

I.U.T. Aix
Département génie informatique



Génie logiciel

1^{er} trimestre 2002/2003 (semaines 1 à 4)

Cours Ada

**Polycopié étudiants pour le cours
première partie (cours 1 à 5B)**

Daniel Feneuille

Nom étudiant :

Panorama complet des 14 premières semaines (numérotées) et, en grisé, les modules où intervient le langage Ada

Voir page suivante le détail du cours semaines 1 à 11 (4h/semaine).
Cette grille sera commentée en cours.

	TD-TP 4 heures	TD-TP 4 heures	Ada Bases (4h)	Cours Ada 2 heures	Cours Ada 2 heures
14	TD-TP 18 Ada	TD-TP 19 Ada	Type Direct		
13	TD-TP 17 Ada	TD-TP 18 Ada	Type séquentiel		
12	TD-TP 15 Ada	TD-TP 16 Ada	Type Perso		
11	TD-TP 14 Ada	TD-TP 13B Ada	Type Date2	Express 13	Interfaçage 14
10	TD et TP Shell	TD-TP 13A Ada	Type Date1	Access 12	Numériques 2
9	TD et TP Shell	TD-TP 12B Ada	Type Texte	Fichier 11 suite	Testabilité
8	TD-TP 12A Ada	Numériques TD	Type Matrice	Fichier 11	Numériques 1
7	TD-TP 11 Ada	Récursivité 2	Type String	Cours 10	Cours 10 suite
6	TD-TP 10 Ada	Récursivité 1	Type Tableau	Cours 8	Cours 9
5	algo-ada 9 TD-TP	Algorithmique numérique	Type Vecteur	Cours 6	Cours 7
4	algo-ada 6,7 TDs	algo-ada 8 TD TP 6, 7, 8	Type Discret	Cours 5	Cours 5 bis E/S
3	algo-ada 3,4 TDs	algo-ada 5 TD TP 3, 4, 5	Type Caractère	Cours 3	Cours 4
2	TD et TP Shell	algo-ada 1 TD-TP	algo-ada 2 TD-TP	Cours 1 (III)	Cours 2
1	Éditeur V.I.	Éditeur V.I.	algo-ada 0 TD-TP	Cours 1 (I)	Cours 1 (II)

Cours n°1 Ada semaines 1 et 2 (généralités)

Objectifs des supports de cours du langage Ada :

Les cours Ada sont faits en demi promotion (suivant la planification détaillée ci-dessous). Ils sont accompagnés de photocopiés (dont celui-ci est le premier exemplaire). Il n'est **pas question**, pour les étudiants, de se satisfaire des **documents accompagnant** le cours car, s'ils reprennent bien l'essentiel de ce qui y sera dit, ils sont **insuffisants pour bien connaître le langage**. Les supports de cours n'ont pas, non plus, la prétention de se substituer aux **bons livres** dont vous trouverez la liste plus loin (notez que pour chacun des titres proposés, au moins un exemplaire est en **bibliothèque** « endroit » que je vous invite à connaître et **fréquenter**). Enfin **le cours est étroitement en synergie avec les TD et les TP** qui sont faits pendant tout le premier trimestre (14 semaines). Là encore des documents seront distribués. **Nous**¹ vous aiderons à en faire la synthèse mais rien ne sera assimilé sans un **travail régulier** et une **pratique sur ordinateur**, elle aussi, **régulière et soutenue**² ! De nombreux ajouts seront proposés (sur le CDRom) sous forme de référence à des fichiers qu'ils vous restera à lire !

Voici le contenu du **cours Ada** au premier trimestre (un aperçu de 11 semaines en 22 pavés de 2 heures repris du planning général de l'enseignement de l'informatique en première année page précédente) :

Semaine 1	Semaine 2	Semaine 3	Semaine 4	Semaine 5	Semaine 6	Semaine 7
Cours n°1 Généralités I	Cours n°1 Généralités III	Cours n°3 Les instructions Compléments	Cours n° 5 Sous programmes	Cours n°6 Types Articles	Cours n°8 Les exceptions	Cours n°10 C.O.O et T.A.D.
Cours n°1 Généralités II	Cours n°2 Types scalaires et attributs	Cours n°4 Types tableaux	Cours n°5 bis E/S simples	Cours n°7 Paquetages Ada hiérarchiques.	Cours n°9 La généricité	Cours n°10 (suite) objets et classes

Semaine 8	Semaine 9	Semaine 10	Semaine 11	Semaine 12	Semaine 13	Semaine 14
Cours n°11 E/S Ada.Text_Io	Testabilité (Boîte Noire et Boîte Claire)	Cours n°12 Type Access	. Cours n°13 Les expressions dérécurivité			
Cours numérique 1 Types digits, delta et décimaux	Cours n°11 E/S (suite) fichiers Séquent. et direct	Cours numérique 2 Convergence, troncature	Cours 14 Interfaçage			

La partie en grisé n'est évidemment plus le cours Ada . Voir la grille générale affichée ou sur le CDRom

Remarque :

Nous avons été obligés de faire un découpage de la matière à enseigner, comme cela est proposé partout, avec des thèmes comme les paquetages, la généricité, les tableaux, les exceptions, les pointeurs, les objets etc. Mais c'est bien là une des difficultés du langage Ada où, en fait, tout se tient, et où il faudrait pouvoir enseigner tout en même temps (impossible !). C'est dans cet esprit que nous serons souvent amenés à **évoquer un thème de façon un peu superficielle** pour mieux y revenir plus tard : par exemple nous parlerons vite des paquetages et les sous-programmes cours généralités III (semaine 2) pour les étudier plus en détail plus tard (semaines 4 et 5).

Langage (Quid ?)

Contrairement à la notion de langue (où on s'intéresse au caractère vocal) la notion de langage fait référence à la notion de signes. Pour le pilotage d'un ordinateur par l'homme la communication passe aussi par un langage (dit artificiel) : **le langage de programmation**. Pour être compris de l'ordinateur le pilotage utilise des instructions

¹ « **Nous** » : sous ce pronom se cachent aussi les noms de quelques collègues (alias les TWO_BE_THREE de l'info) qui formeront l'ossature, au premier trimestre, de l'encadrement et du démarrage Ada TD et TP (ainsi qu'en Algorithmique).

² avertissement que, certains ne prennent pas au sérieux! Voyez les redoublants sincères pour en savoir plus!

codées : **le programme**. Avant d'être exécutable le programme doit être « correct » (compréhensible) c'est l'objectif de la phase de **compilation**. Cette étape est en fait structurée en étapes plus fines (souvent inaperçues) : analyse lexicographique, analyse syntaxique et analyse sémantique. L'ordinateur n'utilise en général pas le même codage que celui du programmeur pour s'exécuter (cela dépend du processeur) il est donc nécessaire de passer par une ultime phase de construction : **l'édition de liens** (notez que ceci n'est pas toujours vrai comme par exemple avec le langage Java). Pour plus de détails voir la page 6 « Langage informatique » (compléments à lire). Ada est un des nombreux langages modernes de programmation et c'est le premier que nous étudierons.

Un peu d'histoire (la petite !).

Ada vous connaissez ce nom ? Cherchez bien, la pub un peu partout ! mais oui : le loueur de voitures. Hélas vous n'y êtes pas ! Si je vous dis, pour vous mettre sur la voie, que c'est l'histoire d'une femme. Alors là, des cinéphiles vont penser à la belle héroïne de « La leçon de piano » mais c'est encore raté ! Non il s'agit de Adélaïde de Augusta Byron, fille du « coquin » poète Lord Byron (seule enfant d'ailleurs qu'il reconnut !). Elle était passionnée de mathématiques, elle travailla avec le génial inventeur Charles Babbage dont elle fut l'égérie. Elle est considérée comme le premier programmeur de l'histoire. Elle mourut ruinée et alcoolique. C'est pour lui rendre hommage que son diminutif « Ada » fut donné comme nom au langage dont vous allez découvrir le contenu en quelques semaines. Si vous êtes intéressés pour en savoir plus, dans les bons ouvrages (liste un peu plus loin) voyez les introductions³. Elles évoquent souvent un peu de l'histoire de cette femme exceptionnelle. Voir aussi les deux pages 7 et 8 (compléments à lire).

La « vraie » histoire du « langage » Ada (résumé succinct) :

Le concept de **langage de programmation** présenté rapidement ci dessus et aussi page 6 (déjà signalé) sera complété oralement en cours.

L'histoire du langage Ada remonte au milieu des années 1970. Le puissant lobby américain qu'est le DOD⁴ décida de ne plus utiliser qu'un **unique** langage de programmation en lieu et place des centaines qu'utilisaient alors tous ses services. Les milieux bien informés s'accordent en général sur le nombre de 400 à 500 langages ou « dialectes » informatiques utilisés à cette époque par le DOD lui-même ou par ses fournisseurs⁵.

De 1975 à 1983 se succédèrent des réunions de comité d'experts qui produisirent un cahier des charges (le génie logiciel était le principal objectif) suivi d'un appel d'offres en 1977⁶. Les lauréats proposèrent en 1979 une première ébauche (avec une « ossature » de Pascal) puis de proche en proche on parvint à la définition d'une norme internationale en 1983 ! Norme que les vendeurs de « compilateurs » doivent respecter⁷. De ce fait les premiers compilateurs « sortirent » en 1985 et hélas cette « inertie » (de 10 ans !) entre les premières descriptions du langage et la possibilité de l'utiliser joua des tours à la promotion du langage.

Aujourd'hui, dans les milieux informatiques le langage Ada est incomparablement reconnu pour la production de logiciels à très haut niveau de fiabilité⁸ mais peu goûté des informaticiens qui programment d'abord et réfléchissent ensuite. En effet, c'est bien **contraignant** Ada car il faut **réfléchir**⁹ et s'imposer des contraintes (en contradiction avec le narcissisme des êtres humains). Mais quand on y souscrit on ne peut que s'en féliciter : par exemple il n'est pas rare de découvrir, avec ce langage, dès la phase de compilation, des erreurs conceptuelles que d'habitude avec un langage permissif on découvre très tard dans la phase de tests¹⁰.

Ce plaidoyer, certainement inaccessible aux débutants informaticiens, sera reformulé et démontré tout au long de ce premier trimestre¹¹. Pour conclure, un dernier regard sur le « petit dernier » il s'agit de « Ada95 » (longtemps appelé Ada9X) et qui est la révision prévue du langage (ce processus prit plus de 5 années : 90-95).

³ par exemple, à la page 18 du livre de Rosen

⁴ le DOD est déjà le « créateur » dans les années 1960 du langage COBOL (encore, hélas ! utilisé en gestion)

⁵ Là encore voir les introductions des bons ouvrages sur Ada (pages 17-19 Rosen et 1-3 Barnes).

⁶ appel d'offres qui sera remporté par une équipe dirigée par un français ! « cocorico ! ».

⁷ On verra, tous ensemble, en TP, ce qu'est un compilateur Ada qui est d'ailleurs, beaucoup plus, qu'un traditionnel compilateur car c'est tout un environnement de développement de logiciels.

⁸ Avionique, fusée (Ariane), conduite ferroviaire (train, métro) etc. (par exemple le tunnel sous la Manche).

⁹ plus précisément il faut beaucoup « spécifier » c'est-à-dire expliciter ce que l'on veut faire !

¹⁰ c'est-à-dire très loin dans le développement du logiciel donc avec un coût prohibitif.

¹¹ Celles et ceux qui le souhaitent pourront d'ores et déjà parcourir le cahier pédagogique n°2 « à propos de génie logiciel » j'y explique le choix pédagogique du langage Ada (à Aix) dans les premières heures de l'apprentissage en I.U.T. en option génie informatique (car telle est notre dénomination). Voir fichier cahier2.doc.

Cette dernière version intègre et permet les objets (concept que l'on verra en détail plus tard) mais concept important et incontournable en informatique d'aujourd'hui¹². Je ferais, au cours n°10 (semaine 7), une présentation des concepts modernes de programmation avec Ada95 et la classification (les objets).

De bons livres « récents » sur Ada¹³ :

La liste est évidemment non exhaustive mais elle nomme les plus complets pour un bon début et que l'on trouve en **bibliothèque** (prêt possible) ! Si vous avez quelques kopecks ce n'est pas une dépense stupide.

- 1997 « programmer en Ada 95 » de BARNES. C'est l'un des premiers livres (au propre comme au figuré). Ce livre de l'actuel président de « Ada Europe » a souvent été réédité. L'ouvrage est traduit en français et présente tout en plus de 700 pages ! **Enorme mais très très bon bouquin** ! Je ferai souvent référence mais à la cinquième édition (attention) : c'est la Bible (ou le Coran ! de tout Ada-tollah).
- 1995 « Méthodes de génie logiciel avec Ada 95 » de Jean-Pierre Rosen. Excellente présentation des concepts de composition (avec Ada 83) et de classification (avec Ada 95) par un excellent pédagogue.
- 1996 « Vers Ada95 par l'exemple » de FAYARD et ROUSSEAU. Des exemples (sources sur Internet).
- 1999 « Programmation séquentielle avec Ada 95 » de Bréguet et Zaffalon aux Presses Polytechniques et Universitaires Romandes (Suisse).

Rappelons que ce cours polycopié est disponible sur le Net soit en ftp://paprika.iut.univ-aix.fr/pub/cours_ada_DF soit chez http://libre.act-europe.fe/french_courses (ACT est l'organisme permettant d'avoir le compilateur gnat.

Objectifs du langage Ada :

Comme on l'a pressenti ce langage avait une vocation **d'unicité** et donc de **langage généraliste** (il fallait pouvoir remplacer tous les langages et dialectes des années 70 !). Mais il fallait en faire un langage durable et professionnel qui intégrait les besoins des années à venir, en clair qui permette de mettre en œuvre les concepts de **génie logiciel** : concevoir et réaliser des « composants logiciels ». Bien sûr, à ce moment de la lecture, peu d'entre vous peuvent imaginer ce que signifie ces concepts. C'est ce que nous découvrirons ensemble pendant tout ce cours associé à des TD et TP. Les maîtres mots à retenir sont ceux de **modules**, de **réutilisabilité**, de **spécifications déclarées et contrôlées**, de **portabilité**, de **généricité**, de **sécurité de fonctionnement**, ... etc.

Le concept de norme (pourquoi normaliser ?) :

Un langage de programmation comme tout langage (au sens général du terme) sert à dialoguer, à se faire comprendre (non seulement par le biais d'un ordinateur particulier mais aussi de tous les ordinateurs et ... de tous les programmeurs qui utilisent les ordinateurs !) Evident ? Pas sûr ! Un langage est soumis à des **règles de syntaxe** (ah la grammaire française pour ne citer qu'elle !). Ces règles sont définies et regroupées dans un **manuel de référence**¹⁴ (L.R.M. en anglais) décrivant la **norme**¹⁵.

Eh bien, il existe très peu de langages informatiques normalisés ! Et quand la norme existe, il y a peu de compilateurs (énormes logiciels (hyper programmes) chargés de vérifier si un programme écrit dans le langage en question respecte la norme) qui respectent eux-mêmes la norme. Ceci signifie par exemple qu'un programme valable sur une machine A contrôlé avec un compilateur B peut ne plus l'être sur une machine C avec le compilateur D. Très fâcheux quand on veut faire du **génie logiciel** c'est-à-dire de la **production industrielle** de logiciels ! Ada est l'un des rares langages qui **oblige les vendeurs de compilateurs** à faire « agréer » chaque année leur compilateur (par une série de tests de validité). A titre d'anecdote s'il est vrai que le langage C (le plus connu et utilisé actuellement) est normalisé il est aussi vrai que rien n'oblige les compilateurs à respecter cette norme (ce qui hélas arrive très souvent). Et l'actualisation moderne du C (nommée C++) langage moderne dit « à objet » que l'on étudiera après Ada (au deuxième trimestre) et très en vogue, n'est normalisé que depuis peu !

¹² pour Ada95 l'erreur de compilateurs tardifs n'a pas été reconduite. Il existe même un compilateur totalement gratuit « free in english » le GNAT copiable via internet (service ftp). Voir fichier Ada_Web.html sur le CDROM.

¹³ le premier « vrai » livre sur Ada remonte à 1982 et c'est : le manuel de référence en anglais (of course!)

¹⁴ disponible gratuitement en ftp (attention beaucoup de pages !). En ligne avec l'interface AdaGide (gnat).

¹⁵ Voir, là encore, la page 6 sur la notion de langage informatique.

La notion de composants logiciels :

De plus en plus, on peut acquérir (gratuitement ou contre argent comptant) des logiciels ou des « briques » de logiciels qu'il suffit de savoir **assembler** entre eux (ou **composer** serait plus juste) de la même façon que l'on peut acheter des composants électroniques à assembler. Cette notion est assez nouvelle et promise à un avenir évident même si des informaticiens (trop conservateurs¹⁶) accrochés à leurs connaissances antérieures et pas actualisées ne veulent rien savoir¹⁷. La diffusion (et l'information) de ces composants ont subi un énorme coup d'accélérateur grâce aux services d'Internet¹⁸ (ftp, Web et Email). Voir le fichier Ada_et_le_Web.html. Savoir **construire et proposer des composants** sérieux et fiables est donc une des missions les plus importantes pour l'informaticien de demain. C'est ce que nous emploierons à montrer dans ce cours et à réaliser en TD-TP. Disons le tout net, Ada nous sera d'une grande commodité pour cet objectif pédagogique¹⁹. Voir aussi **les pages 5 à 9 de Barnes** (5^{ième} édition).

Remarque :

Ce support de cours « généralités I » valable pour la première semaine (2 heures) se prolonge (en première semaine : 2 heures également, et en deuxième semaine : 2 heures encore) avec les documents suivants « généralités II et III » (à suivre dans cet ordre) et ci après :

- Diagrammes syntaxiques (D.S.)
- Les littéraux numériques Ada pour illustrer les D.S.
- Des exemples de programmes (et de composants) commentés, ils sont tirés, au début, des codages des algorithmes conçus dans le cours d'algorithmique qui est enseigné en parallèle.

Résumé :

- Planning des cours Ada
- Langage de programmation (notion)
- Historique de Ada
- Bibliographie
- Norme et génie logiciel
- Composants logiciels

Pages supplémentaires (à lire) : pour le CDRom commencez par la page Intro_cdrom.html

- Texte : Langage (page 6), Byron.et babbage. (pages 7 et 8)
- Fichier Ada_et_le_Web.html (2 pages) sur le CDRom.
- Fichier Cahier2.doc (8 pages) sur le CDRom.
- Fichier AdaFranceTAILL.doc sur le CDRom.



D'après CABU!

¹⁶ et pas forcément « vieux » comme on le croit généralement !

¹⁷ ah mon cher ! Le COBOL, il n'y a que cela de vrai. Ils y reviendront, vous verrez, au COBOL !

¹⁸ le réseau des réseaux. On verra cela plus tard (de façon théorique et pratique). Osez surfer!

¹⁹ c'est l'Ada-tollah qui parle (opposé aux C-rétiques)!

LANGAGE (informatique) : compléments de cours

La notion de **langage** ne doit pas être confondue avec celle de **langue**. Avec la notion de langue on doit tenir compte de la double articulation monème-phonème ainsi que du **caractère vocal**. Le concept de langage est, lui par contre, lié à la **notion de signe**. Un langage est communément défini comme un **système de signes** propre à favoriser la **communication** entre les êtres, quels qu'ils soient, mêmes les êtres inanimés qui ont peut-être une âme ! Mais on s'égaré du sujet ! Trois éléments fondamentaux interviennent dans la détermination du signe : le signifiant, le signifié et le référent. D'où les trois aspects d'un langage : l'aspect **syntactique** (champ des signifiants), l'aspect **sémantique** (relations signifiants-signifiés) et l'aspect **pragmatique**, qui a pour mission de rendre compte du contexte de la communication (champ des référents).

Les langages artificiels et, en particulier, les **langages informatiques** qui interviennent dans la communication homme-machine n'échappent pas à ces distinctions. Ainsi, en programmation, des considérations syntaxiques permettent de définir en quoi un programme est correct, c'est-à-dire conforme aux règles d'écriture imposées par **la norme du langage** (mais ceci ne signifie pas qu'il "marche"). La syntaxe est ici l'ensemble des règles permettant la formation d'expressions valides à partir d'un alphabet (vocabulaire) qui varie selon le langage. Ces règles sont elles-mêmes présentées dans un autre langage ! On parle alors de métalangage ! Il doit être le plus clair et le moins ambigu possible (voir ci dessous).

On distingue trois grandes techniques d'écriture de règles : avec des **phrases très verbeuses** expliquant en langage naturel (utilisées avec le langage COBOL par exemple), des dessins ou **diagrammes syntaxiques** (souvent utilisés dans les manuels de PASCAL et parfois en Ada) et enfin la **notation BNF** (très prisée dans les descriptions des langages tels Algol, C, Ada etc.). En Ada, la grammaire du manuel de référence utilise la notation BNF (voir par exemple dans l'aide de l'IDE AdaGide « Language RM annexe P).

Dans les langages de programmation, l'aspect **sémantique** est mis en évidence par les "performances" du programme, c'est-à-dire par les relations qui existent entre un ensemble d'instructions syntaxiquement correctes et un ensemble de résultats. L'aspect **pragmatique** des langages utilisés en informatique, moins net, trouve quant à lui son illustration dans des situations spécifiques : relations programmes-machines, programmes-compileurs, machines virtuelles, etc.

La communication entre l'homme et la machine ne se fait pas directement, mais par étapes qui impliquent les interventions successives du programmeur et de programmes spécifiques (assembleur, macro-assembleur, compilateur, éditeur de liens, chargeur, interpréteur) ; elle fait donc intervenir plusieurs niveaux de langages. L'intervention du programmeur, lors de la rédaction du programme, a pour objet de traduire le langage naturel (écriture de spécifications) en un langage appelé langage source, qui peut être : un langage symbolique (langage assembleur), le plus éloigné du langage naturel, caractérisé par une **syntaxe rigide** ; un macro-langage, utilisant des "macro-instructions", aux fonctions plus complexes que celles des instructions proprement dites ; un langage **évolué** (Fortran, Algol, APL, Cobol, Basic, PL/1, Pascal, Ada, Eiffel, C, C++, Prolog, Java, etc.), orienté vers les problèmes et dont la structure est la plus proche de celle du langage naturel, avec une syntaxe et un alphabet qui est une extension de l'alphabet naturel (lettres, signes, chiffres, expressions). Tous ces langages forment ce qu'on appelle la classe des **langages de programmation**.

On note aussi les langages dits de requête qui servent à la consultation de bases de données (SQL, par exemple, pour Structure Query Language) et en micro-informatique, des langages spécifiques, comme Dbase (en baisse !), pour utiliser des gestionnaires de données et Microsoft Access plus récent (membre de la trilogie à Bilou). Enfin les langages dits de commande (SHELL) permettent à un opérateur d'échanger des informations et des réponses avec la machine permettant entre autres de piloter les différentes étapes de création d'un programme exécutable évoquées plus haut. Ces commandes sont aujourd'hui plus agréables à utiliser grâce aux interfaces graphiques (WINDOWS) utilisant en harmonie icônes, textes, clavier, écran et souris. En parallèle aussi les langages de script qui fleurissent de façon continue (Javascript, PHP, Python etc.).

Texte écrit à partir d'une page modifiée et complétée de l'encyclopédie "Universalis".

Pour en savoir plus il faut lire les deux pages suivantes sur Ada Byron et Babbage.

Pour connaître l'histoire de l'informatique voir le texte "Une très brève histoire de l'informatique" à l'adresse Internet <http://dept-info.labri.u-bordeaux.fr/~dicky/HistInfo.html>

La famille de “ Ada ” Byron.

Et quelle famille mes aïeux²⁰ !

Les Byron descendaient d'une très ancienne famille de Vikings. Le premier baron Byron, général fidèle aux Stuart, fut anobli en 1643. Le grand-père de Ada était un inconscient prodigue et séduisant. Il avait enlevé, fait divorcer et épousé en premières noces la marquise de Camarthen, qui mourut en 1784, laissant une fille, Augusta Byron (1783-1851)²¹. Il se remaria avec Catherine Gordon de Gight, descendante de l'ancienne famille royale d'Écosse. Ce mariage fut malheureux; les deux époux se séparèrent bientôt laissant un fils George Gordon Byron²². George n'avait pas trois ans quand son père mourut, laissant sa femme dans la misère. L'enfant fut élevé par cette mère instable, passionnée, irascible, à la tendresse tyrannique. Il était pied-bot, soit de naissance, soit à la suite d'une paralysie infantile, et cette infirmité l'affligea profondément. Ils vécurent dans la gêne. A la mort de son grand-oncle, George devint le sixième baron Byron et hérita d'un pauvre domaine. Le château, ancien prieuré normand, était en ruine, et les terres lourdement hypothéquées. Envoyé en pension, George Byron y acquit une solide connaissance du latin et du grec, une grande admiration pour les lettres classiques et la littérature anglaise du XVIII^e siècle, un goût très vif pour la poésie et l'histoire. Il y fréquenta des jeunes gens de son rang et, malgré son infirmité, pratiqua avec brio la natation, le cricket et l'équitation. Son développement affectif fut précoce; à l'en croire, il n'avait que dix ans lorsqu'il tomba amoureux d'une cousine ! Plus tard il s'éprit plus sérieusement d'une autre cousine un peu plus âgée que lui. Plus tard, après avoir pris possession de son siège à la Chambre des lords (13 mars 1809), Byron, s'embarque pour son “ grand tour ”, le 2 juillet 1809. Au plus fort des guerres napoléoniennes, il traverse le Portugal et l'Espagne, gagne Malte, puis l'Albanie. En décembre 1809, il arrive à Athènes d'où il repart pour l'Asie Mineure. Le 3 mai 1810, il traverse, tel Léandre, l'Hellespont à la nage, exploit sportif dont il est très fier. Il séjourne deux mois à Constantinople et regagne Athènes en juillet 1810. Il demeure en Grèce jusqu'en avril 1811, voyageant, étudiant, écrivant. À son retour à Londres, en juin 1811, il rapportait plusieurs poèmes et une sorte de journal de voyage qu'il publia. La réussite, jointe à la jeunesse du poète, la beauté, l'élégance et l'excentricité du dandy, fit de lui l'idole du jour et lui valut de faciles succès féminins. Certaines admiratrices passionnées s'imposèrent littéralement à lui; maîtresses indiscretes, elles entraînent le jeune auteur à la mode, grisé de sa popularité, dans des intrigues compliquées. Ces aventures, colportées, embellies, avilies par des confidences indiscretes, contribuèrent à créer le personnage légendaire de bourreau des cœurs. Au cours de l'été 1813, il revit, après plusieurs années de séparation, sa demi-sœur Augusta. Qu'ils aient eu l'un pour l'autre une affection passionnée et “ étrangement fraternelle ” ne fait pas de doute, comme en témoignent ses poèmes à Augusta. Y eut-il inceste ? Medora Leigh, fille d'Augusta, née le 15 avril 1814, en fut-elle le fruit ? Il n'y eut à l'époque que de très vagues soupçons. La destruction de ses *Mémoires*, aussitôt après sa mort, donna à penser qu'il y avait un scandale à cacher. En dépit du succès de ses ouvrages et en raison d'un train de vie ruineux, le “ noble lord ” était pauvre et criblé de dettes. Il chercha à redorer son blason en épousant une riche héritière. Le mariage eut lieu le 2 janvier 1815. Cette union fut malencontreuse : la mésentente conjugale fut aggravée par des difficultés financières. Le 10 décembre 1815, lady Byron donnait naissance à une fille, Ada²³; le 15 janvier 1816, lord Byron chassait sa femme de chez lui. Puis, regrettant cette décision, il tenta vainement d'obtenir une réconciliation. Il dut se résigner en avril à signer un acte de séparation qui le dépouillait de sa femme, de sa fille²⁴ et de la moitié de sa fortune. L'affaire fit scandale, et la société qui l'avait adulé se déchaîna contre lui...

On arrête là le récit de cette saga ! On pourra en savoir plus dans les bonnes encyclopédies notamment “ Universalis ” dont sont extraites (après épuration ou arrangements) ces quelques lignes. On verra avec le papier sur Babbage (page suivante) que la vie de “ notre Ada ” n'est pas mal non plus ! Tel père telle fille!

Pour en savoir plus voir : <http://www.mo5.com/MHI/HISTOIRE/lovelace.htm> et aussi en <http://www.mo5.com/MHI/HISTOIRE/babbage.htm>.

²⁰ On se contentera du grand-père et du père et c'est déjà pas si mal !

²¹ Elle ce n'est pas “ notre Ada ” mais sa “ demi-tante ” comme on va le voir.

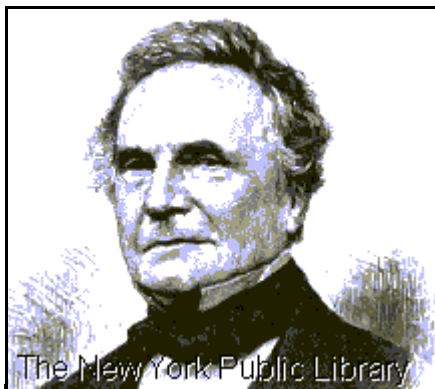
²² Père de notre Ada ! Le plus connu de cette illustre famille (dandy et poète romantique). Coquin à ses heures !

²³ Là oui, cette Ada c'est la nôtre : Augusta Adelaï de Byron future comtesse de Lovelace.

²⁴ La seule enfant, donc, qu'il reconnut.

Babbage : le génial “ inventeur ” et Ada : la première programmeuse.

Babbage, Charles (1791-1871), mathématicien et théoricien de l'organisation industrielle britannique, dont les travaux sur la mécanisation du calcul sont à l'origine de l'ordinateur.



Extraits de l'encyclopédie “ Encarta ”

(texte ci-contre et image).

Dans les années 1820, Babbage conçoit une machine à différences, un appareil mécanique pouvant effectuer des calculs mathématiques simples avec une précision pouvant atteindre 31 chiffres. Cependant, il dut abandonner sa construction pour des raisons financières. Dans les années 1830, il conçoit une machine analytique à cartes perforées, afin d'effectuer des calculs plus compliqués, mais à nouveau cet instrument ne vit jamais le jour. Cette réflexion sur la mécanisation du calcul annonce l'informatique. À l'origine des théories de l'organisation industrielle, Babbage est l'auteur d'un *Traité de l'économie des machines et des manufactures* (1832) analysant la production automatique des manufactures et la division du travail, ce qui lui vaut d'être souvent cité par Karl Marx dans *le Capital*. Ses *Reflections on the Decline of Science* (1830) reprennent la

tradition leibnizienne du calcul infinitésimal.

Fils de banquier, Charles Babbage ne suivra pas la carrière de son père. Il était trop épris de mathématique, d'astronomie et de mécanique. En 1814 il épouse une jeune brunette qui lui fera une grande famille²⁵. A l'origine c'est pour palier les erreurs de calcul qui encombrant les tables astronomiques et nautiques qu'il envisage une machine à calculer plus proche d'une “ machine d'horlogerie ” que des calculateurs actuels ! Après de nombreux échecs et beaucoup d'argent perdu il fait une pause en 1827²⁶. Un an plus tard il reprend ses travaux. Malgré des aides ministérielles la machine à différence sera condamnée. Loin de se résigner il prépare sur “ papier ” cette fois une autre machine plus simple “ la machine analytique ”. Elle ne fut jamais construite de son vivant mais sera fabriquée vers 1990 ! et exposée au “ Science Museum ” de Londres. Les 300 plans et 7000 pages de notes laissés par Babbage permettent de classer cette machine parmi les ancêtres de l'ordinateur actuel. En 1833 il rencontre une brune aux yeux de jais : “ Ada Byron comtesse de Lovelace ”. Passionnée de mathématiques, elle aussi, elle devint sa collaboratrice²⁷ en 1842. Elle écrivit pour la machine analytique “ fictive ” de Babbage des “ programmes ”²⁸ destinés à “ tourner ” dès que l'engin serait opérationnel. Il ne vit jamais le jour de son vivant comme on l'a signalé. Des informaticiens s'amuserent, plus tard, à coder les algorithmes de la comtesse²⁹. Ils “ tournèrent ” paraît-il du premier coup ! On voit ainsi en Ada Lovelace le premier programmeur de l'histoire ! La comtesse mourut à 36 ans d'un cancer après s'être beaucoup adonné à l'alcool, à l'opium et aux courses de chevaux ! Babbage lui mourut bien plus tard à 80 ans en 1871.

