TD-TP 18 expression (simple)

Thème : 2 TD et 2 TP soit 16 heures étalées sur deux semaines pour réaliser la saisie, la validation et le calcul d'une expression arithmétique simplifiée.

Concepts : Expression (voir polycopié cours Ada et Génie Logiciel), analyse lexicale, analyse syntaxique, priorité des opérateurs, diagrammes syntaxiques, utilisation de piles, types étiquetés, types à échelle de classe, pointeurs. Beaucoup de synthèses Ada !

Avant propos :
Il en recommandé de relire, avant de commencer l'exercice, le « cours expression». Nous allons appliquer les notions vues sur un « cas simple » : l'expression arithmétique.

« Définition » de **notre** expression arithmétique : elle sera **simple** c'est-à-dire formée à l'aide :

des opérateurs + - * / (tous binaires)
des parenthèses ( )
des opérandes (littéraux entiers base 10)
finie par un ;

EXEMPLES :

((23-76) + (6*41)/4) * 3 ;

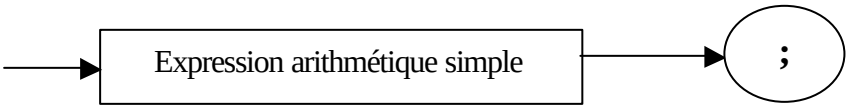
(23-76 + 6*41/4)*3 ;

5 ;

En imaginer d'autres !

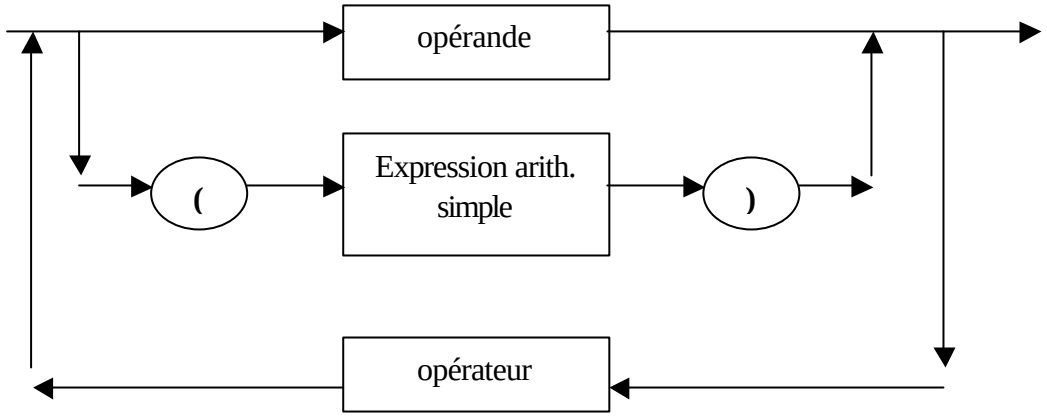
Application du cours : comment modéliser ? Þ Avec des D.S. !

Notre expression arithmétique :



Il faut évidemment détailler !

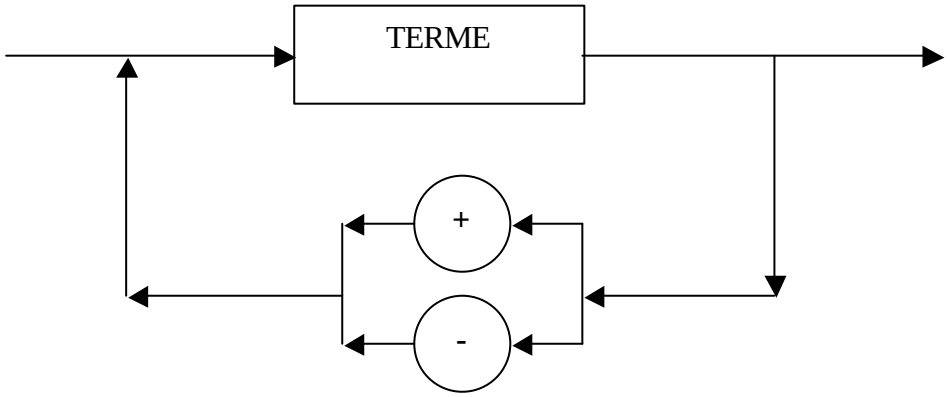
expression arithmétique simple (première présentation) :



