Les fonctions

Chapitre 6

Fonctions en mathématiques

Une fonction `f` définie sur `A` (domaine de définition, espace de départ) à valeurs dans `B` (codomaine, espace d'arrivée, espace image) est une correspondance qui, à tout élément `x` de `A` fait correspondre un élément et un seul, noté `f(x)`, de `B`.

`f(x)` est appelé résultat de l'application de `f` à l'élément `x`.

Spécification

Une spécification est une description dans un langage formel (rigoureux) :

- d'une correspondance entre des données et des résultats ainsi que
- des propriétés de cette correspondance.

Un algorithme est un cas particulier de spécification.

C'est une spécification exécutable.

Les fonctions en informatique

$f(x)=ax^2+bx+c$

`x` est le paramètre formel de la fonction `f`.

`a, b, c` sont les variables globales (libres) de la fonction `f`.

L'expression (ici ax^2+bx+c définissant la fonction est appelée corps de la fonction.

Dans l'application de `f` à une valeur `v`, notée `f(v)`, `v` est appelé paramètre effectif ou argument.

Fonctions totales

En informatique, une fonction doit être totale (partout définie) => tout élément du domaine peut être a priori choisi comme argument.

Les cas particuliers doivent impérativement être traités de manière explicite.

Type d'une fonction

`T1*T2*...*Tn->T`

avec

`T1,T2,...,Tn` : les types des paramètres formels

`T` : le type de la valeur retournée

Déclaration d'une fonction : Syntaxe

```
<déclaration_fonction>::=<en-tete_fonction>;
<en-tete_fonction>::=
   function <ident_fonction> <liste_paramètres>
                                    return <ident_type>
<liste_paramètres>::= vide| (<déclar_paramètres>)
<déclar_paramètres>::=
   <ident_param>:<type_param>
                      [:=<valeur>]{,<déclar_paramètres>}
<valeur>::=une expression du type associé
```

Exemples de déclaration

```
function double(X:Integer) return Integer;
```

`double`: identificateur de la fonction
`X` : paramètre formel
`Integer` : type du paramètre formel
`Integer` : type de la valeur retournée (résultat)
`Integer → Integer` : type de la fonction

```
function max(A,B:Integer) return Integer;
```

`max`: identificateur de la fonction
`A,B` : paramètres formels
`Integer` : type des paramètres formels
`Integer` : type de la valeur retournée (résultat)
`Integer*Integer → Integer` : type de la fonction

Déclaration d'une fonction : Sémantique

La fonction dénotée par son identificateur appartient à l'environnement du programme. A son identificateur est associé son type.

La valeur de la fonction, c'est à dire le code binaire exécutable qui lui est associé, n'est pas encore défini.

Le programmeur déclare ainsi les contraintes d'utilisation de l'objet fonction. Il spécifie ainsi un nouvel outil.

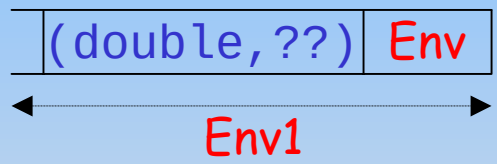
Le compilateur utilisera ces informations pour vérifier que l'utilisation qui est faite de cet objet fonction répond bien a l'intention du programmeur

Sémantique de déclaration (1/2)

Déclaration dans l'environnement Env de la fonction :

```
function double(X:Integer) return Integer;
```

- création de la liaison `(double,??)`
- extension de Env avec cette liaison



- Détermination du type de la fonction :

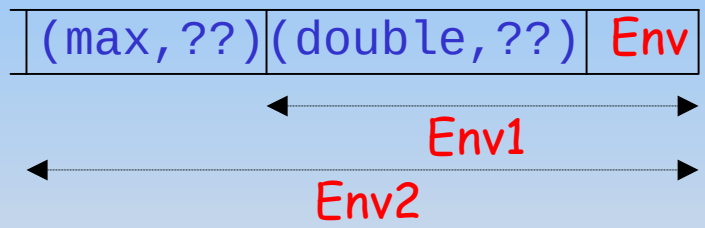
`Integer→Integer`

Sémantique de déclaration (2/2)

