INFO0947: Projet 3 (Récursivité & Élimination de la Récursivité)

Timothy Smeers, S200930

Table des matières

1 Formulation Récursive.

1.1 Notations.

$\forall\, n \in \mathbb{N}\ ,\ n > 0$

$\forall\, case \in hexa\ ,\ \exists\, case \in \{0, 1, \ldots, 9, A, B, \ldots, F\}$

$\exists\, l\_nextCase \in hexa[]\ ,\ l\_nextCase = hexa + 1$

$\exists\, l\_position \in \mathbb{N}\ ,\ l\_position = n - 1$

$\exists\, l\_exposant \in \mathbb{N}, l\_exposant = hexa\_power(l\_position);$

$\exists\, (l\_convert)_{10} \in \mathbb{N}\ ,\ (l\_convert)_{10} = convert(hexa[l\_position])$

$\exists\, (l\_decimal)_{10} \in \mathbb{N}\ ,\ (l\_decimal)_{10} = (l\_convert)_{10} * (l\_exposant)_{10}$

1.2 Peut-on résoudre le problème récursivement ?

- Trois questions
  1. Peut-on trouver un paramètre récursif ?
     ▷ n
  2. Peut-on trouver un cas de base[1.3] ?
     ▷ $n = 1$
  3. Peut-on exprimer le problème de manière récursive[1.4] ?
     ▷ Pour $decimal \in (\mathbb{N})_{10}$, on a

$$(l\_decimal)_{10} = \begin{cases} l\_decimal & \text{si n = 1} \\ l\_decimal + hexa\_dec\_rec(l\_nextCase, l\_position) & \text{sinon.} \end{cases}$$

1.3 Cas de base.

- L'expression $\boxed{n == 1}$ est la condition de terminaison.
- L'instruction $\boxed{return(l\_decimal);}$ est le cas de base.
- L'instruction $\boxed{return(l\_decimal + (hexa\_dec\_rec(l\_nextCase, l\_position)));}$ est l'appel récursif.
  ▷ Puisqu'on dispose d'une invocation de la fonction **hexa_dec_rec()** sur des paramètres effectif différent
  des paramètres formel **(l_position < n && l_nextCase > hexa)**
  Dit autrement, chaque appel récursif tend vers le cas de base.

1.4 Cas récursif.

Nos cas récursif sont donc **l_position** et **l_nextCase** car :

La conversion de **hexa** vaut **l_decimal** si **n = 1**. Sinon, c'est **l_decimal** additioné par la conversion de **l_nextCase** dont la taille restante vaut **l_position**.

Mathématiquement, cela donne :

$$(l\_decimal)_{10} = \begin{cases} l\_decimal & \text{si n = 1} \\ l\_decimal + hexa\_dec\_rec(l\_nextCase, l\_position) & \text{sinon.} \end{cases}$$

2 Spécifications.

Une spécification se définit en deux temps :

La PréCondition implémente les suppositions. Elle caractérise donc les conditions initiales du module, les propriétés que doivent respecter les valeurs en entrée du module.
Elle se définit donc sur les paramètres formels (qui seront initialisés avec les paramètres effectifs).
La PréCondition doit être satisfaite avant l'exécution du module.

La PostCondition implémente les certifications. Elle caractérise les conditions finales du résultat du module. Dit autrement, la PostCondition décrit le résultat du module sans dire comment il a été obtenu. La PostCondition sera satisfaite après l'invocation.

- PréCondition $\equiv (hexa \neq NULL) \wedge n > 0, n \in \mathbb{N} \wedge hexa[0 \dots n-1]init \Rightarrow hexa\ init$
  - ▷ Dans ce cas de figure, ma fonction contient 2 assert qui sont représenté ci-dessus.
    La première condition est que mon String soit différent de null.
    Ensuite, il faut que mon 'n' soit strictement supérieur à 0 car dans le cas contraire cela veux dire que mon String ne contient aucun symbole.
    hexa doit bien entendu être initialisé. Cette notation est peut-être redondante mais je tiens tout de même à le préciser.

