Premiers pas

Chapitre 2

Premier exemple de programme Ada (1/2)

```
with Ada.Integer_Text_io;
with Ada.Text_io;
procedure exemple1 is
  maNote:Natural;
begin
  Ada.Integer_Text_io.get( maNote );
  maNote:=maNote+2;
  Ada.Text_io.put( "Nouvelle note :" );
  Ada.Integer_Text_io.put( maNote );
end exemple1;
```

Premier exemple de programme Ada (2/2)

Ce programme reflète la structure générale de tout programme Ada. On remarque 3 parties distinctes:

1. la première permet d'importer des objets externes (**`with ...`**)
2. la seconde permet de créer des objets localement (**`procedure...begin`**)
3. la troisième est une suite d'instructions qui opère sur ces objets (**`begin...end`**)

```
with Ada.Integer_Text_io;
with Ada.Text_io;
procedure exemple1 is
   maNote:Natural;
begin
   Ada.Integer_Text_io.get( maNote );
   maNote:=maNote+2;
   Ada.Text_io.put( "Nouvelle note :" );
   Ada.Integer_Text_io.put( maNote );
end exemple1;
```

```
with Ada.Integer_Text_io;
with Ada.Text_io;
procedure exemple1 is
   maNote:Natural;
begin
   Ada.Integer_Text_io.get( maNote );
   maNote:=maNote+2;
   Ada.Text_io.put( "Nouvelle note :" );
   Ada.Integer_Text_io.put( maNote );
end exemple1;
```

```
with Ada.Integer_Text_io;
with Ada.Text_io;
procedure exemple1 is
   maNote:Natural;
begin
   Ada.Integer_Text_io.get( maNote );
   maNote:=maNote+2;
   Ada.Text_io.put( "Nouvelle note :" );
   Ada.Integer_Text_io.put( maNote );
end exemple1;
```

Structure générale d'un programme Ada

```
-- imports
procedure nom_proc is
-- spécification des données
begin
-- suite d'instructions qui implantent l'algorithme
end  nom_proc;
```

Les lignes de texte commençant par `--` permettent d'introduire des commentaires
Ils peuvent être placés n'importe où dans le programme.

Notion de type de données

Tous les objets (locaux ou externes) ont un **type**.

Un type définit l'ensemble des valeurs que peut prendre un objet informatique ainsi que les opérations permises sur ces valeurs.

Exemple de type :

Ensemble de valeurs : N

+

Ensemble d'opérations sur ces valeurs (+, -, *, /, =, :=, . . .)

Type de données Ada

Chaque type possède un **nom**.

En Ada, le type défini précédemment se nomme `Natural`.

On associe un objet à son type au moyen d'une **déclaration**.

Dans le programme précédent, la déclaration associe l'objet de nom `maNote` (qui est ici une variable) au type `Natural` :

```
maNote:Natural;
```

```
with Ada.Text_io;
with Ada.Integer_Text_io;
procedure exemple2 is
 subtype Note is Natural range 0..20;
 maNote:Note; car:Character;
begin
 Ada.Text_io.put_line( "debut du programme" );
 Ada.Text_io.put( "Tapez un caractère : " );
 Ada.Text_io.get(car);
 Ada.Text_io.put( "Le caractère tapé est : " );
 Ada.Text_io.put(car); Ada.Text_io.new_line;
-- lecture d'une note à partir du clavier
 Ada.Text_io.put( "Tapez une note : " );
 Ada.Integer_Text_io.get( maNote );
-- on ajoute un bonus
 maNote:=maNote+2; Ada.Text_io.new_line;
 Ada.Text_io.put( "maNote=" );
-- affichage de la nouvelle note sur l'écran
 Ada.Text_io.put( Note'image( maNote ));
 Ada.Text_io.new_line;
 Ada.Text_io.put( "fin du programme" );
end exemple2;
```

Explication (1/2)

Les 3 parties du programme apparaissent de nouveau :

- La première permet d'importer des objets (`get`, `put`, ...) destinés à permettre la lecture et l'écriture de valeurs textuelles.

