# TEA AAP - TP2 2022

Types Abstraits de Données, Piles et Listes chaînées
Résolution du jeu “Le compte est Bon”

Ce TEA fait suite au TP2. Il doit être réalisé en repartant de la situation correspondant à la fin du TP2. Le projet est alors constitué des fichiers suivants :

- ***elt.h*** et ***elt.c*** : module de gestion d’éléments génériques.
  - Définition du type `T_elt`.
  - Trois constantes symboliques permettent de sélectionner le type `T_elt` : `ELT_CHAR, ELT_INT, ELT_STRING`.
    - Il faut en déclarer une seule dans le fichier ***elt.h.***
  - Deux fonctions : `T_elt genElt(void)` et `char * toString(T_elt e)`, permettent de créer un élément et de renvoyer une chaîne de caractères le décrivant.
- ***stack_cs.h*** et ***stack_cs.c*** : module de gestion de piles.
  - Utilise une implémentation contigüe statique.
  - Utilise le type `T_elt` sélectionné dans ***elt.h***.
- ***stack_cd.h*** et ***stack_cd.c*** : module de gestion de piles.
  - Utilise une implémentation contigüe dynamique.
  - Utilise le type `T_elt` sélectionné dans ***elt.h***.
- ***list.h*** et ***list.c*** : module de gestion de listes simplement chaînées.
  - Définition des types `T_node` et `T_list` (`T_list` ⇔ pointeur sur `T_node`)
    - Ces types utilisent le type `T_elt`.
  - Définition de plusieurs fonctions de traitement de listes : création, libération, chaînage, affichage.
- ***stack_cld.h*** et ***stack_cld.c*** : module de gestion de piles.
  - Utilise des listes chaînées linéaires dynamiques gérées par le module précédent.
  - Utilise le type `T_elt` sélectionné dans ***elt.h***.
- ***main.c*** : point d’entrée du programme.
- ***makefile*** et ***makefile_sources*** : un fichier makefile générique permettant la compilation du projet
  - Le fichier ***makefile_sources*** permet de sélectionner les fichiers à compiler sans avoir besoin de retoucher le fichier makefile.

## Organisation

- Ce TEA doit être réalisé en équipe. Chaque équipe doit être constituée au maximum de 4 étudiants, appartenant au même groupe TD. Il ne doit pas y avoir plus de 8 équipes par groupe TD. Les équipes doivent rester les mêmes pour le TEA2, le TEA3 et le fil rouge.
- Chaque équipe rendra une seule archive sur moodle, contenant :
  - Les fichiers sources du programme produit ;
  - Un fichier makefile permettant de compiler le programme
  - Un CR rédigé (avec introduction, développement, conclusion, perspectives et bibliographie) présentant l’organisation du programme et celle du groupe (qui a fait quoi, avec quel planning, quelles ont été les difficultés, etc.)

# Objectif : programme de résolution du jeu “le compte est bon”[1]

Nous nous intéressons dans ce TP à la recherche de solutions pour le jeu «  le compte est bon », toujours diffusé dans le cadre de jeu télévisé « des chiffres et des lettres ». C'est le plus ancien jeu télévisé quotidien toujours diffusé de la télévision française. Nous souhaitons mettre en œuvre une recherche exhaustive des solutions, pour garantir que la solution trouvée est bien la plus proche possible du nombre cherché.

***Règles du jeu “Le compte est bon”***

Le jeu télévisé commence par le tirage au sort de 6 « cartons » choisis aléatoirement parmi les valeurs suivantes : 1;2;3;4;5;6;7;8;9;10;25;50;75;100. Le nombre à trouver doit être compris entre 100 et 999. Seules les opérations usuelles sont possibles : addition, soustraction, multiplication et division. Lors des calculs, tous les résultats intermédiaires doivent être des entiers naturels.