- PostCondition $\equiv (hexa\_dec\_rec)_{10} = Hexa\_Dec\_Rec(hexa, n)$
  $\wedge\ hexa = hexa_0 \wedge n \neq n_0$
  - ▷ Le résultat de ce module devra comprendre les certifications suivante :
    - ⋆ $hexa = hexa_0$ Ne signifie seulement que le String n'a pas changer.
      Cependant, nous déplaçons le pointeur sur le String.
    - ⋆ $n \neq n_0$ Dans ce cas de figure, nous ne modifions pas la taille du String mais nous décrémentons la valeur de n pour pouvoir parcourir hexa de la position 0 à la position $n-1$.

```
1 /*
2  * PréCondition    ≡ (hexa ≠ NULL) ∧ n > 0, n ∈ ℕ ∧ hexa[0 ... n − 1]init ∧ hexa[0 ... n − 1]init
3  *                ⇒ hexa init
4  *
5  * PostCondition   ≡ (hexa_dec_rec)10 = Hexa_Dec_Rec(hexa, n) ∧ hexa = hexa0 ∧ n ≠ n0
6  */
7 unsigned int hexa_dec_rec(char *hexa, int n);
```

Extrait de Code 1 – Spécification

3 Construction Récursive.

Vu que notre fonction **hexa_dec_rec()** est une fonction récursive qui s'appuie sur une structure conditionnelle, alors il nous faut une approche constructive en trois étapes

1. Programmation défensive.
2. Cas de base.
3. Cas récursif.

3.1 Programmation défensive.

Il s'agit d'une vérification de la PréCondition.

```
unsigned int hexa_dec_rec(char *hexa, int n) {
    assert(hexa);
    assert(n > 0);

     /* PréCondition ≡ (hexa ≠ NULL) ∧ n > 0, n ∈ ℕ ∧ hexa[0...n − 1]init ∧ hexa[0...n − 1]init
                  ⇒ hexa init */

    ... /* Reste du code */
}
```

Extrait de Code 2 – hexa_dec_rec()

3.2 Cas de base.

On gère le cas de base où $n = 1$. Juste avant, on gère les cas précis de la préconditon[2]

```
    /* PréCondition ≡ (hexa ≠ NULL) ∧ n > 0, n ∈ ℕ ∧ hexa[0...n − 1]init ∧ hexa[0...n − 1]init
                  ⇒ hexa init */

    if (n == 1)
        /* {n = 1 ⇒ hexa ≠ NULL} */
        return (l_decimal);
        /* {n = n0 ∧ hexa = hexa0 ∧ hexa_dec_rec = l_decimal} */
        /* {⇒ POSTCONDITION} */
```

Extrait de Code 3 – Cas de base

3.3 Cas récursif.

On gère le cas de base où $n > 1$. Juste avant, on gère les cas précis du cas de base [3.2] et de la précondition[2]

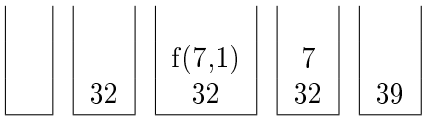
```
    /* PréCondition ≡ (hexa ≠ NULL) ∧ n > 0, n ∈ ℕ ∧ hexa[0...n − 1]init ∧ hexa[0...n − 1]init
                  ⇒ hexa init */

    /* {n > 1 ⇒ hexa ≠ NULL} */
    return (l_decimal + (hexa_dec_rec(l_nextCase, l_position)));
    /* {n ≠ n0 ∧ hexa = hexa0 ∧ hexa_dec_rec = l_decimal + (hexa_dec_rec(l_nextCase, l_position))} */
    /* {⇒ POSTCONDITION} */
```

Extrait de Code 4 – Cas récursif

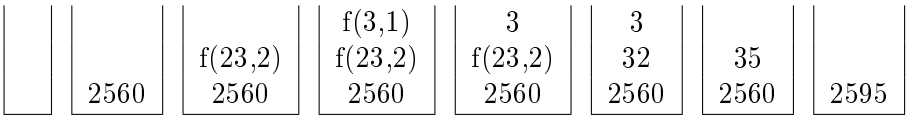
4 Traces d'Exécution.

Soit l'exemple **hexa_dec_rec(char *hexa, int n)** noté **f(s,n)**
On doit indiquer la trace de l'algorithme, sur la Pile, pour la récursivité suivante :
l_decimal + (hexa_dec_rec(l_nextCase, l_position))
Ce qui donne l'expression sous forme postfixée : **f(s,n) f(s1+,n1-)+**

