empiler(Linear_Struct *q, ELT e).* Permet d'empiler un élément 'e' dans la pile passée en paramètre.

- *Code_retour depiler(Linear_Struct *q, ELT *e)*. Permet de dépiler un élément 'e' de la pile passée en paramètre.

- *Code_retour sommet(Linear_Struct q, ELT *e)*. Permet de consulter la valeur du sommet de la pile passée en paramètre.

3.2.3 Le module elem

Ce module est un module « annexe » qui à été défini pour les besoins des modules *conversion* et *calcul_arith*. Celui-ci définit une structure *Elem* destinée à être utilisée comme élément à empiler dans les *linear_struct* utilisés par les deux modules précédemment cités. Des fonctions permettant de manipuler ces structures sont également fournies.

3.2.3.1 Les types et constantes définis

Voici l'implémentation de la structure définie par ce module :

```
typedef struct _elem{
    char valChar;
    int valInt;
} *Elem;
```

Cette structure permet ainsi de stocker un entier et/ou un caractère. Cependant, les modules *conversion* et *calcul_arith* ne doivent souvent stocker qu'un seul élément dans cette structure, et non deux. Il est ainsi nécessaire de mémoriser cette information.

Pour cela, le module *elem* définit une constante *NUM*, dont la valeur est '\0'. Si l'utilisateur de la structure ne souhaite mémoriser qu'un entier, il a alors la possibilité de renseigner le caractère avec la valeur NUM. Ainsi, si le caractère est égal à NUM, cela signifie que l'élément stocké est un entier, sinon il s'agit d'un caractère.

3.2.3.1 Les fonctions définies

Toutes les fonctions suivantes permettent de manipuler des structures de type *Elem* :

- *Elem creerElem().* Permet de créer une nouvelle structure.
- *char getChar(Elem e).* Permet d'obtenir l'attribut de type caractère de la structure.
- *int getInt(Elem e).* Permet d'obtenir l'attribut entier de la structure.
- *void setInt(Elem e, int i).* Permet de fixer l'attribut entier de la structure.
- *void setChar(Elem e, char i).* Permet de fixer l'attribut de type caractère de la structure.

- *void print_elem(ELT i).* Permet d'afficher le contenu d'une structure.

- *Code_retour comparer_char(ELT elem, ELT car).* Permet de comparer une structure (premier argument) avec un caractère (*char* *) passé en paramètre (second argument).

- *int estOperateur(Elem e)*. Renvoie 1 si *e* représente un opérateur, 0 sinon.

- *int estIdentificateur(Elem e)*. Renvoie1 si *e* représente un identificateur, 0 sinon.

3.2.4 Le module tools

Ce module est un module « annexe » qui à été défini principalement pour les besoins des modules *conversion* et *calcul_arith*. Celui-ci définit des fonctions supplémentaires mises à disposition des autres modules.

3.2.4.1 Les fonctions définies

- *int getInEntier().* Cette fonction permet d'obtenir un entier de la part de l'utilisateur. Elle boucle sur la lecture du clavier tant que l'utilisateur ne saisit pas une valeur correcte (un entier).

- *Code_Erreur transferer(Linear_Struct *pile1, Linear_Struct *pile2).* Cette fonction permet de transférer le contenu de la pile1 dans la pile2.

- *Code_Erreur creer_pile_expr(int valInt, char valChar, enum type_item codeRet, Linear_Struct *expr).* Cette méthode est mise à la disposition du développeur qui utilise le module *conversion*. Le module *conversion* travaille effectivement à partir d'une pile contenant l'intégralité de l'expression à convertir. Or le module *read_item* ne retournant les items qu'un par un, *creer_pile_expr* permet alors de créer une pile représentant l'expression dans son intégralité. Elle doit être appelée pour chaque nouvel item à traiter. Celle-ci gère automatiquement les erreurs éventuellement retournées par la fonction *read_item.*

3.2.5 Le module messages

Ce module est un module « annexe » qui à été défini pour les besoins des modules *conversion* et *calcul_arith*. Celui-ci se compose d'un fichier *messages.h* qui définit un ensemble de messages et de codes d'erreurs, et d'un fichier *message.c* qui implémente une fonction permettant l'affichage de messages d'erreur.

