Table des matières

chapitre 5:fonctions (suite)

chapitre 6:types

chapitre 7:types (suite)

    - Corrigé de l'exercice 6.3: puissance récursive
    - Corrigé de l'exercice 6.4: triangle et losange

chapitre 8:tableaux

- Cours
    - Tableaux
    - Fonctions: une discipline
- Exercices
    - Exercice 8.1: remettre dans l'ordre
    - Exercice 8.2: deux erreurs
    - Exercice 8.3: parcours de tableaux
    - Exercice 8.4: tableaux d'entiers
- Corrigés
    - Corrigé de l'exercice 7.1: lendemain
    - Corrigé de l'exercice 7.2: jour de prières
    - Corrigé de l'exercice 7.4: division récursive

chapitre 9:enregistrements

- Cours
    - Enregistrement
    - Complément sur les tableaux
    - Calculs booléens
- Exercices
    - Exercice 9.1: enregistrements
    - Exercice 9.2: des personnes
    - Exercice 9.3: booléens
    - Exercice 9.4: tri de tableau
- Corrigés
    - Corrigé du devoir 2
    - Corrigé de l'exercice 8.2: deux erreurs
    - Corrigé de l'exercice 8.3: parcours de tableaux
    - Exercice 8.4: tableaux d'entiers

chapitre 10:structures de données

- Cours
    - Structures de données complexes
    - Chaînes de caractères
- Exercices
    - Exercice 10.1: petits exercices sur les chaînes
    - Exercice 10.2: cercles
    - Exercice 10.3: bibliothèque
    - Exercice 10.4: basse-cour
    - Exercice 10.5: tableau à moitié vide
- Corrigés
    - Corrigé de l'exercice 9.1
    - Corrigé de l'exercice 9.2
    - Corrigé de l'exercice 9.3: booléens
    - Corrigé de l'exercice 9.4: tri

chapitre 11:procédures

- Cours
    - Procédures
    - Attributs des types tableaux
- Devoir
    - Devoir 3
- Exercices

chapitre 15:déclarations

- Cours
  - Portée de variable
  - Notion d'environnement
  - Retour sur les procédures
- Exercices
  - Exercice 15.1: une horreur de programme
  - Exercice 15.2: paramètres de procédures
  - Exercice 15.3: procédures
  - Exercice 15.4: Tic-Tac-Toe (début)
- Devoir
  - Corrigé du devoir 4

chapitre 16:paquetages

- Cours
  - Paquetage
- Exercices
  - Exercice 16.1: sympa_io
  - Exercice 16.2: bouquets de fleurs
  - Exercice 16.3: tables de vérité
- Corrigés
  - Corrig\'e de l'exercice 15.1: une horreur de programme
  - Corrigé de l'exercice 15.2: paramètres de procédure
  - Corrigé de l'exercice 15.3: procédures
  - Corrigé de l'exercice 15.4: Tic-Tac-Toe

chapitre 17:exceptions

- Cours
  - Levée d'exceptions
  - Récupération d'exceptions
  - Paquetages avec partie privée
  - Compléments sur les tableaux
  - Générateur aléatoire
- Exercices
  - Exercice 17.1: menu
  - Exercice 17.2: bouquet de fleurs bis
  - Exercice 17.3: Tic Tac Toe, la suite
- Corrigés
  - Corrigé de l'exercice 16.1: sympa_io
  - Corrigé de l'exercice 16.2: bouquets de fleurs
  - Corrigé de l'exercice 16.3: tables de vérité

chapitre 18:tableaux non contraints

- Cours
  - Tableaux non contraints
- Exercices
  - Exercice 16.1: tableaux non contraints
  - Exercice 16.2: métro
- Corrigés
  - Corrigé de l'exercice 17.1: menu
  - Corrigé de l'exercice 17.2: bouquet de fleur
  - Corrigé de l'exercice 17.3: tic tac toe

chapitre 19:pointeurs

- Cours
  - Tableaux non contraints et récursivité
  - Pointeurs
- Exercices
  - Exercice 19.1: tableaux non contraints et récursivité
  - Exercice 19.2: tri
  - Exercice 19.3: premiers pas avec les pointeurs
- Corrigés
  - Corrig\'e de l'exercice 18.1: tableaux non contraints
  - Corrigé de l'exercice 18.2: métro

chapitre 20:listes chaînées

- Cours
  - Listes chaînées
  - Comparaison entre listes et tableaux
  - Exemple de listes
  - Les erreurs à éviter
- Exercices
  - Exercice 20.1: familiarisation avec les listes
  - Exercice 20.2: des gammes avec les listes
- Corrigés
  - Corrigé de l'exercice 19.1: tableaux non contraints et récursivité
  - Corrig\'e de l'exercice 19.2: tri
  - Corrig\'e de l'exercice 19.3: premiers pas avec les pointeurs

chapitre 21:conception de logiciel

- Cours
  - Conception de logiciel
  - Exemple: calcul de bulletins de salaire
- Exercices
  - Exercice 21.1: manipulations de listes
  - Exercice 21.2: programmes incorrects
- Corrigés
  - Corrig\'e de l'exercice 20.1: familiarisation avec les listes

chapitre 22:listes (suite)

- Cours
  - Vision abstraite des listes
- Exercices
  - Exercice 22.1: savez-vous lire?
  - Exercice 22.2: récapitulation sur les listes
  - Exercice 22.3: liste triée
- Corrigés
  - Corrigé de l'exercice 21.2

chapitre 23:fichiers

- Cours
  - Fichiers textes
- Exercices
  - Exercice 23.1: petits hors-d'oeuvre
  - Exercice 23.2: listes et fichiers
  - Exercice 23.3: mailing
- Corrigés
  - Corrig\'e de l'exercice 22.1
  - Corrig\'e de l'exercice 22.2

    - Corrig\'e de l'exercice 22.3

chapitre 24:Généricité

- Cours
    - Généricité
- Devoir
    - Devoir 5
- Exercices
    - exercice 24.1
    - Exercice 24.2
    - Exercice 24.3 comptes bancaires
- Corrigés
    - Corrigé de l'exercice 23.1: petits hors d'oeuvre
    - Corrigé de l'exercice 23.2: listes et fichiers
    - Exercice 23.3: mailing

chapitre 25:Généricité (suite)

- Cours
    - Généricité (suite)
- Exercices
    - Exercice 25.1
    - Exercice 25.2: procédure de tri générique
    - Exercice 25.3: recherche d'un élément dans un tableau
- Corrigés
    - Corrigé de l'exercice 24.1
    - Corrigé de l'exercice 24.2
    - Corrigé de l'exercice 24.3

chapitre 26:Généricité (fin)

- Cours
    - Paquetages génériques
- Exercice
    - Exercice 26.1: listes génériques
    - Exercice 26.2: des ensembles avec des listes
- Corrigés
    - Corrigé de l'exercice 25.1
    - Corrigé de l'exercice 25.2
    - Corrigé de l'exercice 25.3

chapitre 27:arbres binaires

- Cours
    - Arbres binaires
- Exercices
    - Exercice 27.1: affichage
    - Exercice 27.2: recherche d'un élément
    - Exercice 27.3: accès à un noeud
- Devoir
    - Corrigé du devoir 5
- Corrigés
    - Corrigé de l'exerice 26.1
    - Corrigé de l'exercice 26.2

chapitre 28:révisions

- Sujets d'examens
  - Enoncé de l'examen de juin 1999
  - Enoncé de l'examen de juin 2000
  - Enoncé de l'examen de septembre 2000
  - Enoncé de l'examen de juin 2001
- Corrigé des exercices
  - Corrigé de l'exercice 27.1
  - Corrigé de l'exercice 27.2
  - Corrigé de l'exercice 27.3
- Corrigés des sujets d'examen
  - Corrig\'e de l'examen de juin 1999
  - Corrigé de l'examen de juin 2001

chapitre 1:présentation, logique

Présentation

Le cours Algoritmique Programmation A (APA) a pour objectif d'étudier les concepts des langages de programmation et de présenter des méthodes de programmation correspondant à des classes d'applications. Le langage utilisé est le langage Ada. Ce cours s'adresse à des personnes qui ont déjà programmé et qui souhaitent approfondir les méthodes et fondements de cette activité ou à des personnes qui n'ont pas cette expérience de la programmation mais qui ont une certaine aptitude a l'abstraction.

De façon a permettre les meilleures chances de succès, ce cours est recommandé :

- aux personnes qui ont déjà une expérience de programmation
- OU aux personnes qui ont le niveau requis par les unites de valeurs Mathematiques pour l'informatique A1 et A2.

L'enseignement à distance utilise essentiellement internet, avec des pages web et du courrier électronique. Pour le suivre, il faut donc impérativement avoir un ordinateur et un accès internet. L'équipement conseillé est un ordinateur de type PC, disposant d'au moins 32 Méga octets de mémoire vive, de 50 Méga octets d'espace libre sur disque, ainsi que le système d'exploitation Windows (95, 98, NT, ...) ou Linux. D'autres configurations sont possibles mais risquent d'être plus complexes à utiliser.

Il faut de plus savoir utiliser cet outil. Si vous ne savez pas déjà utiliser internet, vous ne pourrez pas suivre le cours.

Par ailleurs, il faut que vous installiez un compilateur Ada sur votre ordinateur. Nous vous conseillons le Gnat, que vous pouvez télécharger ou trouver sur un cdrom vendu par la librairie du CNAM. Cette installation se fait généralement sans problème. Nous ne pouvons pas vous aider pour d'éventuels problèmes d'installation.

Il n'existe pas de livre correspondant au programme du cours. Vous pouvez trouver divers documents sur le web, non seulement sur le site du cours à distance, mais aussi sur ceux des cours du soir et du jour, et sur un site de l'école d'ingénieurs du canton de Vaud. Les curieux peuvent aussi faire un tour sur les pages de VARI (il s'agit d'une valeur de cycle B du CNAM) et les sites de référence sur Ada en français (http://www.ada-france.org/) et en anglais (http://www.adahome.com).

La mise en forme du cours à distance, le tutorat et les regroupements sont assurés par François Barthélemy en utilisant des contenus et documents créés par une équipe d'enseignants du CNAM, notamment V. Donzeau-Gouge, T. Hardin, J-F. Peyre, M-V. Aponte, S. Elloumi, M. Simonot, D. Vodislav.

Le choix d'Ada comme langage de programmation pour cette valeur est l'objet d'interrogation de la part des élèves. Ce choix est fondé sur des raisons techniques que vous ne pouvez pas comprendre sans avoir déjà un certain niveau en programmation (ce qui n'est pas a priori votre cas). Notre but n'est pas de faire de vous des programmeurs Ada mais des programmeurs tout court, pour que vous sachiez ultérieurement programmer dans n'importe quel langage. Ada n'est qu'un outil qui nous semble bien adapté. Il est normalisé. Il existe des compilateurs gratuits (vous pouvez l'installer chez vous sans piratage). Il est fortement typé, ce qui signifie que beaucoup d'erreurs sont détectées automatiquement. C'est un point très important pour des élèves peu expérimentés. Il n'existe pas de langage idéal. Ada comme les autres a ses points faibles et ses points forts.

Ce premier cours commence par une introduction générale à la programmation et se poursuit par un autre sujet, un rappel de la logique des prédicats qui joue un rôle essentiel dans le reste du cours.

La programmation

Un ordinateur est un outil permettant d'exécuter des programmes. Comme son nom l'indique, un ordinateur sert à mettre de l'ordre. Dans quoi? Dans des données de toutes sortes: des nombres bien sûr, mais aussi des textes, des images, etc. Un programme manipule des données, les crée, les transforme, les déplace.

Prenons l'exemple de quelques programmes courants. Le traitement de textes permet de créer des textes, de les modifier, de les imprimer. Le tableur est spécialisé dans le traitement des nombres, les calculs. Une base de données permet de ranger une grande masse de données et de les ressortir dans différents ordres, selon différents critères. Le navigateur permet de se connecteur à des serveurs web, d'en obtenir des documents et de les afficher de façon agréable (alors que le document transmis n'est qu'une suite de caractères plus ou moins bizarres).

Utiliser ces différents programmes (on parle aussi d'applications) n'est pas le travail de l'informaticien. De nos jours, on utilise des ordinateurs dans presque toutes les professions, et même hors du cadre professionnel, dans la vie courante. Le travail d'un informaticien est la conception et/ou la création de programmes. Il y a là la même différence qu'entre un chauffeur de taxi qui utilise une voiture et un employé de chez Renault qui conçoit ou fabrique des voitures. De même que cet employé a lui aussi une voiture, l'informaticien utilise des programmes qui l'aident dans son activité professionnelle et privée.

Un programme utilise des données et il leur applique un traitement. Cela donne un certain résultat. Créer un programme nécessite d'identifier les données nécessaires, de savoir le résultat qu'on désire atteindre et de connaître un moyen automatique, mécanique de passer des unes à l'autre. Ce moyen mécanique est ce qu'on appelle un *algorithme*. Un langage de programmation est un outil qui permet d'exprimer cet algorithme sous une forme qui est compréhensible à la fois pour l'informaticien et pour la machine. C'est un outil de communication entre l'homme et la machine.

L'ordinateur sait à la base utiliser quelques types de données (par exemple les nombres entiers, les caractères) et faire quelques opérations dessus (par exemple les additions). Un programme peut décrire des données plus complexes (par exemple, un fiche de paye) en combinant plusieurs de ces données élémentaires, ainsi que des traitements complexes en combinants plusieurs des opérations de base (par exemple, calculer toutes les retenues successives sur un salaire brut).

Nous commencerons dès le chapitre prochain à réaliser des programmes, ce qui permettra d'être plus concret.

Rappels de logique

Pour exprimer nos idées, nous énoncons des jugements, par exemple : ``une carré est constitué de quatres cotés égaux''. La logique vise à formaliser le concept de raisonnement (mathématique) et par là même celui d'énoncés, de jugements.

