Université de Liège

INFO0947

Compléments de Programmation

Un exercice dont vous êtes le Héros · l'Héroïne.

Piles – Manipulation d'Expressions

Simon Liénardy Benoit Donnet

19 avril 2020

Préambule

Exercices

Dans ce « TP dont vous êtes le héros », nous vous proposons de suivre pas à pas la résolution d'un exercice sur l'utilisation des Piles pour l'évaluation des expressions.

Il est dangereux d'y aller seul [1] !

Partagez vos commentaires, questions, solutions alternatives sur le forum eCampus. N'hésitez jamais !

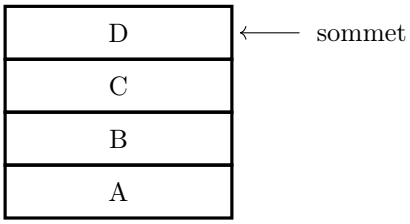
1. Référence vidéoludique bien connue des Héros.

7.1 Rappel sur les Piles

Si vous avez déjà lu ce rappel, vous pouvez directement atteindre l'énoncé de l'exercice.

7.1.1 Pile ?

Une *Pile* (« Stack ») est une structure de données linéaire basée sur le principe *LIFO* (LastIn – FirstOut – Chapitre 7, Slides 5 → 7). Cela signifie qu'on ne peut manipuler (ajout/retrait) la Pile que via son *sommet*. Un exemple de Pile avec quatre valeurs est donnée ci-dessous (la valeur D est sur le sommet de la Pile) :



7.1.2 Opérations

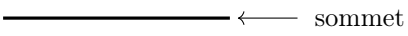
La spécification abstraite partielle [2] d'une Pile est donnée ci-dessous :

```
Type:
  Stack
Utilise:
  Boolean, Element
Opérations:
  empty_stack: → Stack
  is_empty: Stack → Boolean
  push: Stack × Element → Stack
  pop: Stack → Stack
  top: Stack → Element
```

La Pile utilise définit cinq opérations. Nous détaillons, ci-dessous, l'effet de chacune de ces opérations.

7.1.2.1 Créer d'une Pile

L'opération *empty_stack* permet de créer une Pile vide. C'est donc un constructeur.
Nous représentons, graphiquement, la Pile vide de la façon suivante :



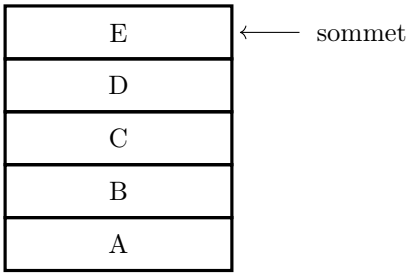
7.1.2.2 Vérifier si une Pile est vide

L'opération *is_empty* permet de vérifier si une Pile est vide. Il s'agit donc d'un observateur.

2. Pour compléter la spécification abstraite, il faut rajouter les Préconditions et les Axiomes (soit la partie sémantique de la spécicification abstraite). Voir Chapitre 7, Slide 9.

7.1.2.3 Empiler

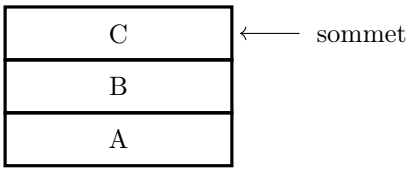
L'opération *push* permet d'ajouter un élément en sommet de Pile. Il s'agit donc d'un transformateur. Si on applique l'opération $push(E)$ à la Pile illustrée en début de rappel, on obtiens la Pile suivante :



On voit bien que l'indicateur de sommet est déplacé et pointe sur l'élément nouvellement ajouté sur la Pile

7.1.2.4 Dépiler

L'opération *pop* permet d'enlever un élément en sommet de Pile. Il s'agit donc d'un transformateur, puisque la Pile est modifiée après cette opération. Attention, c'est une opération partielle : impossible de dépiler une Pile vide. Si on applique l'opération *pop* à la Pile illustrée en début de rappel, on obtient la Pile suivante :



On voit bien que l'indicateur de somme est déplacé et pointe sur l'élément se situant maintenant en sommet de Pile (i.e., C).

7.1.2.5 Inspecter le Sommet

L'opération *top* permet de connaître l'élément se situant en sommet de Pile sans le supprimer (contrairement à *pop*). Cette opération est donc bien un observateur. Attention, à l'instar de *pop*, *top* est une opération partielle : impossible de retourner le sommet d'une Pile vide. Si on applique l'opération *top* à la Pile illustrée en début de rappel, on obtient la valeur D, la Pile étant inchangée.

7.1.3 Interface (`stack.h`)

L'implémentation d'une Pile a peu d'importance dans le cadre de cet exercice. Nous pouvons utiliser directement la Pile grâce à l'interface (i.e., header) suivante :

```
1  #ifndef __STACK__
2  #define __STACK__
3  #include "boolean.h"
4
5  #include "boolean.h"
6
7  typedef struct stack_t Stack;
8
9  Stack *empty_stack(void);
10
```

```
11 Boolean is_empty(Stack *s);
12
13 Stack *push(Stack *s, void *e);
14
15 Stack *pop(Stack *s);
16
17 void *top(Stack *s);
18
19 void free_stack(Stack **s);
20
21 #endif
```

7.2 Rappel sur les Expressions

Si vous avez déjà lu ce rappel ou si vous êtes suffisamment à l'aise avec les expressions et leurs évaluations, vous pouvez directement atteindre l'énoncé de l'exercice.

7.2.1 Expression

Une *expression* (cfr. INFO0946, Chapitre 1, Slide 17) est la description du calcul d'une valeur, le résultat du calcul ayant un certain type.
Plus précisement, une expression est :

1. une constante (ou valeur littérale – e.g., 'a'). L'évaluation retourne le littéral lui-même ;
2. une variable, dénotée par son identificateur (e.g., `x`). Dans ce cas, l'évaluation retourne la valeur courante de la variable ;
3. obtenue par l'application d'opérateurs à d'autres expressions.

L'œil aguerri aura reconnu, ici, une définition par induction. Les points 1 et 2 sont les cas de base, le point 3 étant le cas inductif.

Il existe plusieurs façon de représenter une expression : complètement parenthésée, préfixée, postfixée, et infixée (voir Chapitre 7, Slides 28 → 35). Nous nous intéressons, ici, à la représentaiton postfixée.