Déclaration dans l'environnement Env1 de la fonction:

```
function max(A,B:Integer) return Integer;
```

- création de la liaison `(max,??)`
- extension de Env avec cette liaison



- Détermination du type de la fonction :

`Integer*Integer→Integer`

Exemple (1/2)

```
function perimetre(A:Float) return Float;
PI:constant Float:=3.14159;
```

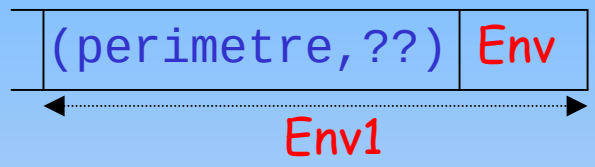
2 objets sont introduits dans l'environnement Env du programme :

- La fonction d'identificateur `perimetre` avec son type :
  `Float-->Float`
- La constante d'identificateur `PI` avec son type : `Float`
- et sa valeur : `3.1459`

Exemple (2/2)

Création de la liaison : `(perimetre,??)`

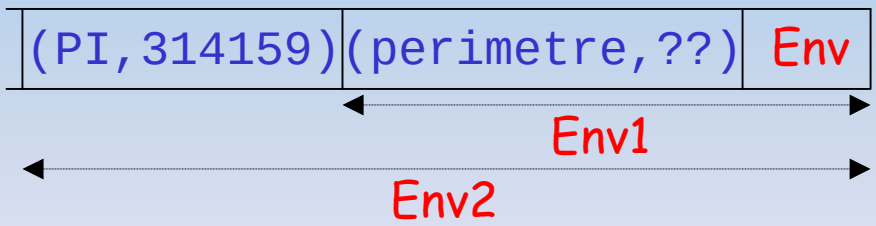
Extension de Env



Détermination du type : `Float-->Float`

Création des liaisons : `(PI,??),` puis de `(PI,3.14159)`

Extension de Env1



Déclaration du corps d'une fonction : **Syntaxe**

```
<declaration_corps_fonction>::=
<en-tete_fonction> is
   <déclaration_objets_locaux>
   begin
       <corps_de_fonction>
   end <ident_fonction>;
```

Exemples

```
function max(A,B:Integer) return Integer is
begin
  if (A>B)
  then
     return A;
  else
     return B;
  end if;
end max;

function double(X:Integer) return Integer is
  begin return 2*X;
end double;
```

Déclaration de fonction locale à une autre fonction

```
function par_2(X:Integer) return Integer is
  function succ(N:Integer) return Integer is
  begin
    return (N+1);
  end succ;
begin
  return succ(succ(X));
end par_2;
```

Surcharge des opérateurs prédéfinis (1/2)

```
declare
type Complexe  is
       record X,Y :Float ;end record ;
function "+"(A,B :Complexe) return Complexe is
begin
       return (A.X+B.X,A.Y+B.Y);
end "+";
R:Complexe;
A:Complexe:=(1.3,6.5);
B:Complexe:=(2.4,3.9) ;
begin
   R:=A+B ;
end;
```

Surcharge des opérateurs prédéfinis (2/2)

```
function "+"(A :Vecteur) return Float is
R:Float:=0.0;
begin
   for i in A'range loop
       R:=R+A(i);
   end loop;
   return R;
end "+";
```

Nous pouvons écrire :

```
V:Vecteur:=(2.1,-5.78,9.54,-3.2);
S:Float:=+V;
```

Déclaration du corps d'une fonction : sémantique

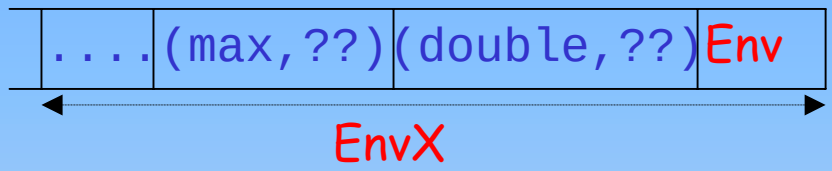
La valeur de la fonction est déterminée et associée à son identificateur.

Cette valeur est le code binaire de la fonction associé au contexte courant.

Elle est appelée fermeture car elle réunit en une seule structure le code binaire et le contexte de la déclaration du corps de la fonction.