Pour la petite histoire soixante quinze ans plus tard (en 1946), en Pennsylvanie, naissait le premier **vrai** calculateur électronique baptisé ENIAC (Electronic Numerical Integrator And Calculator). Pour en savoir plus consulter <http://dept-info.labri.u-bordeaux.fr/~dicky/HistInfo.html> .

²⁵ Un an avant la naissance de “ Ada Byron ” née en 1815.

²⁶ Date à laquelle il perd son père et sa femme.

²⁷ On dit aussi “ égérie ” ou inspiratrice. En aucune façon elle ne fut sa maîtresse ni sa femme comme l'écrivent, parfois, certains textes à sensation sur le langage Ada.

²⁸ pour calculer les nombres de Bernoulli notamment.

²⁹ Codés en langage PL/1 (produit IBM destiné à remplacer FORTRAN et COBOL) et qui n'eut pas de succès.

DIAGRAMMES SYNTAXIQUES (D.S.) ou diagramme de CONWAY

Ada est un **langage** de programmation (on vient de le voir dans les généralités). Comme tous les langages, celui-ci obéit à des **règles grammaticales précises (syntaxe)** qui peuvent être décrites :

- soit par des **phrases** (elles sont souvent **lourdes** et difficiles à comprendre) c'est le cas des présentations d'un vieux langage : le COBOL (définitions verbeuses !).
- soit par des **schémas**, comme des Diagrammes Syntaxiques (en abrégé D.S.). Cette description est très utilisée dans la présentation du langage PASCAL (voir un livre correspondant).
- soit par une troisième description dite **B.N.F.** Utilisée en Ada dans le manuel de référence (document accessible sur le Web ou dans AdaGide). **B.N.F.** est vu après les D.S. en page 3.

I - Définition des symboles utilisés pour écrire un D.S. (Diagrammes Syntaxiques)

- cartouche arrondie ou ronde : le symbole est à prendre "à la lettre". Symbole dit : terminal.
- cartouche rectangulaire : renvoie à une syntaxe définie « ailleurs » (avant ou après).
- flèche : définit le parcours du diagramme (aide à sa lecture).

II - Diagramme de base.

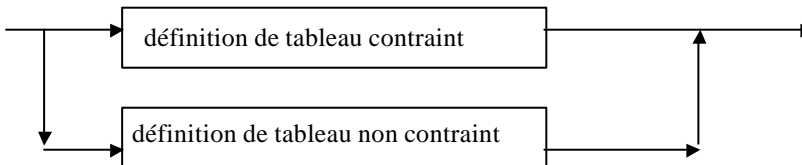
a) schéma séquentiel :

exemple : Déclaration d'une **constante universelle** en Ada (voir exemples page 4 en bas)

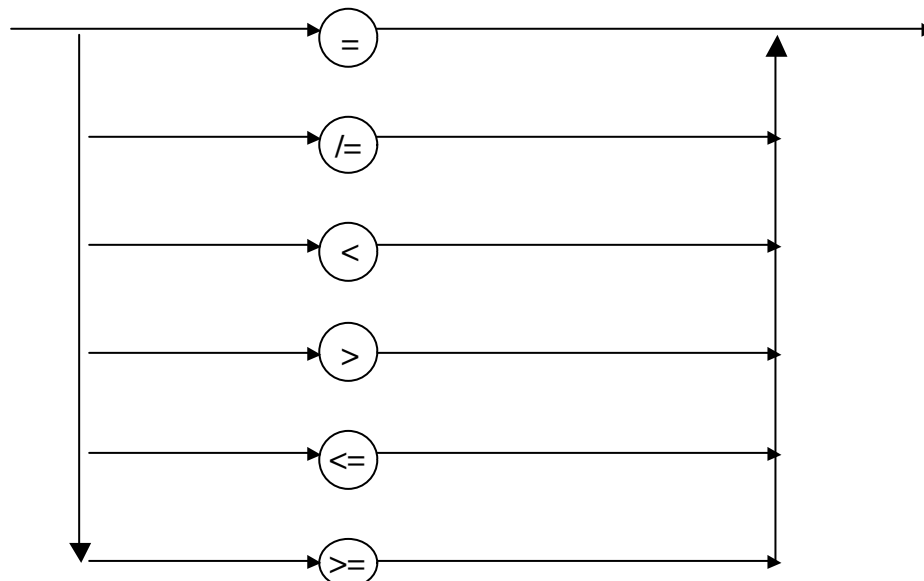


b) schéma alternatif :

exemple : Définition de **type tableau** en Ada (vue au cours Ada n°3)



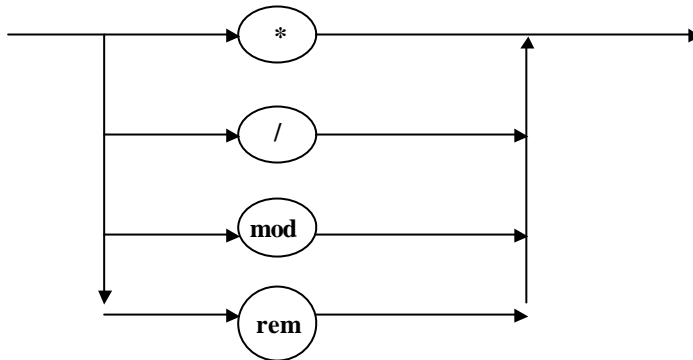
exemple : **opérateur de relation** Ada :



Exercice 1 :

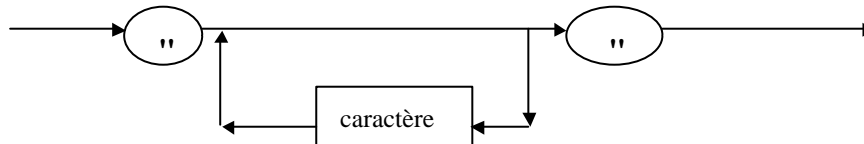
Etablir les trois D.S. correspondant aux définitions (seul le 2^{ème} est corrigé !):

- **lettre** : toutes les lettres de 'A' à 'Z' puis les lettres de 'a' à 'z' (idem pour les chiffres).
- **opérateur multiplicatif** : un opérateur multiplicatif est un des 4 symboles suivants : *, /, mod, rem
- **type prédéfini** : les 6 types suivants sont prédéfinis (c'est-à-dire connus sans qu'il soit besoin d'en donner la définition) : INTEGER, FLOAT, CHARACTER, STRING, BOOLEAN, DURATION.

c) **schéma répétitif** :

exemple : définition d'un **littéral chaîne** :

C'est une suite (éventuellement vide) de caractères encadrés de **guillemets**. Le diagramme syntaxique est le suivant :

Littéral-chaîne

Observez bien le sens des flèches ! La notion de caractère sera vue plus loin. Le problème du caractère guillemet dans un littéral chaîne est résolu à part et n'est pas décrit dans ce D.S.

Exercice 2 : (après la pause d'intercours !)

Donner le D.S. correspondant à la définition suivante : suite de lettres, de chiffres ou de _ (souligné), le premier caractère étant une lettre, le _ ne finissant pas la suite de symboles et n'apparaît pas deux fois consécutivement (c'est la définition d'un **identificateur** da il faudra s'en souvenir !). Voir corrigé page 4.

Exercice 3 :

Ecrire le D.S. correspondant à la définition suivante : Un commentaire Ada est une séquence (éventuellement vide) de n'importe quel caractère précédé par les deux caractères --

Exemples : -- ceci est un commentaire
-- ceci est un commentère

le deuxième commentaire est également correct (les accents sont acceptés et l'orthographe n'est pas contrôlée !)

III - Notations B.N.F.

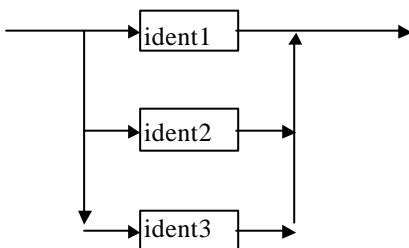
Une représentation très utilisée en théorie des langages (voir cours correspondant niveau BAC + 3) est la notation dite B.N.F. (Backus-Naur-Form). Outre son aspect purement descriptif d'une grammaire, elle est sous-tendue par une théorie mathématique qui la rend indispensable. De plus elle peut être directement utilisée par un traitement de texte sans graphisme ou un analyseur syntaxique. Malgré cela nous ne l'utiliserons pas dans ce cours car **elle est moins lisible que les diagrammes syntaxiques**. Cependant elle est utilisée dans le manuel de référence Ada (voir par exemple Barnes pages 595 à 609) et de ce fait **doit être connue**.

Sept symboles particuliers (méta-symboles) sont utilisés :

- ::= qui signifie "est défini comme"
- | qui signifie « ou bien », ce symbole marque donc l'alternative.
- [] qui signifie que les éléments entre les crochets peuvent être présents 0 ou 1 fois,
- { } qui signifie que les éléments entre les accolades peuvent être présents 0, 1 ou plusieurs fois,
- ' ' qui encadrent des éléments terminaux, (correspondant aux cartouches arrondis des diagrammes syntaxiques),
- < > qui encadrent des éléments non terminaux (correspondant aux cartouches rectangulaires des diagrammes syntaxiques),
- () qui permettent de regrouper des parties de définition, ces parenthèses sont utilisées chaque fois qu'il y a ambiguïté.

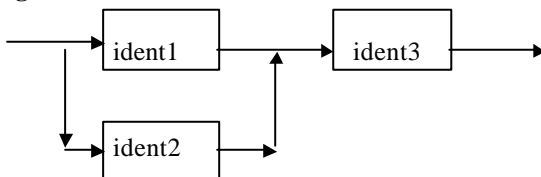
Par exemple, les 4 diagrammes syntaxiques ci-dessous :

diag1:

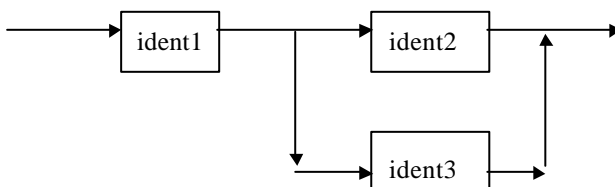


Diagrammes évoqués
au tableau

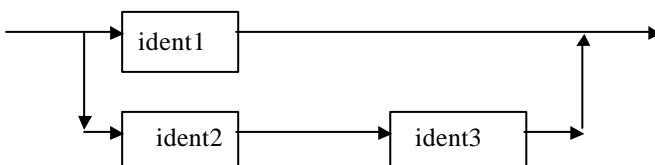
diag2 :



diag3 :



diag4 :



s'écrivent, respectivement en notation B.N.F :

observez bien le rôle des parenthèses! Et la compacité de la présentation!

$\langle \text{diag1} \rangle ::= \langle \text{ident1} \rangle \mid \langle \text{ident2} \rangle \mid \langle \text{ident3} \rangle$

$\langle \text{diag2} \rangle ::= (\langle \text{ident1} \rangle \mid \langle \text{ident2} \rangle) \langle \text{ident3} \rangle$

$\langle \text{diag3} \rangle ::= \langle \text{ident1} \rangle (\langle \text{ident2} \rangle \mid \langle \text{ident3} \rangle)$

$\langle \text{diag4} \rangle ::= \langle \text{ident1} \rangle \mid (\langle \text{ident2} \rangle \langle \text{ident3} \rangle)$

Exemples : (retrouvez ou donnez les D.S. équivalents)

$\langle \text{constante universelle} \rangle ::= \langle \text{liste d'identificateurs} \rangle \text{' ':' constant' '='} \langle \text{expression statique universelle} \rangle$

$\langle \text{entier} \rangle ::= \langle \text{chiffre} \rangle \{ \text{'_' ' '} \} \langle \text{chiffre} \rangle$

Exercice 4 :

Donner la représentation BNF d'un opérateur multiplicatif défini dans l'exercice 1(Cf. le D.S.).

Exercice 5 :

Construire les diagrammes syntaxiques correspondant aux définitions BNF suivantes :

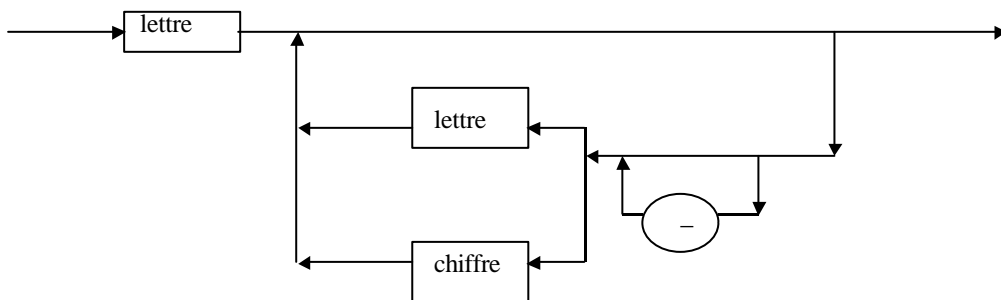
$\langle \text{instruction if} \rangle ::=$
 $\text{'if' } \langle \text{expression} \rangle \text{'then' } \langle \text{suite d'instructions} \rangle \{ \text{'elsif' } \langle \text{expression} \rangle \text{'then' } \langle \text{suite d'instructions} \rangle \}$
 $\{ \text{'else' } \langle \text{suite d'instructions} \rangle \} \text{'end if' ';'}$

$\langle \text{instruction loop} \rangle ::=$
 $[\langle \text{nom simple de boucle} \rangle \text{' ':'}] [\langle \text{schéma d'itération} \rangle] \text{'loop' } \langle \text{suite d'instructions} \rangle$
 $\text{'end loop' } [\langle \text{nom simple de boucle} \rangle \text{' ':'}]$

Exemples de constante universelle Ada (à revoir !):

```
MAX : constant := 30      -- la valeur est un littéral numérique
PI_3 : constant := abs (PI ** 3); -- la valeur est une expression statique
JOUR : constant := LUNDI; -- la valeur est un littéral énumératif
```

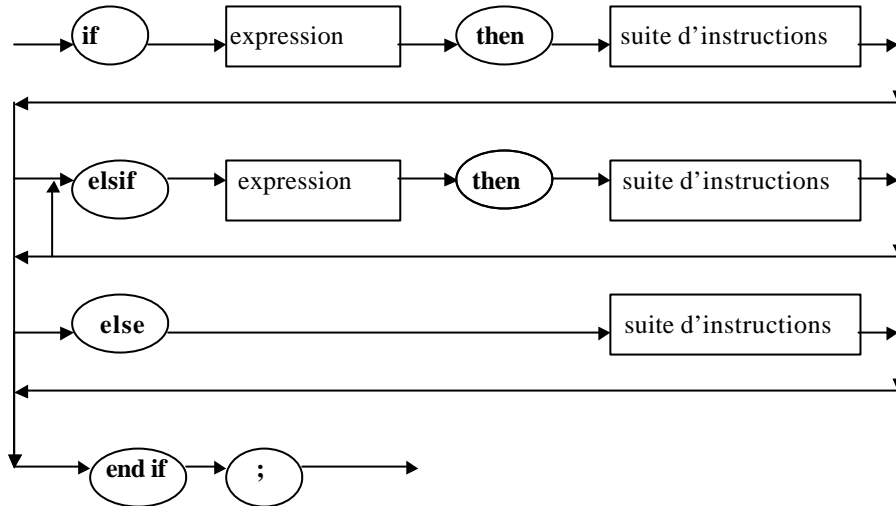
corrigés exercice 2 de la page 2 en DS puis en BNF (à retenir) :



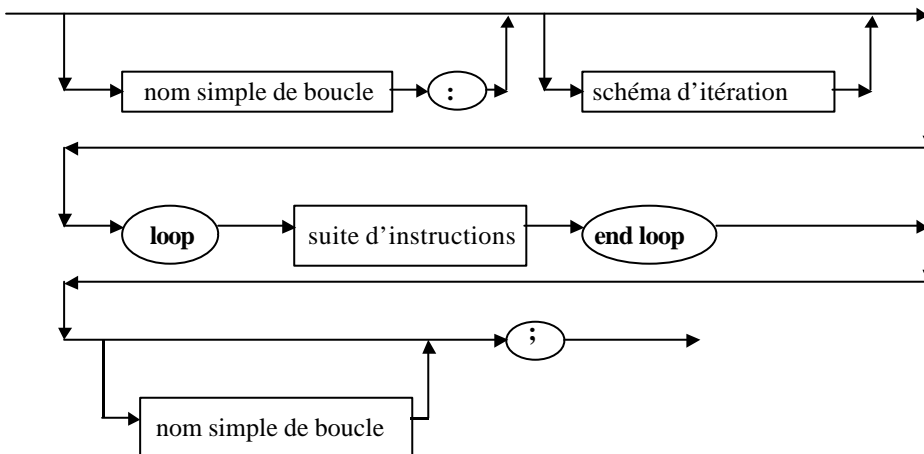
exercice : mettre ce diagramme en B.N.F. (corrigé page suivante)

corrigés exercice 5 (voir cours n° 3 des exemples de codage Ada utilisant **if** et **loop**) :

instruction **if** :



instruction **loop** :



remarques :

- on verra des applications des D.S. dans le fascicule de cours : « les littéraux numériques Ada » page suivante.
- Nous utiliserons souvent les D.S. pour illustrer des concepts Ada.
- Barnes utilise, par contre, la notation BNF dans son livre (déjà cité pages 595 à 609)
- Le manuel de référence utilise lui aussi la notation BNF (déjà dit)

Corrigé de l'exercice bas de page 4 :

<identificateur Ada> ::= <lettre> { ['_'] (<lettre> | <chiffre>) }

LES LITTÉRAUX NUMÉRIQUES Ada

Avertissement : Ce « module » de cours fait suite à celui des Diagrammes Syntaxiques (D.S.) qu'il enrichit en l'illustrant. C'est l'occasion aussi de « faire » de l'Ada sans en avoir l'air. Il s'inscrit donc bien dans le début du cours sur le langage Ada (généralités II).

Le concept de littéral numérique Ada.

Pour définir ce concept on pourrait dire qu'il s'agit de la façon de dénoter (ou représenter) en Ada une valeur numérique. Cette dénotation est bien sûr assujettie à des **règles** strictes que l'on peut énoncer sous forme explicite ou ... sous forme de D.S. ou en BNF (et nous voilà revenus au « module » de cours précédent!).

Exemples d'écriture :

a) les littéraux entiers : (écriture où il n'y a pas de point décimal)

13 est un littéral numérique entier (c'est évident).
 Mais 1_993 est aussi un entier (plus étonnant ! non ? mais ceci est vu en BNF pages 4 et 5!)
 Et que dire également des quatre suivants :
 7 # 123 # E2
 2 # 01_0010 #
 16# 2A3C #
 12E+8

Les trois premiers sont dits **entiers basés** et le quatrième dénote un **grand entier** (à revoir).

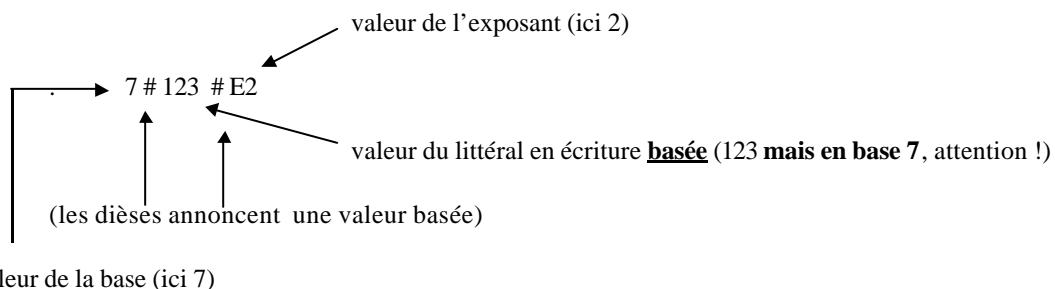
b) les littéraux réels : (on note la présence d'un point décimal)

3.14159_2653 est un littéral numérique réel (facile ?)
 mais aussi les deux suivants : 6.02E-24 et 2 # 101.01 #E1

tout ceci mérite des explications et donc ... des règles!

Remarques : (avant de se lancer dans les diagrammes de CONWAY ou D.S.)

- 1_993 vaut 1993 et 3.14159_2653 vaut 3.141592653. Ceci signifie que le **trait bas** (dit aussi le souligné) est uniquement un **instrument de lisibilité** de la valeur numérique (entière ou réelle). A utiliser donc **sans restriction** si cela doit rendre le nombre plus lisible.
- On distingue **deux classes de littéraux** (et ceci n'a rien à voir avec la classification en entier ou en réel) : les littéraux dits basés et les autres. Les littéraux basés se caractérisent par une valeur (dite basée) **encadrée par deux dièses (#)**, la valeur de la base **précède** le premier dièse, le deuxième dièse étant éventuellement suivi d'une valeur "exposant". Nous allons y revenir. Ainsi : (voici une bonne révision sur la numération !)



d'où : $7 \# 123 \# E2 = (1 \cdot 7^2 + 2 \cdot 7^1 + 3 \cdot 7^0) \cdot 7^2 = 3234$ (mais cette fois en base 10)

D'autres exemples :

- $2 \# 01_0010 \# \Rightarrow$ **entier basé**, de base 2, valeur 010010 dans la base, pas d'exposant soit :
 $1 * 2^4 + 1 * 2^1 = 16 + 2 = 18$ en décimal
- $16 \# 2A3C \# =$
 $2 * 16^3 + 10 * 16^2 + 3 * 16^1 + 12 * 16^0 = 10812$
- $12 E8 = 12 * 10^8$ En l'absence de base (donc de dièse) la base est 10 implicitement !
- $6.02E-24 = 6.02 * 10^{-24}$
- $2 \# 101.01 \# E1 =$
 $(1*2^2 + 1 * 2^0 + 1*2^{-2}) * 2^1 = (5 + 0.25) * 2^1 = 10.5$

autre remarque : la valeur de la base (devant le premier dièse) est toujours en écriture entière décimale (mais pas forcément la valeur basée) et l'exposant (repéré par le E ou un e) est toujours associé à la base (mais la valeur de l'exposant s'exprime, comme pour la valeur de la base, en décimal !).

Comment formaliser ces règles ? (les corrigés sont à la fin du document page 11)

a) on distingue d'abord les 2 classes (basées ou non) d'où la **première règle** :

un littéral numérique est : soit un **littéral basé** ou soit un **littéral décimal**.

exercice : écrire le D.S. correspondant. Voir corrigé page 11.

b) un **littéral basé** est formé :

d'une base, suivie d'un dièse, suivi d'un **entier basé** (éventuellement suivi d'un point et d'un autre entier basé) suivi d'un deuxième dièse (éventuellement suivi d'un **exposant**).

exercice : écrire le D.S. correspondant.

c) un **entier basé** est une suite de **chiffres généralisés** et de soulignés (le souligné ne pouvant ni commencer ni finir l'entier basé, ni être 2 fois consécutivement).

exercice : écrire le D.S. correspondant.

d) un **chiffre généralisé** est soit un chiffre soit une lettre.

exercice : écrire le D.S. correspondant.

e) un **exposant** est formé d'un E (minuscule ou majuscule) suivi d'un entier décimal (l'entier étant éventuellement précédé d'un + ou d'un -)

exercice : écrire le D.S. correspondant.

f) un **entier** à la même définition qu'un entier basé (où l'on remplace chiffre généralisé par chiffre tout simplement).

g) un **littéral décimal** est un entier (suivi éventuellement d'un . et d'un entier) éventuellement suivi(s) d'un exposant (voir le BNF d'un entier pages 4 et 5).

Exercice : écrire le D.S. correspondant.

Fin semaine 1 ! (4 heures de cours)

Exercice n° 1 (début semaine 2)

Dans la liste ci-dessous déterminez tous les littéraux illégaux et dites pourquoi ? Pour tous les littéraux légaux distinguez les entiers et les réels. Voir corrigé page 10.

- a) 25.7 b) .7 c) 16E2 d) 16 e-2
 e) 2#1101 f) 3.14167_345678 g) 13#ABCD#
 h) e+7 i) 16#FFf# j) 1_0#1_0#e1_0
 k) 23.7e_7 l) 2#111#e-1

Exercice n° 2 (tiré de Barnes voir pages 79 à 83) :

Quelle est la valeur décimale des littéraux numériques suivants (corrigé page 10) :

- a) 16#E#E1 b) 2#11#e11
 c) 2#1.1111_1111_111#e11 d) 16#F.FF#E+2

Les opérateurs Ada sur les numériques (littéraux ou variables ; entiers ou réels)**Préliminaires**

Les **opérations prédéfinies** applicables aux types entiers et/ou réels sont :

- + et - opérateurs **binaires** (deux opérandes **entiers** ou deux opérandes **réels**)
- * deux opérandes **entiers** ou deux opérandes **réels**
- / division (idem *)
- + et - opérateurs **unaires** (un seul opérande **entier** ou **réel**)
- rem** reste de la division **entière** (opérandes **entiers** uniquement)
- mod** le modulo (opérandes **entiers**)
- abs** valeur absolue (opérateur unaire et opérande **entier** ou **réel**)
- ** exponentiation (le premier opérande est **entier** ou **réel** et le deuxième est **entier**)

Ces opérateurs sont donnés ci-dessous par ordre de priorité croissante.

+ -

* /

+ - (unaires)

mod rem

abs **

Les quotients entiers : (exemples expliqués page suivante)

- 7 / 3 donne 2
 (-7) / 3 donne -2
 7 / (-3) donne -2
 (-7) / (-3) donne 2

$A = B * Q + R$ soit :

A		B	

rem		A/B = Q	

Car si on note : Q le quotient entier de A/B, on peut écrire $A = B * Q + \text{Reste}$. (Reste se note **rem** en Ada) avec : A et B*Q **sont de même signe** et **abs (A) ≥ abs (B*Q)**

Les restes : (notés **rem**) conséquence de la définition du quotient ci-dessus

$7 \bmod 3 = 1$ c'est évident mais les autres ?

$(-7) \bmod 3 = -1$ car $(-7 / 3) * 3 + R = -7 = (-2) * 3 + R$

$7 \bmod (-3) = 1$ car $(7 / (-3)) * (-3) + R = 7 = (-2) * (-3) + R$

$(-7) \bmod (-3) = -1$ car $((-7) / (-3)) * (-3) + R = -7 = 2 * (-3) + R$

Remarques :

- Le signe du reste est toujours le même que celui du premier opérande A (le dividende)
- Mettez les parenthèses car **rem** est prioritaire sur l'opérateur – unaire (page précédente).

Modulo : noté **mod** (définition voir mathématique \approx notion de classes d'équivalence !)

$n \bmod p \equiv (n + i * p) \bmod p$ (où i est un nombre **relatif**)

exemple : $7 \bmod 3 \equiv 10 \bmod 3 \equiv 13 \bmod 3$ etc..... $\equiv 4 \bmod 3 \equiv 1 \bmod 3 \equiv -2 \bmod 3$ etc.....

Pour choisir **LE** représentant de la classe on arrête quand $(n+ip)$ est inclus dans $[0;p[$ (si $p>0$) ou $]p;0]$ (si $p<0$)

donc dans l'exemple $7 \bmod 3 \equiv 1 \bmod 3 \Rightarrow$ $7 \bmod 3 = 1$ car 1 est compris entre 0 et 3 ($[0;3[$)

$(-7) \bmod 3 = 2$ car $(-7) \bmod 3 \equiv (-4) \bmod 3 \equiv (-1) \bmod 3 \equiv 2 \bmod 3$ $0 \leq 2 < 3$

$7 \bmod (-3) = -2$ car $7 \bmod (-3) \equiv 4 \bmod (-3) \equiv 1 \bmod (-3) \equiv -2 \bmod (-3)$ $-3 < -2 \leq 0$

$(-7) \bmod (-3) = -1$ car $(-7) \bmod (-3) \equiv (-4) \bmod (-3) \equiv (-1) \bmod (-3)$ $-3 < -1 \leq 0$

Remarques :

- mettez bien les parenthèses nécessaires (**mod** est prioritaire sur l'opérateur - unaire)
- le signe du **mod** est celui du 2^{ème} opérande
- **mod** est différent de **rem** (**sauf** quand les opérandes **ont le même signe**)
- Habitué du Pascal, attention : **mod** (en Ada) n'est pas l'opérateur du reste !

Exercice n° 3

Compte tenu des déclarations suivantes évaluer les expressions ci dessous (**il y a des pièges !**)

```
I   : constant := 7 ;
J   : constant := -5 ;
K   : constant := 3 ;
```

- a) $I * J * K$ b) $I / K * K$ c) $I / J / K$
d) $J + 2 \bmod I$ e) $-J \bmod 3$ f) $-J \bmod 3$
g) $J + 2 \bmod I$ h) $K ** K ** K$

Corrigés des exercices

Exercice n° 1 :

25.7	oui	et c'est un réel
.7	non	(il manque l'entier avant le point)
16 E2	oui	et c'est un entier
16 e-2	non	(pas d'exposant négatif pour un entier!). Ce n'est pas non plus un réel!
2 # 1101	non	(il manque le # de la fin)
3.14157_345678	oui	et c'est un réel
13 # ABCD #	non	(pas de « chiffre » D en base 13!)
e+7	non	(il y a l'exposant mais pas « la mantisse »)
16 #FFf #	oui	et c'est un entier
1_0 # 1_0 # e 1_0	oui	c'est un entier peu lisible! Qui vaut $10\#10\#E10 = 10^{11}$
23.7 E_7	non	(il ne faut pas de _ au début d'un entier décimal)
2 # 111 # e-1	non	(comme pour 16 e-2)

Exercice n° 2 :

$$16 \#E\#E1 = 14 * 16^1 = 14*16 = 224$$

$$2 \#11\#e11 = 3 * 2^{11} = 3 * 2048 = 6144$$

$$2 \#1.1111_1111_111\# e11 = 111111111111 \text{ (base 2)} = 1000000000000_{(2)} - 1 = 2^{12} - 1 = 4096 - 1 = 4095.0$$

$$16 \#F.FF\# E+2 = FFF \text{ (base 16)} = 1000_{(16)} - 1 = 16^3 - 1 = 4095.0$$

Exercice n° 3 :

$$I * J * K = 7 * (-5) * 3 = -105$$

$$I / K * K = 7/3 * 3 = 2 * 3 = 6 \neq I!$$

$$I / J / K = 7 / (-5) / 3 = (-1) / 3 = 0$$

$$J + 2 \text{ mod } I = -5 + 2 \text{ mod } 7 = -5 + 2 = -3$$

$$-J \text{ mod } 3 = -(J \text{ mod } 3) = -(-5 \text{ mod } 3) = -(+1) = -1$$

$$-J \text{ rem } 3 = -(J \text{ rem } 3) = -(-5 \text{ rem } 3) = -(-2) = 2 \text{ à comparer avec } -J \text{ mod } 3 !!!$$

$$J + 2 \text{ rem } I = -5 + 2 \text{ rem } 7 = -3 \text{ à comparer avec } J + 2 \text{ mod } I !!!$$

$K ** K ** K$ est illégal !!! (l'explication sera donnée au cours n°13 !). $(K ** K) ** K$ est correct !