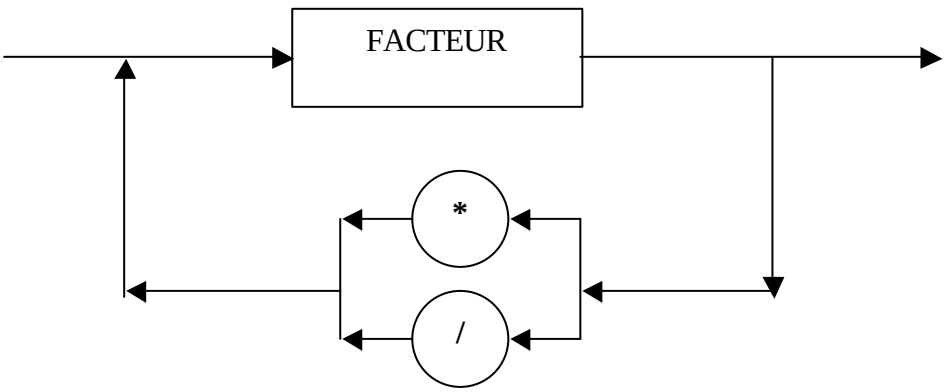
Notez bien les flèches évidemment ! Notez la définition récursive (important !). Les définitions d'opérateur et d'opérande sont évidentes. Faites les (corrigées en TD !). Cette modélisation servira pour la phase d'analyse syntaxique (voir plus loin !).

Autre définition permettant de prendre en compte le concept de priorité des opérateurs :

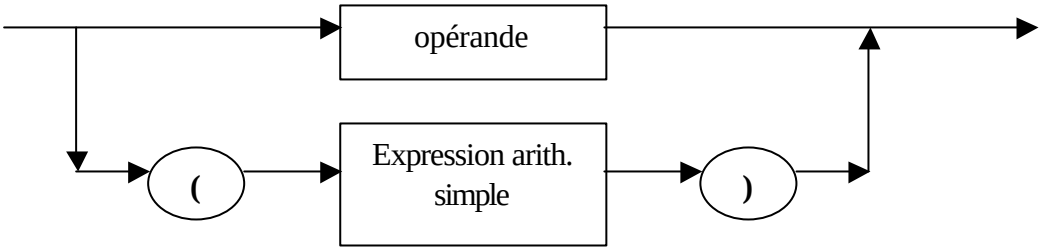
expression arithmétique simple (deuxième présentation) :



TERME :



FACTEUR :



Faites quelques simulations avec des exemples !

PROBLEMES :

Comment résoudre notre problème à savoir : saisie, validation et calcul d'une expression arithmétique simple ?

- lecture (ou saisie) ce sera simple ! (on lira un `Unbounded_String` !)
- validation (analyse lexicographique puis syntaxique : vive la récursivité !)
- exécution (mise en forme spéciale avant le calcul : joli morceau)

Pour la suite on peut remarquer qu'il serait intéressant de manipuler, **plutôt que des « morceaux » de chaînes de caractères**, les 4 entités suivantes :

- Opérandes
- Opérateurs
- Parenthèses
- Terminateur

Ce sont des « **TOKENS** »

Ces entités (opérandes, opérateurs, parenthèses, terminateur) doivent être "partageables". Pour représenter ces quatre entités en mémoire **il serait possible** d'utiliser un article mutant :

```
type T_Nature is (Operande,Operateur,Parenthese,Terminateur);
type T_Operateur  is  ('+', '-', '*', '/') ;
type T_Parenthese is  ('(', ')');
type T_Terminateur is (';');
```

puis

```
type T_Token  (La_Nature : T_Nature := Operande) is
 record
    case  La_Nature is
      when  Operande    =>  L_Operande     : Integer ;
      when  Operateur   =>  L_Operateur    : T_Operateur ;
      when  Parenthese  =>  La_Parenthese  : T_Parenthese ;
      when  Terminateur =>  Le_Terminateur : T_Terminateur ;
    end case ;
 end record ;
```