- La seconde permet de déclarer 3 objets :
  - 2 variables (`maNote` et `car`) et
  - un nouveau type de données appelé `Note` dont l'ensemble des valeurs est le sous-ensemble des valeurs du type `Natural` limité à l'intervalle `0..20`. Les opérations possibles sur ce nouveau type sont les mêmes que celles du type `Natural`.

On notera l'apparition d'un autre type de données (`Character`) dont l'ensemble des valeurs est l'ensemble des caractères ASCII 7 bits (caractères usuels à l'exception des caractères accentués).

Explication (2/2)

La troisième partie est une suite d'instructions qui a pour but :

- de saisir à partir du clavier puis d'afficher sur l'écran un caractère unique compatible avec le type `Character`.
- de lire une valeur compatible avec le type `maNote`, de l'augmenter de `2` puis de l'afficher sur l'écran.

L'attribut image

```
Ada.Text_io.put(Note'image(maNote));
```

Il traduit la valeur entière à afficher (`maNote`) en une représentation sous la forme d'une chaîne de caractères.

En effet, on se rappelle que les objets importés (`put`, `get`) ne manipulent que des données textuelles.

```
Ada.Text_io.put( "maNote=" );
Ada.Integer_Text_io.put( maNote );
Ada.Text_io.new_line;
Ada.Text_io.put( "maNote=" );
Ada.Text_io.put( Note'image( maNote ));
```

résultat de l'affichage

```
maNote=          15
maNote= 15
```

L'unité **`Text_io`**

Elle regroupe un Ensemble d'objets qui permettent de réaliser des opérations d'Entrée/Sortie de caractères simples ou de chaînes de caractères.

```
with Ada.Text_io;
```

Cette clause importe le paquetage (sorte de boite à outils) `Text_io`.

L'ensemble des objets (outils) contenus dans ce paquetage deviennent alors utilisables dans le programme.

Outils de **`Text_io`**

Exemple d'outils disponibles dans `Text_io`

`get, put, put_line, new_line`

Préfixage par `Ada.Text_io`

Généralement, plusieurs boites à outils peuvent être importées.

Pour spécifier sans ambiguïté où est localisé l'outil voulu, il faut préfixé son nom par le nom de l'unité auquel il appartient.

```
with Ada.Text_io;
with Ada.Integer_Text_io;
procedure exemple3 is
 subtype Entier is Integer range -100..100;
 I:Entier:=0;car:Character;
begin
 Ada.Text_io.put_line( "debut du programme" );
 Ada.Text_io.put( "Tapez un caractère : " );
 Ada.Text_io.get(car);
 Ada.Text_io.put( "Le caractère tapé est : " );
 Ada.Text_io.put(car);
 Ada.Text_io.put( "Tapez un entier : " );
 Ada.Integer_Text_io.get(I);
 I:=I+10;
 AdA.Text_io.new_line;
 Ada.Text_io.put( "I=" );
 Ada.Integer_text_io.put(I);
 Ada.Text_io.new_line;
 Ada.Text_io.put( "fin du programme" );
end exemple3;
```

Explications

Le programme utilise les objets (procédures) de 2 bibliothèques (paquetages Ada), `Text_io` et `Integer_Text_io` pour réaliser des entrées et des sorties de données.

Le paquetage `Text_io` permet les lectures et les écritures de chaînes de caractères et de caractères.

Le paquetage `Integer_Text_io` permet les lectures et les écritures sur des valeurs entières.

On remarque aussi l'introduction d'un nouveau type de données (`Integer`) qui correspond à l'ensemble des entiers relatifs.

Lecture d'entiers : `get`

Lorsqu'un appel à la procédure `get` est effectué et que son paramètre est une variable d'un type entier (`get(I)`), le programme attend que l'opérateur tape un entier au clavier.

Les espaces et les caractères de fin de ligne ne sont pas pris en compte dans la saisie.

