Pile
Corrigé

Objectifs

— Définir un module
— Utiliser la généricité
— Définir des sous-programmes, des types utilisateurs, tester, etc.

Exercice 1 : Pile de caractères

Une pile est une structure de données contenant des données de même type, ici caractère. Le principe d'une pile est que le dernier élément ajouté est le premier à en être retiré. Si on considère une pile d'assiettes, on pose une nouvelle assiette au sommet de la pile. Si on prend une assiette, c'est naturellement celle du sommet. On dit que la pile est une structure de données de type LIFO (Last In, First Out : dernier entré, premier sorti). Les opérations sur une pile sont les suivantes :

— Empiler (ajouter) un nouvel élément dans une pile.
— Dépiler une pile (supprimer le dernier élément ajouté). La pile ne doit pas être vide.
— Accéder à l'élément en sommet de pile. La pile ne doit pas être vide.
— Savoir si une pile est vide ou non.

Bien sûr , il faut aussi pouvoir initialiser une pile. La pile est alors vide. Une variable de type Pile ne peut être utilisée que si elle a été initialisée.

1. *Architecture.* Cette notion de pile pourra être utilisée dans différents programmes. Indiquer quel est le concept qui permet de réutiliser cette pile dans différents programmes.

Solution : Module

Rappeler ses principales caractéristiques.

Solution : Un module est constitué d'une **interface** (ou **spécification**) qui spécifie les éléments accessibles d'un utilisateur de ce module (sous-programme ou autre module) et d'un **corps** (ou **implantation**) qui implante les éléments spécifiés dans l'interface.

Ceci permet la masquage d'information en cachant à l'utilisateur l'implantation. Ceci permet de faire évoluer l'implantation sans remettre en cause les programmes ou autres modules qui l'utilisent.

2. *Spécifier la pile.* Écrire la spécification de la pile décrite ci-dessus.

Solution : Ci-après l'interface (ou spécification) du module *Piles*. Notons que le type `T_Pile` n'est pas défini. Il est déclaré comme privé car l'utilisateur n'a pas à le connaître et même très privé car on ne veut pas autoriser l'affectation ni le test d'égalité.

```
1  -- Spécification du module Piles.
2
3  package Piles is
4
5      type T_Pile is limited private;    --! "très privé" en Algorithmique !
6          --! Sur un type privé, on a droit à l'affectation (:=) et l'égalité (=).
7          --! On perd ces opérations avec un type "limited private" (très privé).
8
9
10     -- Initilaiser une pile.  La pile est vide.
11     procedure Initialiser (Pile : out T_Pile) with
12         Post => Est_Vide (Pile);
13
14
15     -- Est-ce que la pile est vide ?
16     function Est_Vide (Pile : in T_Pile) return Boolean;
17
18
19     -- L'élément en sommet de la pile.
20     function Sommet (Pile : in T_Pile) return Character with
21         Pre => not Est_Vide (Pile);
22
23
24     -- Empiler l'élément en somment de la pile.
25     -- Exception : Storage_Exception s'il n'y a plus de mémoire.
26     procedure Empiler (Pile : in out T_Pile; Element : in Character) with
27         Post => Sommet (Pile) = Element;
28
29
30     -- Supprimer l'élément en sommet de pile
31     procedure Depiler (Pile : in out T_Pile) with
32         Pre => not Est_Vide (Pile);
33
34
35     -- Détruire une pile.
36     -- Cette pile ne doit plus être utilisée sauf à être de nouveau initialisée.
37     procedure Detruire (P: in out T_Pile);
38
39
40
41 end Piles;
```

3. *Tester la pile.* Écrire un programme de test de la pile qui :

— initialise une pile,

— empile successivement les caractères 'o', 'k', puis ' ?',

— vérifie que le sommet est '?',

— dépile 3 fois,

— vérifie que la pile est vide.

Solution :

```
with Piles;     use Piles;

-- Programme de "test" du module Pile.
procedure Exemple_Sujet is

    Une_Pile: T_Pile;
begin
    -- Initialiser une pile
    Initialiser (Une_Pile);

    -- empiler successivement les caractères 'o', 'k', puis ' ?'
    Empiler (Une_Pile, 'o');
    Empiler (Une_Pile, 'k');
    Empiler (Une_Pile, '?');

    -- vérifier que le sommet est '?'
    pragma Assert ('?' = Sommet (Une_Pile));

    -- dépiler 3 fois
    for I in 1..3 loop
        Depiler (Une_Pile);
    end loop;

    -- vérifier que la pile est vide
    pragma Assert (Est_Vide (Une_Pile));

end Exemple_Sujet;
```

4. *Afficher un entier naturel.* Les opérations de la pile étant spécifiées, on peut écrire des programmes les utilisant même si l'implantation de ces opérations n'est pas encore définie.

