École Centrale de Lille

AAP - Algorithmique Avancée et Programmation

S5 - TEA 2

Groupe D:

BASTOS Vitória
FERREIRA BRASIL Rebeca
LESSA GUERRA Bernardo
REIS GUEVARA David

1. Introduction

Le travail a été divisé en deux parties et son but ultime est de créer un programme capable de jouer au jeu “le compte est bon”.

La première partie vise à créer un programme capable d'écrire et d'interpréter une suite d'opérations mathématiques en utilisant la notation polonaise inverse. La logique principale de l’algorithme est d'évaluer une expression polonaise inverse en suivant ces étapes :

- 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.

La deuxième partie vise à créer un algorithme capable de résoudre le jeu “le compte est bon” à partir de 6 cartons de départ, un résultat et des opérateurs arithmétiques autorisés dans le jeu. À la fin, l'algorithme doit être capable de trouver le résultat attendu de la manière la plus rapide.

Ce compte rendu contient non seulement les codes qui ont été conçus pour effectuer le TEA, mais nous avons également rassemblé l'explication de leur logique, en nous concentrant sur les deux parties demandées, avec la partie RPN et la fonction *main*.

2. Développement

Le développement a été divisé en deux parties complémentaires, mais qui ont une logique d’application tout à fait particulière à chacun pour les fonctions qu'elles remplissent. La première partie donne le rationnel pour la Notation Polonaise Inverse et la deuxième partie une application directe au jeu “Le Compte est Bon” en utilisant une exploration d’un arbre à la volée.

2.1. Partie 1 - Notation Polonaise Inverse

Pour faire la fonction qui effectue le calcul à partir d'une expression en notation polonaise inverse (ou "Reverse Polish Notation - RPN", en anglais), on a besoin de fonctions auxiliaires qui étaient également utilisées dans le TD 2, qui seront introduites ci-dessous avant la fonction RPN.

2.1.1. Codes support fournis

- ***elt.h*** et ***elt.c*** : Utilisés pour définir de type T_Elt avec les fonctions `T_elt genElt(void)` et `char * toString(T_elt e)`, comme réquis.
    - elt.c:

```c
#include <stdio.h>  // sprintf
#include <string.h> // strdup

//#define CLEAR2CONTINUE
#include "include/traces.h"

#include "elt.h"

////////////////////////////////////////////////////////////////////////////
//
// Type T_elt
// char * toString(T_elt) :
//  Renvoie une chaîne de caractère représentant un T_elt pour l'afficher
// T_elt genElt(void) :
//  Génère un nouveau T_elt différent du précédent
//  (utiliser une variable statique)
////////////////////////////////////////////////////////////////////////////

#ifdef ELT_CHAR
// T_elt est un char...
char *toString(T_elt e) {
  // Il faut transformer un char en chaine...
  static char buffer[2];
  buffer[0] = e;
  buffer[1] = '\0';
  return buffer;
  // Si on allouait de la mémoire pour buffer, il faudrait penser à la
  // libérer...
  // => Risque de fuite de mémoire...
  // On ne peut pas non plus allouer un tableau static sur la pile !
  // => On utilise un buffer déclaré comme variable statique !
  // Dans ce cas, deux appels à toString renverraient la même adresse...
}

T_elt genElt(void) {
  static int indice = 0;
  // on va égrainer les caractères alphabétiques de 'a' à 'z'
  // de manière circulaire pour ne pas déborder...
  return 'a' + (indice++ % 26);
}
#endif

#ifdef ELT_INT
// T_elt est un int...
char *toString(T_elt e) {
  // Il faut transformer un int en chaine...
  static char buffer[11]; // nbr max de chiffres nécessaires ?
  // 2^31 vaut 2147483648...
  sprintf(buffer, "%d", e);
  return buffer;
  // cf. remarques précédentes
}

T_elt genElt(void) {
  static int indice = 0;
  return indice++;
}
#endif

#ifdef ELT_STRING
// T_elt est un char * ...
char *toString(T_elt e) {
  return e; // c'est déjà une chaine !
}

T_elt genElt(void) {
  static int indice = 0;
  // produire une chaine aléatoire...
  // On va écrire elt_<numéro>
  char buffer[15]; // elt_+<11 chiffres>
  sprintf(buffer, "elt_%d", indice++);
  return strdup(buffer);
}
#endif

#ifdef ELT_RPN
```

```c
char * toString(T_elt e) {
  static char buffer[100];
  sprintf(buffer, "v: %d, s: %c", e.result, e.status);
  return buffer;
}

T_elt genElt(void) {
  T_elt aux;
  aux.result = 0;
  aux.status = 'z';
  return aux;
}
#endif

T_elt eltdup(T_elt e) {
  T_elt aux;
  aux.result = e.result;
  aux.status = e.status;
  return aux;
}

/*static char buffer[2];
      buffer[0] = e;
      buffer[1] = '\0';
      return buffer; */

////////////////////////////////////////////////////
```

