4 Déclarations

4.1 Constantes

Une constante est désignée par un identificateur et une valeur, qui sont fixés en début de programme, entre les mots clés `CONST` et `VAR`.

La valeur **ne peut pas** être modifiée, et ne **peut pas** être une expression.

Syntaxe

```
identificateur = valeur_constante;
```

ou

```
identificateur : type = valeur_constante;
```

Dans la première forme, le type est sous-entendu (si il y a un point, c'est un réel, sinon un entier ; si il y a des quotes, c'est un caractère (un seul) ou une chaîne de caractères (plusieurs).

Exemple

```
PROGRAM constantes;
CONST
    faux = false;
    entier = 14;              { constantes NOMMEES }
    reel = 0.0;
    carac = 'z';
    chaine = 'hop';
    pourcent : real = 33.3;   { seconde forme avec type }
VAR
    { variables }
BEGIN
    { instructions }
END.
```

4.2 Variables et affectation

Une variable représente un objet d'un certain type ; cet objet est désigné par un identificateur. Toutes les variables doivent être *déclarées* après le `VAR`.

Syntaxe

`identificateur` : type ;

On peut déclarer plusieurs variables de même type en même temps, en les séparant par des virgules (voir exemple ci-dessous).

À la déclaration, les variables ont une valeur **indéterminée**. On initialise les variables juste après le `BEGIN` (on ne peut pas le faire dans la déclaration).

Utiliser la valeur d'une variable non initialisée est une erreur grave !

Exemple

```
VAR
    a, b, c : integer;
BEGIN
    { Partie initialisation }
```

```
    b := 5;
    { Partie principale }
    a := b + c;  { ERREUR, c n'est pas affecte' }
END.
```

L'opération `identificateur := expression;` est une *affectation*. On n'a pas le droit d'écrire `id1 := id2 := expr` , ni `expr := id` ni `expr1 := expr2` .

5 Expressions

Une *expression* désigne une *valeur*, exprimée par composition d'opérateurs appliqués à des *opérandes*, qui sont : des valeurs, des constantes, des variables, des appels à fonction ou des sous-expressions.

Exemple . Étant donné une variable `x`, une constante `max` et une fonction `cos()`, chaque ligne contient une expression :

```
5
x + 3.14
2 * cos(x)
(x < max) or (cos(x-1) > 2 * (x+1))
```

5.1 Syntaxe

Certains opérateurs agissent sur 2 opérandes :

```
operande1 operateur_binaire operande2
```

et d'autres agissent sur 1 opérande :

```
operateur_unaire operande
```

- Les opérateurs binaires sont :
  - opérateurs de relation `= <> <= < > >=`
  - opérateurs additifs `+ - or`
  - opérateurs multiplicatifs `* / div mod and`
- Les opérateurs unaires sont :
  - opérateurs de signe `+ -`
  - opérateur de négation `not`
- Les parenthèses sont un opérateur primaire, elles peuvent encadrer tout opérande.
- Une fonction est aussi un opérateur primaire, elle agit sur l'opérande placé entre parenthèses à sa droite. Certaines fonctions ont plusieurs paramètres, séparés par des virgules.

5.2 Type des expressions bien formées

Une expression doit être « bien formée » pour que l'on puisse trouver sa valeur. Par exemple, `3 * 'a' - true` n'est pas bien formée, et la compilation Pascal échouera.

Dans la partie 3, *Types prédéfinis*, on a déjà dit quels opérateurs sont applicables sur quels types. Mais il y a encore d'autres règles, dont le simple bon-sens !

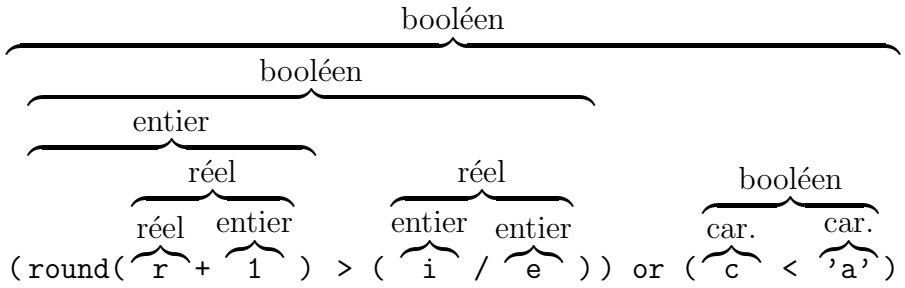
L'expression bien formée a un type, qui dépend des règles d'évaluation de l'expression.