Affichage d'entiers : **put**

Lorsque la procédure `put` prend en paramètre une valeur entière, celle-ci est affichée sur l'écran selon un format standard (taille minimum capable d'accueillir toutes les valeurs entières possibles : 11 caractères pour une machine de 32 bits, 6 pour une machine de 16 bits).

```
put(-56);
```

affichera 8 espaces suivis de `-56`

```
        -56
```

Il est possible de spécifier le format d'affichage en ajoutant un paramètre supplémentaire

```
put(-56,3);
```

affichera la valeur `-56` sur 3 caractères, donc sans espaces préalables

```
-56
```

De même,

```
put(-56,4); -- ajout d'un espace devant
put(-56,1); -- l'affichage n'est pas tronqué
```

E/S sur les nombres flottants

```
with Ada.Text_io;
with Ada.Float_Text_io;
-- importation du paquetage Float_Text_io
procedure exemple4 is
 r:Float;
begin
 Ada.Text_io.put_line( "debut du programme" );
 Ada.Text_io.put( "Tapez un nombre réel : " );
 Ada.Float_Text_io.get(r);
 Ada.Text_io.put( "Le nombre saisi est : " );
 Ada.Float_Text_io.put(r);
 Ada.Text_io.new_line;
 Ada.Text_io.put( "fin du programme" );
end exemple4;
```

Lecture de réels : **get**

Lorsqu'un appel à la procédure `get` est effectué et que son paramètre est une variable d'un type réel (`get(r)`), le programme attend que l'opérateur tape un réel au clavier.

Les chiffres décimaux sont situés derrière un point (exemples : -234.45, 0.0067).

Les espaces et les caractères de fin de ligne ne sont pas pris en compte dans la saisie.

Affichage de réels : **put**

Par défaut, un réel est affiché selon le format suivant : `x.xxxxxE+xx` ou `x.xxxxxE-xx`. Soit 5 chiffres décimaux significatifs et un chiffre avant la virgule.

L'exposant est formé de la lettre `E` suivie du signe et des deux chiffres de l'exposant.

Il est possible de préciser son propre format en utilisant la procédure `put` avec 3 paramètres supplémentaires.

Le second paramètre indique le nombre de caractères avant le point (y compris des espaces si nécessaire car il n'y a toujours qu'un seul chiffre avant le point).

Le troisième paramètre indique le nombre de caractères après le point.

Le quatrième paramètre indique le nombre de caractères après E.

Exemple

```
put(0.023,3,2,1);
  2.30E-2

put(0.0234,1,2,0);
-- pour une notation décimale normale
0.02

put(1.01,5,8,4)
    1.00999999E+000
```

La clause **use**

Intérêt

1. Permet d'alléger l'écriture
2. D'intégrer chaque objet dans l'environnement du programme au lieu d'y accéder grâce au préfixe.

Exemple sans **use**

```
with Ada.Text_io;
with Ada.Integer_Text_io;

procedure exemple6 is
 subtype Entier is Integer range -100..100;
 I:Entier:=12;
begin
 Ada.Text_io.put( "I=" );
 Ada.Integer_Text_io.put(I);
 Ada.Text_io.new_line;
end exemple6;
```

Exemple avec **use**

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;
use Ada.Integer_Text_io;

procedure exemple5 is
 subtype Entier is Integer range -100..100;
 I:Entier:=12;
begin
 put( "I=" );put(I);new_line;
end exemple5;
```

Emploi multiple de la clause **use**

Comment peut-on (le compilateur peut-il) distinguer les outils de `Text_io ( put, put_line, get, ... )` et ceux de `Integer_Text_io ( put, ... )` ?

Réponse : par le type et le nombre des paramètres des procédures impliquées (`put` et `get`) (caractères ou chaînes de caractères dans le premier cas, type Entier dans l'autre).

```
with Ada.Text_io; use Ada.Text_io;
with Ada.Integer_Text_io; use Ada.Integer_Text_io;

