Pascal

Langages d'écriture de systèmes

par **Olivier LECARME**
Docteur ès Sciences
Professeur à l'Université de Nice

*Le langage de programmation **Pascal** a été conçu en 1969 par Niklaus Wirth, professeur à l'École polytechnique fédérale de Zurich. Son nom a été choisi en hommage au mathématicien-philosophe Blaise Pascal (1623-1662). Les deux objectifs principaux annoncés par l'auteur du langage [1] sont les suivants :*

— « rendre disponible un langage qui permette d'enseigner la programmation comme une discipline systématique, fondée sur certains concepts fondamentaux reflétés de façon claire et naturelle par le langage ;

— construire des implantations de ce langage qui soient à la fois fiables et efficaces sur les ordinateurs actuellement disponibles ».

Ces deux objectifs, d'ailleurs difficiles à atteindre, ne laissaient pas prévoir le succès qu'a connu le langage en dehors du domaine restreint pour lequel il avait été initialement conçu, c'est-à-dire l'enseignement de la programmation. Le langage Pascal est en effet devenu en moins de dix ans un des langages de programmation dominants, sans avoir bénéficié d'aucun support officiel de la part des constructeurs d'ordinateurs ou des organismes nationaux et internationaux. Il est actuellement utilisé aussi bien dans l'industrie que dans les universités, et sert au développement de logiciel fiable, efficace et transportable dans des domaines d'application très variés.

Les qualités majeures qui ont permis cette diffusion sont qu'il s'agit d'un langage concis, pratique et général (mais non universel), qui fournit des structures d'énoncés et de données facilitant et encourageant la programmation systématique. Ses utilisateurs peuvent l'apprendre et le manier facilement, et il peut être traduit en programmes efficaces sur les ordinateurs actuels.

Raisons d'être du langage

1. Présentation du langage

1.1 Deux exemples de programmes

Le langage Pascal pouvant être utilisé dans une gamme assez variée d'applications, il est difficile d'en donner une idée suffisante en un seul exemple. Les deux programmes présentés ici concernent donc deux domaines d'application bien différents. De même que la description du langage qui apparaît dans la suite de cet article, ces deux exemples utilisent la forme française des mots clés du langage proposée par le sous-groupe Pascal de l'AFCET. Ils utilisent également la **forme de présentation** du langage, où l'on ne tient pas compte des jeux de caractères restreints imposés par la plupart des matériels informatiques actuels. Cette forme est bien adaptée à la lecture et à l'écriture par des êtres humains, et la transcodification sous la forme acceptée par la plupart des compilateurs du langage se fait très simplement. À titre d'exemple, la figure 1 montre la **forme d'utilisation** des quatre dernières lignes du programme de la figure 2, pour un compilateur qui n'accepte que le sous-ensemble à 64 caractères du jeu ISO (ce sous-ensemble ne contient pas de lettres minuscules).

```
POUR C: = 'A' HAUT 'Z' FAIRE
   SI FREQ[C] < > 0 ALORS ECRIRELN (SORTIE, C: 5, FREQ[C]: 8)
      (* LES LETTRES ABSENTES DANS LE TEXTE N'APPARAISSENT
      PAS DANS LA TABLE *)
FIN (* DU PROGRAMME *).
```

Figure 1 – Forme d'utilisation d'un programme

```
 1 programme fréquence (entrée, sortie);
 2   { Ce programme recopie le texte fourni en données, et établit une table
 3     de fréquence d'apparition des lettres majuscules }
 4   var c: car;
 5     freq: tableau ['A'..'Z'] de entier; { table de fréquence }
 6     lettres: ensemble de 'A'..'Z';
 7 début lettres: = ['A'..'Z']; { on suppose le jeu de caractères ISO }
 8   pour c: = 'A' haut 'Z' faire freq [c]: = 0; { initialisation }
 9   tantque ¬ fdf (entrée) faire { lecture du fichier des données }
10   début écrire (sortie, ' '); { un blanc en début de ligne }
11     tantque ¬ fdln (entrée) faire { lecture d'une ligne de texte }
12     début lire (entrée, c); écrire (sortie, c); { recopie du caractère lu }
13       si c dans lettres alors freq [c]: = freq [c] + 1
14     fin { d'une ligne };
15     écrireln (sortie); lireln (entrée) { passer à la ligne sur les deux fichiers }
16   fin { du texte à lire };
17   page (sortie); écrireln (sortie, 'Table de fréquence des lettres ');
18   écrireln (sortie);
19   pour c: = 'A' haut 'Z' faire
20     si freq [c] ≠ 0 alors écrireln (sortie, c: 5, freq [c]: 8)
21     { les lettres absentes dans le texte n'apparaissent pas dans la table }
22 fin { du programme }.

          La numérotation des lignes ne fait pas partie du programme
```

Figure 2 – Exemple de programme

Historique

Le langage qui a eu le plus d'influence sur Pascal est certainement Algol 60. N. Wirth faisait partie du comité international qui commença à travailler vers 1963 pour définir un successeur à Algol 60. Ce successeur devait étendre le domaine d'application du langage, simplifier certains de ses aspects et en améliorer d'autres. La contribution proposée par N. Wirth et C.A.R. Hoare fut refusée par le comité et publiée indépendamment sous le nom d'Algol W [2] en 1966. Le langage Algol W constitue donc une étape intermédiaire entre Algol 60 et Pascal. Quant au successeur officiel d'Algol 60, il s'agit d'Algol 68 (cf. article spécialisé dans le présent traité).

La première définition de Pascal date de 1969 et s'est accompagnée de la réalisation du premier compilateur (sur ordinateur *Control Data 6 400*), de façon à montrer que le second objectif (cf. introduction) pouvait bien être atteint. Ce premier compilateur présentait la particularité d'être lui-même écrit en Pascal, ce qui était un excellent moyen d'assurer sa concision et sa lisibilité. La très grande majorité des compilateurs réalisés depuis sur d'autres ordinateurs sont également écrits en Pascal, et c'est un des facteurs qui ont le plus contribué à la diffusion du langage, car le coût initial d'une nouvelle implantation est très réduit.

La définition du langage Pascal a été publiée officiellement en 1971 [1], et le compilateur a commencé d'être diffusé à la même époque. Un livre d'introduction à la programmation s'appuyant sur le langage a été publié d'abord en allemand puis en anglais [3]. L'expérience accumulée pendant les trois premières années d'utilisation a permis de réviser la définition du langage, sur un nombre de points d'ailleurs très réduit, et c'est cette définition, accompagnée d'un manuel d'utilisation, qui sert actuellement de référence [4]. Une définition axiomatique de la plus grande partie de la sémantique du langage a été publiée à la même époque [5].

Depuis 1979, les implémentations de Pascal se sont multipliées à tel point que l'on peut garantir leur existence sur tous les ordinateurs actuels, depuis les plus petits microprocesseurs jusqu'aux ordinateurs géants, le plus souvent en prenant pour modèle les compilateurs du CDC 6400. Une étude récente répertorie plus de quarante implémentations importantes et répandues, dues pour la plupart à de grands fabricants de logiciel. Sur de nombreux ordinateurs, il existe même plusieurs implémentations en concurrence, ce qui ne va pas sans poser de difficiles problèmes de compatibilité. C'est dire l'importance des travaux de normalisation internationale décrits au paragraphe 5.

Par ailleurs, les livres d'apprentissage ou d'approfondissement de la programmation qui utilisent Pascal comme support se sont également multipliés ([6] à [11] [14] à [22], par exemple).

La distinction entre forme de présentation et forme d'utilisation n'est pas faite explicitement dans la norme internationale (§ 5). Celle-ci prévoit cependant qu'il puisse exister plusieurs représentations pour les différents symboles, et ne normalise que celle qui doit servir à l'échange de programmes entre différentes implantations. En particulier, les formes possibles pour une lettre donnée (majuscule, minuscule, gras, maigre, romain, italique, souligné, etc.) n'ont pas de signification, sauf à l'intérieur des chaînes de caractères. La forme française des mots-clés et des indentificateurs prédéfinis apparaît en annexe de la norme française. Elle présente de très légères différences avec celle qui est utilisée dans le corps du présent article.

1.1.1 Table de fréquences

Le programme de la figure **2** a pour données un texte quelconque, c'est-à-dire une suite de lignes qui sont elles-mêmes des suites de caractères. Il recopie ce texte et construit en même temps une table de la fréquence d'apparition de chaque lettre majuscule. Cette table est écrite après la recopie du texte. On peut remarquer en particulier l'utilisation d'un caractère comme indice d'un tableau (lignes 5, 8, 13 et 20), celle de la variable lettres qui représente l'ensemble des lettres majuscules (lignes 6, 7 et 13), les prédicats fdf (fin de fichier, ligne 9) et fdln (fin de ligne, ligne 11) qui permettent de reconnaître la structure du texte lu.

1.1.2 Mise à jour de fichier

Le programme de la figure **3** a pour données un fichier d'articles qui représentent une série de modèles d'automobiles, identifiés par une clé numérique, et un fichier de modifications à effectuer. Il fournit comme résultats un fichier mis à jour, et éventuellement quelques messages d'erreurs. Les modifications possibles sont des adjonctions, des suppressions et des remplacements d'articles. On peut remarquer en particulier l'accès à l'article lu sur un fichier (ancien ↑, lignes 18 et 27 ; commandes ↑, lignes 21 et 29), la manipulation des articles de façon globale (lignes 16, 18, 27, 38 et 41) ou champ par champ (lignes 23, 31 et 34), l'énoncé de choix (lignes 36 à 43).