Nous **allons procéder autrement** :

Avec Ada95 nous allons pouvoir définir un type de base « `T_Token` » (voir le paquetage `P_Token` ci dessous) et obtenir par dérivation, des types correspondant aux quatre entités en question : opérandes, opérateurs, parenthèses et terminateur. Ceci nous permettra d'utiliser des types « tagged » et aussi des pointeurs sur types à échelle de classes. Voyons cela.

```
-- fichier p_token.ads

package P_Token is
   type T_Token is abstract tagged null record;
   procedure Ecrire(Token : in T_Token) is abstract;
end P_Token;
```

ce paquetage crée un type dérivable mais « vide » et même abstrait (cf. cours n° 10).

On va s'appuyer sur ce paquetage `P_Token` pour créer les paquetages fils :

- `P_Token.Operateur`
- `P_Token.Parenthese`
- `P_Token.Terminateur`
- `P_Token.Operande`

qui contiendront les types dérivés. Voir par exemple le paquetage `P_Token.Operateur`

```
-- fichier p_token-operateur.ads

package P_Token.Operateur is

   type T_Operateur is ('*','-','+','/');
   type T_Token_Operateur is new T_Token with
     record
      L_Operateur : T_Operateur;
     end record;

   procedure Initialisation (Token : in out T_Token_Operateur;
      Elem  : in  T_Operateur);
   function  Get_Elem (Token : in T_Token_Operateur) return T_Operateur;
   procedure Ecrire (Token : in T_Token_Operateur);

end P_Token.Operateur;
```

A commenter en TD !

Ce paquetage définit le type `T_Token_Operateur` et les trois méthodes :

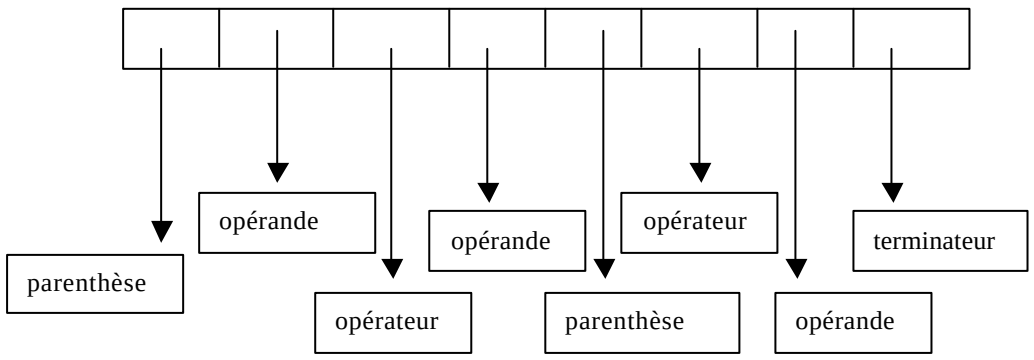
- `Initialisation` pour donner une valeur au token.
- `Get_Elem` pour récupérer la valeur du token.
- `Ecrire` pour afficher la valeur du token.

La réalisation est simple (commentée aussi en TD) :

```
-- fichier p_token-operateur.adb

package body P_Token.Operateur is

   procedure Initialisation (Token: in out T_Token_Operateur;
         Elem : in T_Operateur) is
   begin
      Token.L_Operateur := Elem;
   end Initialisation;

   function Get_Elem (Token : in T_Token_Operateur) return T_Operateur is
   begin
      return Token.L_Operateur;
   end Get_Elem;

   procedure Ecrire (Token : in T_Token_Operateur) is
   begin
      Put(T_Operateur'Image(Token.L_Operateur));
   end Ecrire;
end P_Token.Operateur;
```

Les autres paquetages : `P_Token.Parenthese, P_Token.Terminateur, P_Token.Operande` **sont à réaliser en TP**. Pas très difficile !

