Université de Liège

INFO0947

Compléments de Programmation

Un exercice dont vous êtes le Héros · l'Héroïne

Élimination de la Récursivité – Renversement d'une Chaine

Simon Liénardy Benoit Donnet

4 mai 2020

Préambule

Exercices

Dans ce « TP dont vous êtes le Héros · l'Héroïne », nous vous proposons de suivre pas à pas la résolution de deux exercices. L'un sur un problème concernant la recherche du nombre d'occurrences d'un caractère donné dans une chaîne de caractères et l'autre sur le renversement d'une chaîne de caractères.

Attention ! Dans ce chapitre, nous utilisons un *pseudo-langage* pour exprimer la récursivité et son élimination. Tout le correctif utilise ce pseudo-langage. Les caractéristiques de ce langage sont résumées à la Sec. 9.1. Imprégnez-vous bien de ce pseudo-langage. Il vous sera utile pour le troisième projet.

Il est dangereux d'y aller seul [1] !

Partagez vos commentaires, questions, solutions alternatives sur le forum eCampus. N'hésitez jamais !

1. Référence vidéoludique bien connue des Héros.

9.1 Pseudo-langage

Le cours théorique et ce TP dont vous êtes le Héros·l'Héroïne utilisent un *pseudo-langage*. En voici les principales caractéristiques :

Pointeurs

On ne se préoccupe pas des pointeurs. Un objet qui devrait normalement être stocké en mémoire grâce à un pointeur est juste manipulé grâce à une variable qui le représente. On ne se focalise pas sur les pointeurs dans ce chapitre. On supprime donc le surplus de difficulté apporté par les pointeurs pour se concentrer sur la dérécursification.

Affectation

On utilise la flèche vers la gauche (←) comme symbole d'affectation. La variable `r` est spéciale : à la fin de l'exécution, elle contient le résultat qui a été calculé. Une conséquence d'utiliser ← comme symbole d'affectation est de ne pas permettre les sucres syntaxiques tels que +=.

```c
x = 5;
r = x;
```

Code C

```
x ← 5;
r ← x;
```

Pseudo-Code

Instruction `if`

Les instructions à effectuer si la condition est vraie sont introduites par le mot réservé **`then`**, qui est indenté par rapport au **`if`**. Les instructions sont également toutes indentées par rapport au **`then`**. Les instructions d'un même bloc possèdent la même indentation. Les instructions à effectuer si la condition est fausse sont introduites par le mot réservé **`else`** et suivent des règles d'indentation similaires.

```c
if(condition){
   instructions_true
}
else{
   instructions_false
}
```

Code C

```
if(condition)
   then
      instructions_true
   else
      instructions_false

```

Pseudo-Code

Boucle `until`

Au lieu de préciser le gardien d'une boucle, on précise directement son critère d'arrêt. La boucle s'appelle donc une boucle **`until`** qui signifie « jusqu'à » en anglais. Le corps de la boucle est encadré par les mots réservés **`do`** et **`end`**. Le critère d'arrêt est précisé entre les mots **`until`** et **`do`**.

```c
while(gardien){
   instr_corps
   ...
   instr_corps
}
```

Code C

```
until !gardien do
   instr_corps
   ...
   instr_corps
end
```

Pseudo-Code

C'est bien le critère d'arrêt (i.e., la négation logique du gardien) qui est mentionné après le mot-clé **`until`**.

Définition de fonction

On écrit d'abord le nom de la fonction. La restriction sur les caractères propre au langage C est levée : on peut tout à fait appeler une fonction λ ou ajouter des ′ au nom des fonctions. Tout cela incite à la plus grande créativité [2]. Les paramètres formels (type + nom) sont mentionnés entre deux parenthèses. La signature se finit par le symbole « **:** ». Le corps de la fonction est indenté par rapport à la signature d'un niveau d'indentation [3]. Le type de retour de la fonction ne doit pas être précisé [4].

```c
int ma_fonction(char c, char *s)
{
   instruction
   ...
   instruction
}
```

Code C

```
ma_fonction'(char c, String s):
   instruction
   ...
   instruction
```

Pseudo-Code

Autre

Les autres types de boucles et instructions ne sont pas précisés. Par exemple, rien n'est dit sur l'instruction **`switch`**. Dans le cas où on en aurait besoin, on veillerait à rester proche de la syntaxe déjà introduite. Par exemple, les différents **`case`** seraient indentés par rapport au **`switch`** et les instructions d'un même bloc posséderaient la même indentation.

2. N'abusez pas non plus !
3. Tout se qu'on apprécie dans le Python, en somme.
4. Attention, si on regarde le Slide 8 du Chapitre 9, on se rend compte que la spécification de la fonction `pgcd` indique le type de retour (i.e., `pgcd(int a, int b) = (int r)`). Il s'agit là de la *déclaration explicite* de la fonction (cfr. INFO0946, Chapitre 6, Slide 33). Dans ce TP, on ne s'intéresse qu'à des *déclarations implicites* qui n'exigent pas d'information sur le type de retour.

9.2 Énoncé

Soit l'opération `reverse` qui retourne la chaine passée en argument mais à l'envers. La définition est la suivante :

```
reverse: String → String
```

▷ **Exercice** Spécifiez et construisez une implémentation récursive de l'opération `reverse()`.
▷ **Exercice** Éliminez la récursivité dans l'algorithme obtenu à l'exercice 1. Attention, il n'est pas question ici de fournir un algorithme itératif mais bien d'éliminer la récursivité comme cela a été vu au cours.