1.2 Caractéristiques du langage

Pour clarifier les idées d'un lecteur familier avec d'autres langages de programmation, ce paragraphe effectue une comparaison rapide de Pascal avec Fortran, Algol 60 et PL/I (cf articles spécialisés dans le présent traité). Bien que Pascal permette de traiter des applications pour lesquelles on utilise d'ordinaire Cobol (§ 1.1.2), son apparence extérieure en est trop différente pour qu'une comparaison soit utile. Par rapport aux trois langages ci-dessus, les caractéristiques remarquables de Pascal sont donc les suivantes :

— les **variables** doivent obligatoirement être déclarées (comme en Algol 60, contrairement à Fortran et PL/I) ;

— les **mots-clés**, tels que **début**, **tantque**, **alors**, sont réservés à cet usage et ne peuvent pas servir d'identificateurs ;

— le **point-virgule** est un séparateur d'énoncés (comme en Algol 60), et non pas un terminateur (comme en PL/I) ;

— les **types** attribuables aux objets comprennent les nombres entiers et réels, les valeurs booléennes et les caractères imprimables ; ils peuvent être structurés en **tableaux**, **articles** (appelés **structures** en PL/I et **enregistrements** en Cobol), **ensembles** et **fichiers** (séquentiels) ; ces moyens de structuration peuvent être combinés pour fournir des tableaux d'ensembles, des fichiers d'enregistrements, etc. ; certains objets peuvent être alloués de façon dynamique et repérés par des **pointeurs** ; comme ils peuvent eux-mêmes contenir des pointeurs, cela permet de construire des structures arborescentes quelconques ;

— la structure d'**ensemble** fournit des possibilités similaires à celles de la chaîne binaire de PL/I ;

— les **tableaux** sont de dimensions et de bornes quelconques (comme en Algol 60 ou PL/I), mais ces bornes sont définies à la compilation (comme en Fortran) ;

— l'énoncé composé, identique à celui d'Algol 60, correspond au groupe DO sans répétition de PL/I ;

— les possibilités offertes par l'aiguillage d'Algol 60, le branchement calculé de Fortran ou les tableaux d'étiquettes de PL/I sont fournies en Pascal par l'énoncé **cas** ;

— l'énoncé **pour** correspond à la boucle DO de Fortran, mais le pas ne peut être que 1 ou – 1, et l'énoncé peut ne pas être exécuté du tout si la valeur limite est dépassée dès le début ;

— l'expression conditionnelle d'Algol 60 n'existe pas, non plus que l'énoncé d'affectation multiple d'Algol 60 et de PL/I ;

— les **procédures** et **fonctions** peuvent être appelées récursivement, comme en Algol 60 et en PL/I ;

```
 1 programme Mise à jour (ancien, nouveau, commandes, messages);
 2   { Mise à jour du fichier ancien en fichier nouveau grâce au
 3     fichier de commandes }
 4   const longdesc = 30 { nombre de caractères d'une zonealpha };
 5   type zonealpha = paquet tableau [1..longdesc] de car;
 6     description = article clé: 1..99999;
 7                             modèle, couleur: zonealpha;
 8                             année: 0..99
 9                   fin;
10   var ancien, nouveau: fichier de description;
11     messages: texte;
12     commandes: fichier de article typecomm: car;
13                                    art: description fin;
14     courant: description; trouvé, copie: booléen;
15   procédure copier; { avancer d'un article sur ancien et nouveau }
16   début si copie alors écrire (nouveau, courant);
17       copie: = vrai; prendre (ancien);
18       si ¬ fdf (ancien) alors courant: = ancien ↑
19   fin;
20 procédure erreur (i: entier); { écriture d'un message d'erreur }
21   début avec commandes ↑.art faire
22       écrireln (messages, 'Erreur', i: 1, 'sur la commande'
23           clé: 5, modèle: longdesc + 1, couleur: longdesc + 1, année: 3)
24   fin;
25 début { du programme principal }
26   relire (ancien); relire (commandes); récrire (nouveau); récrire (messages);
27   courant: = ancien ↑; copie: = vrai;
28   tantque ¬ fdf (commandes) faire
29   avec commandes ↑ faire { traitement d'une commande }
30   début { recopier le fichier ancien jusqu'à la clé voulue }
31       tantque ¬ fdf (ancien) ∧ (art.clé > courant.clé) faire
32       début copier;
33           si fdf (ancien) alors trouvé: = faux
34           sinon trouvé: = (art.clé = courant.clé)
35       fin; { clé atteinte ou dépassée }
36       cas typecomm de
37       'A': si trouvé alors erreur (1) { ajouter un article existant }
38           sinon début courant: = art; copie: = vrai fin;
39       'S': si trouvé alors copie: = faux
40           sinon erreur (2) { supprimer un article absent };
41       'M': si trouvé alors début courant: = art; copie: = vrai fin
42           sinon erreur (3) { modifier un article absent }
43 autrement erreur (4) { commande inconnue } fin;
44       prendre (commandes) { passer à la commande suivante }
45   fin { du traitement d'une commande };
46   tantque ¬ fdf (ancien) faire copier
47 fin { du programme }.
```

La numérotation des lignes ne fait pas partie du programme

Figure 3 – Exemple de programme de mise à jour de fichier

— les paramètres sont transmis par **valeur** (comme en Algol 60) ou par **référence** (comme en Fortran) ;

— Pascal fournit une **structure de blocs** similaire à celle d'Algol 60 et de PL/I, mais tous les blocs sont des procédures.

Un nombre assez important de possibilités ou de dispositifs trouvés dans d'autres langages de programmation n'apparaissent pas en Pascal, soit qu'ils aient été considérés comme un encouragement à une mauvaise programmation, soit qu'ils contredisent les objectifs de clarté et de simplicité du langage, soit que les possibilités existantes permettent de construire facilement celles-là en cas de besoin. Il s'agit par exemple de l'opération d'élévation à une puissance, de la concaténation des chaînes, des tableaux à bornes dynamiques, des conversions de type automatiques, des déclarations implicites, des options par défaut, des nombres complexes, etc. C'est grâce à cette sélection rigoureuse effectuée parmi l'éventail des possibilités que les compilateurs du langage Pascal restent des programmes efficaces et relativement simples.

1.3 Structure des programmes

Un **programme** est formé d'un **en-tête** et d'un **bloc**, lequel comprend un ensemble de **déclarations** et de **définitions**, suivi d'un groupe d'**énoncés** qui constituent le corps du programme. L'en-tête sert à fournir le nom du programme et à énumérer ses paramètres, qui sont en général les noms des fichiers qui permettent au programme de communiquer avec son environnement. L'ensemble de déclarations et de définitions comprend la déclaration des **étiquettes** utilisées dans le corps du programme, la définition des **constantes**, celles des **types**, la déclaration des **variables**, celle des **procédures** et des **fonctions**. Toutes ces parties sont facultatives ; mais tout identificateur utilisé doit avoir été déclaré ou défini au préalable (sauf certains identificateurs prédéfinis, que l'on peut supposer déclarés ou définis dans l'environnement du programme : § 2.1 ; § 4.2).

Les énoncés d'un bloc peuvent être étiquetés, mais toutes les étiquettes utilisées doivent être déclarées en tête du bloc ; ce sont des nombres entiers positifs. Leur usage est de toute façon extrêmement restreint en Pascal.

Les définitions de constantes permettent de nommer ces constantes, ce qui facilite la lecture des programmes, et l'adaptation rapide d'un programme à un changement d'environnement, par simple modification de la valeur de certaines constantes (figure 3, ligne 4).

Tous les objets manipulés dans un programme possèdent un **type**, qui est associé aux variables qui permettent de les manipuler. Les types utilisés peuvent être directement décrits au moment de la déclaration des variables (figure 2), ou repérés par un identificateur

défini au préalable (figure 3, lignes 5 à 9). Les descriptions de type elles-mêmes seront vues au paragraphe 2.

Toute variable utilisée dans un programme doit être déclarée au préalable. Cette déclaration associe à chaque variable un type, décrit explicitement ou par référence à un identificateur de type.

D'une façon générale, un identificateur est accessible dans tout le bloc en tête duquel il est déclaré. Cependant, si dans un bloc emboîté dans celui-là, par le moyen d'une définition de procédure ou de fonction, un identificateur homonyme est déclaré, c'est ce dernier identificateur qui est accessible dans ce bloc emboîté. Cette règle est identique à celle que l'on trouve dans les langages dits à **structure de bloc**, par exemple Algol 60 ou PL/I.

2. Objets et opérateurs

Un programme est constitué d'une suite d'**actions**, exprimées par des **énoncés**, qui manipulent des **objets**. Pour classer ces objets en fonction des actions auxquelles il est possible de les soumettre, on associe un **type** aux variables qui permettent d'accéder aux objets. Le type d'une variable définit donc à la fois l'ensemble des **valeurs** qu'elle peut prendre et la nature des **opérations** qui peuvent lui être appliquées. Cela permet de vérifier automatiquement, et en général avant l'exécution du programme, la légalité des opérations contenues dans les énoncés de celui-ci.

Certains types sont **prédéfinis** par le langage, soit à cause de leur importance, soit parce qu'il n'est pas matériellement possible de les définir dans le programme. Quelle que soit leur complexité, tous les types sont finalement définis à partir de types non structurés. Nous étudierons donc successivement les **types simples** fournis par Pascal, puis les outils qu'il contient pour la construction de **types structurés**.