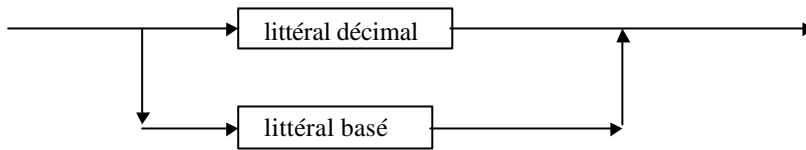
Remarques :

On trouvera en page 12 le « source » d'un programme qui peut permettre de tester les résultats des 3 exercices ci-dessus. Ce programme peut être lu afin de se mettre dans le bain d'un programme simple Ada. A revoir !

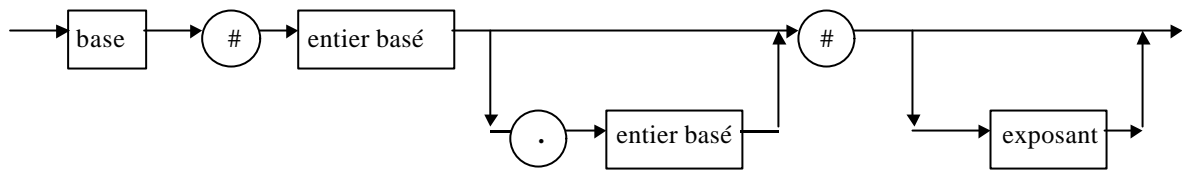
Importante : Le caractère _ est **non significatif dans un littéral** nombre (1998 et 1_998 sont équivalents) mais **il n'en est pas de même dans un identificateur** Ada ainsi NB_DE_L et NBDE_L **sont différents** !

Corrigé des D.S.

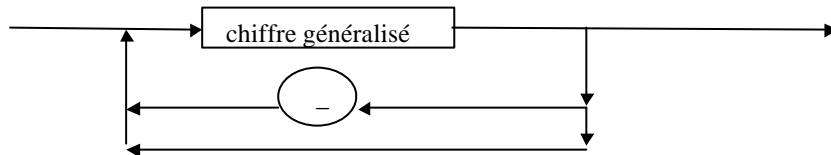
Littéral numérique Ada :



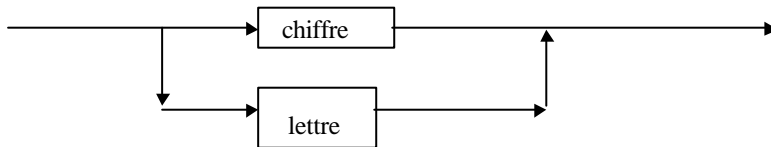
Littéral basé :



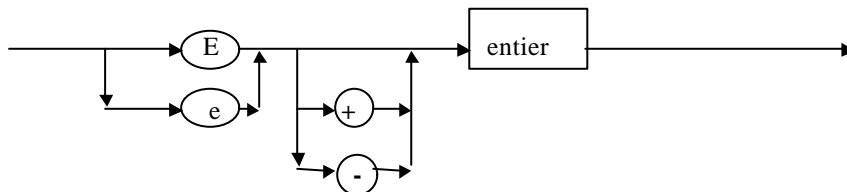
entier basé :



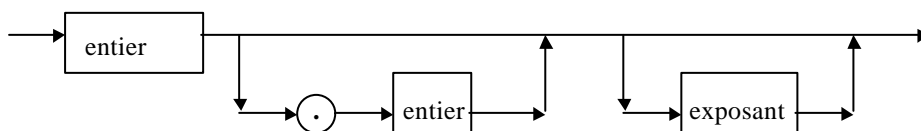
chiffre généralisé :



exposant :



littéral décimal :



Exercice : réécrire ces D.S. en BNF.

Exemple de programme simple Ada permettant de tester les trois exercices de la page 8

```

-- fichier LITTERAU.ADB
-- pour tester le cours sur littéraux (exercice pages 8 et 9)
-- D. Feneuille septembre 98
with P_E_SORTIE;
procedure LITTERAU is
use P_E_SORTIE;

  I : constant := 7;
  J : constant := -5;
  K : constant := 3;

  ENTIER : INTEGER;
  REEL   : FLOAT;
begin
  -- la séquence de deux lignes suivante :
  -- ECRIRE (ENTIER); ou ECRIRE (REEL);
  -- A_LA_LIGNE;
  -- est à ajouter après chaque instruction valide!
  -- en ôtant les deux - (marque de commentaire!)
  -- on retrouvera les valeurs ajoutées
  ENTIER := 7#123#E2; -- 3234
  ENTIER := 2#01_0010#; -- 18
  ENTIER := 16#2A3C#; --10812
  --ENTIER := 16E-2; erreur syntaxe
  --ENTIER := 13#ABCD#; erreur D
  --ENTIER := E+7; erreur
  ENTIER := 16#E#E1;--224
  ENTIER := 2#11#e11;--6144
  REEL := 6.02E-24; --0.0
  REEL := 2#101.01#E1; --10.5
  --REEL := .7; erreur syntaxe
  --REEL := 16E-2; erreur
  --REEL := 23.7E_7; erreur
  REEL := 2#1.1111_1111_111#E11; -- 4095.0
  REEL := 16#F.FF#E+2; -- 4095.0
  ECRIRE (I*J*K); -- -105
  A_LA_LIGNE;
  ECRIRE (I/K*K); -- 6
  A_LA_LIGNE;
  ECRIRE (I/J/K); -- 0
  A_LA_LIGNE;
  ECRIRE (J+2 mod I); -- -3
  A_LA_LIGNE;
  ECRIRE (-J mod 3); -- -1
  A_LA_LIGNE;
  ECRIRE (-J rem 3); -- 2
  A_LA_LIGNE;
  ECRIRE (J+2 rem I); -- -3
  A_LA_LIGNE;
end LITTERAU;

```

Source disponible dans le
répertoire fendan/corrige

Le cours n°1 (généralités II) se termine encore avec quelques pages d'exemples de programmes Ada (en relation avec les TD TP sur l'algorithmique).

Compléments cours n° 1 (Généralités III)

Pour terminer ces six heures de cours nous allons commenter quelques codages Ada d'algorithmes connus (puisque découverts au cours des premiers TD de la semaine passée). Il s'agit évidemment d'un survol assez informel et bien sûr tous les concepts, vus rapidement, seront repris en détail dans les cours à venir.

Exemple n° 1 (« tiré » du polycopié algorithmique TD n°0)

```
with P_MOUCHERON.IMP;
procedure TS_MOUCHERON0 is
use P_MOUCHERON.IMP;
begin
  ECRIRE ("Essai de MOUCHERON");
  A_LA_LIGNE;
  A_LA_LIGNE;
  MONTRER_RUBAN;
  A_LA_LIGNE;
  RESET;
  MONTRER_RUBAN;
  A_LA_LIGNE;
  AUTRE_RUBAN;
  MONTRER_RUBAN;
end TS_MOUCHERON0;
```

Evoque la ressource (composant logiciel) nommée P_MOUCHERON.IMP qui propose des services ou outils tels que : ECRIRE, RESET, etc. On parle avec **with** de clause de contexte

Traduction de l'algorithme résolvant le problème

Nom du programme
TS_MOUCHERON0

On remarque un ensemble de mots : soit des **mots réservés** (ici en minuscule gras) et des **identificateurs** choisis parmi les spécifications de la ressource P_MOUCHERON.IMP. Les points virgules **terminent** chaque action. Les mots réservés **begin** et **end** encadrent l'ensemble des actions mises en œuvre. Cette procédure se suffit à elle-même elle peut devenir un **programme exécutable**. Le **use** évite le préfixage (à revoir).

Exemple n° 2 (« tiré » du polycopié algorithmique TD n°1)

```
procedure NB_CAR (...) is
begin
  COMPTEUR := 0;
  loop
    exit when FIN_RUBAN;
    LIRE;
    COMPTEUR := COMPTEUR + 1; -- incrémentation du compteur
  end loop; ...
  ECRIRE_COMPTEUR; -- pour voir à l'écran le contenu du compteur
end NB_CAR;
```

structure
itérative

C'est toujours la traduction Ada de l'algorithme vu en TD. On remarque d'autres mots réservés **loop**, **exit when**. Ici, cette brique (elle aussi une procédure) n'est **pas indépendante** ; c'est un composant de P_MOUCHERON.EVE.MOUCHERON1 qui accède aux entités : Compteur, Fin_Ruban, Lire et Ecrire_Compteur.

Exemple n° 3 (« tiré » du polycopié algorithmique TD n°1). Même remarques que l'exemple 2.

```
procedure NB_DE_L(....) is
begin
  COMPTEUR := 0;
  loop
    exit when FIN_RUBAN;
    LIRE;
    if (CARCOU = 'L') or (CARCOU = 'l')
    then COMPTEUR := COMPTEUR + 1;
    end if;
  end loop;...
  ECRIRE_COMPTEUR;
end NB_DE_L;
```

structure
alternative

Exemple n° 4 (« tiré » du polycopié algorithmique TD n°1)

with P_MOUCHERON.IMP.MOUCHERON1;

procedure TS_MOUCHERON1 **is**

use P_MOUCHERON.IMP, P_MOUCHERON.IMP.MOUCHERON1;

begin

ECRIRE ("test de NB_CAR, NB_DE_L :");

 A_LA_LIGNE;

loop ...

ECRIRE ("édition du ruban :");

 A_LA_LIGNE;

MONTRER_RUBAN;

 A_LA_LIGNE;

RESET;

ECRIRE ("nombre de caractères : ");

 NB_CAR;

 A_LA_LIGNE;

RESET;

ECRIRE ("nombre de L : ");

 NB_DE_L;

 A_LA_LIGNE;

AUTRE_RUBAN;

end loop;

end TS_ MOUCHERON1;

on « s'appuie » sur la ressource
MOUCHERON1
« petite-fille » de P_MOUCHERON

pour tester en « boucle »
le programme. L'arrêt se
fait sur l'absence de nom
de ruban dans le fichier
testtp1.txt

Cette procédure (indépendante) identifiée TS_MOUCHERON1 utilise le composant logiciel MOUCHERON1 (lui-même composé entre autres des nouvelles briques NB_CAR et NB_DE_L qu'utilise TS_MOUCHERON1).

Exemple n° 5 (« tiré » du polycopié algorithmique TD n°2).

procedure Nb_De_Carac (Car : **in** Character) **is**

begin

 Compteur := 0;

loop

exit when Fin_Ruban;

 Lire;

if Carcou = Car

then Compteur := Compteur + 1;

end if;

end loop;

 Ecrire_Compteur;

end Nb_De_Carac;

Paramètre formel

Il s'agit d'une brique **réutilisable** d'un composant logiciel. Mais, comme on l'a vu en algorithmique, la procédure est « formelle » on dira « paramétrée ». Le paramètre (entre parenthèses n'est pas concret il ne correspond à aucun identificateur vrai). A revoir en semaine 4 (cours n° 5 « sous-programme »).

Exemple n° 6. Notion de **paquetage** (composant Ada privilégié).

Il s'agit de survoler la partie **spécifications** d'un composant logiciel que nous utiliserons pendant quelques semaines. En Ada on peut agréablement modéliser un composant avec ce que l'on appelle un **paquetage** (cette notion sera bien approfondie plus tard semaine 5). Pour aujourd'hui nous allons uniquement nous intéresser à la partie « **contrat** » ou spécifications. En effet un utilisateur potentiel d'une ressource n'a besoin de connaître que « le mode d'emploi » et, en aucune façon, sa réalisation (le : « comment c'est fait » n'a pas d'intérêt). Ici on va découvrir des outils d'entrées-sorties (Lire et Ecrire **surchargés**) des types prédéfinis Ada (Character, Integer, Float, String). Il s'agit d'un extrait d'un document plus conséquent lequel sera disponible dans son intégralité en TD Ada (à copier). On en fait une lecture commentée seulement.

```

with Ada.Text_io, Ada.Strings.Unbounded;
package P_E_Sortie is
  use Ada.Text_Io, Ada.Strings.Unbounded;
  =====
  -- Procédures d'entrées/sorties sur les entiers prédéfinis
  =====

  procedure Ecrire (
    L_Entier : in Integer );
  -- permet de visualiser un INTEGER à partir de la position
  -- initiale du curseur. Si l'on souhaite aller à la ligne ensuite
  -- il faut utiliser en plus la procédure A_LA_LIGNE

  procedure Lire (
    L_Entier : out Integer );
  -- permet d'acquérir un INTEGER en base quelconque (2 à 16) avec
  -- validation (fait recommencer jusqu'à ce la valeur soit valable)

  =====
  -- Procédures d'entrées/sorties sur les flottants prédéfinis
  =====

  procedure Ecrire (
    Le_Reel : in Float );
  -- permet de visualiser un FLOAT sur la ligne à partir de la position
  -- initiale du curseur. Si l'on souhaite aller à la ligne ensuite
  -- il faut utiliser en plus la procédure A_LA_LIGNE.
  -- Le réel est écrit sous la forme:
  -- signe, partie entière, point décimal et partie décimale
  -- le point décimal est toujours présent,
  -- le signe + est remplacé par un espace.
  -- Quant aux deux champs numériques ils suppriment les
  -- zéros non significatifs.

  procedure Lire (
    Le_Reel : out Float );
  -- permet d'acquérir un FLOAT dans une base quelconque (2 à 16) avec
  -- validation (on recommence jusqu'à ce que la valeur soit valable).
  -- Le point décimal est obligatoire à la saisie.

  =====
  -- Procédures d'entrées/sorties sur les caractères
  =====

  procedure Ecrire (
    Carac : Character );
  -- permet de visualiser un caractère à partir de la position
  -- initiale du curseur. Si l'on souhaite aller à la ligne ensuite
  -- utiliser en plus la procédure A_LA_LIGNE

  procedure Lire (
    Carac : out Character );
  -- permet d'acquérir un caractère unique. Lecture tamponnée c'est-
  -- à-dire que l'on peut effacer le caractère saisi d'où nécessité de
  -- terminer la saisie par un RC (appui sur la touche entrée). Le
  -- tampon est purgé après la prise en compte de ce caractère.

```



```

=====
-- Procédures d'entrées/sorties sur les chaînes de caractères
-- On pourra lui préférer plus tard les paquetages prédéfinis
=====

procedure Ecrire (
    La_Chaine : String );
-- permet de visualiser une chaîne de caractères à partir
-- de la position initiale du curseur.

procedure Lire (
    La_Chaine :    out String );
-- permet de saisir une frappe de caractères terminée par entrée
-- si le nombre de caractères est inférieur à la dimension du STRING
-- passé en paramètre le complément est initialisé à espace.
-- Le tampon est purgé.

procedure Lire (
    La_Chaine :    out Unbounded_String );
-- permet de saisir une frappe de caractères terminée par entrée
-- il n'y a pas de limite, ni problème de remplissage

=====
-- Procédures vider clavier et saut à la ligne sur l'écran
=====

procedure Vider_Tampon;
-- peu utile dans le contexte proposé ici puisque toutes les saisies
-- sont purgées. Permet cependant d'attendre la frappe de la touche
-- entrée. C'est une pause (voir ci-dessous) sans message.
procedure A_La_Ligne (
    Nombre : Positive_Count := 1 ) renames New_Line;
-- sans commentaire
procedure Pause;
-- affiche le message: "appuyez sur Entrée pour continuer"
-- et attend la frappe sur la touche entrée.

=====
-- Transformation de FLOAT ou d'INTEGER en Chaîne de caractères
=====

function Image (
    L_Entier : in    Integer ) return String;
-- c'est une "redéfinition" de l'attribut c'est-à-dire INTEGER'IMAGE

function Image (
    Le_Reel : in    Float ) return String;
-- permet une amélioration de l'attribut FLOAT'IMAGE
-- l'image est celle définie dans la procédure ECRIRE avec FLOAT.
-- permet des écritures d'images d'objets de type différent ainsi:
-- ECRIRE(IMAGE(...)&"...."&IMAGE(...)&"..." etc.);

```

```

=====
-- Spécial type discrets (pour les types énumératifs et entiers)
=====
-- Pour éviter d'avoir à instancier (trop tôt!) les sous-paquetages
-- ADA.TEXT_IO.ENUMERATION_IO ou ADA.TEXT_IO.INTEGER_IO
-- (l'exercice n'est pas difficile mais mythique !)
-- on copiera la séquence suivante en adaptant les identificateurs
-- T_DISCRET et DISCRET au type discret et à la variable de ce type
--
-- loop
--   declare
--     CHAI : UNBOUNDED_STRING ;
--   begin
--     LIRE(CHAI);
--     DISCRET := T_DISCRET'VALUE(TO_STRING(CHAIN));
--     exit;
--     exception when others =>
--       null; -- ou message ECRIRE("xxxxx");
--   end;
-- end loop;
end P_E_Sortie;

```

Mots réservés Ada

abort	declare	generic	of	select
abs	delay	goto	or	separate
abstract*	delta		others	subtype
accept	digits	if	out	
access	do	in	package	tagged*
aliased*		is	pragma	task
all	else		private	terminate
and	elsif	limited	procedure	then
array	end	loop	protected*	type
at	entry		raise	
	exception		range	use
begin	exit	mod	record	until*
body			rem	when
		new	renames	while
case	for	not	requeue*	with
constant	function	null	return	
			reverse	xor

Nous allons essayer dans tous nos exemples Ada de représenter ces mots réservés en minuscules et en gras ; suivant en cela le manuel de référence. Les autres identificateurs seront proposés soit en MAJUSCULES soit en notation mixte première lettre (ou lettre suivant un _) en majuscule (le reformateur fait cela).

Remarque :

Il se trouve que les 3 mots : **delta**, **digits** et **range** sont aussi des attributs (concepts à revoir) et devraient quand ils sont attributs¹ (et à leur place) être écrits : **Delta**, **Digits** et **Range** c'est-à-dire en notation mixte (subtilité à dominer plus tard !). En fait, ceci a lieu quand ils sont précédés d'une apostrophe.

Exemples : pour Range et range

```
....T_IND'Range
```

```
....Integer range 1..10;
```

Remarques :

La nouvelle norme Ada (Ada 95) ajoute les 6 mots réservés (marqués d'une *) : **abstract**, **aliased**, **protected**, **requeue**, **tagged**, **until** et au moins un attribut important à connaître: **Class** (nous y reviendrons semaine 7)

A lire (et à relire) :

Rosen (dans son livre pages 22 à 39 livre signalé au cours n°1) réussit le tour de force, en très peu de pages, à faire un joli survol du langage Ada (présentation **à voir absolument** !). Voir aussi son site : <http://perso.wanadoo.fr/adalog/> à visiter de haut en bas !

¹ La notion d'attributs Ada est vue au cours n° 2 (2 heures) qui suit immédiatement cette séance de 2 heures aussi.

Cours Ada n°1 (6 heures) testez vos connaissances

Utile pour bien réaliser les QCM Machine et les QCM papier (partiels !)

Qu'est-ce qu'un langage de programmation ?

Citez en quelques uns.

Qu'est-ce qu'un programme ?

Qu'est-ce que la compilation ?

Qu'est-ce que l'édition de liens ?

Qui était Ada comtesse de Lovelace ?

A quel organisme doit-on la création du langage Ada ?

Quelles sont les appellations des deux versions du langage Ada ?

Dans quelles types d'applications Ada est-il beaucoup employé ?

Pourquoi doit-on normaliser un langage ?

Quel nom donne-t-on au document qui présente le langage normalisé ?

« Ada : vecteur du génie logiciel » ça veut dire quoi ?

Qu'est-ce qu'un composant logiciel ?

Un littéral chaîne de caractères peut-il être vide ?

Un littéral caractère peut-il être vide ?

Comment dénote-t-on un guillemet dans un littéral chaîne de caractères ?

Sachez écrire (en BNF et en DS) la définition d'un identificateur Ada.

Citez les 6 opérateurs de relation Ada

Citez les 4 opérateurs multiplicatifs Ada

Citez les 6 types prédéfinis Ada.

Comment écrit-on un commentaire en Ada ?

Qu'est-ce qu'un littéral numérique ?

Qu'est-ce qui différencie un littéral entier d'un littéral réel ?

Qu'est-ce qu'un littéral numérique basé ?

Quelle est la base implicite d'un littéral non basé ?

Qu'est-ce qu'un opérateur binaire (et un opérateur unaire) ?

Citez les opérateurs unaires agissant sur les numériques

Et les opérateurs binaires.

Dans les deux questions précédentes précisez sur quels sortes de numériques ils agissent et quel est le type du résultat.

Priorités des opérateurs numériques ?

rem et **mod** donnent des résultats différents (sauf ?)

A quoi servent les paquetages `P_E_Sortie.ads` et `P_E_Sortie.adb`

Fichiers à éditer :

- Ceux des paquetages qui servent à réaliser les exercices d'algorithmique.
- Les paquetages `P_E_Sortie.ads` et `P_E_Sortie.adb` (pour voir la réalisation)
- Les fichiers annoncés page 5 cours n°1 (généralités I).

Visitez les sites suggérés dans le fichier `Ada_et_le_Web.html`.

Cours n° 2 Ada type scalaire (2 bonnes heures !)

Thème : le typage Ada et quelques types (prédéfinis ou construits).

Le typage et pourquoi typer?

Typer les objets que va utiliser un algorithme c'est **regrouper** dans une même enveloppe ou dans un même ensemble (bien identifié) les objets ayant **un lien « conceptuel » entre eux**. Il s'agit d'une **partition** au sens mathématique du terme (i.e. pas de mélange). Ainsi les objets de ce type auront les mêmes propriétés, les mêmes opérateurs, la même « plage » de valeur etc. Ce faisant, c'est-à-dire **en typant le plus possible**, on **évitera des erreurs** dites « conceptuelles » (que vérifiera d'ailleurs la phase de compilation) et ces erreurs seront découvertes très tôt dans le processus de mise au point. Ada (comme tout langage) propose des types prédéfinis mais Ada offre aussi des possibilités pour construire (soit même) ses propres types (types construits) **opération à privilégier le plus possible**. Cette technique, sans être impérative, est une composante d'un travail bien fait et rigoureux et **participe aux techniques du génie logiciel**. On privilégiera donc cette approche du typage systématique le plus souvent possible (mais Ada, en fait, comme on le verra, nous y contraint souvent !).

Ada est un langage dit **fortement typé**. Ce qui veut dire que :

- Une variable, une constante ou un littéral appartiennent à **un type et à un seul** (c'est le principe même de **l'instanciation**).
- Toute opération sur une instance d'un type (c'est-à-dire une variable, une constante ou un littéral de ce type) doit appartenir à l'ensemble des opérations **connues** pour les instances de ce type.
- **Aucune conversion implicite** de type n'est effectuée sur une valeur. Même les opérations arithmétiques ne sont définies qu'entre **entiers de même type**, ou qu'entre **réels de même type** (excepté **).

Par exemple avec les déclarations suivantes :

```
IND : INTEGER;    -- IND est de type entier prédéfini
REL : FLOAT;     -- REL est de type réel prédéfini
```

On **ne peut en aucune manière** effectuer directement l'opération `REL + IND` ou écrire `REL := IND` ;

Si on le souhaite vraiment il faut l'exprimer **haut et clair** et faire une **conversion explicite** du type

```
INTEGER(REL) + IND ou IND := INTEGER(REL); -- attention arrondi !
ou REL + FLOAT(IND) ou REL := FLOAT(IND);
```

De même avec la variable `COMPTEUR` déclarée (dans la ressource `P_MOUCHERON`) cf. Algo ainsi :

```
COMPTEUR : T_COMPTEUR; -- à valeur entre 0 et 3000
```

On ne peut pas « mêler » `COMPTEUR` et `IND` (pourtant tout deux de type entier (mais ce n'est pas le même !)).

La conversion d'un réel en entier avec l'écriture `INTEGER(REL)` procède par **arrondi** et non par troncature. L'arrondi se fait vers **l'entier le plus proche**. Pour les valeurs exactement au milieu (c'est-à-dire ayant une décimale .5 exacte) l'arrondi se fait vers l'entier **en s'éloignant de zéro**. Ainsi 1.5 est converti en 2 et -1.5 est converti en -2.

En résumé, on retiendra qu'un type est caractérisé par :

- Un ensemble de valeurs que peut prendre toute instance du type (c'est-à-dire tout objet déclaré de ce type).
- Un ensemble d'opérations (décrites à l'aide d'opérateurs) applicables aux instances du type.
- Eventuellement un ensemble de caractéristiques appelées « attributs »¹. Ces attributs sont en général des opérations qui portent sur les propriétés du type lui-même et non pas, comme les opérateurs, sur les instances du type. Cette notion d'attribut n'est pas évidente ! (nous y reviendrons).

¹ Les lecteurs avertis de la programmation « moderne » auront garde de ne pas confondre ce terme « attribut Ada » avec celui d'attribut (homonyme !) utilisé en programmation « objet » (avec C++ ou Eiffel par exemple).

Les types prédéfinis puis les types construits :

Les types prédéfinis :

Nous avons vu au cours n°1 (généralités II) avec les D.S. les identificateurs de 6 types prédéfinis.

`Integer`, `Float`, `Boolean`, `Character`, `String` et `Duration`.

Ces 6 types prédéfinis existent respectivement pour instancier : des entiers, des réels (nombre à virgule), des booléens (deux valeurs!), des caractères, des chaînes de caractères et enfin des durées. Ces types sont définis dans la ressource (ou paquetage) `STANDARD` (qu'il est inutile d'évoquer avec `with`). Pour des raisons largement **discutées** et **démontrées** plus tard il est très **prudent de refuser d'utiliser** (le plus souvent possible) les 3 types suivants : `Integer`, `Float` et `String`. Nous essaierons de **nous y tenir**.

Le type `CHARACTER` : (nommé aussi `LATIN_1`)

L'ensemble des valeurs associées à ce type est défini dans le paquetage `STANDARD` voir dans le polycopié « paquetages officiels Ada ». Ces valeurs sont **ordonnées** donc possèdent un **rang** (de 0 à 255). Les valeurs éditables ou **graphiques** (du n° 32 au n° 126 et du n° 160 à 255) sont dénotées entre **apostrophes** (ex : 'a', 'P', 'é', ' ', '=' etc.). Les valeurs non éditables ou de **contrôle** (du n° 0 au n° 31 et du n° 127 au n° 159) sont dénotées avec leur identificateur spécifique (en général deux à trois lettres sans les apostrophes évidemment !) par exemple **bel** le caractère de rang 7 qui émet un bip sonore quand on « l'édite » à l'écran. Les identificateurs des caractères (notamment ceux dits de contrôle) sont hélas peu mnémoniques ; aussi, le paquetage `Ada.Characters.Latin_1` (à consulter dans le polycopié) déclare un ensemble de constantes permettant de nommer autrement tous les caractères (seuls les chiffres et les lettres majuscules n'ont pas de nom). Par exemple dans `Ada.Characters.Latin_1` le caractère guillemet est nommé `Quotation` sinon il se dénote ainsi "" dans `STANDARD`.

En corollaire de la relation d'ordre il existe des opérations clairement comprises grâce aux 6 opérateurs :

`=`, `/=`, `>`, `<`, `>=`, `<=` auxquels on ajoutera les opérateurs d'appartenance à un intervalle ou à un sous type : `in` et `not in`.

Exemples :

```
if (Carcou in 'A'..'K') then ...
if (Carcou = 'L') then ...
if (Carcou in T_Majuscules) then ...
if (Carcou /= ' ') then ...
```

Les attributs les plus courants sont : `PRED`, `SUCC`, `VAL` et `POS`.

Auxquels on peut ajouter : `FIRST`, `LAST`, `IMAGE`, `VALUE`, `MAX`, `MIN`, `WIDTH`, ...

Les attributs font d'abord référence au type concerné (ici `Character`) grâce à une apostrophe puis, éventuellement, ils utilisent un ou des paramètres. Exemples :

```
Carcou := Character'Succ ('o') ; -- Carcou vaut 'p'
Carcou := Character'Pred(Carcou) ; -- dépend de la valeur de Carcou
Carcou := Character'Val(32) ; -- Carcou vaut le caractère espace
Carcou := Character'First ; -- Carcou vaut le caractère de rang zéro
```

On verra d'autres exemples d'application en TD (étude du paquetage `Ada.Characters.Handling`). A noter qu'il existe aussi le type `Wide_Character` « sur ensemble » du type `Character` (mais il est ordonné de 0000 à FFFF₁₆ au lieu de 0 à 255 ou 00 à FF₁₆).

Le type BOOLEAN :

L'ensemble des valeurs associées à ce type contient seulement deux valeurs dénotées FALSE et TRUE (dans cet ordre !). On verra que c'est finalement un **exemple le plus élémentaire d'un type énumératif**. Les 6 opérateurs et les attributs **sont les mêmes** que pour le type CHARACTER avec en plus les opérateurs « logiques » :

not, **and**, **or** et **xor**. Attention, les trois derniers **ont même priorité** ! Le premier **not** a une priorité supérieure aux autres. On verra aussi **and then** et **or else** (améliorations de **and** et **or**). L'utilisation de ce type étant très évoqué en TD d'algorithmique donc nous n'y reviendrons pas.

Les types INTEGER et FLOAT :

A retenir pour le moment : les valeurs possibles de ces deux types « numériques » **dépendent de l'ordinateur** et du compilateur sur lequel vous travaillez (donc ils ne sont **pas portables donc dangereux** à utiliser). Heureusement Ada permet de construire ses propres types numériques (les entiers vus plus loin et les réels vus plus tard). On retiendra pour l'immédiat de **ne pas utiliser** le type INTEGER et, en attendant mieux, d'utiliser, quand même, FLOAT!

Les types STRING et DURATION :

Vus plus tard ! Ainsi que Wide_String.

Remarque : Ada propose dans des paquetages complémentaires deux types de chaînes de caractères (donc autre que String) très intéressants (les Unbounded_String et les Bounded_String) reprenez ces identificateurs. Ils n'appartiennent pas aux types prédéfinis mais font partie du langage Ada95. De conception et surtout d'utilisation sûres : **nous les utiliserons souvent**.

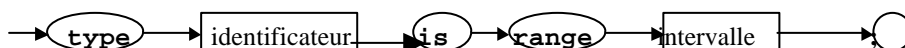
Les types construits et les constructeurs de types :

Il existe en Ada des constructeurs syntaxiques permettant de « fabriquer » ses propres types. On parle alors de types **construits** en opposition avec les 6 types prédéfinis. Il existe des constructeurs de :

types énumératifs.	Cours 2 (ce cours ci : voir pages plus loin)
types tableaux.	Cours 4
types articles.	Cours 6
types accès.	Cours 12
types privés.	Cours 7
types dérivés.	Cours 10 et numérique 1
types entiers	Cours 2 (ce cours ci : voir pages plus loin)
types réels.	Cours numérique 1 et 2
types tâches.	Cours tâches (début trimestre 2 voir planning) et aussi le type « protégé »

Construction d'un type entier signé :

La règle de construction d'un **type entier signé** est définie par le D.S. suivant :



Exemples (remarquez l'utilisation des trois mots réservés Ada : **type**, **is** et **range**) :

```

type T_TEMPERATURE is range -78..123;
type T_ETAGE is range -5..12;
type T_MONTANT is range 0..2_000_000;
type T_COMPTEUR is range 0..3000; -- utilisé en TD-TP Algo !
  
```

On rappelle que ces 4 types d'entiers construits sont **incompatibles** entre eux ! Et c'est heureux !

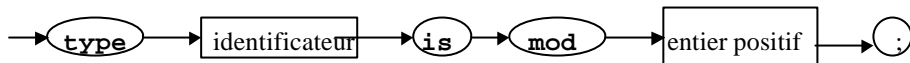
Les opérateurs ont été vus dans le cours précédent (D.S. et littéraux numériques):

`+, -, *, /, mod, rem, abs, **`

Les **attributs** des types entiers sont les mêmes que ceux du type Character et plus généralement sont ceux des types discrets (voir plus loin) mot générique qui regroupe des types ayant des propriétés identiques. Nous ne nous étendrons pas sur les opérateurs connus ! Seuls **mod** et **rem** méritaient un bon développement et ceci a déjà été fait dans le cours précédent. Enfin, **abs** désigne, évidemment, la valeur absolue (bien connue !).