L'intérêt de ce module est de centraliser au sein d'un même fichier tous les messages et codes d'erreur utilisés au sein du programme. Ainsi, si l'on souhaite traduire le programme dans une langue étrangère, ou modifier un message en particulier, il ne suffit de modifier que les constantes présentes dans ce fichier (et il est donc inutile de parcourir tout le code source à la recherche de ces messages).

De plus, considérant qu'un développeur externe souhaitant utiliser les modules *conversion* et *calcul_arith* définis dans ce projet puisse désirer que ceux-ci ne produisent aucun affichage (mais retournent simplement un code en cas d'erreur), ce module met à disposition une fonction *afficher_message* qui doit être utilisée obligatoirement par tous les modules du projet (en dehors du fichier *main.c,* qui lui n'a pas pour vocation d'être utilisé dans la réalisation d'un autre programme) pour afficher un message d'erreur.

Une constante *AFFICHAGE_MESSAGES* permet d'indiquer si l'on souhaite ou non que la fonction *afficher_message* produise un affichage, il est ainsi possible de désactiver tout affichage des modules du projet.

3.2.5.1 Les types et constantes définis

Plusieurs constantes sont définies dans ce module :

- La constante *AFFICHAGE_MESSAGES*. Si celle-ci est fixée à 1, alors la fonction *afficher_message* affiche à l'écran le message qu'elle reçoit en paramètre ; si la constante est fixée à 0, la fonction *afficher_message* ignorera tous les messages qu'on lui demande d'afficher.

- Les constantes dont le nom est *MSG_ERREUR_XX* où XX représente un nombre entier, qui représentent les messages d'erreurs utilisés par les modules du projet.

- Les constantes dont le nom est MESSAGE_X où X est une chaine de caractère, qui représentent des messages quelconques utilisés par les modules du projet.

Un type spécifique à été défini pour les codes d'erreur à retourner par les fonctions du projet. Toutes les fonctions susceptibles de conduire à un échec doivent renvoyer un code définis par ce type :

typedef enum code_erreur Code_Erreur

Chaque code d'erreur se présente sous la forme CDE_ERREUR_XX où XX représente un entier. Les codes d'erreurs sont associés aux messages d'erreurs. Ainsi, si une fonction détecte une erreur menant à produire l'affichage de l'erreur *MSG_ERREUR_02,* la fonction renverra automatiquement le code d'erreur qui lui est associé, c'est-à-dire *CDE_ERREUR_02*.

3.2.5.2 Les fonctions définies

- *void afficher_message(char *message).* Affiche à l'écran le message passé en paramètre si *AFFICHAGE_MESSAGES* est fixé à 1. N'affiche rien sinon.

3.2.6 Le module convertisseur

Il s'agit de l'un des deux modules principaux du projet. Celui-ci permet de convertir une expression arithmétique infixe en une expression arithmétique polonaise post-fixée. Celui-ci à été conçu afin d'offrir à l'utilisateur du module une grande simplicité d'utilisation.

3.2.6.1 Les fonctions définies

- *Code_Erreur conversion(Linear_Struct *expr, Linear_Struct *ident).*
  Il s'agit de la fonction « d'entrée » du module », c'est elle que le développeur doit appeler pour traiter l'expression courante. Celle-ci reçoit comme uniques paramètres une pile *expr* qui contient l'expression sous forme infixe à convertir, et une pile destinée à recevoir la liste des identificateurs présents dans l'expression. Les items de l'expression sont traités un par un. L'expression une fois convertie, est placée dans *expr*.

- *Code_Erreur verif_syntaxe(Linear_Struct expr).* Il s'agit d'une fonction très importante dont l'objectif est de vérifier que l'expression arithmétique possède une syntaxe correcte. L'expression arithmétique passée en paramètre doit être sous forme infixe. Cette fonction gère un grand nombre de cas d'erreurs possibles.

- *Code_Erreur traiterItemCar(Elem element, Linear_Struct *pile1, Linear_Struct *pile2, Linear_Struct *identif).* Cette sous-fonction permet de gérer le cas où l'item à traiter est un caractère.

- *Code_Erreur depiler_op(Linear_Struct *pile1,Linear_Struct *pile2, char op).* Cette fonction permet de gérer le dépilement de la pile temporaire dans le cas où l'item à traiter est un opérateur, ou une parenthèse fermante.

- *Code_Erreur ajouter_ident(Linear_Struct *ident, char i).* Cette fonction permet d'ajouter un identificateur à la liste des identificateurs.