9.2.1 TAD `String`

L'objectif de cet exercice est de travailler sur les chaines de caractères. Pour ce faire, nous disposons d'un type abstrait `String` dont voici les opérations de base[5] (nous n'indiquons pas ici les axiomes liés à ces opérations, ce n'est pas important pour les exercices qui suivent) :

```
Type:
  String
Utilise:
  Integer, Char, Boolean
Operations:
  empty_string: → String
  is_empty: String → Boolean
  first: String → Char
  remainder: String → String
  adj: String × Char → String
  last: String → Char
  except_last: String → String
Préconditions:
  ∀ s ∈ String
  first(s) is defined iff is_empty(s) = False
  remainder(s) is defined iff is_empty(s) = False
  last(s) is defined iff is_empty(s) = False
```

`remainder()` est l'opération qui enlève le premier caractère de la chaine (en paramètre) et retourne les caractères restants. Par exemple, `remainder("toto")` retourne "`oto`". `adj` est l'opération de concaténation d'un caractère en fin de chaine. `last()` est l'opération qui retourne le dernier caractère de la chaine. `except_last()` est l'opération qui retourne la chaine amputée de son dernier caractère.

9.2.2 Méthode de résolution

Nous allons suivre l'approche constructive vue au cours. Pour ce faire nous allons :

1. Formuler le problème récursivement (Sec. 9.3) ;
2. Spécifier le problème complètement (Sec. 9.4) ;
3. Construire le programme complètement (Sec. 9.5) ;
4. Rédiger le programme final (Sec. 9.6) ;
5. Éliminer la Récursivité (Sec. 9.7) ;
6. Rédiger le programme final, sans récursivité (Sec. 9.8).

5. Par rapport au Chap. 9, Slide 14, les opérations `car` et `cdr` sont appelées ici respectivement `first` et `remainder`. L'opération `cons` n'a pas d'équivalent puisque `adj` ajoute en fin de chaine et non en début.

9.3 Formulation récursive

Tout d'abord, il convient de formaliser le problème de manière récursive.

Indice : relisez bien l'énoncé avant de progresser dans la suite !

Si vous voyez directement comment procéder, voyez la suite 9.3.3
Si vous êtes un peu perdu, voyez le rappel sur la récursivité 9.3.1
Si vous vous souvenez du rappel mais que vous avez besoin d'un indice sautez en 9.3.2

9.3.1 Rappels sur la Récursivité

9.3.1.1 Formulation Récursive

On ne va pas se cacher derrière son petit doigt, c'est la principale étape et la plus difficile. Les étapes suivantes de résolution sont rendues beaucoup plus simples dès que la formulation récursive est connue. Comment procéder ?

En fait, il y a deux choses à fournir pour définir récursivement quelque chose :

1. Un cas de base ;
2. Un cas récursif.

Trouver le **cas de base** n'est pas toujours facile, certes, mais ce n'est pas le plus compliqué. En revanche, cerner le **cas récursif** n'est pas une tâche aisée pour qui débute dans la récursivité. Pourquoi ? Parce que c'est complètement contre-intuitif ! Personne ne pense récursivement de manière innée. C'est une *manière de voir le monde*[6] qui demande un peu d'entrainement et de pratique pour être maitrisée. Heureusement, **vous êtes au bon endroit** pour débuter ensemble.

Toutes les définitions récursives suivent le même schéma :

> Un X, c'est *quelque chose* **combiné** avec un X *plus simple*.

Il faut détailler :

— Quel est ce *quelque chose* ?
— Ce qu'on entend par **combiné** ?
— Ce que signifie plus simple ?

Si on satisfait à ces exigences, on aura défini X récursivement puisqu'il intervient lui-même dans sa propre définition.

Un exemple mathématique, pour illustrer ce canevas de définition récursive :

> n ! $= n \times$ (n-1) !

On veut définir ici la factorielle de n ($n!$). On dit que c'est n (notre *quelque chose*) combiné par la **multiplication** à une factorielle plus simple : $(n-1)!$

9.3.1.2 Programmatiquement parlant

Par définition, un programme récursif s'appelle lui-même, sur des données *plus simple*. La tâche du programmeur consiste donc à déterminer comment **combiner** le résultat de l'appel récursif sur ces données plus simples avec les paramètres du programme pour obtenir le résultat voulu.

Dans la rédaction d'un programme récursif, on a donc **toujours** accès à un sous-problème particulier : le programme lui-même. Il convient par contre de respecter deux règles (cf. slides Chap. 4, 23 → 26) :

Règle 1 Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif ;

Règle 2 Tout appel récursif doit se faire avec des données plus « proches » de données satisfaisant une condition de terminaison

6. On dit aussi *paradigme*.

Les cas non-récursifs sont appelés **cas de base**.

Les conditions que doivent satisfaire les données dans ces cas de base sont appelées **conditions de terminaison**.

9.3.1.3 Construction récursive d'un programme

Pour construire un programme récursif, il convient de respecter ce schéma. Notez bien qu'ici n'interviendra pas la notion d'invariant tant qu'il n'y a pas de boucle en jeu [7] !

1. Formuler récursivement le Problème ;
2. Spécifier le problème. Le plus souvent, il suffit d'utiliser la formulation récursive préalablement établie ;
3. Construire le programme final en suivant l'approche constructive. La plupart du temps, c'est trivial, il suffit de suivre la formulation récursive.

Et les sous-problème dans tout ça ? Si le problème principal semble trop compliqué, il se peut qu'une découpe en sous-problèmes soit pertinente. Ce sont alors ces sous-problèmes qui seront récursifs et dont le programme pourra être rédigé en suivant les trois points ci-dessus.

De toute façon, chaque programme récursif possède au moins un sous-problème : lui-même ! Par définition de la récursion. Attention : par l'approche constructive, avant d'appeler n'importe quel sous-problème, il convient d'assurer que sa précondition est respectée. Après l'appel, par l'approche constructive, on peut conclure que sa postcondition est respectée .

9.3.2 Indice – String et récursivité

Un bon moyen de procéder est de définir récursivement une chaine de caractères et d'exprimer son renversement en construisant, récursivement, la chaine à l'envers.