procedure exemple7 is
 subtype Entier is Integer range -100..100;
 I:Entier;
 car:Character;
begin
-- c'est Text_io qui est choisi car le paramètre
-- est une chaîne de caractères
 put_line( "debut du programme" );
 put( "Tapez un caractère : " );
 get(car);
-- c'est Integer_Text_io qui est choisi car I est
-- du type Entier
 put(I);
 ....
 ....
end exemple7;
```

Production d'un programme Ada (1/4)

Un programme source Ada :

- est une suite de caractères (texte écrit dans le langage Ada)
- constitué de plusieurs unités nommées (exemple `Text_io`)
- une unité peut être une fonction, une procédure ou un module (paquetage)
- un programme Ada possède une unité principale (qui est une procédure)

Production d'un programme Ada (2/4)

Un programme Ada doit être compilé :

- Chaque unité est compilée séparément
- Le résultat de la compilation est conservé dans une bibliothèque
- En cas de modification du programme, seules les unités modifiées sont recompilées

Production d'un programme Ada (3/4)

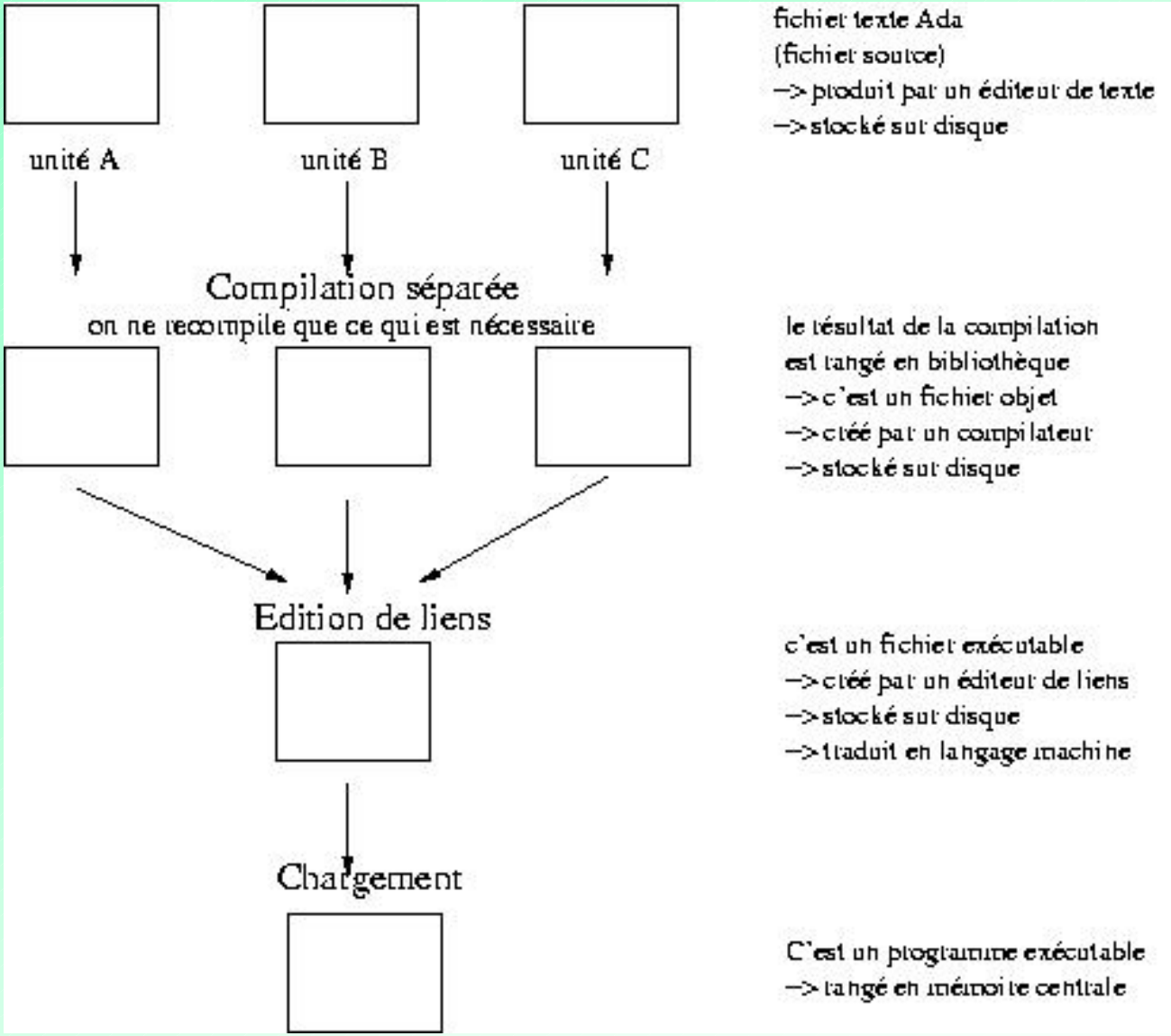
Un programme Ada doit être rendu exécutable :

- L'étape d'édition de liens rassemble les unités compilées en une seule entité traduite en langage machine.
- Des liens peuvent être établis avec des unités précompilées dans des bibliothèques prédéfinies.

Production d'un programme Ada (4/4)

Un programme Ada doit être chargé :

- En mémoire centrale pour être exécuté.



Chaîne de production de programme

Notion de grammaire formelle

Une *grammaire* est définie comme un quadruplet :

<A,VT,V,P>

A : point d'entrée

VT : vocabulaire terminal (+,-,0,1,_,...)

V : vocabulaire non terminal (ident, lettre,symbole,...)

P : règles de productions (élément de V::=suite de terminaux et non terminaux)

Le formalisme BNF étendu

symbole	signification	exemple
[]	partie facultative	[+]
|	alternative	+|-
…	intervalle	0...9
{}	répétition 0 ou n fois	{lettre}
$\{\}^+$	répétition stricte (1 à n fois)	$\{lettre\}^+$
()	mise en facteurs	(+|-)nombre

Analyse lexicale

Transformation de la suite de caractères en une suite de mots du langage

Les mots du langage sont :

- les identificateurs : `X_11,VAR,toto`
- les constantes numériques : `11, 42.21`
- les mots clés : **`procedure, begin, with, is, ...`**
- les opérateurs : `+, -, *, /, ...`

les espaces, les tabulations

Cette transformation est basée sur un ensemble de règles lexicales

Règles de construction d'un identificateur Ada (1/2)