Déclaration du corps de la fonction **max**

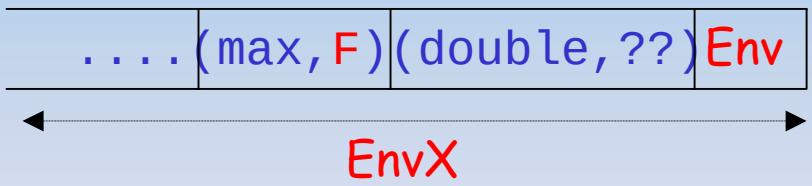
Dans l'environnement EnvX



Détermination de la fermeture F de max :

F = <<A,B→corps de max, EnvX>>

Modification de la liaison dans EnvX

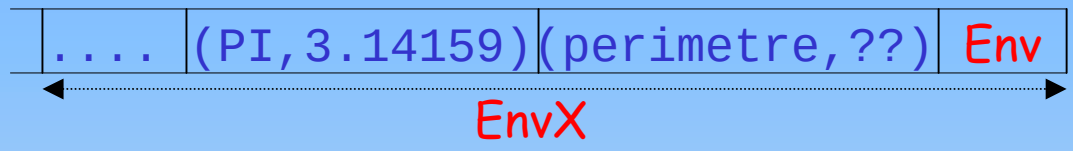


Déclaration du corps de la fonction **perimetre** (1/2)

```
function perimetre (A:Float) return Float is
begin
   return 2.0*PI*R;
end perimetre;
```

Déclaration du corps de la fonction **perimetre** (2/2)

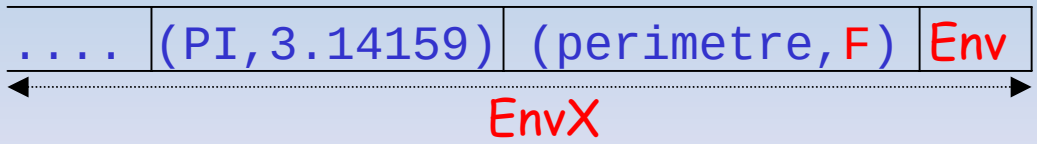
Dans l'environnement EnvX



Détermination de la fermeture `F` de `perimetre` :

`F = <<A→corps de perimetre, EnvX>>`

Modification de la liaison dans EnvX



La constante `PI` appartenant à EnvX est donc dans la fermeture de `perimetre`

Fonction produit de matrice (1/2)

```
type Matrice is
    array(Positive range <>,Positive range <>) of Float;
```

Fonction produit de matrice

```
function produit_matrice(A,B: Matrice) return Matrice is
  C:Matrice(1..A'last(1),1..B'last(2));
begin
  for i in A'range(1) loop
    for j in B'range(2) loop
      declare
        s:Float:=0.0;
      begin
        for k in A'range(2) loop
          s:=s+A(i,k)*B(k,j);
        end loop;
        C(i,j):=s;
      end;
    end loop;
  end loop;
  return C;
end produit_matrice;
```

Application d'une fonction : Syntaxe

```
<application_fonction>::=
            <ident_fonction><liste_paramètres_effectifs>
<liste_paramètres_effectifs>::= vide|(<param_effectifs>)
<param_effectifs>::=
            <val_param>[:=<valeur>]{,<param_effectifs>}
<val_param>::= ident|une expression du type associé
```

Exemples

```
max(78,45)
double(43)
perimetre(37.97)
```

Association entre paramètres effectifs et formels

Soit une fonction dont la déclaration est schématisée ainsi :

```
function g(P1:T1;P2:T2;...;Pn:Tn) return T;
```

Le type de la fonction `g` est : `T1*T2*...*Tn → T`

- ✓ d'après l'ordre d'écriture

  `g(A1,A2,...,An)`
- ✓ en nommant le paramètre formel

  `g(Pk=>Ak,...,Pi=>Ai)`
- ✓ en panachant (les paramètres par position apparaissent en premier et dans l'ordre)

  `g(A1,A2,...,Ak-1,Pk=>Ak,...,Pn=>An)`

Les paramètres effectifs `Ai` doivent être d'un type compatible avec le type du paramètre formel correspondant `Ti`.