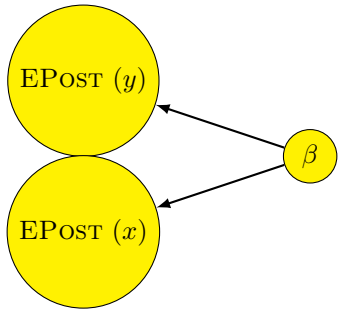
7.2.2 Expression Postfixée

Une expression peut être écrite sous forme *postfixée* (EPost) (cfr. Chapitre 7, Slide 32).

Plus précisement, une EPost est :

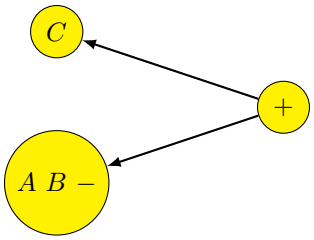
1. une variable ;
2. soient x et y des EPost et β un opérateur binaire, alors $x\ y\ \beta$ est une EPost ;
3. soit x une EPost et α un opérateur unaire, alors $x\ \alpha$ est une EPost.

Le deuxième point de la définition d'une EPost peut se représenter comme suit :

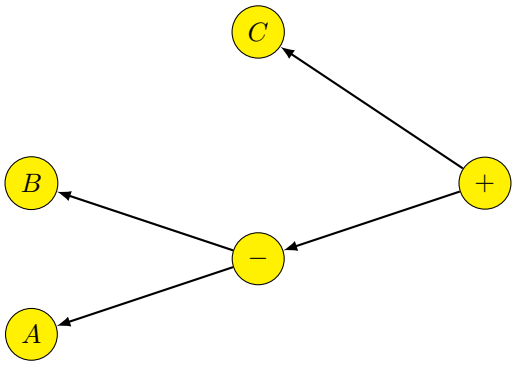


Prenons un exemple : l'expression $((A\ -\ B)\ +\ C)$ se transforme, en EPost, de la façon suivante : $A\ B\ -\ C\ +$

Il s'agit bien d'une EPost puisqu'on dispose d'un opérateur binaire $(+)$ qui s'applique deux EPost de la façon suivante :



C est bien une EPost car il rentre dans le premier point de la définition (i.e., une variable). L'expression $A\ B\ -$ est bien une EPost car on peut la déplier de la façon suivante :



A et B sont bien des EPost car ils correspondent au premier point de la définition (i.e., des variables).

7.3 Énoncé

Spécifiez et construisez un algorithme utilisant une pile et permettant d'évaluer une expression écrite sous forme postfixée (EPost). Nous considérons que l'EPost se présente comme un tableau de caractères.

Soient les variables (A, B, C, D) et l'environnement $(20, 4, 9, 7)$. Indiquez aussi la trace de votre algorithme, sur votre pile, pour l'expression $AB \times CD + /$.

7.3.1 Opérations sur les Expressions

Pour les exercices qui suivent, vous disposez aussi des fonctions/procédures suivantes :

— `operateur(`β`)` qui détermine si β est un opérateur binaire ;

— `unaire(`α`)` qui détermine si α est un opérateur unaire ;

— `variable(`A`)` qui détermine si A est une variable ;

— `valeur(`A`)` qui retourne la valeur associée à la variable A.

7.3.2 Méthode de résolution

Nous allons suivre l'approche constructive vue au cours. Pour ce faire nous allons :

1. Trouver une notation pour exprimer formellement le problème (Sec. 7.4) ;
2. Spécifier le problème complètement (Sec. 7.5) ;
3. Chercher un Invariant (Sec. 7.6) ;
4. Construire le programme complètement (Sec. 7.7) ;
5. Rédiger le programme final (Sec. 7.8) ;
6. Donner la trace d'exécution de votre programme (Sec. 7.9).

Alerte : Exercice difficile !

Cet exercice est probablement l'un des plus compliqués à faire dans le cadre du cours. Il demande donc d'être particulièrement concentré pour être mené à son terme.

Nous insistons sur le fait qu'il vaut mieux prendre le temps de le faire seul et de poser des questions sur eCampus plutôt que de lire directement la solution : cela ne vous servirait **à rien**.

7.4 Notations

Avant de poursuivre avec les spécifications, il faut d'abord réfléchir aux notations dont nous allons avoir besoin dans la suite.

Si vous voyez de quoi on parle, définissez vos notations et rendez-vous à la Section 7.4.3
Si vous voyez de quoi on parle mais que vous ne vous souvenez plus de comment définir une notation, voyez la Section 7.4.1
Enfin, si vous séchez sur l'information à faire apparaitre dans une notation, voyez l'indice à la Section 7.4.2

7.4.1 Rappel – définir une notation

Voici la forme que doit prendre une notation :

NomPredicat(Liste de parametres) ≡ Détail du predicat

NomPredicat est le nom à donner du prédicat. Il faut souvent trouver un nom en rapport avec ce que l'on fait. Mathématiquement parlant, cela ne changera rien donc on pourrait appeler le prédicat « Simon » mais il vaut mieux trouver un nom en rapport avec la définition.

Liste de paramètres C'est une liste de symboles qui pourront être utilisés dans la définition du prédicat. Comment cela fonctionne-t-il ? Le prédicat peut être utilisé en écrivant son nom ainsi qu'une valeur particulière de paramètre. On remplace cette valeur dans la définition du prédicat et on regarde si c'est vrai ou faux (Voir TP1).

Détail du prédicat C'est une combinaison de symboles logiques qui fait intervenir, le plus souvent les paramètres.

Exemple :

$$\text{Pair}(X) \equiv X \ \% \ 2 = 0$$

Le nom du prédicat est « Pair » et prend un argument. $Pair(4)$ remplace $4 \ \% \ 2 = 0$ qui est vrai et $Pair(5)$ signifie $5 \ \% \ 2 = 0$ qui est faux. Le choix du nom du prédicat est en référence à sa signification : il est vrai si X est pair.

▷ **Exercice** Si vous avez compris, vous pouvez définir facilement le prédicat Impair(X).

7.4.1.1 Voir aussi

Comment évaluer un prédicat	TP 1
Signification des opérateurs logiques	Chapitre 1, Slides 6 → 9
Signification des quantificateurs logiques	Chapitre 1, Slides 11 → 24
Signification des quantificateurs numériques	Chapitre 1, Slides 25 → 33

Suite de l'exercice

Si vous voyez de quoi on parle, définissez vos notations et rendez-vous à la Section 7.4.3

Enfin, si vous séchez sur l'information à faire apparaitre dans une notation, voyez l'indice à la Section 7.4.2

7.4.2 Indices – définir une notation

7.4.2.1 Indice 1 : notion à définir

Que nous faut-il comme notation. Pour ce faire, il faut relire l'énoncé. On y trouve la phrase :

« évaluer une expression écrite sous forme postfixée »

La notion d'évaluation d'une EPost gagnerait à être définie par une notation.

