Découpage d'un algorithme en sous-algorithmes (fonctions)

Gilles Trombettoni

IUT MPL-Sète, département info

Développement initiatique

Septembre-octobre 2021

Jeu d'instructions dans les langages de programmation

Rappel des instructions disponibles dans un langage de programmation :

- entrées-sorties ;
- variables et structures de données plus sophistiquées (dont les tableaux) ;
- affectations de variables ;
- instructions conditionnelles (`si alors sinon`) ;
- instructions répétitives (boucles) ;
- **appel à d'autres algorithmes**

  (sous-programmes, procédures, fonctions, méthodes, routines,...)

  $\rightarrow$ **une infinité de sous-programmes possibles existent.**

Rôle des fonctions

- **simplifier** : découper un algorithme complexe en sous-algorithmes plus simples ;
- **travailler localement** : transmettre les données entre fonctions permet à chacune d'elles de travailler en autonomie, avec ses variables locales ;
- **réutiliser** : ne pas réécrire un même algorithme plusieurs fois.

Exemple : appel à la fonction `estBissextile`

Consigne

- Pour une date donnée par son jour, son mois et son année, on doit écrire un algorithme `joursDate` qui calcule le nombre de jours écoulés entre le 1^{er} janvier de cette année et cette date. (C'est un des exercices posés en TD.)
- Pour écrire un algorithme correct, on voudrait utiliser l'algorithme `bissextile` déjà écrit.
- Solution : on va transformer l'algorithme `bissextile` en une fonction `estBissextile` qui prend une année en paramètre et retourne un booléen :

```
fonction estBissextile (an : entier) retourne booleen
```

Fonction `estBissextile`

Définition de la fonction f_2 = `estBissextile`

```
fonction estBissextile (an : entier) retourne booleen
// Résultat: vrai si l'année 'an' est bissextile, faux sinon
Variables // aucune !
Debut
  retourne an%400==0 ou an%4==0 et an%100!=0
Fin estBissextile
```

Appel de la fonction f_2 = `estBissextile` dans f_1 = `joursDateBidon`

```
Algo joursDateBidon
Variables
  res : booleen ; annee : entier
Debut
  res <- estBissextile(1728)
  si res alors
     si estBissextile(1908) alors ... finSi
  finSi
  afficher("Une année SVP ? "); saisir(annee)
  si estBissextile(annee) alors
     afficher("Algo complètement bidon")
  finSi
Fin joursDateBidon
```

Algorithme communiquant

Algorithme solitaire devient **fonction** sociable

Une fonction (procédure) :

- possède du code (des instructions) pour
  - faire des entrées-sorties
  - modifier la valeur de ses variables locales (calculs intermédiaires)
- communique avec d'autres algorithmes.

Remarque

Cette définition de fonction est un peu réductrice car les langages existants n'ont pas que des variables locales, pour des raisons d'efficacité.

On peut néanmoins imaginer un vrai langage (qui compile) de ce type (appelé langage fonctionnel). Ce sera en tout cas notre langage algorithmique pendant un mois !

Les fonctions dans la mémoire (vive)

Représentation simplifiée d'une fonction en mémoire

Un espace mémoire est alloué pour chaque fonction.
Il se divise en deux parties :

- une zone permet de stocker les **données** :
  valeurs des variables locales et des paramètres
- une zone pour le **code** : instructions

La réalité est plus complexe

- Tous les processus exécutables (gérés par le système d'exploitation) possèdent plusieurs zones contiguës en mémoire, dont
- une zone rassemblant les codes de toutes les fonctions et
- une zone rassemblant les valeurs des paramètres et variables locales de toutes les fonctions, zone appelée **pile**.
- Détails dans les cours de système.

La représentation simplifiée suffit à bien comprendre l'exécution d'une fonction.

Echange de données entre fonctions

Principe d'échange de données entre fonctions

- Quand une fonction f_1 **appelle** une autre fonction f_2, il y a échange de données entre ces deux fonctions.

  (f_1 est la fonction **appelante** et f_2 est la fonction **appelée**.)

- La fonction f_1 envoie des informations (valeurs) à la fonction f_2 qui les reçoit en entrée.
- A la fin de son exécution, f_2 renvoit une information en sortie. Cette valeur est récupérée par la fonction appelante f_1 qui continue son traitement.