Paramètres par défaut

Si, lors de la plupart des appels, un paramètre prend la même valeur, alors il est possible de spécifier cette valeur dans la déclaration de la fonction.

Dans ce cas, il n'est pas obligatoire de mentionner le paramètre effectif au moment de l'appel (sauf s'il est suivi d'autres paramètres non définis par défaut).

```
function eliminer(C:Character:=' ';T:String)
                                    return String;
-- cette fonction a pour but de retourner le
-- texte T passé en paramètre après suppression
-- de tous les caractères C du texte.
```

Déclaration de la fonction : **eliminer**

```
function eliminer(C:Character:=' ';T:String) return String is
X:String:=T;deb:Positive:=T'first;fin:Positive:=T'last;
begin
-- on suppose que T n'est pas vide
   loop
   exit when deb>fin;
      if X(deb)=C
      then
         X(X'first..fin-1):=X(X'first..deb-1)&X(deb+1..fin);
        fin:=fin-1;
      else
        deb:=deb+1;
      end if;
    end loop;
    return X(X'first..fin);
end eliminer;
```

Supprimer le caractère espace d'un texte donné

```
with Ada.Text_io;use Ada.Text_io;
procedure eliminer_car is
 texte:String:= "hier c'est le passe;demain c'est le
futur;aujourd'hui, c'est un cadeau.C'est pour ça
qu'on l'appelle present" ;

begin
  put_line(eliminer(T=>texte));
-- Si un paramètre est omis (cas du paramètre par
-- défaut,on doit nommer les autres paramètres
  put_line(eliminer('e',texte));
end eliminer_car;
```

Résultat

```
hierc'estlepasse;demainc'estlefutur;aujourd'hu
ic'estuncadeau.C'estpourçaqu'onl'appellepresent

hir c'st l pass;dmain c'st lfutur;aujourd'hui
c'st un cadau.C'st pour ça qu'on l'appll prsnt
```

Application d'une fonction : sémantique

```
function h(P:T) return T';
function h(P:T) return T' is
begin
-- corps de h
end h;
```

Evaluons `h(exp)`,

- ✓ L'évaluation de h permet de récupérer le code binaire qui lui est associé ainsi que l'environnement dans lequel son corps avait été déclaré.
- ✓ L'évaluation de exp permet d'attribuer une valeur à l'argument.
- ✓ A partir de ces informations, le contexte d'exécution de l'appel est construit. La valeur de exp est associée à l'identificateur du paramètre formel et le code binaire exécuté.
- ✓ A la fin de l'exécution, ce contexte est détruit et remplacé par le contexte d'avant appel.

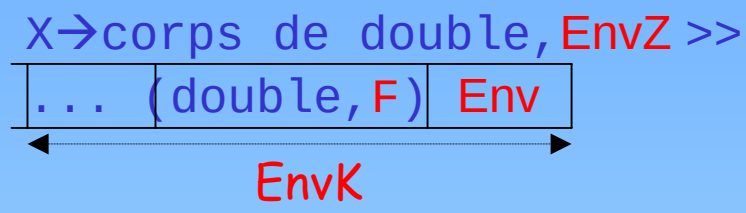
Application de la fonction `double` (1/2)

```
function double(X:Integer) return Integer;
function double(X:Integer) return Integer is
begin
   return 2*X;
end double;
```

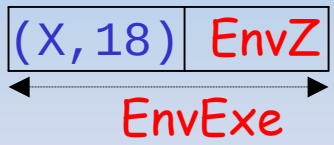
Il s'agit d'évaluer `double(18)`

- ✓ L'évaluation de `double` ramène le code binaire de la fonction
- ✓ Le paramètre formel `X` prend la valeur `18`. Cette association étend l'environnement de la déclaration du corps de la fonction.
- ✓ Le code s'exécute alors, la valeur retournée est `36` et le contexte d'exécution détruit.
- ✓ On revient donc au contexte d'exécution initial.