A partir de ces 4 paquetages, il sera possible de construire le paquetage `P_Expression`. Ce paquetage doit disposer d'une structure de données permettant le stockage des différents Tokens. La structure de données ne pourra contenir que des éléments à échelle de classe (`T_Token'Class`) puisque l'on ne connaît pas à l'avance le type de base du Token. Une solution en Ada pour garder en mémoire une collection d'objets hétérogènes est d'utiliser un tableau de pointeurs vers les divers objets. Soit l'expression (41 + 3) * 27 ;



Il est donc nécessaire de définir des pointeurs, d'une part sur le type de base T_Token et sur les différents types dérivés du type de base T_Token.
On va définir ces types pointeurs dans les paquetages fils suivant :

- `P_Token.F` (voir plus bas)
- `P_Token.Operande.F`
- `P_Token.Operateur.F` (voir plus bas)
- `P_Token.Parenthese.F`
- `P_Token.Terminateur.F`

```
-- fichier p_token-f.ads
package P_Token.F is
   type T_Ptr_Token is access all T_Token'Class;
end P_Token.F;
```

Le paquetage `P_Token.F` contient la déclaration du pointeur sur le type à échelle de classe.
Le paquetage `P_Token.Operateur.F` permet de récupérer la valeur d'un pointeur de type `T_Ptr_Token` mais qui « réfère » un objet de type `T_Operateur`.

```
-- fichier p_token-operateur-f.ads
with P_Token.F; use  P_Token.F;
package P_Token.Operateur.F is
   function Set_Ptr  (Elem : in T_Operateur) return T_Ptr_Token;
end P_Token.Operateur.F;

-- fichier p_token-operateur-f.adb
package body  P_Token.Operateur.F is
   type T_Ptr_Token_Operateur is access all T_Token_Operateur;
   function Set_Ptr (Elem : in T_Operateur) return T_Ptr_Token is
      Ptr_Token : T_Ptr_Token_Operateur;
   begin
      Ptr_Token := new T_Token_Operateur;
      Initialisation(Ptr_Token.all,Elem);
      return Ptr_Token.all'access;
   end Set_Ptr;
end P_Token.Operateur.F;
```

Les 3 autres paquetages fils (à faire) contiennent donc, eux aussi, une fonction `Set_Ptr` renvoyant un pointeur sur un type à échelle de classe `T_Token'Class`.
La structure de données sera définie dans le paquetage `P_Expression` de la façon suivante :

```
 type T_Vect_Token is array (Positive range <>) of T_Ptr_Token ;
```

Les fonctionnalités à réaliser sont spécifiées dans le paquetage P_Expression (cf. page 9). Le corps sera à faire en TP à l'aide d'algorithmes schématiques détaillés ci dessous.

LECTURE et ANALYSE LEXICALE :

Vous utiliserez `Lire` dans le paquetage `P_Expression`. Inutile s'attarder sur cette procédure qui ne vous apprendra rien : c'est un `Get_Line` sur une chaîne de type `Unbounded_String` qui supprime les espaces ! Pour tester vos travaux vous redirigerez vos entrées dans un fichier texte (comme d'habitude!). Pour l'analyse lexicale, une fois la chaîne saisie, il faut créer les différents TOKENs et les placer dans le vecteur de pointeurs de TOKEN.

D'où l'algorithme (schématique) de la procédure `Analyse_Lexicale` (création des TOKENs) :

jusqu'à (fin de la chaine)
répéter
Choix sur caractère courant entre
'0'..'9' ⇒ Créer TOKEN (opérande, Val)
-- val est calculée par Horner
'+'|'-'|'*'|'/' ⇒ Créer TOKEN (opérateur, symbole)
-- symbole est l'un des opérateurs : + - * /
'('|')' ⇒ Créer TOKEN (parenthèse, parent)
-- parent est la valeur de la parenthèse en question
';' ⇒ Créer TOKEN (terminateur, ;)
autre ⇒ erreur
fin choix
ranger le TOKEN dans le vecteur de pointeur de TOKEN à fabriquer.
fin répéter

La chaîne de caractères représentant l'expression est donc transformée en un vecteur de Token ou mieux en un vecteur de pointeurs de Token ! Si le caractère courant n'est pas un caractère lexicalement valide il faut évidemment cesser l'activité.

VALIDATION (à passer en première lecture morceau car **difficile**) **:**

Tout d'abord il faut modéliser à l'aide du diagramme syntaxique (haut page 2) !