Le *Calcul des propositions* est une formalisation rudimentaire du raisonnement : Les énoncés comme celui de notre exemple sont considérés comme des entités indivisibles -des *propositions*- et l'unique propriété que nous retiendrons d'eux est celle d'être vraie ou fausse. A partir des propositions de base dites *atomiques* que nous noterons `A,B,P,Q` ...il sera possible de construire des propositions plus complexes. On pourra par exemple nier une proposition. A chaque proposition, `P` on associera une autre proposition, la négation de `P` que nous désignerons `\non P`. L'essentiel de cette opération est que `\non P` est fausse quand `P` est vraie et vraie quand `P` est fausse. Ceci peut se résumer par un tableau :

s	(non s)

v	f
f	v

Nous sommes bien en train de décrire un calcul puisque grâce au tableau précédent appelé *table de vérité* nous pouvons calculer la *valeur de vérité* de notre nouvelle proposition à partir des valeurs possibles de la proposition plus simple à partir de laquelle elle est formée. On appellera *connexion* toute opération qui permet de former une nouvelle proposition à partir de propositions données de telle sorte que la valeur de vérité du résultat ne dépende que des valeurs de vérité des propositions sur lesquelles on opère. Nous venons de décrire la connexion `non`. Voici les autres , qui portent sur deux propositions `s` et `t`.

	t	(s et t)
	v	v
	f	f
	v	f
	f	f

s		(s ou t)
v	v	v
v	f	v
f	v	v
f	f	f

s	t	(s =>t)
v	v	v
v	f	f
f	v	v
f	f	v

`et`, `ou` et `=>` correspondent respectivement aux expressions ``et", ``ou" et ``si ...alors" de la langue naturelle. Leurs tables de vérité correspondent à leurs usages courant aux deux remarques suivantes près :

- La table de vérité du `ou` correspond a l'usage inclusif du mot ``ou".
- La valeur de verité *vrai* attribuée au cas où la condition de l'implication est fausse est peu naturelle. Dans le langage courant, on ne s'intéresse à la vérité d'un tel énoncé que lorsque la condition est vraie : ``s'il fait beau je vais à la pêche '' n'a d'interêt pratique que s'il fait beau ...Ici, nous voulons définir un calcul et devons donc décider ce cas. Attribuer la valeur *vrai* dans ce cas est par ailleurs tout à fait raisonnable si l'on songe que le fait qu'il ne fasse pas beau ne démontre en rien la fausseté de notre première déclaration d'intention.

Nous allons maintenant définir plus formellement le Calcul des Propositions.

Le langage du Calcul des propositions

On se donne un ensemble de symboles de propositions de base : `A,B,C,`... Cet ensemble constitue l'ensemble des *formules atomiques*. On se donne un ensemble de symboles -les connecteurs- permettant de former les connexions : non, et, =>, ou Les formules sont construites à l'aide des règles suivantes :

1. Les formules atomiques sont des formules.
2. Si `A` et `B` sont des formules alors `A ou B, A et B, A => B` et `non A` sont des formules.

La notion de vérité

Soit `F` une formule et `A_1,..., A_n` les symboles de proposition qu'elles contient et les symboles `vrai` et `faux`. Etant donnée une valuation de `{A1,..., An}` dans `{vrai, faux}`, nous savons calculer, grace aux tables de vérité, la valeur de vérité de `F`.
`F` est une *tautologie* si et seulement si elle prend la valeur `vrai` pour chacune des valuations de `{A1,..., An}` dans `{vrai, faux}` (il y en a 2^n).

La notion de Déduction

Dans les paragraphes précédent, nous nous sommes intéréssés à la notion de vérité d'une proposition. Nous allons maintenant nous intérésser à la notion de raisonnement correct. En voici un exemple :
Le chat mange ou dort.
S'il dort il ronronne.
S'il ronronne il est content.
En ce moment, il ne mange pas.
Donc, en ce moment il est content.
Ce raisonnement est correct au sens où la conclusion de cet ensemble d'énoncé se déduit des quatre premières hypothèses.
Quand dirons nous qu'une proposition `P` se *déduit* des propositions `P1`,... `Pn`? On pourrait adopter la position suivante : supposant `P1`, ...`Pn` formalisées dans le calcul des propositions, `P` se déduit de `P1`,..., `Pn` si la proposition `P1`,..., `Pn => P` est une tautologie. Cette optique ne nous satisfait pas. Pour justifier une déduction, on en montre une *Preuve*, qui conduit des hypothèses à la conclusion par une suite d'étapes évidentes. Se dessine ici une autre approche de la logique: au lieu de s'interesser à ce qui est vrai on s'interesse à ce qui est démontrable. Ce que l'on va chercher à formaliser sera cette fois la notion de *preuve*.

Nous allons décomposer le raisonnement en étapes élémentaires qui correspondent au raisonnement sur chacun des connecteurs. A chaque connecteur *c* correspond deux types de raisonnement :

- Le premier qui permet de *déduire* une formule ayant *c* comme connecteur principal. Par exemple ``de `A` et `B` on déduit `A et B`'' En général, ce raisonnement s'utilise lorsque l'on veut démontrer une formule de la forme `A et B`. Il s'interprète donc de la façon suivante : pour démontrer `A et B`, il suffit de démontrer `A` et de démontrer `B`.
- Le second type de raisonnement permet d'*utiliser* des énoncés ayant *c* comme connecteur principal pour déduire d'autres formules. Par exemple `` de `A et B` on conclut `A`'' .

Voici les raisonnements élémentaires attachés à chacun des connecteurs :

conjonction
Pour démontrer `A et B` il suffit de démontrer `A` et de démontrer `B`.
De `A et B` on conclut `A` et on conclut `B`.

disjonction
Pour démontrer `A ou B` il suffit de démontrer `A` ou de démontrer `B`.
De `A ou B` on conclut `C` dès lors qu'on possède des démonstrations de `C` sous hypothèse `A` et de `C` sous hypothèse `B`. En d'autres termes on utilise un énoncé de la forme `A ou B` en raisonnant par cas : premier cas = `A` est vrai. deuxième cas = `B` est vrai.

implication
Pour démontrer `A => B` on démontre `B` en supposant `A`.
De `A` et de `A => B` on déduit `B`.

négation
pour démontrer `non A` on démontre une contradiction sous l'hypothèse `A`.
De `A` et de `non A` on déduit une contradiction.

Exemple

Les problèmes simples de logique propositionnelle peuvent se traiter au moyen de tables de vérité. Ces tables permettent de déterminer dans quel cas une formule logique est vraie ou fausse.

Prenons un exemple: on cherche à déterminer dans quels cas la formule `A et (B ou (non A))` est vraie.

On va faire une table de vérité en envisageant tous les cas possibles de valeurs pour `A` et `B`. Puis on calculera successivement les valeurs de vérité de toutes les sous-formules de `A et (B ou (non A))`.

Toutes les combinaisons possibles pour `A` et `B` sont: `A` et `B` sont vrais, `A` est vrai et `B` faux, `A` est faux et `B` vrai, `A` et `B` sont faux. Il y a donc quatre cas possibles. On aura quatre lignes dans notre table de vérité.

Les sous-formules de `A et (B ou (non A))` qu'on doit étudier sont: `A`, `B`, `non A`, `B ou (non A)` et `A et (B ou (non A))`. Cela nous donnera les 5 colonnes du tableau.

Dans la formule, on trouve trois opérateurs logiques: et, ou, non. Pour chacun, on a une table de vérité que l'on connait d'avance.

p	q	p et q
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	faux
faux	faux	faux

p	q	p ou q
vrai	vrai	vrai
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

p	non p
vrai	faux
faux	vrai

Pour notre formule, nous allons donc avoir une table de vérité à quatre lignes (plus les titres) et cinq colonnes.

1	2	3	4	5
A	B	non A	B ou (non A)	A et (B ou (non A))

On commence en remplissant les deux premières colonnes avec toutes les combinaisons possibles.

1	2	3	4	5
A	B	non A	B ou (non A)	A et (B ou (non A))
vrai	vrai			
vrai	faux			
faux	vrai			
faux	faux			

On va ensuite calculer la troisième colonne. Pour cela, on va appliquer l'opérateur non sur le contenu de la colonne 1. On va donc temporairement identifier le p de la table de vérité de non et le A de notre formule.

Pour la première ligne, dans la colonne 1, on a vrai. On va chercher dans la table de vérité quelle est la valeur de vérité de non p quand p vaut vrai. On trouve faux. Donc on met faux dans la case première ligne troisième colonne du tableau.

1	2	3	4	5
A	B	non A	B ou (non A)	A et (B ou (non A))
vrai	vrai	faux		
vrai	faux			
faux	vrai			
faux	faux			

On agit de même pour toute la colonne.

1	2	3	4	5
A	B	non A	B ou (non A)	A et (B ou (non A))
vrai	vrai	faux		
vrai	faux	faux		
faux	vrai	vrai		
faux	faux	vrai		

On veut maintenant calculer la quatrième colonne du tableau. Il s'agit de faire un ou entre la deuxième et la troisième colonne. On va utiliser la table de vérité du ou en identifiant temporairement p à `B` et q à `non A`.

La première ligne consiste à calculer la valeur de vrai ou faux. On va chercher dans la table de vérité du ou ce que vaut p ou q quand p vaut vrai et q faux. Cela vaut vrai. On agit de même pour toute la colonne.

1	2	3	4	5
A	B	non A	B ou (non A)	A et (B ou (non A))
vrai	vrai	faux	vrai	
vrai	faux	faux	faux	
faux	vrai	vrai	vrai	
faux	faux	vrai	vrai	

Pour calculer la dernière colonne, on fait un et entre les colonnes 1 et 4. Cela donne:

1	2	3	4	5
A	B	non A	B ou (non A)	A et (B ou (non A))
vrai	vrai	faux	vrai	vrai
vrai	faux	faux	faux	faux
faux	vrai	vrai	vrai	faux
faux	faux	vrai	vrai	faux

On peut conclure de cette table que `A et (B ou (non A))` est vraie seulement quand `A` et `B` sont vrais tous les deux.

On peut aussi montrer avec une table de vérité que `A et (B ou (non A))` est équivalent à `A` et `B`. On calcule séparément la valeur de vérité de chacune des deux propositions et on constate qu'elles ont toujours la même valeur.

1	2	3	4	5	6
A	B	non A	B ou (non A)	A et (B ou (non A))	A et B
vrai	vrai	faux	vrai	vrai	vrai
vrai	faux	faux	faux	faux	faux
faux	vrai	vrai	vrai	faux	faux
faux	faux	vrai	vrai	faux	faux

La limite de la méthode des tables de vérité est la taille des tables. Quand on a deux propositions de base, `A` et `B`, il faut quatre lignes. Si on en a trois, `A`, `B`, `C`, il faut 8 lignes. Si on en a quatre, il faut 16 lines. Si on en a n, il faut 2 puissance n lignes. En pratique, il est difficile de dépasser 4.

Exercice 1.1

Question 1

Donnez les tables de vérité des propositions suivantes:

- (A et B) ou (non B)
- (non A) et (non B)
- (A et B) et (non A)
- non (A et (non B))

Question 2

Pour chacune de ces proposition, trouvez une autre proposition plus simple (c'est à dire avec moins d'opérateurs) équivalente. Rappel: deux propositions sont équivalente si elles ont la même valeur de vérité dans tous les cas possibles.

Exercice 1.2

Soient les propositions:

- (P) il pleut
- (S) la sortie est annulée
- (M) etre mouillé
- (R) rester à la maison

Question 1

Traduire les phrases suivantes en une proposition logique, en utilisant P, S, M et R.

- Si il pleut et si je reste à la maison, je ne serai pas mouillée.
- Si il pleut mais si je reste à la maison, je ne serai pas mouillée.
- Si il pleut et si la sortie n'est pas annulée ou si je ne reste pas à la maison, je serai mouillée.
- que la sortie soit annulée ou non, s'il pleut, je reste à la maison.

Question 2

Dans quels cas la proposition *Si il pleut mais si je reste à la maison, je ne serai pas mouillée* est vraie?

Pour répondre à cette question, on fera une table de vérité où on calculera la valeur de la proposition en fonction des valeurs de P, M et R.

Question 3

Si la proposition *Si il pleut et si je reste à la maison, je ne serai pas mouillée* est vraie et si il ne pleut pas, est-ce que je reste à la maison?

Il ne s'agit pas de faire un raisonnement de la vie courante, mais un raisonnement logique. Pour cela, on fera une table de vérité de la proposition correspondant à *Si il pleut et si je reste à la maison, je ne serai pas mouillée* (ce que vous avez répondu à la question 1.1) et on regardera les lignes telles que cette proposition est vraie et P est fausse. On regardera alors la valeur de R dans ces lignes.

Et on en tirera une conclusion.

Exercice 1.3

Soient a, b et c trois entiers positifs.

Question 1

Montrez que la proposition `(a<b) => (a = b)` a pour négation `a<b`.

Pour cela, vous pouvez donner un nom aux deux propositions élémentaires `a<b` et `a=b` (élémentaire d'un point de vue logique, car < et = ne sont pas des opérateurs logiques. Ce sont des opérateurs arithmétiques).

Ensuite, faites la table de vérité des deux propositions et utilisez vos connaissances sur l'arithmétique pour exclure certains cas qui ne sont pas compatibles avec l'arithmétique. Cela revient à éliminer des lignes de la table.

Tirez la conclusion.

Question 2

Montrez que la proposition `(a<b) => a = a` a pour négation fleche  `a`.

Etapes de la production

Produire un programme se fait par étapes successives.

- spécification
- conception
- codage
- compilation
- test
- mise en service
- maintenance

La première étape, la spécification, consiste à décrire ce que l'on souhaite obtenir. A ce niveau, il ne s'agit pas de dire comment, mais juste de décrire le comportement attendu du programme, ce qui nécessite de détailler un peu ce qu'il faut lui apporter comme données (ce qui va rentrer dans le programme) et ce que sera le résultat (ce qui va sortir). Dans la plupart des exercices qui vous seront proposés au cours de l'année, la spécification sera l'énoncé de l'exercice.

La conception consiste à décrire le programme, c'est à dire les moyens à employer pour obtenir le résultat escompté. Dans le cas de problèmes complexes, il faut opérer par étapes successives. La conception doit déboucher sur un algorithme, c'est à dire un procédé mécanique permettant d'obtenir le résultat à partir des données.

Le codage, ou programmation au sens strict, consiste à exprimer l'algorithme dans un langage de programmation donné. Cela donne le code du programme. Ce code est une suite de caractères que l'on peut taper sur un clavier d'ordinateur. Dans la suite du cours, nous aurons tendance à faire en même temps la conception et le codage, ce qui n'est possible que pour de petits programmes simples. Cela a l'avantage de nous éviter d'utiliser un premier langage pour faire la conception et un second pour la programmation.

Sous sa forme de code Ada, le programme n'est encore pas compréhensible par l'ordinateur. Il faut une étape de traduction dans une forme directement compréhensible par le processeur. Cette traduction est effectuée par un programme qui s'appelle un compilateur. Ce compilateur prend comme donnée un programme Ada et donne comme résultat le même programme dans la langue maternelle du processeur (par exemple, un pentium). C'est ce qu'on appelle le code exécutable. La compilation est une opération automatique et suffisament simple pour qu'on ne la considère généralement pas comme une étape de travail à proprement parler.

Une fois compilé, le programme peut être exécuté. La première chose à faire est de le tester pour s'assurer qu'il fonctionne convenablement. Même à l'échelle de nos petits programmes de début d'année, cette étape est importante. Elle l'est encore bien plus dans un cadre industriel.

Une fois convenablement testé, le programme peut être donné aux utilisateurs et être exploité.

L'activité de maintenance consiste à maintenir le programme en état de marche, en corrigeant les erreurs au fil de leurs apparitions. Des erreurs (les fameux bugs), il y en a presque toujours dans des programmes un tant soit peu complexe. Par ailleurs, il arrive que le programme doive être adapté à un changement de son environnement (par exemple, un changement d'ordinateur).