2.1 Types simples

2.1.1 Types scalaires prédéfinis

Le langage Pascal fournit trois types scalaires prédéfinis, et permet à l'utilisateur d'en définir de nouveaux.

■ Le type prédéfini booléen comprend les deux constantes prédéfinies faux et vrai. Les opérateurs possibles sont l'union logique (V), l'intersection logique (∧) et la négation (¬). De plus, les opérateurs de comparaison et certains prédicats prédéfinis (§ 4.2) fournissent un résultat booléen.

■ Le type prédéfini entier comprend un intervalle des nombres entiers dont l'étendue dépend de l'ordinateur utilisé. La constante prédéfinie entmax donne la limite supérieure de cet intervalle. Les opérateurs sur les entiers sont l'addition (+), la soustraction (–), la multiplication (*), la division entière (div) et le reste de la division (mod). On peut de plus comparer des entiers (<, ⩽, =, ≠, ⩾ et >) et calculer leur valeur absolue (fonction prédéfinie abs) et leur carré (fonction prédéfinie carré). Enfin, le prédicat impair s'applique à un entier.

■ Le type prédéfini car représente l'ensemble des caractères disponibles sur l'ordinateur utilisé. Une constante de ce type est dénotée entre apostrophes (on représente le caractère apostrophe par une double apostrophe). La fonction prédéfinie ord associe un entier à un caractère donné, son nombre ordinal ; la fonction prédéfinie carac, étant donné le nombre ordinal d'un caractère, fournit le caractère lui-même. Cela permet de définir un ordre sur les caractères et de leur appliquer les opérateurs de comparaison ordinaires. De plus, les fonctions prédéfinies succ et pred fournissent respectivement le successeur et le prédecesseur d'un caractère donné (s'il existe).

2.1.2 Définition de types scalaires

Le premier moyen de définition d'un type scalaire consiste à énumérer les constantes qui le constituent. Par exemple, le type prédéfini booléen pourrait être défini par :

type booléen = (faux, vrai)

Les fonctions prédéfinies succ, pred et ord (§ 4.2) sont applicables à tout objet de type scalaire ainsi que les opérateurs de comparaison (qui utilisent l'ordre implicite donné par l'énumération).

Le deuxième moyen de définition consiste à donner un intervalle dans un type scalaire déjà défini (le plus souvent entier ou car). Cela permet de préciser le domaine exact des valeurs que peut prendre une variable, et peut permettre au compilateur de produire un code mieux adapté à l'ordinateur.

Exemples

```
type jour = (lun, mar, mer, jeu, ven, sam, dim);
     travail = lun..ven;
     lettre = 'a'..'z';
```

On peut appliquer à un type intervalle les mêmes opérateurs que ceux qui s'appliquent au type de base dont il est issu.

2.1.3 Type réel

Le type prédéfini réel représente un sous-ensemble, dépendant de l'ordinateur utilisé, des nombres réels. On peut appliquer à une valeur réelle les opérateurs arithmétiques d'addition (+), soustraction (–), multiplication (*) et division (/), ainsi que les opérateurs de comparaison ordinaires. On peut d'autre part lui appliquer les fonctions prédéfinies abs (valeur absolue), carré, sin, cos, arctan, ln (logarithme népérien), exp (exponentielle) et rac2 (racine carrée).

Partout où l'on a besoin d'une valeur réelle, le langage Pascal permet de fournir une valeur entière, le transfert de type ayant lieu automatiquement. L'inverse n'est pas vrai et, pour passer d'une valeur réelle à une valeur entière, il faut utiliser l'une des deux fonctions prédéfinies, à paramètre réel et résultat entier, tronc (suppression de la partie fractionnaire) ou arrondi.

2.2 Types structurés

Un objet d'un type structuré est un groupe d'objets que l'on considère comme un tout. Un type structuré est obtenu par composition des **types des composants**, grâce à un mode de structuration. Le langage Pascal fournit cinq modes de structuration distincts, qui peuvent en général être combinés entre eux puisque les composants d'un objet de type structuré peuvent le plus souvent être à leur tour des objets d'un type structuré.

2.2.1 Tableaux

Un tableau est un ensemble d'un nombre fixe d'objets tous de même type, dénotables de façon explicite et accessibles grâce à un indice. La définition d'un type de tableau nécessite donc de préciser le **type des composants** et le **type des indices**. Le premier est quelconque, alors que le second doit être un type scalaire.

Exemple

On peut déclarer :

```
var malade: tableau [jour] de booléen;
    mémoire: tableau [0..maxmem] de entier;
```

puis accéder à :

```
malade [lun] ou à mémoire [i + j].
```

Si les composants d'un tableau sont des tableaux, on obtient ce que l'on appelle couramment un tableau à deux dimensions :

```
var matrice : tableau [a..b] de tableau [c..d] de réel.
```

Dans ce cas, matrice [i] est du type :

```
tableau [c..d] de réel
```

et matrice [i] [j] (que l'on peut abréger sous la forme matrice [i, j]) est du type réel. On peut aussi abréger la déclaration ci-dessus sous la forme :

```
var matrice: tableau [a..b, c..d] de réel.
```

La définition d'un type de tableau peut être préfixée du mot-clé **paquet** qui précise que le compilateur doit rechercher pour le tableau un encombrement minimal, même si c'est au prix d'une moins bonne vitesse d'exécution du programme. Les tableaux du type particulier :

```
paquet tableau [1..n] de car
```

sont appelés des **chaînes**. Il existe des constantes de type chaîne, dénotées par la suite de leurs caractères entre apostrophes.

Les seules opérations possibles sur les tableaux complets sont l'affectation et la transmission comme paramètre (§ 4.1). Elles ne sont applicables qu'à des tableaux de même type. De plus, on peut appliquer les comparaisons =, ≠, <, ⩽, ⩾ et > aux chaînes de même longueur, l'ordre d'énumération du jeu de caractères utilisé déterminant la relation d'ordre.

2.2.2 Articles

Le moyen le plus général d'obtenir un objet structuré consiste à grouper plusieurs objets de types quelconques (éventuellement structurés) pour les considérer et les traiter comme un objet unique. C'est ce que l'on appelle structure en PL/I ou en Algol 68, enregistrement en Cobol, et que nous appellerons article en Pascal.

Un article est formé de **champs** munis de noms distincts. Le nom permet d'accéder à chaque champ individuellement, étant donné la désignation de l'article lui-même. Les types des champs sont quelconques, et un champ donné peut très bien être à nouveau un article, ce qui permet de construire une structure hiérarchique (figure **3**, lignes 12 et 13).

Dans certaines situations, tous les champs n'ont pas tous de signification ensemble. On peut représenter cela au moyen d'une **partie variante** qui commence au mot-clé **cas** et fournit plusieurs choix, selon la valeur d'un champ particulier appelé champ indicateur. L'exemple ci-dessous met en application cette possibilité, combinée avec celle que l'on a de construire des articles de structure très complexe.

Exemple

```
type jour = 1..31;
     mois = (janv, févr, mars, avril, mai, juin,
             juil, août, septe, octo, nove, déce);
     année = 1900..2000;
     date = article j: jour; m: mois; a: année fin;
     vital = (poitrine, taille, hanches);
     personne = article
     nom, prénom: paquet tableaux [1..n] de car;
          naissance: date;
          sitfamille: (célibataire, marié, veuf,
                       divorcé, concubin);
          cas sexe: (féminin, masculin) de
              féminin: (mensurations: tableau
                       [vital] de 50..150);
              masculin: (poids: réel; barbu,
                       chauve: booléen)
          fin;
```

La référence à un champ d'un article se fait grâce à la référence à l'article, suivie du nom du champ. Les références à des tableaux et des articles peuvent se combiner.

Exemple

Si l'on déclare :

```
var foule: tableau [1..M] de personne;
```

foule[i] est de type personne, foule[i].naissance est de type date, foule [i].naissance.m est de type mois ; la référence à foule[i]. mensurations [taille] n'a de sens que si foule[i]. sexe = féminin. Voir aussi le paragraphe 3.2.1 pour un énoncé qui simplifie l'accès aux champs d'un article donné.

Il est possible d'omettre le champ indicateur d'un article avec variante, ce qui permet, en particulier, de décrire une instruction ou un mot d'état d'un langage machine, et plus généralement les situations où un même emplacement de mémoire doit pouvoir être décrit de plusieurs façons différentes.

Exemple

```
type catégorie = (motréel, motentier, caractères,
                  instruction);
         mot = article
               cas catégorie de
               motréel: (valeurréelle: réel);
               motentier: (valeurentière: entier);
               caractères: (valeurcar: tableau
                          [1:4] de car);
               instruction: (opération: (plus,
                            moins, mult,...);
                            opérande:...)
               fin;
```

2.2.3 Ensembles

La notion d'ensemble, fournie par le langage Pascal, rappelle celle du langage courant et celle des mathématiciens, mais elle est plus restrictive. Un objet du type :

type T = **ensemble de** T_0

est un ensemble de représentants pris parmi les objets du type de base T_0. Ce dernier doit être scalaire, mais aussi de cardinalité réduite (16 à 64 suivant les machines) et le nombre ordinal de son plus petit élément doit être nul. Grâce à ces restrictions, il est possible de réaliser la notion d'ensemble de façon très efficace sur les ordinateurs actuels et de fournir un type d'objet semblable à un **tableau** [T_0] **de** booléen, mais d'utilisation beaucoup plus pratique.