Réalisée avec la procédure `Syntaxe` elle est "facile" avec la récursivité (on utilisera comme d'habitude une procédure récursive interne) : `Valid`. La procédure `Syntaxe` appelle la procédure `Valid` puis la procédure `Fin` et gère les exceptions notamment celle qui aura permis de briser les retours récursifs en cas d'erreur (cf. TD récursivité !).
On comprendra mieux l'algorithme si on suit les diagrammes syntaxiques présentés au début.

Algorithme (schématique) de `Valid` (procédure récursive de `Syntaxe`)

action `Valid`
si Token_Courant est opérande
alors Suite
sinon si Token_Courant est (
alors Valid -- récursivité type 2 !
si Token_Courant est un Token)
alors Suite
sinon Erreur
fin si
sinon Erreur
fin si
fin si
fin action

La séquence Suite est :

si Token_Courant est Token opérateur
alors Valid
fin si

Le module Fin est :

si Token_Courant est terminateur
alors **si** dernier Token du vecteur
alors OK
sinon Erreur
finsi
sinon Erreur
fin si

L'expression étant validée il reste à l'évaluer ! C'est la dernière étape et la plus jolie. Elle n'est pas facile sauf si on la transforme auparavant en notation polonaise inversée (connue sur les calculatrices !).

Exemple:

((23-76) + (6*41)/4) * 3;
devient:
23 76 - 6 41 * 4 / + 3 *

D'autres exemples en TD !

d'où l'algorithme (schématique) de Polonaise :

jusqu'à (fin de l'expression mise en TOKEN)
répéter
choix sur nature du Token _Courant entre :
opérande => Token rangé dans Polon
parenthèse => **si** (augmenter la priorité
si) diminuer la priorité
terminateur => fin
opérateur => Calculer la priorité courante du signe
examiner les opérateurs précédents et
leur priorité (donc deux piles).
tant qu'il y a un autre opérateur meilleur
répéter le ranger dans Polon
fin répéter
empiler l'opérateur et sa priorité
fin choix
fin répéter
dépiler le reste des opérateurs restants

L'algorithme (schématique) de Calcul est alors très simple

Jusqu'à (fin du vecteur Polon)
répéter
choix sur nature du Token entre
opérande => on empile (la valeur seulement pas le pointeur)
opérateur => on dépile 2 opérandes
on effectue l'opération (ordre!!)
on empile le résultat
autre choix => Erreur
fin choix
fin répéter

Le résultat est en haut (et bas !) de la pile

Spécifications du paquetage `P_Expression` **:**

```
-- fichier p_expression.ads

with P_Token,P_Token.F, Ada.Strings.Unbounded.Text_Io;
use  P_Token,P_Token.F,Ada.Strings.Unbounded;
use  Ada.Strings.Unbounded.Text_Io;

package P_Expression is

   Max_Car : constant := 80; -- modifiable !

   type T_Vect_Token is array (Positive range <>) of T_Ptr_Token;

   procedure Lire (L_Expres : out Unbounded_String);

   procedure Analyse_Lexicale (V  : out T_Vect_Token;
         L_V : out Natural;
         C   : in  Unbounded_String);

   procedure Syntaxe   (V  : in T_Vect_Token;
         Ok : out Boolean);

   procedure Polonaise ( V_Entree  : in T_Vect_Token;
         V_Result  : out T_Vect_Token;
         Long_Res  : out Natural);

   function  Calcul ( V : in T_Vect_Token) return Integer;

   procedure Ecrire (V : in T_Vect_Token);

   Exc_Erreur : exception;

end P_Expression;
```