Suite de l'exercice

À vous ! Formalisez le problème de manière récursive et passez à la Sec. 9.3.3.

7. Peut-on mêler Invariant et récursion ? Bien sûr : souvenez-vous en lorsqu'on vous parlera de l'algorithme de tri `quicksort`.

9.3.3 Mise en commun de la formulation récursive

Nous observons, ceci :

> Une chaine de caractères, c'est *un premier caractère* **concaténé à** une chaine de caractères plus courte.

Nous avons donc défini la structure « chaine de caractères » récursivement. Nous allons donc formuler le renversement d'une chaine de caractères sur base de cette définition récursive.

Appelons notre notation $Reverse(s)$, avec s une chaine de caractères.

Cas de base Une chaine de caractères vide est une chaine de caractères. Quel est le résultat de l'inversion d'une chaine vide ? Toujours une chaine vide.

$$Reverse(\text{“”}) = \text{“”}$$

Cas récursif Pour déterminer le cas récursif, observons un exemple :

"ANIMAL" $\rightarrow$ "LAMINA"

Imaginons que l'on découpe la chaine "ANIMAL" selon la définition récursive, on a

"A" || "NIMAL" $\rightarrow$ "LAMIN" || "A"

On voit qu'il faut ajouter le premier caractère à l'inversion du reste de la chaine. cela donne le cas récursif suivant :

$$Reverse(s) = adj(Reverse(remainder(s)), first(s))$$

Synthèse La formulation récursive de $Reverse$ est donc la suivante :

$$Reverse(s) = \begin{cases} empty\_string() & \text{si is\_empty(s)} \\ adj(Reverse(remainder(s)), first(s)) & \text{sinon.} \end{cases}$$

9.4 Spécification

Il faut maintenant spécifier le problème.

Pour un rappel sur la spécification, voyez la Section 9.4.1
La correction de la spécification est disponible à la Section 9.4.3

9.4.1 Rappel sur les spécifications

Voici un rappel à propos de ce qui est attendu :

La Précondition C'est une condition sur les données. Pour l'établir, il faut se demander quelles doivent être les données du problème. Ici, dans notre exemple, cela parle de tableau. Quelles conditions devons-nous imposer sur ce tableau ? N'hésitez pas à vous référer au cours INFO0946, Chapitre 6, Slides 54 → 64).

La Postcondition C'est une condition sur le résultat du problème, si tant est que la Précondition est remplie (si tel n'est pas le cas, on pratique la *programmation défensive* (voir INFO0946, Chapitre 6, Slides 65 → 70) et on arrête l'exécution du programme.

La signature est souvent nécessaire lors de l'établissement de la Postcondition des fonctions (voir INFO0946, Chapitre 6, Slide 11). En effet, le résultat retourné par l'invocation de la fonction est représenté par le nom de la fonction dans la Postcondition. Le nom de la fonction doit donc être défini avant d'en parler.

9.4.2 Suite de l'exercice

Spécifiez le problème. Rendez-vous à la Sec. 9.4.3 pour la correction !

9.4.3 Spécification du problème

Précondition Normalement, si on manipulait des pointeurs, on devrait vérifier que celui représentant la chaine est valide. Puisqu'on n'en manipule pas dans notre pseudo-langage simplifié [8], on passe outre.

Pre : /

Postcondition On réutilise la notation précédemment introduite :

$Post$: `reverse` = $Reverse(s)$

Alerte : Studentite chronique !

Si votre Postcondition n'est pas aussi simple que celle ci-dessus, c'est que soit :

- Vous êtes sadomasochiste et vous aimez vous faire du mal. On ne vous juge pas, rassurez-vous ;
- Vous avez brûlé des étapes dans les précédents TP. Retournez donc à la case « TP dont vous êtes le Héros·l'Héroïne – Récursivité » sans passer par la case "Départ" et sans toucher F 4.000 !
- Vous êtes atteint de studentite chronique. Vous cherchez systématiquement midi à quatorze heure. Certains aspects du cours sont complexes mais cela ne veut pas dire que tous les points d'une résolution d'exercice doivent être extrêmement compliqués. Calmez-vous. Inspirez. Attendez un peu. Expirez. Quand une tâche vous parait suspicieusement complexe, éloignez-vous un moment de l'exercice et faites autre chose. Revenez-y par la suite. Changer d'activité pourrait vous redonner de l'inspiration.

Signature

```
1 reverse(String s):
```

8. Voir le rappel sur le pseudo-langage.

9.5 Construction du programme par l'approche constructive

Il faut maintenant construire le programme en suivant *l'approche constructive*. La section suivante vous fournit un petit rappel.

Le corrigé est disponible à la Sec. 9.5.2.

9.5.1 Approche constructive et récursion

Les principes de base de l'approche constructive ne changent évidemment pas si le programme construit est récursif :

- Il faut vérifier que la **Précondition** est respectée en pratiquant la **programmation défensive** ;
- Le programme sera construit autours d'une instruction conditionnelle qui discrimine le·s cas de base des cas récursifs ;
- Chaque **cas de base** doit amener la **Postcondition** ;
- **Le cas récursif** devra contenir (au moins) un **appel récursif** ;
- Pour tous les appels à des sous-problème (y compris l'appel récursif), il faut vérifier que la **Précondition** du module que l'on va invoquer est respectée
- Par le principe de l'approche constrictive, la **Postcondition** des sous-problèmes invoqués est respectée après leurs appels.

Il faut respecter ces quelques règles en écrivant le code du programme. La meilleure façon de procéder consiste à suivre d'assez près la formulation récursive du problème que l'on est en train de résoudre. Le code est souvent une « traduction » de cette fameuse formulation. Il ne faut pas oublier les **assertions intermédiaires**. Habituellement, elles ne sont pas tellement compliquées à fournir.

Qu'en est-il de `INIT`, `B`, `CORPS` et `FIN` ?