***Fourniture du problème au programme et affichage du résultat***

Notre programme prendra en arguments les 6 cartons tirés au sort suivis du nombre à trouver. Par exemple, si les cartons tirés sont 3 5 7 9 25 50 et que le nombre à découvrir est 859, il devra être appelé ainsi :

***./compteestbon.exe 3 5 7 9 25 50 859***

Le programme devra afficher les opérations à réaliser le plus vite possible, à défaut de toute autre chose, à raison d’une opération par ligne, dans laquelle opérandes et opérateurs seront séparés par un espace, de la manière suivante :

***25 + 7 = 32***
***32 x 9 = 288***
***288 x 3 = 864***
***864 - 5 = 859***

- Un seul ensemble d’opérations devra être affiché, même s’il existe plusieurs manières de parvenir au résultat.
- Au cas où la réponse exacte n’aurait pas été trouvée, le programme devra afficher le calcul permettant de s’approcher le plus possible du nombre recherché.
- Le programme ne devra pas afficher d’opérations inutiles, ne participant pas à la fourniture du résultat final. Ainsi, le dernier nombre situé sur la dernière ligne affichée (ci-dessus, 859), sera considéré comme le nombre le plus proche trouvé par le programme.
- Les éventuelles traces d’exécution permettant le débogage du programme devront être générées exclusivement sur la ***sortie d’erreur.***

Un script shell fourni avec ce sujet (script.sh) pourra être utilisé afin de vérifier que la sortie de votre programme répond bien aux contraintes du sujet. Seuls les programmes vérifiant les tests de ce script participeront au challenge organisé pour ce TEA permettant de classer les programmes par vélocité.

[1] Description tirée du site https://fr.wikipedia.org/wiki/Des_chiffres_et_des_lettres.

# Partie 1 : Evaluation d'expressions en Notation Polonaise Inverse[2]

La notation polonaise inverse (en anglais RPN pour Reverse Polish Notation), également connue sous le nom de notation post-fixée, permet d'écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses. Dérivée de la notation polonaise présentée en 1924 par le mathématicien polonais Jan Łukasiewicz, elle s'en différencie par l'ordre des termes, les opérandes y étant présentés avant les opérateurs et non l'inverse.

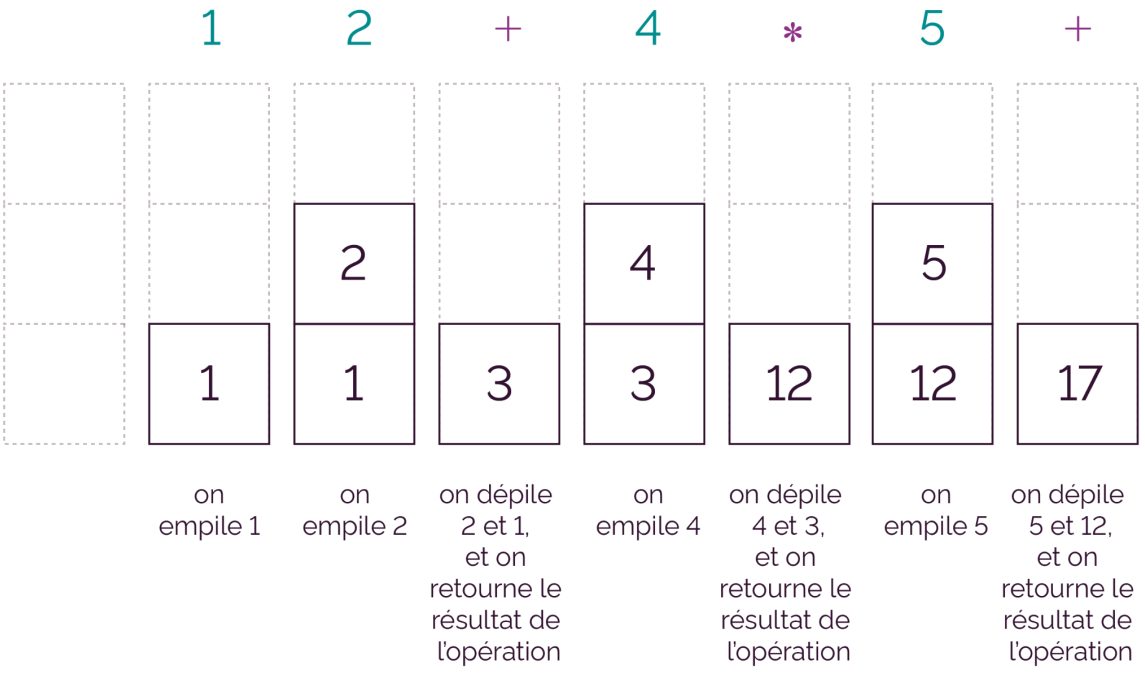
Par exemple, le calcul ***(1+2) * 4 + 5*** se représente en RPN par l'expression ***1 2 + 4 * 5 +.***