7.4.2.2 Indice 1 : comment faire ?

il faut relire le rappel sur les notations postfixées. On voit que la définition d'une EPost est récursive :

Une EPost est :

1. une variable ;
2. soient x et y des EPost et β un opérateur binaire, alors $x\ y\ \beta$ est une EPost ;
3. soit x une EPost et α un opérateur unaire, alors $x\ \alpha$ est une EPost.

Une EPost est obtenue en combinant des EPost plus simples. Pour définir ce qu'est l'évaluation d'une EPost, il faudra donc sûrement suivre ce schéma récursif.

Suite de l'exercice

Vous avez toutes les clés en main pour définir une notation qui exprime comment calculer la valeur d'une EPost. Rendez-vous à la Sec. 7.4.3.

7.4.3 Mise en commun – définir une notation

Appelons « *Eval* » la fonction qui permet d'évaluer une expression postfixée. Elle prend en argument une EPost.

Cas de base Une variable est une EPost. La valeur de l'EPost est donc la valeur associée à cette variable :

$$Eval(A) \equiv valeur(A) \text{ si } A \text{ est une variable}$$

valeur est la fonction donnée dans l'énoncé.

Cas récursif 1 Soit x, une EPost et α un opérateur unaire, la valeur de $x\ \alpha$ est :

$$Eval(x\ \alpha) \equiv \alpha\ Eval(x)$$

On applique l'opérateur unaire α sur le résultat de l'évaluation de l'EPost x.

Cas récursif 2 Soient x et y, deux EPost et β un opérateur binaire, la valeur de $x\ y\ \beta$ est :

$$Eval(x\ y\ \beta) \equiv Eval(x)\ \beta\ Eval(y)$$

On applique l'opérateur binaire β sur les valeurs des EPost x et y. β est replacé au centre de l'évaluation pour qu'on comprenne bien dans quel ordre on évalue : par exemple, si β est l'opérateur de division, $Eval(x)$ sera le dividende et $Eval(y)$ le diviseur.

Synthèse En résumé, la notation obtenue est donc

$$Eval(E) \equiv \begin{cases} valeur(E) & \text{si } E \text{ est une variable ;} \\ \alpha\ Eval(x) & \text{si } E \text{ est de la forme } x\ \alpha ; \\ Eval(x)\ \beta\ Eval(y) & \text{si } E \text{ est de la forme } x\ y\ \beta. \end{cases}$$

Ll faut maintenant passer aux spécifications. Voir la Sec. 7.5.

7.5 Spécification

Il faut maintenant spécifier le problème.

Pour un rappel sur la spécification, voyez la Section 7.5.1
La correction de la spécification est disponible à la Section 7.5.3

7.5.1 Rappel sur les spécifications

Voici un rappel à propos de ce qui est attendu :

La Précondition C'est une condition sur les données. Pour l'établir, il faut se demander quelles doivent être les données du problème. Ici, dans notre exemple, cela parle de tableau. Quelles conditions devons-nous imposer sur ce tableau ? N'hésitez pas à vous référer au cours INFO0946, Chapitre 6, Slides 54 → 64).

La Postcondition C'est une condition sur le résultat du problème, si tant est que la Précondition est remplie (si tel n'est pas le cas, on pratique la *programmation défensive* (voir INFO0946, Chapitre 6, Slides 65 → 70) et on arrête l'exécution du programme.

La signature est souvent nécessaire lors de l'établissement de la Postcondition des fonctions (voir INFO0946, Chapitre 6, Slide 11). En effet, le résultat retourné par l'invocation de la fonction est représenté par le nom de la fonction dans la Postcondition. Le nom de la fonction doit donc être défini avant d'en parler.

7.5.2 Suite de l'exercice

Spécifiez le problème. rendez-vous à la Sec. 7.5.3 pour la correction !

7.5.3 Spécification du problème

Précondition L'EPost se présente comme un tableau de caractères. Il est évident qu'on ne peut pas évaluer une expression qui n'existe pas (cfr. la définition d'une EPost). On va donc imposer que notre tableau de caractères, `expr`, est initialisé et valide. Soit :

$$Pre : expr \neq NULL$$

Postcondition La fonction, `eval`, retourne la valeur de l'évaluation de l'EPost qui est une valeur entière. On réutilise la notation que nous avons définie précédemment. La Postcondition est donc :

$$Post : eval = Eval(expr)$$

Alerte : Studentite aigüe

Si vous n'avez pas une Postcondition aussi simple, c'est que vous avez contracté une studentite aigüe, la maladie caractérisée par une inflammation de l'étudiant qui cherche midi à quatorze heures.

Heureusement, ce n'est pas dangereux tant que vous êtes confiné ! Faites une petite pause dans cette résolution. Prenez le temps de vous oxygéner, en faisant un peu d'exercice physique par exemple. Trois répétitions de dix pompes devraient faire l'affaire.

Si vous avez **déjà** contracté la studentite aigüe précédemment, vous risquez d'atteindre le stade chronique ! Vous vous engagez donc dans un traitement prophylactique [a] de fond : trois répétitions de 20 pompes.

a. Se dit de quelque chose qui prévient une maladie

Signature

```
1 int eval(char *expr);
```

Suite

Passez maintenant à la recherche d'un Invariant : Sec. 7.6.

7.6 Invariant

L'Invariant de ce problème n'est pas des plus simples à trouver (c'est même probablement le plus compliqué depuis de début de l'année académique). En fait, il faut avoir pris connaissance de quelques éléments avant de se lancer dans sa rédaction :

— Il faut décider si on parcourt la chaine de caractères de gauche à droite ou de droite à gauche. Comment décider ? Il faudrait être familier avec les EPost pour y aller à l'intuition. Ce n'est pas notre cas mais nous pouvons par exemple réfléchir sur base d'un exemple. Essayons avec l'évaluation d'une expression :

$$A\ B\ C\ D\ +\ -\ \times,\ (A, B, C, D) = (17, 42, 18, 25)$$

— Il faut se demander de quelle structure de données nous avons besoin. Le titre du Chapitre à beau être « les Piles », cela n'a **aucun sens** d'utiliser une Pile juste parce que le professeur ou l'assistant ont commodément chapitré le cours ! Nous savons que l'EPost est représentée sous la forme d'une chaine de caractère. Nous savons, par la pratique des invariants, que nous devrons dessiner une *ligne de démarcation* dans l'EPost en train d'être évaluée. Pour faire avancer le programme, nous devrons utiliser « ce qui a déjà été calculé » dans les itérations précédentes. La manière dont nous devons stocker « ce qui a déjà été calculé » déterminera la structure de donnée dont nous aurons besoin.