Essayez de suivre ! Ces zones correspondent au code produit lorsqu'on écrit une boucle ! S'il n'y a pas de boucle, il n'y aura pas, par exemple, de gardien de boucle !

Et la fonction de terminaison ?

Là non plus, s'il n'y a pas de boucle, on ne peut pas en fournir une. **Par contre**, on s'assure qu'on a pas oublié le·s **cas de base** et que les différents **appels récursifs** convergent bien vers un des cas de base (la convergence vers un des cas de base est appelée *condition de terminaison* – Chapitre 4, Slides 22 → 27).

Suite de l'exercice

À vous maintenant de construire le programme par l'approche constructive. Un corrigé est disponible à la Sec. 9.5.2.

9.5.2 Approche constructive

9.5.2.1 Programmation défensive

Il n'y a pas de Précondition

9.5.2.2 Cas de base

On renvoie la chaine vide si la chaine d'entrée est vide.

```
1 reverse(String s):
2    // {Pre}
3    if (is_empty(s))
4    then
5       // {s = ""}
6       r ← empty_string();
7       // {Post}
```

9.5.2.3 Cas récursifs

On suit [9] la formulation récursive de *Reverse*. On vérifie les Préconditions de `first` et `remainder` mentionnées dans la définition du TAD String puisque `s` n'est pas la chaine vide.

```
1    else
2       // {s ≠ "" ⇒ Pre_first ∧ Pre_remainder}
3       // {Pre_REC ∧ Pre_adj}
4       r ← adj(reverse(remainder(s)), first(s));
5       // {(Post_first ∧ Post_remainder ∧ Post_adj ∧ Post_REC) ⇒ Post}
```

Suite de l'exercice

Le programme final est donné dans la section suivante. Il sera utile pour l'élimination de la récursivité.

9. Le code fonctionne si `remainder` envoie une copie du reste de `s` sans modifier `s`, sinon il faudrait calculer les paramètres de `adj` avant son appel et l'ordre des opérations importerait. Idem pour `adj` qui doit envoyer une copie de la chaine après ajout. On voit bien que si on devait manipuler des pointeurs et gérer la mémoire – en particulier, éviter les fuites –, tout ceci serait plus compliqué !

9.6 Programme final

```
1  reverse(String s):
2     // {Pre}
3     if (is_empty(s))
4     then
5        // {s = ""}
6        r ← empty_string();
7        // {Post}
8     else
9        // {s ≠ "" ⇒ Pre_first ∧ Pre_remainder}
10       // {Pre_REC ∧ Pre_adj}
11       r ← adj(reverse(remainder(s)), first(s));
12       // {(Post_first ∧ Post_remainder ∧ Post_adj ∧ Post_REC) ⇒ Post}
```

9.7 Élimination de la Récursivité

Marche à suivre

Relisez attentivement le rappel sur la dérécursivation 9.7.1
Éliminez ensuite la récursivité du programme présenté à la section 9.6 9.7.2

9.7.1 Rappel

9.7.1.1 Principes de base

On peut classer les fonctions récursives en deux catégories : les fonctions récursives terminales et les autres. [10]

> *Une fonction récursive terminale est une fonction récursive dans laquelle les appels récursifs sont exécutés uniquement en* ***dernière instruction*** *de* ***tous*** *les cas récursifs.*

Cette distinction permet de décrire aisément la démarche de dérécursivation d'une fonction récursive :

1. Si la fonction à dérécursiver est récursive terminale, il suffit d'appliquer un algorithme de dérécursivation.
2. Si la fonction à dérécursiver n'est pas récursive terminale :
   - On peut essayer de l'écrire sous forme d'une fonction récursive terminale suivant certaines conditions et on applique ensuite l'algorithme.
   - Si les conditions ne sont pas remplies pour déterminer une fonction récursive terminale, on applique le cas général qui consiste à simuler la pile d'appel des fonctions.

9.7.1.2 À partir d'une fonction récursive terminale

La « *dérécursivation* » d'une fonction récursive terminale est aisée [11] : il suffit de suivre l'algorithme décrit ci-dessous [12].

Soit une fonction récursive terminale tout à fait générale décrite dans le code ci-dessous, à gauche. Il convient d'identifier plusieurs éléments :

x : la variable sur laquelle est basée la récursion. Elle fait partie des paramètres formels et c'est sur base de sa valeur que l'on discrimine les cas de base des cas récursifs.

cond(x) : La condition selon laquelle on doit appliquer le cas de base (i.e., la *condition de terminaison*) ;

base(x) : Ce sont les opérations à effectuer lorsqu'on atteint le cas de base et dont le résultat est retourné par la fonction. On note ces opérations comme un appel à une fonction appelée « base » ;

Traitement(x) : Ce sont des opérations que l'on peut effectuer avant l'appel récursif. On les notes ici comme un appel à une fonction appelée « Traitement » ;

Appel récursif : Pour dérécursiver, autant détecter où sont les appels récursifs. Il convient de bien vérifier que cet appel est **dans tous les cas** la dernière instruction exécutée. On vérifie donc que notre fonction est bien récursive terminale ;

progression(x) : Comme paramètre effectif de l'appel récursif, on détermine les opérations qui sont effectuées sur les différentes variables. On note ici cette opération comme l'appel à une fonction « progression ».