L'évaluation d'une expression RPN nécessite l'utilisation d'une ***pile d'évaluation***, qui va servir à enregistrer les opérandes rencontrées et stocker les résultats intermédiaires. L'algorithme d'évaluation d'une expression RPN s'exprime ainsi :

- Tant qu'il reste des éléments dans l'expression à évaluer :
  - Lire le prochain élément de l'expression ;
  - Si l'élément lu est un opérande, alors l'empiler ;
  - Sinon, (l'élément lu est un opérateur) :
    - dépiler deux opérandes de la pile ;
    - réaliser un calcul en appliquant l'opérateur sur les deux opérandes dépilés ;
    - empiler le résultat de ce calcul.

Une fois tous les caractères lus, la pile ne contient qu'un seul élément qui correspond à la valeur de l'expression RPN à évaluer.

La figure ci-dessous présente l'exécution de cet algorithme pour l'expression ***1 2 + 4 * 5 +*** et le contenu de la pile après la lecture de chaque caractère.




[2] Descriptions tirées des sites https://fr.wikipedia.org/wiki/Notation_polonaise_inverse et https://www.maxicours.com/se/cours/utiliser-une-pile-pour-evaluer-une-notation-en-polonais-inverse/.

***Travail demandé***

- Ajouter au module de gestion de T_elt un type ELT_RPN permettant de représenter les éléments extraits d'une expression RPN ainsi les résultats possibles d'une évaluation de RPN. Ce type T_elt devra contenir :
    - un champ value (entier) représentant un résultat ou un calcul intermédiaire
    - ainsi qu'un champ status permettant de représenter la nature du T_elt : résultat entier, résultat non entier, expression valide, expression invalide, opérateurs…
        - Les valeurs possibles du champ status devront être définies sous forme de constantes symboliques dans le fichier elt.h.
- Développer le module d'évaluation d'expressions RPN comportant les fichiers rpn.h et rpn.c. Ce module devra implémenter au minimum :
    - la fonction T_List s2list(char * exp) qui transforme une expression RPN sous forme de chaîne de caractères en une liste de T_elt.
    - la fonction T_elt rpn_eval(char * exp) qui évalue une expression RPN,
    - ainsi que toutes les fonctions que vous jugerez nécessaires pour gérer les RPN et leurs affichages, notamment l'affichage sous la forme demandée par le programme dans la description des objectifs.
- Une constante symbolique située dans rpn.h devra permettre de sélectionner le type d'implémentation choisie pour les piles d'évaluation, afin de pouvoir étudier l'impact de ce choix sur la vitesse de votre programme. Vous rendrez le programme le plus performant possible et justifierez le choix mis en oeuvre dans votre CR.