Le programme de test aura **en gros** l'allure suivante :

```
-- programme principal Expression
-- à compléter pour testabilité sérieuse!

with Ada.Text_Io;                  use Ada.Text_Io;
with Ada.Exceptions;               use  Ada.Exceptions;
with P_Expression;                 use  P_Expression;
with Ada.Strings.Unbounded.Text_Io;
use Ada.Strings.Unbounded,Ada.Strings.Unbounded.Text_Io;

procedure Expression is

   package P_Entier is new Ada.Text_Io.Integer_Io(Integer);
   use P_Entier;
   Marqueur: Exception_Occurrence;
   L_Expression : Unbounded_String;
   Exp_Arith    : T_Vect_Token (1..Max_Car); -- pour Analyse_Lexicale
   Lon_A        : Natural;
   Polon        : T_Vect_Token (1..Max_Car); -- pour Polonaise
   Lon_P        : Natural;
   Syntaxe_Ok   : Boolean;

begin
   loop
      exit when End_Of_File;
      begin
         Lire(L_Expression);
         Put (L_Expression);
         Ada.Text_Io.New_Line(2);
         Analyse_Lexicale(Exp_Arith,Lon_A,L_Expression);
         Ada.Text_Io.Put_Line("Analyse passée");
         Ecrire(Exp_Arith(1..Lon_A));
         Syntaxe (Exp_Arith(1..Lon_A),Syntaxe_Ok);
         Ada.Text_Io.Put_Line("Syntaxe passée ");
         if Syntaxe Ok
         then
            Polonaise (Exp_Arith(1..Lon_A),Polon,Lon_P);
            Ecrire (Polon(1..Lon_P));
            Ada.Text_Io.New_Line;
            Ada.Text_Io.Put ("résultat : ");
            Put(Calcul(Polon(1..Lon_P)));
            Ada.Text_Io.New_Line;
         else
            Ada.Text_Io.Put_Line ("erreur de syntaxe");
         end if;
      exception
         when Marqueur:others =>
            Ada.Text_Io.Put_Line(Exception_Information (Marqueur));
      end;
   end loop;
end Expression;
```

Les lignes en gras sont évidemment celles qui testent les algorithmes (à réaliser en TP)..
Le fichier de test contiendra une expression par ligne (valide ou non).

Quelques éléments pour les TP n° 18

Vous avez besoin (ou vous aurez à votre disposition) pour les deux TP:

A) les paquetages (fichiers specs)

- p_token.ads
- p_token-operande.ads (le corps adb est à réaliser)
- p_token-operateur.ads et p_token-operateur.adb (pour voir)
- p_token-terminateur.ads (le corps adb est à réaliser)
- p_token-parenthese.ads (le corps adb est à réaliser)

ainsi que les fichiers .o et .ali correspondants.

B) les paquetages fils (fichiers specs)

- p_token-f.ads
- p_token-operande-f.ads (le corps adb est à réaliser)
- p_token-operateur-f.ads et p_token-operateur-f.adb (pour voir)
- p_token-parenthese-f.ads (le corps adb est à réaliser)
- p_token-terminateur-f.ads (le corps adb est à réaliser)

ainsi que les fichiers .o et .ali correspondants.

C) votre paquetage pile générique fait précédemment (ou P_Pile_4 offert)

- p_pile_4.ads
- p_pile_4.adb

ainsi que les fichiers .o et .ali . Vous pouvez prendre votre P_Pile personnel !

D) le paquetage P_Expression dont il faut **refaire** le body (livré compilé)

- p_expression.ads

ainsi que les fichiers .o et .ali

E) la procédure Expression (programme d'essai à modifier !))

- expression.adb

PREMIER TRAVAIL (premier TD-TP):

0°) Testez d'abord `expression` (l'exécutable copié) avec le fichier `expression.in` (redirigé) avec la commande `./expression < expression.in > expression.out`. Imprimez le résultat. Faites un bon fichier de test (`expression.in`)

1°) Ecrire les corps des paquetages : (à l'aide de P_Token_Operateur)

- p_token-operande.adb
- p_token-parenthese.adb
- p_token-terminateur.adb

ainsi que les paquetages fils :

- p_token-operande-f.adb
- p_token-parenthese-f.adb
- p_token-terminateur-f.adb
-

Ne faites sur ces fichiers que la compilation avec gnatgcc.