— Pour décrire exactement l'Invariant le plus formellement possible, nous allons avoir besoin de dégager une propriété des EPost. Comme au point précédent, il faut examiner une EPost dans laquelle on a dessiné une ligne de démarcation et essayer, pour la partie déjà examinée, de dégager une « propriété utile pour la décrire ». Quoi précisément ? C'est cela qu'il faut chercher !

Suite de l'exercice

Commencez à chercher un Invariant pour ce problème en ayant en tête les trois éléments ci-dessus. Essayez vraiment d'avoir une ébauche assez aboutie (c'est-à-dire que vous avez des arguments pour répondre aux trois questions soulevées) avant de regarder la solution ou l'idée de la solution.

7.6.1 Rappels sur l'Invariant

Si vous voyez directement ce qu'il faut faire, continuez en lisant les conseils 7.6.1.1.

Petit rappel de la structure du code (cfr. Chapitre 2, Slide 28) :

```
// {Pre}
INIT
// {Inv}
while(B){
   // {Inv ∧ B}
   ITER
   // {Inv}
}
// {Inv ∧ ¬B}
END
// {Post}
```

La première chose à faire, pour chaque sous-problème, est de déterminer un Invariant. La meilleure façon de procéder est de d'abord produire un Invariant Graphique et de le traduire ensuite formellement. Le code sera ensuite construit sur base de l'Invariant.

Règles pour un Invariant Satisfaisant

Un bon Invariant Graphique respecte ces règles :

1. La structure manipulée est correctement représentée et nommée ;
2. Les bornes du problèmes sont représentées ;
3. Il y a une (ou plusieurs) ligne(s) de démarcation ;
4. Chaque ligne de démarcation est annotée par une variable ;
5. Ce qu'on a déjà fait est indiqué par le texte et utilise des variables pertinentes ;
6. Ce qu'on doit encore faire est mentionné.

Ce sera un encore meilleur Invariant dès que le code construit sur sa base sera en lien avec lui.

7.6.1.1 Quelques conseils préalables

Une fois n'est pas coutume, les conseils ont été donnés d'emblée au début de la Section. Une bonne manière de précéder est de suivre les conseils !

Suite de l'exercice

Indice – Pour la détermination du sens du parcours de l'EPost 7.6.2
Indice – Pour la détermination des structures de données nécessaires 7.6.3
Indice – Pour la propriété qui nous sera nécessaire par la suite 7.6.4
Alerte divulgâchage – Pour l'idée de la solution 7.6.5
Pour l'Invariant Graphique 7.6.6
Pour l'Invariant Formel 7.6.7

7.6.2 Indice – Sens de parcours de l'EPost

Soit l'expression à évaluer :

$$A\ B\ C\ D\ +\ -\ \times,\ (A,B,C,D) = (17,42,18,25)$$

Avant de lire la suite, évaluez-là, vous devriez obtenir le résultat -17.

Tentative n° 1 – De la droite vers la gauche Supposons que l'on commence à parcourir l'EPost par la droite. Dans notre cas particulier, nous lisons l'opérateur $\times$ mais nous ne connaissons pas ses opérandes. Celles-ci seront calculées par la suite mais on ne sait pas quand. Quand nous rencontrons la première variable (ici : D), nous ne savons pas non plus qu'en faire. Par contre, une fois que nous avons rencontré la variable C, on peut se souvenir qu'on a vu précédemment un $+$ et qu'il faut donc effectuer le calcul $C + D$. On voit alors :

— lorsqu'on voit un opérateur, on ne sait pas quand on devra l'utiliser. On pourrait, certes, le stocker dans une structure de données ;

— lorsqu'on voit une variable, on ne peut décider ce qu'on en fait qu'en fonction du contexte : soit on doit retenir qu'on a vu cette variable (mieux vaut d'ailleurs retenir sa valeur plutôt que son nom), soit on peut lancer un calcul. Déterminer clairement la condition qui discrimine ces deux cas pourrait s'avérer complexe.

Tentative n° 2 – De la gauche vers la droite Supposons que l'on commence à parcourir l'EPost par la droite. Nous rencontrons la variable A, qui répond à la définition des EPost et pour l'instant, on ne sait pas si l'EPost continue après ce seul symbole. La meilleure chose que nous pouvons faire c'est déterminer la valeur de A et sauvegarder ce résultat. Passons au caractère suivant et nous rencontrons la variable B. Nous pouvons au moins l'évaluer et retenir le résultat. Idem pour C et D. Quand nous rencontrons l'opérateur $+$, nous pouvons nous rappeler de ses opérandes car nous les avons déjà vues et calculer facilement $C + D$ que faire du résultat calculé ? Le retenir en mémoire comme les autres ! On voit alors :

— lorsqu'on voit un opérateur, on **est sûr** qu'il faut effectuer l'opération correspondantes. La valeur des opérandes est *quelque part* en mémoire. Où précisément, c'est l'objet d'un autre indice.

— lorsqu'on voit une variable, il faut l'évaluer et retenir son résultat.

Conclusion Des deux méthodes, une seule possède des règles de décisions **claires et précises** quant à ce qu'il faut faire du caractère lu : la seconde tentative qui correspond à lire l'expression de la gauche vers la droite.

Alerte : Excès de confiance en soi

Si vous vous dites « Putain, ils sont cons de justifier le parcours de la gauche vers la droite, on a toujours fait comme ça » (ou une variante). De un, calmez-vous ! De deux, rappelez-vous l'intitulé de vos études : *Sciences* Informatiques. Attention donc à ces mauvaises justifications :

L'habitude : n'a jamais été un argument scientifique et ne peut donc pas justifier notre manière de procéder. Remettre en question les connaissances est un invariant de la démarche scientifique ;

La chance : peut vous sourire si vous avez choisi par défaut le bon sens de parcours mais rien n'indique que vous continuerez d'être chanceux ;

Le manque de rigueur : « parce que » n'est pas une justification valable non plus. Il faut donner de la profondeur à notre argumentation.

Les comparaisons foireuses : les EPost sont certes écrites de gauche à droite comme on le fait en français mais cela n'a rien à voir avec le problème qui nous préoccupe.