Construction d'un type entier modulaire :

La règle de construction d'un type entier modulaire est définie par le D.S. suivant :



Exemples (remarquez l'utilisation des trois mots réservés Ada : **type**, **is** et **mod**) :

```

type T_Heure is mod 24; -- valeurs entre 0 et 23
type T_Octet is mod 256; -- valeurs entre 0 et 255
  
```

La **différence essentielle** par rapport au type entier signé est l'arithmétique (modulo) ainsi si on a :

```

L_Heure : T_Heure := 23 ;
Puis L_Heure := L_Heure + 1 ; -- L_Heure vaut alors 0
  
```

Quelques problèmes à dominer :

```

L_Heure := 25 ;
L_Heure := 24 ;
  
```

sont des écritures incorrectes (« not in range » dit le compilateur !) car les valeurs littérales admises vont de 0 à 23 d'après la définition ci dessus (mod 24). Donc 24 et 25 ne sont pas dans l'intervalle.

Mais :

```

L_Heure := -2 ;
  
```

est une écriture **correcte** car le - ne fait pas partie de la définition c'est un opérateur unaire ! C'est équivalent à :

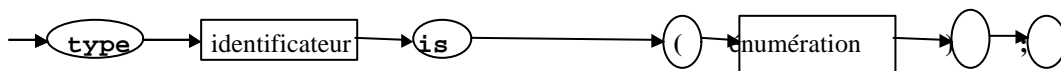
```

L_Heure := 0 - 2 ;
  
```

L'écriture est aussi équivalente à `L_Heure := 22 ;` il faut revoir les exercices du cours précédent !

Les types énumératifs construits :

La règle de construction d'un type énumératif est régit par le D.S. :



Exemples :

```

type T_COCORICO is (BLEU, BLANC, ROUGE) ;
type T_COULEUR is (VIOLET, INDIGO, BLEU, VERT, JAUNE, ORANGE, ROUGE) ;
type T_FEUX is (VERT, ORANGE, ROUGE);
type T_JOUR is (LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE);
type T_MOIS is (JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET, AOUT,
  SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE) ;
  
```


Les types énumératifs sont construits en indiquant (en énumérant !) simplement les valeurs **symboliques** (séparées par des virgules) que peuvent prendre leurs instances (**à ne pas confondre avec les littéraux chaînes de caractères** : le symbole ou identificateur BLEU est différent du littéral chaîne "BLEU").

On remarque (dans les exemples) la présence du symbole énumératif (ou littéral énumératif) ROUGE à trois endroits. On parle de **surcharge**. (Même remarque pour ORANGE, BLEU et VERT il y a encore surcharge).

Comme pour les types construits ou prédéfinis vus précédemment, le type énumératif possède des attributs.

```
T_COCORICO' PRED(ROUGE) vaut BLANC
T_COULEUR' PRED(ROUGE) vaut ORANGE sans ambiguïté!
```

L'attribut PRED, qui opère bien sur la valeur ROUGE, doit être qualifié par le type (ceci est vrai quelle que soit l'utilisation et pas seulement comme ici en cas de surcharge). Attention T_COCORICO' PRED(BLEU) entraîne une **erreur d'exécution** dite CONSTRAINT_ERROR. (A revoir en TD mais on comprend pourquoi !).

```
T_JOUR' FIRST vaut LUNDI
T_FEUX' LAST vaut ROUGE
```

S'il existe une procédure ECRIRE elle même surchargée (pour les trois types) il y a ambiguïté à écrire :
 ECRIRE (ROUGE) ; Il faut **qualifier** par exemple ainsi : ECRIRE (T_COULEUR' (ROUGE)) ;

Les déclarations des variables (ou instanciation des types) sont semblables à celles déjà étudiées :

```
DEBUT_SEMAINE, DEBUT_WEEK_END : T_JOUR;
PEINTURE : T_COULEUR;
```

Une variable pourra prendre une quelconque des valeurs du type. Par exemple :

```
DEBUT_SEMAINE := LUNDI;
```

Les symboles qui dénotent les littéraux des types énumératifs sont des identificateurs comme les autres. On peut aussi, dans un type énumératif, mêler les **caractères** (et seulement cela) et les symboles

```
type T_REPONSE is (OUI, NON, '1', 'A', AUTRE); -- pas facile! à revoir
```

Un type énumératif est ordonné, par exemple pour le type T_COULEUR on aura les relations

```
VERT < JAUNE < ORANGE < ROUGE
```

L'ordre est induit par celui de l'énumération.

Parmi les attributs définis sur ces types, IMAGE permet de **convertir** un symbole de type énumératif en la chaîne de caractères associée et VALUE est l'opération réciproque.

```
T_COULEUR' VALUE("ROUGE") sera l'identificateur ROUGE
T_COULEUR' IMAGE(ROUGE) sera la chaîne de caractères "ROUGE"
```

Autres exemples d'attributs :

```
T_COULEUR'FIRST      sera le symbole VIOLET
T_COULEUR'SUCC(ORANGE) sera le symbole ROUGE
T_COULEUR'PRED(ORANGE) sera le symbole JAUNE
T_FEUX'PRED(ORANGE) sera le symbole VERT
T_COULEUR'POS(BLEU) sera la valeur entière 2
T_COULEUR'VAL(0) sera le symbole VIOLET
```

Attention le rang (POS) du **premier élément d'un énumératif** est égal à 0. Par contre l'attribut POS sur un **type (ou un sous type) entier** est l'opérateur « identité » \Rightarrow `INTEGER'POS (L_ENTIER)` est égale à la valeur de la variable `L_ENTIER`. Exemple : `INTEGER'POS (-5)` vaut -5. A retenir!

Rappel : les types prédéfinis CHARACTER et BOOLEAN sont en fait des types énumératifs. Ainsi :

```
CHARACTER'POS('A') = 65 et
CHARACTER'VAL(65) = 'A'
```

Les types discrets : (il s'agit de la réunion des types entiers et énumératifs)

les types entiers (type **prédéfini** (INTEGER par exemple) et les types entiers **construits**)
 les types énumératifs (type **prédéfini** (comme BOOLEAN ou CHARACTER) et **construits**)

Les types scalaires : Il s'agit de la réunion des types discrets et des types réels.

Ce type scalaire est caractérisé par les 4 propriétés suivantes (notez donc que ces 4 propriétés s'appliquent à tous les types discrets vus précédemment) :

- Pour ce type sont définies l'affectation et les opérations d'égalité et d'inégalité (= et /=). Ce sont les propriétés communes de tout type dit à « affectation ».
- Une relation d'ordre est définie entre les diverses valeurs possibles.
 On dispose donc en plus :
 - des 4 opérateurs de relation : <, >, <=, >=
 - des tests d'appartenance à un intervalle ou à un sous type : **in** et **not in**
- Les types scalaires possèdent une borne inférieure et une borne supérieure que l'on peut connaître par les attributs FIRST et LAST. Exemples : `INTEGER'FIRST` et `INTEGER'LAST` ou `FLOAT'FIRST` et `FLOAT'LAST` (dans ce cas ces valeurs dépendent de l'**implémentation** numérique).
- Des contraintes d'intervalle (avec **range**) peuvent être appliquées sur les types scalaires.

Différence : les attributs vus avec les types discrets (ci dessous) s'appliquent aussi aux types scalaires **sauf deux** : VAL et POS. En effet avec les types **discrets** chaque valeur possède un rang, avec les types **réels**, la notion de rang **n'existe pas**.

Donc **ni VAL ni POS avec les réels** (seule différence à retenir).

Les **types scalaires donc aussi les types discrets** ont (entre autres) les propriétés suivantes :

- toute valeur d'un type scalaire ou discret, sauf la dernière, possède un **successeur**, SUCC
- toute valeur d'un type scalaire ou discret, sauf la première, possède un **prédécesseur**, PRED
- toute valeur (VAL) d'un type **discret** possède un rang (POS). Spécifique aux types discrets (rappel).

Résumé :

Les **attributs** Ada associés aux types **scalaires et discrets** sont (vus en TD semaines 3 et 4):

WIDTH	POS(*)	VAL(*)	SUCC	MAX
PRED	IMAGE	VALUE	BASE	MIN
FIRST	LAST	ADDRESS	SIZE	

(*) **discrets** seulement rappel!

Les sous-types :

Certaines instances d'un type donné pourraient ne prendre **qu'une partie** des valeurs appartenant à l'ensemble du type donné (limité par un **intervalle** de contrainte). Mais par contre elles utiliseraient l'ensemble des opérations définies pour ce type. Pour réaliser cela, Ada offre la **notion de sous-type**. Un sous-type caractérise un sous-ensemble de valeurs d'un type donné lui-même appelé **type de base du sous-type**. Le type de base d'un type est **le type lui-même!**

Exemples : (notez bien la différence avec les types entiers construits plus haut : présence du mot **subtype**)

```

subtype T_PETIT_ENTIERS is INTEGER range -128..+127;
subtype T_ANNEE is POSITIVE range 1_582..9_999;-- grégorien !
subtype T_NUM_MOIS is INTEGER range 1..12;
subtype T_NUM_JOUR is INTEGER range 1..31;
subtype T_MAJUSCULES is CHARACTER range 'A' .. 'Z' ;
subtype T_MINUSCULES is CHARACTER range 'a' .. 'z' ;

```

Ada prédéfinit deux sous types d'entiers signés. Les sous types NATURAL et POSITIVE. Ils sont définis ainsi dans le paquetage STANDARD (retrouvez les dans le polycopié « paquetages ... »):

```

subtype NATURAL is INTEGER range 0..INTEGER'LAST;
subtype POSITIVE is INTEGER range 1..INTEGER'LAST;

```

A noter que :

- la **déclaration d'un sous-type ne définit pas un nouveau type**.
- si l'indication de sous type ne comprend pas de contrainte (rare mais possible) alors le sous-type est identique au type. Cette particularité est parfois intéressante on en verra des exemples plus tard.

Classification des types de données. (à passer en première lecture mais à revoir souvent bien sûr)

Les types de données sont classés de la façon suivante (une **présentation arborescente** serait plus lisible) :

- Type à affectation.
 - type de données accès
 - sur sous-programmes
 - sur objets (statiques ou dynamiques)
 - type de données scalaires
 - types discrets
 - types énumératifs.
 - types entiers
 - signés
 - modulaires
 - types réels
 - types flottants
 - types fixes
 - binaires
 - décimaux
 - types de données composites
 - type tableaux
 - types articles
 - types étiquetés
- Types sans affectation
 - types tâches, types protégés
 - types privés limités
 - types fichiers.

A retenir

Opérations (attributs) sur les types discrets (CHARACTER, entiers et énumératifs) et parfois sur les **réels**.

T est un type ou un sous type (mais attention dans le cas d'un sous type il y aura des surprises!) et OBJET est une instance ou un littéral du type T.

fonctions sans paramètre :

- T'WIDTH rend la longueur maximale des images des valeurs du type T (scalaire) **ou du sous type T**.
Exemple : T_JOUR'WIDTH vaut 8 à cause de MERCREDI, VENDREDI ou DIMANCHE
- T'FIRST rend la borne inférieure du type T (scalaire) **ou du sous-type T**.
Exemples : INTEGER'FIRST vaut -2^{31} sur nos PC avec le compilateur GNAT; NATURAL'FIRST vaut 0
- T'LAST rend la borne supérieure du type T (scalaire) **ou du sous-type T**.
Exemple : T_MAJUSCULES'LAST vaut 'Z' ; INTEGER'LAST vaut $2^{31}-1 = 2_147_483_647$ avec le GNAT
- OBJET'Size renvoie le nombre de bits d'implémentation de OBJET. Ne s'applique pas sur un type mais sur une instance de type.

fonctions avec paramètre(s) :

- T'POS(OBJET) OBJET est considéré comme instance du **type de base** de T (**discret**). Rend la valeur numéro de position de OBJET (dans le type T ou le **type de base** du sous type T).

Exemple : T_MAJUSCULES'POS('A') vaut 65 et pas 0 ! On ne relativise pas par rapport au sous type !

- T'VAL(X) : X est valeur entière. Rend la valeur dans le type T (**discret**) ou le **type de base** du sous type T dont le numéro est la valeur entière X.
- T'SUCC(OBJET) : OBJET est considéré comme instance du **type de base** de T (scalaire). Rend la valeur dans le type de base de T immédiatement supérieure à celui de OBJET.
- T'PRED(OBJET) : OBJET est considéré comme instance du **type de base** de T (scalaire). Rend la valeur dans le type de base de T immédiatement inférieure à celui de OBJET.

Exemple : CHARACTER'POS(CHARACTER'PRED(CHARACTER'LAST)) vaut 254.

- T'IMAGE(OBJET) : OBJET est considéré comme instance du **type de base** de T (scalaire). Rend la chaîne de caractères identique à la valeur de OBJET.
- T'VALUE(CHAIN) : CHAIN est de type STRING, le résultat est du type de base de T (scalaire) c'est la valeur associée correspondante.
- T'MAX(OBJET1,OBJET2) est une fonction qui rend la valeur maximale des valeurs de OBJET1 et de OBJET2.
- T'MIN(OBJET1,OBJET2) est une fonction qui rend la valeur minimale des valeurs de OBJET1 et de OBJET2.

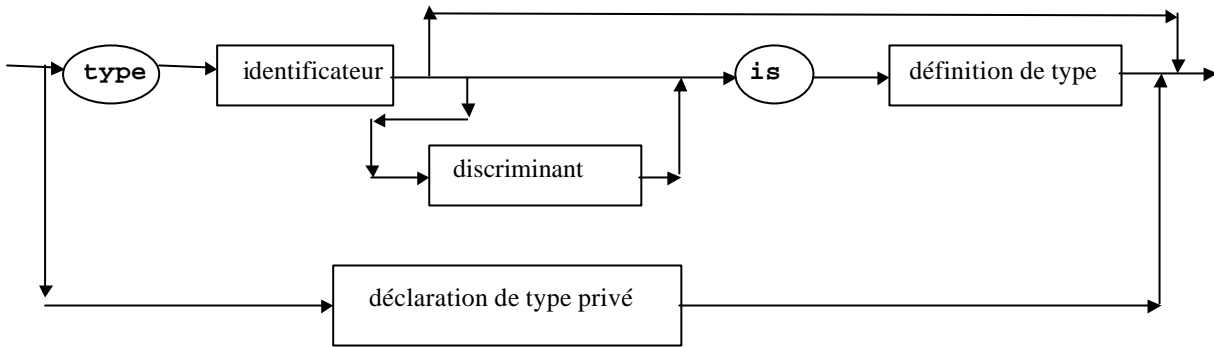
Remarques :

On doit se souvenir que VAL et POS d'une part, et d'autre part que IMAGE et VALUE sont des fonctions réciproques.

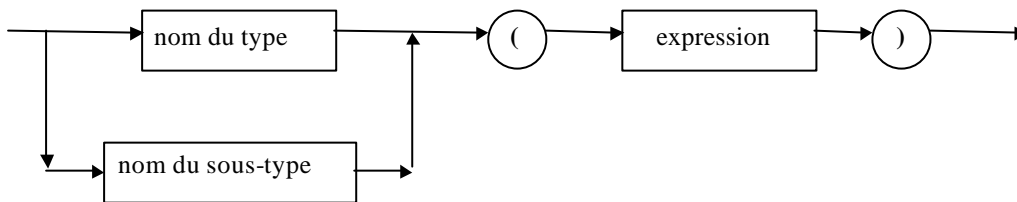
MAX et MIN opèrent sur tout type **scalaire** donc aussi sur les réels. IMAGE, VALUE, SUCC, PRED et WIDTH aussi d'ailleurs. Seuls VAL et POS **sont réservés aux types discrets** ! Rappel !

On **consultera souvent** le document « attributs prédéfinis I » (dans le **fichier attribut1.doc** sur le CDROM). Des questions sur la connaissance des attributs « tombent » souvent aux partiels ! Qu'on se le dise !

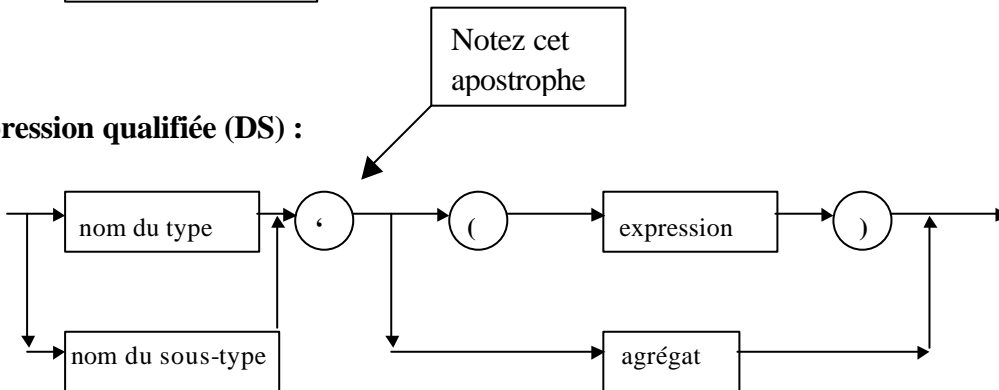
Déclaration de type (DS) :



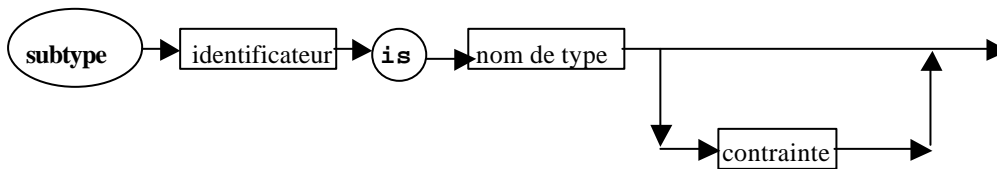
Conversion de type (DS) :



Expression qualifiée (DS) :



Déclaration de sous type (DS) :



Exercices sur le cours n°2 (ada)

Thème : Préparer sur papier des algorithmes Ada puis les saisir en temps différé, les faire tourner, en cas de problème venez aux ateliers du jeudi après midi. La notion de date sous la forme élaborée (exercice 3) : jour, n° du jour, mois et année est le centre de l'étude à réaliser (difficile!). Lecture conseillée : cahier pédagogique n°1 (fichier cahier1.doc). Travaillez prenez de la peine! Ces exercices sont des défis qui vous sont lancés, relevez les. Il serait dommage que vous ne vous y atteliez pas, car seul le travail sur machine paiera! N'attendez pas les corrigés (ils seront mis à votre disposition), les lire sera bien mais à condition de les comparer à votre solution.

Concepts : Les attributs élémentaires, l'utilisation de la ressource P_E_SORTIE, les types énumératifs et les sous types, modèle de saisie d'un type énumératif.

Travail à réaliser : (il y a deux exercices faciles mais le troisième est ardu !)

On considère les déclarations suivantes (extraites d'un programme simple) :

```

type T_JOUR   is (LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE);
type T_FRUIT  is (ORANGE, POMME, POIRE);

subtype T_JOUR_OUVRABLE is T_JOUR range LUNDI..VENDREDI;
subtype T_FIN_SEMAINE   is T_JOUR range SAMEDI..DIMANCHE;

type T_ENTIER is range 8..23000;

LE_JOUR      : T_FIN_SEMAINE := DIMANCHE;

ENTIER       : INTEGER       := -6; -- entier prédéfini
NATUREL      : NATURAL       := 6;
L_ENTIER     : T_ENTIER      := 10;

```

On remarque deux types énumératifs déclarés avec deux sous types et la déclaration d'un type entier contraint entre 8 et 23000. Quatre variables sont déclarées ensuite et initialisées.

Exercice n° 1: Evaluer les expressions (consultez "attributs prédéfinis" fichier attribut1.doc) :

Question_1 : T_JOUR 'SUCC (T_JOUR_OUVRABLE 'LAST)

Question_2 : T_JOUR_OUVRABLE 'SUCC (T_JOUR_OUVRABLE 'LAST)

Question_3 : T_FRUIT 'POS (POIRE)

Question_4 : INTEGER 'POS (ENTIER)

Question_5 : NATURAL 'POS (NATUREL)

Question_6 : T_FIN_SEMAINE 'POS (LE_JOUR)

Question_7 : T_ENTIER 'POS (L_ENTIER)

Aide : On trouvera en partie commune sur machine le programme dont est extrait cet exercice. On copiera le fichier associé (EXO1_SUP.ADB) dans sa partition et on lui fera subir le cycle classique : compilation, édition de liens puis exécution pour vérifier les questions posées (ceci constituera le corrigé).

Exercice n° 2 (à faire après le cours n°3):

Il s'agit de donner le nombre de jours d'un mois donné. On rappelle que les mois de AVRIL, JUIN, SEPTEMBRE et NOVEMBRE font 30 jours, le mois de FEVRIER fait 28 ou 29 jours et tous les autres 31!

Vous partirez des déclarations suivantes :

```
with P_E_SORTIE ;

procedure EXO2_SUP is

type T_MOIS is (JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET,
                AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE) ;

subtype T_ANNEE is POSITIVE range 1582..9999 ;

MOIS   : T_MOIS ;
ANNEE  : T_ANNEE ;
```

Aide : l'algorithme est simple à réaliser (en trois parties) dans l'ordre suivant :

- saisie du mois MOIS on utilisera le modèle de saisie d'un énumératif (cf. paquetage P_E_SORTIE) avec un bloc.
- test de la réponse : si c'est avril, juin, septembre ou novembre alors 30 sinon si c'est février alors demande de l'année (utiliser LIRE de P_E_SORTIE c'est validé) et on décide si c'est 28 ou 29 (voir cahier pédagogique n°1) et enfin si ce n'est pas février le mois fait 31 jours.
- On éditera à l'écran "ce mois fait x jours". Ce petit programme fait une bonne page de listing et fait bien le tour des if imbriqués et d'autres simples éléments algorithmiques.

Cet algorithme sera ensuite proposé en corrigé (partie commune) mais il est recommandé de s'y frotter avant ! Ne vous contentez pas de prendre et lire les corrigés ! Ceci dit c'est déjà bien car certains ne le feront pas (attention!).

Exercice n° 3 :

Attention cet exercice n'est pas du gâteau car c'est plutôt long. On commencera l'analyse de ce travail qui sera à mettre au point progressivement. **Bel investissement.**

Connaissant une date sous la forme jour, numéro du jour, mois et année il faut éditer la date complète du lendemain (voyez l'exécutable EXO3_SUP sur machine).

Exemple :

DIMANCHE 28 FEVRIER 1900 ⇒ LUNDI 1 MARS 1900

Conseils : Partagez votre travail en deux parties :

1. **prise en compte de la date initiale.** Au début, évitez cette partie en faisant une initialisation à la déclaration (version statique). Plus tard (quand la partie 2 sera écrite) faites une saisie des 4 composants d'une date **dans l'ordre suivant** : (année, mois, n° puis jour) pour faire des tests dynamiques.
2. **décision sur le lendemain** (c'est la partie finalement la plus courte).

Pour éviter des saisies fastidieuses on peut créer un fichier textes une fois pour toute contenant (à raison d'une par ligne) la saisie d'une date. Puis à l'exécution on redirige les entrées dans le fichier textes. A voir en TP.

Bon courage ! L'enfer commence ! Fin semaine 2 !

Cours n° 3 Ada : les instructions (2 heures) semaine 3

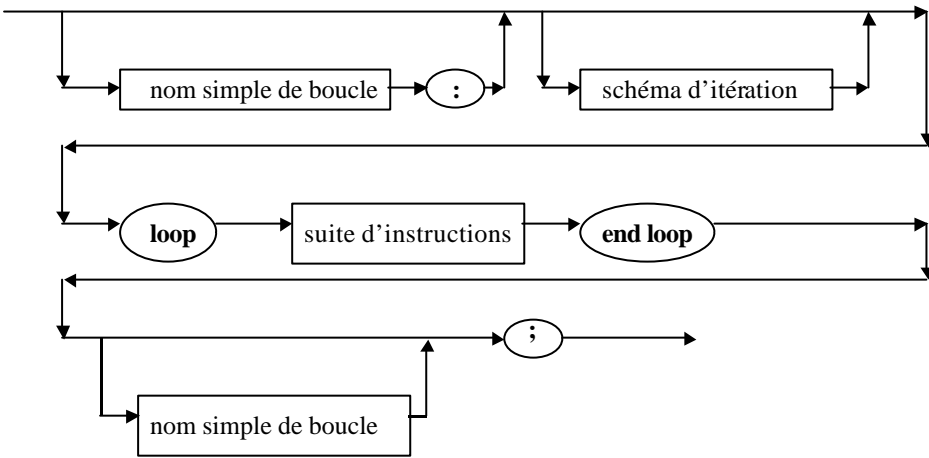
Thème : les instructions de base (répétitive, alternative, bloc, déclarations)

Certains des concepts, listés ci-dessus, ont déjà été vus ou évoqués précédemment mais succinctement. C'est le moment de concrétiser avec rigueur ces instructions fondamentales qui composent statistiquement la moitié d'un codage Ada. Malheureusement ce n'est pas pour autant que nous sommes à la moitié de notre effort!

Les instructions répétitives

Le schéma algorithmique « répéter » est, en Ada, codé à l'aide de la structure « **loop** ». Le D.S. a déjà été proposé dans la partie correspondante du cours n°1. Etudions-le en détail après l'avoir rappelé.

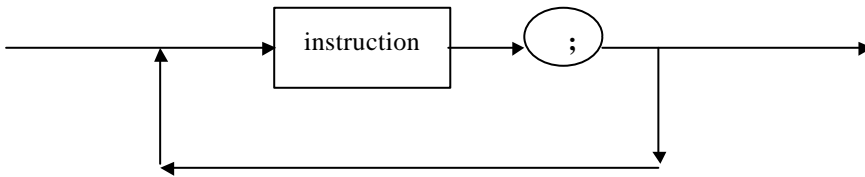
instruction loop :



Pour être complet il faut « finir » les 3 parties en « attente » dont la définition est retardée, à savoir : **nom simple de boucle, schéma d'itération et suite d'instructions.**

Nom simple de boucle est tout simplement un identificateur Ada qui permet de nommer ou caractériser la boucle. Cette technique permettra de sortir de boucles imbriquées (à revoir dans Barnes page 110). La notion de **suite d'instructions** est élémentaire. Soit le D.S. :

suite d'instructions :



La seule remarque importante est qu'il **faut au moins une instruction**, mais **exit en est une** (à la rigueur l'instruction « vide » **null** qui ne fait rien !). Exemple « simple » mais formel :

```
BOUCLE_1 : -- c'est le nom de la boucle
loop
.....
exit when (condition);
.....
end loop BOUCLE_1;
```

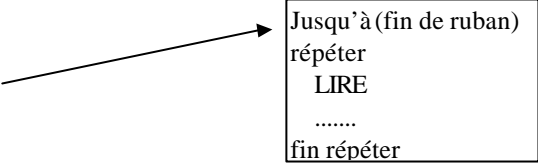
Il n'y a rien de bien nouveau si ce n'est « l'étiquetage » de la boucle avec BOUCLE_1.

Schéma d'itération :

En plus du codage de la condition de sortie, avec **exit when**, on peut utiliser le très traditionnel « tant que » (« outil » que possède tous les langages). Cette technique n'apporte rien de plus puisqu'elle n'est que la « contraposée » de la condition de sortie (le tant que exprimant la condition pour « rester dans la boucle »). Mais, le « tant que » (**while**) ne peut s'inscrire **qu'en tête du loop** (et une seule fois) perdant ainsi la « belle » propriété du **exit** qui **pouvait être partout** dans la boucle (et même à de **multiples endroits**).

Voyons un exemple (bien connu cf. Algorithmique!) :

```
loop
exit when (FIN_RUBAN);
  LIRE;
  .....
end loop;
```

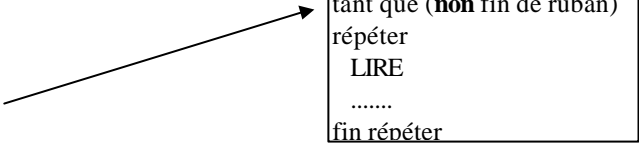


Jusqu'à (fin de ruban)
 répéter
 LIRE

 fin répéter

se code autrement avec le tant que ainsi :

```
while not (FIN_RUBAN)
loop
  LIRE;
  .....
end loop;
```



tant que (**non** fin de ruban)
 répéter
 LIRE

 fin répéter

Les concepteurs du langage Ada ont proposé le « tant que » (le **while**) pour « séduire » les programmeurs qui auraient été fâchés de ne pas retrouver leur bon vieil outil ! Ceci permet ainsi de « traduire » facilement en Ada des algorithmes écrits dans un autre langage acceptant le **while**.

Le **while** et le **exit when** qui permettent de traduire respectivement le « tant que » et le « jusqu'à » sont compatibles dans un **loop** : on peut préfixer la boucle avec un **while** et en plus ajouter des **exit when** partout ! Ces deux outils permettent de coder des structures répétitives conditionnelles. Il s'agit dans ce cas d'une itération **indéterminée** d'un énoncé. Il en va autrement si l'on connaît **a priori** le nombre de répétitions à effectuer ; par exemple « éditer » les lettres majuscules. On peut bien sûr toujours écrire :

```
CAR := 'A'; -- CAR est à déclarer ainsi CAR : CHARACTER;
loop
  ECRIRE (CAR);
  exit when (CAR = 'Z');
  CAR := CHARACTER'SUCC(CAR);
end loop;
```

Mais Ada permet (comme les autres bons langages) une « commodité » : la boucle **for**.
 Par exemple (traduisons le même algorithme) :

```
for CAR in CHARACTER range 'A'..'Z'
loop
  ECRIRE (CAR);
end loop;
```

et c'est fini. **Ada s'occupe de tout** : la **déclaration** de CAR **n'est plus à faire** (alors qu'elle devait l'être avec le **exit**), l'**initialisation** avec 'A' est faite implicitement, l'**évolution de la valeur** et l'**arrêt** sont gérés automatiquement.

On met en garde contre la tentation de ne pas typer l'intervalle. Par exemple si on écrit :

```
for CAR in 'A'..'Z'
loop
  ECRIRE (CAR);
end loop;
```

à la place de :

```
for CAR in CHARACTER range 'A'..'Z'
loop
    ECRIRE (CAR);
end loop;
```

le compilateur ne peut déterminer le type de CAR ! En effet n'oublions pas que s'il existe bien un type CHARACTER il existe aussi le type Wide_Character **sur-ensemble** de Character. L'intervalle 'A'..'Z' appartient aux deux ensembles !

Dans le même ordre d'idée voyons un exemple (cf. Barnes pages 108-109) avec surcharge d'identificateurs :

```
type T_Planètes is (Mercure, Vénus, Terre, Mars, Jupiter, Saturne,
    Uranus, Neptune, Pluton) ;
type T_Dieux is (Janus, Mars, Junon, Vestales, Vulcain, Saturne,
    Mercure, Minerve) ;
```

Remarquez que Ada permet les accents même dans les identificateurs ! (Vénus, T_Planètes)

Si l'on écrit :

```
for IND in Mars..Saturne
loop
..
end loop ;
```

Problème !

l'intervalle Mars..Saturne est ambigu le compilateur ne sait pas décider dans quel type il faut réaliser IND (**en Ada rien n'est implicite**) il faut aider le compilateur. Par exemple en **qualifiant** une des bornes :

```
for IND in T_Planètes'(Mars)..Saturne
loop
..
end loop ;
```

Qualification !

il n'y a plus d'ambiguïté

Mais le **plus lisible** est encore de typer l'intervalle ainsi :

```
for IND in T_Planètes range Mars..Saturne
loop
..
end loop ;
```

Mieux !