La maintenance d'un programme est suffisament coûteuse et importante pour qu'il soit intéressant de la prendre en compte par avance. En effet, il faut toujours prendre soin de la lisibilité de son programme, c'est à dire de la facilité de lecture au moment où on voudra corriger une erreur ou réaliser une adaptation.

Dans un cadre industriel, cela peut se produire des années plus tard, et les personnes amenées à réaliser la maintenance peuvent ne pas être celles qui ont réalisé le programme.

Dans notre contexte scolaire, nous n'avons pas les mêmes contraintes, mais nous auront quand même toujours la préoccupation de privilégier la lisibilité par rapport à d'autres qualités telles que la concision. Nous ne demanderons pas de documentation pour les programmes, bien que ce soit la base de la maintenance.

Concrètement

Nous allons reprendre les étapes de création d'un programme d'un point de vue très concret, pour une application immédiate.

La spécification sera donné par un cours énoncé en français, par exemple: écrire un programme qui affiche la somme de deux nombres tapés au clavier.

Il faut écrire un programme Ada. Dans un premier temps, on peut le griffoner approximativement sur un bout de papier. Il s'agit d'un brouillon. On peut tout aussi bien réaliser ce brouillon à l'écran. C'est une question de goût et d'habitude. Dans l'un et l'autre cas, il s'agit de jeter des idées un peu en vrac et de les organiser progressivement.

Il faut créer un fichier qui contienne le programme Ada. Pour cela on utilise un éditeur de texte. C'est une sorte de traitement de texte, mais dont le but n'est pas de mettre en page le texte en vue de l'imprimer, mais juste de créer un fichier. Dans un éditeur de texte, on ne modifie pas l'apparence des caractères (taille, souligné, italique, gras), on ne donne pas de structure (page, table des matières, titres). On se contente de taper les caractères les uns après les autres.

Sur votre PC, vous pouvez utiliser `Edit` dans une fenêtre `ms-dos`, ou alors l'éditeur fourni avec le compilateur Gnat, ce qui est plus pratique. Nous verrons cela dans les exercices 2.1 et 2.2.

Une fois le programme tapé, on le sauve sur disque, ce qui a pour effet de créer (ou éventuellement mettre à jour) le fichier correspondant. Ce fichier en langage Ada doit avoir l'extension .adb (attention aux éditeurs qui veulent vous imposer l'extension .txt!).

Ensuite la compilation se fait soit à l'aide d'une commande `ms-dos` (`gnatmake`), soit en cliquant sur une icône de Gnat. Souvent, la compilation se passe mal, parce qu'il y a des erreurs dans le programme. Ce n'est pas du bon Ada. Il faut le corriger dans l'éditeur de texte et recommencer la compilation.

Quand il n'y a plus d'erreur dans le programme, le compilateur crée un fichier exécutable `ms-dos`, un fichier ayant l'extension `.exe`.

C'est ce fichier qu'on utilise pour exécuter le programme et le tester. Quand on l'a testé suffisament pour être convaincu qu'il marche bien, on peut considérer le travail comme terminé.

Structure d'un programme

Voyons un premier exemple de programme, celui qui affiche la somme de deux nombres tapés au clavier.

```
with Ada.Text_Io; use Ada.Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
```

```
procedure Somme_2_Entiers is
   X: Integer;
   Y: Integer;
   Somme: Integer;
begin
   Put("Entrez un premier nombre : ");
   Get(X);
   Put("Entrez un second nombre : ");
   Get(Y);
   Somme := X + Y;
   Put("La somme vaut : ");
   Put(Somme);
   New_Line;
end Somme_2_Entiers;
```

La structure d'un programme est la suivante:

```
with Text_Io;                 |
use Text_Io;                  | entete du programme
with Ada.Integer_Text_Io;     |
use Ada.Integer_Text_Io;      |

procedure nom_du_programme is
                              |
                              | partie declaration
                              |
begin
                              |
                              | partie instruction
                              |
end nom_du_programme;
```

Le rôle des différentes parties:

- entête du programme: on n'expliquera pas son rôle ni son sens pour l'instant. Cela deviendra clair dans quelques mois. D'ici là, il suffit de voir cela comme une formule que l'on met systématiquement en tête de tous les programmes. Cela permet d'utiliser les instructions `Put`, `Get` et `New_Line` que nous présenterons dans ce chapitre.
- la partie déclaration contient des déclarations diverses. Dans un premier temps, nous ne ferons que des déclarations de variables. Pour chaque variable, on déclare son nom et son type. On verra ci-dessous ce que sont un nom et une variable.
- la partie instruction contient une suite d'instructions exécutées l'une après l'autre lors de l'exécution du programme. Toutes les variables apparaissant dans cette partie doivent avoir été déclarées dans la partie déclaration.

Nous allons souvent faire un parallèle entre un programme et une recette de cuisine. Dans une recette, souvent, on commence par une liste des ingrédients et on donne après une suite des opérations à réaliser avec ces ingrédients. La partie déclaration, c'est la liste des ingrédients du programme. La partie instruction, c'est la suite des opérations. Il ne doit pas y avoir dans les opérations une utilisation d'un ingrédient qui ne serait pas dans la liste.

Une variable, c'est un espace de rangement dans lequel on peut stocker une certaine valeur. Dans notre exemple, nous manipulons des nombres entiers. Les variables que l'on utilise sont des espaces dans lesquels on peut ranger un entier. Pour désigner chacun de ces espaces, on utilise un nom. C'est ce nom qui doit être déclaré.

Lorsqu'on déclare une variable, on a le choix du nom, on peut l'appeler comme on veut, mais dans certaines limites. Tout ne peut pas servir de nom.

Un nom correct (on appelle cela un *identificateur*) a les caractéristiques suivantes:

- il ne comprend que des lettres, des chiffres ou le caractère souligné `_`.
- il commence par une lettre
- un caractère souligné est nécessairement suivi d'une lettre ou d'un chiffre.

Le caractère souligné sert à relier deux mots que l'on voudrait séparer par un espace (mais on ne peut pas mettre un espace à l'intérieur d'un identificateur). Par exemple: `premier_programme` ou `somme_2_entiers`.

Par ailleurs, il est préférable de ne pas utiliser des caractères accentués (des lettres avec accent ou cédille) dans un programme.

Dans la partie instruction, le nom de la variable pourra être utilisé pour désigner ce qui est rangé dedans.

Dans la déclaration apparaît aussi le type de la variable. Dans notre exemple, il s'agit d'`integer` (nombre entier en anglais). Ce type définit le genre de choses que l'on pourra ranger dans la variable.

Notion de type

Un type est un ensemble de valeurs.

Pour commencer, nous allons utiliser les types suivants:

- integer: l'ensemble des nombres entiers
- float: l'ensemble des nombres avec un point décimal (virgule)
- character: l'ensemble des caractères que l'on peut taper au clavier.
- boolean: l'ensemble des valeurs de vérité.
- string: l'ensemble des chaines de caractères (suites de caractères).

De ces ensembles, seul celui des boolean est facile à donner en extension, en énumérant tous ses éléments, car il n'y en a que deux: `{true, false}` (true signifie vrai et false faux en anglais).

Le type des character contient 256 éléments, dont toutes les lettres majuscules et minuscules et les caractères de ponctuation et autres (dièse, dollar, etc). Il contient aussi quelques caractères bizarres que l'on ne peut pas afficher. Les valeurs du type character sont notées entre apostrophes, par exemple: `'A', 'a', '3', '='`. Il ne faut pas confondre `X` qui est un nom de variable avec `'X'` qui est un character.

Les autres types contiennent beaucoup plus d'éléments.

Les ensembles de valeur que sont les types sont utilisés en Ada dans les déclaration. Quand une variable est déclarée d'un certain type, cela signifie qu'elle contiendra à tout moment une valeur appartenant à cet ensemble.

Par exemple, quand on déclare:
```
X: integer;
```
on veut dire par là qu'à tout moment dans le programme, il y aura un entier dans X.

Cette déclaration permet de réserver pour X une place en mémoire suffisante pour stocker un entier quel qu'il soit. Dans beaucoup de cas, ce sera 4 octets, c'est à dire 32 bits. Pour nous programmeur, il n'est généralement pas très important de savoir combien de bits il y a. Mais quel que soit le nombre de bits de mémoire réservés pour X, il faut bien comprendre que chaque bit est soit à 0 soit à 1 et qu'il n'est pas possible que X ne contienne rien. Il y a toujours un entier dans X. Mais entre la déclaration et le moment où on met un nombre dans X, il y a un certain entier qu'on ne connait pas.

Le compilateur vérifie que les déclarations et les instructions sont cohérentes du point de vue du type. Par exemple, si on déclare que X est un integer et qu'on essaie de mettre un boolean dedans, cela provoque une erreur à la compilation.

Premières instructions

Les instructions que nous allons voir cette semaine sont des instructions de base, simples. La semaine prochaine, nous verrons des instructions plus complexes qui permettent d'entrer réellement dans le monde de la programmation.

Nous avons des instructions d'affichage à l'écran: `Put` et `New_Line`. `Put` permet d'afficher ce qu'on lui donne comme argument entre parenthèse. Cela peut être un entier, un character, une string. Cela peut également être un nom de variable, auquel cas ce qui est affiché n'est pas ce nom, mais le contenu de la variable.

Par exemple:

- `Put(18);` affiche l'entier 18
- `Put('A');` affiche le caractère `A` (sans les apostrophes)
- `Put("Ok");` affiche la chaînede caractères `Ok` (sans les guillemets)
- `Put(X)` affiche le contenu de la variable `X`, par exemple 15. Cela ne marche que si `X` est le nom d'une variable déclarée dans le programme et si elle est d'un des types integer, character ou string.

`New_Line` est une instruction qui passe à la ligne à l'écran.

La séquence d'instructions:

```
Put("Bonjour ");
Put("monsieur ");
New_Line("le Maire");
```

Affiche le texte:

```
Bonjour monsieur
le Maire
```

L'instruction `Get` permet de transférer quelque chose du clavier vers le programme. Ce quelque chose doit être une valeur d'un des types integer, character ou string. Cette valeur est mise dans une variable qui doit être précisée entre parenthèse, comme argument de `Get`.

Par exemple `Get(X)` va lire quelque chose au clavier et le mettre dans la variable `X`. Supposons que cette variable soit de type integer. Alors, il faut qu'un nombre soit tapé au clavier, faute de quoi c'est une erreur et le programme s'arrête.

La dernière instruction de la semaine est la plus importante, il s'agit de l'affectation. Elle permet de changer le contenu d'une variable.

Exemple: `Somme := X + Y;`

A gauche, il y a le nom de la variable (`Somme`). Au milieu, le signe `:=` qu'on peut lire comme *devient égal*. A droite, il y a le calcul à faire pour obtenir la nouvelle valeur à mettre dans `Somme`. Ici, il s'agit

d'additionner le contenu de `X` et le contenu de `Y`. L'instruction complète peut se lire: *Somme devient égale à ce que vaut `X` plus ce que vaut `Y`.*

L'ancienne valeur de `Somme` est perdue. Il n'est pas possible de la retrouver.

Le calcul de la partie droite de l'affectation est effectuée avant de modifier la valeur de la variable, si bien que c'est son ancienne valeur qui est prise en compte si elle apparaît en partie droite.

Prenons un exemple d'affectation très classique, que nous verrons dix mille fois au cours de l'année:
`X := X + 1;`
Le `X` à gauche du `:=` désigne la variable dont on veut changer la valeur. Le `X` à droite du `:=` désigne la valeur contenue dans `X` avant qu'on effectue le changement. C'est un certain entier auquel on veut ajouter 1.
`X := X + 1;` se lit: `X` devient égal à la valeur de `X` plus un. Autrement dit: ajouter 1 à la valeur de `X`.

Sémantique d'un programme

On appelle sémantique le sens, la signification d'un programme. Ce sens est lié à ce qui se passe à l'exécution. La façon la plus simple de décrire ce sens est de faire référence à des états de la mémoire.

Nous avons vu qu'un programme comporte des variables qui sont déclarées et que ces variables sont des espaces en mémoires qui contiennent à tout moment une valeur d'un certain type. La valeur est susceptible de changer au fil du temps, notamment du fait des affectations et des `Get`.

Un état de la mémoire est caractérisé par l'ensemble des valeurs stockées dans les variables à un moment donné. Certaines instructions changent la valeur d'une ou plusieurs variables. L'exécution de l'instruction fait alors passer d'un état de la mémoire à un autre. L'exécution du programme peut se décrire au moyen de la succession des états mémoire.

Certaines instructions ne changent pas l'état de la mémoire mais des choses extérieures au programme tels que le clavier et l'écran. Ces instructions sont ce qu'on appelle des *effets de bord*. On parle aussi d'*entrées-sorties*, le clavier (et plus tard dans le cours, des fichiers) permettant de faire entrer des données dans le programme et l'écran (plus tard, les fichiers) permettant d'afficher des résultats et des messages.

La sémantique d'un programme s'exprime au moyen de la notion d'état de la mémoire, mais doit également prendre en compte les effets de bord.

Nous allons prendre l'exemple de la sémantique de notre petit programme d'addition. Pour cela, nous allons numéroter ses lignes.

```
1   with Ada.Text_Io; use Ada.Text_Io;
2   with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3   procedure Somme_2_Entiers is
4     X: Integer;
5     Y: Integer;
6     Somme: Integer;
7   begin
8     Put("Entrez un premier nombre : ");
9     Get(X);
10    Put("Entrez un second nombre : ");
11    Get(Y);
12    Somme := X + Y;
13    Put("La somme vaut : ");
14    Put(Somme);
15    New_Line;
```

```
16  end Somme_2_Entiers;
```

L'exécution proprement dite commence à la ligne 8, les déclarations ne font que définir les variables que l'on utilise. Ici, il y en a 3: `X`, `Y` et `Somme`. Lorsque le programme débute, chacune de ces variables contient un entier, mais on ne sait pas lequel. Dans le doute, nous noterons par un point d'interrogation cet entier inconnu, qui existe, mais qui ne sera pas forcément le même à chaque exécution.

Le tableau suivant retrace une exécution du programme avec l'état mémoire, l'écran et le clavier. En début de ligne, nous notons le numéro de la ligne de programme exécutée pour atteindre l'état correspondant.

num	état	clavier	écran
	X vaut ?, Y vaut ?, Somme vaut ?		
8	X vaut ?, Y vaut ?, Somme vaut ?		Entrez un premier nombre :
9	X vaut 5, Y vaut ?, Somme vaut ?	5	5
10	X vaut 5, Y vaut ?, Somme vaut ?		Entrez un second nombre :
11	X vaut 5, Y vaut 9, Somme vaut ?	9	9
12	X vaut 5, Y vaut 9, Somme vaut 14		
13	X vaut 5, Y vaut 9, Somme vaut 14		La somme vaut :
14	X vaut 5, Y vaut 9, Somme vaut 14		14
15	X vaut 5, Y vaut 9, Somme vaut 14		(passage à la ligne)

On voit dans cet exemple qu'il y a trois états successifs, avec de moins en moins d'incertitude, c'est à dire avec de moins en moins de `?`. Notons que les valeurs inconnues `?` ne sont jamais affichées ni utilisées. Ces inconnues n'influencent donc pas le résultat de l'exécution. Ni l'affichage ni l'état de la mémoire n'en dépendent.