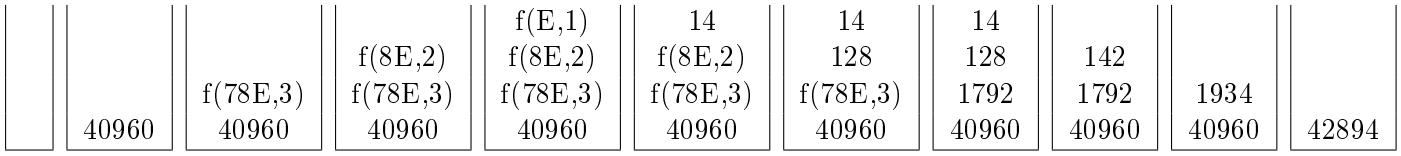
- Soit l'exemple **f('27', 2)**



- Soit l'exemple **f('A23', 3)**



- Soit l'exemple **f('A78E', 4)**



5 Complexité.

```c
/**
 * @fn unsigned int hexa_dec_rec(char*, int)
 * @brief Convertir un nombre hexadécimal en nombre décimal
 *
 * @pre ≡ (hexa ≠ NULL) ∧ n > 0, n ∈ ℕ ∧ hexa[0...n − 1]init ∧ hexa[0...n − 1]init
 *       ⇒ hexa init
 * @post ≡ (hexa_dec_rec)10 = Hexa_Dec_Rec(hexa, n) ∧ hexa = hexa0 ∧ n ≠ n0
 * @param hexa, le nombre hexadécimal
 * @param n, le nombre de symboles dans le nombre hexadécimal
 * @return la représentation décimale du nombre hexadécimal.
 */
unsigned int hexa_dec_rec(char *hexa, int n) {
   assert(hexa);
   assert(n > 0);

   char *l_nextCase;
   unsigned int l_convert;
   unsigned int l_position;
   unsigned int l_decimal;
   unsigned int l_exposant;

   l_position = n - 1;
   l_nextCase = hexa + 1;
   l_convert = convert(*hexa);
   l_exposant = hexa_power(l_position);
   l_decimal = l_convert * l_exposant;

   if (n == 1)
      return (l_decimal);

   return (l_decimal + (hexa_dec_rec(l_nextCase, l_position)));
}
```

Extrait de Code 5 – hexa_dec_rec()

```c
   l_position = n - 1;                    //
   l_nextCase = hexa + 1;                 //
   l_convert = convert(*hexa);            // T(a)
   l_exposant = hexa_power(l_position);   //
   l_decimal = l_convert * l_exposant;    //
```

Extrait de Code 6 – T(a)

```c
   if (n == 1)                    // T(a')
      return (l_decimal);         //
```

Extrait de Code 7 – T(a')

```c
   return (l_decimal + (hexa_dec_rec(l_nextCase, l_position))); // T(s+1,n+1)
```

Extrait de Code 8 – T(s+1,n+1)

Soit $T_{(n)}$, le coût d'un appel à $hexa\_dec\_rec(s, n)$.
En ce qui concerne le cas de base, on va dire qu'il prend $T_{(a)}$ opérations, $T_{(a)}$ étant constant.
Pour ce qui est du cas récursif, il prend également $T_{(a)}$ opérations ainsi que le nombre d'opérations
nécessaires par l'appel récursif, c'est-à-dire $T_{(s+1,n-1)}$. Il vient donc le sustème suivant :

$$T_{(n)} = \begin{cases} T_{(a)} + T_{(a')} & \text{si n = 1} \\ T_{(a)} + T_{(s+1,n-1)} & \text{sinon.} \end{cases}$$

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

$$\begin{aligned} T_{(n)} &= T_{(a)} + T_{(s+1,n-1)} \\ &= T_{(a)} + T_{(a)} + T_{(s+2,n-2)} \\ &= T_{(a)} + T_{(a)} + T_{(s+2,n-2)} \\ &= 2T_{(a)} + T_{(s+2,n-2)} \\ &= 3T_{(a)} + T_{(s+3,n-3)} \\ &= \dots \\ &= k \times T_{(a)} + T_{(s+k,n-k)} \end{aligned}$$

$T_{(n-k)}$ est une formulation tout à fait générale du temps d'exécution du k^e appel récursif. On ne connait qu'une valeur particulière de $T_{(\cdot)}$: $T_{(1)} = 0$. On va donc essayer de faire apparaitre cette valeur particulière. On a :

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

On insère cette valeur dans $k \times T_{(a)} + T_{(s+k,n-k)}$, et il vient :

$$T_{(n)} = (n-1) \times T_{(a)} + T_{(s+(n-1))} \in \mathcal{O}_{(n)}$$

Nous avons donc une complexité d'ordre Linéaire.