⇒ La décision de parcourir de la gauche vers la droite doit être rigoureusement justifiée.

7.6.3 Indice – Quelle·s structure·s de données utiliser ?

Nous avons précédemment déterminé qu'il fallait lire l'EPost de la gauche vers la droite (pas convaincu·e : relisez la Sec. 7.6.2).

Soit l'EPost

$$A\ B\ C\ D\ +\ -\ \times,\ (A, B, C, D) = (17, 42, 18, 25)$$

est composée de 7 caractères. Nous les lisons dans l'ordre. Nous allons représenter la mémoire comme un ensemble. Le but va être de déterminer si la politique d'ajout ou de retrait d'éléments dans cette mémoire suit une règle particulière :

1. On lit A et on retient $\{17\}$.
2. On lit B et on retient aussi 42, on a retenu jusqu'à présent $\{17, 42\}$.
3. On lit C et on retient aussi 18, on a retenu jusqu'à présent $\{17, 18, 42\}$.
4. On lit D et on retient aussi 25, on a retenu jusqu'à présent $\{17, 18, 25, 42\}$.
5. On lit $+$, on effectue $18 + 25 = 43$, on retient 43, on a retenu jusqu'à présent $\{17,\ 42,\ 43\}$.
6. On lit $-$, on effectue $42 - 43 = -1$, on retient -1, on a retenu jusqu'à présent $\{-1, 17\}$.
7. On lit $\times$, on effectue $17 \times -1 = -17$, on retient -17, on a retenu jusqu'à présent $\{-17\}$.

Que peut-on en dégager ? Les valeurs entrent en mémoire dans le sens ou elles apparaissent dans l'EPost. Par contre, quand il s'agit de les sortir de la mémoire, par exemple à l'étape 5, on sélectionne 18 et 25 qui se trouvent être les dernières valeurs ajoutées. On peut s'assurer qu'il en est de même aux étapes suivantes.

Conclusion On doit donc utiliser une structure de donnée dont la principale propriété est une politique d'accès qui consiste à sortir d'abord les dernières valeurs entrées : **LastIn – FirstOut**. En d'autres mots, il nous faut une Pile ! L'Invariant devra donc **décrire précisément** quel est son contenu.

7.6.4 Indice – Propriété intéressante des EPost

Reprenons notre EPost d'exemple : $A\ B\ C\ D\ +\ -\ \times$. Nous savons que nous allons la parcourir de gauche à droite (pas convaincu·e : relisez la Sec. 7.6.2). Nous savons qu'au moment d'établir l'Invariant, nous allons tracer une ligne de démarcation dans l'EPost et nous allons devoir décrire ce qui a déjà été calculé à partir de ce qui se trouve à gauche de la ligne de démarcation. Voici les possibilités de découpe de l'exemple :

1.	| A	B	C	D	$+$	$-$	$\times$
2.	A |	B	C	D	$+$	$-$	$\times$
3.	A	B |	C	D	$+$	$-$	$\times$
4.	A	B	C |	D	$+$	$-$	$\times$
5.	A	B	C	D |	$+$	$-$	$\times$
6.	A	B	C	D	$+$ |	$-$	$\times$
7.	A	B	C	D	$+$	$-$ |	$\times$
8.	A	B	C	D	$+$	$-$	$\times$ |

Table 1 – Quelle est la propriété communes à tous ces préfixes d'EPost ?

Pour la ligne 2, on a le cas de base d'une EPost. Pour la ligne 8, on a complètement l'EPost de départ. Essayons maintenant de caractériser les préfixes suivants. Pour la ligne 3, on a une succession d'EPost de base : A et B. Idem pour les lignes 4 et 5. Regardons la ligne 6, on a A et B qui sont des EPost (C et D aussi d'ailleurs) mais en ajoutant le caractère $+$, on voit qu'on a une succession de 3 EPost : A, B et $C\ D\ +$. Un schéma se répète :

> « *Un préfixe d'*EPost *est aussi une concaténation d'*EPost. »

Si un seul exemple ne vous convient pas, essayez-en d'autres !

Nous avons donc une propriété sur les préfixes d'EPost mais est-elle vraie ? En fait, si nous voulions être 100% rigoureux, nous en apporterions la preuve. C'est tout à fait possible [3] !

Maintenant que nous pouvons caractériser le préfixe d'EPost qui a été déjà analysé, nous pouvons beaucoup plus facilement écrire l'Invariant qui doit, entre autre, expliquer ce que sera le contenu de la Pile.

3. Vous êtes invité à en discuter sur eCampus. Des indices : Chapitre 1, $\prec$, induction forte (Slide 56). Vous pouvez aussi vous convaincre et démontrer qu'une telle décomposition est unique.

7.6.5 Indice – Idée de résolution

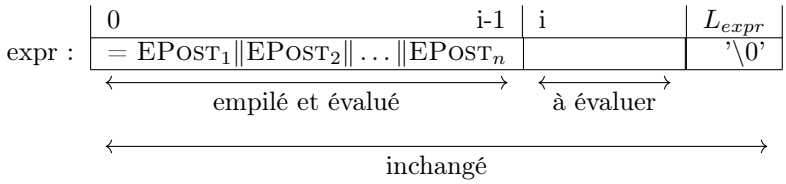
Pour évaluer une expression postfixée du type $A\ B \times C\ D + /$, il faut pouvoir sauvegarder les résultats intermédiaires $A\ B\times$ et $C\ D+$ et ensuite recharger ces résultats pour effectuer la division. L'expression sera balayée de gauche à droite. Si une variable est rencontrée, elle est mise sur la Pile. Si un opérateur est rencontré, on dépile ses arguments et on effectue l'opération. Le résultat ainsi produit est tout de suite empilé et on passe au caractère suivant de l'expression.

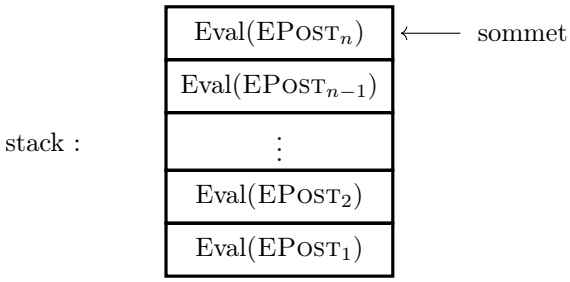
Pour un rappel sur les Piles, rendez-vous à la Sec. 7.1.
L'Invariant Graphique est disponible à la Sec. 7.6.6

7.6.6 Invariant Graphique

Couper la chaine *expr* en deux et dire qu'une partie a été évaluée et l'autre pas n'est pas suffisant. Il faut aussi et surtout **décrire le contenu de la Pile**.

