Fonctions et Procédures de test [ss]
Support de Cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 15 mai 2018

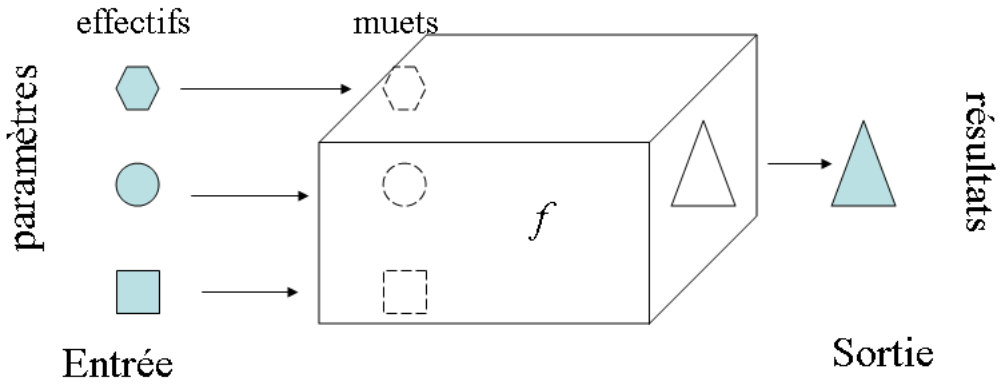
Table des matières

C++ - Fonctions et Procédures de test (Cours)

Mots-Clés Algorithmes paramétrés, Fonction, Procédure de test ■
Requis Structures de base, Structures conditionnelles ■
Difficulté • ○ ○ (1 h) ■

Introduction

Ce module (en deux parties) traite des **fonctions** et **procédures** qui sont des **algorithmes paramétrés** autonomes et spécialisés qui présentent l'intérêt de pouvoir réutiliser du code. Celui-ci introduit : les définitions, les concepts, la **fonction** et les **procédures de test**.



1 Motivation

Programmation modulaire, module
Technique utilisée lors de la conception d'algorithmes complexes, la **programmation modulaire** consiste à diviser un algorithme en sections (groupe d'instructions) et à associer un nom à une section qui peut alors être activé par l'appel de cet identifiant. Chaque section est appelée un **sous-algorithme** ou **module**.

Exemple
On peut réaliser le tri par ordre croissant de trois nombres comme suit :

```
(1) ordonner les deux premiers nombres a et b
(2) ordonner les deux derniers b et c
(3) ordonner à nouveau les deux premiers a et b
```

En effet, à l'issue des deux premières opérations, `c` contiendra le plus grand des trois nombres et à l'issue des trois opérations, `a` mémorisera le plus petit des trois. Donc `b` sera le médian des trois.

Dans cet exemple, l'approche modulaire consiste à définir deux modules conservant chacun leur spécificité (l'un réalisant l'ordonnancement de deux nombres et l'autre celui de trois nombres) et de leur permettre de communiquer entre eux pour s'échanger des données **et** des résultats.

Exemple
Un deuxième type de module est celui **renvoyant une valeur**. Considérons le problème : « calculer le maximum de trois nombres ». Pour le résoudre, voici une approche :

```
(1) tmp <- calculer le maximum des deux premiers nombres
(2) tmp <- calculer le maximum de tmp et du troisième nombre
```

Ici l'idéal est de pouvoir définir deux modules (l'un renvoyant le maximum de deux nombres et l'autre le maximum de trois nombres) et de leur permettre de communiquer entre eux pour s'échanger des données.

Conclusion
De façon imagée, un module est un algorithme dont l'évaluation (on dit l'**appel** ou **invocation**) correspond à l'exécution de la portion de programme qui lui est associée et fournit l'éventuelle valeur de retour.

De plus, pour pouvoir faire communiquer les modules entre eux, il faut les équiper d'une « interface » de transmission des entités qui contient une déclaration de variables qu'on appellera les **paramètres** du module.

2 Définitions et concepts

2.1 Module, Procédure, Fonction

Module

Appelé aussi *sous-programme* en programmation, c'est un bloc d'instructions ayant un **début** et une **fin**, identifié par un **nom** (l'identifiant) et associé à la définition d'une interface explicite par le biais d'une spécification de **paramètres**.

```
Module nom ( paramètres )
Début
| ...
Fin
```

Prototype, Arité

Le **prototype**, appelé aussi **profil**, **signature** ou **en-tête** d'un module, est le couple « identifiant, paramètres ». L'**arité** est son nombre de paramètres.