Ce tableau retrace une exécution du programme. Ce programme peut avoir bien d'autres exécutions, selon ce qu'on va entrer au clavier. La sémantique ou sens du programme comprend toutes ces exécutions potentielles. Il faut donc un outil plus puissant que notre rudimentaire tableau pour exprimer cette sémantique. Mais cela dépasse le cadre du cours Algorithmique-Programmation pour lequel nous nous contenterons d'une approche simple

Exercice 2.1: Gnat brut de fonderie

Dans un premier temps, nous allons utiliser le Gnat sans son interface graphique. Cela se rapproche de ce qui se pratique sous Unix, dans les salles de TP du CNAM.

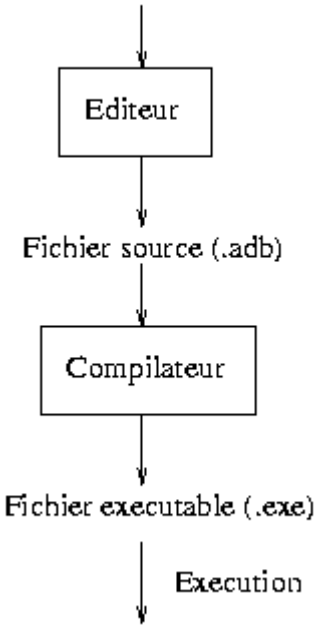
Dans cet exercice, nous allons taper et compiler un petit programme Ada tout simple, dont le code suit:

```
with text_io; use text_io;
procedure premieressai is
begin
   put("Bonjour world");
end premieressai;
```

Il faut d'abord taper le programme en utilisant un éditeur de texte. Le programme est alors sauvé dans un fichier. Le programme doit ensuite être compilé, c'est à dire traduit d'Ada en langage machine, directement compréhensible par le processeur. Cette traduction crée un nouveau fichier qui contient la

traduction. C'est un fichier exécutable ms-dos. Ce programme peut être exécuté autant de fois que l'on veut.

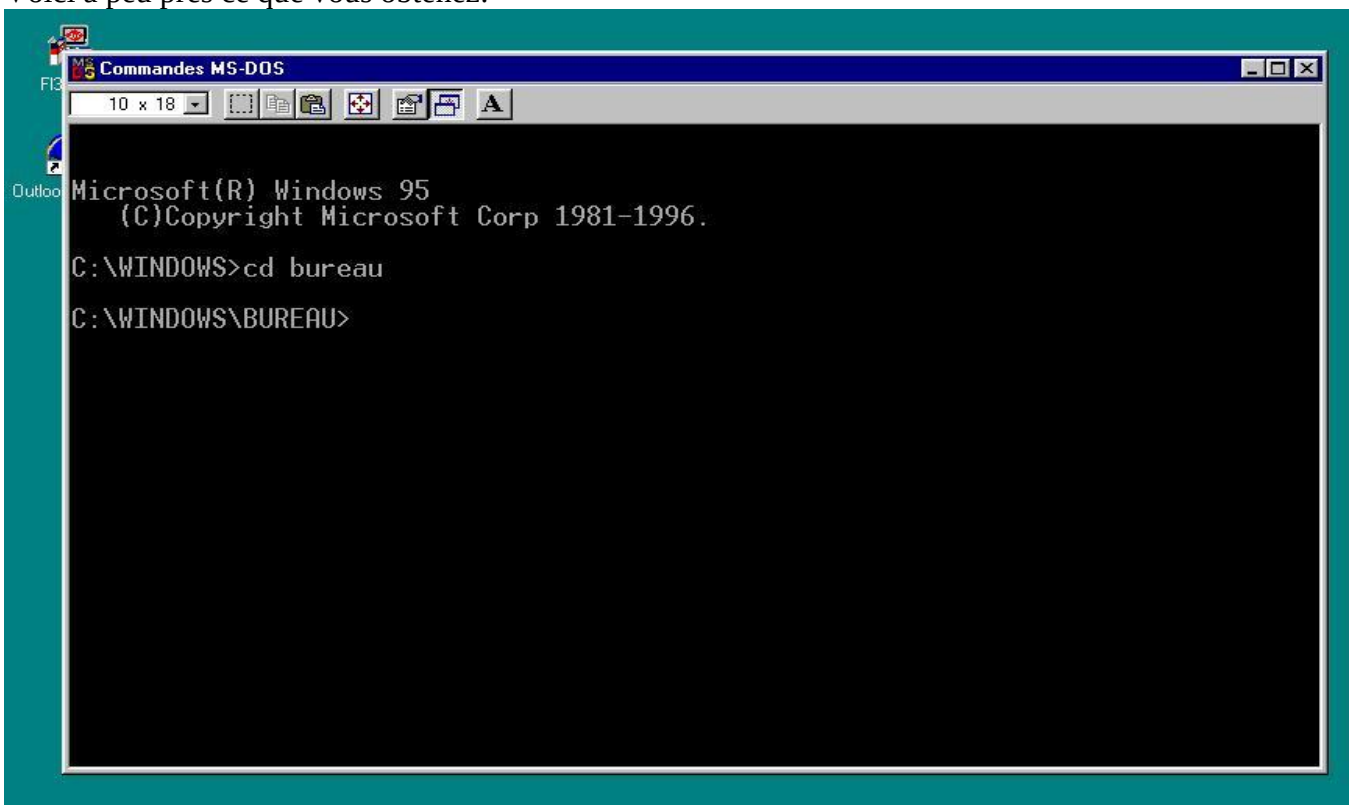
Voici un schéma retraçant les différentes étapes.



Nous allons réaliser les différentes étapes pas à pas. Lisez peut-être une fois cette page, puis reprenez pas à pas toutes les étapes sur votre ordinateur.

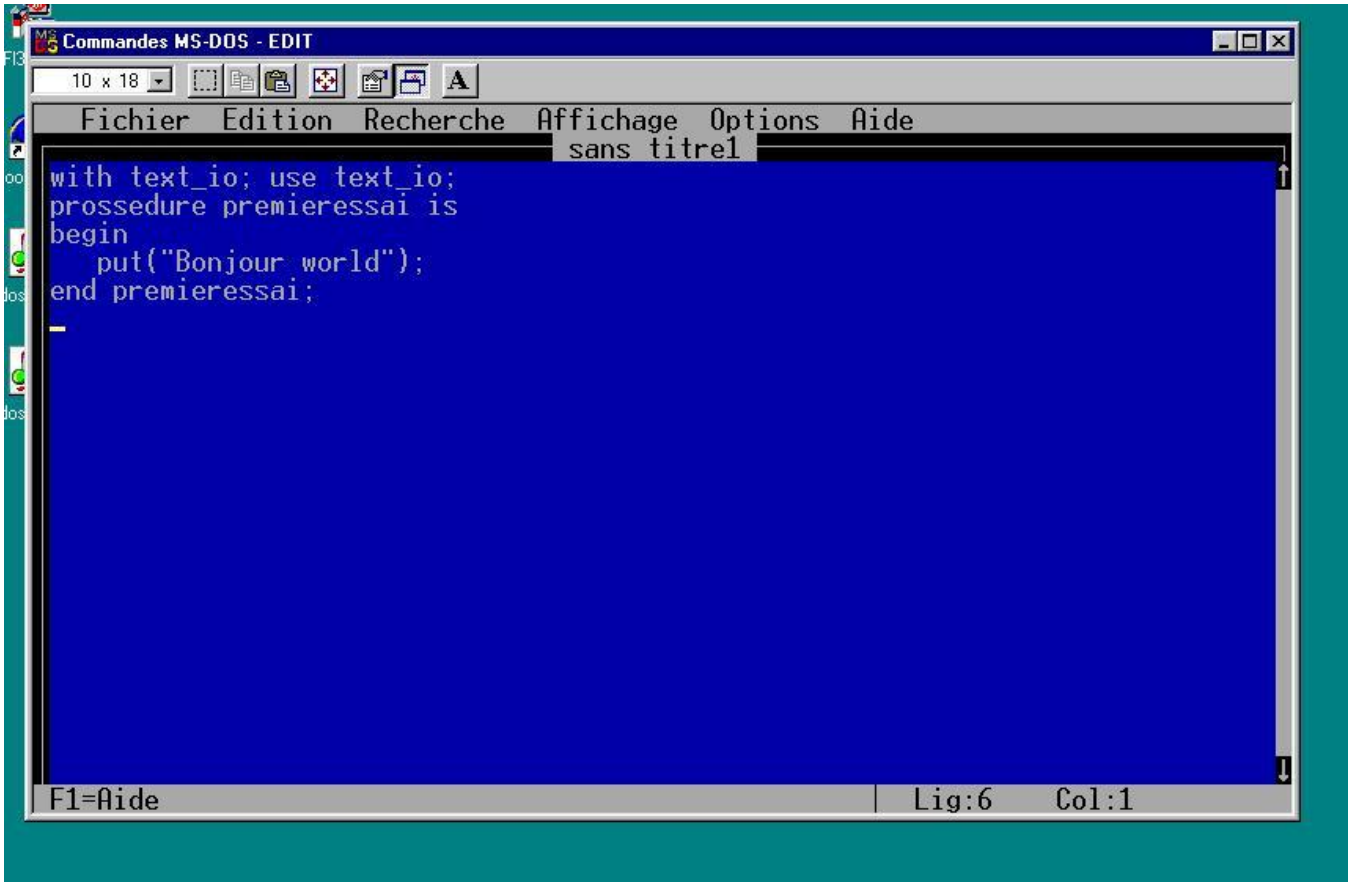
D'abord, ouvrez une fenêtre ms-dos en choisissant la bonne option du menu `demarrer` de windows. Allez dans le bureau en faisant `cd bureau` (ou au pire `cd c:\windows\bureau`).

Voici à peu près ce que vous obtenez:



Lancez l'éditeur edit en tappant `edit` dans la fenêtre dos. Photo optionnelle.

Tappez le petit programme.

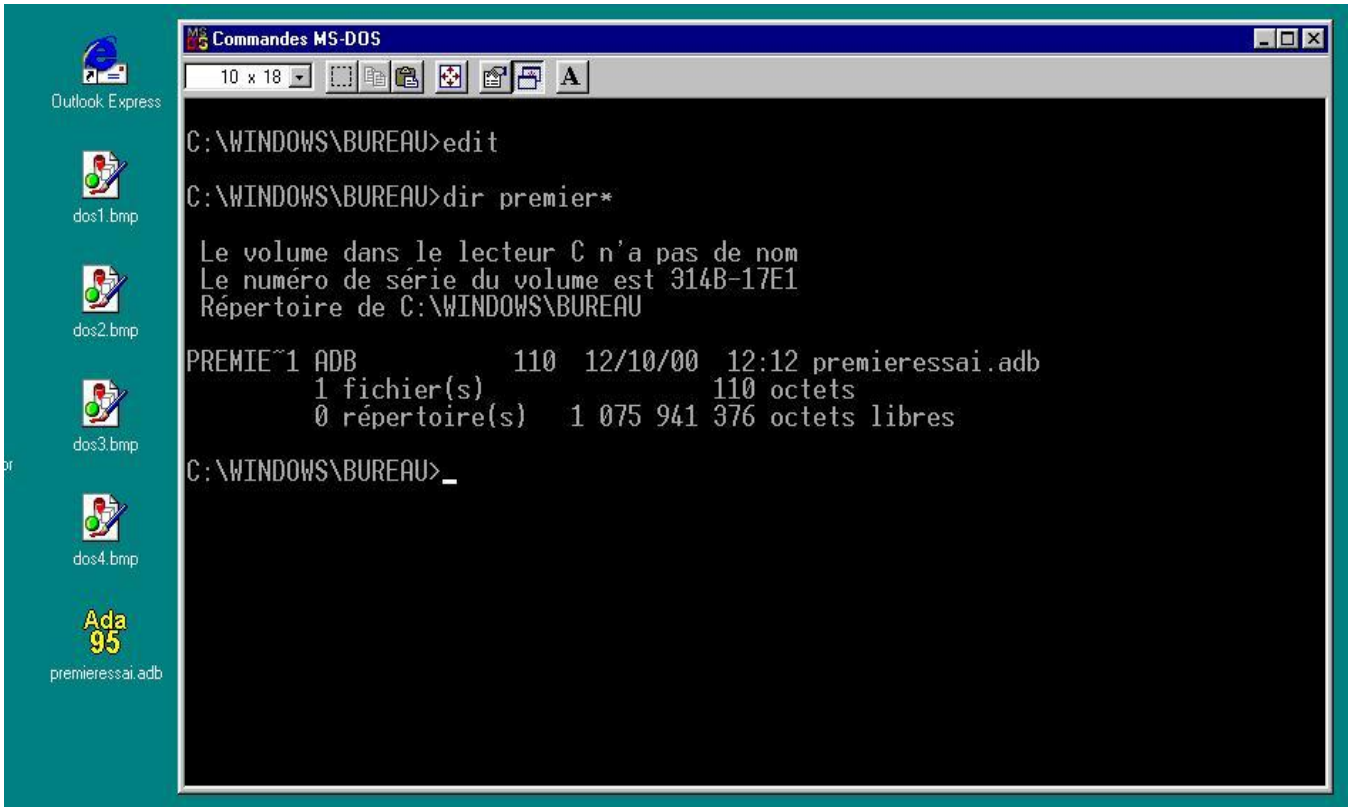


Il faut sauver ce programme dans un fichier. Par convention, le nom du fichier doit être le nom de la procédure écrit en minuscules suivi de l'extension `.adb`.

