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

Timothy Smeers, S200930

Table des matières

1 Formulation Récursive

1.1 Notations

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

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

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

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

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

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

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

$$l\_size(s) = \begin{cases} 0 & \text{si s = `` ``} \\ 1 + l\_size(cdr(s)) & \text{sinon.} \end{cases}$$

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

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

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 :

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

2 Spécification

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 $\equiv (hexa \neq NULL) \wedge n > 0, n \in \mathbb{N}$
  $\wedge\, (n =\mid \{hexa[0] - hexa[n]\} \mid) \Rightarrow (\{hexa[0], \ldots, hexa[n-1]\} \in \{0, \ldots, 9, A, \ldots, F\}$
  $\wedge\, (hexa[n] = \backslash 0))$
  - ▷ Dans ce cas de figure, ma fonction contient 4 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. Le n doit ensuite être égale à l'amplitude de l'ensemble $\{hexa[0], \ldots, hexa[n]\}$. Cela implique que l'intervale $\{hexa[0], \ldots, hexa[n-1]\}$ doit être compris dans les symboles de l'hexadécimal et le symbole à la position $hexa[n]$ devra être égale à $'\backslash 0'$

- PostCondition $\equiv N = \sum_{i=0}^{n-1} a_i \times b^i, a_i \in \{0, 1, \ldots, b-1\} \wedge N \in (\mathbb{N})_{16}$

  Un système de numération d'un nombre naturel N est constitué
  1. D'une base b $\in \mathbb{N}, (b > 1)$
  2. D'un ensemble de b symboles, appelés $chiffres$, compris entre 0 et $b - 1$.

  La représentation d'un naturel N via le système de numération repose sur la structure ci-dessus
  - ▷ $b = 16$ et l'ensemble des symboles est $\{0, 1; \ldots, 9, A, \ldots, F\}$

```
/*
 * PréCondition ≡ (hexa ≠ NULL) ∧ n > 0
 *   ∧(n =| {hexa[0] − hexa[n]} |) ⇒ ({hexa[0], . . . , hexa[n − 1]} ∈ {0, . . . , 9, A, . . . , F}
 *   ∧(hexa[n] = \0))
 * PostCondition ≡ hexa_dec_rec = Hexa_Dec_Rec(hexa, n) ∧ hexa = hexa0 ∧ n = n0
 */
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);

    unsigned int l_size;
    unsigned int l_convert;

    l_size = size_char(hexa);
    l_convert = convert(hexa[0]);

    assert(n == l_size);
    assert(l_convert != -1);

     /* PréCondition ≡ (hexa ≠ NULL) ∧ n > 0
     ∧(n =| {hexa[0] − hexa[n]} |) ⇒ ({hexa[0], ..., hexa[n − 1]} ∈ {0, ..., 9, A, ..., F}
     ∧(hexa[n] = \0)) */

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

Extrait de Code 2 – hexa_dec_rec()

```c
static unsigned int size_char(char *p_hexa) {

   unsigned int i;
   i = 0;

   while (p_hexa[i])
      i++;

   return (i);
}
```

Extrait de Code 3 – size_char()

Cette fonction statique permet de Vérifiez toute position du String *p_hexa* en incrémentant i pour chaque valeur du String $\neq$ '\0'

	0	i	'\0'
p_hexa	Déjà vérifié	Reste à vérifié	
	Incrémente i		

```c
static unsigned int convert(char hex) {
   switch (hex) {
   case '0':
      return (0);
   case '1':
      return (1);
   case '2':
      return (2);
   case '3':
      return (3);
   case '4':
      return (4);
   case '5':
      return (5);
   case '6':
      return (6);
   case '7':
      return (7);
   case '8':
      return (8);
   case '9':
      return (9);
   case 'A':
      return (10);
   case 'B':
      return (11);
   case 'C':
      return (12);
   case 'D':
      return (13);
   case 'E':
      return (14);
   case 'F':
      return (15);
   }
   return (-1);
}
```

Extrait de Code 4 – convert()

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 =| {hexa[0] − hexa[n]} |) ⇒ ({hexa[0], ..., hexa[n − 1]} ∈ {0, ..., 9, A, ..., F}
   ∧(hexa[n] = \0)) */

if (n == 1)
   /* {n = 1 ⇒ hexa ≠ NULL} */
   return (l_decimal);
   /* {n = n_0 ∧ hexa = hexa_0 ∧ hexa_dec_rec = l_decimal} */
   /* {⇒ POSTCONDITION} */
```

Extrait de Code 5 – 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 =| {hexa[0] − hexa[n]} |) ⇒ ({hexa[0], ..., hexa[n − 1]} ∈ {0, ..., 9, A, ..., F}
   ∧(hexa[n] = \0)) */

/* {n > 1 ⇒ hexa ≠ NULL} */
return (l_decimal + (hexa_dec_rec(l_nextCase, l_position)));
/* {n ≠ n_0 ∧ hexa = hexa_0 ∧ hexa_dec_rec = l_decimal + (hexa_dec_rec(l_nextCase, l_position))} */
/* {⇒ POSTCONDITION} */
```

Extrait de Code 6 – Cas récursif

4 Traces d'Exécution

					7		
		4		9	9	16	
	20	20	80	80	80	80	5

5 Complexité

6 Dérécursification