```
<ident>  ::=<lettre>{<symbole>|_<symbole>}
<symbole>::=<lettre>|<chiffre>
<lettre> ::=A...Z|a...z
<chiffre>::=0...9
```

Exemples

```
X_1, A_, _A, 1_ADA, TINTIN_ET_MILOU, O__K, un$,
   A+B, OK
```

Règles de construction d'un identificateur Ada (2/2)

Identificateurs non valides : `A_, _A, 1_ADA, O__K, un$, A+B`

Identificateurs valides : `X_1, TINTIN_ET_MILOU, OK`

Analyse syntaxique

Transformation d'une suite de symboles en une suite de phrases du langage

Cette transformation est basée sur un ensemble de règles syntaxiques décrivant les structures syntaxiques du langage et exprimées dans le formalisme BNF (par exemple)

Pour éviter les ambiguïtés d'interprétation des phrases du langage, à une suite donnée de symboles ne peut correspondre qu'une seule phrase du langage

Structure syntaxique des expressions arithmétiques

Les éléments non terminaux de la grammaire sont notés entre < et >

<expression>::=<facteur>|<facteur><opadd><expression>
<facteur>::=<terme>|<terme><opmult><facteur>
<terme>::=<ident>|<nombre>|<expression>|(<expression>)
<opadd>::=+|-
<opmult>::=*|/
<nombre>::={<chiffre>}$^{+}$
<chiffre>::=0..9

Analyse syntaxique de l'expression : 7+3(34-6)*

Cette expression est une suite de symboles analysés un à un de gauche à droite.

<expression>::=<facteur>

puis

<facteur>::=<terme>

puis

<terme>::=<ident> ***Impossibilité***

<terme>::=<nombre> le symbole 7 est reconnu

donc la structure <terme> est reconnue

ainsi que les structures <facteur> puis <expression>

L'analyse échoue puisqu'elle se termine sans avoir reconnu la totalité des symboles constituant l'expression.

Analyse syntaxique de l'expression : 7+3*(34-6)