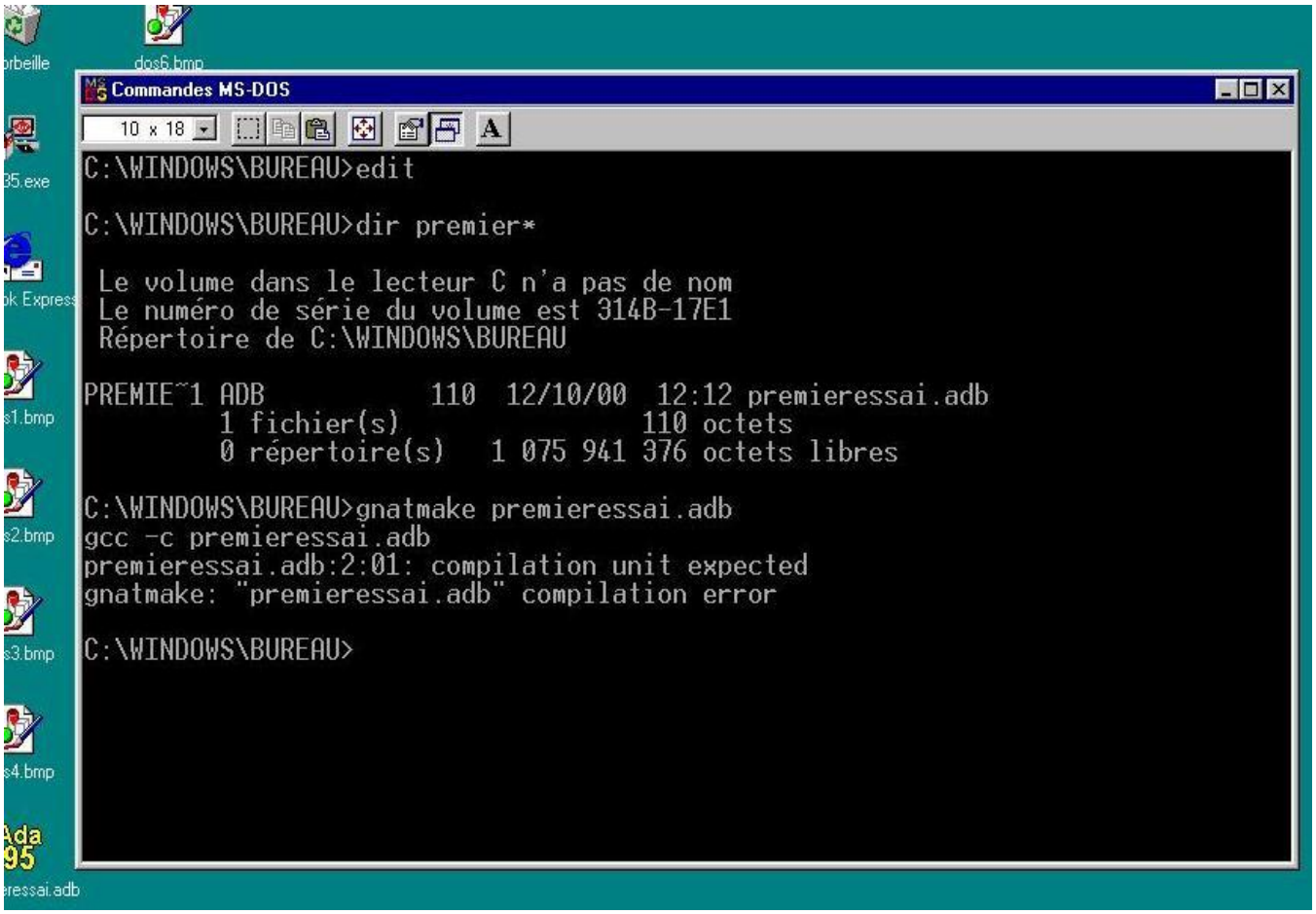
Pour sauvez, utilisez l'option `Enregistrer sous` du menu `Fichier` Photo optionnelle et entrez le nom sans oublier l'extension (`premieressai.adb`) Photo optionnelle.

Quittez edit en utilisant l'option `Quitter` du menu `Fichier`. Photo optionnelle.

Si on tape la commande `dir` de ms-dos permettant d'avoir la liste des fichiers du répertoire, on voit qu'il existe un fichier `premier.adb`.



Pour compiler, nous allons utiliser la commande `gnatmake`. C'est une commande qu'on peut utiliser dans ms-dos pour exécuter le compilateur. Il faut faire suivre cette commande d'un espace et du nom de fichier.

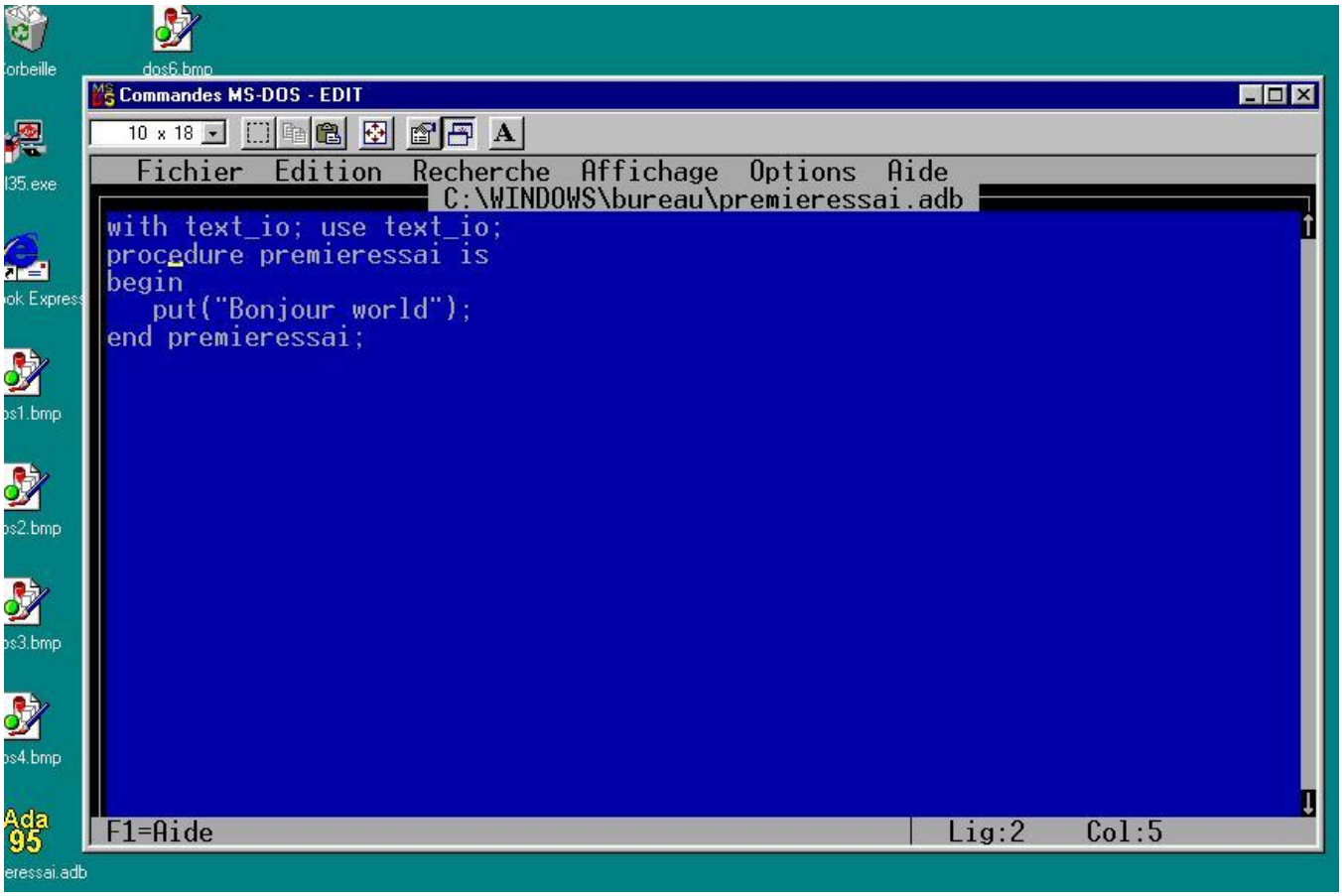


On voit sur cette image un exemple de compilation qui ne marche pas: il y a une erreur dans le fichier: le mot-clé `proc\'edure` est mal orthographié. Notre programme n'est pas de l'Ada correct. Le compilateur signale les erreurs avec des messages plus ou moins explicites et en anglais.

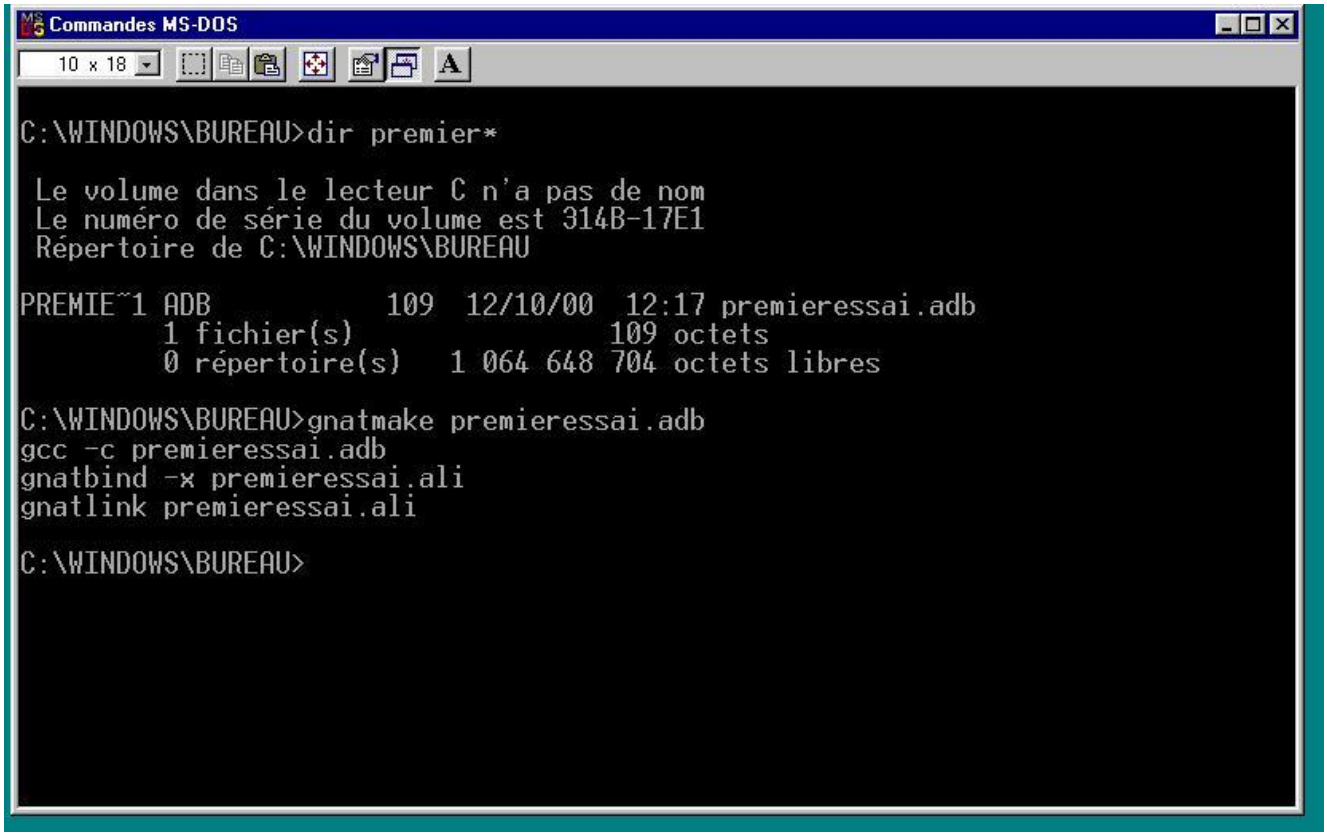
Une information est facile à exploiter: les numéros de ligne et de colonne où l'erreur a été détectée. Ce n'est pas toujours l'endroit précis de l'erreur, mais ça l'est parfois, et généralement l'erreur est située peu avant l'endroit où elle est détectée.

Le message d'erreur est parfois assez difficile à comprendre et à interpréter. Dans notre exemple, le message signifie: *unité de compilation attendue*, ce qui n'est pas forcément très explicite pour le néophyte. Il faut se familiariser avec les messages d'erreurs. C'est l'objet de l'exercice 2.3.

Nous devons corriger l'erreur du programme. Pour cela, il faut retourner dans edit, ouvrir notre fichier avec l'option `Ouvrir` du menu `Fichier`. Puis corriger l'erreur, sauver à nouveau le fichier et quitter edit.

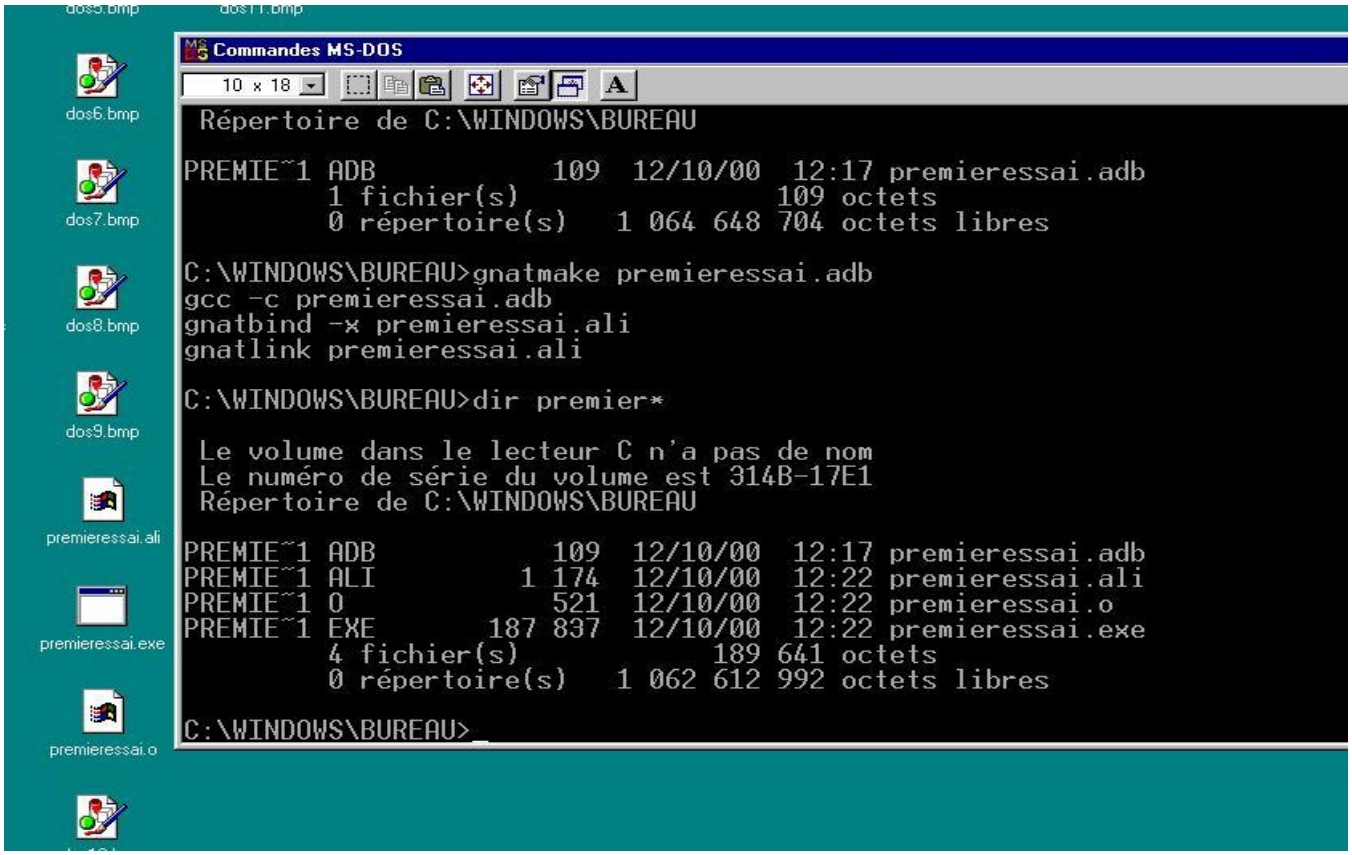


Puis on réessaye de compiler avec la commande `gnatmake`.