4.1. Pour afficher un entier, on peut utiliser le sous-programme qui affiche un caractère et une pile de caractères pour conserver les caractères correspondant aux chiffres de l'entier. Écrire un tel sous-programme.

Solution : Maintenant que les opérations sur la pile ont été spécifiées, il est possible de les utiliser pour écrire le sous-programme demandé.

```
with Ada.Text_IO;               use Ada.Text_IO;
with Piles;     use Piles;

procedure Afficher_Entier is

    -- Afficher un entier naturel sur la sortie standard.
    procedure Afficher (N : in Integer) with
```

```
8           Pre => N >= 0
9       is
10          Nombre   : Integer;       -- le nombre à afficher (copie de N)
11          Unite    : Integer;       -- un chiffre de Nombre
12          Chiffre  : Character;     -- le caractère correspondant à Unite.
13          Chiffres : T_Pile;        -- les chiffres de Nombre
14
15      begin
16          -- Empiler les chiffres de l'entier
17          Initialiser (Chiffres);
18          Nombre := N;
19          loop
20              -- récupérer le chiffre des unités
21              Unite := Nombre Mod 10;
22
23              -- le convertir en un caractère
24              Chiffre := Character'Val (Character'Pos('0') + Unite);
25
26              -- l'empiler
27              pragma Assert (not Est_Pleine (Chiffres));
28              Empiler (Chiffres, Chiffre);
29
30              -- réduire le nombre en supprimant les unités
31              Nombre := Nombre / 10;
32          exit when Nombre = 0;
33          end loop;
34          pragma Assert (Nombre = 0);
35          pragma Assert (not Est_Vide (Chiffres));
36
37          -- Afficher les chiffres de la pile
38          loop
39              -- Obtenir le chiffre en sommet de pile
40              Chiffre := Sommet (Chiffres);
41
42              -- afficher le caractère
43              Put (Chiffre);
44
45              -- supprimer le caractère de la pile
46              Depiler (Chiffres);
47          exit when Est_Vide (Chiffres);
48          end loop;
49      end Afficher;
50
51
52
53  begin
54      Put ("10 = ");
55      Afficher (10);
56      New_Line;
57
```

```
58      Put ("0 = ");
59      Afficher (0);
60      New_Line;
61
62      Put ("Integer'Last = ");
63      Afficher (Integer'Last);
64      New_Line;
65  end Afficher_Entier;
```

4.2. Aurait-il été possible d'écrire cette application sans utiliser la pile (ni une autre structure de données : liste, tableau...) ?

Solution : On peut le faire en utilisant la récursivité.

On peut aussi le faisant une série de Div, mais il faut commencer par trouver la puissance de 10 qui correspond au chiffre significatif.

4.3. Comment faire si on veut utiliser ce sous-programme dans différents programmes ?

Solution : Il faut définir un module.

En Ada, il serait possible de définir une spécification et une implantation pour un seul sous-programme. C'est une unité de compilation, un genre de module dégénéré...

5. *Implanter la pile.* On choisit de représenter la pile par un enregistrement composé d'un tableau de 20 caractères (les éléments mis dans la pile) et un entier (le nombre d'éléments dans la pile). Le i^e élément ajouté dans la pile est à la i^e position du tableau.

Écrire l'implantation de la pile.

Solution : 20 est arbitraire. Il faut en faire une constante...

L'implantation choisie limite la taille de la pile. Il faut donc revoir les sous-programme de la spécification du module et les programmes qui l'utilisent !

Remarque : Notons qu'il est important de déclarer le type `T_Pile` comme très privé car l'égalité proposée par Ada ne fonctionnera pas car elle va comparer tous les éléments du tableau de 1 à Capacité alors que seuls les éléments de 1 à Taille doivent être comparés.

On pourrait surcharger l'opération de comparaison pour la rendre disponible à l'utilisateur.