Application de la fonction **double** (2/2)



- ✓ Evaluation de `double (18)` dans `(EnvK,MemK)`
  - l'évaluation de l'identificateur double retourne la fermeture
    `F = << X→corps de double, EnvZ >>`
  - 18 s'évalue en lui-même
- ✓ Construction de l'environnement d'exécution EnvExe de l'application



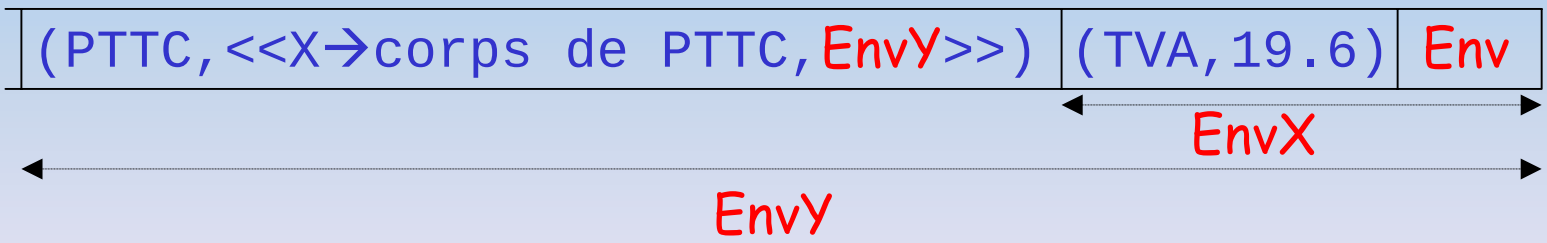
- ✓ Exécution du corps de double dans l'état (EnvExe,MemK) : la valeur retournée est 36
- ✓ Destruction de EnvExe et restitution de l'environnement initial EnvK

Exemple

```
with Ada.Text_io;
procedure calculPrix is
  TVA : constant Float:=19.6;
  function PTTC(X:Float) return Float is
  begin
    return X+0.01*TVA*X;
  end PTTC;
begin
  Ada.Text_io.put(PTTC(10.0));
end calculPrix;
```

Evaluation des déclarations

- ✓ L'évaluation de la déclaration de la constante `TVA` associe l'identificateur `TVA` au type `Float` et à la valeur `19.6`
- ✓ L'évaluation de la déclaration de la fonction `PTTC` associe l'identificateur `PTTC` au type `Float→Float` et à une valeur indéfinie.
- ✓ L'évaluation du corps de la fonction crée sa fermeture qui remplace la valeur indéfinie



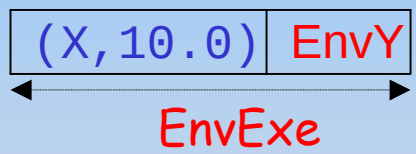
Application `put(PTTC(10.0))`(1/2)

Evaluation de `PTTC` et `10.0`

`PTTC => <<X→corps de PTTC,EnvY>>`

`10.0 => 10.0`

Construction de l'environnement d'exécution de `PTTC(10.0)`



Exécution du corps de `PTTC` dans l'état `(EnvExe,Mem)`

`=> X+0.01*TVA*X=10.0+0.01*19.6*10.0=11.96`

Destruction de EnvExe, restitution de EnvY

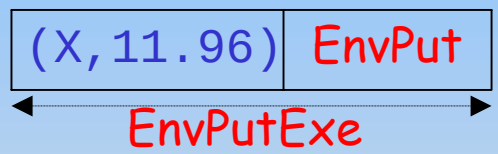
Application `put(PTTC(10.0))`(2/2)

Evaluation de `put(PTTC(10.0))` (dans Env)