Le langage fournit le moyen de construire des ensembles :

```
var c: ensemble de (rouge, jaune, bleu);
c: = [rouge, bleu]
```

Il fournit aussi l'opérateur booléen d'appartenance à un ensemble, les opérateurs d'union, intersection et différence d'ensembles, et les opérateurs booléens d'inclusion, égalité et inégalité. La figure 2, lignes 6, 7 et 13, montre un exemple très simple mais très pratique d'utilisation de cette notion.

Remarque : la restriction sur la cardinalité du type de base n'est pas imposée par la norme, mais la plupart des implantations tendent cependant à autoriser au moins une cardinalité suffisante (supérieure ou égale au nombre de caractères disponibles) pour permettre la définition d'ensembles de caractères.

2.2.4 Fichiers

Les fichiers séquentiels constituent pour le langage Pascal des objets structurés assez semblables aux autres, si ce n'est qu'ils ne peuvent pas apparaître comme composants d'autres objets structurés, et que les opérations qui leur sont applicables sont restreintes à un jeu réduit de procédures et de fonctions prédéfinies. En revanche, les composants d'un fichier peuvent être d'un type quelconque. À un fichier déclaré sous la forme :

```
var f: fichier de T;
```

est associée une **variable-tampon** notée f↑ qui représente la notion de zone-tampon utilisée couramment : c'est le composant actuellement accessible sur le fichier f.

Pour manipuler un fichier, on dispose des fonctions et procédures prédéfinies suivantes :

— relire (f) prépare le fichier à une lecture, c'est-à-dire que f↑ désigne alors le premier composant du fichier ;
— prendre (f) passe au composant suivant sur un fichier en cours de lecture ;
— lire (f, c) affecte à la variable c la valeur de f↑ et passe au composant suivant ;
— fdf (f) est vrai s'il ne reste plus de composant à lire, faux dans le cas contraire ;
— récrire (f), mettre (f) et écrire (f, c) travaillent de façon symétrique aux trois premières procédures précédentes, sur un fichier en écriture.

Les fichiers dont les composants sont des caractères sont appelés **textes** et traités de façon particulière, car tout texte est structuré en lignes. Le prédicat fdln (f) est vrai en fin de ligne et faux ailleurs, les procédures écrireln (f) et lireln (f) permettent de passer à la ligne, respectivement en écriture et en lecture. De plus, les procédures lire et écrire permettent dans le cas de fichiers de texte de traiter des objets des types entier et réel, ainsi que booléen et chaîne en écriture. En écriture, on peut de plus choisir la largeur du champ utilisé pour écrire l'objet et la précision d'écriture des nombres réels, ce qui fournit, sous une forme très simple, toutes les facilités offertes dans d'autres langages par le moyen puissant mais lourd et complexe des formats.

2.2.5 Objets dynamiques et pointeurs

En plus des objets désignés par les variables du programme, qui existent pendant toute la durée de vie de ces variables, le langage Pascal permet de créer et de manipuler des **objets dynamiques** qui existent tant que l'on ne demande pas explicitement leur destruction. On manipule un objet dynamique à l'aide d'un **pointeur**, qui est un nouveau type d'objet lié au type de l'objet qu'il repère.

Étant donné le type personne du paragraphe 2.2.2, on peut par exemple déclarer :

```
var individu: ↑personne; {individu est du type
" pointeur sur personne "}
```

La procédure prédéfinie créer permet de créer un objet dynamique et d'affecter le pointeur qui le repère à une variable du type pointeur correspondant. Dans le cas particulier qui nous occupe, on peut de plus préciser la valeur du champ indicateur :

```
créer (individu, masculin);
```

Il est alors possible d'affecter des valeurs aux champs de l'objet accessible grâce au pointeur.

Exemple

```
avec individu ↑ faire
 {on accède maintenant directement aux champs de
l'article pointé par individu}
début nom: ='Dupont de Nemours';
 prénom: ='Népomucène-Hippolyte';
 avec naissance faire
 début j: = 31; m: = juil; a: = 1903 fin;
 sitfamille: = veuf; sexe: = masculin;
 poids: = 69.5; barbu: = vrai; chauve: = vrai
fin
```

Étant donné que l'un des composants d'un objet dynamique peut être à son tour un pointeur associé au même type, il est possible de construire dynamiquement des structures en liste, en arbre ou en graphe les plus générales.

3. Énoncés

De la même façon que pour les objets, le langage Pascal fournit des **énoncés simples**, qui servent à exprimer des actions élémentaires, et des **énoncés structurés**, obtenus par application de moyens de structuration à des énoncés, simples ou structurés.

3.1 Énoncés simples

3.1.1 Affectation

L'énoncé d'affectation sert à évaluer une expression et à affecter la valeur obtenue à une variable. L'expression s'obtient par combinaison d'opérateurs et d'opérandes, en respectant les règles de compatibilité des types des opérandes. L'ordre dans lequel doivent s'effectuer les opérations est précisé par des règles de priorité et d'associativité des opérateurs, ainsi que par l'usage de parenthèses.

Les opérateurs applicables à chaque type d'objet ont été examinés au cours du paragraphe 2. Dans le cas des objets structurés, il n'existe pas d'opérateur qui s'applique globalement à tous les composants des objets, mais l'affectation est cependant possible dans tous les cas (sauf celui des fichiers) dès l'instant que le type de l'objet à affecter est identique à celui qui est associé à la variable.

3.1.2 Appel de procédure

La notion de procédure étant étudiée au paragraphe 4.1, il suffit de mentionner ici que ce deuxième énoncé de base du langage Pascal s'écrit tout simplement comme le nom de la procédure, suivi s'il y a lieu de la liste des paramètres effectifs. Les opérations d'entrée et sortie sont effectuées par l'appel de procédures prédéfinies (§ 2.2.4 et 4.2), et il n'existe pas d'énoncés spécialisés pour ces traitements, à part de légères extensions dans la syntaxe des appels de procédures.

3.1.3 Branchement

Pour suppléer certaines situations complexes où le jeu d'énoncés structurés n'est pas suffisant, le langage Pascal fournit un énoncé de branchement, mais décourage autant que possible l'utilisateur de s'en servir en imposant de très grandes restrictions sur les étiquettes. Ces dernières sont de simples nombres entiers, elles doivent être déclarées au préalable, et elles ne constituent pas des objets du langage, ce qui interdit sur elles toute manipulation, évaluation ou autre. Le style de programmation adopté par les utilisateurs du langage évite donc tout naturellement l'usage de l'énoncé de branchement, sauf quelques cas particuliers, très rares.

3.2 Énoncés structurés

3.2.1 Groupement d'énoncés

Les énoncés structurés du langage Pascal sont construits à l'aide de suites d'énoncés, incluses dans différents moyens de structuration. Le moyen le plus simple, l'**énoncé composé**, sert simplement à regrouper plusieurs énoncés pour en constituer un seul, en les incluant entre les symboles **début** et **fin**. L'énoncé composé peut apparaître partout où l'on a besoin d'un énoncé simple.

Un deuxième énoncé structuré, l'énoncé **avec**, joue un rôle similaire, mais il sert en plus à permettre de se référer directement aux champs d'un article (exemple du paragraphe 2.2.5).

3.2.2 Énoncés répétitifs

Les énoncés répétitifs servent à demander qu'un groupe d'énoncés soit exécuté un certain nombre de fois. Si le nombre de répétitions est connu à l'avance, on utilise d'ordinaire l'énoncé **pour** ; dans le cas contraire, on utilise l'énoncé **tantque** ou l'énoncé **répéter**.

L'énoncé **tantque** demande la répétition des énoncés concernés tant qu'une certaine condition, fournie par une expression booléenne, est vraie (figure **2** : lignes 9 et 11 ; ou figure **3** : lignes 31 et 46). Si cette condition est fausse la première fois qu'on l'évalue, le groupe d'énoncés n'est pas exécuté du tout.

Dans l'énoncé **répéter**, le groupe d'énoncés est exécuté de façon répétitive jusqu'à ce qu'une certaine condition devienne fausse ; il est en tout cas exécuté au moins une fois.

Dans l'énoncé **pour**, une variable de type scalaire prend successivement toutes les valeurs comprises dans un certain intervalle ; pour chacune de ces valeurs, le groupe d'énoncés est exécuté (figure **2** : lignes 8 et 19). Il existe deux formes pour cet énoncé, suivant que la variable parcourt l'intervalle par valeurs croissantes ou par valeurs décroissantes. Si l'intervalle est vide, le groupe d'énoncés n'est pas exécuté du tout.

3.2.3 Énoncés conditionnels

Dans un énoncé conditionnel, un seul des énoncés qui composent le groupe doit être exécuté. Le choix de l'énoncé à exécuter dépend du mode de structuration. Pour l'énoncé **cas**, on fournit une expression d'un type scalaire, et les énoncés du groupe sont étiquetés par des constantes de ce type ; l'énoncé à exécuter est celui qui a pour étiquette la valeur de l'expression (figure **3** : lignes 36 à 43).

Pour l'énoncé conditionnel, on fournit une expression booléenne et deux énoncés seulement ; le premier (qui suit le symbole **alors**) est exécuté si l'expression est vraie, et le second (qui suit le symbole **sinon** et peut être omis) est exécuté si elle est fausse (figure **3** ; lignes 16, 18, 33, 37, 39 et 41).

4. Procédures et fonctions

4.1 Définition

Les procédures du langage Pascal (et les fonctions qui en sont un cas particulier) jouent le même rôle que les procédures d'Algol 60 et PL/I ou que les sous-programmes de Fortran, c'est-à-dire qu'elles constituent l'un des moyens d'expression les plus fondamentaux du langage.

