

INFO0947: Récursivité et Elimination de la Récursivité

PEISSONE DUMOULIN, s193957

1 Formulation Récursive

1.1 Cas de base

Si $n = 1$:

$$\text{hexa_dec_rec}(\text{hexa}, n) = \text{convert}(\text{hexa}[n - 1])$$

1.2 Cas récursif

Si $n > 1$:

$$\text{hexa_dec_rec}(\text{hexa}, n) = \text{convert}(\text{hexa}[n - 1]) + 16 * \text{hexa_dec_rec}(\text{hexa}, n - 1)$$

2 Spécification

```
1 //PréConditions : hexa != NULL, n > 0
2 //PostConditions : hexa_dec_rec = decimal ∧ hexa = hexa0 ∧ n = n0
3 unsigned int hexa_dec_rec(char *hexa, int n);
```

3 Construction Récursive

3.1 Programmation Défensive

On vérifie que la précondition est respectée en interdisant à hexa d'être NULL et n ne peut pas être strictement négatif

```
1 unsigned int hexa_dec_rec(char *hexa, int n){
2     assert(hexa != (void*)0 && n > 0);
3     // {PréConditions ≡ hexa ≠ NULL ∧ (longueur(hexa) > 0 ⇒ n > 0)}
4 }
```

3.2 Cas de Base

On gère le cas de base où $n = 1$ après s'être assuré que les préconditions sont bien respectées.

```
1 // {PréConditions  $\equiv$   $hexa \neq \text{NULL} \wedge (\text{longueur}(hexa) > 0 \implies n > 0)$ }
2 if(n == 1)
3     // { $hexa \neq \text{NULL} \wedge n = 1$ }
4     return convert(hexa[n - 1]);
5     // { $hexa\_dec\_rec = \text{convert}(hexa[n - 1]) \wedge hexa = hexa_0 \wedge n = n_0$ }
6     // { $\implies$  PostCondition}
7 }
```

3.3 Cas Récursif

Il y a un seul cas récursif, lorsque $n > 1$. $\{\text{PréConditions}_{REC}\}$ et $\{\text{PostConditions}_{REC}\}$ sont respectivement les PréConditions et les PostConditions de l'appel récursif.

```
1 else
2     // { $hexa \neq \text{NULL} \wedge n > 1$ }
3     // { $\implies$  PréConditionsREC}
4     return convert(hexa[n - 1]) + 16 * hexa_dec_rec(hexa, n - 1);
5     // {PostConditionsREC  $\equiv$ 
6     //  $hexa\_dec\_rec(hexa, n) = \text{convert}(hexa[n - 1]) + 16 * hexa\_dec\_rec(hexa, n-1)$ 
7     //  $\wedge n = n - 1 \wedge hexa = hexa_0$ }
8     // { $\implies$  PostConditions}
```

3.4 Code complet

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

4 Traces d'Exécution

Voici les traces d'exécution concernant les 3 exemples du fichier main-hexadécimal.c

4.1 hexa_dec_rec("27", 2)

hexa_dec_rec("27", 2)	$7 + 16 * 2 = 39$
-----------------------	-------------------

4.2 hexa_dec_rec("A23", 3)

hexa_dec_rec("A23", 3)	$3 + 16 * (2 + 16 * 10) = 2595$
------------------------	---------------------------------

4.3 hexa_dec_rec("A78E", 4)

hexa_dec_rec("A78E", 4)	$14 + 16 * (8 + 16 * (7 + 16 * (10))) = 42894$
-------------------------	--

5 Complexité

Avec $b \in [0, 15]$, $n \in \mathbb{N}$, on a :

$$T(n) = 16 * T(n - 1) + b$$

5.1 Cas de base

Dans le cas où $n = 1$:

$T(n)$ est linéaire car $T(1) = b$ et $b \in [0, 15]$

5.2 Cas récursif

Dans le cas où $n > 1$:

La fonction hexa_dec_rec(hexa, n) va s'appeler récursivement en décrémentant la valeur courante de n à chaque appel jusqu'à atteindre le cas de base. On a donc n - 1 appels récursifs.

$$T(n) = 16 * T(n - 1) + b \rightarrow O(n)$$

La complexité de la fonction hexa_dec_rec(hexa, n) est **linéaire**.

6 Dérécursification

Pour procéder à la dérécursification, on va utiliser le pseudo langage qu'on a vu dans les gamecodes associés.

6.1 Code récursif

```
1 hexa_dec_rec(String hexa, int n):  
2   if(n = 1)  
3     then  
4       r <- convert(hexa[n - 1]);  
5     else  
6       r <- convert(hexa[n - 1]) + 16 * hexa_dec_rec(hexa, n - 1);
```

6.2 Code dérécursivé

```
1 hexa_dec_rec'(String hexa, int n):  
2   s <- hexa;  
3   u <- n;  
4   until u = 1 do  
5     s = s;  
6     u = u - 1;  
7   end  
8   r <- convert'(hexa[u - 1]);
```