Exemple

Soit `r` un réel, `i` un entier, `e` une constante entière, `c` un caractère. L'expression

```
(round(r+1) > (i/e)) or (c < 'a')
```

est bien formée, et sont type est booléen comme on le montre ici :



Remarque

Le fait qu'une expression est bien formée n'implique pas que son évaluation est sans erreur, ce qui peut être le cas ici si `e` est nul.

5.3 Règles d'évaluation

L'expression `a + b * c` est évaluée `a + (b * c)` et non pas `(a + b) * c` : ceci parce que le `*` est *prioritaire* par rapport à `+`.

On classe les différents opérateurs par *ordre de priorité*, les opérateurs de plus forte priorité étant réalisés *avant* ceux de plus faible priorité.

Lorsque deux opérateurs sont de priorité égale, on évalue de gauche à droite. Par exemple `a + b - c` est évalué `(a + b) - c`, et non pas `a + (b - c)`.

Voici la table des priorités classées par ordre décroissant, les opérateurs sur une même ligne ayant une priorité égale.

`() fonction()`	primaire
`+ - not`	unaire
`* / div mod and`	multiplicatif
`+ - or`	additif
`= <> < <= >= >`	relation

Remarque

Est-ce que l'expression `a < b or c <= d` est bien formée ? Quel est son type ?

Réponse : non ! Ecrire une telle expression booléenne sans parenthèses est une erreur classique.

En effet dans la table de priorités, l'opérateur `or` a une priorité plus élevée que les opérateurs `<` et `<=`, et donc l'expression sera évaluée `a < (b or c) <= d` , ce qui est faux.

L'expression bien formée est ici `(a < b) or (c <= d)` .

6 Nouveaux types

On a vu les types pré-déclarés `boolean`, `integer`, `real` et `char`.

Nous verrons par la suite comment créer de nouveau types. Nous commençons par les plus simples, le type intervalle et le type énuméré.

6.1 Type intervalle

C'est un sous-ensemble de valeurs *consécutives* d'un type hôte.

Syntaxe `N..M`

où `N` et `M` sont des constantes du même type, et sont les bornes inférieures et supérieures de l'intervalle, `N` et `M` inclus.

Exemple

```
VAR
    pourcentage : 0 .. 100;        { le type hote est integer }
    digit       : '0' .. '9';      { le type hote est char    }
    reponse     : false .. true;   { le type hote est boolean }
```

Remarques

- ▷ Il faut impérativement que le type hôte soit codé sur *un entier* (signé ou non, sur un nombre de bits quelconque). On dit alors que ce type hôte est un *type ordinal*.
- ▷ Ainsi les types `integer`, `char` et `boolean` sont des types ordinaux.
- ▷ Seul un type ordinal admet les opérateurs `pred`, `succ` et `ord` (le précédent, le successeur et le numéro d'ordre dans le codage).
- ▷ Par contre le type `real` n'est pas ordinal, et donc on ne pas créer un type intervalle avec des réels, il n'y a pas de notion de « réels consécutifs ».
- ▷ Un autre cas de type non ordinal est le type `string` pour les chaînes de caractères, qui n'est pas codé sur *un* mais sur *plusieurs* entiers. On ne peut donc pas déclarer `'aaa'..'zzz'`.

Bonne habitude Utiliser des constantes nommées pour borner les intervalles : de la sorte on pourra consulter ces valeurs pendant le programme, et ces bornes ne seront écrites qu'une seule fois.

Exemple

```
CONST
    PMin = 0;
    PMax = 100;
VAR
    pourcentage : PMin .. PMax;
BEGIN
    writeln ('L''intervalle est ', PMin, ' .. ', PMax);
END.
```

6.2 Type énuméré

Il est fréquent en programmation que l'on aie à distinguer plusieurs cas, et que l'on cherche à coder le cas à l'aide d'une variable.

Exemple

```
VAR
    feux : 0..3;   { rouge, orange, vert, clignotant }
BEGIN
    { ... }
    if feux = 0
    then Arreter
    else if feux = 1
    then Ralentir
    else if feux = 2
    { ... }
END.
```

Ceci est très pratique mais dans un programme un peu long cela devient rapidement difficile à comprendre, car il faut se souvenir de la signification du code.