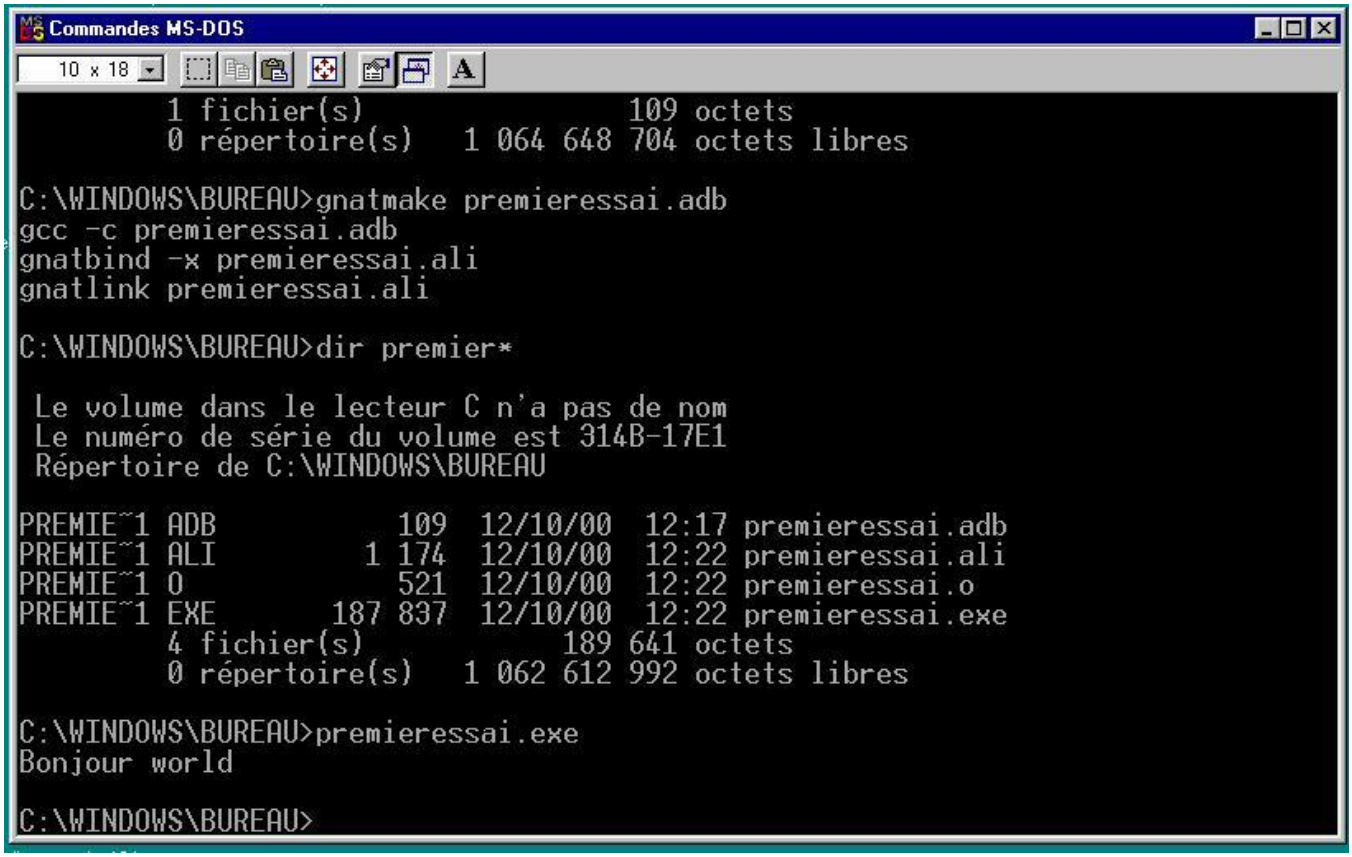


Les messages qui s'affichent sont normaux. Ici, il n'y a pas d'erreur. Notre programme est compilé.

Si on regarde les fichiers qui existent dans le répertoire en exécutant la commande `dir`, on s'aperçoit que `gnatmake` a créé trois fichiers: `premieressai.o`, `premieressai.ali` et `premieressai.exe`. Ce dernier est un fichier exécutable dos. C'est notre programme.



Pour exécuter le programme, il suffit de tapper `premieressai.exe`.



Voilà, objectif atteint, nous savons désormais écrire, compiler et exécuter un programme Ada.

Peut-être qu'en tapant le programme, vous avez fait des erreurs, des fautes de frappe autres que le `prossedure`.

Face à ce problème, deux attitudes possibles:

- Trouver, comprendre et corriger vos erreurs jusqu'à ce que ça marche enfin. Sur un tout petit programme, ça doit être possible.
- Reprendre le texte du programme dans cette page en copiant avec la souris et le coller dans l'éditeur.

Vous ne devez considérer l'exercice comme terminé que lorsque vous avez exécuté le programme.

Lorsque vous avez terminé, vous pouvez mettre les fichiers à la poubelle ou les transférer dans le répertoire de votre choix, si vous voulez le garder

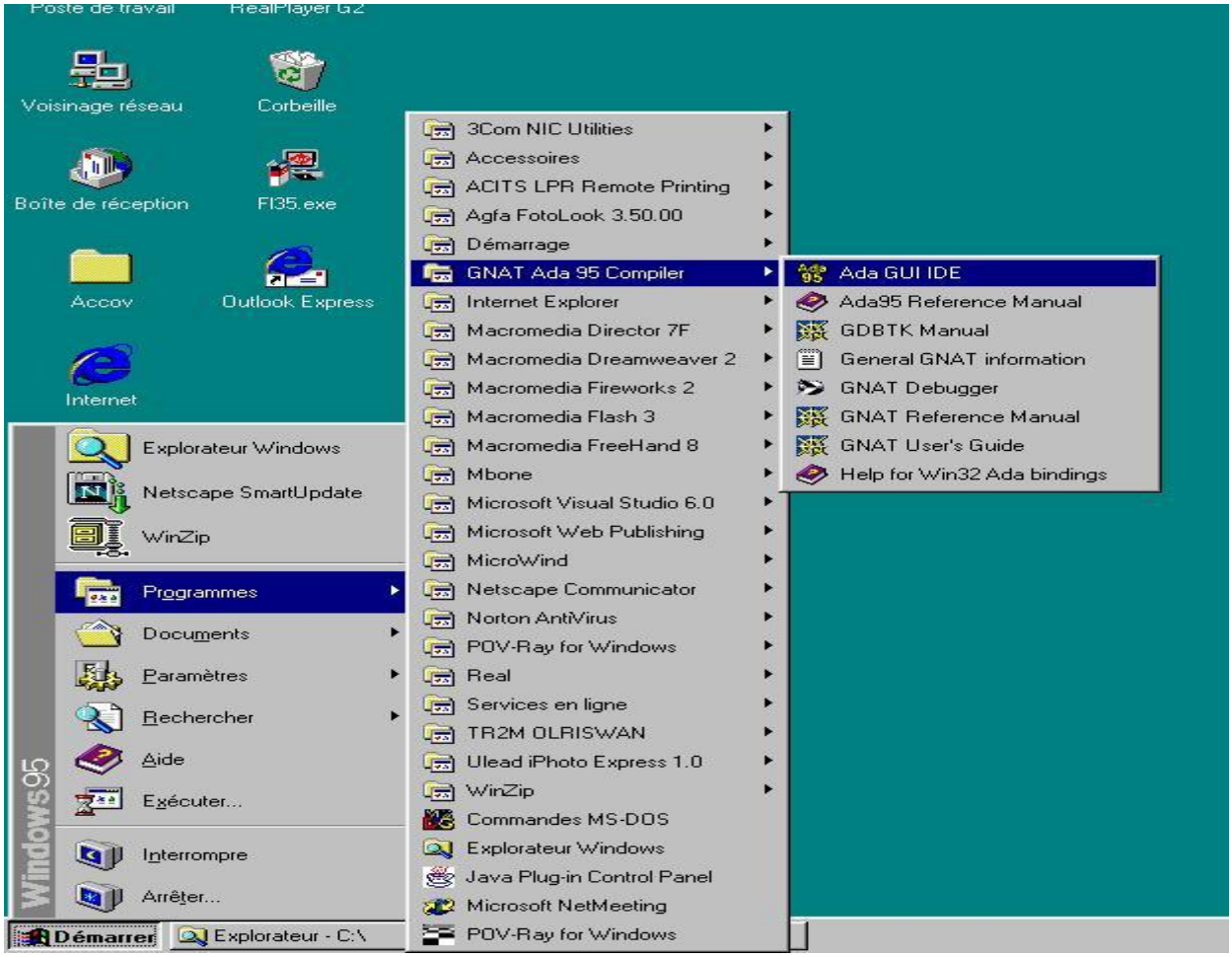
Exercice 2.2: Gnat avec ses fanfreluches

Avant de faire cet exercice, il est vivement conseillé de faire l'exercice 2.1.

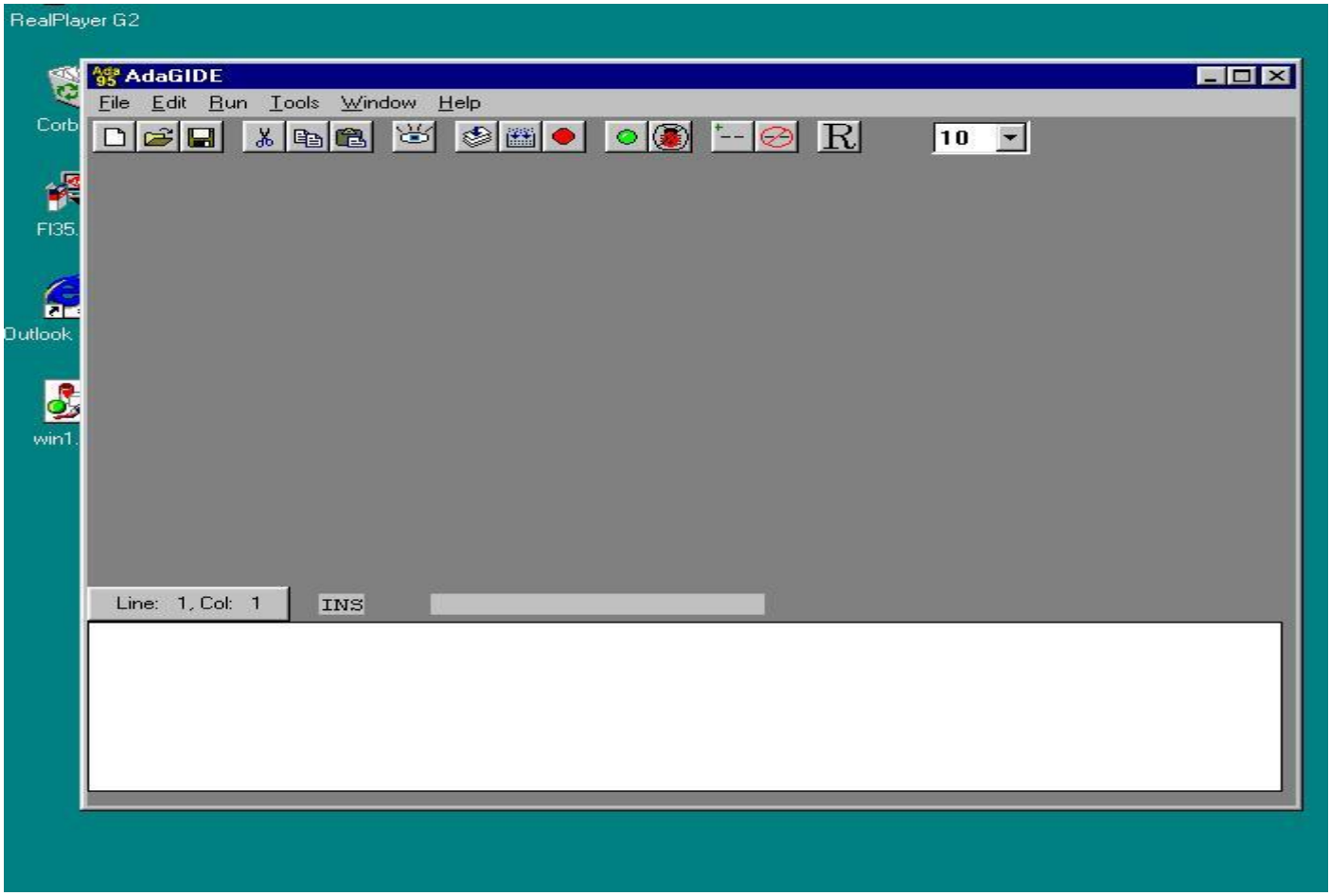
Nous allons refaire la même chose que dans l'exercice 2.1, à savoir, taper, compiler et exécuter un petit programme Ada, mais cette fois en utilisant le compilateur sous Windows, avec son environnement graphique GIDE. Cet environnement fourni un éditeur intégré et des boutons permettant de compiler et d'exécuter le programme. Vous allez refaire chez vous les manipulations décrites dans cette page.

Les différentes étapes restent les mêmes: taper le programme, compiler, interpréter. Mais elles se font toutes avec le même outil au lieu d'être fait par différents appels de commandes DOS.

Allez chercher l'option `ADA GUI IDE` du sous-menu `GNAT Ada 95 Compiler` du sous-menu `Programmes` du menu `Demarrer`.