Procédure

Module paramétré dont l'appel est assimilable à une instruction : c'est un macro-traitement.

Fonction

Module paramétré qui renvoie obligatoirement **une (et une seule) valeur résultat** au module appelant : c'est une macro-expression.

Algorithme (nouvelle définition)

Ensemble de procédures et de fonctions.

Procédure v.s. Fonction

Certains langages de programmation (comme les langages C/C++) confondent fonctions et procédures en donnant aux fonctions des propriétés supplémentaires qui leur permettent de se comporter comme des procédures. En algorithmique, nous tenons à distinguer les fonctions (qui renvoient une valeur) des procédures (qui exécutent une action). Libre ensuite au programmeur de s'adapter au langage qu'il utilise.

Catégories de modules

On distingue :

- Les **modules prédéfinis**.
- Les **modules définis par l'utilisateur**.

Ils ont un comportement identique. La différence est que les premiers sont déjà écrits dans des bibliothèques, tandis que les seconds sont à définir, c.-à-d. à écrire et à les faire évaluer par la machine (compiler) avant utilisation.

2.2 Paramètre, Argument

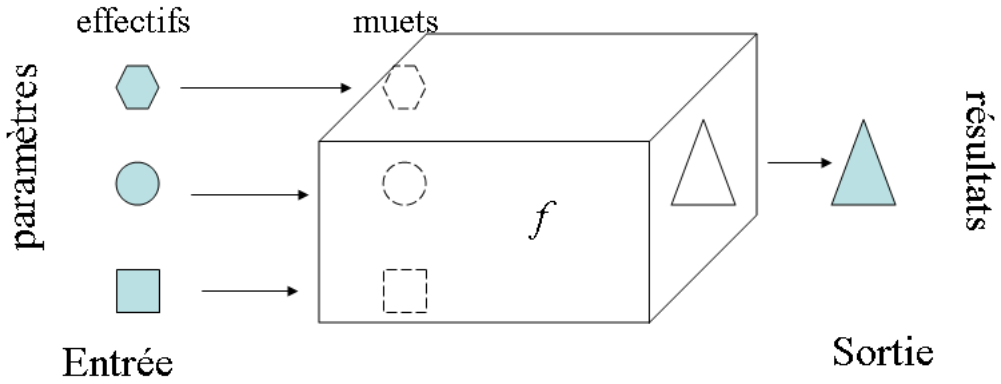
Paramètres

Dits aussi **paramètres formels** ou **paramètres muets**, ce sont les identifiants indiqués dans l'en-tête d'un module. Ils permettent de donner des noms provisoires aux entités transmises. Ce sont ces identifiants qui **doivent être utilisés** pour décrire le traitement du module.

Arguments

Dits aussi **paramètres effectifs**, ce sont les valeurs ou variables passées au module lors de son appel : c'est réellement avec ces éléments que travaillera le module.

Résumé schématique



Intérêts des paramètres

Deux intérêts principaux :

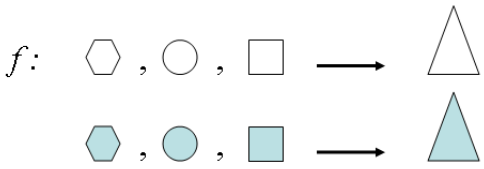
- Le premier concerne la **généralité des traitements**. En effet, ils fournissent un mécanisme de substitution qui permet à un module d'être exécuté plusieurs fois sur des données ou des variables différentes.
- Le second intérêt est **documentaire**. Les paramètres autorisent la mise en évidence des informations utilisées ou modifiées par un module. Cela permet de comprendre plus rapidement les effets du traitement sur les données de l'algorithme.