`put => <<X→corps de put,`envPut`>>`

`PTTC(10.0)) => 11.96`

Construction de l'environnement d'exécution de `put(PTTC(10.0))`



Exécution du corps de `put` dans l'état `(`EnvPutExe`,`Mem`)`

`=> affichage de la valeur 11.96 à l'écran`

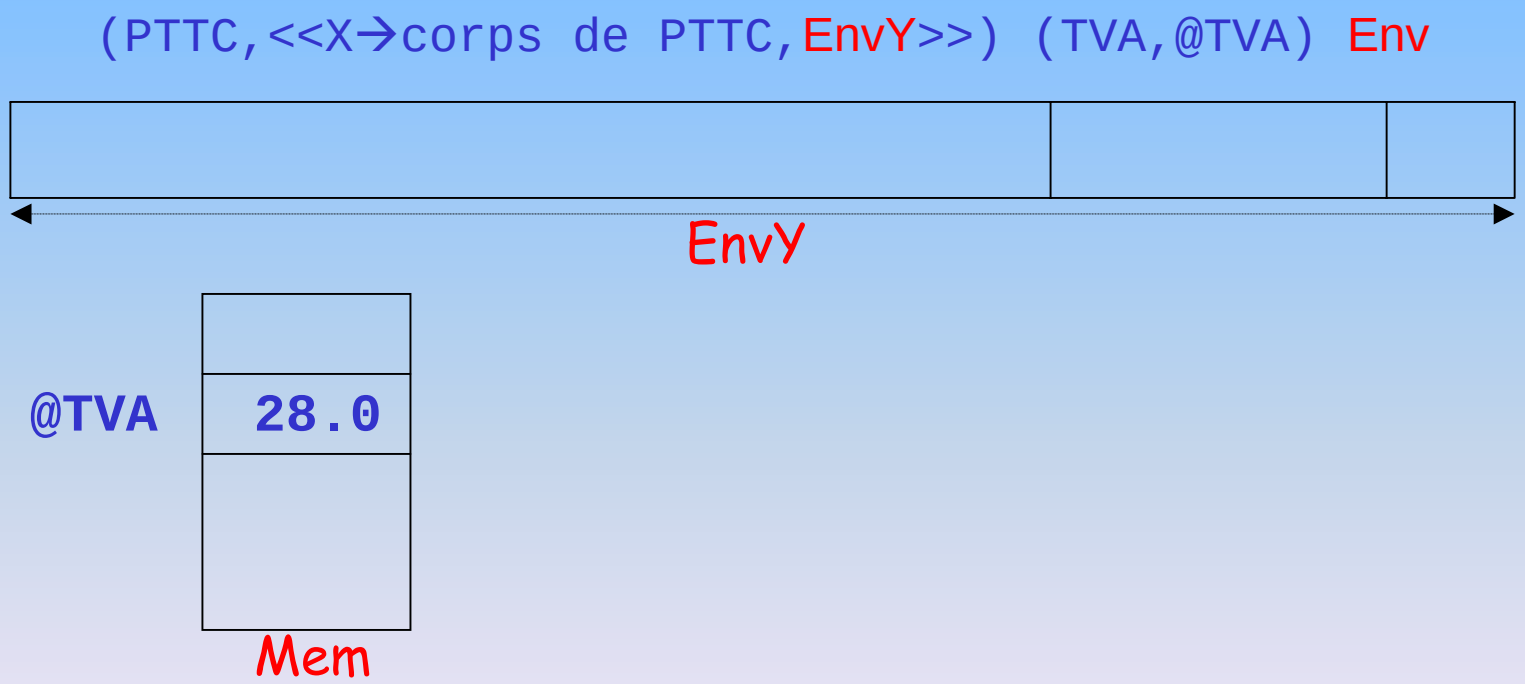
Destruction de EnvPutExe, restitution de EnvY

Exemple : variante (1/6)

```
with Ada.Text_io;
procedure calculPrix is
  TVA : Float:=28.0;
  function PTTC(X:Float) return Float is
  begin
    return X+0.01*TVA*X;
  end PTTC;
begin
  Ada.Text_io.put(PTTC(10.0));
end calculPrix;
```

Exemple : variante (2/6)

Après évaluation des déclarations, nous obtenons l'état suivant :



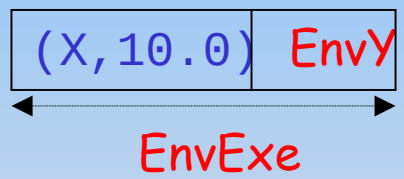
Exemple : variante (3/6)

Evaluation de `PTTC` et `10.0`

`PTTC => <<X→corps de PTTC,EnvY>>`

`10.0 => 10.0`

Construction de l'environnement d'exécution de `PTTC(10.0)`



Exécution du corps de `PTTC` dans l'état `(EnvExe,Mem)`

`=> X+0.01*TVA*X=10.0+0.01*28.0*10.0=12.8`

Destruction de EnvExe, restitution de EnvY

Exemple : variante (4/6)