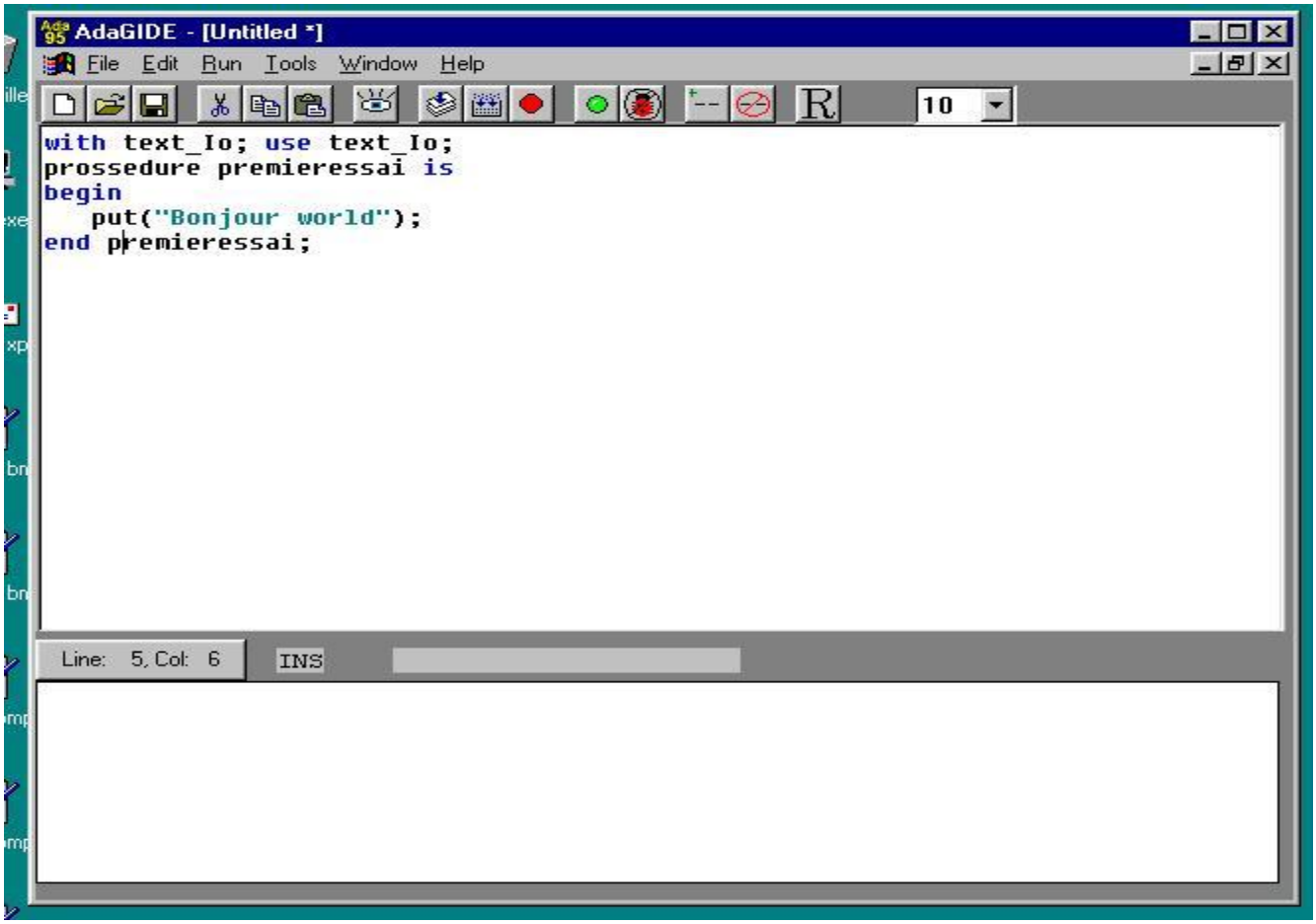
La fenêtre `AdaGide` apparaît avec l'apparence suivante:



Allez choisir l'option `New` du menu `File` pour créer un nouveau fichier (celui qui contiendra le programme). Photo optionnelle

On obtient alors une fenêtre avec deux zones blanches: une grande en haut, où l'on tape le programme, une plus petite en bas, oùle système affiche les messages (messages d'erreur et autres). Photo optionnelle

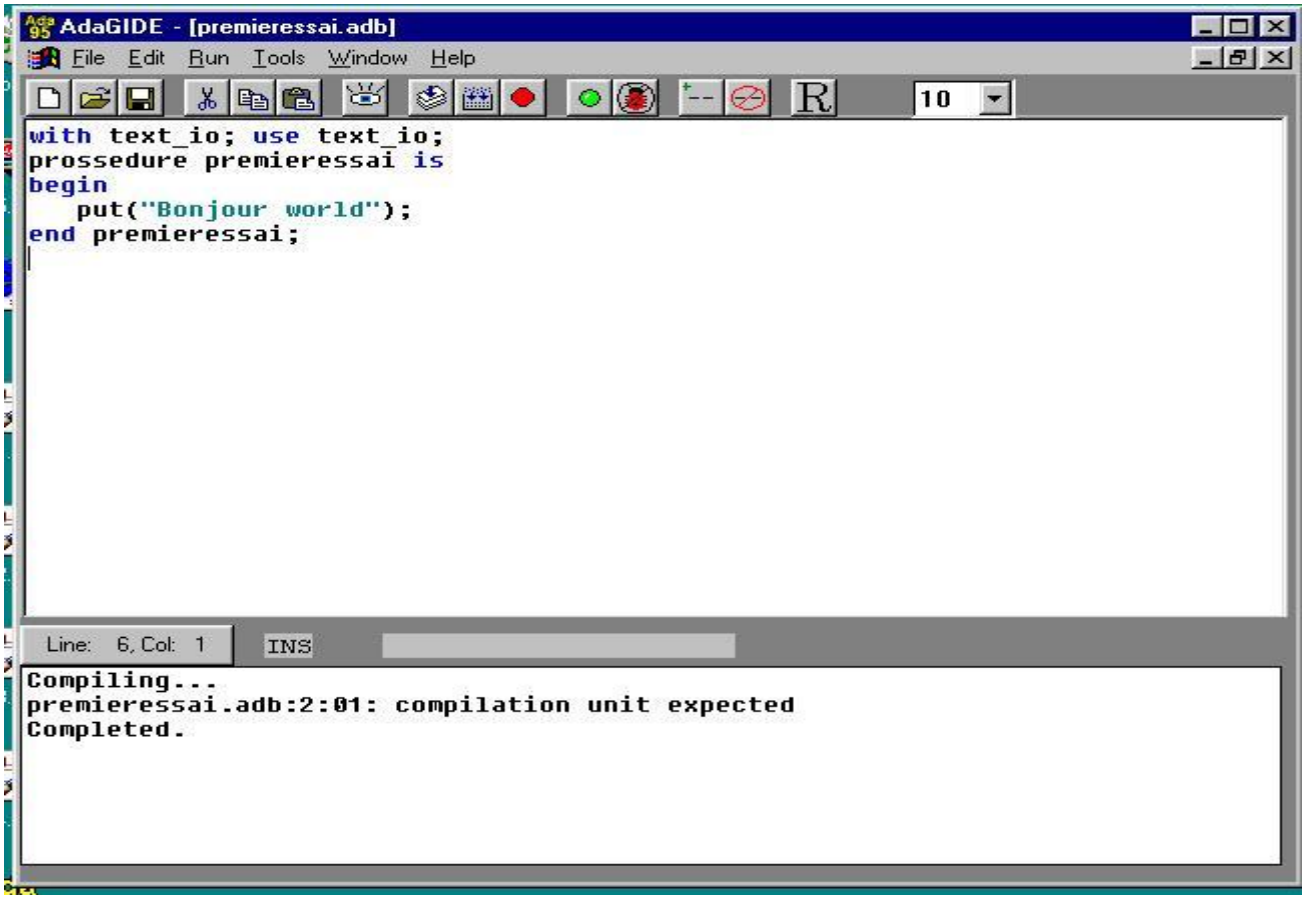
Tapons le programme.



Il faut ensuite sauver le programme sur disque en lui donnant un nom. La règle est toujours la même: le nom du fichier doit être le nom de la procédure en minuscule, suivie de l'extension .adb. Photo optionnelle 1 Photo optionnelle 2

Pour compiler, on peut soit choisir l'option `Compile file` du menu run (Photo optionnelle), soit utiliser la touche `f2`, soit cliquer sur le bouton-icône:

Si il y a des erreurs, elles s'affichent dans le cadre du bas de la fenêtre.



Le message est exactement le même que lorsqu'on utilise la commande `gnatmake` (cf. exercice 2.1).

Il faut corriger le programme, sauver à nouveau et recompiler.

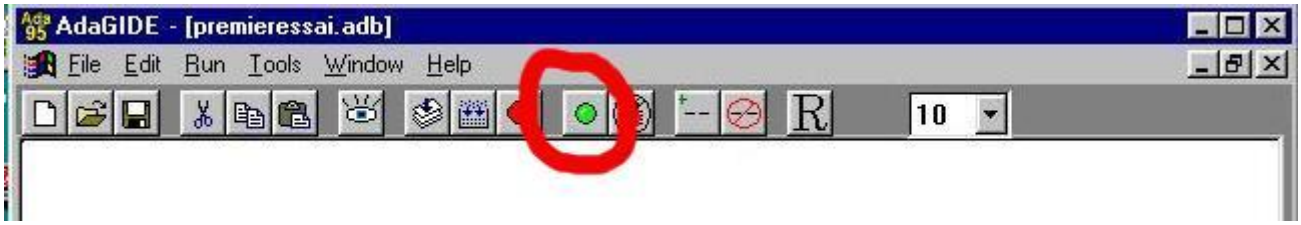
Photo optionnelle

Ensuite, il faut faire une autre étape qui consiste à construire le programme (build). Cette étape est faite automatiquement pas la commande `gnatmake` après la compilation. Avec l'environnement `Ada GIDE`, ce n'est pas automatique. Il faut soit choisir l'option `Build` du menu run (Photo optionnelle), soit utiliser la touche `f3`, soit cliquer sur le bouton-icône:

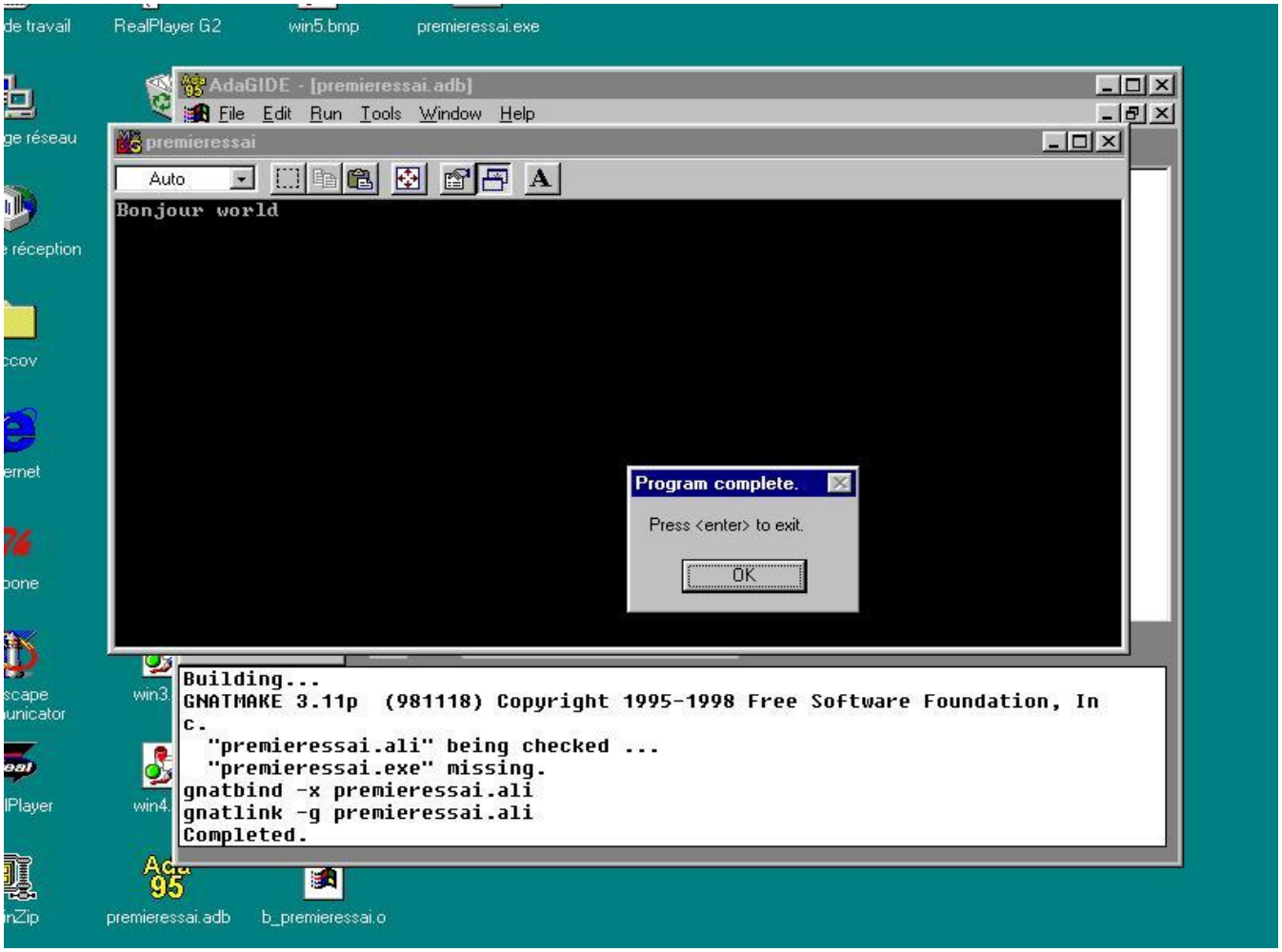
Des choses s'affichent dans le cadre du bas (Photo optionnelle).

A ce niveau, les trois fichiers `premieressai.o`, `premieressai.ali` et `premieressai.exe` ont été créés.

On peut exécuter le programme soit en choissant l'option `Execute` du menu run (Photo optionnelle), soit utiliser la touche `f3`, soit cliquer sur le bouton-icône:



Pour exécuter le programme, le système ouvre automatiquement une fenêtre ms-dos, exécute le programme `premieressai.exe` dans cette fenêtre et vous propose de fermer cette fenêtre avec une boite de dialogue.



Voilà. Nous avons terminé notre première prise en main de GNAT. Evidemment, il y a beaucoup de possibilités que nous n'avons pas utilisée. Pour débuter, nous nous sommes contentés du minimum. A vous de vous familiariser avec l'outil petit à petit. Toute une partie est assez simple à prendre en main (menus `File` et `Edit`) car cela ressemble à d'autres outils que vous connaissez.

On peut quitter AdaGIDE avec l'option `Exit` du menu `File`.

L'exercice est terminé quand le programme a été exécuté.

Exercice 2.3: messages d'erreur

Le but de cet exercice est de se familiariser avec des messages d'erreur que vous rencontrerez plus ou moins fréquemment par la suite.

Suivent plusieurs programmes avec chacun une erreur. Pour être sûr d'obtenir l'erreur souhaitée et pas d'autres que vous fairiez en recopiant, vous devez copier ces programmes dans l'éditeur à la souris, par copier/coller.

Pour chaque programme, compilez, regardez le message d'erreur, essayez de le comprendre. Corrigez le programme et compilez-le.