10. Voir Chapitre 4, Slides 33 → 41.
11. Voir Chapitre 9, Slides 12 → 18.
12. `gcc` implémente un tel algorithme. Pour lui demander d'optimiser votre code pour décérursiver vos fonction terminales, il faut ajouter l'option `-foptimize-sibling-calls`, qui sera ajouté automatiquement si l'une des options d'optimisation `-O2` ou `-O3` est activée.

```
1 f(x):
2    if(cond(x))
3       then
4          r ← base(x);
5       else
6          Traitement(x);
7          r ← f(progression(x));
```

Code récursif

```
1 f'(x):
2    u ← x;
3    until cond(u) do
4       Traitement(u);
5       u ← progression(u);
6    end
7    r ← base(u);
```

Code dérécursivé

Algorithme de transformation Dès que les différents éléments ont été repérés, on peut écrire la boucle sans Invariant, en modifiant seulement l'ordre des éléments identifiés comme suit (les numéros de ligne font référence au code écrit à droite ci-dessus) :

1. Commencer la définition d'une fonction ayant les mêmes paramètres formels, Précondition et Postcondition que la fonction d'origine (lg. 1) ;
2. Pour chaque variable, introduire une nouvelle variable synonyme (lg. 2) ;
3. Commencer la rédaction de la boucle **`until`** en prenant comme condition d'arrêt la condition terminaison précédemment identifiée. Dans la condition, il faut évidemment remplacer les variables par leurs synonymes introduits au point précédent. Dans le cas où il y aurait plusieurs cas de base (et donc plusieurs conditions de terminaison), la condition d'arrêt de la boucle **`until`** est la combinaison de toutes ces conditions au moyen d'un OU logique ($\vee$) (lg. 3) ;
4. On poursuit par l'écriture du corps de la boucle. On recopie les opérations faites sur les variables avant l'appel récursif. On n'oublie pas les changements de variables ! (lg. 4) ;
5. On finit le corps de la boucle par les instructions de progression. Il faut identifier, dans l'appel récursif, les modification qui sont effectuées sur chaque variable lors de l'appel et les recopier en fin de boucle. Prenons, par exemple, une fonction à trois arguments :

```
1 f(x, y, z):
2    ...
3    f(x - 1, y * 2, z / 3);
```

Dans la fonction dérécursivée, on aura évidemment remplacé `x`, `y` et `z` par de nouvelles variables. On identifie la transformation effectuée sur chaque paramètre : `x` est décrémenté de 1, `y` est multiplié par 2, `z` est divisé par 3. Il vient le code dérécursivé suivant :

```
1  f'(x, y, z):
2     u ← x;
3     v ← y;
4     w ← z;
5     until (...) do
6        ...
7        u ← u - 1;
8        v ← v * 2;
9        w ← w / 3;
10    end
11    ...
```

On fait subir aux variables remplaçant `x`, `y` et `z` les **mêmes** transformations précédemment identifiées. S'il y a plusieurs cas récursifs différents, ils sont distingués dans la fonction récursive par des conditions. On recopie l'instruction conditionnelle dans le corps de la boucle et pour chaque cas récursif, on fait progresser les variables, dans chaque clause de l'instruction conditionnelle, suivant leurs modifications respectives lors des différents appels récursifs (lg. 5) ;

6. Après la boucle, on retourne la valeur calculée par les opérations faites dans le cas de base de la fonction récursive. Comme toujours, on n'oublie pas le changement de variables. S'il y avait plusieurs cas de base, il faut les discriminer grâce à leurs conditions respectives.

Exemple (Chap. 9, Slide 17) : déterminer si un caractère appartient à la chaine

```
is_member(Char c, String s):
   if(is_empty(s))
      then r ← False;
   if((car(s) = c)
      then r ← True;
      else r ← is_member(c, cdr(s));
```

```
is_member'(Char c, String s):
   k ← c;
   l ← s;
   until is_empty(l) ∨  car(l) = k do
      k ← k;
      l ← cdr(l);
   end
   if(is_empty(l))
      then r ← False;
   if((car(l) = k)
      then r ← True;
```

1. On a deux paramètres : `c` et `s`. On nomme la fonction `is_member'`.
2. On introduit deux variables `k` et `l`, mécaniquement et sans se poser de question.
3. On a identifié deux conditions de terminaison que l'on combine avec un $\vee$.
4. Il n'y a pas de traitement avant l'appel récursif (voir la remarque 2).
5. Lors de l'appel récursif, `c` n'est pas modifié et le deuxième paramètre est `cdr(s)`. On a donc deux instructions à écrire (dont une qui ne fait rien [13]).
6. Après la boucle, on doit retourner un résultat. Vu qu'il y avait deux cas de base, on les discrimine à l'aide de leur condition respective.

Remarque 1 : Un bon compilateur parviendrait évidemment à simplifier le code obtenu : il détecterait que `k` (donc `c`) n'est pas modifié et que l'instruction de choix final peut être simplifiée en « **`if ... then ... else`** » mais cela ne fait pas, à proprement parler, partie de l'algorithme de dérécursivation.

Remarque 2 : Si on considère que le cas de base est juste le test de la chaine vide et que le reste fait partie du cas récursif, on obtient la transformation suivante, qui est sémantiquement équivalente à l'autre, bien entendu :

```
is_member(Char c, String s):
   if(is_empty(s))
      then r ← False;
   if(car(s) = c)
      then r ← True;
      else r ← is_member(c, cdr(s));
```

```
is_member"(Char c, String s):
   k ← c;
   l ← s;
   until is_empty(l) do
      if(car(l) = k)
         then r ← True;
      k ← k;
      l ← cdr(l);
   end
   r ← False;
```

9.7.1.3 Obtenir une fonction récursive terminale à partir d'une fonction récursive non-terminale

Si la fonction n'est pas récursive terminale, cela veut dire que le résultat de l'appel récursif est combiné avec une autre valeur dans une expression :

```
f(x):
   if(cond(x))
      then
         r ← base(x);
      else
         Traitement(x);
         r ← valeur(x) α f(progression(x));
```

13. On applique aveuglément l'algorithme, on est pas censé réfléchir à ce stade

Dans le code ci-dessus, on a appelé « α » l'opérateur qui combine la valeur obtenue par l'appel récursif et l'autre valeur.