- ***stack_cs.h*** et ***stack_cs.c*** : Pour le cas d'une implémentation contigüe statique et pour la gestion de piles.
    - stack_cs.c:

```c
#include <assert.h> // assert

//#define CLEAR2CONTINUE
#include "include/traces.h"

#include "stack_cs.h" // type et protypes liés à T_stack

#ifdef IMPLEMENTATION_STATIC_CONTIGUOUS

void showStack (const T_stack * p) {
  int i;
  // Afficher la pile
  // On affiche en commençant par le sommet de la pile

  if (p->sp ==0) {
    printf("Pile vide !\n"); // pile vide !
    return;
  }

  printf("Sp = %d\n",p->sp);
  for(i=p->sp-1;i>=0; i--) {
    printf("P[%d]= %s\n",i, toString(p->data[i]));
  }
}

void emptyStack (T_stack *p) {
  // Vider la pile
  p->sp = 0;
}

T_stack newStack(void) {
  // Créer une pile vide
  T_stack s = {0};
  return s;
}

// Nouvelles fonctions à réaliser pour l'exercice 3

T_stack exampleStack(int n) {
  // Renvoie une nouvelle pile d'exemple, contenant n éléments
```

```
41    int i;
42    T_stack s;
43
44    assert(n<=STACK_NBMAX_ELT);
45
46    s = newStack();
47    for(i=0;i<n;i++) {
48      push(genElt(),&s);
49    }
50
51    return s;
52  }
53
54  T_elt pop(T_stack *p) {
55    // Cette opération extrait de la pile, l'élément au sommet,
56    // modifiant ainsi son état
57    assert(p != NULL);
58    assert(p->sp > 0);
59    return p->data[--p->sp];
60    // D'abord on décrémente, puis on regarde ce qui se trouve à cet endroit...
61    // C'est ce que l'on renvoie
62  }
63
64  T_elt top(const T_stack *p) {
65    // Cette opération permet d'accéder à l'élément en sommet de pile
66    assert(p != NULL);
67    assert(p->sp > 0);
68    return p->data[p->sp-1];
69  }
70
71  void push(T_elt e, T_stack *p) {
72    // Opération consistant à empiler un élément e
73    // sur la pile modifiant ainsi son état
74    assert(p!=NULL);
75    p->data[p->sp++] = e;
76  }
77
78  int isEmpty (const T_stack * p) {
79    // Cette opération permet de tester si la pile est vide
80
81    assert(p!=NULL);
82    if (p->sp == 0) return 1;
83    else return 0;
84  }
85
86  #endif
```