Si un sous type existe déjà par exemple :

```
subtype T_Explorer is T_Planètes range Mars..Saturne ;
```

alors l'écriture peut se restreindre à :

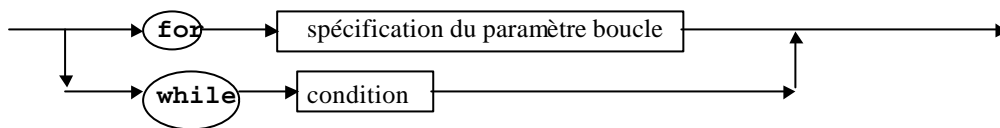
```
for IND in T_Explorer
loop
..
end loop ;
```

Variante !

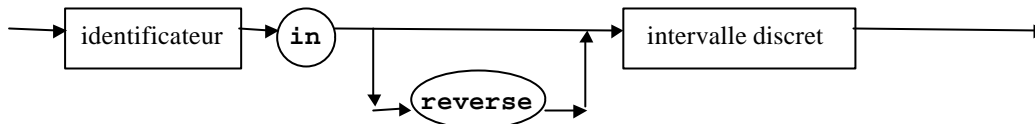
Cette structure **for** est très pratique (elle existe aussi dans de nombreux langages). Elle trouve sa pleine utilisation dans les **parcours de tableaux** notamment (nous reverrons cela dès le cours suivant et plus tard).

Le **while** et le **for** sont régis syntaxiquement par la définition « schéma d'itération » (retardée dans le D.S. général plus haut). D'où :

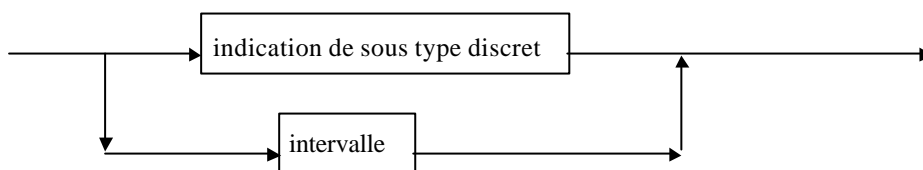
schéma d'itération :



spécification du paramètre boucle :



intervalle discret :



Exemple (notez ici le **reverse**) :

```
for CAR in reverse CHARACTER range 'a'..'r'
loop
    ECRIRE (CAR);
end loop;
```

édite la ligne suivante : rqpnmkljihgfedcba c'est-à-dire à « l'envers » (et à cause du **reverse**) ou autrement (codage plus traditionnel) sans le **for** qui était pourtant bien pratique :

```
CAR := 'r'; -- ici CAR doit être déclaré en plus et au dessus !
loop
    ECRIRE (CAR);
    exit when (CAR = 'a');
    CAR := CHARACTER' PRED(CAR);
end loop;
```

Attention (piège) : ne pas écrire (à la place du **reverse**) :

```
for CAR in CHARACTER range 'r'..'a'
```

.....

cette écriture **n'est pas fautive** (syntaxiquement!) elle n'a que l'inconvénient de présenter un intervalle vide ('r'..'a') et alors **la boucle ne se fait pas** puisque le test d'arrêt est vrai tout de suite !

Remarques :

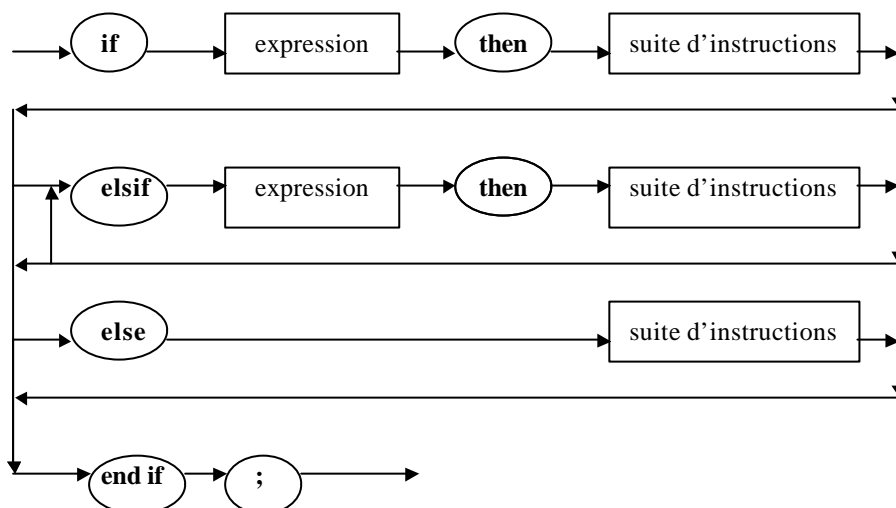
- Le paramètre de boucle nommé par l'identificateur (CAR dans notre exemple) est une variable **locale** à l'instruction de boucle **for**. Son type (en l'absence de typage explicite) est celui de l'intervalle discret. Ce paramètre ne **sera pas déclaré** dans le module dans lequel il est utilisé. **Ce paramètre ne peut être modifié mais il peut être utilisé**. Sa valeur n'est **plus accessible après la sortie** de boucle car l'objet n'existe plus. Sa « durée de vie » **se réduit** à la « portée » de la boucle (**loop...end loop**).
- Entre chaque itération le paramètre prend pour valeur le **successeur** dans l'ensemble auquel il appartient ou le prédécesseur si on utilise l'option **reverse**. Il vaut mieux parler du successeur, que du "pas de variation", qui fait trop référence aux entiers.
- On peut toujours « sortir » au milieu d'une boucle **for** avec des **exit** (sortie impérative) ou des **exit when** (sortie conditionnelle).

Les instructions alternatives

L'instruction de choix : **if**

Nous avons vu en Algorithmique (et en codage Ada associé) l'instruction de choix **if** dans sa forme simple (**if then .. end if**) et dans sa forme double (**if then .. else .. end if**). En Ada l'instruction **if** peut gérer bien plus que ces deux cas. La syntaxe est la suivante (D.S. déjà proposé dans le cours n°1) :

instruction if :



Ce qui nous intéresse ici c'est, au milieu, la « cascade » de **elsif** que l'on peut intercaler entre le **then** et le **else**. Le **elsif** signifie **else if** mais n'implique pas de **end if**. Cette forme permet de gérer efficacement la succession de plusieurs **if** avec **un seul end if**. L'expression est booléenne évidemment.

Exemple : (à voir dans l'exercice : « nombre de jours d'un mois » exercice n°2 du cours n°2 !)

```

type      T_MOIS is (JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN,
                       JUILLET, AOUT, SEPTEMBRE, NOVEMBRE, DECEMBRE);

MOIS : T_MOIS;

if MOIS = FEVRIER
then .....-- le mois fait 28 ou 29 jours
elsif (MOIS = AVRIL) or (MOIS = JUIN) or (MOIS = SEPTEMBRE)
        or (MOIS = NOVEMBRE)
then .....-- le mois fait 30 jours
else -- ce sont les autres mois restants : MARS, MAI,....
        .....-- le mois fait 31 jours
end if;
  
```

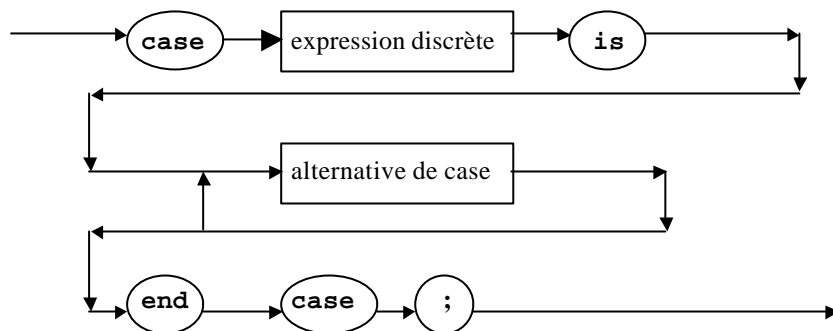
Rappel : plusieurs branches **elsif** peuvent être présentes (ce n'est pas le cas ici).

Nous verrons de nombreuses applications dans les TD et TP Ada.

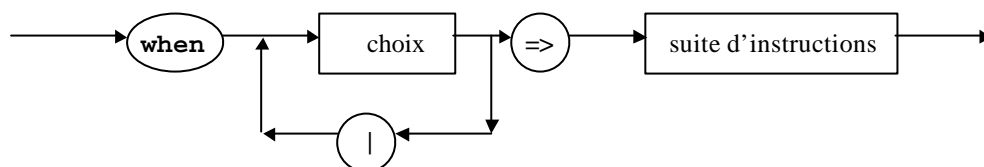
L'instruction de choix multiple (sur valeur discrète) : **case**

Voyons le D.S. :

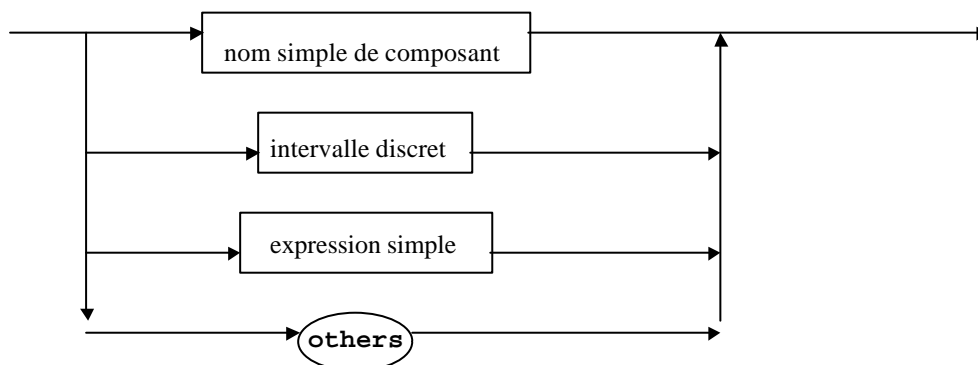
instruction **case** :



alternative de case :



choix :



Cette structure permet de choisir entre plusieurs séquences d'instructions pour plusieurs valeurs d'une expression **discrète**. C'est la valeur de l'expression **discrète** qui fixe la séquence choisie.

Une instruction choix multiple a la forme générale suivante (toutes les possibilités sont présentées) :

```

case expression is
  when valeur_1 => .....
  when valeur_2 | ..... | valeurs_n => .....
  when valeur_début..valeur_fin => .....
  when others => .....
end case ;

```

L'expression et les « choix » ou cas (conditions du **when**) doivent être **d'un même type discret**. Toutes les valeurs possibles de l'expression doivent figurer **une fois et une seule**. Le choix **others** doit être utilisé dans la dernière « branche » pour couvrir toutes les valeurs qui n'ont pas été données dans les alternatives précédentes. Si toutes les valeurs ont été examinées le choix **others** n'est pas obligatoire. Si certains choix ne nécessitent aucune instruction, l'instruction **null** doit être utilisée. Rappelons que le type **discret exclut les réels**; ne pas confondre le type discret avec le type scalaire (qui regroupe type réels et type discret).

Exemple :

```

type T_MENTION is (MAUVAIS, PASSABLE, ASSEZ_BIEN, BIEN, TRES_BIEN);

NOTE          : INTEGER; -- déclaration pas terrible.
LA_MENTION   : T_MENTION;

case NOTE is
  when 0 | 1 | 2 | 3 | 4 => LA_MENTION := MAUVAIS;
  when 5 | 6 | 7 | 8   => LA_MENTION := PASSABLE;
  when 9..12          => LA_MENTION := ASSEZ_BIEN;
  when 13 | 14 | 15   => LA_MENTION := BIEN;
  when 16..20        => LA_MENTION := TRES_BIEN;
  when others       => null; -- obligatoire car NOTE est
                                -- déclarée INTEGER!

end case;

```

Autre exemple (plus intelligent !) :

```

subtype T_UNE_NOTE is INTEGER range 0..20; -- c'est mieux

LA_NOTE      : T_UNE_NOTE;
LA_MENTION   : T_MENTION;

case LA_NOTE is
  when 0..4      => LA_MENTION := MAUVAIS;
  when 5..8      => LA_MENTION := PASSABLE;
  when 9..12     => LA_MENTION := ASSEZ_BIEN;
  when 13..15   => LA_MENTION := BIEN;
  when 16..20   => LA_MENTION := TRES_BIEN;

end case;

```

La branche **others** n'est maintenant pas utilisée, car toutes les valeurs du type énumératif T_UNE_NOTE sont envisagées. Ce ne serait pas une faute de laisser le **others** (mais il ne sert à rien !).

L'instruction « bloc » :

Voyons un bon exemple connu (lecture validée d'un type discret cf. Paquetage P_E_SORTIE) :

```

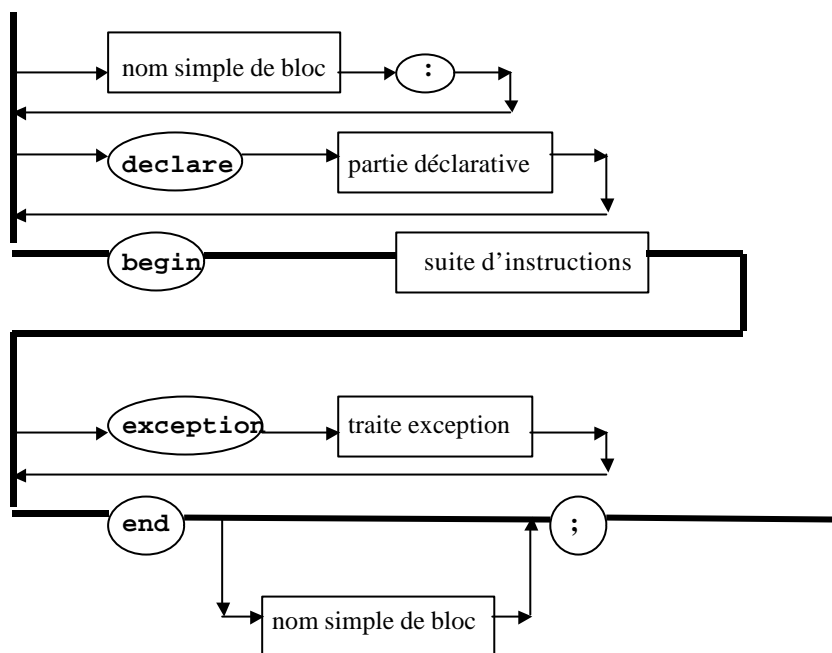
loop
  declare
    CHAI : UNBOUNDED_STRING;
  begin
    LIRE (CHAI);
    VAR_DISCRET := T_DISCRET'VALUE(TO_STRING(CHAI));
    exit;
    exception when others =>
      null; -- ou message ECRIRE("xxxxx");
  end;
end loop;

```

L'instruction bloc est, ici, « encapsulée » dans une structure répétitive **loop**.

Le D.S. :

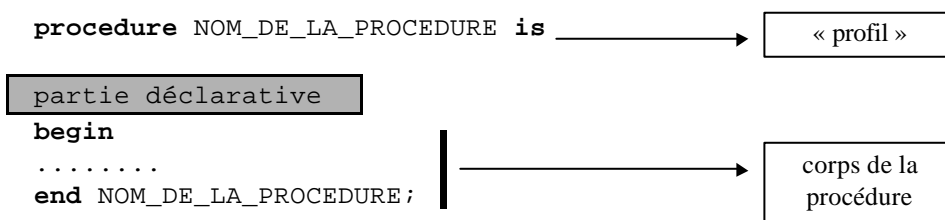
instruction bloc :



Le « chemin » minimum (obligatoire) est en trait plus gras! On remarque que l'on peut nommer un bloc (comme pour une boucle « **loop** »), que l'on peut déclarer des objets « momentanément » (c'est-à-dire dans la **portée** du bloc entre les **begin** et **end**). Le traitement d'exception (compliqué) sera étudié en détail dans le cours n°8.

Déclarations des objets (pour aujourd'hui : types, variables et constantes).

Un des endroits privilégiés pour déclarer ces objets est la partie dite déclarative d'un bloc (voir ci-dessus) mais aussi dans la partie déclarative d'une procédure ou d'une fonction entre le « profil » et le **begin** (début du corps). On verra le D.S. de ces deux concepts au cours n°5 voyons schématiquement l'endroit :



Les déclarations de types (ou de sous types) sont précédés du mot clé **type** (ou **subtype**!). Les déclarations de variables commencent par l'écriture de l'identificateur suivi de « : », quant à la déclaration d'une constante elle a été vue en D.S. (constante universelle). Voyons d'autres exemples :

déclarations de constantes « typées » :

```
DIX_MILLE :   constant  INTEGER    := 10_000 ;
RACINE_2    :   constant  FLOAT     := 1.414 ;
DEPART     :   constant  CHARACTER := 'A' ;
VRAIE      :   constant  BOOLEAN   := TRUE ;
```

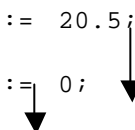
les constantes universelles (**absence de type**) ont été vues : en illustration des DS au cours n°1!

déclarations de variables :

```

CLASSE      :    BOOLEAN;
TAUX_TVA    :    FLOAT      := 20.5;
CARCOU      :    CHARACTER;
NB_DE_LE    :    T_COMPTEUR := 0;

```



On peut initialiser a priori la variable avec une valeur particulière (bonne habitude).

Quelques exercices faciles : (travaillez, prenez de la peine!)
Exercice n° 1 :

Montrez que la séquence suivante :

```

for IND in 1..5
loop
    .... traitement utilisant IND
end loop;

```

ne peut pas toujours se traduire par :

```

IND := 1;
while IND <= 5
loop
    .... traitement utilisant IND
    IND := IND + 1;
end loop;

```

Réfléchissez !

Aide : imaginez ce qui se passe avec IND (dans le deuxième schéma) typé et déclaré par :

```

subtype T_INDICE is INTEGER range 1..5;
IND : T_INDICE;

```

Exercice n° 2 (EXO2_C3.ADB) :

Ecrire un petit programme qui :

- édite à l'écran les 10 premiers nombres entiers
- édite à l'écran les lettres minuscules dans l'ordre normal
- édite à l'écran les lettres majuscules dans l'ordre inverse

Exercice n° 3 (EXO3_C3.ADB) :

Ecrire un programme qui lit (un à un) une suite de nombres entiers non nuls. La fin de la saisie est marquée par la saisie du nombre nul. Calculez et éditez la moyenne arithmétique des nombres.

Rappel :

Il y avait des exercices (plus difficiles !) en annexe du cours n°2. A faire (pour progresser !)

BON courage!

Je retiens n°1

Quelques termes, informations ou conseils à retenir après deux « bonnes » semaines de cours Ada.

- Le terme **statique** s'applique aux informations déterminables à la **compilation**.
- Le terme **dynamique** s'applique aux informations déterminables à l'**exécution**.
- La typographie d'une lettre (majuscule ou minuscule) est sans importance sauf dans les littéraux chaînes (STRING) et dans les littéraux caractères (CHARACTER).
- Les littéraux chaînes sont délimités par des guillemets (") et les littéraux caractères par des apostrophes (').
- Un littéral chaîne peut être vide "", un littéral caractère ne peut l'être (LATIN_1.NUL n'est pas vide!).
- Les **espaces ne sont pas permis** dans les identificateurs, les mots réservés, les littéraux numériques etc.
- Les **espaces sont permis** dans les chaînes et les commentaires.
- La présence d'un **point** permet de distinguer les littéraux numériques réels des littéraux numériques entiers.
- Le trait bas (_) est **significatif** dans les identificateurs mais pas dans les littéraux numériques.
- Un littéral numérique entier **ne peut pas avoir d'exposant négatif**.
- Le signe - (et éventuellement +) ne fait pas partie de la définition syntaxique d'un littéral numérique c'est un opérateur unaire.
- « **Compiler** » c'est l'action (ou la commande) qui présente au compilateur un fichier (source) contenant du codage Ada dont on souhaite contrôler la syntaxe. En cas de succès, le gestionnaire de bibliothèque (library) le répertorie à l'**aide de son identificateur de procédure** ou de **paquetage**.
- « **Faire l'édition de liens** » c'est l'action qui demande à l'éditeur de liens (link) de construire un **programme exécutable**. Seuls les procédures peuvent être « linkées » et non les paquetages (qui sont des ressources). L'identificateur à fournir est celui figurant dans la bibliothèque il est répertorié après la compilation (et non celui du fichier compilé). Avec gnat il est recommandé de prendre le même !
- Les déclarations et les instructions **sont terminées par un point virgule**.
- L'affectation est symbolisée par :=
- Une valeur peut être affectée à un objet dès sa déclaration (initialisation).
- Chaque définition de type introduit un **type distinct**. Attention aux types muets (dits aussi anonymes).
- Un sous type **n'est pas un nouveau type**. C'est le type de base mais avec une contrainte éventuelle.
- Un type est **statique**. Un sous type **peut ne pas l'être**.
- Les types numériques (entiers et réels) **ne se mélangent pas**. Pas d'arithmétique mixte (sauf **).
- **mod** et **rem** sont seulement équivalents pour des **opérandes de même signe (positives ou négatives)**.
- Les **opérateurs arithmétiques** ont des priorités différentes (à connaître).
- Un type scalaire n'est jamais vide, son sous type peut l'être.
- POS, VAL, SUCC, PRED agissant sur un sous type agissent en fait sur le type de base. Mais FIRST et LAST par contre font bien référence au sous type.
- La qualification utilise une Apostrophe. Signée J. Barnes !
- L'ordre d'évaluation des opérateurs logiques binaires **n'est pas défini dans la norme** sauf pour **or else et and then**.
- Les terminaisons d'instructions structurées (**loop, if, case** etc.) doivent se correspondre.
- L'abus de **elsif** n'est pas interdit !
- Les « alternatives de choix » dans un **case** sont statiques.
- Toutes les possibilités du **case** doivent être couvertes (d'après le type de l'expression sélectionnée).
- Dans le **case** si **others** est utilisé il doit être l'**unique choix de la dernière alternative**.
- On peut **qualifier l'expression sélectionnée** d'un **case** pour diminuer le nombre des alternatives.
- Un paramètre de boucle **for** se comporte comme une variable locale du **loop**.
- Un **loop** nommé (ou un bloc nommé) doit comporter un nom à ses deux extrémités.
- Pour avoir des listings propres utilisez un reformateur (sous Linux c'est un script identifié par gnatformat sur notre système; sous NT il est inclus dans l'environnement AdaGide). Utilisez la commande d'impression I.
- Voir aussi le questionnaire fin du cours n°1 !

Editer encore un fichier intéressant (sur le CDROM : cahier1.doc). Visitez les sites de Ada_et_le_Web.html.

Cours n° 4 Ada les tableaux (2 heures)

Thème : Le type tableau.

Le type tableau est un type « **composite** » (revoir la classification des types cours n°2 page 7). Un type composite permet de **regrouper plusieurs éléments** dans une **même structure**. On verra dans une semaine l'autre type composite c'est-à-dire le type article (**record**) cours n°6.

L'intérêt de regrouper, sous une même entité, des éléments que l'on appelle des **composants** est fort utile en informatique. L'intérêt des tableaux (on parle aussi de vecteurs quand ils sont mono-indices) est reconnu depuis l'avènement des premiers langages (le FORTRAN par exemple). On va voir cette semaine (n°3) en Algorithmique la « manipulation » des entités MOTS que l'on pourra, en Ada, réaliser (on dit aussi implémenter) avec des tableaux.

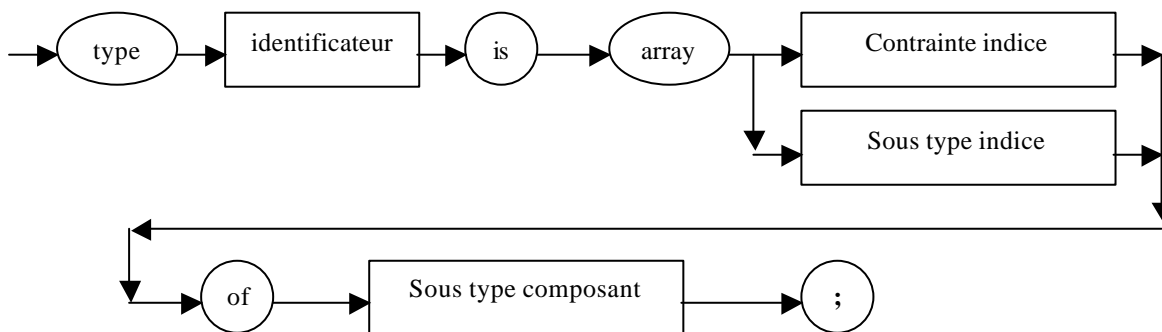
Un objet tableau est un objet composé **d'éléments qui sont tous du même type**. Les éléments sont accessibles (et repérés) au moyen **d'indices** (on parle aussi d'index). Il y a donc deux **concepts** dans un type tableau Ada : le ou les **indices** du tableau d'une part et les **composants** du tableau d'autre part.

La définition d'un tableau comportera :

- d'une part la définition du type des indices (type **discret seulement**) donc : type énumératif ou type entier
- et d'autre part la définition du type des éléments (ou composants) du tableau (tout type sans limite).

En Ada (gros apport) nous pourrons définir (voir le D.S. dans le cours n°1) des **modèles** de type tableau dans lesquels les domaines des indices sont **incomplètement spécifiés** (on parle de tableaux **non contraints** car la « plage » d'indices est « imprécise »). Mais, de façon effective, on travaillera, finalement (à l'exécution), sur des tableaux **contraints** c'est-à-dire dont l'intervalle est connu et borné. **Cette notion là n'est pas facile.**

De façon générale on a le D.S. définition d'un type tableau (et pour faire « court »!) :



Exemples (attention ici : type tableaux **non** contraints ou **modèle** de tableaux) :

```

type T_VECTEUR1 is array (POSITIVE range <>) of FLOAT;
type T_VECTEUR2 is array (NATURAL range <>) of INTEGER;
type T_VECTEUR_DE_B is array (CHARACTER range <>) of BOOLEAN;
type T_MAT is array (T_ENTIER range <>, T_ENTIER range <>) of FLOAT;
type T_NOM is array (INTEGER range <>) of STRING (1..30);
  
```

La notation <> s'appelle boîte (box en anglais) elle indique que l'intervalle des indices est retardée (avec **range**). A partir de ces types de base il est possible de **déclarer** des objets à l'aide de **contraintes** d'indice.

Exemples (d'instanciation i.e. de déclaration d'objets de type tableau):

```

VECT_N : T_VECTEUR1 (1..10); -- 10 composants réels
VECT_B : T_VECTEUR_DE_B ('A'..'T'); -- 20 composants booléens
  
```

A cette façon de procéder, on **préférer** la séquence suivante :

1) Donner la définition d'indice mais contrainte par un intervalle :

```
subtype T_INDICE_N is POSITIVE range 1..10;
subtype T_INDICE_B is CHARACTER range 'A'..'T';
```

2) Instancier les objets de la façon suivante :

```
VECT_N : T_VECTEUR1 (T_INDICE_N);
VECT_B : T_VECTEUR_DE_B (T_INDICE_B);
```

ou **mieux encore** après l'étape 1:

2) déclarer un sous type tableau

```
subtype T_VECT1_10 is T_VECTEUR1 (T_INDICE_N);
(Le sous type tableau T_VECT1_10 devient alors contraint)
```

3) et instancier des objets de la façon suivante :

```
VECT_N : T_VECT1_10;
```

C'est plus long mais
c'est une bonne
méthode !

La plupart des opérations sur les objets «tableaux» c'est-à-dire sur les instances de type tableaux portent presque toujours sur les **composants** du tableau. On « nomme » (ou on distingue) le ou les composants :

- soit **individuellement** (on cite la « valeur » de l'indice » associé à l'identificateur de l'instance). Par exemple `VECT_N(2) := 1.5;` on travaille avec le composant d'indice «repéré » par le numéro 2. Ou encore `VECT_B('G') := TRUE;` l'indice est de type caractère et le composant est de type booléen !
- soit par **paquet** (on cite l'intervalle ou « tranche » d'indice). Par exemple avec la notation `VECT_N(4..8)` on travaille avec 5 composants à la fois (joli mais à revoir !).
- ou encore **globalement** (il suffit de citer seulement l'identificateur de l'instance **sans indice**). Par exemple avec `VECT_N` (qui signifie `VECT_N(1..10)`) on travaille alors avec tous les composants ensembles.

Mais on verra que l'on peut mettre en œuvre des **opérations sur les indices** d'un tableau grâce aux **attributs**.

Généralités sur les types tableaux (tranche ou global).

Quelques précisions (pour les objets tableaux **manipulés en tranche ou globalement**). Les tableaux appartiennent aux types à affectation si le type des composants permet l'affectation. Les opérations d'affectation et de test d'égalité et d'inégalité sont donc autorisées sur **tous les tableaux de même type** (de composants et d'indices) et de mêmes dimensions. Ces **opérations portent sur les composants** évidemment. Pour les tableaux **unidimensionnels** les opérations supplémentaires comprennent les opérateurs relationnels (`<`, `>`, `<=` et `>=`) si le type des composants des tableaux est un type **discret** (ce qui exclut les réels) et on a aussi l'opérateur de concaténation (`&`) voir exemple plus loin. De plus si le type des composants est un type booléen, alors les opérations comprennent aussi l'opérateur logique (unaire) **not** et les opérateurs logiques **and**, **or**, et **xor**.

Opérateur d'égalité (règle) :

Pour deux valeurs tableaux l'opérande de gauche **est égal** à l'opérande de droite **si pour chaque** composant de l'opérande de gauche **il existe** un composant « homologue » de l'opérande de droite **qui est identique**.

Opérateur de relation (règles) :

Un tableau vide sera toujours inférieur à un tableau contenant au moins un composant. Pour des tableaux non vides l'opérande de gauche est inférieur à celui de droite dès que le premier composant de l'opérande de gauche est inférieur à son « homologue » de l'opérande de droite. On peut donc **comparer des tableaux unidimensionnels de tailles différentes** mais de composants **discrets** (rappel !).

Par exemple (les littéraux chaînes caractères étant aussi des tableaux !):

- "Anticonstitutionnellement" est **inférieur** à "Zoulou" ! cf. le dictionnaire !
- de même (et c'est a priori plus surprenant):
- "320" est **supérieur** à "2999". Attention ne pas confondre avec les « valeurs numériques » 320 et 2999 !

Exemples (déclarations et utilisations) :

```
type T_VECT_CAR    is array (POSITIVE range <>) of CHARACTER;

subtype T_INDICE  is POSITIVE range 1..20;
subtype T_MOTS    is T_VECT_CAR (T_INDICE);

PREMOT, MOTCOU   : T_MOTS;
```

PREMOT et MOTCOU sont alors des objets tableaux de 20 **caractères** indicés de 1 à 20.

Quelques exemples (ici 4 exemples) d'opérations possibles :

- PREMOT (11..13) := PREMOT (14) & MOTCOU (18..19);
- PREMOT(8) := PREMOT (2);
- if PREMOT (2..4) /= MOTCOU (4..6)
- then
- exit when MOTCOU (1..8) > MOTCOU (3..6);

Tranches différentes !

Tableaux de tableaux.

Exemples :

```
type T_LIGNE is array (POSITIVE range <>) of CHARACTER;
subtype T_INDICE_COLONNE is POSITIVE range 1..80;
type T_ECRAN is array (POSITIVE range <>)
  of T_LIGNE (T_INDICE_COLONNE);
subtype T_INDICE_LIGNE is POSITIVE range 1..24;
```

```
ECRAN      : T_ECRAN (T_INDICE_LIGNE);
LIGNE      : T_LIGNE (T_INDICE_COLONNE);
```

On peut écrire

```
ECRAN(2)      := LIGNE; -- 2ième ligne
ECRAN(3) (2) := 'A';  -- 3ième ligne et 2ième caractère
```

Notez bien : (3) (2)
différent du multi-
indiaçage (à suivre)

Remarque : le composant (dans un tableau de tableaux) est donc lui même un tableau ! A la déclaration du tableau de tableaux le composant **doit être un tableau contraint**.