Voici les étapes à suivre pour transformer une fonction récursive non-terminale en une fonction récursive terminale [14] :

1. Vérifier que l'opération α est commutative et associative, c'est à dire que :
   - $x \,\alpha\, y = y \,\alpha\, x$
   - $x \,\alpha (y \,\alpha\, z) = (x \,\alpha\, y) \,\alpha\, z$
2. Créer une fonction λ pour éliminer la récursivité, avec plus de paramètres (dont un (ou plus) *accumulateur·s*) L'accumulateur, comme son nom l'indique, sert à **accumuler** des résultats au fil des appels récursifs. Les calculs sont donc maintenant effectués lors des appels récursifs (i.e., *descente récursive* et non plus après ceux-ci (i.e., *remontée récursive*) ;
3. Déterminer les paramètres effectifs de λ : la fonction qui appelle λ lui passe ses paramètres et initialise les accumulateurs ;
4. Écrire le corps de la fonction λ :
   - Le 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.

   Les conditions de terminaisons et des différents cas récursifs ne changent pas.

Ensuite, Il faut dérécursiver λ en se référant aux étapes présentées précédemment.

Exemple (Chap. 9, Slide 26) : factorielle

```
1 fact(n):
2    if(n = 0)
3       then
4          r ← 1;
5       else
6          r ← n * fact(n-1)
```

```
1 λ(int n, int acc):
2    if(n = 0)
3       then
4          r ← acc;
5       else
6          r ← λ(n-1, acc * n);
7
8 fact(n):
9    r ← λ(n, 1);
```

1. L'opération de multiplication est bien commutative et associative ;
2. On crée une fonction lambda avec un accumulateur, astucieusement nommé « `acc` » ;
3. Puisque l'accumulateur va accumuler des produits, on l'initialise avec la valeur d'un produit à 0 terme (i.e., le neutre pour la multiplication : 1). Une autre façon de procéder est de d'abord écrire le corps de λ et le comparer ensuite à celui de `f` pour constater que `acc` doit être initialisé à 1 (on peut le voir en inspectant leurs cas de base respectifs).
4. Le cas récursif consiste à ajouter un terme dans le produit contenu dans l'accumulateur. Le cas de base consiste à renvoyer la valeur accumulée.

Remarque : La Postcondition de la fonction λ est la suivante :

$$Post : \lambda = acc * n!$$

Comparer cette Postcondition et celle de `fact` ($Post_{\text{fact}} = n!$) permet de voir aisément que l'accumulateur doit être initialisé à 1 pour que ces deux fonctions calculent le même résultat.

14. Voir Chapitre 9, Slides 20 → 24

9.7.1.4 Cas général : Simuler la Pile d'appels

Attention : En lisant la suite, faites bien attention à voir la distinction entre **code appelant** et **code appelé**.

Pour simuler la pile d'appels, il faut se souvenir de ce qu'il se passe lors d'un appel de fonction :

1. Sauvegarde sur la Pile du contexte du code appelant ;
2. Création d'un contexte pour le code appelé ;
3. Copie des paramètres effectifs sur la Pile ;
4. Exécution du code appelé ;
5. Retour au code appelant, récupération de la valeur de retour ;
6. Destruction du contexte du code appelé ;
7. Restauration du contexte du code appelant.

La Pile possède trois opérations principales [15] : `Push`, `Top` et `Pop` :

Push Correspond aux moments où des valeurs sont mises sur la Pile :
- — la sauvegarde du contexte avant un appel
- — la création du contexte et le passage des paramètres

Top Correspond aux moments où des valeurs sont lues sur la Pile :
- — la récupération de la valeur des paramètres lors de l'exécution du code appelé.
- — la restauration du contexte lors du retour au code appelant.

Pop Correspond au moment où de l'espace est libéré sur la Pile :
- — destruction du contexte d'un code appelé dont l'exécution est terminée.
- — la destruction de l'espace nécessaire à la sauvegarde du contexte du code appelant.

Prenons en exemple cette fonction récursive. Les symboles `cond(•)`, `base(•)`, `progression(•)` et `operation(•)` représentent des **expression** [16] tout à fait générales :

```
1  f(int x):
2     // pc = 1
3     if(cond(x))
4        then
5           r ← base(x);
6        else
7           tmp ← f(progression(x));
8           // pc = 2
9           tmp ← f(progression(x));
10          r ← operation(tmp, x);
```

Le code de cet exemple a été divisé en 2 zones : lorsque la fonction `f` est appelée et qu'elle commence à s'exécuter, la prochaine instruction est la conditionnelle de la ligne 3. Le code qui est en train de s'exécuter a la main jusqu'à l'appel récursif de la ligne 7. Avant de passer la main au code appelé par l'appel récursif,

15. Voir Chapitre 7 et l' « exercice dont vous êtes le Héros · l'Héroïne » associé.
16. Ça n'a pas changé depuis septembre, on parle toujours de la description de la façon dont on calcule une valeur. Par exemple, `cond(x)` peut représenter la condition `x == 1`

il faut sauvegarder le contexte du code appelant **en particulier**, l'adresse de la prochaine instruction (celle de la ligne 9 – stocker la valeur de retour de l'appel récursif dans `tmp` : la dernière ligne du bloc rose et la première du bloc lavande ne forment en réalité qu'une seule instruction : `tmp` ← `f(...);`).

Nous allons donc ajouter une variable qui représente l'information sur la prochaine instruction à effectuer. Appelons cette variable `pc`[17]. Inutile de s'encombrer en stockant des adresses dans `pc` : on considère que `pc` = 1 quand on entre dans la fonction et `pc` = 2 au retour de l'appel récursif, comme mentionné en commentaire dans le code ci-dessus.

Nous allons simuler la Pile d'appels. Nous allons stocker dessus des contextes. Dans notre exemple, nous avons deux variables : `x` et `pc`. Nous stockerons donc sur la Pile des doublets du type : (x, pc)[18].