- ***stack_cd.h*** et ***stack_cd.c*** : Pour le cas d'une implémentation contigüe statique et pour la gestion de piles.
    - stack_cd.c:

```c
#include <assert.h> // assert

//#define DEBUG
#include "include/check.h"

//#define CLEAR2CONTINUE
#include "include/traces.h"

#include "stack_cd.h" // type et protypes liés à T_stack

#ifdef IMPLEMENTATION_DYNAMIC_CONTIGUOUS
#define STACK_THRESHOLD 5

/*
typedef struct {
  int sp;
  int nbMaxElt;
  T_elt * data;
} T_stack;
*/

T_stack newStack(int size) {
  // Opération permettant d'allouer la mémoire pour une pile dynamique
  T_stack s;
  s.nbMaxElt = size;
  s.sp = 0;
  CHECK_IF(s.data = malloc(size * sizeof(T_elt)), NULL, "malloc newStack");
  return s;
}

void freeStack (T_stack *p) {
  // Libère la pile en désallouant la mémoire
  p->sp = 0;
  p-> nbMaxElt = 0;
  free(p->data);
  p->data = NULL;
}

void push(T_elt e, T_stack *p) {
  // Opération consistant à empiler un élément T
```

```
  // sur la pile modifiant ainsi son état
  // BONUS : Prévoir des seuils d'agrandissement
  // pour éviter des realloc systématiques
  // STACK_THRESHOLD
  assert(p!=NULL);
  if (p->nbMaxElt == p->sp) {
    printf("Allocation nécessaire ! \n");
    CHECK_IF(p->data = realloc(p->data, (p->nbMaxElt+STACK_THRESHOLD) * sizeof(T_elt)), NULL, "realloc newStack");
    p->nbMaxElt += STACK_THRESHOLD;
  }
  p->data[p->sp++] = e;
}

// A Adapter (déjà fait) ////////////////////////////////////////////////////////////

T_stack exampleStack(int n) {
  // Renvoie une nouvelle pile d'exemple, contenant n éléments
  T_stack s;
  int i;

  s = newStack(n);
  for(i=0;i<n;i++) {
    push(genElt(),&s);
  }

  return s;
}

// RIEN NE CHANGE ///

void showStack (const T_stack * p) {
  int i;
  // Produire une fonction d'affichage de la pile
  // On affiche en commençant par le sommet de la pile

  if (p->sp ==0) {
    printf("Pile vide !\n"); // pile vide !
    return;
  }

  printf("Sp = %d\n",p->sp);
  for(i=p->sp-1;i>=0; i--) {
    printf("P[%d]= %s\n",i, toString(p->data[i]));
  }
}

T_elt pop(T_stack *p) {
  // Cette opération extrait de la pile, l'élément au sommet,
  // modifiant ainsi son état
  assert(p != NULL);
  assert(p->sp > 0);
  return p->data[--p->sp];
  // D'abord on décrémente, puis on regarde ce qui se trouve à cet endroit...
  // C'est ce que l'on renvoie
}

T_elt top(const T_stack *p) {
  // Cette opération permet d'accéder à l'élément en sommet de pile
  assert(p != NULL);
  assert(p->sp > 0);
  return p->data[p->sp-1];
}

void emptyStack (T_stack *p) {
  // Vide la pile
  p->sp = 0;
}

int isEmpty (const T_stack * p) {
  // Cette opération permet de tester si la pile est vide.
  assert(p!=NULL);
  if (p->sp == 0) return 1;
  else return 0;

}

#endif
```

- ***list.h*** et ***list.c*** : Pour la définition des types `T_node` et `T_list`. Définition de plusieurs fonctions de traitement de listes : création, libération, chaînage, affichage.
    - list.c:

```c
#include <assert.h>

//#define CLEAR2CONTINUE
#include "include/traces.h"

//#define DEBUG
#include "include/check.h"

#include "elt.h" // T_elt
#include "list.h" // prototypes

/*
typedef struct node {
  T_elt data;
  struct node *pNext;
} T_node, * T_list;
*/



static T_node * newNode(T_elt e) {
  // Créer une nouvelle cellule contenant l'élément e

  T_node * pNode;
  CHECK_IF(pNode = malloc(sizeof(T_node)), NULL, "malloc allocateNode");
  pNode->data = e;
  pNode->pNext = NULL;

  return pNode;
}

T_node * addNode (T_elt e, T_node * n) {
  // Créer une maille (node), la remplir
  // et l'accrocher en tête d'une liste existante (ou vide)
  // Renvoyer la nouvelle tête

  T_node * pNode;
  pNode = newNode(e);
  pNode->pNext = n;
  return pNode;
}

void showList(T_list l) {
  // Afficher la liste en commençant par sa tête
  // A faire en itératif

  if (l==NULL) {
    printf("Liste Vide !\n");
    return;
  }

  while(l != NULL) {
    printf("%s ->", toString(l->data));
    l = l->pNext;
  }
}

void freeList(T_list l) {
  // Libérer la mémoire de toutes les cellules de la liste l
  // A faire en itératif

  T_node * pAux;
  // Il faut un pointeur auxiliaire :
  // on ne doit libérer qu'après avoir enregistré quelque part
  // l'adresse de la maille suivante

  assert(l != NULL);

  while(l != NULL) {
    printf("Libération de %s\n", toString(l->data));
    pAux = l->pNext;
    free(l);
    l = pAux;
  }
}

T_elt getFirstElt(T_list l) {
  // Renvoyer l'élément contenu dans la cellule de tête de la liste

  assert(l != NULL);

  return l->data;
}

T_list removeFirstNode(T_list l) {
  // Supprimer la tête de la liste
  // Renvoyer la nouvelle liste privée de sa première cellule

  assert(l!= NULL);
  T_node * p =l->pNext;
  free(l);
  return p;
}

// A produire en version récursive (+ tard dans le sujet)

void showList_rec(T_list l) {
  // Afficher la liste en commençant par sa tête
  // A faire en récursif

  if (l == NULL) {
    // case de base
    return;
  } else {
    // cas général
    printf("%s -> ", toString(l->data));
    showList_rec(l->pNext);
  }
}

void showList_inv_rec(T_list l) {
  // Afficher la liste en commençant par sa queue
```

```c
  // A faire en récursif

  if (l == NULL) {
    // case de base
    return;
  } else {
    // cas général
    showList_inv_rec(l->pNext);
    printf("%s <-", toString(l->data));
  }
}

void freeList_rec(T_list l) {
  // Libérer la mémoire de toutes les cellules de la liste l
  // A faire en récursif

  if (l == NULL) {
    return;
  } else {
    freeList_rec(l->pNext);
    printf("Libération de %s\n",toString(l->data));
    free(l);
  }
}

T_list tailAddNode(T_elt e, T_list l){

    if(l!=NULL){
        T_list aux = l;
        while(aux->pNext!=NULL){
            aux = aux->pNext;
        }

        aux->pNext = newNode(e);
        return l;
    }
    else return newNode(e);
}

unsigned int getSize(const T_list l){

    int size = 0;
    T_list aux = l;
    while(aux!=NULL){
        size++;
        aux = aux->pNext;
    }

    return size;
}
```