# Partie 2 : Exploration d'un arbre à la volée

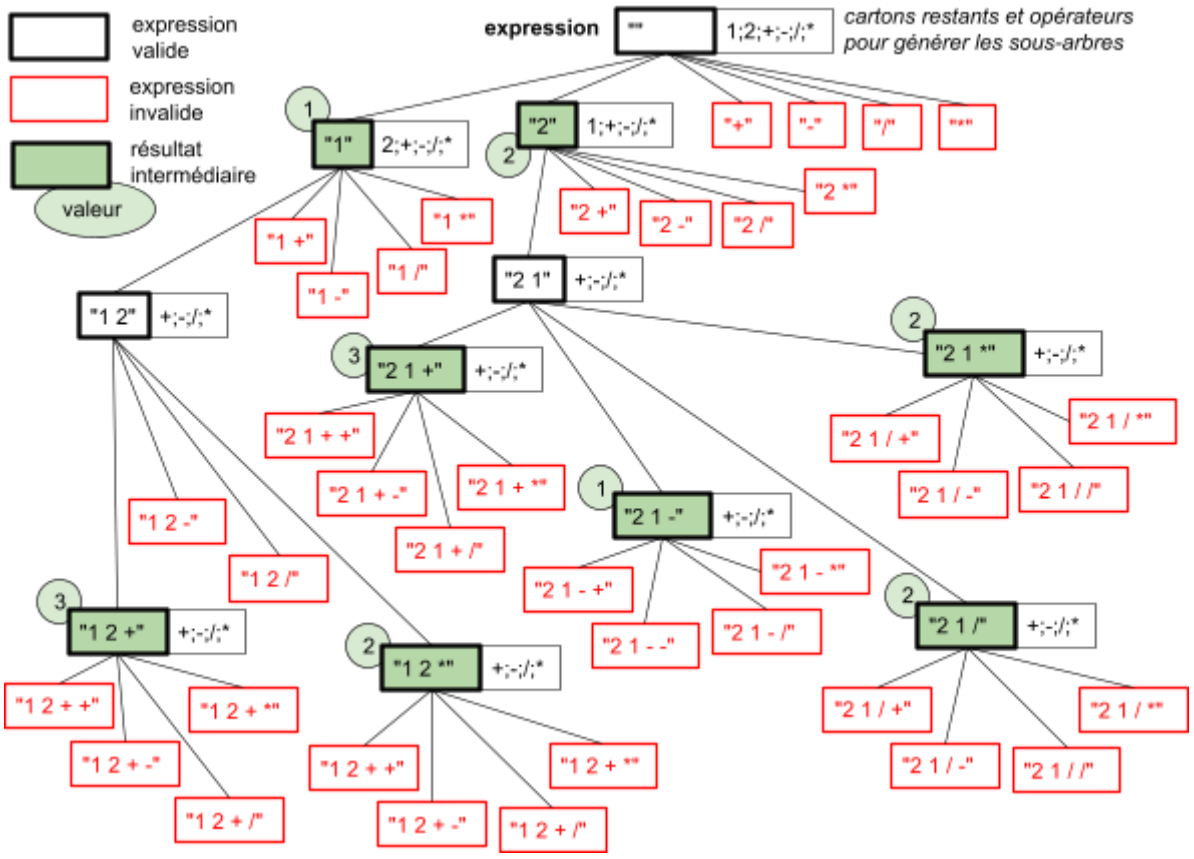
Pour résoudre le problème du compte est bon, nous souhaitons construire toutes les expressions possibles formées à partir des 6 cartons de départ et des opérateurs arithmétiques +,-,*,/. Pour cela, nous mettons en œuvre une méthodologie de recherche ***à la volée en profondeur d'abord***.

Chaque carton ne pouvant être utilisé qu'une seule fois, nous développons une boucle qui parcourt les cartons restants et les 4 opérateurs possibles (qui – au contraire des cartons - peuvent être réutilisés à chaque fois). Pour chaque opérateur ou opérande, nous l'ajoutons à l'expression en cours de construction et appelons récursivement la procédure de recherche en lui passant cette nouvelle expression ainsi que les cartons restants.

Au début de chaque appel récursif, nous testons si l'expression passée en paramètre est valide de manière à ne pas parcourir inutilement le sous-arbre correspondant, et éviter par exemple de produire des expressions construites uniquement à partir de suites d'opérateurs. Lorsqu'elle est valide, et qu'elle permet de produire un résultat qui améliore le meilleur résultat obtenu jusque-là, on enregistre l'expression dans notre programme pour pouvoir l'afficher lorsque la recherche sera terminée. Lorsqu'elle est valide et qu'elle permet d'obtenir le nombre demandé, on arrête la procédure et on affiche les calculs menant au résultat exact.

Parcourir à la volée signifie que nous n'avons pas besoin de stocker en mémoire l'ensemble des expressions possibles, c'est la ***pile d'exécution de notre processus***, qui contient les arguments passés en paramètre des appels récursifs, qui nous sert de stockage temporaire.

Cette recherche s'apparente au parcours d'un arbre comme le montre la figure suivante avec les cartons valant 1 et 2 :




Le parcours en profondeur est pertinent dans la mesure où nos expressions ont une taille maximale connue, puisqu'on ne peut assembler au maximum que 6 opérandes et 5 opérateurs, soit une profondeur maximale de 11 appels récursifs.

***Travail demandé***

- Dans votre CR :
  - Choisir la manière de représenter les cartons restant à utiliser dans les appels à la fonction de parcours à la volée en profondeur d'abord
  - Ecrire le prototype de cette fonction
  - Ecrire l'algorithme permettant d'implémenter cette fonction
  - Sur le schéma précédent, déterminer dans quel ordre les noeuds de l'arbre seront parcourus par votre algorithme en ajoutant pour chacun son indice dans le parcours
- Utiliser les modules développés précédemment pour implémenter la fonction précédente et permettre la résolution d'un problème du compte est bon, en respectant les contraintes indiquées dans les objectifs du sujet.