Ecriture d'une fonction

Syntaxe

```
fonction f2 (<var1>: <type1>, <var2>: <type2>, ...)
            retourne <type3>
Variables
  <Declaration d'autres variables>
Debut
  <instructions>
  retourne <valeur>
Fin f2
```

Sémantique

1. **Entrée** : Des valeurs sont transmises (par une fonction appelante) dans les paramètres `<var1>`, `<var2>`, etc.
2. f_2 s'exécute comme un algorithme, en utilisant éventuellement les valeurs des **paramètres** `<var1>`, `<var2>`, etc.
3. **Sortie** : Une valeur (`<valeur>`) est renvoyée à la fonction/algorithme appelant(e).

Ecriture d'une fonction

Remarque 1

A l'intérieur de la fonction, les paramètres sont considérés :

- comme des variables locales ;
- qui ont déjà une **valeur** ;
- qu'il vaut mieux éviter de modifier.

Il s'agit du **passage de paramètres** (≡ transmission d'informations entre fonctions) par **valeurs** : des valeurs sont **copiées** dans des paramètres qui sont des variables locales.

Remarque 2 : on pourrait se passer d'algorithmes !

On pourrait supprimer le mot-clé `Algo`. On peut voir l'algorithme principal comme une fonction principale (`main`) qui appelle d'autres fonctions.

Attention

Il peut y avoir plusieurs instructions `retourne` dans une fonction. Non recommandé car chaque `retourne` interrompt la fonction brutalement.

Appel de fonction

Définition : Un **appel de fonction** (appelée) est une instruction possible dans une autre fonction (appelante).

Syntaxe :

```
fonction f1 (...) retourne ...
Variables
  resultat : <type3>
Debut
  ...
  resultat <- f2(<val1>, <val2>, ...) // <== APPEL DE FCT
  ...
Fin
```

Exemple 1 : appel de la fonction `estBissextile`

Appel de la fonction f_2 = `estBissextile` dans f_1 = `joursDateBidon`

```
Algo joursDateBidon
Variables
  res : booleen ; annee : entier
Debut
  res <- estBissextile(1728)
  si res alors
     si estBissextile(1908) alors ... finSi
  finSi
  afficher("Une année SVP ? "); saisir(annee)
  si estBissextile(annee) alors
     afficher("Algo complètement bidon")
  finSi
Fin joursDateBidon
```

Définition de la fonction f_2 = `estBissextile`

```
fonction estBissextile (an : entier) retourne booleen
// Résultat: vrai si l'année 'an' est bissextile, faux sinon
Variables // aucune !
Debut
  retourne an%400==0 ou an%4==0 et an%100!=0
Fin estBissextile
```

Exemple 2 : algorithme `plage`

Pas de faux espoir, on n'ira pas à la plage ! ;-)

Algorithme

```
Algorithme plage
 // Action: saisit un tableau de 100 entiers,
 //      calcule et affiche le plus petit
 //      et le plus grand de ces nombres.
Variables
  i, mini, maxi : entier
  tab           : tableau de 100 entiers
Debut
  tab <- saisirTabEntiers(100) // appel de fonction
  mini <- minimum(tab)  // appel de fonction
  maxi <- maximum(tab)  // appel de fonction
  afficher("Les nombres sont compris dans la plage",
           "[", mini, " , ", maxi, "]");
Fin plage
```

Exemple : fonction `saisirTabEntiers`

```
fonction saisirTabEntiers (n : entier)
                          retourne tableau d'entiers
// Résultat : un tableau de n entiers
//            rempli par l'utilisateur.
Variables
  i : entier
  t : tableau de n entiers
Début
  pour i dans 0..(n-1) faire
     afficher("Donner un entier ")
     saisir(t[i])
  finPour
  retourne t
Fin saisirTabEntiers
```

Exemple : fonction `minimum`

```
fonction minimum (t : tableau d'entiers)
                 retourne entier
// Resultat : le plus petit élément du tableau t
Variables
  i, min   : entier
Debut
  min <- t[0]
  pour i dans 1..(t.longueur-1) faire
     si t[i] < min alors
        min <- t[i]
     finSi
  finPour
  retourne min
Fin minimum
```

Exemple : fonction `maximum`

```
fonction maximum (t : tableau d'entiers)
                 retourne entier
// Resultat : le plus grand élément du tableau t
Variables
  i, max   : entier
Debut
  max <- t[0]
  pour i dans 1..(t.longueur-1) faire
     si t[i] > max alors
        max <- t[i]
     finSi
  finPour
  retourne max
Fin maximum
```

Exemple : fonction `maximum` (version 2)

```
fonction maximum(t: tableau d'entiers) retourne entier
// Résultat : le plus grand élément du tableau t
// Stratégie : utilise la fonction minimum en prenant les
               valeurs opposées des éléments de t
Variables
  i         : entier
  tOpposes : tableau de t.longueur entiers
Debut
  pour i dans 0..(t.longueur-1) faire
     tOpposes[i] <- - t[i]
  finPour
  retourne - minimum(tOpposes)
Fin maximum
```

Questions

- Avantages et désavantages de cette version 2 ?
- Quelle est la trace des appels de fonctions dans l'algo `plage` ?