- ***stack_cld.h*** et ***stack_cld.c*** : Module de gestion de piles en utilisant les listes.
    - stack_cld.c:

```c
#include <assert.h> // assert

//#define DEBUG
#include "include/check.h"

//#define CLEAR2CONTINUE
#include "include/traces.h"

#include "stack_cld.h" // type et protypes liés à T_stack

// typedef T_node * T_stack;

#ifdef IMPLEMENTATION_DYNAMIC_LINKED

T_stack newStack() {
  // Créer une pile vide

  return NULL;
}

void freeStack (T_stack *p) {
  // Libérer la mémoire associée à une pile

  freeList(*p);
}

void push(T_elt e, T_stack *p) {
  // Opération consistant à empiler un élément e
  // sur la pile modifiant ainsi son état

  *p = addNode(e, *p);
}

void showStack (const T_stack * p) {
  // Afficher la pile
  // On affiche en commençant par le sommet de la pile

  showList(*p);
}

```

```c
T_elt pop(T_stack *p) {
  // Cette opération extrait de la pile, l'élément au sommet,
  // modifiant ainsi son état

  T_elt e = getFirstElt(*p);
  (*p) = removeFirstNode(*p);
  return e;
}

T_elt top(const T_stack *p) {
  // Cette opération permet d'accéder à l'élément en sommet de pile

  return getFirstElt(*p);
}

void emptyStack (T_stack *p) {
  // Vider la pile

  freeStack (p);
  *p=NULL;
}

int isEmpty (const T_stack * p) {
  // Cette opération permet de tester si la pile est vide

  return ((*p)==NULL);
}

#endif


```

- ***makefile*** et ***makefile_sources***

makefile:

```
PWD=$(shell pwd)
REP=$(shell basename $(PWD))
SOURCES=$(shell cat makefile_sources)
CIBLE=$(REP).exe
CFLAGS=-Wall

# makefile générique pour produire un code source
# dont le nom correspond au nom du répertoire qui le contient

all: $(CIBLE)
  @echo "Le programme $(CIBLE) a été produit dans le répertoire $(REP)"

$(CIBLE) : $(SOURCES)
  @echo -n "Production de $(CIBLE)"
  @echo " à partir des fichiers : $(SOURCES)"
  gcc $(CFLAGS) $(SOURCES) -o $@

clean:
  @echo "Nettoyage de $(CIBLE)"
  @rm -rf $(CIBLE)
```

makefile_sources:

```
main.c elt.c list.c rpn.c stack_cd.c stack_cld.c stack_cs.c
```

- ***main.c*** : point d'entrée du programme (expliqué en détail dans la section 2.2.1.).

2.1.2. Code *rpn.c*

Le code rpn suit une logique qui dépend d'abord d'une entrée correcte dans le format de notation polonais inverse approprié.

Pour obtenir l`implémentation la plus efficace, on a utilisé l'implémentation IMPLEMENTATION_DYNAMIC_LINKED parce qu'il a le temps d'exécution le plus court comparé aux autres alternatives.

