

INFO0947: Compléments de Programmation  
Récursivité et Élimination de la Récursivité

LUCA MATAGNE, s190632

## Table des matières

1	Formulation Récursive	3
1.1	Définition récursive . . . . .	3
1.2	Cas de base . . . . .	3
1.3	Cas récursif . . . . .	3
1.4	Synthèse . . . . .	4
2	Spécification	4
3	Construction Récursive	5
3.1	Programmation défensive . . . . .	5
3.2	Cas de base . . . . .	5
3.3	Cas récursif . . . . .	5
3.4	Code récursif complet . . . . .	6
4	Traces d'exécution	6
4.1	Descente récursive . . . . .	6
4.2	Tableau 1 . . . . .	6
4.2.1	Tableau 2 . . . . .	6
4.2.2	Tableau 3 . . . . .	6
4.3	Remontée récursive . . . . .	6
4.3.1	Tableau 4 . . . . .	6
4.3.2	Tableau 5 . . . . .	7
4.3.3	Tableau 6 . . . . .	7
5	Complexité	7
6	Dérécursification	7

# 1 Formulation Récursive

## 1.1 Définition récursive

Nous observons que :

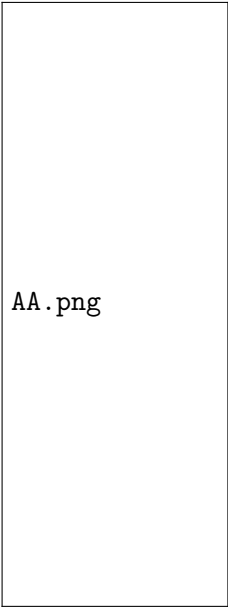
```
1
2 Un nombre décimal est un nombre hexadécimal ayant subi une transformation et
3 qu'un nombre hexadécimal est un caractère représentant une puissance de 16
4 ajouté à la suite d'une chaîne de caractères représentant les puissances de 16
5 supérieures
```

Nous avons donc défini la structure « chaîne de caractères (représentant un nombre hexadécimal) » récursivement. Nous allons donc formuler la transformation d'une chaîne de caractères en nombre décimaux sur base de cette définition récursive.

Appelons notre notation  $Traitement(s, n)$ , avec  $s$  une chaîne de caractères et  $n$  sa longueur et appelons  $Transfo(s[n])$  la transformation (permettant de passer de hexadécimal à décimal) de l'élément de rang  $n$  dans la chaîne de caractères  $s$ .

## 1.2 Cas de base

Une chaîne de caractères composée d'un seul caractère est une chaîne de caractère représentant la puissance 0 de 16 en nombre hexadécimal. Lorsqu'on lui applique la transformation en nombre décimal, on obtient le résultat suivant :



AA.png

Exemple :

$$s = 'A, / 0'$$
$$Traitement(s, 1) = Transfo(s[0]) = 11$$

## 1.3 Cas récursif

Pour déterminer le cas récursif, observons un exemple :

$$'A23' \rightarrow 2595$$

Imaginons que l'on découpe la chaîne '**A23**' selon la définition récursive, on a

$$\begin{aligned}
& \text{'A2 || 3'} \rightarrow \text{'A2'} + 3 * 1 \\
& \quad = \\
& \text{'A2 || 3'} \rightarrow \text{'A2'} + 3 \\
& \text{(Le 3 après le + est bien le nombre décimal et plus le caractère)}
\end{aligned}$$

Et puis,

$$\begin{aligned}
& \text{'A || 23'} \rightarrow \text{'A'} + 2*16 + 3*1 \\
& \quad = \\
& \text{'A || 23'} \rightarrow \text{'A'} + 35
\end{aligned}$$

On voit qu'il faut transformer le dernier caractère de la chaîne et additionner le résultat de cette transformation au produit du facteur 16 (explication dans le paragraphe suivant) avec le futur résultat du traitement de la chaîne de caractères exemptée du caractère déjà transformé.

La transformation citée depuis le début de ce rapport ne contient pas les puissances de 16 permettant de passer effectivement d'un nombre hexadécimal à un nombre décimal. En effet, cette transformation ne faisait qu'allusion à la fonction *convert(char hex)* préenregistrée pour nous. C'est pourquoi il est nécessaire de multiplier l'appel récursif par 16.