<expression>::=<facteur><opadd><expression>

- 7 est reconnu par <facteur> puis
- + par <opadd>

3*(34-6) devrait l'être par <expression>

Suite au backtracking, elle est reconnue par <terme><opmult><facteur>

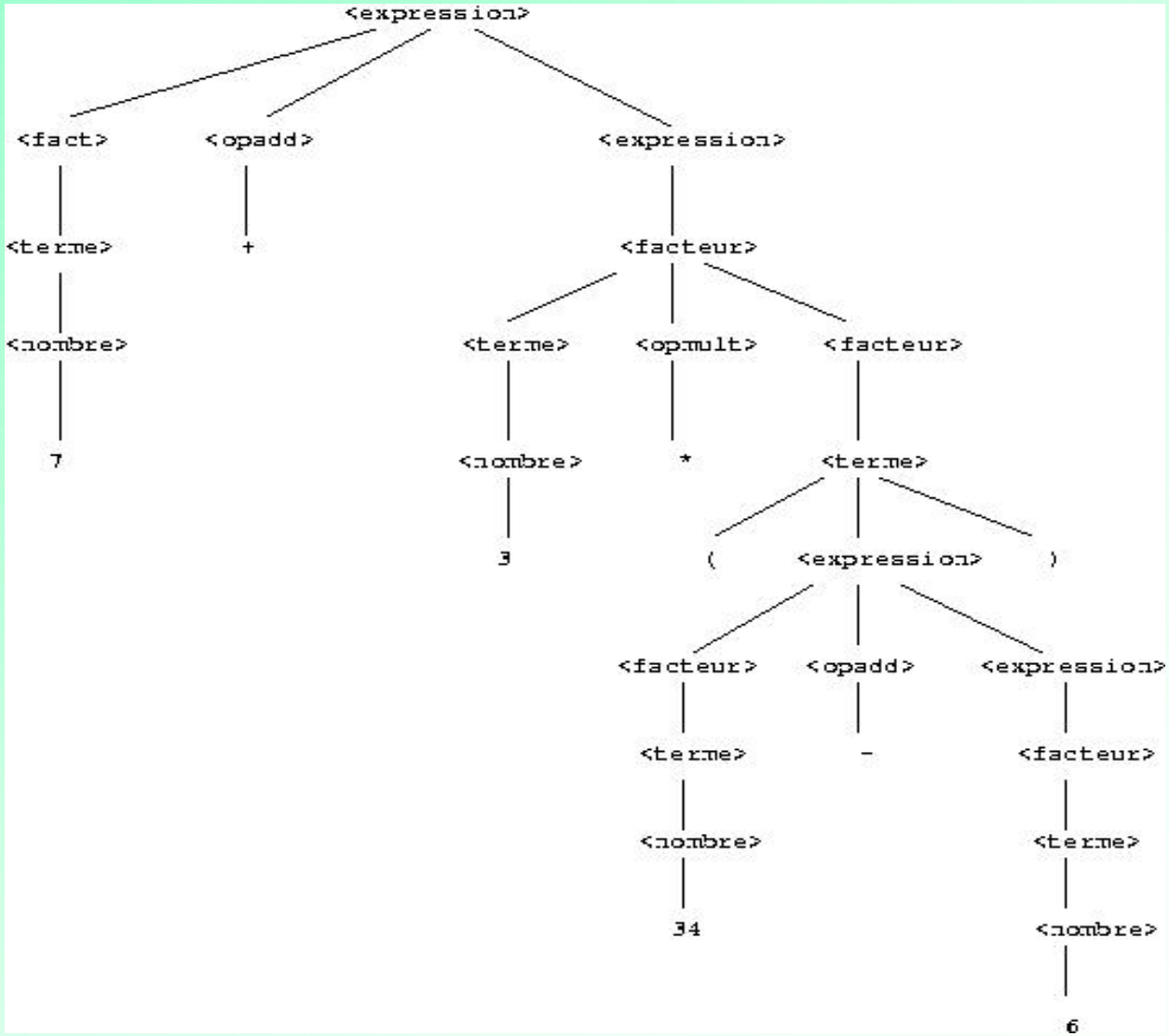
- 3 est reconnu par <terme>,
- * par <opmult>

Il reste (34-6).

- On voit que <facteur> se dérive en (<expression>) et que
- 34-6 correspond à la structure <facteur><opadd><expression>

7+3*(34-6) est donc une expression arithmétique bien formée.

Analyse syntaxique de l'expression : 7+3(34-6)*



Analyse sémantique

Vérifie que le type d'une construction syntaxique correspond au type attendu

Exemple

L'opérateur `mod` est une fonction qui, à 2 entiers, fait correspondre un entier. Elle nécessite donc 2 opérandes d'un type entier. Le contrôle de type vérifiera que c'est bien le cas. Elle vérifiera aussi que le résultat est lui-même du type entier.

Génération de code

A partir d'une représentation intermédiaire du code source produite par les phases antérieures de compilation et de la table des symboles, le générateur de code produit le code cible.

Ce code cible peut être exprimé :

- En langage machine translatable (les adresses des objets sont relatives). Ceci permet la compilation séparée de modules (unités)
- En code d'assemblage (les adresses et les instructions sont symboliques). Cette technique facilite la production de code mais nécessite une phase d'assemblage ultérieure

Edition de liens

- Réunit les différents modules dans un même espace d'adressage
- Permet l'utilisation, dans un programme, de modules déjà compilés

Texte source erroné

