Master Sciences, Technologies, Santé
Mention Mathématiques - spécialité Enseignement des mathématiques

Algorithmique et graphes, thèmes du second degré

Algorithmique
Premiers compléments

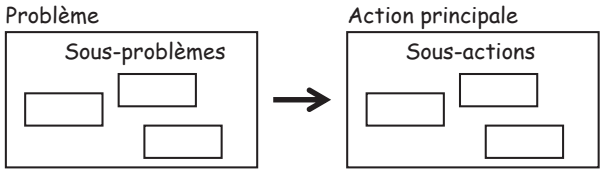
Éric SOPENA, eric.sopena@labri.fr

Première partie

Actions et fonctions

Principe d'analyse descendante

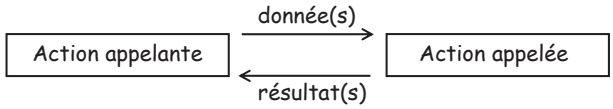
La conception d'un algorithme procède en général par des affinements successifs : on décompose le problème à résoudre en différents sous-problèmes, puis les sous-problèmes à leur tour si nécessaire, jusqu'à obtenir des problèmes "faciles à résoudre" (ou dont la solution est déjà connue).



La notion d'action

Une ***action*** permet de résoudre un problème bien défini ; elle fournit ainsi un (ou plusieurs) résultat(s), à partir de données.

Une telle action peut alors être utilisée (appelée) par n'importe quelle autre action (appelante).



Il est indispensable de bien préciser la nature des informations échangées (données et résultats).

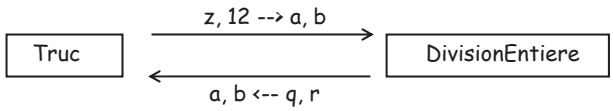
La notion d'action - Exemple

```
Action DivisionEntiere
       ( E a, b : entiers ; S q, r : entiers )
# cette action détermine le quotient q et le reste r
# de la division entière de a par b
début
    q = a div b
    r = a mod b
fin

Action Truc
... Entrer(z)
    DivisionEntiere ( z, 12, a, b )
```

- z, 12, a et b sont les ***arguments d'appels*** de l'action DivisionEntiere.
- a, b, q et r sont les ***paramètres d'entrée*** (a, b) et ***de sortie*** (q, r) de l'action DivisionEntiere

La notion d'action - Exemple

```
Action DivisionEntiere
       ( E a, b : entiers ; S q, r : entiers )
# cette action détermine le quotient q et le reste r
# de la division entière de a par b
début
    q = a div b
    r = a mod b
fin

Action Truc
... Entrer(z)
    DivisionEntiere ( z, 12, a, b )
```



La notion d'action - Exemple

```
Action DivisionEntiere
       ( E a, b : entiers ; S q, r : entiers )
# cette action détermine le quotient q et le reste r
# de la division entière de a par b
début
    q = a div b
    r = a mod b
fin

Action Truc
... Entrer(z)
    DivisionEntiere ( z, 12, a, b )
```

Un appel d'action s'utilise ainsi comme une nouvelle opération disponible dans notre langage algorithmique (nouvelle instruction en programmation).

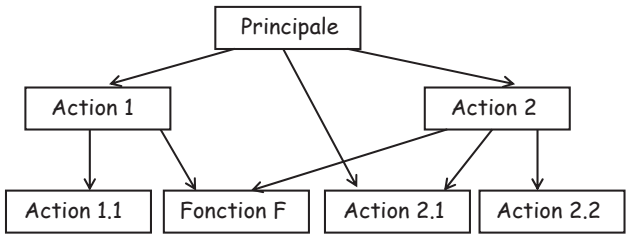
La notion de fonction

Une ***fonction*** n'est qu'une action particulière, ayant un seul paramètre résultat (de type précisé). La mise sous forme de fonction permet une utilisation plus "souple" : l'appel de fonction peut être utilisé dans toute expression de type compatible.

```
Fonction Maximum ( x, y : entiers ) : entier
# retourne le maximum des entiers x et y
début
    si ( x < y ) alors retourner (y) sinon retourner (x)
fin

Action Truc
...
a ← 2 * ( a + Maximum ( alpha, Maximum ( beta, 15 ) ) )
Afficher ( "resultat : ", Maximum ( a, b ) )
```