De cette façon, les puissances seront toujours respectées étant donné que pour atteindre le rang 2 (où nous avons donc 16<sup>2</sup>) il y a 2 appels récursifs (et 1 appel de base). Nous aurons donc un facteur 16\*16 au final pour ce rang, ce qui est correct.

I

$$\textit{Traitement}(s, n) = \textit{Transfo}(s[n-1]) + 16 * \textit{Traitement}(s, n-1)$$

## 1.4 Synthèse

La formalisation récursive de *Traitement* est donc :

$$\begin{aligned}
& \text{si } n = 0 \\
& \quad \textit{Traitement}(s, n) = \\
& \quad \quad \textit{Transfo}(s[0]) \\
& \quad \quad \textit{ET} \\
& \quad \quad \text{sinon} \\
& \quad \quad \textit{Traitement}(s, n) = \\
& \textit{Traitement}(s, n) = \textit{Transfo}(s[n-1]) + 16 * \textit{Traitement}(s, n-1)
\end{aligned}$$

## 2 Spécification

Etant donné l'utilisation du Pseudo Langage vu au cours, il n'est pas nécessaire de vérifier que le pointeur représentant la chaîne de caractères est valide cependant nous allons vérifier que notre chaîne existe grâce à sa longueur.

Notons que dans cette section nous reprendrons les notations introduite au début de ce rapport.

$$\text{PréCondition} \equiv n > 0$$

L'objectif de notre fonction est de transformer un nombre hexadécimale (sous forme de chaîne de caractères) en un nombre décimale. En accord avec le code fourni, appelons la *hexa\_dec\_rec*. Elle prendra comme argument la chaîne de caractère donnée et sa longueur sous forme d'un entier.

$$\text{Postcondition} \equiv \text{hexa\_dec\_rec} = \text{Traitement}(s, n)$$

Au final l'interface de la fonction est :

```

1 /*
2  * @pre: n > 0
3  * @post: hexa_dec_rec = Traitement(s,n)
4  */
5 hexa_dec_rec(s,n)=(int x)

```

### 3 Construction Récursive

La structure générale d'une fonction/procédure récursive s'appuie sur une structure conditionnelle. On va donc remplir cette structure en trois étapes : (i) programmation défensive, (ii) cas de base, (iii) cas récursif(s).

#### 3.1 Programmation défensive

En accord avec les préconditions, nous devons nous assurer que la chaîne est utilisable et donc que sa longueur n'est pas nulle.

```

1 hexa_dec_rec(s,n):
2     if ( n<=0 )
3     then
4         r<-NULL;

```

Dans ce contexte de Pseudo-Langage théorique, je vérifie mes préconditions à l'aide d'une structure conditionnelle. Dans le code cette vérification est traduite par un *assert* comme ceci :

```

1 unsigned int hexa_dec_rec(char *hexa, int n){
2     assert(hexa!=NULL && n>0);

```

#### 3.2 Cas de base

Si la chaîne de caractères n'en contient qu'un, on transforme directement ce caractère en son équivalent décimal.

```

1 hexa_dec_rec(s,n):
2     // PréCondition
3     if (n = 1)
4     then
5         // int x
6         r<-transfo(s[n-1]);
7         // PostCondition

```

#### 3.3 Cas récursif

On suit la formulation récursive de *Traitement*.

```

1 else
2     r<-Traitement(s, n) = Transfo(s[n-1]) + 16 * Traitement(s, n-1)
3     // PostCondition

```

### 3.4 Code récursif complet

```

1 hexa_dec_rec(s,n):
2   if (n = 1)
3     then
4       r<-transfo(s[n-1]);
5   else
6       r<-Traitement(s, n) = Transfo(s[n-1]) + 16 * Traitement(s, n-1)

```

## 4 Traces d'exécution

Dans cette section nous allons voir les valeurs qui sont stockées sur la pile lors de l'exécution de notre fonction. Pour cela nous allons utiliser le même exemple que lors de la formalisation récursive de la fonction : 'A23'.

			10	2560		
	2	32	2	2592		
3	3	3	3	3	2595	
1	2	3	4	5	6	

Pour une facilité d'explication, se trouvent en dessous de chaque tableau représentant la pile le numéro lui correspondant.