Lors de l'écriture du code Pascal, on pourra instrumenter les préconditions et les postconditions en utilisant le module DBC vu en TP (et les procédures « nécessite » et « assure »).

```
1  -- Implantation du module Piles.
2
3
4  package body Piles is
5
6
7      procedure Initialiser (Pile : out T_Pile) is
8      begin
9          Pile.Taille := 0;
10     end Initialiser;
11
12
13     function Est_Vide (Pile : in T_Pile) return Boolean is
14     begin
15         return Pile.Taille = 0;
```

```
16      end Est_Vide;
17
18
19      function Est_Pleine (Pile : in T_Pile) return Boolean is
20      begin
21          return Pile.Taille >= Capacite;
22      end Est_Pleine;
23
24
25      function Sommet (Pile : in T_Pile) return Character is
26      begin
27          return Pile.Elements (Pile.Taille);
28      end Sommet;
29
30
31      procedure Empiler (Pile : in out T_Pile; Element : in Character) is
32      begin
33          Pile.Taille := Pile.Taille + 1;
34          Pile.Elements (Pile.Taille) := Element;
35      end Empiler;
36
37      procedure Depiler (Pile : in out T_Pile) is
38      begin
39          Pile.Taille := Pile.Taille - 1;
40      end Depiler;
41
42
43  end Piles;
```

Exercice 2 : Généralisation de la pile

Dans l'exercice 1, nous avons défini une pile de caractères de capacité 20. En réalité, c'est l'utilisateur de la pile qui est le mieux placé pour décider de la capacité de la pile. Ce même utilisateur pourrait aussi souhaiter avoir une pile d'entiers, de réels, de dates, etc.

1. Expliquer comment faire pour permettre à l'utilisateur de la pile de choisir le type de ses éléments et sa capacité.

Solution : Grâce à la généricité, on paramètre le module Piles par le type des éléments et la capacité qui deviennent des paramètres de généricité.

2. Adapter en conséquence le module ainsi que les programmes qui l'utilisent.

Solution :

- On définit les deux paramètres de généricité : la capacité (`Capacite`) et le type des éléments de la pile (`T_Element`).
- On supprime la constante `Capacite`.
- On remplace les occurrences de `Character` par `T_Element` dans l'interface et le corps du module.

Voici l'interface du module Piles.

```
-- Spécification du module Piles.

generic
    Capacite : Integer;   -- Nombre maximal d'éléments qu'une pile peut contenir
    type T_Element is private;  -- Type des éléments de la pile

package Piles is

    type T_Pile is private;

    -- Initilaiser une pile.  La pile est vide.
    procedure Initialiser (Pile : out T_Pile) with
        Post => Est_Vide (Pile);

    -- Est-ce que la pile est vide ?
    function Est_Vide (Pile : in T_Pile) return Boolean;

    -- Est-ce que la pile est pleine ?
    function Est_Pleine (Pile : in T_Pile) return Boolean;

    -- L'élément en sommet de la pile.
    function Sommet (Pile : in T_Pile) return T_Element with
        Pre => not Est_Vide (Pile);

    -- Empiler l'élément en somment de la pile.
    procedure Empiler (Pile : in out T_Pile; Element : in T_Element) with
        Pre => not Est_Pleine (Pile),
        Post => Sommet (Pile) = Element;

    -- Supprimer l'élément en sommet de pile
    procedure Depiler (Pile : in out T_Pile) with
        Pre => not Est_Vide (Pile);

private

    type T_Tab_Elements is array (1..Capacite) of T_Element;

    type T_Pile is
        record
            Elements : T_Tab_Elements;    -- les éléments de la pile
            Taille: Integer;        -- Nombre d'éléments dans la pile
        end record;

end Piles;
```

Voici le corps du module Piles.

```
-- Implantation du module Piles.

package body Piles is
```

```
4
5       procedure Initialiser (Pile : out T_Pile) is
6       begin
7           Pile.Taille := 0;
8       end Initialiser;
9
10      function Est_Vide (Pile : in T_Pile) return Boolean is
11      begin
12          return Pile.Taille = 0;
13      end Est_Vide;
14
15      function Est_Pleine (Pile : in T_Pile) return Boolean is
16      begin
17          return Pile.Taille >= Capacite;
18      end Est_Pleine;
19
20      function Sommet (Pile : in T_Pile) return T_Element is
21      begin
22          return Pile.Elements (Pile.Taille);
23      end Sommet;
24
25      procedure Empiler (Pile : in out T_Pile; Element : in T_Element) is
26      begin
27          Pile.Taille := Pile.Taille + 1;
28          Pile.Elements (Pile.Taille) := Element;
29      end Empiler;
30
31      procedure Depiler (Pile : in out T_Pile) is
32      begin
33          Pile.Taille := Pile.Taille - 1;
34      end Depiler;
35
36  end Piles;
```

Le module Piles étant maintenant générique, on ne peut pas l'utiliser directement. Il faut commencer par l'instancier en précisant la valeur de ses paramètres de généricité. On peut alors utiliser le module instancié.