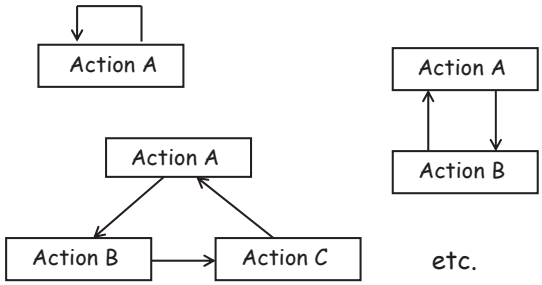

Graphe des appels

Un algorithme est donc composé d'un certain nombre d'actions ou fonctions, certaines en appelant d'autres. On peut alors définir le ***graphe des appels*** :

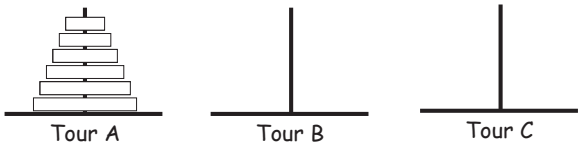


Récursivité

Un algorithme est ***récursif*** dès que le graphe des appels contient un ***circuit*** :



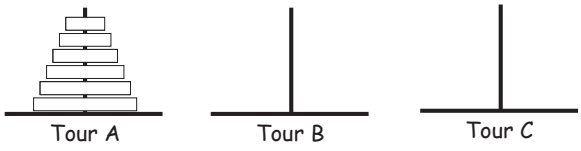
Récursivité - Exemple : les tours de Hanoï



Pour déplacer N disques de A vers C :

- déplacer N-1 disques de A vers B,
- déplacer le grand disque de A vers C
- déplacer N-1 disques de B vers C

Récursivité - Exemple : les tours de Hanoï




```
Action ToursHanoi
    ( E n : entier ; E tour1,tour2,tour3 : caractères )
début
    si ( n = 1 )
    alors  Afficher ( tour1, ' ---> ', tour3 )
    sinon  ToursHanoi ( n-1, tour1, tour3, tour2 )
           Afficher ( tour1, ' ---> ', tour3 )
           ToursHanoi ( n-1, tour2, tour1, tour3 )
    fin_si
fin
```

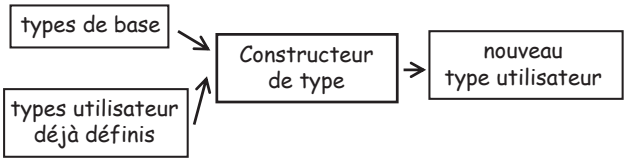
Deuxième partie

Constructeurs de types

Types de base vs types utilisateurs

Tout langage (algorithmique ou de programmation) offre un certain nombre de types de base prédéfinis (entiers, caractères, ...).

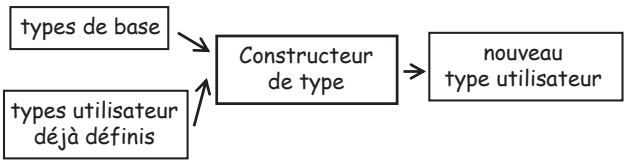
Il est généralement possible de bâtir ses propres types (types utilisateurs) à l'aide de ***constructeurs de type***.



Constructeurs de types

Un constructeur de type offre :

- une ***syntaxe*** pour définir un nouveau type,
- des ***primitives*** (opérations de base) pour manipuler les objets du type ainsi défini.