2.3 Passage de paramètre

Passage de paramètre par position

Les arguments sont associés **par position** aux paramètres :

- Le premier argument est associé au premier paramètre.
- Le deuxième argument effectif au deuxième paramètre.
- etc.



f : ensemble de départ → ensemble d'arrivée
f : paramètres formels → description du résultat

Exemple : Appel/description – Paramètres effectifs/formels

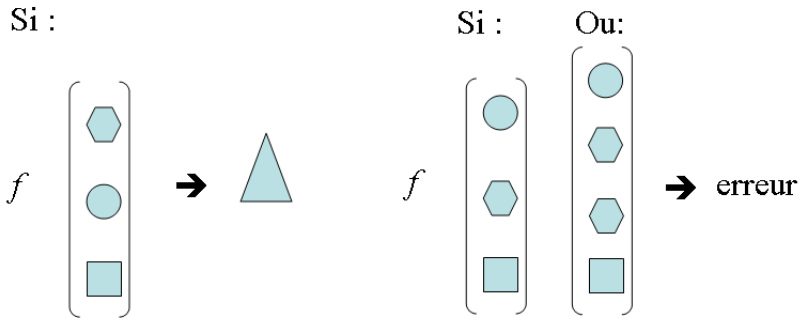
```
description:  ssprg(x : TypeParam; y : TypeParam; ...)    Formel
appel:        ssprg(a, b, ...)                            Effectif
```

Le paramètre `x` représentera l'argument (variable effective) `a` et le paramètre `y` la variable effective `b`.

Passage de paramètre par position

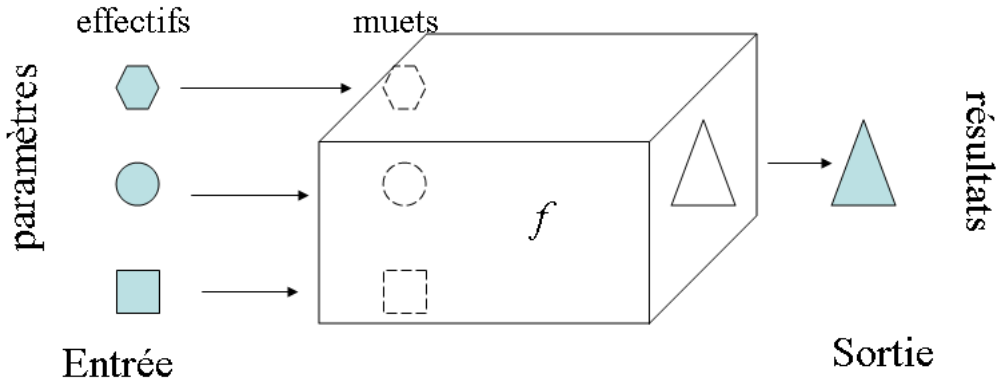
Les règles à respecter :

- Il faut le même nombre d'arguments que de paramètres.
- Types et arités des arguments aux paramètres doivent correspondre.



2.4 Algorithme d'un module

Algorithme d'un module (procédure ou fonction)
Description de ce qu'il y a à l'intérieur de la « boîte » et qui permet la production du résultat à partir des paramètres. **Élaborer un algorithme** c'est construire la « boîte ». Ceci peut se faire à partir d'autres modules déjà existants.



Non unicité de l'algorithme
Il peut exister plusieurs algorithmes pour un même module.
Exemple : Réel -> Réel, x -> double de x

- Algorithme 1 : x + x
- Algorithme 2 : 2 * x

2.5 Appel d'un module

Appel d'un module

Il suffit de spécifier son nom (avec ses éventuels arguments).

Mécanisme d'évaluation d'un module

Celui-ci s'effectue comme suit :

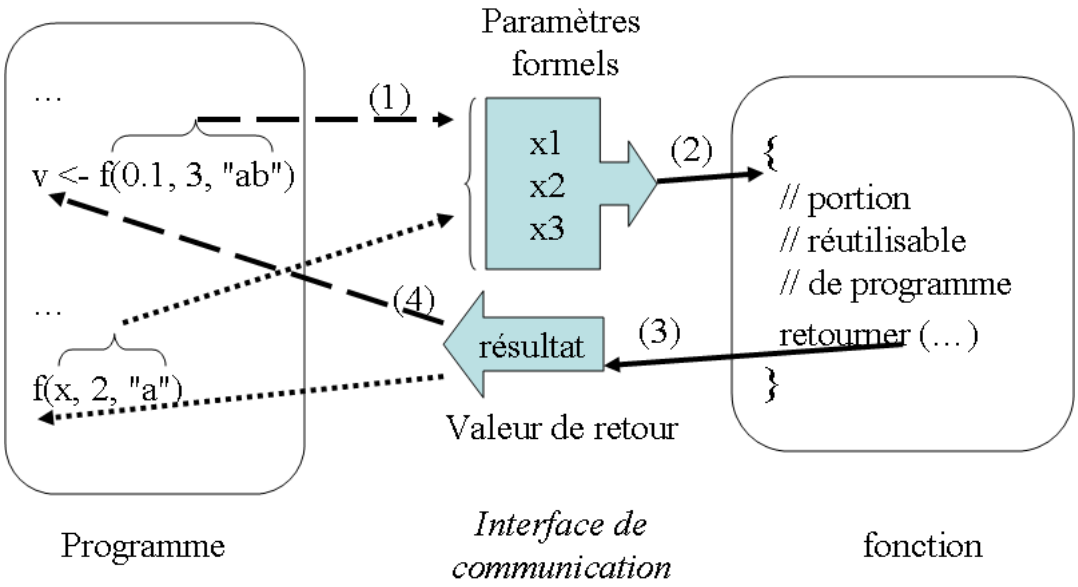
(0) Évaluation de l'identifiant `f` et des arguments