Multi-indice.

Une déclaration de type tableau peut faire intervenir plusieurs indices (à ne pas confondre avec le tableau de tableaux vu précédemment). **Notez la virgule entre les indices dans la parenthèse.** C'est traditionnellement en mathématiques (ou analyse numérique) l'implémentation idéale pour travailler avec des « matrices ».

Exemple :

```
type T_MAT is array (INTEGER range <>, INTEGER range <>) of FLOAT;

subtype T_COL is POSITIVE range 1..20;
subtype T_LIG is POSITIVE range 1..10;

MATRICE, MAT1, MAT2 : T_MAT (T_LIG, T_COL);
```

Notation MATRICE(3,2)
pour utiliser la troisième
ligne et deuxième colonne

Attributs portant sur les tableaux :

Les attributs sont intéressants pour travailler, a priori, avec des variables tableaux **dont le type d'indice est retardé**, plus précisément pour travailler avec des «paramètres» **tableaux non contraints**. On verra cela dès le cours n°5 avec les sous-programmes. Pour le moment on admettra que l'on peut faire référence à des travaux portant sur des objets tableaux non contraints c'est-à-dire des tableaux dont la «plage» d'indices **n'est pas encore fixée** complètement. Cette plage sera fixée plus tard et on va, avec les attributs faire référence à cette situation par **anticipation** ! Les attributs sont cependant aussi utilisables systématiquement avec des objets (ou instances) de tableaux contraints. Mais ils **perdent de leur puissance puisque la contrainte est connue**.

Liste des attributs permettant les opérations sur les tableaux (mais en fait **ils portent sur les indices**) :

```
FIRST      LAST      RANGE      LENGTH
FIRST(N)   LAST(N)   RANGE(N)   LENGTH(N)
```

VECT_B et VECT_N sont définis
et déclarés page 2 !

FIRST rend la **valeur de l'indice** du **premier** composant.

Exemple : VECT_B ' FIRST vaut 'A'

LAST rend la **valeur de l'indice** du **dernier** composant.

Exemple : VECT_N ' LAST vaut 10

RANGE rend **l'intervalle des indices** du tableau. Utile pour une boucle **for**. A voir plus bas.

LENGTH rend **le nombre d'éléments** du tableau (nombre effectif de composants).

Exemple : VECT_B ' LENGTH vaut 20 (nombre d'indices dans l'intervalle 'A' .. 'T')

Remarque importante : Les attributs portent sur les (ou font référence aux) instances de type tableau (c'est nouveau !) mais ils **agissent en fait sur le type des indices** du tableau en question mais ... **quand l'indice sera contraint**. On note toujours l'opération avec l'apostrophe mais le préfixe n'est plus un type (comme au cours n°2) c'est (le plus souvent) l'identificateur du tableau. Dur, dur ! Voyons cela.

Exemples :

PREMOT ' RANGE est **l'intervalle** entier 1 .. 20 (type T_MOTS page précédente) mais n'est pas identique à T_INDICE car ce n'est ni un type ni un sous type. De même : PREMOT ' FIRST vaut 1 et PREMOT ' LAST vaut 20. Avec ces notations il n'est **pas nécessaire de connaître ou de se rappeler l'identificateur** (ici T_INDICE) du sous type intervalle d'indice contraint ! Si l'on change la ligne suivante telle que :

```
subtype T_INDICE is POSITIVE range 4..19;
```

sans modifier les autres déclarations on a alors **respectivement** les nouvelles valeurs 4..19, 4, 19 pour les trois attributs précédents et 16 pour l'expression : PREMOT ' LENGTH.

Attention : ' RANGE ne donne pas un type mais un intervalle (déjà dit). C'est une erreur courante des étudiants. On **ne peut pas** par exemple écrire PREMOT ' RANGE ' SUCC(IND) ni IND : PREMOT ' RANGE ;

Si un tableau possède plusieurs indices (par exemple les objets précédemment déclarés MAT1 ou MAT2), il faut indiquer le numéro de la dimension pour les 3 attributs FIRST, LAST et RANGE.

Exemples :

```
MAT1 ' FIRST( 1 ) vaut 1
MAT1 ' RANGE( 2 ) vaut 1..20
```

ou encore

```
MAT1 ' LAST( 1 ) vaut 10
MAT2 ' LAST( 2 ) vaut 20
```

¹ Jusqu'à maintenant les attributs était dénotés sur des types (exemple : CHARACTER ' RANGE)

L'agrégation et les agrégats.

C'est une **opération spécifique aux types composites** (ici donc sur les types tableaux). Elle est souvent utilisée pour **donner des valeurs initiales** aux objets composés. On peut aussi l'utiliser, dans l'algorithme, pour **modifier des valeurs** d'un objet tableau. Il s'agit, en fait, avec l'agrégat, de **littéraux** (comme on a pu le voir avec les autres types scalaires) mais cette fois ce sont des **littéraux tableaux** (voir exemples plus bas).

On travaille, avec l'agrégation, sur le tableau en **entier** ou sur la **tranche**. Il est inutile de recourir aux agrégats pour initialiser un ou quelques composants isolés car l'agrégation **oblige à tous les citer**. L'écriture d'un agrégat ressemble à celle d'un énumératif. C'est normal, car **il faut énumérer** toutes les valeurs attendues.

L'association (c'est-à-dire la **correspondance entre une valeur et un composant**) ne doit être faite qu'une seule fois dans l'énumération.

L'association se fait (il y a **deux cas s'excluant**) :

1) par **position** (dans l'ordre)

```
TABLE(1..6):=(0,-2,7,4,5,-9); -- l'ordre est respecté
TABLE (3..5):=(5,8,12);
```

2) par **nomination** (on cite l'indice puis la valeur associée après le symbole =>)

```
TABLE(4..6):=(5..6 => 0,4 => 1); -- il n'y a plus d'ordre
TABLE(1..6):=(1..3 => 0,4..6 => 1);
```

ou en ajoutant **others** (avec position ou avec nomination) **mais une seule fois et à la fin** (voire seul !).

```
TABLE := (-2,4,9, others => 0);
TABLE := (others => 1);-- équivaut à :
TABLE := (T_TABLE'FIRST..T_TABLE'LAST => 1);
TABLE := (1|3 => 0,2 => 1, others => 0);
TABLE := (1..3 => 1, others =>0);
```

On utilise le caractère | pour indiquer des positions non contiguës. Attention dans les **agrégats de tableaux on ne peut pas mélanger position et nomination**. Les règles de comportement ne sont pas simples. On va voir des exemples mais pour en savoir plus voyez une bonne référence : Barnes pages 123 à 128.

Exemples quand le type tableau a plusieurs indices.

```
type T_MATRICE(POSITIVE range <>, POSITIVE range <>) of INTEGER;
```

ici un double indice :

```
MATRI : T_MATRICE(1..3,1..4); -- matrice de 3 lignes et 4 colonnes
```

```
MATRI := (others => (others => 0));-- tout est mis à zéro
```

ou autre initialisation (différente de la mise à zéro totale) :

```
MATRI := ((1,2,3),
          (4,0,0),
          (5,0,0),
          (6,0,0));
```

ou initialisation équivalente :

```
MATRI  := ((1,2,3),
           (4,others => 0),
           (5,others => 0),
           (6,others => 0));
```

ou, toujours identique :

```
MATRI  := ((1,2,3),
           (1 => 4,2..3 =>0),
           (1 => 5,2..3 =>0),
           (1 => 6,2..3 =>0));
```

Les STRING (chaînes de caractères) type prédéfini

L'utilisation des types non contraints est une activité assez fréquente, en particulier lorsque l'on manipule des chaînes de caractères. Jusqu'à maintenant nous avons évité de manipuler le type **prédéfini** STRING et même après l'avoir découvert nous lui préférons le plus souvent le type `Unbounded_String`. Nous en reparlerons.

Il est temps d'examiner ce fameux type chaîne de caractères prédéfini Ada, c'est-à-dire le type STRING pour une critique sérieuse on peut consulter le cahier pédagogique n° 4 (fichier cahier4.doc dans le CDROM) mais la lecture pour un débutant n'est pas évidente !

Le type STRING est prédéfini par les déclarations suivantes (qu'il ne faut pas écrire ! Puisque « prédéfini »). Cette déclaration (prédéfinie) est faite, comme toujours, dans le paquetage STANDARD (on est prié de consulter le polycopié « paquetages » !):

```
subtype POSITIVE is INTEGER range 1..INTEGER'LAST;
type     STRING  is array (POSITIVE range <>) of CHARACTER;
```

On remarque ainsi que STRING est un type tableau de caractères **non contraint** (à cause du `range <>`). Pour « manipuler » des variables du type STRING il faut **tout d'abord contraindre les indices**. Par exemple :

```
subtype T_STR is STRING(1..80);
```

puis instancier les objets : `A, B : T_STR;`

ou directement instancier avec un type « muet »

```
A, B : STRING(1..80); -- sous-type anonyme
```

ou éventuellement avec initialisation (rappel des agrégats!) :

```
A, B : STRING(1..80) := (others => ' '); -- mis à blanc !
```

On peut également écrire des **constantes** STRING **donc typées** :

```
E : constant STRING := ""; -- vide!
F : constant STRING := (1..0 => 'a'); -- vide aussi!
HELLO : constant STRING := "Bonjour";
```

L'intervalle d'indice pour les constantes STRING est déduit de l'affectation ! pour HELLO c'est 1..7. Pour les deux autres c'est 1..0 (intervalle vide) !

Les valeurs extrêmes des indices sont rarement utiles à retenir (pour les utiliser), car les attributs `FIRST` et `LAST` suffisent à les nommer. Il est possible de ne pas commencer le premier indice d'un STRING par 1 (il suffit d'imaginer la manipulation d'une tranche !) mais c'est une utilisation rare car peu pratique.

Remarque :

Les opérateurs associés au type `STRING` sont ceux permis pour les types tableaux à composants discrets (ce qui est bien le cas du type `CHARACTER` !) donc on a les 7 opérateurs `=`, `/=`, `<`, `>`, `<=`, `>=` et `&` (concaténation de caractères).

Premiers problèmes (on verra plus tard cours n°5 ceux des paramètres `STRING`)

1) La saisie d'un `STRING` n'est pas évidente. Il existe dans le paquetage `Ada.Text_IO` deux procédures : `GET` (à rejeter) et `GET_LINE` (plus pratique). On en reparlera dans le cours 5 bis sur les entrées-sorties. Utilisez pour le moment le `LIRE` de `P_E_SORTIE` mais sachez qu'il y a un ajout d'espaces si la saisie ne sature pas la variable (c'est-à-dire si l'on n'a pas tapé autant de caractères que l'instanciation en a déclarés). Il reste un problème si on a tapé exactement le nombre de caractères déclarés mais ceci est une autre histoire !

2) Les opérations traditionnelles sur les chaînes de caractères reviennent souvent à manipuler des tranches de tableaux de caractères. Il est possible **d'affecter** une tranche de tableau à une autre tranche de tableau mais il faut (affectation oblige !) des **tranches de même taille** (pas forcément de même plage d'indices). Il faut donc connaître les intervalles de ces tranches. Ce n'est pas toujours le cas, de façon simple. Voyons un exemple :

Comment affecter une chaîne à une autre chaîne **si on ne connaît pas, a priori, sa longueur** ?

Soit l'expression très utilisée : `T_JOUR'IMAGE (JOUR)` vue au cours 2. Cette écriture donne un `STRING` puisque c'est le résultat de l'attribut `IMAGE` (voir aussi le document « attributs prédéfinis I: fichier attribut1.doc»). Ce résultat ne porte pas en lui sa longueur ! Et c'est bien le problème ! Comment affecter cette chaîne à une variable `CHAINE_80` de type `T_STR` sans savoir le nombre de caractères ? En deux instructions :

```
LONG_CHAINE := T_JOUR'IMAGE(JOUR)'LENGTH; -- récupère la taille du STRING
CHAINE_80 (1..LONG_CHAINE) := T_JOUR'IMAGE (JOUR); -- vraie affectation
```

La première calcule la longueur de la chaîne « image », la deuxième fait l'affectation dans une tranche. On utilise, hélas, deux fois l'opération `T_JOUR'IMAGE (JOUR)`. C'est peut-être lourd mais fort puissant et ... dynamique (c'est à dire non figé et s'adaptant à toute variable !). Cependant `CHAINE_80` n'est pas saturée d'espaces à la fin et il faut retenir (mémoriser) la valeur `LONG_CHAINE` pour travailler plus tard avec la longueur utile ! Dur !

Pour aider les programmeurs susceptibles d'utiliser les chaînes de caractères `STRING` Ada met à leur disposition des paquetages (ressources) **très utiles** : `Ada.Strings` et `Ada.Strings.Fixed` (une trentaine de sous-programmes à **connaître** et à voir dans le photocopie « paquetages ... »). Il existe aussi un paquetage très simple mais tout aussi utile pour le traitement des caractères : `Ada.Characters.Handling` (à voir aussi).

Recommandation :

On essaiera rapidement (surtout quand on utilisera des variables de type chaînes de caractères) de se servir du type `Unbounded_String` (qui transporte la longueur utile avec la variable et qui est extensible sans limite) voir la ressource (paquetage) `Ada.Strings.Unbounded` (analogue au paquetage `Ada.Strings.Fixed` cité ci-dessus). Le type `Unbounded_String` est identique dans son comportement au `STRING` du langage PASCAL (où la longueur utile est transportée dans le `STRING` lui-même et non limité à 255 caractères !)

Si, par exemple, on a déclaré une variable : `CHAINE : Unbounded_String;`

en lieu et place d'un `STRING` comme `CHAINE_80`, il suffit d'écrire **en une instruction seulement** :

```
CHAINE := To_Unbounded_String (T_JOUR'IMAGE(JOUR));
```

La longueur est gérée implicitement ! Plus souple non ?

De même la saisie d'un `Unbounded_String` est extrêmement facile à utiliser :

```
CHAINE := Get_Line ; -- c'est tout !
```

Avec cette instruction le système attend la frappe **d'autant de caractères que vous voulez** jusqu'à la frappe de la touche Entrée (synonyme de fin de saisie). Le nombre de caractères transportés par `CHAINE` peut être connu à tout moment grâce à la fonction `LENGTH` (sorte d'attribut !) avec l'expression `LENGTH (CHAINE)`.

Exercice n°2 du cours 2 (corrigé) :

```

with Ada.Strings.Unbounded_String, P_E_Sortie;
procedure EXO2_TD2 is
type T_MOIS is (JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET,
               AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE);
subtype T_ANNEE is POSITIVE range 1582..9999;

MOIS : T_MOIS;
ANNEE : T_ANNEE;

begin
  P_E_SORTIE.ECRIRE ("entrez le mois s.v.p.");
  -- modèle de saisie d'un énumératif (cf. paquetage P_E_SORTIE)
  loop
    declare
      use Ada.Strings.Unbounded_String;
      REponse : Unbounded_STRING ;
    begin
      LIRE(REponse) ;
      MOIS := T_MOIS'VALUE(To_String(REponse)); -- ici exception possible
      exit; -- ici tout va bien
      exception when others => ECRIRE("pas valable recommencez:");
        A_LA_LIGNE;
    end;
  end loop;
  -- fin du modèle de saisie d'un énumératif
  -- test de la réponse
  if (MOIS = AVRIL) or (MOIS = JUIN) or (MOIS = SEPTEMBRE)
    or (MOIS = NOVEMBRE)
  then
    P_E_SORTIE.ECRIRE("ce mois a 30 jours");
  elsif MOIS = FEVRIER
  then loop
    declare use P_E_SORTIE;
    begin -- pour l'exception
      ECRIRE(" entrez l'année : ");
      LIRE(ANNEE); -- ici tout va bien
      exit; -- sortie de boucle
      exception
        when others => ECRIRE("non conforme entre 1582 et 9999");
          A_LA_LIGNE;
    end; -- ici reprise de boucle
  end loop;
  if ((ANNEE rem 4 = 0) and then (ANNEE rem 100 /= 0))
    or else (ANNEE rem 400 = 0)
  then
    P_E_SORTIE.ECRIRE ("ce mois a 29 jours");
  else
    P_E_SORTIE.ECRIRE ("ce mois a 28 jours");
  end if;
  else -- tous les autres mois
    P_E_SORTIE.ECRIRE ("ce mois a 31 jours");
  end if;
  P_E_SORTIE.A_LA_LIGNE;
end EXO2_SUP; -- tester sérieusement (préparer un jeu de test dans un fichier)

```

Voir le fichier
cahier1.doc

Exercice sur cet exercice remplacer le **if** par un **case** (facile).

Remarques :

- Notez les portées du **use** limitée au bloc dans lequel il est déclaré (il s'agit d'un effet de style pour démonstration).
- Une année bissextile est multiple de 4 sans être multiple de 100 à moins d'être multiple de 400. Ainsi 1900 n'est pas bissextile tandis que 2000 l'est. 2004 est bissextile (voir cahier n°1.doc).
- Cet exercice est intéressant car il propose et présente un grand nombre de concepts et d'écriture d'instructions et de déclarations Ada. A comprendre et à consulter sans modération !

Cours 5 Ada sous programmes (3 heures) semaine 4

"La programmation est la branche la plus difficile des mathématiques appliquées". DIJKSTRA.

Thème : les sous programmes.

Avec le concept de **sous programmes** nous allons élargir nos connaissances en outils de **génie logiciel**. Nous avons déjà évoqué le **fort typage** Ada comme composante de l'art de **produire du logiciel fiable**. La structuration en sous programmes renforcera cette technique en l'élargissant considérablement. Les notions que nous allons voir sont en interaction forte avec le cours d'algorithmique puisque nous allons facilement implémenter les **modules** si chers à la structuration modulaire d'un algorithme. Ce faisant nous allons aussi commencer à comprendre le concept de **réutilisation** mais partiellement cependant car il sera largement complété avec les **paquetages** et surtout avec la **généricité**. Vos plus tard (cours n°9), un peu de patience donc !

L'approche modulaire (rappel ?)

Nous en sommes, dans le module d'enseignement d'**algorithmique** vers la fin des exercices avec la machine Moucheron. Nous avons donc déjà, de façon implicite mais aussi explicite, montré combien il était commode de décomposer un problème en sous-problèmes allant du **général au particulier**. La lecture des « fiches » (notamment celle de WIRTH le père du langage Pascal) nous y invite. Le principe est de concevoir son algorithme à l'aide de structures de contrôle répétitives, alternatives, de schémas séquentiels d'actions et de bloc de programmes (appelés procédures, modules ou actions complexes). S'il est possible de glisser des assertions bien spécifiques aux endroits stratégiques on tend vers une conception de programmes optimum qui dans les années 70-80 a été dénommée la « **programmation structurée** ». Au début (en 1970) cette approche a été d'une grande portée car auparavant on programait de façon « sauvage » avec des outils tels que l'organigramme qui obligeait le langage d'implémentation à utiliser le **goto**¹, **perfide** instruction permettant de faire presque n'importe quoi. Vers les années 1980 la programmation structurée a montré ses limites à cause surtout de la « dimension » impressionnante de certains logiciels (centaines de milliers d'instructions voire des millions !). La **réutilisation** à grande échelle s'imposait et la modularité n'était plus suffisante, on allait vers la **généricité**, la **dérivation** de classes donc vers « **les objets** ». En 1983 Ada proposera avec les **paquetages** et la **généricité** une première étape : **l'approche objet** (par composition) nous nous y entraînerons bientôt. Puis autour de 1990 arrivent les langages à objets tels Eiffel et C++². Depuis 1995, et pour rester au top, Ada (renommé **Ada95**) **permet les objets** ainsi que de bien belles autres choses³ (notamment pour la gestion et le temps réel) nous y reviendrons. Pour le moment restons modestes et voyons les éléments de **programmation modulaire**.

D'une séquence d'actions au sous programme non paramétré.

En nous appuyant sur une séquence déjà vue nous allons introduire simplement le concept de procédure qui sera approfondi ensuite. Soit le module de lecture d'un objet LA_VALEUR d'un type discret (schéma bien connu présenté dans P_E_SORTIE) formé d'un **loop** engobant un **bloc**.

```

loop
  declare
    CHAI : UNBOUNDED_STRING;
  begin
    LIRE (CHAI);
    LA_VALEUR := T_DISCRET'VALUE (TO_STRING (CHAI));
    exit;
    exception when others => null - ou message !;
  end;
end loop;

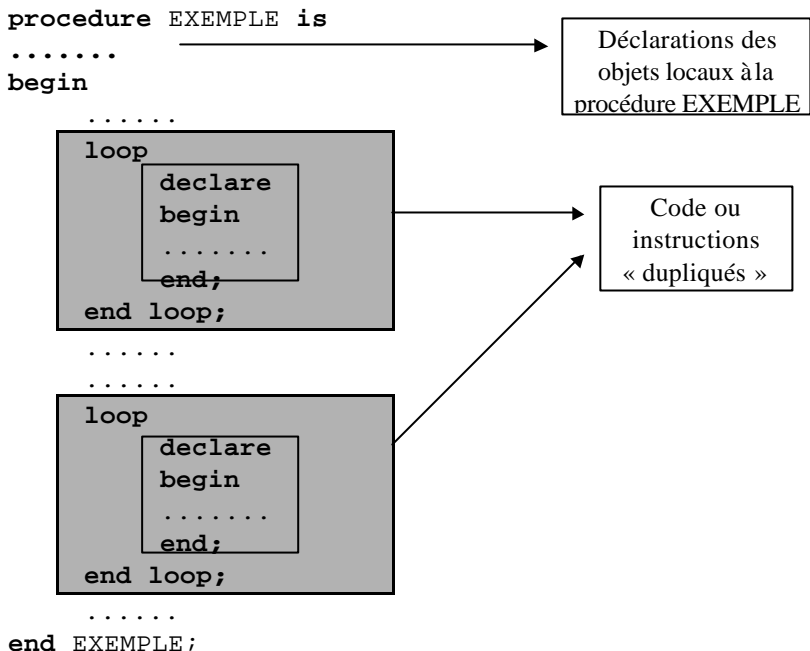
```

¹ Les concepteurs de Ada, la mort dans l'âme (mais pour des raisons de **conversion** de programmes) ont, eux aussi, conservé l'horrible **goto**!

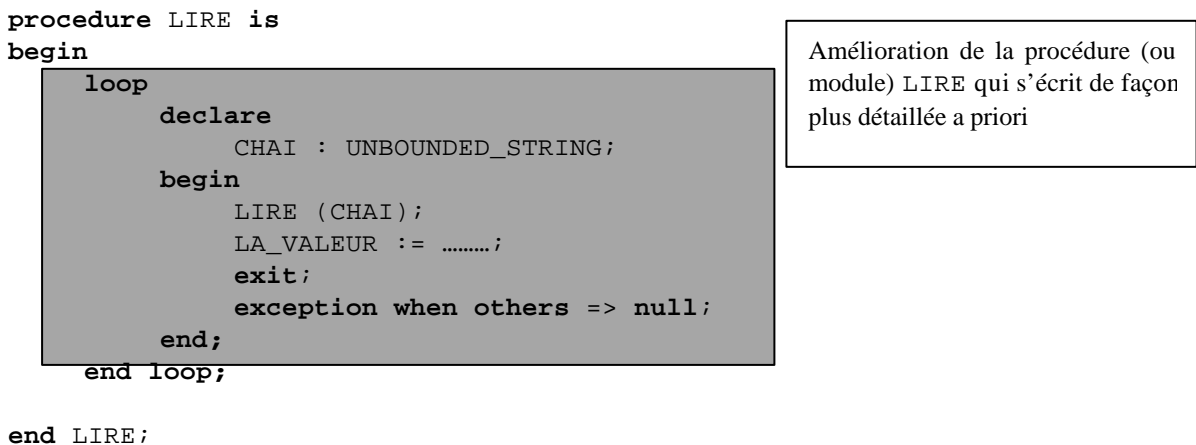
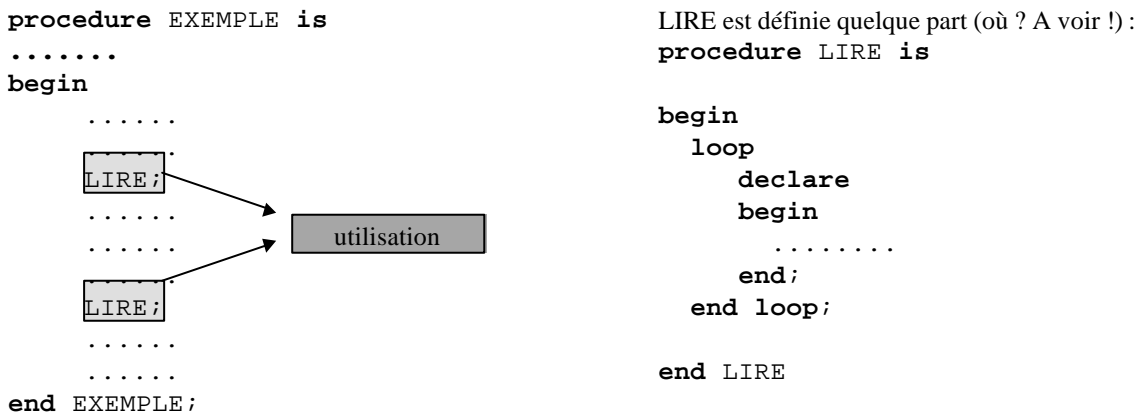
² Respectivement : sûrement le meilleur (Eiffel) et évidemment le plus utilisé (C++)

³ C'est toujours l'Ada-tollah qui parle!

On a une structure répétitive (à cause des erreurs éventuelles) qui permet de recommencer la saisie et un bloc interne qui déclare à **chaque fois** une variable UNBOUNDED_STRING qui est transformée après lecture en la valeur discrète équivalente (cette notion a été expliquée dans un cours précédent). On désire à deux ou plusieurs endroits d'un algorithme mettre en œuvre cette séquence. Voyons le programme utilisateur EXEMPLE :



On remarque bien les deux séquences « **identiques** » qui « gonflent » anormalement l'algorithme. On va écrire **une fois** pour toute la séquence (mais ailleurs), l'**identifier** puis l'**utiliser** en la nommant autant de fois que nécessaire sans la réécrire. Les marquent un peu partout d'autres instructions possibles ou manquantes. Schématiquement il vient :



Après amélioration (expliquée plus loin) la procédure devient :

```

procedure LIRE is
  CHAI : UNBOUNDED_STRING;
begin
  loop
    begin
      LIRE (CHAI);
      LA_VALEUR := T_DISCRET'VALUE(TO_STRING(CHAI));
      exit;
      exception when others => null - ou message ;
    end;
  end loop;
end LIRE;

```

On remarque la déclaration qui a changé de place

surcharge de l'identificateur LIRE (à revoir) ce n'est pas le même !

La déclaration de la variable chaîne de caractères CHAI qui avait lieu à chaque reprise de boucle peut être faite une seule fois (partie **déclarative** de la procédure) nous y reviendrons. On verra aussi la notion très importante en Ada de **surcharge** qui permet d'identifier sans ambiguïté plusieurs objets avec le même nom.

Qu'est-ce qu'un module (première approche simple) ?

C'est une **entité** de programme, en général simple (au sens où elle est susceptible d'être analysée isolément) que l'on peut utiliser dans un programme (et même plusieurs fois) voire réutilisable dans un autre programme. Par exemple dans les exercices d'algorithmique l'incontournable SAUTER_LES_BLANCS.

Méthodes et principes (comment faire et que faire?)

Pour être efficace la **décomposition** en modules doit simplifier la **conception** et la **réalisation** de l'algorithme (évident !) mais aussi sa **lisibilité** (donc sa **validation** même partielle) et enfin sa **maintenance** c'est-à-dire son **évolution** et son **adaptabilité** au cours du temps (vaste projet !).

Pour parvenir à toutes ces qualités il faut que chaque module soit sous-tendu par une « **idée** » **claire** (on parle **d'abstraction**), idée elle-même sous-jacente au problème initial. Il convient aussi que la présentation du module (à l'utilisateur potentiel) c'est-à-dire sa **dénomination** soit explicite et associée à des commentaires décrivant la fonction interne (**spécifications**). La réalisation du module peut aussi parfaitement **rester cachée** à l'utilisateur qui s'intéresse au « **pourquoi** » et **pas au « comment »** (notion de **masquage** et **d'encapsulation**). Enfin le nombre de modules structurant un module plus général (voire l'algorithme lui-même) doit être « raisonnable ». Le nombre de **7** modules maximum est le plus souvent cité par les bons manuels ; cette valeur 7 ne vient pas d'une quelconque croyance en la magie de ce nombre mais tout simplement d'études faites par les cognitivistes sur les capacités de la mémoire à gérer efficacement des tâches (tant sur le plan mnésique que cognitif). En terme de volume d'instructions par module on s'accorde aussi à ne pas dépasser (si possible) une page de listing (une cinquantaine de lignes) ou parfois une page d'écran (un peu juste !).

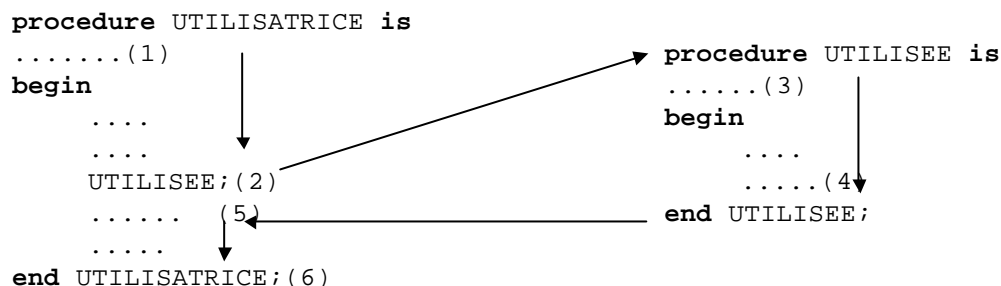
Un autre problème (plus ardu encore) est le critère de **faible couplage**. Quid ? On a vu que deux modules qui se suivent dans un algorithme étaient en général dépendants l'un de l'autre (principe même de la séquentialité). Le faible couplage consiste à **minimiser** le plus possible la dépendance entre les modules. L'idéal serait qu'ils soient totalement indépendants et alors ils pourraient même **s'exécuter en parallèle** !

Voici en gros et en bref la technique de **conception modulaire** à appliquer :

- Tenter **d'isoler**, de **reconnaître** les grandes « **idées** ».
- Préciser leur **nature**, leur **rôle**.
- Les **identifier** pour les repérer et les retenir (faire un choix mnémorique).
- Soigner l'**interface** entre les modules.
- Décrire avec clarté leur **fonction** ce qui permettra une meilleure compréhension.
- Enfin, et seulement après, **analyser** de façon **plus fine** le module en « réitérant » la méthode ou en recourant à une **décomposition en séquences** de contrôles élémentaires.

« Ces choses là sont rudes, il faut pour les comprendre avoir fait des études. » Victor Hugo.