La forme d'une **déclaration de procédure**, qui doit précéder normalement ses utilisations, est semblable à celle d'un programme, si ce n'est son en-tête : dans le cas d'une procédure, ce dernier précise le nom de la procédure, puis le nom, le type et le mode de transmission des paramètres formels. Tous les types existant dans le langage peuvent être utilisés pour les paramètres formels.

Il existe deux modes de transmission pour les paramètres. Dans le mode **par valeur**, choisi par défaut et inspiré d'Algol 60, le paramètre formel se comporte comme une variable locale à la procédure, qui prend pour valeur initiale celle du paramètre effectif. Dans le mode **par variable**, qui sert pour éviter la recopie d'un paramètre de type structuré ou quand la procédure doit fournir un résultat, le paramètre effectif est évalué comme une référence et substitué au paramètre formel.

Le type du paramètre formel et celui du paramètre effectif doivent être compatibles, ce qui signifie qu'ils doivent être identiques, sauf en cas de transmission de valeur et si les types sont issus du même type de base. C'est une contrainte très sévère dans le cas des tableaux, puisque cela oblige à préciser les valeurs des bornes dans chaque dimension dès la définition de la procédure. La norme ISO (*International Organization for Standardization*) (§ 5) a donc introduit, pour résoudre ce problème spécifique, la possibilité de **paramètres tableaux ajustables**, où seul le type de base des bornes est spécifié dans la définition de procédure. On retrouve ainsi les possibilités présentes dans la plupart des langages, mais sous une forme disciplinée et qui autorise toutes les vérifications automatiques de conformité des paramètres.

Une **fonction** est une procédure qui fournit une valeur scalaire et peut être appelée dans une expression.

Il est possible de définir des procédures ou des fonctions qui acceptent pour paramètre un nom de procédure ou de fonction.

La norme ISO impose que le nombre et le type des paramètres de cette procédure ou fonction paramétrique soient entièrement spécifiés, pour permettre les vérifications de compatibilité par le compilateur.

La figure **3** comportait deux exemples très simples de procédures (lignes 15 à 19 et 20 à 24). Les figures **4** et **5** proposent deux autres exemples bien différents. La figure **4** montre une fonction d'élévation d'un nombre réel à une puissance entière non négative, par une méthode beaucoup plus efficace que la simple multiplication du nombre par lui-même. La figure **5** montre une procédure récursive qui, étant donné un arbre généalogique sans mariage consanguin, l'imprime sous forme lisible en écrivant, sous le nom de chaque personne apparaissant dans l'arbre, le nom et la généalogie de ses parents, décalés sur la droite de *k* caractères.

4.2 Procédures et fonctions prédéfinies

Les procédures et les fonctions énumérées dans le tableau **1** font partie de la définition du langage Pascal, c'est-à-dire qu'elles sont fournies dans toutes les réalisations sur ordinateur de ce langage. La plupart de ces réalisations fournissent de plus un jeu de procédures et de fonctions prédéfinies locales adaptées à l'ordinateur et à son système d'exploitation.

Tableau 1 – Procédures et fonctions prédéfinies du langage Pascal

	ALLOCATION DYNAMIQUE
créer (p)	alloue un nouvel objet dynamique du type associé à la variable de type pointeur p, et le rend accessible *via* ce pointeur
créer (p, i_1 , i_2 , ..., i_n)	permet d'allouer un objet dynamique de type article comportant une hiérarchie de parties variantes ; les champs indicateurs de la hiérarchie prennent les valeurs i_1 , i_2 , ..., i_n
libérer (p) et libérer (p, i_1 , i_2 , ..., i_n)	libèrent l'espace alloué à l'objet dynamique p↑ par l'un des deux appels précédents
	TRANSFERT DE TYPE
tronc (x)	entier obtenu par suppression de la partie fractionnaire du nombre réel x
arrondi (x)	entier obtenu par arrondi du nombre réel x : tronc (x + 0,5) si x ⩾ 0 ou tronc (0,5 – x) si x < 0
ord (x)	nombre ordinal (entier) caractéristique de l'objet scalaire x
carac (x)	caractère qui a pour nombre ordinal l'entier x, s'il existe
tasser (t, i, tp)	transforme le tableau ordinaire t en un tableau paqueté tp, à partir du composant d'indice i
détasser (tp, i, t)	effectue l'opération inverse
	FONCTIONS NUMÉRIQUES
abs (x)	valeur absolue de x, entier ou réel
carré (x)	carré de x, entier ou réel
sin, cos, arctan, exp, ln et rac 2	respectivement sinus, cosinus, arctangente, exponentielle, logarithme népérien et racine carrée de leur argument entier ou réel ; résultat toujours réel
impair (x)	prédicat vrai si l'entier x est impair
succ (x)	successeur de x, s'il existe, dans le type scalaire auquel il appartient
pred (x)	prédécesseur de x, s'il existe, dans le type scalaire auquel il appartient
	ENTRÉES ET SORTIES SUR TOUS FICHIERS
récrire (f)	prépare le fichier f à une écriture (son ancien état est donc perdu)
mettre (f)	écrit sur le fichier f la valeur actuelle de la variable-tampon f↑
écrire (f, x)	écrit sur le fichier f le composant x
relire (f)	prépare le fichier f à une lecture ; son premier composant, s'il existe, est accessible par f↑
prendre (f)	passe au composant suivant du fichier f, s'il existe, et le rend accessible par f↑
lire (f, x)	lit le composant suivant du fichier f, s'il existe, et l'affecte à x
fdf (f)	prédicat vrai s'il ne reste plus de composant à lire sur le fichier f

```
fonction puissance (x: réel; y: entier): réel; { y ⩾ 0 }
  { élévation de x à la puissance y en ϴ (log2 (y)) multiplica-
      tions }
  var z: réel;
début z: = 1;
  tantque y > 0 faire
  début
    tantque ¬ impair (y) faire
    début y: y div 2; x: = carré (x) fin;
    y: = y – 1; z: = x * z
  fin;
  puissance: = z
fin
```

Figure 4 – Exemple de fonctions

```
type généalogie = article individu: paquet tableau [1..n] de car;
                                père, mère: parent
                  fin;
     parent = ↑ généalogie;
procédure imprimer (x: parent; d: entier);
début si x = nil alors écrireln ('': d, '?') { parent inconnu }
  sinon avec x ↑ faire
  début
    écrireln (' '; d, individu);
    imprimer (père, d + k) { généalogie du père };
    imprimer (mère, d + k) { généalogie de la mère }
  fin
fin
```

Figure 5 – Exemple de procédure

Les procédures et les fonctions prédéfinies pour les entrées et sorties sur fichiers de caractères ont été étudiées au paragraphe 2.2.4.

5. Normalisation du langage

5.1 Pascal

La normalisation des langages de programmation, bien que mal connue de la plupart des informaticiens, est sans doute l'activité qui a sur les langages l'influence la plus profonde, aussi bien sur le plan économique que sur le plan technique. Elle a pour but de maintenir le maximum de **compatibilité entre les différentes implantations** et de fournir un texte de référence finale aux utilisateurs et surtout aux implémenteurs. Elle se fait dans les différents organismes de normalisation nationaux [AFNOR (Association française de normalisation) en France, ANSI (American national standards institute) aux États-Unis, BSI (British Standards Institution) au Royaume-Uni, etc.], ainsi que dans l'ISO, fédération internationale des organismes nationaux.

Les principaux langages de programmation normalisés par l'ISO sont Fortran, Cobol, Algol 60, BASIC, PL/I et Ada. Pour certains, la normalisation a simplement consisté à approuver un document existant, alors que, pour d'autres, elle a abouti à un document nouveau, long et complexe, décrivant en fait un nouveau langage de programmation. Le plus fréquemment, l'initiative est venue de l'ANSI, et l'ISO n'a fait qu'entériner le document proposé. Actuellement, le processus de normalisation internationale des langages de programmation s'est notablement accéléré, puisque des travaux sont en cours, parfois achevés ou presque achevés, pour Lisp, C, APL, Prolog, Modula-2 et Pascal étendu, sans compter bien sûr les six langages cités précédemment sauf Algol 60, pour lesquels des extensions ou révisions sont à l'étude.

La norme de Pascal est historiquement le premier exemple qui concerne un langage d'ambitions initialement aussi réduites, et qui est l'œuvre d'une seule personne. C'est la première norme établie en dehors de l'ANSI (en l'occurrence par le BSI : BS 6 182-1982), et

une de celles qui ont été développées le plus rapidement (ou plutôt le moins lentement). Le travail s'est terminé en 1981, en même temps que la promulgation de la norme britannique. La norme internationale (ISO 7 185-1983) et les normes nationales des pays qui ont jugé bon d'en établir une en particulier (États-Unis et France : NF Z65 300) sont datées de 1983 ou 1984.

Deux conflits importants ont vu le jour pendant le processus de normalisation, en bonne partie à cause des particularités de ce processus. D'une part, il a été très difficile de déterminer si le document normatif devait clarifier, réviser ou étendre les documents existants ainsi que le langage décrit. Le résultat est un document qui clarifie et précise, mais qui fait aussi deux extensions très délimitées, qui seront évoquées plus loin (§ 5.2). Le deuxième conflit s'est produit entre l'ANSI et l'ISO : n'étant pas habitués à devoir simplement étudier et critiquer le travail fait par d'autres, au lieu de le faire eux-mêmes, les normalisateurs de l'ANSI ne sont pas parvenus à accepter telle quelle la norme proposée. Bien que le processus entier ait été retardé de plus d'une année pour permettre l'établissement d'un consensus, la norme américaine est différente de la norme internationale et elle a servi de base à la **définition de Pascal étendu** (§ 5.2).