Voici l'exemple d'utilisation.

```
1   with Piles;
2
3   -- Programme de "test" du module Pile.
4   procedure Exemple_Sujet is
5
6       package Pile_Caractere is
7           new Piles (Capacite => 20, T_Element => Character);
8       use Pile_Caractere;
9
10      Une_Pile: T_Pile;
```

```
11  begin
12      -- Initialiser une pile
13      Initialiser (Une_Pile);
14
15      -- empiler successivement les caractères 'o', 'k', puis ' ?'
16      Empiler (Une_Pile, 'o');
17      Empiler (Une_Pile, 'k');
18      Empiler (Une_Pile, '?');
19
20      -- vérifier que le sommet est '?'
21      pragma Assert ('?' = Sommet (Une_Pile));
22
23      -- dépiler 3 fois
24      for I in 1..3 loop
25          Depiler (Une_Pile);
26      end loop;
27
28      -- vérifier que la pile est vide
29      pragma Assert (Est_Vide (Une_Pile));
30
31  end Exemple_Sujet;
```

3. On souhaite disposer d'une nouvelle opération sur une pile : l'afficher sur le terminal. Ajouter cette opération et l'utiliser.

Solution : On peut facilement la spécifier dans l'interface du module :

```
1       -- Afficher les éléments de la pile
2       procedure Afficher (Pile : in T_Pile);
```

Mais quand on veut écrire le corps de sous-programme, on rencontre une difficulté : on ne sait pas afficher un élément puisqu'on ne connait pas son type !

La solution est donc de définir le sous-programme qui affiche un élément de la pile comme un paramètre de généricité. Où définir ce paramètre de généricité ?

On pourrait le définir sur le module Piles mais ceci a deux inconvénients :

1. Il faudra le fournir pour toute instanciation du module Piles, même si on utilise pas sa procédure Afficher. Par exemple, pour afficher un entier, on a utiliser une pile sans utiliser la procédure Afficher.
2. On aura qu'une seule possibilité pour afficher la pile. Si on peut afficher la pile en affichant différemment ses éléments, il faudra créer une nouvelle instance du module Piles (et créer une nouvelle pile à partir de la première pour utiliser la procédure Afficher du nouveau module).

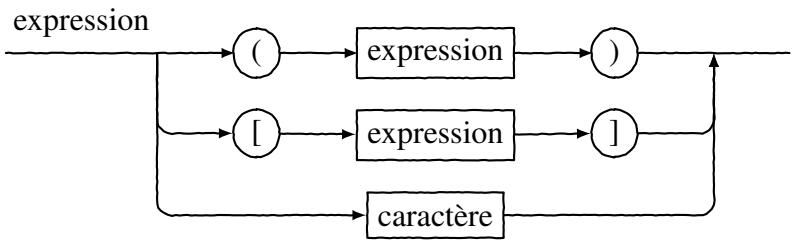
La bonne solution est donc de définir le paramètre de généricité sur la procédure Afficher. Ainsi, ce n'est que quand on voudra afficher une pile qu'il faudra instancier cette procédure générique et fournir la procédure qui affiche un élément de la pile. De plus, on pourra l'instancier plusieurs fois si on veut plusieurs manière d'afficher un élément de la pile. Voici la spécification de cette procédure générique Afficher (dans l'interface du module Piles).

```
1       -- Afficher les éléments de la pile
2       generic
3           with procedure Afficher_Element (Un_Element: in T_Element);
4       procedure Afficher (Pile : in T_Pile);
```

Exercice 3 : Expressions bien parenthésées

l'objectif de cet exercice est de vérifier que des expressions sont bien parenthésées par rapport à l'imbrication des parenthèses et des crochets.

Une expression est bien parenthésée si elle respecte le diagramme syntaxique suivant :



Sachant que « caractère » est n'importe quel caractère, à l'exception des parenthèses et des crochets bien sûr.

Voici quelques exemples d'expressions bien parenthésées :

```
2 * (x + y)
occurrences[chr(ord('0') + (entiers[i] mod 4))]
[ ( ( [ ] ) [ ( ) ( ) ] ) ]
```

et d'expressions mal parenthésées :

```
2 * (x + y))
occurrences[chr(ord('0')] + (entiers[i] mod 4))
[[[[
]
( ( ] )
( [ ) ]
```

Pour vérifier qu'une expression est bien parenthésée, on peut utiliser une pile. Les symboles ouvrant sont empilés et lorsque l'on rencontre un symbole fermant, on vérifie que le sommet de la pile contient bien le symbole ouvrant correspondant. Si ce n'est pas le cas, c'est que l'expression est mal formée. De plus, en fin d'analyse de l'expression, la pile doit être vide ce qui garantit que tous les caractères ouvrants ont été fermés.

Écrire un sous-programme qui vérifie si une chaîne de caractères est bien parenthésée et qui indique, si la chaîne est mal parenthésée, l'indice que caractère qui n'est pas appairé.

Solution : Cet exercice sera repris en TP.