(1) Copie ou liaison des paramètres aux arguments

(2) Réalisation du calcul dans cet environnement

(3) Retour de la valeur (en cas de fonction)

(4) Continuation du programme après l'appel `f`



Remarque

Lorsque les instructions du module sont terminées, l'algorithme ou le programme reprend à l'instruction qui suit l'instruction d'appel dans le bloc appelant.

2.6 Résumé et portée des modules

Module (Nouvelle définition)

Élément logiciel caractérisé par :

- Un **identifiant** : référence à l'élément « procédure » ou « fonction ».
- Un **corps** : bloc de code réutilisable. Induit une portée — locale — pour les éléments (variables, constantes) définis au sein du module.
- Des **paramètres formels** : copies ou liaisons à des variables définies à l'extérieur du module dont les valeurs sont potentiellement utilisées dans le corps du module.
- Un **type** et une **valeur de retour** : le type précise la nature du résultat retourné par la fonction et la valeur sera indiquée dans son corps.

Mise en oeuvre d'un module

Elle se fait en deux étapes :

- Le **prototypage** ou **spécification** (≡ déclaration de l'en-tête).
- La **définition** ou **implémentation** (≡ écriture des instructions du corps — l'éditeur de liens vérifie que le module est défini).

Portée des modules

Les modules (procédures et fonctions) ont une portée (et une visibilité).

- Tout module **doit** être prototypée avant d'être utilisé.
- Les prototypes seront placés **avant** le corps de la fonction `main`.

3 Fonction

3.1 Profil de fonction

Profil de fonction
Constitué par le nom de la fonction, la liste des types des paramètres d'entrée et le type du résultat.

C/C++ **Profil de fonction**

```
TypeRes nomFcn(T1 param1, ..., Tn paramN)
```

Explication
Définit le **profil** de la fonction d'identifiant `nomFcn` ayant pour paramètres formels les `paramI` de type correspondants `Ti`. Le **type** de la valeur renvoyée est `TypeRes`. La liste est vide si la fonction n'a pas besoin de paramètres.

Nature de l'information retournée
Précisez toujours `TypeRes` même si le langage définit le type `int` par défaut.

Remarque
En théorie, le type de la valeur retournée peut être un type simple (entier, réel, booléen...), un type structuré, un tableau ou même un objet (ces types seront vus dans les modules suivants). En pratique il conviendra de s'en tenir aux limitations du langage utilisé.

Remarque
Les paramètres formels deviennent automatiquement des variables locales (cf. plus bas, @[Variable locale]) du module.

3.2 Instruction de retour

Instruction de retour

```
return expression;
```

Explication

Renvoie (retourne) au module appelant le résultat de l'`expression` placée à la suite du mot-clé.

C/C++ : return

L'instruction provoque la terminaison de la fonction.

Dans une fonction

Il doit **toujours** y avoir l'exécution d'une primitive `return`, et ceci quelles que soient les situations (conditions).

En effet, si dans un cas particulier, la fonction s'exécute sans être passée par cette primitive, ceci révèle une incohérence dans la conception de votre fonction car celle-ci aura une valeur inconnue et aléatoire.

3.3 Appel d'une fonction

Appel d'une fonction

```
v = nomFcn( a1, ..., aN )
```

Explication

Appelle (on dit aussi *invoque*) la fonction `nomFcn` avec les (éventuels) arguments `aI`. La valeur retournée peut être utilisée en tant que macro-expression.