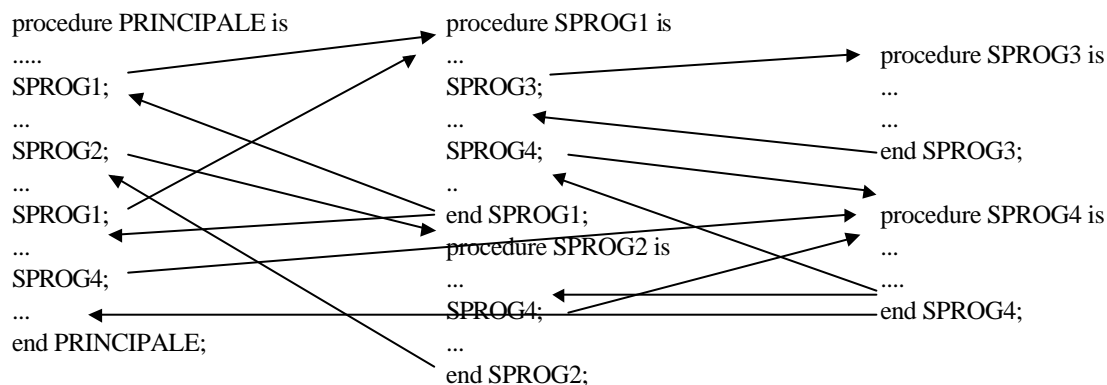
Déroulement et enchaînement des modules (utilisateurs et utilisés). Schématiquement on a 6 points remarquables :



Les points importants ont été numérotés soit : en (1) les variables déclarées dans le module utilisateur (dites **locales**) sont créées (**élaboration**) à l'exécution de la procédure UTILISATRICE. Les instructions s'exécutent jusqu'à l'instruction nommée UTILISEE (2), à cet endroit logiquement on « quitte » la procédure UTILISATRICE (momentanément) pour « exécuter » la procédure UTILISEE avec, de nouveau, élaboration de variables locales (3) puis exécution des actions jusqu'en (4), au retour vers la procédure UTILISATRICE il y a « disparition » des variables locales créées en (3)⁴; enfin on « retourne » exécuter les instructions qui suivent l'identificateur UTILISEE de (5) jusqu'à (6) : fin de la procédure et fin des variables locales déclarées en (1).

Généralisation et complexité :

comme un bon dessin vaut mieux qu'un long discours voyons un exemple simple ? :



L'exécution de ces utilisations (ou appels) est dans l'ordre :

PRINCIPALE, SPROG1, SPROG3, SPROG4, SPROG2, SPROG4, SPROG1, SPROG3, SPROG4, SPROG4.

Je vous laisse imaginer la **complexité** qu'il y a à suivre donc à comprendre l'organisation d'utilisations plus professionnelles ce qui est le cas de tous les algorithmes « vrais » dignes de ce nom !

Le paramétrage.

Introduction.

Revenons à notre problème simple du début à savoir : lecture d'un objet LA_VALEUR de type T_DISCRET. Compliquons : il ne s'agit plus de lire 2 fois le même objet LA_VALEUR mais de lire une fois cet objet et une autre fois un autre objet L_AUTRE_VALEUR mais tout deux de même type T_DISCRET.

En reprenant notre programme EXEMPLE on aurait deux fois la (**presque**) même séquence c'est-à-dire la séquence dite de lecture où changerait seulement la ligne d'affectation de la chaîne lue CHAI à la variable en question respectivement LA_VALEUR et L_AUTRE_VALEUR. Schématiquement :

⁴C'est la même idée que pour le paramètre d'une boucle **for** qui n'existait plus à la sortie : **end loop!**

```

procedure EXEMPLE is
.....
begin

```

```

loop
  declare
  begin
    LA_VALEUR := .....
  end;
end loop;

```

```

code « presque »
identique

```

```

loop
  declare
  begin
    L_AUTRE_VALEUR := ....
  end;
end loop;

```

```

.....
end EXEMPLE;

```

L'idée (pour **factoriser**) reste la même : **écrire une seule fois** la procédure LIRE et l'utiliser souvent. Le problème c'est de ne plus nommer la variable (à lire) de son vrai nom (impossible puisqu'il change !) mais d'utiliser un nom **formel** (c'est-à-dire **abstrait** sans signification particulière) et de lui substituer au moment de l'utilisation le nom **effectif** qui doit remplacer le nom **formel**. C'est la notion de **procédure paramétrée**. On parle alors de **paramètre formel** et de **paramètre effectif** (on verra comment s'effectue l'association).

Schématiquement (mais on reviendra là-dessus) :

```

procedure EXEMPLE is
.....
begin
  .....
  LIRE(LA_VALEUR);
  .....
  .....
  LIRE(L_AUTRE_VALEUR);
  .....
  .....
end EXEMPLE;

```

```

procedure LIRE (V_Formel : ...) is
CHAI : UNBOUNDED_STRING;
begin
  loop
    begin
      .....
      V_FORMEL := ...;
      ...
    end;
  end loop;
end LIRE;

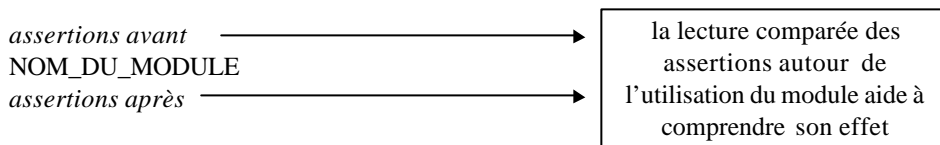
```

La procédure LIRE est **définie** (à droite) avec un paramètre **formel** nommé ici V_FORMEL et elle est **utilisée** (à gauche) avec des paramètres **effectifs** (respectivement LA_VALEUR et L_AUTRE_VALEUR).

On dit que l'on a rendu **paramétrable** l'objet sur lequel portait l'algorithme (il n'est pas figé mais dynamique). Attention cette façon de procéder convient pour des objets de **même type** (ici discret). La question (la bonne question !) est : « ne pourrait-on pas faire la même chose avec des objets différents et de type différent ? » La réponse (la bonne réponse Ada !) est oui (mais c'est la **généricité**) patience ça viendra !

Caractéristiques d'un objet par rapport à un module.

En algorithmique on a observé que des **assertions** bien placées aux endroits « stratégiques » permettaient de mieux comprendre l'effet (la fonction) du module. Les « bons » endroits se situent au début (**antécédent**) du module et à la fin (**conséquent**). En effet si l'on ramène la description du module (au moment de l'utilisation) à son identificateur on peut lire la séquence suivante :



traduit en Ada :

```
-- commentaires sur l'antécédent
NOM_DU_MODULE;
-- commentaires sur le conséquent
```

Les assertions (ou leur traduction en commentaires programme) doivent porter sur les objets intéressants

- au début : sur les objets utiles, nécessaires au module pour sa mise en œuvre (on parle d'objet **donnée**) c'est **l'antécédent**.
- à la fin : sur les objets que le module a modifiés ou initialisés (on parle d'objet **résultat**) c'est le **conséquent**.
- Quand un objet se trouve être, à la fois, un objet donnée et un objet résultat on parle d'objet **donnée-résultat** tout simplement. L'objet apparaît dans **l'antécédent** et le **conséquent**.

La grande **sécurité** (le meilleur contrôle) pour une programmation optimale est de **transmettre** systématiquement les **objets données et les objets résultats en paramètre** du module (sous-programme). Donc en Ada, les objets intéressants seront mis entre parenthèses : soit de façon **formelle** dans la **définition** de la procédure soit de façon **effective** dans **l'utilisation** de la procédure. Ainsi la lisibilité et le contrôle se trouvent facilités. Mais comme c'est une activité très contraignante alors beaucoup de programmeurs la transgressent un peu⁵. D'ailleurs on pourra remarquer que dans les exercices d'algorithmique nous avons trop négligé cette systématisation (faute de pratique et faute de connaissance sur le sujet il est vrai !).

On peut remarquer que les objets **données** (exclusivement données) **ne sont pas modifiés** par le module (ou plutôt ne devraient pas l'être). De même les objets **résultats** (exclusivement résultats) ne sont **que** modifiés on ne s'intéresse pas à leur valeur à l'entrée du module. En Ada le programmeur est obligé de « marquer » la **caractéristique** (appelé mode d'utilisation) du paramètre : donnée avec **in**, résultat avec **out** ou donnée-résultat avec **in out** et ceci dans la définition du module c'est-à-dire dans **la liste des paramètres formels**. Cette contrainte permet au compilateur de vérifier qu'aucun paramètre formel annoncé **in** (donc donnée) ne se trouve à gauche du signe `:=` cette vérification est, notez le, d'une grande sécurité. En effet, dans la conception de l'algorithme vous postulez qu'un objet est « donnée » et vous le signalez en mettant **in** puis vous transgressez cette hypothèse dans la programmation en mettant cet objet à gauche d'un signe d'affectation par exemple. Donc de deux choses l'une : ou vous avez mal programmé ou bien votre hypothèse est fautive : Ada vous le signale dans la phase de contrôle syntaxique (c'est-à-dire dès la première tentative d'implémentation). On a là une erreur de type « conceptuel » signalée à la compilation⁶. De même aucun paramètre formel **out** (donc résultat) ne devrait se trouver à droite du signe `:=` cette propriété était contrôlée avec la version 83 de Ada elle ne l'est plus (hélas) avec la nouvelle norme : une variable résultat peut donc être consultée ! Si elle n'est pas initialisée c'est la catastrophe ! Il n'y a aucun contrôle syntaxique sur les objets de caractéristique **in out** ce qui est normal.

Variables globales, variables locales d'un module.

Notez, avant tout, que la propriété nommée **globale** ou **locale** pour une variable est une notion **relative** et s'applique par rapport à un module particulier ; par exemple un objet peut être local à un module et peut être global par rapport à un autre module comme on va le voir dans des exemples.

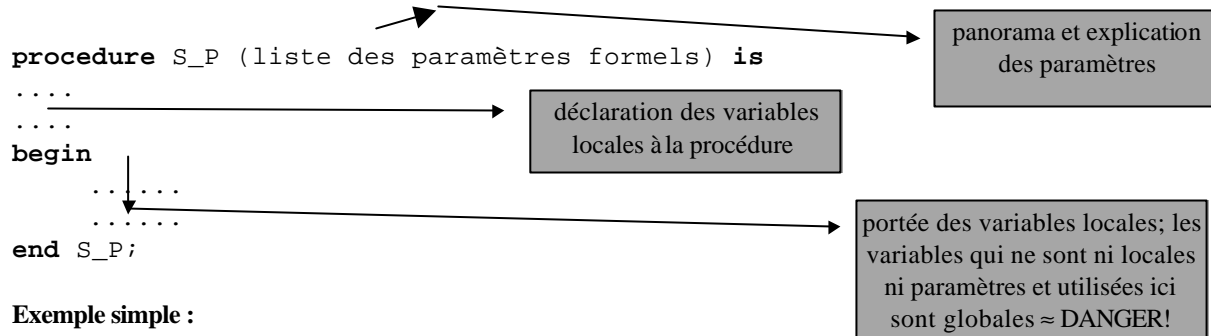
Variable locale :

Une variable est **locale à un module** lorsqu'elle n'existe pas avant la mise en œuvre du module (donc avant son utilisation) et qu'elle est utile au module. La variable locale est « **solidaire** » de **son** module. On dit que sa « portée » est **circonscrite** au module, d'ailleurs la **variable locale est déclarée dans le module**, elle prend « naissance » (élaboration voir ce terme dans le glossaire : fichier glossaire.doc) avec l'utilisation du module et elle « disparaît » avec le module quand celui-ci est achevé.

⁵ et pour d'autres c'est beaucoup, voire passionnément! Alors qu'il faudrait : pas du tout!

⁶ c'est beau Ada!

Une variable est dite **globale à un module** quand elle n'est ni locale, ni paramètre du module. Elle existe avant la mise en œuvre du module, elle est utilisée et/ou modifiée dans le corps du module et elle existera encore après l'utilisation du module. Ce sont en fait des **données-résultats non signalées**. Ce sont des variables **extrêmement dangereuses pour la qualité du logiciel**. On contrôle très mal leur effet puisqu'elles sont modifiables à tout moment. On parle d'**effet de bord** devant une modification intempestive et mal contrôlée d'une variable globale. La **qualité** d'un logiciel pourrait se **mesurer** (entre autres choses) à son nombre minimum de variables globales. En algorithmique CARCOU était (hélas !) un objet global !
On a l'allure générale suivante (déclaration et définition d'une procédure) :



Exemple simple :

On désire écrire un module **COMPTER** qui, dans un objet chaîne de caractères (**STRING**), va compter le nombre d'occurrences d'un caractère particulier. En Ada on écrirait les **spécifications** de la procédure ainsi (on est toujours du côté **formel**) :

```
procedure COMPTER (CAR : in CHARACTER; VEC : in STRING; NB : out NATURAL);
```

Il est clair que :

- seule est intéressante la valeur du caractère **CAR** à chercher donc c'est une donnée ⇒ **in**
- de même le vecteur **VEC** n'est pas modifié et seul son contenu nous intéresse ⇒ **in**
- enfin le nombre de caractères **NB** trouvé est typiquement un résultat ⇒ **out**

Pour **l'utilisation** on pourrait mettre en œuvre la procédure avec des objets « vrais » ou effectifs qui remplaceront les objets « fictifs » ou formels ainsi :

```
COMPTER ('L', PREMOT(1..12), NB_DE_L);
COMPTER ('A', MOTCOU(4..10), NB_DE_A);
```

Notez :

- les **points virgules** séparent les paramètres **formels**, les **virgules** séparent les paramètres **effectifs**,
- le point virgule après la **déclaration** de **COMPTER** (ici ce n'est **pas une définition**). Il annonce seulement le « profil » de la procédure. La définition, qui reste à faire, est caractérisée par le remplacement du point virgule par un **is** suivi du bloc comme on va le voir plus loin sur les diagrammes syntaxiques (**corps**).
- La taille des tranches de **PREMOT** et de **MOTCOU** sont différentes. Voir la réalisation (page 15) de **COMPTER**

Autre exemple de procédure paramétrée bien connue (TD n°2 page 4 algorithmique) :

```
procedure NB_DE_CARAC (CAR : in CHARACTER) is
begin
  COMPTEUR := 0;
  loop
    exit when FIN_RUBAN;
    LIRE;
    if CARCOU = CAR
    then COMPTEUR := COMPTEUR + 1;
    end if;
  end loop;
  ECRIRE_COMPTEUR;
end NB_DE_CARAC;
```

CARCOU et COMPTEUR sont hélas globales, il n'y a pas de variables locales, CAR est paramètre donnée.

Les sous-programmes Ada (détails).

On vient de voir qu'un sous programme est écrit afin de résoudre un sous-problème apparu lors de l'analyse d'un problème plus général. Il peut donc être mis au point séparément et, au besoin, se diviser lui-même en sous-programmes. Un sous-programme est appelé une **unité de programme** donc peut être réutilisé dans d'autres programmes. Les sous-programmes peuvent être **regroupés** de manière logique avec d'autres sous-programmes ou d'autres entités dans un **paquetage** (exemple : P_E_SORTIE) voir détails au cours n° 7.

Il existe deux sortes de sous-programmes :

- les procédures.
- les fonctions.

Un sous programme, comme toute entité du langage : constantes, type, variables ... etc. (on verra d'autres entités), est **introduit par une déclaration**. Cette déclaration de sous programme apparaît, en général, dans une **partie déclarative** soit d'un autre sous-programme soit d'un paquetage soit d'une autre unité.

L'écriture d'un sous-programme se fait (devrait se faire) en deux étapes séparées :

- **déclaration** du sous-programme c'est-à-dire son en-tête (**profil ou spécification**).
- **définition** du sous-programme avec présentation de son **corps** (ou **réalisation**).

Spécification de sous-programme.

Une spécification de sous-programme constitue **l'interface** entre le sous-programme et les unités utilisatrices de ce sous-programme. La spécification précise ce qu'il est nécessaire de savoir pour utiliser le sous-programme. Son nom, la liste de ses paramètres (paramètres formels), plus précisément : leur identificateur, leur mode d'utilisation (caractéristique) et leur type, enfin le type de l'objet retourné dans le cas d'une fonction. Cet ensemble d'informations est **unique** et s'appelle le « profil » du sous-programme. Mais la spécification ne dit rien sur **ce que fait** le sous-programme. Un commentaire sérieux permettra de compléter la spécification. La spécification est l'élément essentiel de la déclaration du sous programme c'est un **contrat**.

Exemples de spécifications :

```

procedure PLACER (LIGNE, COLONNE : in T_COORDONNEE);
-- positionne le curseur sur un écran

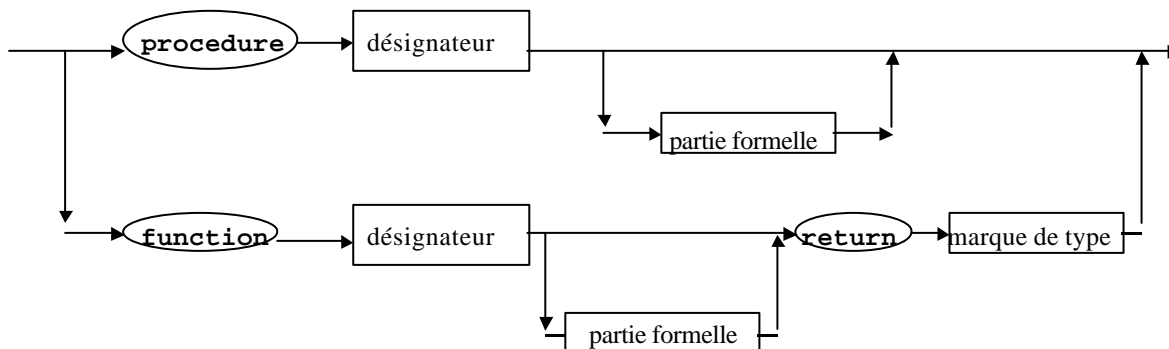
procedure TRIER (A,B : in FLOAT; MIN,MAX : out FLOAT);
-- trie deux nombres A et B: MIN le plus petit, MAX le plus grand

function "*" (X,Y : in MATRICE ) return MATRICE;
-- effectue le produit de deux matrices (hard !)

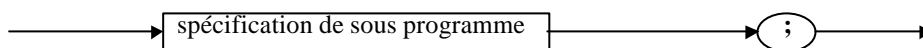
function LE_PLUS_GRAND (A,B : in FLOAT) return FLOAT; -- MAX !
-- action identique à l'attribut MAX !

```

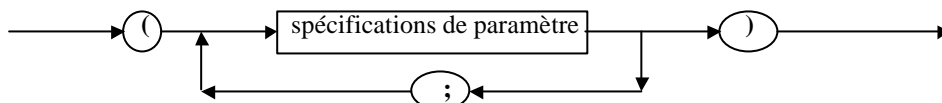
spécification de sous programme :



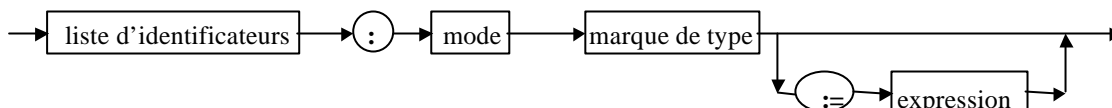
déclaration de sous programme :



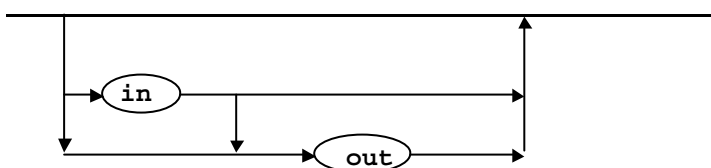
partie formelle :



spécifications de paramètre :



mode :



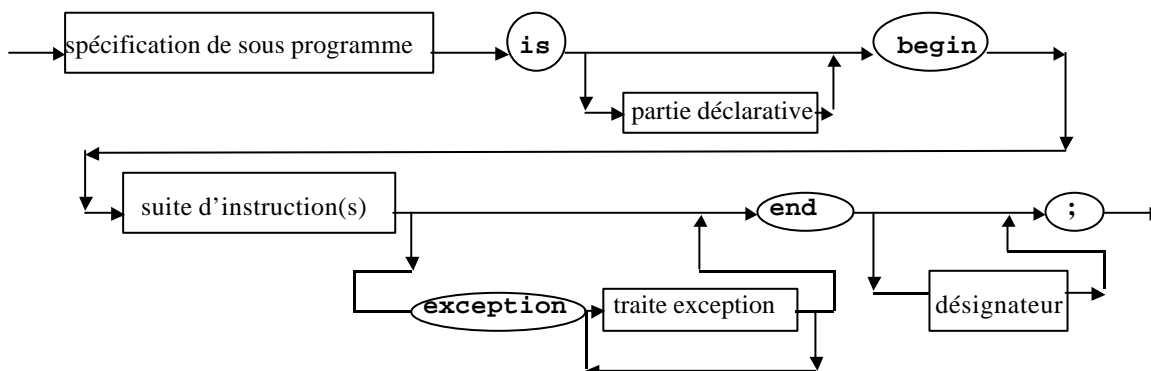
Remarques :

- les **spécifications** de paramètre (formel !) sont séparées par un **point virgule**,
- les identificateurs de paramètres peuvent être regroupés (liste d'identificateurs) et séparés par une virgule,
- on peut **initialiser** un paramètre formel (de mode **in** seulement ; à revoir),
- le mode peut être absent (implicitement c'est alors de mode **in**).

La réalisation du sous-programme.

Le corps constitue la partie **réalisation** du sous-programme. Il va indiquer comment sont réalisées les opérations que la spécification laisse espérer.

Corps de sous programme :



Remarques :

- On retrouve « au début » du corps la partie spécification (reprise « **intégrale** » de la déclaration)
- la partie après le **is** est presque une instruction « bloc » (mais pas de **declare**)
- La partie déclarative "locale" peut ne pas exister lorsque aucune déclaration n'est nécessaire.
- La partie « bloc » qui suit le **begin** contient la description d'une séquence d'instruction(s) (au moins une). Cette description est réalisée à l'aide des entités connues en utilisant les structures de contrôle du langage.
- La partie traite exception sera vue au cours n°8. Elle est (notez le) toujours **située en fin de bloc**

La partie déclarative définit les entités nécessaires à la mise en œuvre. Les entités sont :

- les types.
- les constantes.
- les variables.
- les exceptions.
- les sous-programmes.
- les paquetages.
- les tâches et les objets protégés.
- les clauses.

Exemples de corps de sous-programme ; on les a choisis « courts » et connus (mais à savoir !) :

```

procedure TRIER (A,B : in FLOAT; MIN,MAX : out FLOAT) is
-- pas de déclarations locales
begin
    if A < B
    then MIN := A;
        MAX := B;
    else MIN := B;
        MAX := A;
    end if;
end TRIER;

function LE_PLUS_GRAND (A,B : in FLOAT) return FLOAT is
begin
    if B < A
    then return A;
    else return B;
    end if;
    -- ou tout simplement en une ligne : return FLOAT'MAX(A,B) ;
end LE_PLUS_GRAND;

```

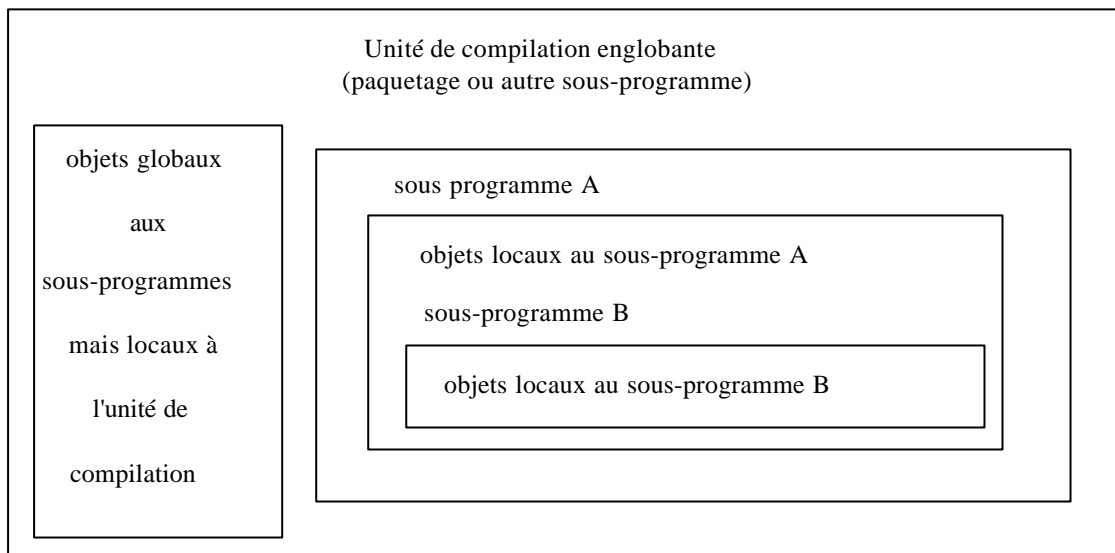
A retenir : la présentation « complète » d'un sous-programme se fait traditionnellement en deux temps (d'abord **déclaration** avec **spécification** puis plus tard **définition** ou **corps** avec **spécification** et **bloc**). Cependant Ada tolère **l'absence de déclaration** distincte de la définition du sous-programme (car cette dernière reprend les spécifications). La présentation en deux temps n'est donc pas obligatoire on peut l'omettre quand les algorithmes sont **simples et compacts**. On verra que cette présentation en deux parties est automatiquement demandée quand le sous programme est issu d'un paquetage (cours n° 7).

Elaboration, portée, visibilité et surcharge.

L'élaboration d'une définition (cf. le glossaire fichier dans le CDROM) est le processus par lequel la déclaration produit ses effets. Ainsi une variable locale à un sous-programme n'est véritablement « créée » **qu'au moment** où le sous-programme est **mis en œuvre** donc au moment où le sous programme **s'exécute**. On gardera à l'esprit cette notion qui permet d'imaginer un certain dynamisme dans la définition d'une variable locale d'un sous-programme. Se souvenir que l'on **élabore une déclaration** et que l'on **exécute une instruction**.

La **portée** d'un objet **local** à un sous programme (c'est à dire un objet déclaré dans la partie déclarative), « s'étend » depuis la déclaration de l'objet jusqu'à la fin du corps du sous programme. Hors de cette « zone » l'objet (donc son identificateur) est **inaccessible**, la portée ne s'étend pas à « l'extérieur » du sous-programme.

Remarquons que si le sous programme déclare aussi dans sa partie déclarative des sous programmes alors ces sous programmes appartiennent à la portée de l'objet local du sous programme initial; cet objet local au sous programme initial peut être utilisé par les sous programmes secondaires dont il est alors objet global ! Hard !



Visibilité :

Dans la figure ci-dessus, le sous-programme B « connaît » ses objets locaux ainsi que les objets locaux du sous-programme A (l'un de ces objets étant le sous-programme B), il « connaît » aussi les objets locaux de l'unité englobante. Le sous-programme A ne connaît pas les objets locaux du sous-programme B. L'unité de compilation englobante ignore tout des objets locaux des sous-programmes A et B. Les objets locaux à une unité supérieure, et donc accessibles à un sous-programme interne, sont dits globaux pour ce sous-programme.

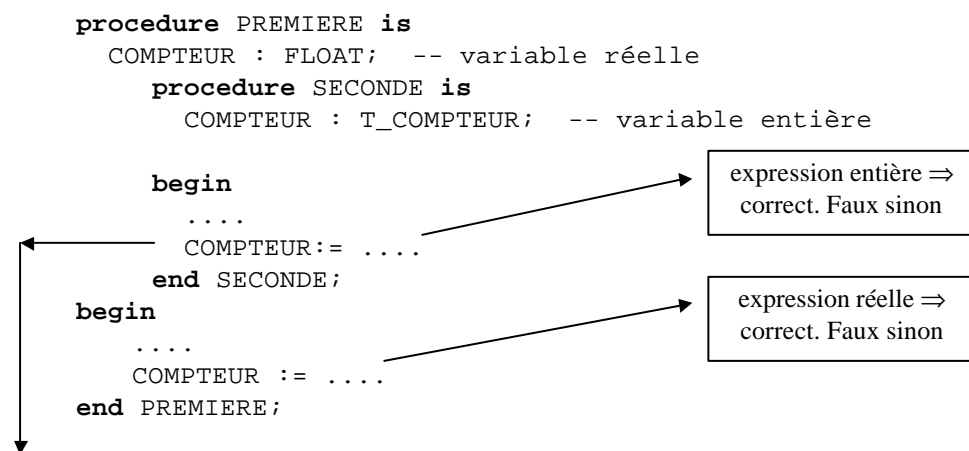
Règles sur la portée et la visibilité :

- Une entité n'est pas accessible (donc pas référençable) tant qu'elle n'a pas été déclarée. Elle n'a pas besoin d'être réalisée pour qu'on y fasse référence (cas d'un sous programme présenté en deux temps).
- Dans un sous-programme, les entités accessibles sont les entités déclarées dans ce sous-programme (en fonction de la règle d'ordonnancement), et les entités déclarées dans le corps des sous-programmes qui l'englobent.
- La portée d'une entité déclarée dans un sous-programme s'étend à ce sous-programme et à tous les sous-programmes qu'il déclare.

Masquage de visibilité :

Une entité peut ne pas être directement visible si une autre entité, désignée par un même identificateur, a été déclarée dans un sous-programme **plus interne**. Cette nouvelle entité masque la première.

Exemple : (masquage de visibilité ici d'un identificateur de variable)



→ A cet endroit la variable COMPTEUR réelle est masquée par la variable COMPTEUR entière. Toutefois il est possible d'accéder à la première entité dans la procédure SECONDE ainsi : PREMIERE.COMPTEUR en utilisant la notation pointée et préfixée par PREMIERE.

Surcharge.

Le langage Ada permet, en général, à plusieurs déclarations de posséder le même identificateur. C'est le phénomène de **surcharge**. Quand un identificateur surchargé est utilisé (deux sous-programmes par exemple) le compilateur les distingue par leurs profils à condition qu'ils soient différents, sinon il signale l'ambiguïté. Cette différence peut se matérialiser par :

- le nombre de paramètres effectifs,
- le type et l'ordre des paramètres effectifs,
- le type du résultat (pour les fonctions),
- le nom des paramètres formels dans le cas d'une notation nominale (voir plus loin).

Exemples connus : Les procédures `ECRIRE` ou `LIRE` dans le paquetage `P_E_SORTIE`. Lorsque le compilateur rencontre : `ECRIRE (RESULTAT)` ; il décide en fonction du type de `RESULTAT` (paramètre effectif) lequel des sous-programmes `ECRIRE` doit être activé.

On peut surcharger AUSSI tous les opérateurs suivants (c'est très pratique) :

`+ - /* mod rem ** abs not and or xor` (attention on ne peut surcharger **in** !)

Fonctions.

Une fonction est un sous-programme qui « retourne » une valeur **et une seule**. Tous les paramètres d'une fonction sont obligatoirement des paramètres "donnée". Il est recommandé d'indiquer ce **mode** par le mot clé **in** (même si c'est implicite et non obligatoire).

Exemple (surcharge et masquage) avec le quotient réel de deux entiers :

```
function "/" (I, J : in T_ENTIER) return FLOAT;
```

On pourra écrire par exemple

```
Y          : FLOAT;
K1, K2     : T_ENTIER;

Y := "/"(K1, K2);          -- ou mieux Y := K1 / K2 ;
```

La notation curieuse `"/" (K1, K2)` est parfaitement correcte. C'est même la seule forme autorisée si la fonction `"/"` appelée est présente dans un paquetage évoquée avec la clause **with** seule (sans **use**). A revoir !

Normalement l'opérateur `/` (qui est prédéfini sur les entiers) donne un quotient entier (voir cours sur les littéraux numériques) il est ici **surchargé**. Le compilateur choisit : soit l'opérateur « entier » prédéfini quand l'affectation est entière : `K1 := K1 / K2` ; il choisit l'opérateur redéfini ci-dessus si le type de la variable affectée est `FLOAT` comme dans `Y := K1 / K2` ;

Il est possible en Ada de déclarer des fonctions qui renvoient des variables de type composite (tableau ou article). A revoir page 15 plus loin et cours n°6.