- *Code_Erreur ajouter_elem(Linear_Struct *pile, int valInt, char valChar)*. Cette fonction permet d'ajouter un élément de type *Elem*, construit à partir de l'entier et du caractère passés en paramètre, à la *Linear_Struct* passée en paramètre.

- *Code_Erreur fin(Linear_Struct *pile1,Linear_Struct *pile2).* Cette sous-fonction est appelée lorsque tous les items de l'expression ont été traités. Elle transfert le contenu de la pile temporaire *pile1* dans *pile2*, et inverse *pile2* pour retourner l'expression dans son sens de lecture.

3.2.1 Le module calcul_arith

Il s'agit de l'un des deux modules principaux du projet. Celui-ci permet d'évaluer une expression arithmétique en notation polonaise post-fixée. Celui-ci à été conçu afin d'offrir à l'utilisateur du module une grande simplicité d'utilisation.

3.2.1.1 Les fonctions définies

- *Code_Erreur calcul_arith(Linear_Struct expr, Linear_Struct ident, int *result).* Principale fonction du module. C'est elle qui doit être appelée pour évaluer une expression arithmétique en notation polonaise.

- *Code_Erreur calculer(Linear_Struct *calcul, char operateur).* Il s'agit de la fonction qui effectue les calculs. Celle-ci applique l'opérateur sur les deux derniers éléments de la pile passée en paramètre. Elle vérifie également que l'opération effectuée est correcte.

- *Code_Erreur remplacer_ident(Linear_Struct ident, Linear_Struct *expr).* Cette fonction permet de remplacer les identificateurs de l'expression par leur valeur entière, après en avoir demandé la valeur auprès de l'utilisateur.

3.2.2 Le module main

Il ne s'agit pas d'un module à proprement parler, mais d'un unique fichier *main.c*. Celui-ci à pour seul objectif d'assembler les différents modules précédemment conçus afin de créer l'application finale.

Etant donné la conception par module de l'application, le code de la méthode *main* est relativement simple, puisqu'il consiste essentiellement à appeler les méthodes *read_item, conversion, et calcul_arith* définis précédemment. Enfin, il ne lui reste plus qu'à gérer les erreurs retournées par ces modules (par défaut, les modules gèrent les erreurs en interne en produisant un affichage, et en stoppant le traitement en cours, ce qui simplifie encore le *main*, voir la partie 3.3) et à afficher le résultat.

3.3 Gestion des erreurs

La gestion des erreurs au sein du programme à fait l'objet d'une attention particulière. Chaque module étant divisé en un ensemble de fonctions, chacune de ces fonctions s'assure d'effectuer un traitement correcte, si ses paramètres sont correctes.

En cas d'erreur détectée, la fonction à l'origine de cette dernière demande auprès de la fonction *afficher_message* du module *messages* de produire un affichage de l'erreur afin d'en informer l'utilisateur.

Ensuite, cette fonction stoppe le traitement en cours d'exécution et renvoie un code d'erreur indiquant la raison de l'interruption auprès de la fonction appelante, qui interrompra à son tour tout traitement en cours pour propager l'erreur, et ainsi de suite, jusqu'à ce que l'erreur arrive auprès d'une méthode appelante définie en dehors du module (dans le *main*).

4 Programmation

Cette partie vise à présenter le fonctionnement général des principales fonctions du programme. Pour obtenir plus de détails, le code source du programme présente un nombre suffisant de commentaires visant à faciliter leur compréhension.

Par ailleurs les cas d'erreurs pour chacune de ses fonctions pouvant être nombreux, ceux-ci n'ont pas été détaillés dans cette partie. Nous supposerons donc ici que le programme s'exécute sans rencontrer la moindre erreur, en présentant son fonctionnement général.

Dans toutes les fonctions qui suivent, *pile1* est une pile temporaire servant à stocker des opérateurs, *pile2* est la pile dans laquelle figurera l'expression en notation polonaise à la fin du traitement. Pour plus de détails, voir la partie 2.2.2.

4.1 Module conversion

4.1.1 Fonction conversion

Prototype de la fonction :
*Code_Erreur conversion(Linear_Struct *expr, Linear_Struct *ident)*

Principe :
Convertit l'expression infixe reçue en paramètre (*expr*) en expression post-fixée. La fonction retourne également (*ident*) la liste des identificateurs utilisés dans l'expression.

