INFO0947: Récursivité et Elimination de la Récursivité

Peissone Dumoulin, s193957

1 Notations

Avant de commencer à parler de la formulation récursive, introduisons d'abord les notations utilisées par la suite :

— $hexa \equiv$ chaîne de charactères représentant un nombre hexadécimal
— $n \equiv$ taille de la chaîne de charactères $hexa$
— $hexa\_dec\_rec \equiv$ résultat de la conversion d'un nombre hexadécimal en sa traduction décimale.

2 Formulation Récursive

2.1 Cas de base

Une chaîne de caractères vide ne pouvant pas être convertie, le premier cas à considérer est le cas où l'on a un seul charactère à savoir $n = 1$. Afin d'obtenir le nombre décimal à partir de son homologue en hexa, il suffit juste d'utiliser la fonction convert(). Fonction permettant de convertir un nombre hexadécimal en un nombre décimal correspondant.

Si $n = 1$:

$$hexa\_dec\_rec(hexa, n) = convert(hexa[n-1])$$

2.2 Cas récursif

Pour procéder de manière récursive, on prend tous les cas possibles dans l'ordre croissant à partir du cas de base + 1. Autrement dit lorsque n est strictement supérieur à 1.

Si $n > 1$:

$$hexa\_dec\_rec(hexa, n) = convert(hexa[n-1]) + 16 * hexa\_dec\_rec(hexa, n-1)$$

2.3 Synthèse

En faisant la synthèse du cas de base et du cas récursif, on obtient la formulation récursive de $hexa\_dec\_rec$:

$$hexa\_dec\_rec(hexa, n) = \begin{cases} convert(hexa[n-1]) & \text{si n} = 1 \\ convert(hexa[n-1]) + 16 \times hexa\_dec\_rec(hexa, n-1) & \text{sinon} \end{cases}$$

3 Spécification

L'énoncé nous dit que la fonction `hexa_dec_rec` est de type `unsigned int` et prend comme paramètres `hexa` et `n` avec `hexa` une chaîne de charactères (`char*`) et `n` sa taille (`int`)

3.1 PréCondition

Comme vu précédemment dans le cas de base, n ne peut être nul. Sachant que n est un entier, il paraît naturel de dire que n doit être strictement supérieur à 0. De plus, on ne peut pas convertir une chaîne de charactères qui n'existe pas. De ce fait, on exprime $hexa \,!= \text{NULL}$. Ce qui nous donne la préCondition suivante :

$$\text{PréCondition} \equiv hexa \,!= \text{NULL} \wedge n > 0$$

3.2 PostCondition

En postCondition, nous voulons que $hexa$ et n ne soient pas modifiés et que $hexa\_dec\_rec$ soit égal à la notation introduite précédemment à savoir $hexa\_dec\_rec(hexa, n)$ Celà se traduit par la postCondition suivante :

$$\text{PostCondition} \equiv hexa\_dec\_rec = hexa\_dec\_rec(hexa, n) \wedge hexa = hexa_0 \wedge n = n_0$$

3.3 Résumé

```
//PréCondition : hexa != NULL, n > 0
//PostCondition : hexa_dec_rec = hexa_dec_rec(hexa, n) ∧ hexa = hexa0 ∧ n = n0
unsigned int hexa_dec_rec(char *hexa, int n);
```

4 Construction Récursive

4.1 Programmation Défensive

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

```
unsigned int hexa_dec_rec(char *hexa, int n){
    assert(hexa != (void*)0 && n > 0);
    // {PréCondition ≡ hexa ≠ NULL ∧ n > 0)}
}
```

4.2 Cas de Base

On gère le cas de base où $n = 1$ après s'être assuré que la préCondition est bien respectée.

```
// {PréCondition ≡ hexa ≠ NULL ∧ n > 0)}
if(n == 1)
    // {n = 1 ∧ hexa = hexa_0 ∧ n = n_0}
    return convert(hexa[n - 1]);
    // {hexa_dec_rec(hexa, n) = convert(hexa[n - 1]) ∧ hexa = hexa_0 ∧ n = n_0}
    // { ⟹ PostCondition}
}
```

4.3 Cas Récursif

Il y a un seul cas récursif, lorsque n est strictement supérieur à 1. $\{\text{PréCondition}_{REC}\}$ et $\{\text{PostCondition}_{REC}\}$ sont respectivement la PréCondition et la PostCondition de l'appel récursif.

```
else
    // {hexa ≠ NULL ⟹ hexa_REC ≠ NULL ∧ n > 1}
    // { ⟹ PréCondition_REC}
    return convert(hexa[n - 1]) + 16 * hexa_dec_rec(hexa, n - 1);
    // {PostCondition_REC ≡ hexa_dec_rec = hexa_dec_rec(hexa, n) ∧ n = n_0 ∧ hexa = hexa_0}
    // {hexa_dec_rec = hexa_dec_rec(hexa, n) ∧ n = n_0 ∧ hexa = hexa_0}
    // { ⟹ PostCondition}
```

4.4 Code complet

```
unsigned int hexa_dec_rec(char *hexa, int n){
  assert(hexa != (void*)0 && n > 0);//Précondition

  if(n == 1)
    return convert(hexa[n - 1]);//Cas de base
  else
    return convert(hexa[n - 1]) + 16 * hexa_dec_rec(hexa, n - 1);//Cas récursif
}
```

5 Traces d'Exécution

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

5.1 hexa_dec_rec("27", 2)

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

5.2 hexa_dec_rec("A23", 3)

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

5.3 hexa_dec_rec("A78E", 4)

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

6 Complexité

On prendra la découpe suivante :

```c
unsigned int hexa_dec_rec(char *hexa, int n){
  assert(hexa != (void*)0 && n > 0);//Précondition

  if(n == 1)
    return convert(hexa[n - 1]);//Cas de base
  else
    return convert(hexa[n - 1]) + 16 * hexa_dec_rec(hexa, n - 1);//Cas récursif
}
```

A

B

T(n-1)

— Dans le cas où $n = 1$:

$T(n)$ est constant car $T(1) = a$

— 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.

$T(n)$ est linéaire car $T(n) = T(n-1) * b$

En résumé, on a :

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

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

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

7 Dérécursification

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

7.1 Code récursif

```
hexa_dec_rec(String hexa, int n):
  if(n=1)
    then
      r ← convert(hexa[n - 1]);
    else
      r ← convert(hexa[n - 1]) + 16 * hexa_dec_rec(hexa, n - 1);
```

7.2 Code dérécursivé

```
hexa_dec_rec'(String hexa, int n):
    s ← hexa;
    u ← n;
    until u = 1 do
        s ← s;
        u ← u - 1;
    end
    r ← convert'(hexa[u - 1]);
```