Exemple :

```
function "+" (C1, C2 : in T_COMPLEXE) return T_COMPLEXE;
```

Le type `T_COMPLEXE` est un type formé de deux composants (connu ?). On définit ainsi l'addition de deux complexes, il suffit de définir tous les autres opérateurs (`-`, `*`, `/`) pour fabriquer un vrai type `T_COMPLEXE` c'est-à-dire une **structure de données** mais avec ses **opérateurs** (voir plus tard la notion de **type abstrait de données**). On réalisera le type `T_COMPLEXE` au cours n° 7. En TP on verra le type `T_Rationnel`.

Les procédures (rappel ?).

Une procédure est un sous programme qui utilise et/ou modifie un certain nombre de paramètres (éventuellement ne fait aucune modification comme la procédure `ECRIRE` de `P_E_SORTIE`).

Comme pour les fonctions, les paramètres **données** sont déclarés en faisant précéder leur type du mode **in**. De plus, les paramètres **résultats** sont déclarés en faisant précéder leur type par le mode **out** dans la spécification du sous-programme. Les paramètres **données résultats** seront déclarés en faisant précéder leur type par le mode **in out** dans la spécification du sous-programme.

Lors de **l'utilisation** d'une procédure (on parle parfois **d'appel** de la procédure) les paramètres effectifs correspondant à des paramètres formels résultats **out** (ou données-résultats **in out**) doivent être des variables obligatoirement (**pas de littéraux**). Mais les paramètres effectifs correspondant à des paramètres formels données **in** peuvent être soit des variables soit des littéraux (plus largement ce sont des expressions mais ce concept n'est pas si simple et sera vu plus tard !).

Remarques (rappels !):

Si un paramètre est défini uniquement par son type (sans le mode) c'est le mode **in** qui est retenu par défaut (à éviter). L'utilisation de paramètres dans le mode **in** interdit au sous-programme de modifier leurs valeurs. Cela est vérifié à la compilation. L'utilisation en mode **out** n'interdit pas la « lecture » de ce paramètre attention ! Le mode **in out** est interdit pour les paramètres de fonctions (**out** aussi d'ailleurs).

L'instruction `return`.

L'instruction **return** peut être mise partout et jusqu'à plusieurs fois dans une fonction (et même dans une procédure). On utilise l'instruction **return** seule dans une procédure et **return valeur** du résultat dans une fonction. L'oubli de l'instruction **return** dans une fonction est signalé à la compilation (Warning).

Remarque : l'instruction **return** est évidemment optionnelle dans une procédure, quand elle est utilisée elle permet de « quitter » la procédure avant son achèvement en fin de bloc (peu conseillée \approx **goto**).

Exemple de fonction :

```

function OUI (TEXTE : in STRING) return BOOLEAN; -- et commentaires
.....
.....
function OUI (TEXTE : in STRING) return BOOLEAN is
REPONSE : CHARACTER := 'a'; -- mais pas de 'O' ou de 'N'
begin
  while (REPONSE /= 'O') and (REPONSE /= 'N' ) loop
    ECRIRE (TEXTE); LIRE (REPONSE);
    REPONSE := To_Upper(REPONSE); -- mise en majuscules
  end loop;
  return REPONSE = 'O';
end OUI;
```

Autre forme : (bien plus élégante notez le)

```

function OUI (TEXTE : in STRING) return BOOLEAN is
REPONSE : CHARACTER;
begin
  loop
    ECRIRE (TEXTE); LIRE (REPONSE);
    REPONSE := To_Upper(REPONSE); -- en majuscules
  exit when (REPONSE = 'O') or (REPONSE = 'N');
  end loop;
  return REPONSE = 'O';
end OUI;
```

La sortie d'une fonction autrement qu'avec **return** provoque l'exception `PROGRAM_ERROR`.

« Appel » de sous-programmes.

L'**appel** (c'est-à-dire l'**utilisation effective**) d'un sous-programme est possible dans la portée de sa déclaration. L'appel entraîne d'abord le « passage » (ou « transmission ») des paramètres (c'est l'association formel \Leftrightarrow effectif) ensuite vient l'**élaboration** de la partie déclarative du sous-programme, puis enfin l'exécution de la séquence d'instructions de la partie instructions du bloc.

L'écriture d'une **utilisation** de sous-programme est composée du nom du sous-programme « appelé » suivi éventuellement d'une liste de paramètres **effectifs**. La correspondance entre paramètres effectifs et paramètres formels peut être faite de trois manières différentes :

- La notation **positionnelle**. Le rappel du nom du paramètre formel est absent de la liste des paramètres effectifs. C'est l'ordre des paramètres qui permet d'effectuer la correspondance. (Voir exemple 1)
- La notation **nominative**. Le nom du paramètre formel est « rappelé » avec son homologue effectif. Les paramètres formels peuvent être donnés dans n'importe quel ordre. (Voir exemple 2)
- La notation **mixte**. Il est permis d'utiliser à la fois la notation positionnelle et la notation nominative, la première doit toutefois précéder la seconde. (Voir exemple 3)

Exemple :

```
procedure BORNES (A, B, C : in FLOAT; X1, X2 : out FLOAT) ;
```

L'appel s'écrira :

1. BORNES (1.0,3.4,2.8,Z1,Z2); -- pas d'apparition de nom formel
2. BORNES (B => 3.4,A => 1.0,C => 2.8,X1 => Z1,X2 => Z2);
3. BORNES (1.0,3.4,X1 => Z1,C => 2.8,X2 => Z2); -- mélange !

Valeurs par défaut.

Il est possible d'associer aux paramètres formels (de mode **in seulement**) des valeurs par défaut. Ces valeurs sont attribuées aux paramètres formels lorsque **leurs homologues effectifs ne sont pas présents** dans la liste lors de l'utilisation du sous-programme.

Exemple :

```
procedure PLACER(LIGNE : in T_COORDONNEE; COLONNE : in T_COORDONNEE := 1);
```

en cas d'absence de paramètre effectif n°2 alors la valeur de la colonne est « forcée » à 1.

Les appels suivants sont autorisés :

```
PLACER (20); -- équivaut à PLACER (20,1);
PLACER (10,5);
PLACER (12,COLONNE => 11);
PLACER (COLONNE => 11, LIGNE => 12);
```

Les paramètres formels (de mode **in**) qui n'ont pas de valeur par défaut, doivent avoir, à l'appel, un paramètre effectif correspondant (littéral ou variable). Les autres paramètres (**out** ou **in out**) doivent, rappelons le, avoir comme paramètre effectif correspondant un identificateur de variable (pas de littéral).

Types non contraints comme paramètres de sous-programme.

La référence à des **tableaux non contraints comme paramètres formels** de sous-programmes permet la transmission de paramètres effectifs de même type tableau, mais ayant, au choix, des domaines d'indices différents. Le tableau "**effectif**" est, lui, **contraint obligatoirement à l'utilisation**.

Exemple :

```
type T_TABLE is array (T_ENTIER range <>) of FLOAT;-- non contraint
subtype T_TABLE10 is T_TABLE(15..24); -- contraint
```

```
T1 : T_TABLE10;          -- 10  valeurs
T2 : T_TABLE(100..200); -- 101 valeurs
T3 : T_TABLE(-50..50);  -- 101 valeurs
```

```
procedure TRI (T : in out T_TABLE) is
```

```
  AUX : FLOAT;
```

```
begin
```

```
  for IND in T'FIRST..T_ENTIER'PRED(T'LAST)
```

```
  loop
```

```
    for J in T_ENTIER'SUCC(IND)..T'LAST
```

```
    loop
```

```
      if T(J) < T(IND)
```

```
      then
```

```
        AUX := T(IND);
```

```
        T(IND) := T(J);
```

```
        T(J) := AUX;
```

```
      end if;
```

```
    end loop;
```

```
  end loop;
```

```
end TRI;
```

Paramètre formel de
type non contraint !

Utilisation d'attribut
pour accéder aux
valeurs inconnues

Seule l'utilisation des attributs (ici FIRST, LAST) permet de s'affranchir de la taille **mais aussi du type discret** de l'indice de l'objet tableau passé en paramètre effectif qui, lui, est forcément contraint. Cependant on est obligé avec l'utilisation de PRED et SUCC de revenir au type d'indice (ici T_ENTIER) car ces deux attributs opèrent sur un type et non sur un intervalle. **Simulez cet algorithme** (trace !). On peut maintenant écrire dans le corps de la procédure englobante les utilisations : TRI(T1); ou TRI(T2); ou TRI(T3);. On a des appels (ou utilisations) du même sous-programme avec des tableaux de tailles différentes. Cette utilisation de tableau non contraint va être constamment prise en compte dans les TD-TP.

Le type STRING. (approfondissement et ... problèmes !)

Nous avons vu au cours n°4 que le type STRING est un type tableau non contraint. Il peut donc servir de type pour les paramètres formels d'un sous-programme comme on vient de le voir ci-dessus. Un exemple simple est celui de la réalisation de la procédure COMPTEUR vue page 7. On a remarqué à cette occasion que l'on pouvait utiliser un vecteur de caractères (sous forme d'une tranche). Ce vecteur a une taille variable et ne commence pas (ni ne finit) par les bornes extrêmes du vecteur initial. Pour prendre en compte cet aspect dynamique il suffit de travailler avec un vecteur non contraint en paramètre formel ici le type STRING.

```
procedure COMPTEUR (CAR:in CHARACTER;VEC:in STRING;NB:out NATURAL) is
```

```
  COMPT : NATURAL := 0;
```

```
begin
```

```
  for IND in VEC'FIRST..VEC'LAST -- ou encore for IND in VEC'RANGE
```

```
  loop
```

```
    if CAR = VEC (IND)
```

```
    then
```

```
      COMPT := COMPT + 1;
```

```
    end if;
```

```
  end loop;
```

```
  NB := COMPT;
```

```
end COMPTEUR;
```

Voyez comment l'algorithme reste valable même avec un vecteur VEC vide (ou tout au moins le paramètre effectif associé). Voyez comment avec VEC'FIRST..VEC'LAST ou VEC'RANGE (ce qui est pareil) on s'affranchit de l'intervalle effectif que l'on ne connaît pas encore. Ada c'est le pied !

Le type `STRING` permet de réaliser beaucoup d'opérations. On peut écrire des sous-programmes permettant par exemple d'extraire une sous-chaîne, de rechercher une sous-chaîne, de passer en majuscules ou minuscules une chaîne etc. bref, tout ce qu'il faut pour régler le traitement des chaînes ! A voir dans l'étude des paquetages `Ada.Strings.Maps` et `Ada.Strings.Fixed`. Ceci est vu en TD.

Exemple de fonction capable de remplacer partout dans une chaîne OU une sous chaîne QUOI par la sous chaîne PARQUOI avec des `STRING`. Essayez de comprendre ; Pas facile ! on reverra cela avec à la récursivité.

```
function REMPLACER (OU, QUOI, PARQUOI : in STRING) return STRING is
  DEBUT : constant NATURAL := OU'FIRST;
  FIN   : constant NATURAL := OU'LAST;
begin
  if OU'LENGTH < QUOI'LENGTH then return OU; -- trop court
  elsif OU(DEBUT..DEBUT + QUOI'LENGTH - 1) = QUOI
    then -- le début de OU contient QUOI
      return PARQUOI &
        REMPLACER (OU(DEBUT + QUOI'LENGTH..FIN), QUOI, PARQUOI);
    else -- on n'a pas trouvé, on avance d'un caractère dans OU
      return OU(DEBUT) &
        REMPLACER(OU(DEBUT + 1..FIN), QUOI, PARQUOI);
    end if;
end REMPLACER;
```

Acceptons cette fonction sans en comprendre le corps mais bien sa spécification (pourtant c'est beau la récursivité !).

Supposons déclarées les variables `MOT1` et `MOT2` de type `STRING`. On veut mettre dans `MOT2` le résultat du remplacement de "CECI" par "CELUI-CI" dans `MOT1`. Si on écrit :

```
MOT2 := REMPLACER (MOT1, "CECI", "CELUI-CI");
```

Problème !

la notation `MOT2 :=` stipule l'affectation globale à tout l'objet `MOT2`. Si `MOT2` a été déclaré contraint (par exemple `MOT2 : STRING(1..MAXI)`) avec une taille maximale suffisamment grande pour tout accepter, alors `MOT2 := . . .` équivaut à `MOT2 (1..MAXI) :=` En général `MOT2` est utilisé avec des éléments en nombre plus réduit que cette taille maximum. C'est le cas ici ; il est rare que `REPLACER` renvoie exactement un objet de taille `MAXI`. Il manque, pour l'affectation à `MOT2`, la connaissance de la "taille". L'affectation nécessite une tranche, l'écriture correcte est :

```
MOT2(1..?) := . . . . .
```

Il faut alors procéder en 2 temps :

- 1) `LONG := REMPLACER (MOT1, "CECI", "CELUI-CI")'LENGTH;`
pour connaître la longueur de la chaîne rendue par `REPLACER` et l'affecter à `LONG` puis enfin :
- 2) `MOT2(1..LONG) := REMPLACER (MOT1, "CECI", "CELUI-CI");`
soit 2 appels (ou utilisation) de `REPLACER`. affreux, affreux comme disait la marionnette J.P.P.!

On comprend qu'il serait commode d'avoir un type chaîne de caractères qui "porte" avec lui sa longueur et que celle ci soit variable (avec le temps) ce sont les fameux `Unbounded_String`. On les étudiera en TD-TP

Si on a déclaré :

```
CHAINE : Unbounded_String ;
```

Alors on réalise le même exercice en une instruction :

```
CHAINE := To_Unbounded_String(REPLACER (MOT1, "CECI", "CELUI-CI"));
```

C'est une fonction qui transforme un String en Unbounded_String

Cours 5 bis Ada Entrées-Sorties simples (1 heure : fin semaine 4)

Avertissement : On entend ici, par les termes **Entrée** et **Sortie**, les notions de **lecture** (clavier) et **d'écriture** (écran) d'où le qualificatif de **simples** pour ces entrées sorties. Le contexte d'Entrée et Sortie sur fichiers sera évoqué dans un autre document (cours n°11). De plus nous nous contenterons de mettre en œuvre des lectures et des écritures de type prédéfini ne demandant pas **d'instanciation génériques** puisque cette notion est vue plus tard (cours 8). Le terme de lecture est-il d'ailleurs bien compris des étudiants ? Pas toujours ! En effet il y a ambiguïté si l'on oublie que ces fonctionnalités s'adressent au programme et non à l'utilisateur. Une **lecture pour le programme** consiste à demander à l'utilisateur d'écrire avec le clavier ! Une **écriture du programme** demandera une lecture sur l'écran à l'utilisateur ! Attention au quiproquo. SAISIR à la place de LIRE et AFFICHER à la place de ECRIRE seraient plus mnémoniques !

Introduction. On rappelle que, pour faire, jusqu'à aujourd'hui, des lectures et des écritures de type prédéfinis : caractère (CHARACTER), chaînes de caractères (STRING), variables de type INTEGER ou type FLOAT, on utilisait les ressources du paquetage P_E_SORTIE. Toutes ces ressources possédaient les identificateurs surchargés (bien commodes) de LIRE et de ECRIRE. Voir les pages du cours n°1 (généralités III) où l'on avait montré les **spécifications** de ce paquetage. Aujourd'hui on va s'intéresser un peu à leur **réalisation** et peut-être songer à remplacer ces LIRE ou ECRIRE par des outils « vrais » de Ada. On pourra posséder (et voir) aussi les réalisations (**body**) du paquetage P_E_SORTIE (**éditer** un listing).

Lecture et écriture de variables de type CHARACTER.

La ressource « vraie » Ada (à évoquer avec **with** pour réaliser ces fonctionnalités sur le type Character) est le paquetage ADA.TEXT_IO. Pour lire on utilise GET et pour écrire on utilise PUT. Pour aller à la ligne on utilise NEW_LINE et pour « vider » le tampon du clavier on utilise SKIP_LINE.

Quand on observe le contenu du paquetage ADA.TEXT_IO (voyez le polycopié « paquetages ... » en annexe A.10.1) on découvre après le commentaire *-- Character Input-Output* les deux procédures :

```
procedure GET (ITEM : out CHARACTER);
```

```
procedure PUT (ITEM : in CHARACTER);
```

La procédure PUT est **exactement** la procédure ECRIRE de P_E_SORTIE. D'ailleurs pour s'en convaincre on peut voir dans les spécifications le renommage (**renames**). Donc PUT édite sur l'écran à la position courante du curseur le caractère en question. Puis le curseur progresse d'une position (colonne) dans la ligne courante.

La procédure GET **n'est pas identique** à la procédure LIRE de P_E_SORTIE ! La réalisation de LIRE est :

```
procedure LIRE (CARAC : out CHARACTER) is
begin
  TEXT_IO.GET(CARAC);
  TEXT_IO.SKIP_LINE;
end LIRE;
```

Le SKIP_LINE est en plus !

- Avec LIRE (CARCOU) si on tape au clavier successivement A puis B puis C et que l'on appuie ensuite sur la touche ENTREE alors CARCOU « reçoit » le caractère 'A' et les autres frappes sont « annulées » (elles sont purgées à cause du SKIP_LINE qui revient au début du tampon).
- Avec GET(CARCOU) si on effectue **les mêmes saisies** (A puis B puis C puis Entrée) alors CARCOU reçoit bien 'A' mais les autres frappes restent dans le tampon de lecture elles rempliront automatiquement le prochain GET à venir (sans frappe au clavier) même si le GET est situé de nombreuses instructions plus loin. Notez que l'utilisateur n'a plus à taper de caractères (ni à appuyer sur ENTREE). Intéressant mais évidemment à connaître. Notez encore que, après un troisième GET, le caractère 'C' serait « lu » mais pas la marque correspondant à Entrée (le tampon n'est pas purgé) ce qui peut poser un problème si la prochaine instruction d'entrée est la saisie d'un String (qui sera alors vide ; 0 caractère !). Commenté en cours ! Difficile !

En **résumé** : l'instruction SKIP_LINE permet de vider le tampon s'il n'est pas vide (y compris la marque modélisant la touche ENTREE). Mais un SKIP_LINE quand le tampon est déjà purgé attend des frappes quelconques terminées par la touche ENTREE et nettoie tout ! Voyez, pour comprendre, dans P_E_SORTIE les réalisations des procédures VIDER_TAMPON et PAUSE.

Lecture et écriture de variables de type **STRING**.

On rappelle qu'une variable de type **STRING** est un tableau de caractères contraint par un nombre maximum de caractères (et ceci à l'instanciation de la variable). Quand on observe le contenu du paquetage `ADA.TEXT_IO` on découvre après le commentaire `-- String Input-Output` les quatre procédures :

```

procedure GET (ITEM : out STRING);
procedure PUT (ITEM : in STRING);
procedure GET_LINE (ITEM : out STRING; LAST : out NATURAL);
procedure PUT_LINE (ITEM : in STRING);

```

La procédure `PUT` est **exactement** la procédure `ECRIRE` de `P_E_SORTIE`. D'ailleurs pour s'en convaincre on peut voir dans les spécifications du paquetage le renommage (**renames**). Donc `PUT` édite sur l'écran à la position courante du curseur la chaîne de caractères en question. Le curseur progresse d'autant de positions (colonnes) que le **STRING** contient de caractères instanciés.

Exemples : `PUT (MESSAGE(5..8))` ; écrit les 4 caractères de la tranche depuis la position du curseur puis se place « au bout » en attente d'autres éditions. Tandis que `PUT (MESSAGE)` ; édite « tout » le **STRING** avec le nombre de caractères dont il a été contraint. Si les caractères à l'extrémité n'ont pas reçu de valeur il y aura édition de caractères « parasites » tels que la mémoire est censée les représenter à cet instant là. N'oubliez pas que des caractères espaces sont placés a priori au bout du **STRING** (ce que faisait la procédure `LIRE` !).

La procédure `GET` **n'est pas identique** à la procédure `LIRE` de `P_E_SORTIE` ! En effet la procédure `GET` demandera autant de caractères à saisir qu'il y en a eu de déclarés dans la contrainte d'instanciation ! Il n'y a pas de dynamisme ! Si on appuie quand même sur `ENTREE` avant d'avoir tapé les quelques caractères attendus l'algorithme ne progresse pas et attend le « reste » ! **Instruction à proscrire** sauf sur fichier éventuellement (nous y reviendrons). Si vous tapez plus de caractères et appuyez sur `ENTREE` le surplus restera dans le tampon et sera utilisé par le `GET` suivant (éventuel !) à moins qu'un `SKIP_LINE` ne vide tout cela !

`PUT_LINE` est simple à comprendre c'est un `PUT` terminé par un `NEW_LINE` ! Attention cette instruction est **réservée** aux **STRING**. Pas de `PUT_LINE` avec une variable de type `CHARACTER`, `INTEGER`, `FLOAT` etc. Donc `PUT_LINE (MESSAGE)` affiche sur écran à la position initiale du curseur la totalité du **STRING** (ici `MESSAGE`) qui sera passé en paramètre puis **place le curseur au début de la ligne suivante**.

`GET_LINE` est **la** solution au manque de dynamisme critiquée plus haut à propos du `GET` (qui est à proscrire). Attention comme pour le `PUT_LINE` cette instruction est **réservée** aux **STRING**. Notez aussi (c'est important) le deuxième paramètre formel `LAST` (résultat **out**). Quand on utilise `GET_LINE` pour saisir (lire) un **STRING** par exemple avec l'instruction `GET_LINE (MESSAGE, NB_CAR_SAISIS)` on n'est plus obligé de taper autant de caractères que l'instanciation en a déclarés. Ouf ! Le dynamisme est revenu. Le deuxième paramètre est le nombre de caractères saisis (ici `NB_CAR_SAISIS`). A charge ensuite pour l'algorithme de ne mettre en œuvre que la tranche valide (c'est-à-dire `MESSAGE(1..NB_CAR_SAISIS)`) **c'est important à noter**.

On doit instancier une variable **STRING** d'une **capacité suffisante** pour gérer des saisies de taille différentes. On peut imaginer par exemple des demandes de C.V. où les utilisateurs doivent donner leur nom. En prenant un **STRING** grand on peut espérer des saisies de noms plus modestes. Sinon si l'on saisit plus de caractères que l'instanciation n'en a prévus le surplus ne reste pas dans le tampon comme précédemment et le nombre de caractères en résultat est saturé à la taille du **STRING**.

Mais il y a un problème quand on saisit exactement (pile poil !) le nombre de caractères maximum instanciés. En effet, dans ce cas, la marque de fin de saisie (c'est-à-dire la touche `ENTREE`) **reste dans le tampon** et le prochain `GET` ou `GET_LINE` d'un autre **STRING** n'a pas le temps de s'effectuer; l'utilisateur ne peut prendre « la main » et la saisie est vide ! C'est bien regrettable ! Il faut absolument quand on utilise un `GET_LINE` contrôler si le nombre de caractères saisis est égal à la taille maximum instanciée (`NB_CAR_SAISIS = MESSAGE'LENGTH`) et dans ce cas là il faut purger la marque de fin de saisie restante ! Dur ! On peut prévoir un **STRING** plus grand mais ce n'est pas toujours réaliste. On verra la réalisation du `LIRE` d'un **STRING** dans `P_E_SORTIE`. On peut « refaire » la procédure `GET_LINE` ainsi (surcharge) :

```

procedure GET_LINE (S : out STRING; L : out NATURAL) is
  S_BIS : STRING (S'RANGE);
  L_BIS : NATURAL;
begin
  TEXT_IO.GET_LINE(S_BIS,L_BIS);
  if L_BIS = S'LENGTH
  then TEXT_IO.SKIP_LINE;
  end if;
  S := S_BIS;
  L := L_BIS;
end GET_LINE;

```

En conclusion : les saisies de caractères et de chaîne de caractères ne sont pas triviales. Le point crucial reste le tampon qui n'est pas toujours purgé ! (et un skip_line systématique ne règle rien !). Une solution originale existe elle utilise l'instruction Set_Col (à voir prochainement cours fichier).

Lecture et écriture de variables de type ENTIER (signé ou modulaire).

Il existe une solution à ces objectifs dans la paquetage TEXT_IO (plus précisément dans son paquetage interne INTEGER_IO). Mais elle nécessite les concepts de généricité (cours n°9). De plus pour la lecture elle demandera un contrôle de **validation** supplémentaire. Pour ces raisons il est recommandé de plagier ce qui est fait dans le paquetage P_E_SORTIE à propos du type INTEGER en l'adaptant au type ENTIER souhaité.

Par exemple :

```

type T_ENTIER is range -6..40009;
.....
procedure GET(L_ENTIER : out T_ENTIER) is
  REPONSE : UNBOUNDED_STRING;
  LONG : NATURAL;
begin
  loop
    begin
      REPONSE := Get_Line;
      L_ENTIER := T_ENTIER'VALUE(To_String(REPONSE));
      exit;
      exception when others =>
        TEXT_IO.PUT_LINE("saisie non valable recommencez");
    end;
  end loop;
end GET;

```

Cette lecture est **validée** elle utilise les notions vues précédemment Elle lit une chaîne puis la convertie dans le numérique souhaité. Elle s'adapte à tout type ENTIER. Elle pourra être reprise quand nous ferons la généricité. Le concept de généricité consistera tout simplement à rendre **paramétrable** le type T_ENTIER. C'est tout. En ce qui concerne l'écriture d'un type ENTIER on peut là encore en attendant la généricité plagier ce qui est fait dans le paquetage P_E_SORTIE.

```

procedure PUT (L_ENTIER : in T_ENTIER) is
begin
  TEXT_IO.PUT(T_ENTIER'IMAGE(L_ENTIER));
end PUT;

```

Lecture et écriture de variables de type FLOAT.

On utilisera encore `P_E_SORTIE` jusqu'à la connaissance de la généricité et celle du paquetage interne (`FLOAT_IO`) de `TEXT_IO`.

Lecture et écriture de variables de type énumératif.

Là encore, jusqu'à la maîtrise de la généricité et du paquetage interne (`ENUMERATION_IO`) de `TEXT_IO` on plagiera, pour lire un énumératif le bloc de lecture proposé dans `P_E_SORTIE` en fin de spécifications. Soit :

```

loop
  declare
    CHAI : UNBOUNDED_STRING;
    LONG : NATURAL;
  begin
    CHAI := Get_Line;
    ENUM := T_ENUM'VALUE(To_String(CHAI));
    exit;
  exception when others =>
    null; -- ou message ECRIRE("xxxxx");
  end;
end loop;

```

Cette lecture est **validée** et nickel !

En ce qui concerne l'écriture merci l'attribut `IMAGE` ! Soit `PUT` ou `PUT_LINE` :

```

PUT (T_ENUM'IMAGE(ENUM));
PUT_LINE (T_ENUM'IMAGE(ENUM));

```

Cette partie de cours se poursuivra, avec le temps, en fonction des questions qui nous seront posées.

Remarque :

Toutes les possibilités de `TEXT_IO` n'ont pas été évoquées ! On verra en TD-TP quelques fonctionnalités supplémentaires telle par exemple la procédure `Get_Immediate` (lecture non tamponnée d'un caractère).

Quelques interrogations

Dans un littéral chaîne comment représente-t-on les guillemets (puis qu'ils sont délimiteurs de chaîne) ?

Deux solutions :

- On double le guillemet pour le représenter.
- On utilise la compatibilité du type caractère et de la chaîne avec l'opérateur `&`

Exemples respectifs :

Créer la chaîne de 3 caractères : `A"B`. Il est clair que `"A"B` est incorrect car avec `"A"` le compilateur considère une chaîne de un seul caractère le A. Puis trouvant `B"` il pète les plombs (avec message d'injure en anglais!). D'où la bonne écriture : `"A""B"`. Le doublon `""` vaut un seul `"`. De même pour fabriquer la chaîne d'un caractère contenant l'unique caractère `"` : il faut écrire `""""`. Cqfd !

Avec l'autre approche on a : `"A" & "" & "B"`. Tordu non! Mais de toute façon il est incontournable qu'il n'est pas facile de représenter un symbole quand il est, lui même, délimiteur.

Remarque : Une solution (plus élégante déjà évoquée page 3) à la surcharge du `Get_Line` (page 3 en haut) sera vue dans le cours 11 (cours fichier) cependant d'ores et déjà il est bien plus sûr et performant de « passer » par les `Unbounded_String` !

Je retiens n°2

Quelques termes, informations ou conseils à retenir après le cours n° 5 bis Ada (semaines 3 et 4).

Cette fiche fait suite à la fiche « je retiens n°1 » après les semaines 1 et 2.

- Rendez (ou conservez) des listings « formatés » (rappel de la fiche « je retiens n°1 »).
- Le type tableau non contraint est déclaré avec **range** <>.
- Une variable de type tableau **doit être contrainte**. Un paramètre formel de type tableau peut être **non contraint** (c'est le paramètre effectif associé qui sera contraint!).
- Évitez de déclarer des variables tableaux de type anonyme (ou muet).
- Un agrégat avec **others** doit posséder un contexte permettant d'évaluer ses bornes.
- Les attributs FIRST, LAST, RANGE et LENGTH s'appliquent à des types ou sous types **contraints**. Ils s'appliquent aussi à des **variables** de type tableaux (mais toujours contraints!). Enfin ils peuvent s'appliquer à des paramètres formels tableaux **non contraints** (mais en fait référencent le paramètre effectif associé!).
- Dans une affectation de tableaux le nombre de composants doit être le même mais les bornes peuvent être différentes.
- Dans une comparaison de tableaux (mono dimension) les « tailles » peuvent être différentes.
- Un agrégat à un seul composant doit utiliser la notation nominative.
- Une fonction ne peut « atteindre » son **end** final (il faut « quitter » avec un **return**).
- L'ordre des évaluations des paramètres d'un sous programme n'est pas défini par la norme.
- L'expression par défaut d'un paramètre formel « donnée » **in** n'est évaluée que lors de l'utilisation du sous programme « son appel » et aussi si le paramètre effectif est omis.
- Si un paramètre formel tableau est contraint son paramètre effectif doit avoir les mêmes bornes!
- Le mécanisme du « passage » des paramètres pour les tableaux n'est pas défini par la norme.
- Les déclarations des paramètres formels sont « séparés » par des points-virgules. La liste des paramètres effectifs utilise naturellement la virgule.
- La notation nominative pour le « passage » des paramètres est très prisée pour une bonne lisibilité.
- Ne pas confondre **déclaration et définition** (notamment pour les sous-programmes).
- Un sous programme déclaré dans la partie déclarative d'un autre sous-programme appartient uniquement à ce dernier. On retrouve l'esprit des objets locaux.
- Les qualificatifs de local et de global sont des notions **relatives** (elle font références à un sous-programme particulier).
- Un **pragma** est une information fournie accessoirement au compilateur. Par exemples :
pragma OPTIMIZE (SPACE) et **pragma** OPTIMIZE (TIME) demandent un code exécutable optimisé (en place mémoire ou en temps d'exécution). Voir quelques pragmas dans le fichier pragma.doc.
- On n'évoque pas le paquetage STANDARD (avec **with**). On ne le redéfinit pas non plus!
- Le paquetage ADA.TEXT_IO est le paquetage officiel pour les entrées-sorties textes.
- Le type STRING est un type tableau de caractères **non contraint**. Il est « prédéfini » car déclaré est défini dans le paquetage STANDARD.
- Ne pas confondre SKIP_LINE et NEW_LINE. SKIP_LINE est une instruction « d'entrée » tandis que NEW_LINE est une instruction de « sortie ».
- Les « lectures » clavier doivent être toujours validées.
- Voyez bien tous les fichiers et les paquetages suggérés : Standard, attribut1.doc, Ada.Strings, Ada.Strings.Fixed, Ada.Characters.Handling, Ada.Strings.Unbounded, glossaire.doc, Ada.Text_Io, body de P_E_SORTIE, Ada.Characters.Latin_1. Et étudiez les !
- On ne peut comparer des objets tableau de taille différente que si leurs composants sont de type discret (i.e. pas de réels !).