L'Invariant Graphique est le suivant :





Le tableau de caractères représentant l'EPOST est bien formé. Pour rappel, toute chaine de caractères se termine par le caractère de terminaison, i.e., '\0' (cfr. INFO0946, Chapitre 5, Slides 105 → 109). L'Invariant Graphique représente cette situation en indiquant, dans la dernière case du tableau, le caractère de terminaison. Ce caractère se situe à la position « L_{expr} » qui correspond à la longueur de la chaine. Il n'est pas obligé de l'indiquer tant que le caractère de fin de chaine est correctement dessiné.

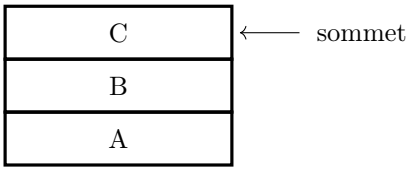
Le sous-tableau `expr[0 ...i-1]` forme le préfixe d'une EPOST. Selon la propriété introduite précédemment, c'est donc une concaténation (symbole ‖ dans l'Invariant Graphique) de plusieurs EPOST. Les valeurs de ces EPOST ont été empilées sur la Pile, `stack`, dans leur ordre de rencontre. Le sommet de la Pile contient donc toujours le résultat de la dernière évaluation ($\text{Eval}(\text{EPOST}_n)$).

La traduction de cet Invariant Graphique en Invariant Formel est disponible à la Sec. 7.6.7.

7.6.7 Invariant Formel

7.6.7.1 Notation formelle pour la Pile

Il n'y a pas de notation formelle pour les Invariants sur les Piles dans le cours. On peut donc on peut en inventer une. Par exemple, ici, on va encadrer les éléments par les symboles $\lfloor$ et $\rceil$, les espacer par le symbole $\uparrow$ qui rappelle que les éléments sont empilés. Le sommet de la Pile sera mis en évidence entre deux crochets ($[\ ]$). En d'autres termes, le dessin suivant :

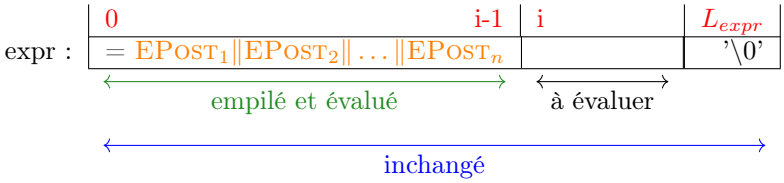


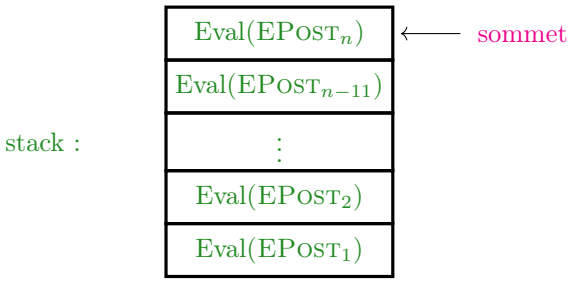
sera représenté par la notation suivante :

$$\lfloor A \uparrow B \uparrow [C] \rceil$$

7.6.7.2 Invariant

On traduit l'Invariant Graphique en Invariant Formel :





$$Inv : expr = expr_0 \tag{1}$$
$$\wedge\ 0 \leq i \leq L_{expr} \tag{2}$$
$$\wedge\ (\exists n, expr[0..i-1] = \mathop{\Big\|}_{j=1}^{n} \text{EPost}_j, \tag{3}$$
$$stack = \lfloor Eval(\text{EPost}_1) \uparrow \ldots \uparrow Eval(\text{EPost}_{n-1}) \uparrow [Eval(\text{EPost}_n)] \rceil) \tag{4}$$

Ce qui signifie :

1. `expr` n'est pas modifié ;
2. présentation de i qui est compris entre 0 et la longueur de la chaine de caractère ;

3. on commence une quantification existentielle qui va exprimer qu'il existe un nombre n de EPost qui, une fois concaténés, sont égaux au préfixe `expr[0, ...,i-1]` ;
4. on continue l'existentielle en disant que la pile doit également contenir n valeurs qui correspondent à l'évaluation de ces n EPost, dans l'ordre de leur rencontre de gauche à droite.

Remarque Comme n est une variable liée, l'Invariant ne dépend pas de n (n est la taille de la Pile en fait). Les variables libres, qui sont donc des variables qui doivent être présentes dans le programme sont : `expr`, `i`, `stack`.[4]

4. Variable libre vs. Variable liée ? cfr. Chapitre 1, Slides 22 → 24.

7.7 Construction du Code

Voici les différentes étapes de l'Approche Constructive (Chapitre 2, Slides 25 → 37)

INIT De la Précondition, il faut arriver dans une situation où l'Invariant est vrai.

B Il faut trouver d'abord la condition d'arrêt $\neg B$ et en déduire le gardien, B.

ITER Il faut faire progresser le corps de la boucle puis restaurer l'Invariant.

END Vérifier si la Postcondition est atteinte si $\{Inv \wedge \neg B\}$, sinon, amener la Postcondition.

Fonction t Il faut donner une fonction de terminaison pour montrer que la boucle se termine (cfr. INFO0946, Chapitre 3, Slides 97 → 107).

Tout ceci doit être justifié sur base de l'Invariant, à l'aide d'assertions intermédiaire. Notez que l'ordre dans lequel vous faites ces étapes n'a pas d'importance !

Suite de l'exercice

La suite de l'exercice est disponible à la Sec. 7.7.1.

7.7.1 Approche constructive pour notre problème

7.7.1.1 Programmation défensive

Il suffit de tester l'adresse du tableau.