2.1.2.1. *rpn.h*

Tout d'abord, *elt.h* utilise la fonction *s2list* pour convertir l'expression donnée (une chaîne de caractères) en une liste *T_elt*, en utilisant le pointeur * exp. Après ça, la fonction *rpn_eval* est utilisé pour évaluer l'expression donnée. C'est un fichier d'en-tête (header file) essentiel pour faire *rpn.c* fonctionner.

2.1.2.2. rpn.c

Au début, il faut inclure les bibliothèques et les fichiers de base:

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "elt.h"
#include "list.h"
#include "stack_cld.h"

T_elt operation(int a, int b, char op);//qui fait les operations
```

Image - Extrait du code *rpn.c*

Les fonctions utilisées:

- T_elt *operation*

La fonction vérifie l'opérateur (s'il s'agit d'une addition, d'une soustraction, d'une multiplication ou d'une division) et effectue le calcul avec les deux entiers donnés, comme indiqué ci-dessous.

```c
T_elt operation(int a, int b, char op){
  T_elt out;
  switch(op){
    case SOMME:
        out.result = a + b;
        break;
    case SOUSTRACTION:
        if(a-b<0){
          out.status=RESULTAT_NONENTIER;
          out.result=-1;
        }
        else{
          out.result = a - b;
          out.status='e';
        }
        break;
    case DIVISION:
        if(a%b!=0){
          out.status=RESULTAT_NONENTIER;
          out.result=-1;
        }
```

```
        else{
          out.result = a / b;
          out.status='e';
        }
        break;
    case MULTIPLICATION:
        out.result = a * b;
        out.status='e';
        break;
    }

  return out;
}
```

Image - Extrait du code *rpn.c*

- T_list *s2list*

Cette fonction doit transformer une expression en notation polonaise inverse sous forme de chaîne de caractères en une liste de T_elt.

C'est possible de faire ça quand on crée le pointeur *pt, copie la séquence de caractères de l'expression polonaise donnée, puis extrait les éléments syntaxiques. Le *while* est utilisé pour vérifier chaque élément, s'il s'agit d'un opérateur ou d'un nombre, un *Node* du type est ajouté à la liste selon le cas.

Finalement, il y a un *return* de la liste.

```

#ifdef IMPLEMENTATION_DYNAMIC_LINKED


T_list s2list(char *exp) {
  char *pt, op[100];
  strcpy(op, exp);
  // float num;
  T_node *Lista = NULL;
  pt = strtok(op, " ");
  T_elt e;

  while (pt != NULL) {
    if (pt[0] == '+' || pt[0] == '-' || pt[0] == '/' || pt[0] == '*') {
      e.result = -1;
      e.status = pt[0];
      Lista = tailAddNode(eltdup(e), Lista);
    } else {
      e.result = atoi(pt);
      e.status = OPERANDE;
      Lista = tailAddNode(eltdup(e), Lista);
    }
    //printf("%s\n", toString(e));
    pt = strtok(NULL, " ");
  }
  return Lista;
}


```

Image - Extrait du code *rpn.c*

- T_elt *rpn_eval*

Tout d'abord, une liste est créée avec la notation polonaise inverse reçue via la fonction *s2list*. Ensuite, une pile est créée et nous associons le pointeur *pAux à la liste afin que les calculs puissent être effectués dans le loop.

Avec le *while*:

Si un opérande est trouvé (et pas un opérateur), on entre l'opérande dans la pile et on empile avec *push*. Si un opérateur est trouvé, on dépile 2 éléments avec *pop*, on réalise l'opération et on empile de nouveau le résultat avec *push*.

À la fin, on a le résultat en utilisant le méthode démandé.

```
74 T_elt rpn_eval(char *exp){
75   T_list Lista = NULL;
76   Lista = s2list(exp);
77   T_stack Pilha;
78   Pilha = newStack();
79   T_node *pAux = Lista;
80   T_elt e, e1, e2, aux, aux2;
81   aux.status = EXPRESSION_INVALIDE;
82   aux.result = -1;
83
84   while(pAux != NULL){
85
86     e = pAux->data;
87
88     if(e.status == OPERANDE){
89       push(e, &Pilha);
90     }
91     else if(e.result == -1){
92       if(getSize(Pilha) < 2) return aux;
93       e2 = pop(&Pilha);
94       e1 = pop(&Pilha);
95
96     push(operation(e1.result, e2.result, e.status), &Pilha);
97
98     }
99
```