### 4.1 Descente récursive

#### 4.1.1 Tableau 1

Appel de la fonction.  $n > 1$  donc on entre dans le cas récursif. Le résultat vaut donc 3 qu'on empile.

#### 4.1.2 Tableau 2

2ème appel de la fonction.  $n > 1$  donc on entre dans le cas récursif. Le résultat vaut 2. qu'on empile.

#### 4.1.3 Tableau 3

3ème appel de la fonction.  $n == 1$  donc on entre dans le cas de base. Le résultat vaut 10 qu'on empile.

### 4.2 Remontée récursive

Il est temps de faire les calculs.

#### 4.2.1 Tableau 4

Etant donné qu'il y avait déjà eu 2 appels de la fonction lorsque le caractère évalué nous rendait 10 comme résultat, nous devons multiplier ce résultat par  $16^2$  ce qui donne 2560.

#### 4.2.2 Tableau 5

N'oublions d'ajouter à ça (2560) l'appel précédent qui rendait 2 comme résultat mais avant lequel il y avait déjà eu 1 appel de la fonction. Il va donc de soi que nous devons multiplier 2 par  $16^1 = 32$  avant de faire l'addition. (Multiplication avant addition car PEMDAS!). Nous avons donc maintenant une somme valant 2592.

#### 4.2.3 Tableau 6

Cette fois, il nous faut additionner 2592 avec le premier appel de la fonction qui renvoyait 3 comme résultat. Par définition, il n'y avait jamais eu d'appel de la fonction avant. Il est de ce fait inutile de chercher par quel facteur 3 doit être multiplié. Nous pouvons désormais faire notre dernière addition ( $2592 + 3$ ), ce qui nous donne 2595 comme résultat final.  $2595_{10}$  équivaut bien à  $A23_{16}$ .

### 5 Complexité

Nous allons, ici, nous charger de déterminer la complexité de notre fonction récursive. Pour cela nous allons éliminer la récurrence de proche en proche.

Le code de notre fonction étant :

```
1 unsigned int hexa_dec_rec(char *hexa, int n){
2     assert(hexa!=NULL && n>0);
3
4     if(n==1)
5         return convert(hexa[n-1]);
6     else
7         return convert(hexa[n-1]) + 16 * hexa_dec_rec(hexa, (n-1));
8 }//fin hexa_dec_rec()
```

Nous allons découper ce code en trois pour y voir plus clair :

a :

```
1 if(n==1)
2     return convert(hexa[n-1]);
```

b :

```
1 if(n==1)
2     return convert(hexa[n-1]);
3 else
4     return convert(hexa[n-1]) + 16 * hexa_dec_rec(hexa, (n-1));
```

T(n-1) :

```
1     return convert(hexa[n-1]) + 16 * hexa_dec_rec(hexa, (n-1));
```

Soit  $T(n)$ , le coût d'un appel à `hexa_dec_rec(hexa, (n))`. En ce qui concerne le cas de base, on va dire qu'il prend  $a$  opérations,  $a$  étant constant. Pour ce qui est du cas récursif, on va dire qu'il demande  $b$  opérations ainsi que le nombre d'opérations nécessaires par l'appel récursif, c'est à dire  $T(n-1)$ . Il vient le système suivant :

$$\begin{aligned} T(n) &= a \text{ si } n=1 \\ &= b+T(n-1) \text{ sinon} \end{aligned}$$

L'élimination de proche en proche consiste à réécrire plusieurs fois cette équation en utilisant le cas récursif afin de faire apparaître un motif reconnaissable.

$$\begin{aligned} T(n) &= b+T(n-1) \\ &= b+b+T(n-2) \\ &= 3b+T(n-3) \\ &= 4b+T(n-4) \\ &= \dots \\ &= k \times b + T(n-k) \end{aligned}$$