Fonction = macro-expression

Une fonction retourne **toujours** une information à l'algorithme appelant. C'est pourquoi l'appel d'une fonction **ne se fait jamais** à gauche du signe d'affectation.

3.4 Schéma d'une fonction

Schéma d'une fonction

```
TypeRes nomFcn(TypeParam1 param1,TypeParam2 param2,...)
{
  TypeRes resultat = valeurInitiale;
  calcul_du_resultat;
  return resultat;
}
```

Explication

Pour des questions de lisibilité et de preuve de programme, il vous est fortement recommandé d'adopter le schéma ci-dessus.

C/C++ : Pas de fonctions imbriquées

Contrairement à d'autres langages (comme Pascal ou Python), il n'est pas possible en C/C++ de définir de fonctions imbriquées : elles doivent toutes se trouver au « premier niveau » du programme, indépendamment de leur place dans la hiérarchie de décomposition de l'algorithme. Ceci explique la structure générale d'un programme C/C++ : celui-ci est composé de fonctions – dont une seule a pour nom `main()` : c'est le point d'entrée du programme – et de variables globales, ces deux éléments pouvant étant répartis dans différents fichiers sources (le résultat de leurs compilations se regroupant dans un exécutable).

Expression fonctionnelle

```
TypeRes nomFcn(TypeParam1 param1,TypeParam2 param2,...)
{
  return expression;
}
```

Explication

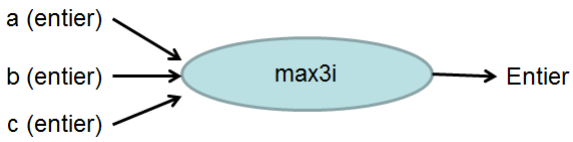
Dans le cas d'une expression calculable directement, on peut regrouper le tout : on parle alors d'**expression fonctionnelle**.

3.5 Exemple : Maximum de trois entiers

Retour sur l'exemple

Voici l'algorithme du deuxième problème de la section @[Motivation] :

« calculer le maximum de trois nombres (ici des entiers) »



Rappelons l'approche de résolution :

```
(1) tmp <- calculer le maximum des deux premiers nombres
(2) tmp <- calculer le maximum de tmp et du troisième nombre
```

D'où la définition de deux fonctions :

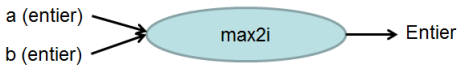
- Fonction `max2i` : maximum de deux entiers
- Fonction `max3i` : maximum de trois entiers

Fonction max2i

```
int max2i(int a, int b)
{
  return (a > b ? a : b);
}
```

Explication

La fonction prend deux entiers `a` et `b` (paramètres d'entrée) et renvoie le plus grand de `a` et `b`.



Remarque

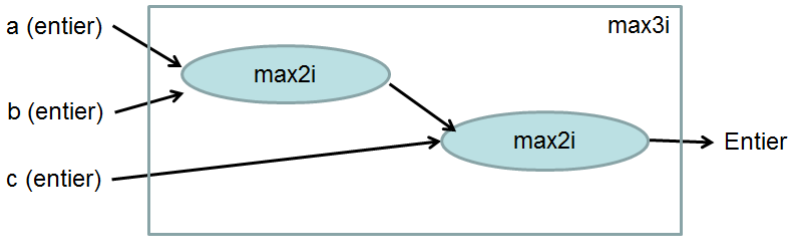
Sous cette forme, toute fonction est devenue « inactive » : elle ne lit rien ni n'affiche rien mais elle est cependant prête à être utilisée. Il suffira d'écrire son nom suivi d'un nombre de variables (ou en entrée d'expressions) en accord avec son en-tête.

Fonction max3i

```
int max3i(int a, int b, int c)
{
  int tmp = max2i(a,b);
  return max2i(tmp,c);
}
```

Explication
La fonction `max3i` calcule et renvoie le plus grand de ses trois paramètres d'entiers par appel à la fonction `max2i` comme suit :

- Le contenu des variables `a` et `b` est affecté aux paramètres d'entrée `a` et `b` de `max2i` puis cette fonction entre en action.
- A l'issue de la fonction `max2i`, celle-ci renvoie le maximum de `a` et `b` qui est stocké dans la variable locale `tmp`.
- Puis le contenu des variables `tmp` et `c` est affecté aux paramètres d'entrée `a` et `b` de `max2i` qui alors entre en action.
- A l'issue de la fonction `max2i`, elle renvoie le maximum de `tmp` et `c`.