```
100       pAux = pAux->pNext;
101     }
102
103 ▼   if(Pilha != NULL && getSize(Pilha) == 1) {
104       aux = top(&Pilha);
105       aux.status = EXPRESSION_VALIDE;
106
107       return aux;
108     }
109 ▼   else{
110       return aux;
111     }
112     }
113
114
115 #endif
116
117 //-----------------------------------------------------------------------------
    --------------------
```

Image - Extrait du code *rpn.c*

2.2 Partie 2 - Exploration d'un arbre à la volée

En fait, la deuxième partie une application directe au jeu "Le Compte est Bon" en utilisant une exploration d'un arbre à la volée.

2.2.1. Fonction main

On commence la fonction *main* comme toutes les fonctions par déclarer les bibliothèques et les fichiers de base à utiliser. Pour la *main*, on déclare aussi le pointeur *dropfromTab*.

```
1  #include <assert.h>
2  #include <stdio.h>
3
4  #include "include/traces.h"
5
6  #include "elt.h"
7  #include "list.h"
8  #include "rpn.h"
9  #include "stack_cld.h"
10
11 int* dropfromTab(int* tableau, int index, int length);
```

dropfromTab est défini après comme:

```
148 ▼ int* dropfromTab(int* tableau, int index, int length){
149
150     int *new_tableau = (int*)malloc((length-1) * sizeof(int));
151     int j = 0;
152
153 ▼   for (int i = 0; i < length; i++){
154       if(i == index) continue;
155       new_tableau[j] = tableau[i];
156       j++;
157     }
158     return new_tableau;
159 }
160
```

où il utilise *malloc* pour réserver de l'espace (allouer un bloc de mémoire) et après c'est utilisé un *for* pour stocker les élements dans un nouveau tableau.

Maintenant, on rentre la fonction main:

- Il y a comme point de départ une valeur cherché, 6 nombres (qu'on appellera a,b,c,d,e,f) et 4 opérateurs (i, j, k, m).
- Une boucle est faite avec *for* qui va parcourir les 6 nombres un par un, en sélectionnant le premier chiffre (Si le premier nombre est dejà la valeur cherché, on a retour sur ça. Sinon, on continue.) et le second afin de permuter les combinaisons possibles, en sélectionnant également un opérateur.
- Nous mettons ensuite cette opération en notation polonaise inverse et on utilise la fonction rpn_eval pour valider et après on calcule le résultat
    - Ce calcul utilise et s'intègre aux fonctions demandées précédemment, ainsi qu'au fichier rpn.c en général.
- Si une opération avec 2 nombres est dejà la valeur cherché, on a un retour sur ça. Sinon, on continue.
- Un autre nombre parmi les possibilités est sélectionné et le boucle teste toutes les possibilités et parcourt également les opérateurs. Ces tests sont effectués par le même moyen de notation polonaise inverse.
- Si une opération avec 3 nombres est dejà la valeur cherché, on a un retour sur ça. Sinon, on continue.
- Ceci est répété jusqu'à ce que toutes les possibilités avec les 6 numéros soient épuisées.
- Pour enregistrer les résultats partiels, nous utilisons le tableau généré et le *dropfromTab* déjà vu
- Pour finir la *main*, il reste seulement la sortie, qui prend les éléments enregistrés du tableau généré et fabrique la résolution étape par étape comme demandé, quand il y a un "match" du valeur cherché avec les opérations.

L'algorithme permettant d'implémenter cette fonction (comme décrit ci-dessus):

```c
int main(void) {


  ////////////////////////////////////////////////
  // Partie 2 - Exploration d'un arbre à la volée//
  ////////////////////////////////////////////////

  // a,b,c,d,e,f pour les nombres i,j,k... pour les operateurs

  int ValeurCherche = 90;
  int nombres[6]={3, 10, 7, 5, 16,88};
  int length = 6;

  char operateurs[] = {'+', '-', '*','/'};
  int *nombres_mod;
  int v1,v2,v1_2,v3,v1_2_3,v4,v1_2_3_4,v5,v1_2_3_4_5,v6;
  int a,b,c,d,e,f,i,j,k,l,m;