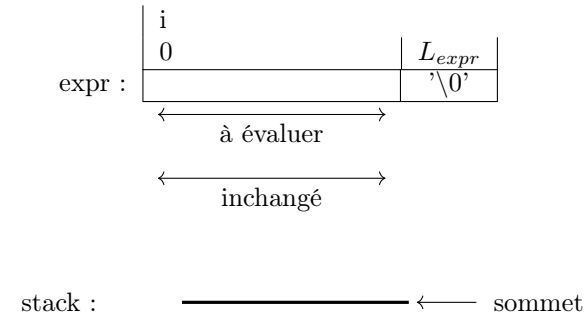
```
int eval(char *expr){
  assert(expr);
  // {Pre}

  //à suivre
}
```

7.7.1.2 INIT

Méthode formelle Il faut se mettre dans une situation où l'Invariant est d'une part vrai mais où les valeurs particulières des variables expriment que rien n'a été fait jusqu'à présent. On se doute qu'il faut que la Pile soit vide. [5] L'existentielle présente dans l'Invariant est vraie si la Pile et le préfixe sont tous les deux vides (en effet, un préfixe vide est une concaténation de 0 EPost dont les évaluations sont bien sur la Pile vide puisqu'il n'y en a aucune). Vu les valeurs possibles pour `i`, $i = 0$ correspond à cette situation.

Méthode graphique La méthode graphique consiste à prendre la ligne de démarcation de la chaine de caractère et de la glisser vers la gauche. On obtient le dessin suivant :

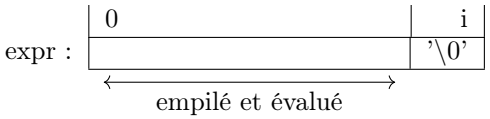


On obtient évidemment les même résultat qu'avec la méthode formelle : i doit être initialisé à 0, la Pile doit être vide et on ne touche pas à $expr$.

```
// {Pre}
unsigned int i = 0; // Signé ou non est OK
// {expr[0..i − 1] = "" ⇒ expr[0..i − 1] = ||_{j=1}^{0} EPost_j}
Stack *stack = empty_stack();
// {stack = ⌊[ ]⌋ }
// {⇒ (∃n, expr[0..i − 1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]⌋) ⇒ Inv}
```

7.7.1.3 B

Méthode formelle L'Invariant dit que la Pile contient des évaluation d'EPost qui, concaténées, forment un préfixe de $expr$. La situation finale consisterait à ce que $expr[0..i-1] = expr$. Pour une chaine de caractères, c'est possible quand $i = L_{expr}$ et donc que `expr[i] == '\0'`.



Méthode graphique L'Invariant Graphique est ici très utile. Si on reprend le tableau `expr` et qu'on adapte l'Invariant Graphique pour décrire la situation finale (i.e., décalage de la ligne de démarcation vers la droite), on obtient :

Un fois que la valeur à l'indice `i` dans `expr` sera le caractère de terminaison, cela signifiera que l'entièreté de l'EPost a été évaluée. Il vient donc :

$$B \equiv expr[i] \neq' \backslash 0'$$

Dans le code, on ne comparera pas explicitement `expr[i]` à la valeur `'\0'` puisque cette dernière a déjà la valeur entière 0 qui représente la valeur Booléenne fausse.

7.7.1.4 ITER

L'Invariant exprime que `expr[0 .. i-1]` a déjà été traité, il faut donc évidemment traiter l'élément `expr[i]`. Il y a plusieurs possibilités :

- `expr[i]` est une variable ;
- `expr[i]` est un opérateur unaire ;
- `expr[i]` est un opérateur binaire ;
- `expr[i]` est autre chose.

Examinons ces différentes possibilités :

`expr[i]` est une variable . De l'Invariant, on sait qu'il existe un n tel que :

$$expr[0..i-1] = \mathop{||}_{j=1}^{n} \text{EPost}_j$$

Puisque `expr[i]` est une variable (et donc une EPost), il existe aussi un n' tel que :

$$expr[0..i] = \mathop{||}_{j=1}^{n'} \text{EPost}_j$$

L'$\text{EPost}_{n'}$ n'est autre que `expr[i]`. Si nous voulons faire progresser le programme, que nous indique l'Invariant ? Clairement que la valeur de $\text{EPost}_{n'}$ doit être empilée.

```
1 if(variable(expr[i]){
2     // {∃n, expr[0..i] = ||_{j=1}^{n} EPost_j ∧ EPost_n = expr[i], stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_{n-1})]⌉}
3     stack = push(stack, valeur(expr[i]));
4     // {Post_eval ∧ Post_push ⇒ ∃n, expr[0..i] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]⌉}
5 }
```

5. Pour les opérations disponibles sur une Pile, reportez-vous au rappel.

`expr[i]` est un opérateur unaire De l'Invariant, on sait que :

$$\exists n, expr[0..i-1] = \Big\|_{j=1}^{n} \text{EPost}_j, stack = \lfloor Eval(\text{EPost}_1) \uparrow \ldots \uparrow [Eval(\text{EPost}_n)]]$$

On forme une nouvelle EPost avec l'opérateur unaire. L'Invariant nous informe que la valeur de son opérande est sur le sommet de la Pile. On dépile donc cette valeur et on rempile la valeur de l'EPost formée avec l'opérateur unaire.

```
if(unaire(expr[i]){
    // {∃n, expr[0..i − 1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]]}
    op1 = top(stack);
    // {Post_top ⇒
    //     (∃n, expr[0..i − 1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]] ∧ op1 = Eval(EPost_n))}
    stack = pop(stack);
    // {Post_pop ⇒
    //     (∃n, expr[0..i − 1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_{n−1})]] ∧ op1 = Eval(EPost_n))}
    stack = push(stack, eval_unaire(expr[i], op1));
    // {Post_eval ∧ Post_push ⇒ ∃n, expr[0..i] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]]}
}
```

`expr[i]` est un opérateur binaire On effectue globalement la même chose que pour un opérateur binaire sauf qu'il faut dépiler deux opérandes. Attention, la seconde opérande est au-dessus de la première dans la Pile.

```
if(operateur(expr[i]){
    // {∃n, expr[0..i − 1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]]}
    op2 = top(stack);
    // {Post_top ⇒
    //     (∃n, expr[0..i − 1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]] ∧ op2 = Eval(EPost_n))}
    stack = pop(stack);
    // {Post_pop ⇒
    //     (∃n, expr[0..i − 1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_{n−1})]] ∧ op2 = Eval(EPost_n))}
    op1 = top(stack);
    // {Post_top ⇒
    //     (∃n, expr[0..i − 1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_{n−1})]]
    //                                                 ∧ op2 = Eval(EPost_n) ∧ op1 = Eval(EPost_{n−1}))}
    stack = pop(stack);
    // {Post_pop ⇒
    //     (∃n, expr[0..i − 1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_{n−2})]]
    //                                                 ∧ op2 = Eval(EPost_n) ∧ op1 = Eval(EPost_{n−1}))}
    stack = push(stack, eval(expr[i], op1, op2));
    // {Post_eval ∧ Post_push ⇒ ∃n, expr[0..i] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]]}
}
```

`expr[i]` est autre chose Cela correspond à une erreur de format dans la chaine `expr`. Ce n'était pas demandé par l'énoncé mais s'il fallait traiter ce genre de problème, ce serait ici.