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Float_Text_io;use Ada.Float_Text_io;
procedure prixTTC is
   TVA:Float:=19.6;
   function PTTC(X:Float) return Float is
   begin
       return X+0.01*TVA*X;
   end PTTC;
begin
   put("PTTC(10.0)=");put(PTTC(10.0),1,2,0);new_line;
   TVA:=28.0;
   put("PTTC(10.0)=");put(PTTC(10.0)1,2,0);
end prixTTC;
```

résultat

```
PTTC(10.0)=11.96
PTTC(10.0)=12.80
```

Exemple : variante (5/6)

On remarque que cette façon de coder ne préserve pas la notion de fonction car, appliquée à un même paramètre effectif, PTTC retourne 2 résultats différents.

Nous avons transmis de l'information à la fonction par effet de bord et non par paramètre.

Comment éviter cela ?

Par l'introduction d'un paramètre supplémentaire qui permettra de transmettre la nouvelle valeur de la TVA de manière explicite

Exemple : variante (6/6)

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Float_Text_io;use Ada.Float_Text_io;
procedure prixTTC is
   TVA:Float;
   function PTTC(X:Float;Y:Float) return Float is
   begin
        return X+0.01*Y*X;
   end PTTC;
begin
   TVA:=19.6;
   put("PTTC(10.0,19.6)=");put(PTTC(10.0,TVA),1,2,0);new_line;
   TVA:=28.0;
   put("PTTC(10.0,28.0)=");put(PTTC(10.0,TVA)1,2,0);
end prixTTC;
```

résultat

```
PTTC(10.0,19.6)=11.96
PTTC(10.0,28.0)=12.80
```

Exemple

```
declare
X:Matrice(1..5,1..3):= -- agrégat d'initialisation
Y:Matrice(1..3,1..8):= -- agrégat d'initialisation
C:Matrice(1..5,1..8);
begin
  C:=produit_matrice(X,Y);
  for i in X'range(1) loop
    for j in Y'range(2) loop
      put(C(i,j));
-- Ada.Float_Text_io appartient à l'environnement
    end loop;
  new_line;
  end loop;
end;
```

Le paramètre formel A (resp B) (de type Matrice non contraint) prendra la valeur du paramètre effectif X (resp Y). Le code binaire de la fonction sera exécuté pour les valeurs X et Y des paramètres formels A et B.

Fonctions récursives

Une fonction récursive est une fonction qui fait appel à elle-même pour son exécution.

La fermeture de toute fonction Ada contient une liaison pour cette même fonction donc toute fonction est naturellement récursive.

Autrement dit, toute fonction Ada se connaît elle-même.
Elle est donc en mesure de s'appeler elle-même.

Exécution d'une fonction récursive (1/2)

Soit la fonction `SommeCarres(n)` qui
calcule la somme des carrés des `n` premiers entiers positifs ou nuls.
Elle est définie récursivement par :

```
SommeCarres(n)= si n=0 alors 0
                sinon SommeCarres(n-1)+n*n
```

Traduction Ada

```
function SommeCarres(n:Natural) return Natural is
begin
 if n=0 then return 0;
 else return SommeCarres(n-1)+n*n;
end SommeCarres;
```

Exécution de SommeCarres(3)

étapes du calcul	environnement d'exécution	pile
SommeCarres(3)	n=3	[]
SommeCarres(n-1)+n*n	n=3	[]
SommeCarres(2)	n=2	[n=3]
SommeCarres(n-1)+n*n	n=2	[n=3]
SommeCarres(1)	n=1	[n=2,n=3]
SommeCarres(n-1)+n*n	n=1	[n=2,n=3]
SommeCarres(0)	n=0	[n=1,n=2,n=3]
0	n=0	[n=1,n=2,n=3]
0+n*n-->1	1	[n=2,n=3]
1+n*n-->5	2	[n=3]
5+n*n-->14	3	[]

Conception d'une fonction récursive

Eliminer toutes les occurrences d'un caractère donné dans une chaîne.