  T_elt test;
  char op[100];

  for(a=0; a<6;a++){
    v1 = nombres[a];
    printf("\n Premier nombre: %d \n",v1);
    for(b=0; b<5; b++){
      nombres_mod = dropfromTab(nombres, a, length);
      v2 = nombres_mod[b];
      printf("Deuxième nombre: %d \n",v2);
      for(i=0; i<4; i++){
        printf("Operateur: %c\n", operateurs[i]);
        sprintf(op, "%d %d %c", v1, v2, operateurs[i]);
        test = rpn_eval(op);
        printf("Result: %d\n", test.result);

        if(test.result == ValeurCherche){
          printf("La combinaison pour calculer le valeur est (en RPN): %d %c %d = %d \n", v1, operateurs[i], v2, test.result);
          return 0;
        }
        if(test.result == -1) continue;
        v1_2=test.result;

        for(c=0; c<4; c++){
          printf("\n Combinaison 1°,2°: %d \n",v1_2);
          nombres_mod = dropfromTab(nombres, a, length);
          nombres_mod = dropfromTab(nombres_mod, b, length-1);
          v3 = nombres_mod[c];
          printf("\nTroisième nombre: %d \n",v3);
          for(j=0; j<4; j++){
            printf("Operateur: %c\n", operateurs[j]);
            sprintf(op, "%d %d %c", v1_2, v3, operateurs[j]);
            test = rpn_eval(op);
            printf("%d\n", test.result);

            if(test.result == ValeurCherche){
              printf("La combinaison pour calculer le valeur est (en RPN): %d %c %d = %d \n", v1_2, operateurs[j], v3, test.result);
              return 0;
            }
            if(test.result == -1) continue;
            v1_2_3=test.result;
```

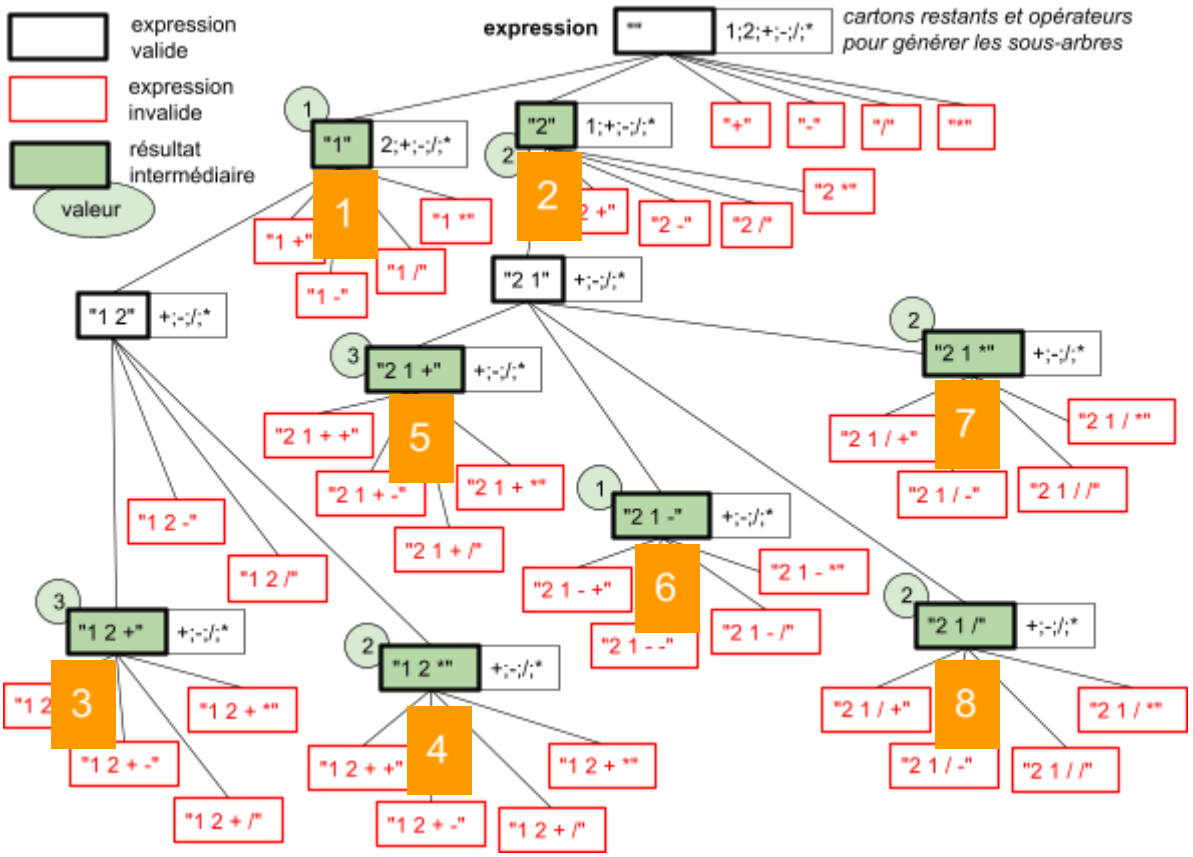
```
74
75 ▼             for(d=0; d<3; d++){
76                printf("\n Combinaison 1°,2°,3°: %d \n",v1_2_3);
77                nombres_mod = dropfromTab(nombres, a, length);
78                nombres_mod = dropfromTab(nombres_mod, b, length-1);
79                nombres_mod = dropfromTab(nombres_mod, c, length-2);
80                v4 = nombres_mod[d];
81                printf("\nQuatrième nombre: %d \n",v4);
82 ▼              for(j=0; j<4; j++){
83                  printf("Operateur: %c\n", operateurs[j]);
84                  sprintf(op, "%d %d %c", v1_2_3, v4, operateurs[j]);
85                  test = rpn_eval(op);
86                  printf("%d\n", test.result);
87
88 ▼                if(test.result == ValeurCherche){
89                    printf("La combinaison pour calculer le valeur est (en RPN): %d %c %d = %d \n",
   v1_2_3, operateurs[j], v4, test.result);
90                    return 0;
91                  }
92                  if(test.result == -1) continue;
93                  v1_2_3_4=test.result;
94
95 ▼                for(e=0; e<2; e++){
96                    printf("\n Combinaison 1°,2°,3°,4°: %d \n",v1_2_3_4);
97                    nombres_mod = dropfromTab(nombres, a, length);
98                    nombres_mod = dropfromTab(nombres_mod, b, length-1);
99                    nombres_mod = dropfromTab(nombres_mod, c, length-2);
100                   nombres_mod = dropfromTab(nombres_mod, d, length-3);
101                   v5 = nombres_mod[e];
102                   printf("\nCinqième nombre: %d \n",v5);
103 ▼                 for(j=0; j<4; j++){
```