7.7.1.5 END

Il faut voir quelle informations sont à notre disposition en combinant l'Invariant et la condition d'arrêt (i.e. $\{Inv \wedge \neg B\}$)

$$\neg B \Rightarrow expr = expr[0..i-1]$$

$$Inv \wedge \neg B \Rightarrow \left(\exists n, expr = \mathop{||}_{j=1}^{n} \text{EPost}_j, stack = \lfloor Eval(\text{EPost}_1) \uparrow \ldots \uparrow [Eval(\text{EPost}_n)] \rceil \right)$$

Puisque l'expression *expr* est elle-même une EPost, ce n existe bien et doit valoir 1. Il vient donc :

$$expr = \mathop{||}_{j=1}^{1} \text{EPost}_1$$

L'évaluation de cette EPost est sur la Pile, c'est garanti par l'Invariant :

$$stack = \lfloor [Eval(\text{EPost}_1)] \rceil = \lfloor [Eval(expr)] \rceil$$

Il faut donc retourner la valeur stockée au sommet de la Pile. Il ne faut pas oublier de détruire proprement celle-ci.

```
1 // {¬B ⇒ expr = expr[0..i − 1]}
2 // {Inv ∧ ¬B → (∃n, expr = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]⌉) ⇒ Inv}
3 // {EPost = expr_0 = expr ⇒ (expr = ||_{j=1}^{1} EPost_j ∧ (stack = ⌊[Eval(EPost_1)]⌉ = ⌊[Eval(expr)]⌉))}
4 int result = top(stack);
5 // {result = Eval(expr)}
6 free_stack(&stack);
7 return result;
8 // {Post}
```

7.7.1.6 Fonction de terminaison

Ici, il n'y en a pas à proprement parler : si un petit malin passe une chaine de caractères mal formatée au programme, rien ne permet de garantir que le programme s'arrête. En bonne pratique, lorsque l'on opère sur les chaines de caractères, on passe toujours une longueur maximale de chaine au-delà de laquelle on arrête les opérations (Voir toutes les fonctions de la glibc dont le nom commence par « `strn...` » plutôt que « `str...` ». Si on avait introduit une telle longueur maximale de chaine appelée `L`, la fonction de terminaison serait :

$$t = L - i$$

Dans un monde théorique où la chaine `expr` est correctement formatée, $L = L_{expr} - 1$ montre que le code se terminera toujours dans le cas d'un formatage correct.

Le programme final est disponible à la Sec. 7.8.

7.8 Programme final

```
#include"stack.h"

int eval(char *expr){
   assert(expr);
   // {Pre}
   unsigned int i = 0; // Signé ou non est OK
   // {expr[0..i-1] = "" ⇒ expr[0..i-1] = ||_{j=1}^{0} EPost_j}
   Stack *stack = empty_stack();
   // {stack = ⌊[  ]⌉ }
   // {⇒ (∃n, expr[0..i-1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]⌉) ⇒ Inv}

  /* Variables de travail */
   int op1; // opérande 1
   int op2; // opérande 2

   // {Inv}
   while(expr[i]){
      // Inv ∧ B
    if(variable(expr[i]){
         // {∃n, expr[0..i] = ||_{j=1}^{n} EPost_j ∧ EPost_n = expr[i], stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_{n-1})]⌉}
         stack = push(stack, valeur(expr[i]));
         // {Post_eval ∧ Post_push ⇒ ∃n, expr[0..i] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]⌉}
      }
      else if(unaire(expr[i]){
         // {∃n, expr[0..i-1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]⌉}
         op1 = top(stack);
         // {Post_top ⇒
         //    (∃n, expr[0..i-1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]⌉ ∧ op1 = Eval(EPost_n))}
         stack = pop(stack);
         // {Post_pop ⇒
         //    (∃n, expr[0..i-1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_{n-1})]⌉
         //                                                                  ∧ op1 = Eval(EPost_n))}
         stack = push(stack, eval_unaire(expr[i], op1));
         // {Post_eval ∧ Post_push ⇒ ∃n, expr[0..i] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]⌉}
      }
      else if(operateur(expr[i]){
         // {∃n, expr[0..i-1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]⌉}
         op2 = top(stack);
         // {Post_top ⇒
         //    (∃n, expr[0..i-1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]⌉ ∧ op2 = Eval(EPost_n))}
         stack = pop(stack);
         // {Post_pop ⇒
         //    (∃n, expr[0..i-1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_{n-1})]⌉
         //                                                                  ∧ op2 = Eval(EPost_n))}
         op1 = top(stack);
         // {Post_top ⇒
         //    (∃n, expr[0..i-1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_{n-1})]⌉
         //                                        ∧ op2 = Eval(EPost_n) ∧ op1 = Eval(EPost_{n-1}))}
```

```
49          stack = pop(stack);
50          // {Post_pop ⇒
51          //     (∃n, expr[0..i − 1] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_{n−2})]]
52          //                                                  ∧ op2 = Eval(EPost_n) ∧ op1 = Eval(EPost_{n−1}))}
53          stack = push(stack, eval(expr[i], op1, op2));
54          // {Post_eval ∧ Post_push ⇒ ∃n, expr[0..i] = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]]}
55      }
56  }
57  // {¬B ⇒ expr = expr[0..i − 1]}
58  // {Inv ∧ ¬B → (∃n, expr = ||_{j=1}^{n} EPost_j, stack = ⌊Eval(EPost_1) ↑ ... ↑ [Eval(EPost_n)]]) ⇒ Inv}
59  // {EPost = expr_0 = expr ⇒ (expr = ||_{j=1}^{1} EPost_j ∧ (stack = ⌊[Eval(EPost_1)]] = ⌊[Eval(expr)]]))}
60  int result = top(stack);
61  // {result = Eval(expr)}
62  free_stack(&stack);
63  return result;
64  // {Post}
65 }
```

La trace d'exécution pour l'EPost $A\ B\ \times\ C\ D\ +\ /$ est disponible à la Sec. 7.9.

7.9 Trace d'exécution

Soient les variables (A, B, C, D) et l'environnement $(20, 4, 9, 7)$. Indiquez aussi la trace de votre algorithme, sur votre pile, pour l'expression $A\ B\ \times\ C\ D\ +\ /$.