Les normalisateurs ont pris pour document de départ la définition de Pascal par son auteur [4]. Leur travail a consisté principalement à en faire un texte absolument complet, qui donne une réponse précise et sans ambiguïté à toutes les questions que peuvent se poser les implémenteurs et les utilisateurs. Ils ont dû ainsi clarifier la notion de types compatibles, qui intervient dans l'énoncé d'affectation et dans les transmissions de paramètres de procédures, et pour laquelle deux choix différents et incompatibles avaient été faits pour les implantations existantes.

Malgré la très forte pression des utilisateurs, les normalisateurs se sont sagement astreints à ne faire aucune modification au langage existant, reportant à une étape ultérieure toute définition d'extension. Sur deux points précis, cependant, et avec l'accord de l'auteur du langage, ils ont profité de la normalisation pour corriger des défauts du langage. Le premier point n'a qu'une importance minime, même s'il supprime une insécurité gênante : il s'agit de l'obligation de spécifier le type et le mode de transmission des paramètres des procédures ou fonctions qui servent elles-mêmes de paramètres. Le deuxième point est beaucoup plus important, et tente de corriger un défaut reproché au langage Pascal depuis son origine. Il s'agit de la notion de paramètre tableau ajustable, qui permet de définir une procédure ou fonction manipulant des tableaux dont la taille n'est pas connue avant l'appel effectif. La notion choisie dans la norme est simple et limitée ; elle permet une implantation efficace et n'a pas de conséquence sur le reste du langage. C'est cependant ce point précis qui a provoqué le conflit avec l'ANSI, et qui n'apparaît pas dans la norme américaine.

Parallèlement au processus de normalisation a été développée une **suite de validation**, ensemble de plus de quatre cents programmes de petites dimensions, qui permet de vérifier la conformité à la norme d'une implantation particulière. Cet outil précieux sert à la fois aux implémenteurs, pour leur signaler tous les points délicats ou litigieux, et aux utilisateurs, pour leur permettre d'évaluer et de choisir les implantations qui leur sont proposées.

5.2 Pascal étendu

Le fait que la norme internationale de Pascal ne comporte pas (ou presque pas) d'extension par rapport au langage défini par Niklaus Wirth avait été assez mal accepté par la communauté des utilisateurs. En même temps que la normalisation du langage s'achevait, le travail a donc commencé sur les extensions.

L'initiative est venue principalement des comités américains, qui ont procédé en particulier par enquête publique pour déterminer quelles étaient les extensions les plus généralement souhaitées. La longue liste obtenue a été ensuite évaluée à la lumière des critères suivants : indépendance des machines, compatibilité par rapport à la norme existante, cohérence conceptuelle, définition rigoureuse, expérience acquise, objectifs principaux de l'extension proposée.

Ce processus s'est poursuivi de 1980 à 1986, pour aboutir à un nouveau projet de norme internationale, prévu non pas pour remplacer la norme de Pascal, mais pour définir un nouveau langage, nommé Pascal étendu. Cette fois-ci, la collaboration entre les organismes nationaux a abouti à ce que la nouvelle norme, dont les différences principales par rapport à Pascal sont décrites au paragraphe 6.4, soit la même pour les organismes nationaux (ANSI et IEEE aux États-Unis, BSI au Royaume-Uni) et pour l'ISO. Sa promulgation aura lieu en 1990.

6. Langages de programmation similaires à Pascal

Parmi les nombreux langages de programmation qui se sont développés au cours des dix dernières années, Pascal peut être classé à la fois dans deux catégories : celle des langages à fins pédagogiques et celle des **langages d'écriture de systèmes**. Ces derniers sont des langages de programmation qui permettent l'écriture des programmes composant les systèmes d'exploitation : compilateurs, interprètes, assembleurs, éditeurs de liens, mais aussi le noyau du système lui-même, assurant des fonctions très complexes de gestion des différentes ressources fournies par l'ordinateur.

Les langages d'écriture de systèmes ont pour caractéristiques communes qu'ils doivent permettre l'écriture de programmes particulièrement fiables et efficaces, étant donné l'utilisation intensive à laquelle ces programmes sont soumis. S'ils doivent servir à l'écriture du noyau du système, il faut de plus qu'ils permettent d'accéder facilement à toutes les ressources matérielles de l'ordinateur. Le langage Pascal, étant indépendant de tout ordinateur, n'a pas cette dernière propriété, et ne peut donc pas servir à l'écriture de tous les composants des systèmes d'exploitation.

Parmi les véritables langages d'écriture de systèmes, nous en considérerons rapidement deux, comme représentants de deux familles différentes : dans la première famille, on considère que l'accès complet à toutes les ressources matérielles de l'ordinateur nécessite que l'on se débarrasse des contraintes imposées par la notion de type ; dans la deuxième famille, au contraire, on considère la notion de type comme un des outils les plus fondamentaux dont dispose le programmeur pour construire des programmes fiables et efficaces.

6.1 Langages sans type : Bliss

Le langage Bliss a été conçu en 1970, à l'université Carnegie-Mellon par l'équipe du professeur W. Wulf [12], comme un langage d'écriture de systèmes pour l'ordinateur Digital Equipment Corporation PDP-10. Le langage a depuis été adapté à l'ordinateur DEC PDP-11. La figure **6** donne un exemple de fragment de programme écrit en Bliss, qui sert à effectuer l'allocation dynamique d'une zone de mémoire.

Le langage Bliss est un **langage d'expressions**, c'est-à-dire que toute construction exécutable a une valeur (comme en Algol 68 par exemple), avec la structure de blocs et de procédure que l'on trouve dans les langages de la famille Algol. Les objets manipulés sont des **segments** formés de **mots**, eux-mêmes formés de suites de chiffres binaires. Une suite de chiffres binaires contigus constitue l'unité d'information manipulée par le langage, grâce à un **nom** dont la valeur est un pointeur sur ce champ. Il est également possible de se référer aux **registres** de l'ordinateur.

```
begin
  bind k = 13;
  structure vecteur [i] = [i] (.vecteur + .i);
  own vecteur ℓ[k];
  global vecteur m [1 ↑ k];
  routine lier (a,b) =
    begin m[.a] ←..b; .b ←.a end;
  routine délier (de) =
    begin local t; t ←..de; .de ←...de; .t end;
  own vecteur
    taille = (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192),
    masque = (0, 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 8191);
  macro copains (a,b,n) = (((a) xor (b)) egl .taille [n]) $,
    base (a,n) = ((a) and not (.size [n])) $,
    inclut (a,b,bn) = (((a) xor (b)) lss .taille [bn]) $;
  macro essayer (de, vers, id, cond) =
      begin local id;
      incr i from (de) to (vers) do
        begin id ←.l [.i];
          while .id neg – 1 do
            if (cond) then return – 1 else id ←.m [.id];
        end
    end $;
  routine fusion (a,n) =
    begin local tl,tn;
        tl ← l [.n]; tn ← .l [.n]; .....
```

Figure 6 – Fragment de programme en Bliss

La valeur représentée par un identificateur est celle du pointeur associé, et non pas celle du champ accessible grâce à ce pointeur. Le langage utilise donc un opérateur explicite (le point) pour permettre l'accès à la valeur d'un champ.

Les structures d'énoncés sont assez riches, puisqu'elles fournissent six formes d'énoncés itératifs ou répétitifs, deux formes d'énoncés de choix, un énoncé conditionnel, huit mécanismes de sortie des énoncés fermés et un énoncé de lancement de coroutine.

Nota : une **coroutine** est une procédure qui est en relation d'égalité avec la procédure qui l'appelle, et non en relation de dépendance comme dans le cas ordinaire.

La définition d'un mode de structuration des objets consiste à définir un algorithme d'accès qui doit être associé avec une classe d'objets structurés. Il n'existe aucun mode de structuration prédéfini, mais l'utilisateur peut en revanche construire n'importe quelle structure puisque rien ne limite l'algorithme d'accès.

6.2 Langages avec type : LIS

Le langage LIS a été conçu en 1972 à la Compagnie internationale pour l'informatique, par l'équipe de J.D. Ichbiah [13], comme un langage d'écriture de systèmes qui encourage au maximum l'écriture de programmes fiables et efficaces. Il s'inspire beaucoup du langage Pascal, directement ou par l'intermédiaire du langage Sue, mais il inclut deux caractéristiques originales très importantes : le langage comprend deux niveaux, l'un semblable à Pascal et muni de types et de toutes les garanties qu'ils apportent, l'autre à un niveau beaucoup plus bas, dépendant de l'ordinateur utilisé et fortement isolé du reste ; d'autre part, le langage encourage une **programmation indépendante des données**, en séparant la sémantique des objets manipulés de leur technique de réalisation. La figure 7 donne un exemple de fragment de programme en LIS, qui effectue le même travail que celui de la figure 6.