D'où l'intérêt d'utiliser un type *énuméré*, qui permet de donner un nom aux valeurs de code :

```
VAR
    feux : (Rouge, Orange, Vert, Clignotant);
BEGIN
    { ... }
    if feux = Rouge
    then Arreter
    else if feux = Orange
    then Ralentir
    else if feux = Vert
    { ... }
END.
```

- En écrivant cette ligne, on déclare en même temps :
  – la variable `feux`, de type énuméré (toujours codée sur un entier),
  – et les constantes nommées `Rouge`, `Orange`, `Vert` et `Clignotant`.

• À ces constantes sont attribuées les valeurs 0, 1, 2, 3 (la première constante prend toujours la valeur 0).

– On ne peut pas choisir ces valeurs soi-même, et ces identificateurs ne doivent pas déjà exister.

– L'intérêt n'est pas de connaître ces valeurs, mais d'avoir des noms explicites.

• Le type énuméré étant codé sur un entier, il s'agit d'un type ordinal et on peut :

– utiliser les opérateurs `pred`, `succ` et `ord` (exemple : `pred(Orange)` est `Rouge`, `succ(Orange)` est `Vert`, `ord(Orange)` est 1).

– déclarer un type intervalle à partir d'un type énuméré (exemple : `Rouge..Vert`).

6.3 Déclarer un type

Créer un type, c'est bien, mais le nommer, c'est mieux. On déclare les noms de types entre les mots clés `TYPE` et `VAR`.

Syntaxe

```
nom_du_type = type;
```

Exemple

```
TYPE
    couleurs_feux_t = (Rouge, Orange, Vert, Clignotant);
VAR
    feux : couleurs_feux_t;
```

De la sorte `couleurs_feux_t` est un nom de type au même titre que `integer` ou `char`.

Exemple complet

```
PROGRAM portrait;
CONST
    TailleMin = 50;   { en cm }
    TailleMax = 250;
TYPE
    taille_t   = TailleMin .. TailleMax;
    couleurs_t = (Blond, Brun, Roux, Bleu, Marron, Noir, Vert);
    cheveux_t  = Blond .. Roux;
    yeux_t     = Bleu .. Vert;
VAR
    taille_bob, taille_luc   : taille_t;
    cheveux_bob, cheveux_luc : cheveux_t;
    yeux_bob, yeux_luc       : yeux_t;
BEGIN
    taille_bob  := 180;
    cheveux_bob := Brun;
    yeux_bob    := Noir;
END.
```

Remarque Observez bien les conventions d'écriture différentes que j'ai employées pour distinguer les constantes des types et des variables ; cela aussi aide à la lecture.

6.4 Type enregistrement

Il s'agit simplement de regrouper des variables `V1`, `V2`, ... de différents types `T1`, `T2`, ... dans une variable « à tiroirs ».

Syntaxe

```
Record
    V1 : T1;
    V2 : T2;
    { ... }
End;
```

Soit `r` une variable de ce type ; on accède aux différents *champs* de `r` par `r.V1`, `r.V2`, ...

Reprenons l'exemple du programme `portrait`.

```
{ ... }
TYPE
    { ... }
    personne_t = Record
                   taille  : taille_t;
                   cheveux : cheveux_t;
                   yeux    : yeux_t;
                 End;
VAR
    bob, luc : personne_t;
BEGIN
    bob.taille  := 180;
    bob.cheveux := Brun;
    bob.yeux    := Noir;
    luc := bob;
END.
```

Remarque La seule opération globale sur un enregistrement est : recopier le contenu de `r2` dans `r1` en écrivant : `r2 := r1;`

Ceci est équivalent (et plus efficace) que de copier champ à champ ; en plus on ne risque pas d'oublier un champ.

Il y a une condition : les 2 variables doivent être exactement du même type.

Intérêt de ce type

Il permet de structurer très proprement des informations qui vont ensemble, de les recopier facilement et de les passer en paramètres à des procédures (on y reviendra).

Remarque générale

Lorsqu'on crée un type `T2` à partir d'un type `T1`, ce type `T1` doit *déjà exister* ; donc `T1` doit être déclaré *avant* `T2`.