6 Dérécursification.

6.1 Algorithme de transformation non-terminale.

Voici les étapes à suivre pour transformer une fonction récursive non-terminale en une fonction récursive terminale :

1. Vérifier que l'opérateur α est commutative et associative.
   - commutatif
     - ▷ $x + y = y + x$
   - associatif
     - ▷ $x + (y + z) = (x + y) + z$

2. Créer une fonction λ pour transformer la récursivité non-terminale en récursivité terminale, avec plus de paramètres (dont un (ou plusieurs) accumulateur·s).

```
λ(String h, int n, int nAccu):
```

Extrait de Code 9 – lambda_1

3. Déterminer les paramètres effectifs de λ : la fonction qui appelle λ lui passe ses paramètres et initalise les accumulateurs ;

```
λ(String h, int n, int nAccu):
      c;
      e;
      a;
      d;
```

Extrait de Code 10 – lambda_2

4. Écrire le corps de la fonction λ :
   - Les résultat à renvoyer lors du cas de base dépend maintenant de l'(des) accumulateur·s ;
   - Les opérations intermédiaires sont maintenant effectuées sur l'(les) accumulateur·s.

```
λ(String h, int n, int nAccu):

      c ← convert(h[0]);
      e ← hexa_power(nAccu - 1);
      a ← c * e;

      if( n = 1)
         then
            d ← c;
         else
            d ← λ( h+1, n-1, a);
```

Extrait de Code 11 – lambda_3

Les conditions de terminaisons et des différents cas récursifs ne changent pas.
Ensuite, il faut dérécursiver λ en se référant aux étapes de l'algorithme de transformation terminale.

6.2 Algorithme de transformation terminale.

1. Commencer la définition d'une fonction ayant les mêmes paramètres formels, PRÉCONDITION et POSTCONDITION que le fonction d'origine.

```
/*
 * PréCondition    ≡ (hexa ≠ NULL) ∧ n > 0, n ∈ ℕ ∧ hexa[0...n − 1]init ∧ hexa[0...n − 1]init
 *                 ⇒ hexa init
 *
 * PostCondition   ≡ (hexa_dec_rec)10 = Hexa_Dec_Rec(hexa, n) ∧ hexa = hexa0 ∧ n ≠ n0
 */
f'(String hexa, int n):
```

Extrait de Code 12 – terminale_1

2. Pour chaque variable, introduire une nouvelle variable synonyme.

```
f'(String hexa, int n):

    m ← n;
    g ← hexa;
    a ← 0;
```

Extrait de Code 13 – terminale_2

3. Commencer la rédaction de la boucle **until** en prenant comme Critère d'Arrêt, la condition terminaison. Dans la condition, il faut évidemment remplacer les variables par leurs synonymes.

```
f'(String hexa, int n):

    m ← n;
    g ← hexa;
    a ← 0;

    until m = 1 do

    end
```

Extrait de Code 14 – terminale_3

4. On poursuit par l'écriture du Corps de la Boucle. On recopie les opérations faites sur les variables avant l'appel récursif avec les synonymes des variables.
5. On finit le Corps de la Boucle par les instructions de progression. En identifiant, dans l'appel récursif les modifications qui sont effectuées sur chaque variables lors de l'appel et les recopier en fin de boucle.

```
f'(String hexa, int n):

    m ← n;
    g ← hexa;
    a ← 0;

    until m = 1 do
        g ← g + 1;
        c ← convert(g[0]);
        m ← m - 1;
        e ← hexa_power(m);
    end
```

Extrait de Code 15 – terminale_4

6. Après la boucle, on retourne la valeur calculée par les opérations faites dans le cas de base de la fonction récursive. Sans oublier le changement de variables par les synonymes.

```
/*
 * PréCondition    ≡ (hexa ≠ NULL) ∧ n > 0, n ∈ ℕ ∧ hexa[0...n − 1]init ∧ hexa[0...n − 1]init
 *                 ⇒ hexa  init
 *
 * PostCondition   ≡ (hexa_dec_rec)_10 = Hexa_Dec_Rec(hexa, n) ∧ hexa = hexa_0 ∧ n ≠ n_0
 */
f'(String hexa, int n):

   m ← n;
   g ← hexa;
   a ← 0;

   until m = 1 do
      g ← g + 1;
      c ← convert(g[0]);
      e ← hexa_power(m - 1);
      m ← m - 1;
   end
   r ← convert(g[0]);
```

Extrait de Code 16 – fonction non récursive