Un programme en LIS est un ensemble de fragments, compilables séparément, reliés hiérarchiquement, et qui décrivent des actions (segments de programme) ou des objets (segments de données). Un segment de données peut être suivi d'une partie de bas niveau qui précise les techniques de réalisation. Pour permettre une programmation indépendante de la réalisation des données, le langage adopte une notation uniforme pour l'accès à un attribut d'un

```
implementation
    type lien =
          plex succ, pred: ref lien; end;
    classemax: constant integer = = 12;
    disponible: array (1..classemax) of lien;
    taillezone: constant integer = = 4096;
    zonelib: zone of taillezone word;
    basezone: ref lien = = @ zonelib;
    init: action;
    efface: action (item: ref lien);
    insère: action (item: ref lien, nbliste: integer);
    réserve, libère: action (bloc: ref, classe: integer);
    ouexclusif: integer action (n, v: integer);
    interface ouexclusif
          input (n: rl, v: 12);
          output (retour: r1);
    begin eor, 1     2 end;
    program efface
          avant, après: ref lien;
    begin avant: = item.pred; après: = item.succ;
          avant.succ: = après; après. pred: = avant;
    end;
    program insère
              avant, après: ref lien;
    begin après: = disponible (nbliste).succ;
          avant: = après. pred; avant.succ: = item; ....
```

Figure 7 – Fragment de programme en LIS

objet, qu'il s'agisse d'un champ d'article, d'un composant de tableau, d'un élément d'ensemble, d'un objet pointé ou de la valeur d'une fonction.

Les structures d'objets sont semblables à celles de Pascal, mais l'utilisateur peut choisir, en cas de besoin, la représentation des types énumérés, la disposition des champs des articles, la technique de réalisation des tableaux, etc. La notation positionnelle, utilisée par la très grande majorité des langages de programmation pour les paramètres des procédures, est supprimée en LIS pour des raisons de sécurité. Les autres caractéristiques du langage sont plus classiques, et la place manque pour les décrire ici.

Le langage LIS est le principal inspirateur du langage Ada.

6.3 Relations entre Pascal et Ada

Le langage Ada (cf article spécialisé [H 2 280] dans le présent traité), étant à vocation universelle, devrait pouvoir en particulier permettre de traiter toutes les applications que l'on programme en Pascal, langage avec lequel il a d'ailleurs une certaine parenté. Une brève comparaison est donc utile.

Indépendamment de nombreuses différences lexicales et syntaxiques entre les deux langages, qui ont leur importance pour les utilisateurs mais ne touchent pas aux possibilités sémantiques, deux aspects de Pascal manquent à Ada : la structure d'ensemble et les procédures ou fonctions transmises en paramètres. En revanche, les aspects d'Ada qui n'apparaissent pas dans Pascal sont innombrables ; citons, en particulier, la généricité, la modularité, la compilation séparée, les processus parallèles, les traitements d'exceptions, la définition précise des types numériques, les entrées-sorties sur fichiers directs, les possibilités de surcharge et de redéfinition des noms, les spécifications de représentation, etc.

Est-ce à dire qu'Ada, étant beaucoup plus puissant que Pascal, rend ce dernier langage périmé et inutile ? La réponse est très clairement négative, car les deux langages ne sont pas destinés à un même usage. Pascal, langage simple, concis et limité, est très adapté à l'apprentissage de la programmation et à la réalisation, par des individus isolées ou de petites équipes, de programmes de taille réduite ou moyenne, en général monolithiques. Ada, langage universel, très complexe et de très grandes dimensions, est prévu pour la programmation d'énormes applications, réalisées par des équipes de grande taille. Son utilisation n'a vraiment de sens que dans le cadre d'un vaste environnement de programmation qui facilite la construction et la gestion des programmes, alors que Pascal s'utilise de plus en plus sur les ordinateurs personnels, où il tend à détrôner BASIC.

L'auteur se gardera d'être prophète et de tenter de prévoir les places respectives que tiendront Pascal et Ada en l'an 2000. On peut cependant penser que Pascal continuera d'être utilisé à cause de ses qualités intrinsèques, et Ada à cause du lourd support que lui apporte le ministère de la Défense des États-Unis. Le sort qu'ont connu les deux précédents langages à vocation universelle, Algol 68 et PL/I, peut cependant suggérer qu'Ada ne délogera pas Pascal, pas plus d'ailleurs que Cobol, APL ou bien d'autres langages de programmation.

6.4 Pascal étendu

Pascal étendu est un nouveau langage de programmation, qui contient tout le langage défini par la norme américaine de Pascal (c'est-à-dire sans les paramètres tableaux ajustables) et y ajoute les points principaux suivants.

6.4.1 Modularité et compilation séparée

Le but est de permettre la compilation séparée des composants d'un programme, tout en conservant la sécurité fournie par le typage fort. Chaque module exporte une ou plusieurs interfaces, qui contiennent des noms définis dans ce module : valeurs, types, schémas, variables, procédures et fonctions. La visibilité des noms exportés peut être contrôlée plus finement grâce aux variables protégées et aux types restreints. La forme même du module sépare clairement ses interfaces de ses détails de réalisation. Tout bloc peut importer une ou plusieurs interfaces, et toute interface peut être utilisée en partie ou en totalité. On peut accéder aux noms avec ou sans qualification par le nom du module. On peut aussi renommer les noms, en les exportant ou en les important. Chaque module peut contenir des actions d'initialisation et d'achèvement. Enfin, la modularité fournit un cadre pour la définition et l'implantation de bibliothèques de composants logiciels, programmés en Pascal ou dans d'autres langages.

Exemple de module

```
module ModulePile ;
 exporte InterfacePile = (TypePile, empiler,
                dépiler, pilevide, pilepleine);
 importe TypeÉlément;
 const max = ...;
 type ArticlePile = article
      sommet: 0..max valeur 0;
      pile: tableau [1..max] de élément
    fin;
    TypePile = restreint ArticlePile;
 procédure empiler (var p: ArticlePile; val:
          élément);
 procédure dépiler (var p: ArticlePile; var val:
          élément);
 fonction pilevide (p: ArticlePile): Booléen;
 fonction pilepleine (p: ArticlePile): Booléen;
fin {en-tête de module};
procédure empile;
début avec p faire si sommet < max alors
 début sommet : = sommet + 1 ;
 pile [sommet]: = val
 fin
 sinon {erreur}
fin ;
 .
 .
 .
fin {du module}.
```

6.4.2 Schémas

Un schéma est une collection de types apparentés. On peut ainsi définir de nouveaux types, statiquement ou dynamiquement, à partir de schémas existants. Les types obtenus par sélection statique d'un schéma ont exactement les mêmes propriétés que les types ordinaires. Les types obtenus par sélection dynamique servent à remplacer et généraliser les paramètres tableaux ajustables. Ainsi, un paramètre formel schématique emprunte les bornes du paramètre effectif correspondant. La déclaration d'une variable locale peut sélectionner dynamiquement le type (et par conséquent la taille) de la variable allouée. De même, la procédure d'allocation dynamique nouveau peut sélectionner dynamiquement le type de la variable à allouer.

Exemples de schémas

```
type intervalle = 1..100;
 vecteur (N: intervalle) = tableau [1..N] de réel;
 matrice (M, N: intervalle) = tableau [1..M, 1..N]
                              de réel;
var k1, k2: entier;
 v: vecteur (10);
 m: vecteur (50);
procédure retourner (var v: vecteur);
 var m, i : intervalle;
  u: type de v;
 début m: = v.N;
 pour i: = 1 haut m faire u[i]: = v[m + 1 – i];
  v: = u
 fin;
procédure VDA (j, k: entier);
 var p: matrice (j + 1, 2*k – 1);
```

6.4.3 Traitements de chaînes

Les possibilités de traitement de chaînes de Pascal étendu incorporent les chaînes de taille fixe et les caractères de Pascal, en les incluant dans le concept plus général de **chaînes de taille variable** (dont la taille est choisie à la déclaration). Toutes les chaînes sont compatibles entre elles pour l'affectation et les comparaisons. Les comparaisons permettent de compléter ou non les chaînes avec des blancs. Des fonctions prédéfinies permettent d'accéder aux chaînes, et en particulier d'en extraire des sous-chaînes. Des procédures prédéfinies étendent aux chaînes les procédures d'entrée-sortie depuis et vers les fichiers de texte.

6.4.4 Autres possibilités

La liste ci-après, non exhaustive, énumère les principales extensions non citées auparavant :

— liaison des variables internes avec des objets extérieurs au programme, en particulier les fichiers ;
— fichiers à accès direct, avec les procédures et fonctions prédéfinies correspondantes ;
— expressions constantes ;
— constructeurs de valeurs structurées ;
— généralisation du type de résultat des fonctions ;
— valeur initiale des variables ;
— relâchement de l'ordre imposé des déclarations ;
— accès aux caractéristiques d'implantation ;
— extensions concernant l'énoncé **cas**, les variantes de type article, les ensembles ;
— nombres complexes ;
— opérateurs booléens avec court-circuit ;
— paramètres de procédures protégés ;
— nombreuses procédures et fonctions prédéfinies supplémentaires.

Au total, Pascal étendu est un nouveau langage, beaucoup plus complet mais également beaucoup plus complexe que Pascal. À la différence de ce qui avait été fait pour Pascal, le processus de normalisation n'a pas servi à réglementer une pratique existante, mais à proposer un cadre pour les extensions à faire dans les implantations à venir. Il reste à savoir si les implémenteurs se conformeront effectivement au cadre qui leur est ainsi proposé, en particulier ceux pour lesquels cela signifie de modifier les implantations existantes.

6.5 Modula-2

6.5.1 Généralités

Le langage Modula-2 a été défini par Niklaus Wirth, le concepteur de Pascal, à la suite de travaux sur la multiprogrammation, et en même temps qu'une nouvelle structure de machine dont ce langage serait l'unique langage de programmation. Il est dérivé à la fois de Pascal, en particulier pour l'ensemble des structures de données qu'il fournit, et de Modula, langage purement expérimental défini par Niklaus Wirth vers 1975. De ce dernier langage, Modula-2 prend les constructions syntaxiques et les concepts permettant la modularité.