Fonctionnement général de la fonction :
La fonction commence par créer deux piles *pile1* et *pile2*.

Puis, celle-ci appelle la sous-fonction *verif_syntaxe* qui vérifie si la syntaxe de l'expression passée en paramètre est correcte.

La fonction traite alors les items de l'expression un par un. Pour chaque item, celle-ci ajoute directement l'item dans *pile2* s'il s'agit d'un entier, et fait appelle à la sous-fonction *traiterItemCar* s'il s'agit d'un caractère

4.1.2 Fonction verif_syntaxe

Prototype de la fonction :
Code_Erreur verif_syntaxe(Linear_Struct expr)

Principe :
Vérifie que la syntaxe de l'expression arithmétique infixe passée en paramètre est correcte, et renvoie une erreur dans le cas contraire. L'imbrication des parenthèses est vérifiée par les méthodes *depiler_op* et *fin*.

Fonctionnement général de la fonction :
Pour chaque élément d'indice i au sein de la pile *expr*, la fonction effectue plusieurs vérifications. Chaque condition non remplie génère une erreur :

S'il s'agit d'un opérateur, alors celui-ci ne doit pas être placé au début ou à la fin de l'expression. L'élément i+1 doit être soit un nombre, soit une parenthèse ouvrante. L'élément i-1 quand à lui doit être une parenthèse fermante ou un nombre.

S'il s'agit d'une parenthèse ouvrante, celle-ci ne doit pas être placée à la fin de l'expression. L'élément i+1 doit être un nombre ou une parenthèse ouvrante. L'élément i-1 doit être un opérateur ou une parenthèse ouvrante.

S'il s'agit d'une parenthèse fermante, celle-ci ne doit pas être placée au début de l'expression. L'élément i+1 doit être un opérateur ou une parenthèse fermante. L'élément i-1 doit être un nombre ou une parenthèse fermante.

Enfin, s'il s'agit d'un identificateur ou d'un nombre, celui-ci ne doit pas être précédé ou suivi par un identificateur ou un nombre.

4.1.3 Fonction traiterItemCar

Prototype de la fonction :
*Code_Erreur traiterItemCar(Elem element, Linear_Struct *pile1, Linear_Struct *pile2, Linear_Struct *identif)*

Principe :
Permet de traiter un item (*element*) de type caractère selon l'algorithme donné en 2.2.2.

Fonctionnement général de la fonction :
Si l'item à traiter est un identificateur, alors la fonction empile l'identificateur dans *pile2* et dans *identif*, pile qui contient la liste de tous les identificateurs.

S'il s'agit d'un opérateur, alors la fonction *traiterItemCar* appelle la sous fonction *depiler_op* qui gère les dépilements/empilements des opérateurs (entre les 2 piles *pile1* et *pile2),* selon leur priorité par rapport à l'opérateur courant. Puis, l'opérateur courant est empilé dans *pile1*.
Les parenthèses ouvrantes sont quand elles directement empilées dans *pile1*.

Lorsqu'une parenthèse fermante est rencontrée, la fonction *traiterItemCar* appelle alors la sous-fonction *depiler_op*, qui gèrera les transferts d'opérateurs entres les 2 piles. Les parenthèses fermantes ne sont jamais empilées.

4.1.4 Fonction depiler_op

Prototype de la fonction :
*Code_Erreur depiler_op(Linear_Struct *pile1,Linear_Struct *pile2, char op)*

Principe :
Gère les dépilement/ré-empilement d'opérateurs entre les piles *pile1* et *pile2* selon l'algorithme donné en 2.2.2. L'opérateur provocant un éventuel dépilement est passé en paramètres (*op*).

Fonctionnement général de la fonction :
Si la fonction doit traiter un opérateur, alors celle-ci dépile tous les opérateurs de *pile1* de priorité supérieure ou égale à *op*, pour les empiler dans *pile2* (dans leur ordre de dépilement).

Si la fonction doit traiter une parenthèse fermante, celle-ci dépile alors tous les opérateurs de *pile1* jusqu'à rencontrer une parenthèse ouvrante. La parenthèse ouvrante est alors elle-même dépilée, mais celle-ci n'est pas transférée dans *pile2*. La fonction vérifie par ailleurs que le parenthèsage de l'expression est correcte.