Tri par sélection

Le tri par sélection est un algorithme simple (et inefficace) pour trier des nombres entre eux (par ordre croissant).

```
Algo triSelection
// Stratégie : parcourt le tableau de gauche à droite ;
//       à chaque itération i, échange l'élément i avec
//       le plus petit élément à droite de i.
Variables
  tab : tableau de 100 entiers
  i, iMin, sauveMin : entier
Debut
  // Saisie du tableau tab :
  tab <- saisirTabEntiers(100)
  ...
```

Tri par sélection (suite)

```
  ...
  // Tri par sélection :
  pour i de 0 à 98 faire
     iMin <- indiceMinimum (tab, i)
     si iMin != i alors // echange tab[i] <-> tab[iMin]
        sauveMin <- tab[iMin]
        tab[iMin] <- tab[i]
        tab[i] <- sauveMin
     finSi
  finPour
  // Affichage du résultat :
  afficherTabEntiers(tab)
Fin triSelection
```

Tri par sélection (suite)

Fonction afficherTab

```
Fonction afficherTabEntiers(t : tableau d'entiers)
         retourne void
// Action : affiche les cases du tableau t
Variables
  i : entier
Debut
  pour i dans 0..(t.longueur-1) faire
     afficher (t[i], " ")
  finPour
  sautLigne()
Fin afficherTableauEntiers
```

Tri par sélection (suite)

Fonction `indiceMinimum`

```
fonction indiceMinimum (t: tableau d'entiers, j: entier)
                       retourne entier
// Résultat : l'indice du + petit elt de t à partir de j
// Pré-requis : 0 <= j < n
Variables
  i, iMin   : entier
Debut
  iMin <- j
  pour i dans (j+1)..(t.longueur-1) faire
     si t[i] < t[iMin] alors
        iMin <- i
     finSi
  finPour
  retourne iMin
Fin minimum
```

Question : doit-on écrire les deux fonctions `minimum` et `indiceMinimum` de ce cours ? Sinon, laquelle des deux ? ?

Procédures versus fonctions

- Certains langages (comme Pascal ou Ada) font la différence entre procédure et fonction.

  Ils proposent des mots-clefs différents pour les deux notions.

  Ils considèrent qu'une fonction ne doit pas faire d'**effet de bord**, c'est-à-dire peut seulement modifier son environnement local :

  - pas de modification de variables « globales » (cf. cours ultérieur)
  - pas d'entrées-sorties

- De nombreux langages considèrent simplement qu'une procédure est une fonction qui ne renvoie rien (`void`).

  Ce sera le cas dans le langage algorithmique.

- Exemple : `afficherTabEntiers` est une procédure (qui effectue des entrées-sorties).

Passage de paramètres par variable/pointeur/données-résultats

Patientons pour pouvoir écrire :

```
Algo triSelection
Variables
  tab : tableau de 100 entiers
  i, min : entier
Debut
  tab <- saisirTabEntiers(100)
  pour i de 0 à 98 faire
     min <- indiceMinimum (tab, i)
     echange(tab, i, min)
  finPour
  afficherTabEntiers(tab)
Fin triSelection
```

car la procédure `echange` modifie la variable `tab`.

Conclusion : rôle des fonctions

- **simplifier** : découper un algorithme complexe en sous-algorithmes plus simples ;
- **travailler localement** : transmettre les données entre fonctions permet à chacune d'elles de travailler en autonomie, avec ses variables locales ;
- **réutiliser** : ne pas réécrire un même algorithme plusieurs fois.

Les commentaires sont importants

Pourquoi des commentaires ?

- Un programme peut contenir des milliers de fonctions.
- Un logiciel peut être développé par des dizaines de programmeurs.
- Il faut tester le logiciel et donc vérifier que chaque fonction corresponde à sa spécification.

$\Rightarrow$ Très important de commenter ses programmes, en particulier ses fonctions.

Commentaires dans le langage algorithmique

- **Action** : rôle de la procédure
- **Résultat** : valeur renvoyée par la fonction
- **Stratégie** : principe algorithmique utilisé (en langage naturel)
- **Pré-requis** : hypothèses faites sur les valeurs des paramètres d'entrée ou les valeurs saisies (ex : variable *age* strictement positive).

  Inutile de tester les hypothèses !