Pour des raisons similaires à celles qui ont prévalu pour Pascal, le langage Modula-2 s'est répandu bien au-delà de ce qui avait été prévu par son auteur. Il présente en effet la plupart des caractéristiques de simplicité et de cohérence de Pascal, dans un cadre syntaxique mieux adapté, tout en y ajoutant les très puissants outils qui permettent la modularité, et en offrant simultanément toutes les caractéristiques qui permettent la programmation de systèmes.

Modula-2 est actuellement beaucoup moins répandu que Pascal, même si beaucoup voient en lui le successeur normal de ce langage. Cependant, la communauté de ses utilisateurs s'est suffisamment développée et les applications et les implantations se sont suffisamment multipliées pour que le processus de normalisation internationale du langage ait été entrepris au début des années 80. Ce processus, animé dans l'ISO par le comité britannique, est actuellement en cours d'achèvement, et on peut espérer la promulgation d'une norme internationale pour 1990.

6.5.2 Différences entre Modula-2 et Pascal

On peut séparer les différences entre les deux langages en trois catégories.

■ Les **différences lexicales** sont minimes, elles ne proviennent pas de raisons très fondamentales. Les identificateurs réservés (mots-clés) de Modula-2 sont en majuscules. Certains signes de ponctuation ont changé, d'autres sont placés différemment : les constructeurs d'ensembles sont entre accolades, les intervalles entre crochets et les commentaires entre '(*' et '*)'. Un type pointeur se décrit par la notation POINTER TO...

■ Les **différences syntaxiques** sont plus nombreuses et plus raisonnées. L'idée majeure a été de donner à toute structure d'énoncé une fin unique, qui évite un changement de structure important simplement parce qu'une structure donnée commande plusieurs énoncés au lieu d'un. C'est ainsi que le délimiteur END sert à déterminer toutes les structures d'énoncé et que, du même coup, l'énoncé composé disparaît. Dans un but analogue, les choix d'un énoncé CASE sont séparés par le délimiteur 'I'. D'autres différences syntaxiques servent à lever des limitations gênantes de Pascal. C'est ainsi qu'il est permis d'avoir plusieurs variantes dans un type article, que l'on peut spécifier un pas dans l'énoncé FOR, qu'il existe une nouvelle structure de boucle LOOP ... END avec sortie explicite par l'énoncé EXIT, et que la sortie d'une procédure peut se faire explicitement par l'énoncé RETURN.

■ Mais les plus importantes sont bien sûr les **différences sémantiques**. La plus visible est l'adjonction du concept de module, avec la séparation d'un module en une *partie définition* et une *partie implantation*. Dans la partie définition, on trouve les noms exportés par le module, et par conséquent visibles depuis l'intérieur d'autres modules s'ils les importent. Dans la partie implantation, on trouve les définitions du corps des procédures exportées, ainsi que toutes les autres définitions internes au module et utilisées par ce qui est exporté. Modula-2 permet de plus de définir des modules internes aux procédures, mais ce concept assez complexe n'est que rarement utilisé.

Une autre différence sémantique importante est la suppression totale, par rapport à Pascal, du type fichier et de tout ce qui y est lié. Le langage ne définit donc pas par lui-même d'opérations d'entrée-sortie. Cette suppression drastique est rendue possible par les facilités qu'offre Modula-2 pour la programmation de bas niveau, c'est-à-dire l'accès aux ressources offertes par la machine et le système d'exploitation hôte. Il est par conséquent possible de programmer uniquement en Modula-2 un module prédéfini qui exporte un ou plusieurs types fichiers, ainsi que les procédures qui permettent de les manipuler. De la même manière, les objets dynamiques sont gérés par un allocateur de mémoire également programmé en Modula-2, et les bibliothèques de modules prédéfinis fournis par les implantations peuvent offrir ainsi tous les services d'un système d'exploitation.

6.5.3 Exemple de module

Cet exemple, adapté de [23], permet de dessiner sur un écran graphique à l'aide d'une souris (figure **8**).

```
MODULE Dessiner;
  FROM Terminal IMPORT LectureActive;
  FROM DessinLigne IMPORT
    largeur, hauteur, Px, Py, point, ligne, zone, effacer;
  FROM Souris IMPORT
    touches, Sx, Sy, Curseur, SuivreSouris, MontrerMenu;

  CONST L = 512; (* Taille du carré de tracé *);
    ESC = 33C (* touche 'escape' *); DEL = 177C;

  VAR i, couleur, x0, y0, x1, y1: INTEGER;
    minx, maxx, miny, maxy: INTEGER;
    ca: CHAR;

  PROCEDURE InitialiserEcran;
  BEGIN zone (1, 0, 0, largeur, hauteur);
    Px: = minx; Py: = miny; zone (0, Px, Py, L, L);
    ligne (0, L); ligne (2, L); ligne (4, L): ligne (6, L)
  END InitialiserEcran:
BEGIN
  minx: = (largeur – L) DIV 2; miny: = Hauteur – L) DIV 2;
  maxx: = minx + L; maxy: = miny + L; couleur: = 3;
  InitialiserEcran; Curseur; (* ceci place un curseur sur l'écran *) )
  REPEAT SuivreSouris;
    IF 14 IN touches THEN
      MontrerMenu (''blanc: gris 0: gris 1: noir'', i);
      If i # 0 THEN couleur: = i – 1 END
    ELSIF (15 IN touches) & (minx < = Mx) & (Mx < maxx)
        & (miny < = My) & (My < maxy) THEN
      x1: = (Mx–minx) DIV 8; y1: = (My–miny) DIV 8;
      IF (x1 # x0) OR (y1 # y0) THEN
        Curseur; (* le curseur disparaît *)
        zone (couleur, minx + x1*8, miny + 1*8, 8, 8);
        x0: = x1; y0: = y1;
        Curseur (* le curseur réapparait *)
      END
    END;
    LectureActive (ca);
    IF ca = DEL THEN
      Curseur; InitialiserEcran; Curseur
    END
  UNTIL ca = ESC;
  effacer
END Dessin.
```

Figure 8 – Exemple de programme en Modula-2

Pascal

Langages d'écriture de systèmes

par **Olivier LECARME**
Docteur ès Sciences
Professeur à l'Université de Nice

Références bibliographiques

[1] WIRTH (N.). – *The programming language Pascal. Acta Informatica* 1, p. 35-63 (1971).

[2] HOARE (C.A.R.) et WIRTH (N.). – *A contribution to the development of Algol.* Communications of the ACM 9, n° 6 (1966).

[3] WIRTH (N.). – *Systematic programming : an introduction.* Prentice-Hall (1973) ; (Traduit en français : *Introduction à la programmation systématique*. Masson) (1977).

[4] JENSEN (K.) et WIRTH (N.). – *Pascal user manual and report.* Springer-Verlag (traduit en français) (1975).

[5] HOARE (C.A.R.) et WIRTH (N.). – *An axiomatic definition of the programming language Pascal.* Acta Informatica 2, p. 335-55 (1973).

[6] WIRTH (N.). – *Algorithms + data structures = programs.* Prentice-Hall (traduit en français) (1976).

[7] FINDLAY (W.) et WAH (D.A.). – *Pascal : an introduction to methodical programming.* Pitman (1978).

[8] GROGONO (P.). – *Programming in Pascal.* Addison-Wesley (1978).

[9] BOWLES (K.L.). – *Microcomputer problem solving using Pascal.* Springer-Verlag (1977).

[10] ALAGIC (S.) et ARBIB (M.A.). – *The design of well-structured and correct programs.* Springer-Verlag (1978).

[11] SCHNEIDER (G.M.), WEINGART (S.W.) et PERLMAN (D.M.). – *An introduction to problem solving and programming with Pascal.* Wiley (1978).

[12] WULF (W.), RUSSEL (D.) et HABERMANN (A.). – *Bliss : a language for systems programming.* Communications of the ACM 14, n° 12 (1971).

[13] ICHBIAH (J.D.), RISSEN (J.P.) et HÉLIARD (J.C.). – *The two-level approach to data independent programming in the LIS system implementation language.* Dans : par van der POEL (W.L.) et MAARSSEN (L.A.). – *Machine oriented higher level languages.* North-Holland (1974).

[14] WILSON (I.P.) et ADDYMAN (A.M.). – *A practical introduction to Pascal.* Macmillan (1978).

[15] ATKINSON (L.). – *Pascal programming.* Wiley (1980).

[16] BARRON (D.W.), rédacteur. – *Pascal : the language and its implementation.* Wiley (1981).

[17] BROWN (P.J.). – *From Basic to Pascal.* Wiley (1981).

[18] NEBUT (J.L.). – *Théorie et pratique du langage Pascal.* Technip (1980).

[19] WELSH (J.) et ELDER (J.). – *Introduction to Pascal.* Prentice-Hall (1980).

[20] TENENBAUM (A.M.) et AUGENSTEIN (M.J.). – *Data structures using Pascal.* Prentice-Hall (1981).

[21] LECARME (O.) et NEBUT (J.L.). – *Pascal for programmers.* McGraw-Hill (1984). Également *Pascal pour programmeurs.* McGraw-Hill (1985).

[22] TISSERANT (A.). – *Pascal. Norme ISO et extensions.* Dunod (1987).

[23] WIRTH (N.). – *Programming in Modula-2.* Springer-Verlag (1983).

Normalisation

BS	6192	1982	Specification for computer programming language Pascal.
NF Z 65-300		8.84	Traitement de l'information. Langages de programmation Pascal.
ISO	7185	1983	Programming Languages. Pascal.