4.2 Module calcul_arith

4.2.1 Fonction calcul_arith

Prototype de la fonction :
*Code_Erreur calcul_arith(Linear_Struct expr, Linear_Struct ident, int *result)*

Principe :
Permet d'evaluer une expression arithmetique representee en notation polonaise postfixee (*expr*) en utilisant notamment une pile indiquant tous les identificateurs présents dans l'expression (*ident*). Le résultat est placé dans *result*.

Fonctionnement général de la fonction :
La fonction commence par appeler la sous-fonction *remplacer_ident* qui s'assure de remplacer tous les identificateurs de l'expression par une valeur entière, saisie par l'utilisateur.

Puis, la fonction dépile les élements de *expr,* et transfert tous les opérandes dans une pile temporaire *calcul*. Lorsque *calcul_arith* rencontre un opérateur, la fonction appelle alors la sous-fonction *calculer* qui applique alors l'opérateur sur les deux derniers opérandes de *calcul*, et ré-empile le résultat dans *calcul*.

Une fois tous les élements de *expr* dépilés et traités, le résultat du calcul se trouve au fond de la pile *calcul*. Ce résultat est alors placé dans la variable *result* dont un pointeur est passé en paramètre.

4.2.2 Fonction remplacer_ident

Prototype de la fonction :
*Code_Erreur remplacer_ident(Linear_Struct ident, Linear_Struct *expr)*

Principe :
Permet de remplacer tous les identificateurs de *expr* par une valeur entière saisie par l'utilisateur. Les identificateurs présents dans l'expression sont stockés dans la pile *ident*.

Fonctionnement général de la fonction :
La fonction parcourt entièrement la pile *ident* qui contient la liste des identificateurs présents dans l'expression. Pour chacun de ses identificateurs, la fonction va :

- Demander à l'utilisateur de saisir la valeur correspondante, via la méthode *getInEntier* définie dans le module *tools* ;

- Rechercher toutes les occurrences de cet identificateur présentes dans *expr* grâce à la fonction *rechercher* définie dans le module *linear_struct*, et les remplacer par la valeur saisie par l'utilisateur.

4.2.3 Fonction calculer

Prototype de la fonction :
*Code_Erreur calculer(Linear_Struct *calcul, char operateur)*

Fonctionnement général de la fonction :
La fonction dépile les deux derniers opérandes de *calcul* et applique l'opérateur passé en paramètre sur ceux-ci. Le résultat est alors empilé dans *calcul*.

La fonction vérifie également que l'opération effectuée est correcte, c'est-à-dire qu'elle est de la forme *operande operateur operande*. Tous les autres cas sont rejetés.

4.3 Module tools

4.3.1 Fonction creer_pile_expr

Prototype de la fonction :
*Code_Erreur creer_pile_expr(int valInt, char valChar, enum type_item codeRet, Linear_Struct *expr)*

Principe :
Permet de créer une pile représentant une expression arithmétique (*expr*) à partir des éléments retournés par la fonction *read_item* qu'elle reçoit en paramètres.

Fonctionnement général de la fonction :
La fonction analyse la valeur de *codeRet*, indiquant quel est le type de l'item en court de traitement.

Les entiers, opérateurs et identificateurs sont automatiquements empilés dans *expr*.

Lorsque *codeRet* signale une erreur, *creer_pile_expr* lève elle-même une erreur.

Enfin, lorsque *codeRet* signal que tous les items ont été traités, *creer_pile_expr* inverse la pile *expr* afin que l'expression dans la pile soit présentée dans son sens de lecture.

5 Tests

Afin de s'assurer du bon fonctionnement du programme dans son ensemble, il est nécessaire de le tester avec des arguments significatifs. Des tests unitaires ont été effectués pour chaque fonction non élémentaire, ainsi que des tests impliquant l'application dans son ensemble.

5.1 Tests unitaires

L'algorithme étant composé d'un ensemble de fonctions, il est tout d'abord impératif de s'assurer que chacune des fonctions déclarées fonctionne correctement.

Ainsi, des tests unitaires ont été réalisés pour toutes les fonctions non élémentaires.

Les jeux de tests se doivent d'être le plus significatif possible. Chaque jeu de test se doit de vérifier que toutes les instructions de la fonction testée soient exécutées. Ainsi, si une fonction contient des structures conditionnelles (if then else), le jeu de test doit s'assurer que chaque bloc d'instruction de la condition est exécuté au moins une fois (le then et le else) par le jeu de tests.