En d'autres termes :

— Empiler un doublet (x, 1) revient à empiler le paramètre `x` et à demander un appel récursif.

— Empiler un doublet (x, 2) revient à sauvegarder la valeur de `x` et à se souvenir qu'il faut reprendre l'exécution de la fonction à la ligne 9).

La conversion de la fonction récursive en boucle 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             // Début du bloc rose
17             if (cond(x))
18                then
19                   r ← base(x);
20                else
21                   // Avant l'appel récursif, on se souvient qu'il faudra continuer dans le bloc
22                   // lavande au retour de l'appel.
23                   s ← push(s, (x, 2))
24                   // On empile ensuite l'appel récursif
25                   s ← push(s, (progression(x), 1))
26             // Fin du bloc rose
27          else // pc = 2, donc
28             // Début du bloc lavande
29             // On récupère la dernière valeur de retour
30             tmp ← r;
31             // On retourne le résultat calculé
32             r ← operation(tmp, x);
33             // fin du bloc lavande
34    end
35
36    // la variable r contient donc la valeur finale
```

17. Cette variable joue exactement le même rôle que le registre PC (pour *Program Counter*) que vous avez vu en Organisation des Ordinateurs (INFO0061, Slide 107), d'où le nom.

18. Ceci est évidemment une simplification de l'utilisation de la pile mais nous pouvons nous y tenir : nous n'avons pas besoin de simuler l'entièreté du comportement réel de la pile lors d'un appel ou d'un retour de fonction

Exemple (Chap. 9, Slides 34 → 41) : croissance des nénufars

```
1  // pc = 1
2  nenufar(int n):
3     if(n = 1)
4        then
5           r ← 1.5;
6        else
7           tmp ← nenufar(n - 1);
8  // pc = 2
9           tmp ← nenufar(n - 1);
10          r ← 2 * tmp - 0.5;
```

On obtient le code suivant :

```
1  nenufar'(int n):
2     s ← empty_stack();
3     s ← push(s, (n, 1));
4
5     until is_empty(s) do
6        (x, pc) ← top(s);
7        s ← pop(s);
8
9        if(pc = 1)
10          then
11             // Début du bloc rose
12             if (n = 1)
13                then
14                   r ← 1.5;
15                else
16                   s = push(s, (n, 2))
17                   s = push(s, (n - 1, 1))
18             // Fin du bloc rose
19          else // pc = 2
20             // Début du bloc lavande
21             tmp ← r;
22             r ← 2 * tmp - 0.5;
23             // fin du bloc lavande
24    end
25    // la variable r contient donc la valeur finale
```

Différence avec le Slide 39 Ce code est différent de celui disponible sur le Slide 39. On peut voir qu'on empile systématiquement deux contextes d'affilée aux lignes 16 et 17 et qu'on dépile systématiquement le contexte où `pc` $= 1$ au début de l'itération suivante. En définitive, seuls les contextes où `pc` $= 2$ vont rester longtemps sur la Pile. Dans le Slide 39, on n'empile que ces contextes (correspondant donc à la ligne 16), puis on applique le cas de base et ensuite, on dépile tous les contextes en appliquant le bout de code correspondant à `pc = 2` dans notre version. Les deux solutions sont évidemment équivalentes.

Suite de l'exercice

Appliquez maintenant la dérécursivation à la fonction `reverse`. Une mise en commun est disponible à la section 9.7.2

9.7.2 Élimination de la récursivité de la fonction `reverse`

9.7.2.1 Observation du code à dérécursiver

```
reverse(String s):
   if (is_empty(s))
   then
      r ← empty_string();
   else
      r ← adj(reverse(remainder(s)), first(s));
```

On note les éléments suivants :

— La fonction n'est pas récursive terminale ;

— Inutile de se demander si la fonction `adj` est commutative ou associative : ce n'est déjà **pas** une **opération interne** sur un ensemble.

Nous en concluons qu'il faut appliquer le cas général de dérécursivation.

9.7.2.2 Simulation de la pile

On va réécrire le code pour mettre en évidence l'appel récursif (cette modification ne fonctionne que si remainder ne modifie pas son paramètre puisque `first(s) est appelé par la suite`). On va aussi indiquer les endroits où le contrôle change entre le code appelant et le code appelé (en rose et en lavande, parce que c'est relativement joli). On a renommé `s` en `c` parce que `s` sera utilisé pour la pile (Stack).

```
reverse(String c):
   // pc = 1
   if (is_empty(c))
   then
      r ← empty_string();
   else
      tmp ← reverse(remainder(c));
      // pc = 2
      tmp ← reverse(remainder(c));
      r ← adj(tmp, first(c));
```

Le code qui simule la pile va empiler des doublets (c, pc) composé d'une chaine et d'une variable `pc` représentant la prochaine instruction à réaliser :

`pc = 1` signifie une entrée dans la fonction `reverse`.

`pc = 2` signifie un retour depuis un appel récursif vers le code appelant.

On va obtenir le code suivant :