```
104                   printf("Operateur: %c\n", operateurs[j]);
105                   sprintf(op, "%d %d %c", v1_2_3_4, v5, operateurs[j]);
106                   test = rpn_eval(op);
107                   printf("%d\n", test.result);
108 ▼                 if(test.result == ValeurCherche){
109                     printf("La combinaison pour calculer le valeur est (en RPN): %d %c %d = %d
    \n", v1_2_3_4, operateurs[j], v5, test.result);
110                   return 0;
111                   }
112                   if(test.result == -1) continue;
113                   v1_2_3_4_5=test.result;
114
115 ▼                 for(f=0; f<1; f++){
116                     printf("\n Combinaison 1°,2°,3°,4°,5°: %d \n",v1_2_3_4);
117                     nombres_mod = dropfromTab(nombres, a, length);
118                     nombres_mod = dropfromTab(nombres_mod, b, length-1);
119                     nombres_mod = dropfromTab(nombres_mod, c, length-2);
120                     nombres_mod = dropfromTab(nombres_mod, d, length-3);
121                     nombres_mod = dropfromTab(nombres_mod, d, length-4);
122                     v6 = nombres_mod[f];
123                     printf("\nSixième nombre: %d \n",v6);
124 ▼                   for(j=0; j<4; j++){
125                       printf("Operador: %c\n", operateurs[j]);
126                       sprintf(op, "%d %d %c", v1_2_3_4_5, v6, operateurs[j]);
127                       test = rpn_eval(op);
128                       printf("%d\n", test.result);
129 ▼                     if(test.result == ValeurCherche){
130                         printf("La combinaison pour calculer le valeur est (en RPN): %d %c %d =
    %d \n", v1_2_3_4_5, operateurs[j], v6, test.result);
131                       return 0;
132                       }
```

```
133                                 }
134                               }
135                             }
136                           }
137                         }
138                       }
139                     }
140                   }
141                 }
142               }
143             }
144
145             return 0;
146         }
```

Images - Extraits du code *main.c*

Concernant l'exploration d'un arbre à la volée:
Le chemin plus logique pour obtenir l'efficace de la résolution du problème est représenté dans l'arbre ci-dessous avec des indices de 1 à 8 (en orange), 1 étant le premier au parcours à suivre et 8 le dernier. On va énumerer seulement les opérations possibles, parce que par exemple "1 - 2" nous donne un nombre négatif.



3. Conclusion

Enfin, nous récapitulons que tous les codes ont été élaborés et intégrés pour trouver la solution du jeu "le compte est bon" en effectuant les calculs à l'aide d'une pile en notation polonaise inverse.

4. Réferences

- Matériel AAP Séance 2
- AAP - TEA S2