En respectant ces impératifs, les jeux de tests s'assurent ainsi de couvrir un grand nombre de possibilités.

Pour mettre en place ces tests, une méthode *main* à été mise en place dans chaque fichier .c afin d'effectuer les tests de toutes les fonctions non élémentaires.

5.2 Tests globaux

Une fois les tests unitaires passés avec succès, il est important de tester le programme dans sa globalité.

Le jeu de test global s'assure ainsi de représenter le nombre maximum de cas susceptibles de poser problème, afin de vérifier si l'erreur est détectée ou non par le programme, et quelle est sa réaction face à cette erreur.

5.2.1 Jeu de test global

Tests portant sur le parenthèsage de l'expression :

Tests qui doivent générer une erreur :

Test : (
Résultat : *Attention, votre expression est mal parenthesée*

Test :)
Résultat : *Attention, votre expression est mal parenthesée*

Test : ()
Résultat : *Attention, votre expression est mal parenthesée*

Test : (1+2
Résultat : *Attention, votre expression est mal parenthesée*

Test : 1+2)
Résultat : *Attention, votre expression est mal parenthesée*

Test : 2(3+4)
Résultat : *Attention, votre expression est mal parenthesée*

Test : (3+4)2
Résultat : *Attention, votre expression est mal parenthesée*

Test : 1+2()
Résultat : *Attention, votre expression est mal parenthesée*

Test : 5(2+1)*
Résultat : *Attention, votre expression est mal parenthesée*
Commentaire : test intéressant, puisqu'en appliquant simplement l'algorithme de conversion, on obtient une expression cohérente qui peut être ensuite évaluée. L'erreur générée prouve que le module de conversion vérifie correctement l'expression avant de la convertir.

Tests avec un parenthèsage correcte :

Test : (1+2)
Résultat : 3

Test : (1+2)*3
Résultat : 9

Test : 3*(1+2)
Résultat : 9

Test : ((3*(((((1+2))+2)))))
Résultat : 15

Tests portant sur les opérateurs de l'expression :

Test : +
Résultat : *Attention, l'expression ne peut pas debuter par un operateur*

Test : +1
Résultat : *Attention, l'expression ne peut pas debuter par un operateur*

Test : 1+
Résultat : *Attention, l'expression ne peut pas terminer par un operateur*

Test : 1++1
Résultat : *Attention, un operateur est mal place ou manquant*

Test : 1+1+
Résultat : *Attention, l'expression ne peut pas terminer par un operateur*

Test : 2/(1-1)
Résultat : *Attention, division par 0*

Autres tests :

Test : ab
Résultat : *Attention, un operateur est mal place ou manquant*
Commentaire : Deux identificateurs ne peuvent se suivre au sein d'une expression.

Test : <VIDE>
Résultat : *Attention, vous devez taper une expression*

Test : a+b*(c+d-(e+f/g))
Résultat :
Merci de saisir la valeur associee a : g
2
Merci de saisir la valeur associee a : f
1
Merci de saisir la valeur associee a : e
2
Merci de saisir la valeur associee a : d
3
Merci de saisir la valeur associee a : c

5
Merci de saisir la valeur associee a : b
2
Merci de saisir la valeur associee a : a
1
Resultat : 13

Test : (a*((b-a)+d))
Résultat : Merci de saisir la valeur associee a : d
1
Merci de saisir la valeur associee a : b
3
Merci de saisir la valeur associee a : a
2
Resultat : 4

Test : 1-2-3
Résultat : -4

Test : 1+2+3*4
Résultat : 15

6 Conclusion

En conclusion, ce projet aura été bien plus intéressant dans sa réalisation que le projet CamL. Effectivement, la conception de ce dernier était imposée en grande partie par le cahier des charges. Pour ce projet C, les spécifications imposent d'effectuer certains choix conceptuels, mais laissent malgré tout une grande liberté pour la conception et la programmation du projet.

Au final, ce projet aura été réalisé sans problèmes particuliers, les seuls ennuis rencontrés sont effectivement certains cas de syntaxe incorrecte pour l'expression infixe que je n'avais pas envisagés dans la première version de l'algorithme, et des codes d'erreur qui ne remontaient pas jusqu'au *main*. Problèmes qui ont rapidement été fixés.