2°) A partir du paquetage `P_Expression`, créons un paquetage fils `A_Moi`. Ce qui vous enlèvera la tentation de « toucher » à `P_Expression` (de toute façon les specs sont invariantes). Pour cela vous avez les fichiers (partiels) copiés :

- p_expression-a_moi.ads et p_expression-a_moi.adb

Ce paquetage fils ne contient, **pour le moment**, que la redéfinition de la procédure `Lire`.

```
-- fichier p_expression-a_moi.ads

with P_Token.F,Ada.Strings.Unbounded.Text_Io;
use  P_Token,P_Token.F,Ada.Strings.Unbounded;
use  Ada.Strings.Unbounded.Text_Io;

package P_Expression.A_Moi is
   procedure Lire (L_Expres : out Unbounded_String);
   -- à compléter au fur et à mesure :
   -- d'abord Calcul au TP1, puis, au TP2 :
   -- avec Analyse_Lexicale, Polonaise, Syntaxe, Ecrire
end P_Expression.A_Moi;
```

2°) Voyez le programme `Expression` (fichier expression.adb page 10). Il est organisé ainsi :

lecture d'une expression avec `Lire` (redirigez les entrées dans un fichier !)
la mettre en TOKEN avec `Analyse_Lexicale` (du paquetage père)
la valider avec `Syntaxe` (du paquetage père)
si valide la mettre en polonaise avec `Polonaise` (du paquetage père)
puis en calculer la valeur avec `Calcul` (du paquetage père)

Le programme boucle sur fin de fichier (clavier) donc on peut faire un fichier de test (une expression par ligne) et on redirige le clavier. Ceci pourra servir ensuite d'Oracle !

Ajoutez la référence au paquetage `P_Expression.A_Moi` (`with` et `use`). Remplacez `Lire` par `A_Moi.Lire`.
Compilez le programme. Rendez le exécutable. Faites des tests (l'Oracle).

Remarque importante : N'utilisez pas la commande gnatmake. Effectuez les opérations dans l'ordre :

- La compilation avec gnatgcc (gnatgcc –c expression.adb)
- L'édition de liens avec gnatbl (gnatbl –o expression expression.ali)

3°) Réécrire Calcul du paquetage `P_Expression` en écrivant `Calcul` dans `A_Moi`. Pour contrôler que **votre** procédure `Calcul` marche correctement, exécutez le programme `Expression` après avoir indiqué que la procédure `Calcul` provient bien du paquetage fils (Notation pointée `A_Moi.Calcul` comme pour `A_Moi.Lire`).

Il est possible de « voir » le vecteur de TOKEN avec `Ecrire` à chaque étape (pour contrôler et valider)

DEUXIEME TRAVAIL (et dernier TD-TP !):

Refaire tous les autres sous-programmes de `P_Expression` dans le paquetage fils `A_Moi` (un à un !) dans l'ordre: (sans oublier `Calcul` d'abord mais … déjà fait au premier TP !)

1°) `Analyse_Lexicale`
2°) `Polonaise`
3°) `Syntaxe` (c'est simple si on a vu la récursivité!)

Remarque (ou rappel) concernant `A_Moi` :