```
1  reverse'(String c):
2     // Déclaration et initialisation de la pile :
3     s ← empty_stack();
4
5     // On place le premier appel à la fonction. On entre en premier dans la zone rose où pc = 1.
6     s ← push(s, (c, 1));
7
8     until is_empty(s) do
9        // On dépile toujours le premier contexte sur la pile
10       (c, pc) ← top(s);
11       s ← pop(s);
12
13       // On choisit les opérations à effectuer suivant la valeur de pc :
14       if (pc = 1)
15       then
16          // Début du bloc rose
17          if (is_empty(c))
18          then
19             r ← empty_string();
20          else
21             // On va devoir effectuer l'appel récursif
22             // On se souvient d'abord qu'il faudra continuer dans le bloc lavande :
23             s ← push(s, (c, 2));
24             // On empile ce qui correspond à l'appel récursif :
25             s ← push(s, (remainder(c), 1));
26          // Fin du bloc rose
27       else // i.e. pc = 2
28          // Début du bloc lavande
29          // r contient le résultat de l'appel récursif
30          tmp ← r;
31          // On retourne le résultat calculé
32          r ← adj(tmp, first(c));
33          // Fin du bloc lavande
34    end
35    // r contient la chaine inversée
```

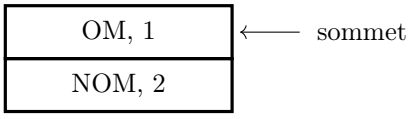
On voit dans ce code que l'on va garder longtemps sur la pile des doublets où `pc = 2` (Les doublets où `pc = 1` sont empilés à une itération et dépilés à l'itération suivante. Au dernier appel récursif, on aura `remainder(c) = ""`. On effectuera le cas de base (i.e. ligne 19) qui retournera une chaine vide. Les itérations suivantes vont consister à dépiler les doublets où `pc = 2` et à concaténer un caractère supplémentaire dans la chaine résultat.

Illustration avec une chaine courte

1.

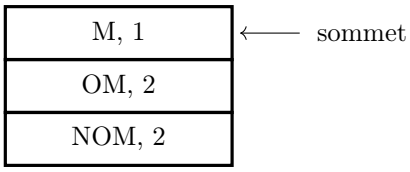
→ On commence par empiler la chaine de base et `pc = 1`

2.



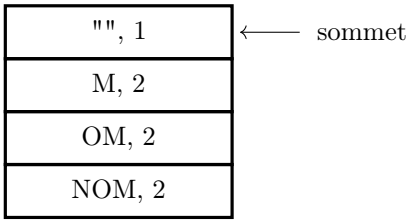
→ On dépile un doublet et on en rempile 2.

3.



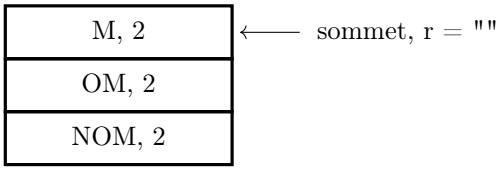
→ On dépile un doublet et on en rempile 2, encore

4.



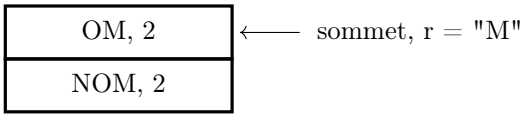
Cet appel (ci-dessus) va mettre la valeur de `r` à `""` (c'est le cas de base).

5.



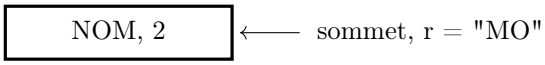
→ r sera ensuite égal à "M"

6.



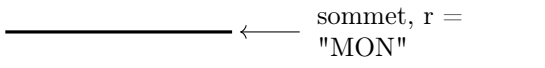
→ r sera ensuite égal à "MO"

7.



→ r sera ensuite égal à "MON"

8.



→ r content bien l'inversion de la chaine.

9.8 Programme final sans récursivité

```
reverse'(String c):

   s ← empty_stack();
   s ← push(s, (c, 1));

   until is_empty(s) do
      (c, pc) ← top(s);
      s ← pop(s);

      if (pc = 1)
      then

         if (is_empty(c))
         then
            r ← empty_string();
         else
            s ← push(s, (c, 2));
            s ← push(s, (remainder(c), 1));
      else
         tmp ← r;
         r ← adj(tmp, first(c));
   end
   // r contient la chaine inversée
```

Alerte : Fakenews !

Est-ce que Simon est coupable de **Fakenews** ? Ou serait-ce une *erreur pédagogique* utilisée avec brio pour illustrer un concept ?

En effet, il est possible d'écrire une fonction λ pour la dérécursivation bien que `adj` ne soit pas commutative ni associative ! Ce n'est juste pas **automatique** tels que présentés dans les algorithmes plus haut. La commutativité et l'associativité de l'opération sont des conditions suffisantes pour pouvoir trouver une fonction λ mais pas des conditions nécessaires !

Fonction λ pour `reverse` [19]

Prenons un exemple de tranformation de chaine de caractères et d'un accumulateur qui contiendrait une chaine partiellement inversée. On se rappelle que `adj` ne permet que des ajouts à la fin. Donc on commence par supprimer le dernier caractère de notre chaine :

Chaine	Accumulateur
ANIMAL	""
ANIMA	L
ANIM	LA
ANI	LAM
AN	LAMI
A	LAMIN
""	LAMINA

Pour obtenir la découpe ANIMAL → ANIMA || L, on utilise les fonctions `except_last` et `last`. Il vient cette fonction :

19. Je ne numérote pas ces sections pour ne pas divulgâcher par une consultation de la table des matières...

```
λ(String s, String acc):
   if (is_empty(s))
   then
      r ← acc;
   else
      r ← λ(exept_last(s), adj(acc, last(s)));

reverse(String s):
   r ← λ(s, empty_string)
```

Qui est tout à fait récursive terminale ! Pour le cas de base, l'illustration ci-dessus montre pourquoi il faut retourner l'accumulateur quand la chaine s est vide. La dérécursivation est donnée dans la suite :

```
λ(String s, String acc):
   if (is_empty(s))
   then
      r ← empty_string();
   else
      r ← λ(exept_last(s),
            adj(acc, last(s)));
```

```
λ'(String s, String acc):
   k ← s;
   a ← acc;
   until is_empty(k) do
      a ← adj(a, last(k));
      k ← except_last(k);
   end
   r ← a;
```

Code dérécursivé pour **reverse**

```
reverse'(String s):
   k ← s;
   a ← empty_string();
   until is_empty(k) do
      a ← adj(a, last(k));
      k ← except_last(k);
   end
   r ← a;
```