Principaux constructeurs de types

- ***Les intervalles***

```
Type   TMois = 1 .. 12
```

- ***Les énumérations***

```
Type   TJour = ( lun, mar, mer, jeu, ven, sam, dim )
```

- ***Les agrégats***

```
Type   TDate = agrégat {
          jour : TJour
          mois : TMois
          année : entier }
Variable
       d : TDate
...
    d.jour ← mar
```


Principaux constructeurs de types

- ***Les tableaux*** : un tableau est un ensemble (de taille fixée) de cellules contiguës repérées par un indice (qui démarre à 0).

```
Constante  MAX = 20
Type   TTableau : tableau de MAX entiers
Variables
       t1 : TTableau
       t2 : tableau de 100 caractères
       i : entier
...
    t1[0] ← 23
    pour i de 0 à 99
    faire  Afficher ( t2[i] )
    fin_pour
...
```


Principaux constructeurs de types

Un tableau est une structure statique, dont la taille maximale est fixée a priori.

Remarque. *Les listes AlgoBox sont en fait des tableaux dynamiques, dont la taille n'est pas bornée*

- ***Les listes*** : une liste est un ensemble de cellules repérées par un rang (qui démarre à 0), offrant des possibilités de mise à jour étendues.

```
Type   TListe = liste d'entiers
Variables
       liste1 : TListe
       liste2 : liste de chaînes
```

Principaux constructeurs de types

Les primitives offertes sur les listes permettent :

- de définir un contenu en extension :

```
liste1 ← [ 3, 1, 10, 3, 14, 7 ]
```

- d'accéder à un élément à partir de son rang :

```
elem ← liste2[i]
```

- de connaître le nombre d'éléments d'une liste

```
nb ← NombreElements ( liste1 )
```

- de concaténer deux listes

```
liste3 ← liste2 + [ 'a', 'b', 'c' ]
```


Principaux constructeurs de types

- d'extraire une "sous-liste" :

```
liste1 ← [ 3, 1, 10, 3, 14, 7 ]
liste3 ← liste1 [0:4]  # sous-liste des éléments de
          # rang 0 à 4 non compris : [ 3, 1, 10, 3 ]
```

La concaténation et l'extraction permettent de faire les mises à jour suivantes :

```
liste1 ← [ elem ] + liste1
                  # ajout de elem en début de liste1
liste1 ← liste1 + [ elem ]
                  # ajout de elem en fin de liste1
nb ← NombreElements(liste1)
liste1 ← liste1[0,i] + [ elem ] + liste1[i,nb]
                  # insertion de elem en rang i
liste1 ← liste1[0,i] + liste1[i+1,nb]
               # suppression de l'élément de rang i
```


Troisième partie

Algobox et les listes

Les listes Algobox (1)

Algobox permet de manipuler des listes de nombres. Les variables correspondantes sont déclarées comme étant de type LISTE :

```
maListe EST_DU_TYPE LISTE
```

Les éléments de la liste sont numérotés (rang de l'élément) à partir de 0.

On peut affecter une valeur à un élément de la liste ou à une suite d'éléments consécutifs :

```
maListe[0] PREND_LA_VALEUR  8
maListe[1] PREND_LA_VALEUR  1:2:3
```


Les listes Algobox (2)

Pour afficher le contenu d'une liste, il faut parcourir un à un ses éléments :

```
POUR I ALLANT_DE 0 A 4
  DEBUT_POUR
  AFFICHER maListe[I]
  FIN_POUR
```

L'accès à un élément de la liste non défini provoque une erreur (d'exécution ou d'affichage).

Les fonctions prédéfinies

Algobox offre plusieurs fonctions de manipulation de listes (voir l'aide Algobox) :

```
ALGOBOX_SOMME
ALGOBOX_MOYENNE
ALGOBOX_VARIANCE
ALGOBOX_ECART_TYPE
ALGOBOX_MEDIANE
ALGOBOX_QUARTILE1
ALGOBOX_QUARTILE3
ALGOBOX_QUARTILE1_BIS
ALGOBOX_QUARTILE3_BIS
ALGOBOX_MINIMUM
ALGOBOX_MAXIMUM
ALGOBOX_POS_MINIMUM
ALGOBOX_POS_MAXIMUM
```