```
with Ada.Text_io;
with Ada.Integer_Text_io;

procedure exemple8 is
 subtype Entier is Integer range -100..100;
 I:Integer;
 car:Character:='X';
begin
 put( "Tapez un caractère : " );
 Ada.Integer_Text_io.get(I);
 Ada.Integer_Text_io.put(car);
end exemple8;
```

Résultat de la compilation, première erreur

```
9       put( "Tapez un caractère : " );
                  <1>
  1 **IDE No declaration having this name is visible at
this point - RM 8.3.
= More Information=======================
                -> Note the potentially visible item(s) :
ENUMERATION_IO.PUT at line 357 of TEXT_IO
Specification, procedure specification
FLOAT_IO.PUT at line 281 of TEXT_IO
Specification, procedure specification
INTEGER_IO.PUT at line 241 of TEXT_IO
Specification, procedure specification
        -> Direct visibility might be achieved by the
   appropriate use clause
```

Résultat de la compilation, seconde erreur

```
11      Ada.Integer_Text_io.put(car);
   1
  1 **IDE There is a type inconsistency in this expression
= More Information=======================
                -> Different interpretations exists for PUT :
-PUT at line 2 of Ada.Integer_Text_io
specification, its profile is
(STANDARD.STRING;STANDARD.INTEGER;TEXT_IO.NUMBER_BASE)
-PUT at line 2 of Ada.Integer_text_io
specification, its profile is
   (TEXT_IO.FILE_TYPE;STANDARD.INTEGER;TEXT_IO.FIELD;TEXT_IO.NUMBER_
   BASE)
etc ...
        -> But the expression has the following type
  - STANDARD.INTEGER
```

Type énumératif

```
with Ada.Text_io;use Ada.Text_io;

procedure exemple9 is
 type Jour is
   (lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
 type WE is (samedi,dimanche);
 unJour:Jour:= lundi;
 repos:WE   := dimanche;
begin
  if unJour = lundi then
    put( "1er jour de la semaine" );
    put( Jour'image(unJour) );
  elsif repos = WE'( dimanche ) then
       put( "c'est dimanche" );
     else
       put( "c'est samedi" );
  endif;
end exemple9;
```

Un nouveau programme erroné

```
with Ada.Text_io;use Ada.Text_io;
procedure exemple10 is
 type Jour is
  (lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
 type WE is (samedi,dimanche);
 unJour:Jour:= lundi;
 repos :WE  := mardi; -- erreur 1
begin
  if unJour = lundi then
    put( "1er jour de la semaine" );
    put( Jour'image( pred(lundi)) ); -- erreur 2
  elsif repos = WE'(dimanche) then
       put( "c'est dimanche" );
    else
       put( "c'est samedi" );
  endif;
end exemple10;
```

Explication

Erreur 1

Le compilateur détecte que la valeur `mardi` n'appartient pas au type `WE`. Une valeur ne peut appartenir qu'à un seul type. On parle de typage fort.

Erreur 2

`Jour'pred(lundi)` rend la valeur qui précède `lundi` dans le type `Jour`. Or cette valeur n'existe pas.

Ada envoie un signal `(CONSTRAINT_ERROR)` indiquant que l'on dépasse les bornes de l'intervalle `lundi..dimanche`.

Boucle for

La boucle `for` est une structure de contrôle itérative. Elle exprime la répétition d'une suite d'instructions.

Pour employer la boucle `for`, il faut connaître le nombre d'itérations souhaitées. C'est la raison pour laquelle, à la boucle `for`, est associé une un compteur. Dans les exemples suivants, les compteurs se nomment `i` ou `unJour`.

Les 2 instructions qui constituent le corps de la boucle seront répétées pour toutes les valeurs du type `Jour` comprises entre `vendredi` et `dimanche`.

Exemples