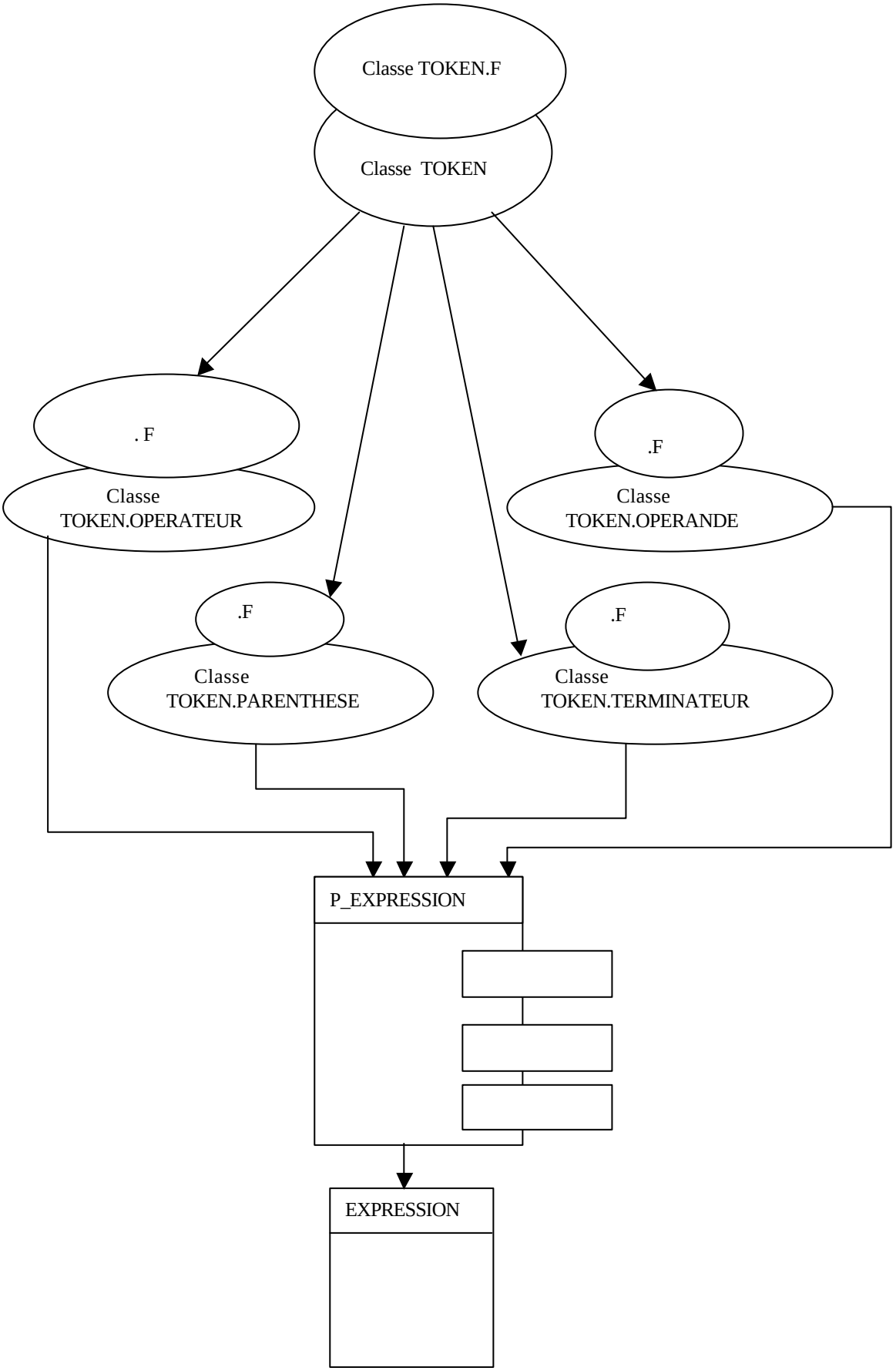
Au début seule la procédure `Lire` est définie. Testez votre programme principal avec `with Expression.A_Moi`. Ensuite vous déclarez la procédure `Calcul` dans la partie spécification du paquetage fils et vous écrivez la réalisation de `Calcul` dans le body de `A_Moi`. Puis vous retestez avec `Expression`, et vous recommencez avec les autres procédures en utilisant toujours votre `P_Pile_4` etc.

RESUME :

- TP1 : Vous réalisez les corps des paquetages des divers types de TOKENs. Vous testez Expression rapidement (fin du TD-TP 1) avec des plans de tests et rapports (**testabilité**). Vous réalisez si possible CALCUL.

- TP2 : Vous refaites un maximum de méthodes de P_Expression.

REMARQUE : Le but d'un paquetage fils n'est pas de redéfinir l'ensemble des procédures et fonctions définies dans le paquetage père, mais bien de compléter ce dernier avec de nouvelles méthodes. Ici la réalisation du paquetage fils A_Moi est **purement pédagogique** et n'a qu'un seul but, vous faire écrire les diverses procédures et fonctions qu'après les avoir utilisées. De plus en procédant ainsi vous êtes limités dans votre tentation de modifier les spécifications du paquetage père `P_Expression`.


Classe TOKEN.F
Classe TOKEN
. F
Classe
TOKEN.OPERATEUR
.F
Classe
TOKEN.OPERANDE
.F
Classe
TOKEN.PARENTHESE
.F
Classe
TOKEN.TERMINATEUR
P_EXPRESSION
EXPRESSION