Raisonnement par récurrence sur la longueur de la donnée. Soit `T` le texte et `C` le caractère à éliminer :

On connaît le résultat pour une valeur de la longueur :

```
si longueur(T)=0, le résultat est T (ou "")
```

On suppose que l'on sait éliminer toutes les occurrences de `C` pour un texte `T` de longueur `i-1`.

On montre que l'on peut éliminer tous les caractères `C` de `T` de longueur `i`;

Algorithme

Soit une chaîne représentée par le tableau `T(1..i)`

si `longueur(T)=0` **alors** le résultat est `T`

sinon

```
    si le premier caractère = C
    alors
        le résultat provient de l'élimination de toutes
          les occurrences de C du texte T(2..i)(longueur
          i-1)
    sinon
        le résultat est construit par la concaténation
          du 1er caractère T(1) avec la chaîne T(2..i)
          dans laquelle toute occurrence de C a été
          supprimée
```

Implantation Ada de la fonction récursive

```
function eliminer(C:Character:=' ';T:String) return
   String is
      X:String:=T;
      deb:Positive:=T'first;
      i:Positive:=T'last;
    begin
      if i<deb
      then return "";
      elsif  X(deb)=C
         then return eliminer(C,X(deb+1..i));
         else return X(deb)&eliminer(C,X(deb+1..i));
         end if;
    end eliminer;
```

Implantation Ada

```
with Ada.Text_io; use Ada.Text_io;
procedure eliminer_rec is
  function eliminer(C:Character:=' ';T:String)
                                    return String is
  ...
  end eliminer;
  texte:String:="     oui et non   ";

begin
  put_line(eliminer(' ',texte));
  put_line(eliminer('n',texte));
end eliminer_rec;
```

Fonctions mutuellement récursives

La séparation entre la déclaration d'une fonction et la déclaration de son corps, permet les fonctions mutuellement récursives.

Exemple : Soient deux fonctions définies par leurs suites récurrentes,

$U_n = 2 * V_{n-1} + U_{n-1}$

$U_0 = 1$

$V_n = 2 * U_{n-1} + V_{n-1}$

$V_0 = -1$

```
function V(N:Integer) return Integer;
function U(N:Integer) return Integer;
function U(N:Integer) return Integer is
begin
   if N=0 then
      return 1;
   else
      return 2*V(N-1)+U(N-1);
   end if;
end U;
function V(N:Integer) return Integer is
begin
   if N=0 then
       return (-1);
    else
       return 2*U(N-1)+V(N-1);
    end if;
end V;
```

Articles et fonctions (1/4)

```
type Point is
   record
        x,y:Integer;
   end record;
function creer_point(abs,ord:Integer) return Point;
type Rectangle
   record
        bg,hd:Point;
   end record;
function a_l_interieur(p:Point;r:Rectangle)
                                          return Boolean;
ecran:Rectangle:=
            (creer_point(0,0),creer_point(XMAX,YMAX));
```

Articles et fonctions (2/4)

```
function creer_point(abs,ord:Integer) return Point is
  p:Point;
begin
   p.x:=abs;p.y:=ord;return p;
end fabrique_point;

function a_l_interieur(p:Point;r:Rectangle)
                                    return Boolean is
begin
   return (p.x>=r.bg.x and p.x<r.hd.x and p.y>=r.bg.y
   and p.y<r.hd.y);
end a_l_interieur;
```

Articles et fonctions (3/4)

```
function bissextile(a:Natural) return Boolean is
begin
   return (a mod 400=0 or (a mod 100/=0 and a mod 4=0));
end bissextile;

subtype NbJour is Natural range 1..7;
subtype NbJoursMois is Natural range 1..31;
type Date is
   record
        jour: NbJour;
        mois: NbJoursMois;
        an:Natural;
   end record;
```

Articles et fonctions (4/4)

```
function quantieme(d:Date) return Positive is
   type TabMois is
        array(Boolean,0..12) of NbJoursMois ;
   t:TabMois:=
   ((false,(0,31,28,31,30,31,30,31,31,30,31,30,31)),
    (true,(0,31,29,31,30,31,30,31,31,30,31,30,31)));
   res:Positive:=d.jour;
begin
   for i in 1..d.mois-1 loop
        res:=res+t(bissextile(d.an),i);
   return res;
end quantieme;
```