Fonction max3i (Expression fonctionnelle)

```
int max3iv2(int a, int b, int c)
{
  return max2i(max2i(a,b),c);
}
```

Explication
L'expression fonctionnelle permet une écriture plus compacte.

Programme

```
#include <iostream>
using namespace std;
#include "UtilsSS.cpp"

int main()
{
  int n1, n2, n3;
  cout<<"Trois entiers? ";
  cin>>n1>>n2>>n3;
  int vmax = UtilsSS::max3i(n1,n2,n3);
  cout<<"Max des trois "<<vmax<<endl;
}
```

3.6 Inclure un fichier

C++ : Inclure un fichier

```
#include "NomFichier"
```

Explication
Insère le fichier `NomFichier` à la place de la directive `#include`. Par défaut, l'extension du fichier est ".hpp".

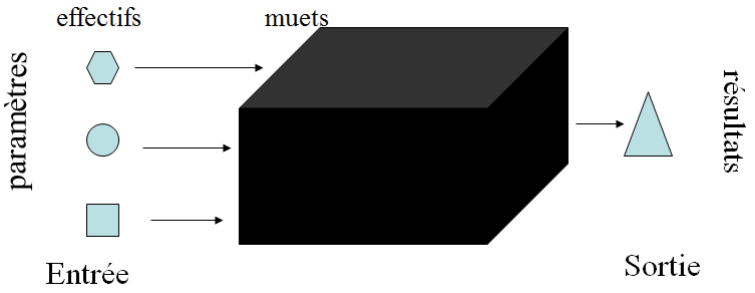
Remarque
Dans le cas de l'exemple, elle insère le fichier `"UtilsSS.cpp"` lequel contient les fonctions `max3i` et `max2i`.

3.7 Variable locale

Variable locale

Toute variable est **locale** au module dans lequel elle apparaît, ce qui veut dire que son existence est ignorée en dehors de ce module.

De façon imagée, vous ne voyez pas ce qu'il y a à l'intérieur de la « boite noire ».



Conséquence

Plutôt qu'une restriction, cette propriété est une aide confortable au programmeur : si, de façon fortuite, des variables appartenant à des modules différents possédaient le même nom, il n'y aurait pas d'interférence entre ces variables, ni confusion possible de leur contenu.

Exemple

Dans le module `max3i` (cf @[Exemple : Maximum de trois entiers]), les paramètres d'entrée `a`, `b` et `c` ainsi que les variables temporaires `tmp` et `vmax` sont locaux à ce module.

Règle des paramètres formels et variables d'un module

Les paramètres formels d'un module ne doivent plus être déclarés dans la partie « déclaration de variables » de ce module. Mais toutes les variables utilisées dans un module doivent être déclarées, soit dans son en-tête, soit dans sa déclaration. Le non-respect de cette règle provoque une erreur d'exécution.

Remarque

Une variable locale est aussi **dynamique**, c.-à-d. qu'un emplacement en mémoire lui est réservé durant l'exécution du module où elle est déclarée. Une fois l'exécution terminée, cet emplacement est récupéré en mémoire.

4 Procédure de test

Motivation

L'encapsulation d'un programme dans une **procédure de test** `test_xxx` permet de :

1. Décomposer un problème (= l'algorithme) en sous-problèmes (= les modules).
2. Vérifier chacun des sous-problèmes de façon plus ou moins indépendante.
3. Valider le problème final en écrivant un unique algorithme (= le programme).

C/C++ **Procédure de test**

```c
void test_xxx()
{
  // écriture des instructions du mini-"main"
}

int main()
{
   test_xxx(); // appelle de la procédure de test
}
```

Remarque

Au fur et à mesure de l'avancement, l'unique modification dans le corps du programme sera le nom de la procédure de test en cours de vérification.

Que signifie "Testez..." ?

Dans tout exercice, l'énoncé :

« Testez avec les exemples d'exécution. »

Signifie :

« Appelez la procédure `test_xxx` dans le programme principal
puis exécutez-la avec les exemples d'exécution. »