$T(n-k)$  est une formulation tout à fait générale du temps d'exécution du Kème appel récursif. On ne connaît qu'une valeur particulière de  $T(.)$  :  $T(1) = 0$ . On va donc essayer de faire apparaître cette valeur particulière. On a :

$$\begin{aligned} n-k &= 1 \\ -k &= 1-n \\ k &= n-1 \end{aligned}$$

On insère cette valeur dans  $k \times b + T(n-k)$ , il vient :

$$T(n) = (n-1) \times b + T(1) = (n-1) \times b + a \in \mathcal{O}(n)$$

## 6 Dérécursification

### 6.1 Introduction du cas général

Notre algorithme n'est pas de récursivité terminale et bien que la multiplication et l'addition soient chacune commutative et associative, la combinaison des deux ne l'est plus (  $(5*3)+2 \neq 5*(3+2)$  ). Nous allons donc utiliser le cas général de dérécursification vu au cours. Dans ce cas, nous allons simuler la pile grâce à un TAD dont voici la sémantique :

Type :

- Stack

Utilise :

- Boolean, Element

Opérations :

- $\text{empty\_stack} : \text{stack} \rightarrow \text{stack}$
- $\text{is\_empty} : \text{stack} \rightarrow \text{boolean}$
- $\text{push} : \text{stack} \times \text{element} \rightarrow \text{stack}$
- $\text{pop} : \text{stack} \rightarrow \text{stack}$
- $\text{top} : \text{stack} \rightarrow \text{element}$

Préconditions :

- $\text{pop}(s)$  is defined iff  $\text{is\_empty}(s) = \text{False}$
- $\text{top}(s)$  is defined iff  $\text{is\_empty}(s) = \text{False}$

Axiomes :

- $\text{is\_empty}(\text{empty\_stack}) = \text{True}$
- $\text{is\_empty}(\text{push}(s, e)) = \text{False}$
- $\text{pop}(\text{push}(s, e)) = s$
- $\text{top}(\text{push}(s, e)) = e$



## 6.2 Algorithme de transformation

La conversion de la fonction récursive en itérative sera constituée d'une boucle qui ne s'arrêtera que quand la Pile sera vide. Au début de chaque itération, on dépile un contexte et on exécute les opérations en fonction de la valeur de pc. Ce qui donne le code suivant :

```
1 f (int x):
2   // Il faut évidemment déclarer et initialiser une Pile vide.
3   s <- empty_stack ();
4
5   // On place sur la Pile le premier appel de la fonction, forcément, pc = 1 :
6   s <- push(s, (x, 1));
7
8   until is_empty(s) do
9     // On dépile le premier contexte sur la pile
10    (x, pc) <- top(s);
11    s <- pop(s);
12
13    // On choisit sur base de pc les opérations à effectuer:
14    if (pc = 1)
15      then
16        if (cond(x))
17          then
18            r <- base(x);
19          else
20            /* Avant l appel récursif, on se souvient qu il faudra
21             continuer dans le bloc
22            */
23            s <- push(s, (x, 2))
24            // On empile ensuite l'appel récursif
25            s <- push(s, (progression(x), 1))
26          else // pc = 2, donc
27            // On récupère la dernière valeur de retour
28            tmp <- r ;
29            // On retourne le résultat calculé
30            r <- operation(tmp , x);
31  end
32
33 // la variable r contient donc la valeur finale
```

On considère que pc = 1 quand on entre dans la fonction et pc = 2 au retour de l'appel récursif.

## 6.3 Transformation du code en un code générique récursif

Code récursif :

```
1 hexa_dec_rec(s,n):
2   if (n = 1)
3     then
4       r<-transfo(s[n-1];
5     else
6       r<-Traitement(s, n) = Transfo(s[n-1]) + 16 * Traitement(s, n-1)
```

Code récursif générique :

```

1 f(s,n):
2   if (cond(n))
3   then
4     r<-g(s);
5   else
6     r<-G(s, f(s,n));

```

Code itératif générique :

```

1 f_derec(s, n):
2   p <- empty_stack();
3   a <- 0;
4   b <- s;
5   until cond(a) do
6     p <- push(p, b[a]);
7     a <- h(a);
8   end
9   until is_empty(p) do
10    b <- top(p);
11    p <- pop(p);
12    r <- G(s, r);
13  end

```

Code itératif final :

```

1 hexa_dec_rec_derec(s, n):
2   p <- empty_stack();
3   a <- 0;
4   b <- s;
5   until a = n do
6     p <- push(p, b[a]);
7     a <- a+1;
8   end
9   until is_empty(p) do
10    b <- top(p);
11    p <- pop(p);
12    r <- transfo(b) + 16 * r ;
13  end

```