```
for i in vendredi..Jour'( dimanche ) loop
   Adda.Text_io.put( Jour'image(i) )
   Ada.Text_io.put( "fin de semaine" );
end loop;

for unJour in Jour loop
   Ada.Text_io.put( Jour'image(unJour) );
   Ada.Text_io.new_line;
end loop;

for i in WE loop
   Ada.Text_io.put( WE'image(i) );
   Ada.Integer_Text_io.put( WE'pos(i) );
   -- le rang de i dans la suite des valeurs de WE est
   affiché
end loop;
```

Commentaires

Pour des raisons de portabilité du programme, en l'occurrence d'indépendance du code par rapport au format des données, on écrira :

```
for unJour in Jour loop
```

plutôt que

```
for unJour in lundi..dimanche loop
```

De cette manière, il sera toujours possible de modifier les valeurs du type sans que le reste du programme en soit affecté. On pourrait, par exemple angliciser les valeurs du type `Jour (monday, tuesday, ...)` sans avoir à modifier la boucle.

La qualification est utilisée pour ne pas confondre la valeur dimanche du type `Jour` et celle du type `WE`.

```
vendredi..Jour'(dimanche)
```

Type tableau (1/2)

Pour pouvoir écrire des programmes un peu plus intéressants, il est nécessaire d'appliquer des traitements à des séquences de données. Une manière classique est de les rassembler dans un tableau.

Dans l'exemple qui suit, on réunit dans un tableau le nombre d'heures d'ensoleillement de chaque jour de la semaine.

4	8	2	6	9	3	1
lundi	mardi	mercredi	jeudi	vendredi	samedi	dimanche

Exemple de tableau d'heures d'ensoleillement hebdomadaire

Type tableau (2/2)

Un tableau est donc une structure de données qui réunit des valeurs (données) d'un même type (le type `Natural` dans l'exemple ci-dessus).

On peut le voir comme une suite de cases contiguës repérées (indicées) par une valeur d'un autre type (ici le type `Jour`).

Un tableau constitue une nouvelle valeur. Or toute valeur doit appartenir à un type. Il est donc nécessaire de définir un nouveau type auquel ces éléments (ces valeurs) appartiendront.

Par exemple, on définit le type `Soleil` comme un ensemble de tableaux contenant des entiers (du type `Natural`) indicés par des valeurs du type `Jour`.

Il est alors possible de déclarer une variable (`uneSemaine`) de ce nouveau type, d'enregistrer des valeurs dans les cases de ce tableau, de sélectionner une case connaissant son indice (`uneSemaine(i)`).

Exemple

Dans l'exemple qui suit, on saisit dans un tableau le nombre d'heures d'ensoleillement de chaque jour d'une semaine. On calcule ensuite quelle est la journée la plus ensoleillée puis on l'affiche à l'écran.

```
-- nombre d'heures d'ensoleillement journalier d'une semaine
donnée
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;

procedure exemple12 is
 type Jour is
(lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
 type Soleil is array(Jour) of Natural;
 uneSemaine:Soleil ;
 nbHeures  :Natural;
 unJour    :Jour;
begin
-- construction du tableau uneSemaine
 for i in Jour loop
   put( "taper le nb d'heures de soleil" );
   put( " pour la journee de " );
   put( Jour'image(i) );
   put( " : " );
   Ada.Integer_Text_io.get( uneSemaine(i) );
   new_line;
 end loop;
```

```
-- calcul du jour le plus ensoleille
 nbHeures:=uneSemaine(lundi);
 unJour  :=lundi;
 for x in mardi..dimanche loop
   if uneSemaine(x)>nbHeures
   then
     nbHeures:=uneSemaine(x);
     unJour  :=x;
   end if;
 end loop;

-- affichage du résultat
 put( "le jour le plus ensoleille de la semaine fut " );
 put( Jour'image(unJour) );
 put( " avec " );
 Ada.Integer_Text_io.put( uneSemaine(unJour),WIDTH=>2 );
 put_line( " heures d'ensoleillement" );
end exemple12;
```