Programme 1

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada_Integer_Text_Io;

procedure Programme_1 is
   X: Integer;
begin
   Put("donnez-moi un nombre ");
   Get(X);
   Put("ce n'est pas assez, donnez-moi plutot ");
   Put(X * 2);
   Put(" !");
   New_Line;
end Programme_1;
```

Dans cet exemple, il n'y a qu'une erreur qui génère trois messages d'erreur. Il faut retenir l'idée qu'une erreur peut rejaillir à plusieurs endroits du programme.

Programme 2

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;

procedure Programme 2 is
   X: Integer;
begin
   Put("donnez-moi un nombre ");
   Get(X);
   Put("ce n'est pas assez, donnez-moi plutot ");
   Put(X * 2);
   Put(" !");
   New_Line;
end Programme 2;
```

Dans cet exemple, le message d'erreur passe complètement à côté du problème. C'est un cas de figure assez fréquent. En revanche, la position dénotée par le numéro de ligne et le numéro de colonne est significative.

Programme 3

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;

procedure Programme_3 is
   X: Integer;
begin
   Put("donnez-moi un nombre ");
   Get(Y);
   Put("ce n'est pas assez, donnez-moi plutot ");
```

```
   Put(X * 2);
   Put(" !");
   New_Line;
end Programme_3;
```

Un cas assez simple.

Programme 4

```
with Text-Io; use Text-Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;

procedure Programme-4 is
   X: Integer;
begin
   Put("donnez-moi un nombre ");
   Get(X);
   Put("ce n'est pas assez, donnez-moi plutot ");
   Put(X * 2);
   Put(" !");
   New_Line;
end Programme-4;
```

Programme 5

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;

procedure Programme_5 is
   X: Integer;
begin
   Put("donnez-moi un nombre ");
   X:=Get(X);
   Put("ce n'est pas assez, donnez-moi plutot ");
   Put(X * 2);
   Put(" !");
   New_Line;
end Programme_5;
```

Programme 6

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;

procedure Programme_6 is
   X: Integer;
begin
   Put('donnez-moi un nombre ');
   Get(X);
   Put("ce n'est pas assez, donnez-moi plutot ");
   Put("X" * 2);
   Put(" !");
   New_Line(" ");
end Programme_6;
```

Ici, à l'opposé du programme 1, il y a plusieurs erreurs, mais seule la première est détectée et fait l'objt d'un message. Il faut plusieurs passes pour corriger le programme

Commentaires sur l'exercice 2.3

Note: vous ne devez lire ces commentaires qu'après avoir fait l'exercice 2.3.

Voici quelques-uns des messages d'erreur rencontrés:

```
programme_x.adb:2:31: "Ada_Integer_Text_Io" is undefined
```

traduction: `"Ada_Integer_Text_Io"` n'est pas défini
sens: le programme utilise un élément non déclaré ou il y a une faute de frappe dans ce nom.

```
programme_x.adb:8:04: invalid parameter list in call (use -gnatf for details)
```

traduction: mauvaise liste de paramètre dans un appel
sens: les paramètres donnés à une fonction ou à une procédure ne conviennent pas (soit il n'y en a pas le bon nombre, soit ils ne sont pas de bon type).

```
programme_x.adb:10:04: possible missing instantiation of Text_IO.Integer_IO
```

traduction: il manque peut-être une instanciation de `Text_IO.Integer_IO`
sens: difficile à expliquer ici.

```
programme_x.adb:8:08: "Y" is undefined
```

traduction et sens: cf ci-dessus.

```
programme_x.adb:1:10: missing ";"
```

traduction: il manque ";"
sens: il manque ";"

```
programme_x.adb:7:08: strings are delimited by double quote character
```

traduction: les chaines de caractère sont délimitées par le caractère double quote
sens: le caractère double quote est celui-ci: `"`

```
programme_x.adb:4:11: warning: file name does not match unit name, should be
"programme_6.adb"
```

traduction: avertissement: le nom de fichier ne correspond pas au nom d'unité, il devrait être `"programme_6.adb"`
sens: le nom du fichier n'est pas le nom de la procédure écrit en minuscule et suivi de l'extension .adb.

```
programme_x.adb:10:08: expected type universal integer
programme_x.adb:10:08: found a string type
```

traduction: le type universal integer est attendu, le type string est trouvé
sens: il y a une valeur ou une expression de type string là où Gnat attend une valeur ou une expression de type integer.

Un avertissement (warning) est quelque chose qui semble anormal mais qui n'empêche pas la compilation, contrairement à une erreur.

```
programme_x.adb:4:01: compilation unit expected
```

traduction: unité de compilation attendue
sens: il y a une erreur quelque part avant le mot-clé `procedure`. Une unité de compilation est (en première approximation) une procédure.

Ce très rapide tour d'horizon ne couvre bien sûr pas tous les messages que vous pouvez rencontrer. Faire attention aux messages, comprendre ce qu'ils veulent dire est très important pour réaliser des programme

Corrigé de l'exercice 1.1

Question 1

Donnez les tables de vérité des propositions suivantes:

- (A et B) ou (non B)
- (non A) et (non B)
- (A et B) et (non A)
- non (A et (non B))

On utilise la méthode présentée au chapitre 1.

A	B	A et B	non B	(A et B) ou (non B)
vrai	vrai	vrai	faux	vrai
vrai	faux	faux	vrai	vrai
faux	vrai	faux	faux	faux
faux	faux	faux	vrai	vrai

A	B	non A	non B	(non A) et (non B)
vrai	vrai	faux	faux	faux
vrai	faux	faux	vrai	faux
faux	vrai	vrai	faux	faux
faux	faux	vrai	vrai	vrai

A	B	A et B	non B	(A et B) et (non B)
vrai	vrai	vrai	faux	faux
vrai	faux	faux	vrai	faux
faux	vrai	faux	faux	faux
faux	faux	faux	vrai	faux

A	B	non B	A et (non B)	non (A et (non B))
vrai	vrai	faux	faux	vrai

vrai	faux	vrai	vrai	faux
faux	vrai	faux	faux	vrai
faux	faux	vrai	faux	vrai

Question 2

Pour chacune de ces proposition, trouvez une autre proposition plus simple (c'est à dire avec moins d'opérateurs) équivalente. Rappel: deux propositions sont équivalente si elles ont la même valeur de vérité dans tous les cas possibles.

Il n'y a pas de méthode particulière à mettre en oeuvre: il faut faire appel à son intuition.

proposition	proposition équivalente
(A et B) ou (non B)	B => A
(non A) et (non B)	non (A ou B)
(A et B) et (non A)	faux
non (A et (non B))	A => B

Question 2

Pour chacune de ces proposition, trouvez une autre proposition plus simple (c'est à dire avec moins d'opérateurs) équivalente. Rappel: deux propositions sont équivalente si elles ont la même valeur de vérité dans tous les cas possibles.

Corrigé de l'exercice 1.2

Soient les propositions:

- (P) il pleut
- (S) la sortie est annulée
- (M) etre mouillé
- (R) rester à la maison

Question 1

Traduire les phrases suivantes en une proposition logique, en utilisant P, S, M et R.

- Si il pleut et si je reste à la maison, je ne serai pas mouillée.
  `P et R => non M`
- Si il pleut mais si je reste à la maison, je ne serai pas mouillée.
  `P et R => non M`
- Si il pleut et si la sortie n'est pas annulée ou si je ne reste pas à la maison, je serai mouillée.
  `((P et non S) ou non R) => M`
- que la sortie soit annulée ou non, s'il pleut, je reste à la maison.
  `(S ou non S) et P => R`
  ou ce qui est logiquement équivalent:
  `P => R`

Remarque: le choix de traduire la construction *si* par une implication peut être discutée. La langue naturelle ne respecte pas forcément les règles de la logique. Les autres choix possibles sont l'équivalence et la conjonction *et*.

Question 2

Dans quels cas la proposition *Si il pleut mais si je reste à la maison, je ne serai pas mouillée* est vraie?

Pour répondre à cette question, on fera une table de vérité où on calculera la valeur de la proposition en fonction des valeurs de P, M et R.

Calculons la table de vérité de la proposition:
`P et R => non M`

P	R	M	P et R	non M	(P et R) => non M
vrai	vrai	vrai	vrai	faux	faux
vrai	vrai	faux	vrai	vrai	vrai
vrai	faux	vrai	faux	faux	vrai
vrai	faux	faux	faux	vrai	vrai
faux	vrai	vrai	faux	faux	vrai
faux	vrai	faux	faux	vrai	vrai
faux	faux	vrai	faux	faux	vrai
faux	faux	faux	faux	vrai	vrai

On voit que la proposition est presque toujours vraie. Le seul cas où elle est fausse, c'est si il pleut, je reste à la maison et je suis mouillé. Est-ce que ce cas peut se produire? Peut-être si je prends une douche ou si le toit n'est pas étanche ...

Question 3

Si la proposition *Si il pleut et si je reste à la maison, je ne serai pas mouillée* est vraie et si il ne pleut pas, est-ce que je reste à la maison?

Il ne s'agit pas de faire un raisonnement de la vie courante, mais un raisonnement logique. Pour cela, on fera une table de vérité de la proposition correspondant à *Si il pleut et si je reste à la maison, je ne serai pas mouillée* (ce que vous avez répondu à la question 1.1) et on regardera les lignes telles que cette proposition est vraie et P est fausse. On regardera alors la valeur de R dans ces lignes.

Et on en tirera une conclusion.

Il faut construire la table de vérité de la proposition restreinte aux cas où P a la valeur faux et où `P et R => non M` a la valeur vrai.

P	R	M	P et R	non M	(P et R) => non M
faux	vrai	vrai	faux	faux	vrai
faux	vrai	faux	faux	vrai	vrai
faux	faux	vrai	faux	faux	vrai
faux	faux	faux	faux	vrai	vrai

On voit dans la colonne de M qu'il y a des cas où M est vrai et des cas où M est faux. Si il ne pleut pas, on ne peut donc pas savoir si je suis mouillé ou pas. La proposition *Si il pleut et si je reste à la maison, je ne serai pas mouillée* ne m'apprend rien d'utile à ce sujet.

C'est là un effet de la forme particulière de la table de vérité du connecteur *implique* qui est souvent difficile à manipuler intuitivement.

Evidemment, le raisonnement logique aurait conduit à un autre résultat si j'avais choisi de traduire le *si* de la proposition par le connecteur d'équivalence. Il y a donc plusieurs façons de répondre à l'exercice, aussi valables les unes que les autres à condition de savoir justifier ses choix.

Corrigé de l'exercice 1.3

Soient a, b et c trois entiers positifs.

Question 1

Montrez que la proposition `(a<b) => (a = b)` a pour négation `a<b`.

Pour cela, vous pouvez donner un nom aux deux propositions élémentaires `a<b` et `a=b` (élémentaire d'un point de vue logique, car < et = ne sont pas des opérateurs logiques. Ce sont des opérateurs arithmétiques).

Appelons P la proposition `(a<b)` et Q la proposition `(a = b)`.

Ensuite, faites la table de vérité des deux propositions.

P	Q	P => Q	non (P => Q)
vrai	vrai	vrai	faux
vrai	faux	faux	vrai
faux	vrai	vrai	faux
faux	faux	vrai	faux

Utilisez vos connaissances sur l'arithmétique pour exclure certains cas qui ne sont pas compatibles avec l'arithmétique. Cela revient à éliminer des lignes de la table.

P	Q	P => Q	non (P => Q)	possible arithmétiquement
vrai	vrai	vrai	faux	non
vrai	faux	faux	vrai	oui
faux	vrai	vrai	faux	oui
faux	faux	vrai	faux	oui

Tirons la conclusion: pour toutes les lignes arithmétiquement possibles, la première et la dernière colonne de la table sont égales. On en déduit que les deux propositions sont équivalentes. La première colonne correspond à la valeur de vérité de `a<b` et la dernière est la négation de `(a<b) => (a = b)`.

Question 2

Montrez que la proposition `(a<b) => a = a` a pour négation fleche  `a`.

Appelons R et S les propositions `(a<b)` et `a = a`.

R	S	R => S	non (R => S)	non S	possible arithmétiquement
vrai	vrai	vrai	faux	faux	oui
vrai	faux	faux	vrai	vrai	non
faux	vrai	vrai	faux	faux	oui
faux	faux	vrai	faux	vrai	non

Les quatrième et cinquième colonnes sont égales sur les lignes arithmétiquement possibles. Non S est (non a = b) ce qui peut s'écrire $a \neq a$.

chapitre 3:instructions simples

Quelques notions syntaxiques

La syntaxe d'un langage est l'ensemble des règles qui définissent les propriétés de forme des textes corrects. Ainsi la syntaxe d'Ada définit comment doivent être écrit les programmes pour être du bon Ada.

Nous avons déjà vu un peu de cette syntaxe la semaine dernière. Nous avons vu un exemple de programme. Nous avons également donné la définition des identificateurs, qui servent notamment comme noms de variables. La règle est qu'un identificateur commence par une lettre, qu'ensuite il peut y avoir des lettres, des chiffres et le caractère souligné _ en nombre quelconque, mais pas deux soulignés de suite.

Nous allons présenter quelque notions importantes de la syntaxe d'Ada avec le double but d'éviter de faire des erreurs en programmant et de comprendre les erreurs qui surviennent après qu'on les a faites.

Définition des instructions

Une instruction est un morceau de programme qui change l'état de la machine:

- soit en changeant le contenu de la mémoire
- soit en changeant l'affichage de l'écran

Les instructions que vous connaissez pour le moment sont:

- l'affectation, qui consiste à changer le contenu d'une variable. Cela change la mémoire.
  Exemple: `X := 25;`
- le Put: il affiche quelque chose à l'écran.
  Exemple: `Put(25);`
- le Get: il lit quelque chose au clavier et le met dans une variable. Cela change donc le contenu de la mémoire, comme une affectation.
  Exemple: `Get(X);`

Put et Get sont des procédures prédéfinies d'Ada. De façon générale, toute utilisation de procédure (ce qu'on nomme un *appel de procédure*), composé d'un nom de procédure et d'une liste de paramètres entre parenthèse est une instruction.

Les instructions ne sont que dans la partie instruction du programme. Elles ne peuvent pas apparaître dans le préambule ni dans la partie déclaration.

Essayez de compiler les programmes suivants qui ne respectent pas cette règle:

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
   Put(30);
procedure Teste_Instr is
   X25: Integer;
begin
   X25 := abs(-15);
   Put(X25);
end Teste_Instr;

with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Teste_Instr is
```

```
   X25: Integer;
   Put(30);
begin
   X25 := abs(-15);
   Put(X25);
end Teste_Instr;
```

Définition des expressions

Une expression est un morceau de programme permettant de calculer une valeur. Contrairement à une instruction, il ne change pas la mémoire ni l'écran.

Une expression apparait à l'intérieur d'une instruction, elle en est un des composants.

Une expression peut être:

- un littéral, c'est à dire un élément permettant de noter une valeur donnée.
  Exemples:
  - 1, 456 et -125 sont des littéraux de type `integer`
  - `true`, `false`, sont des littéraux de type `boolean`
  - `'a'`, `':'`, sont des littéraux de type `character`
  - 1.2 et 125.69 sont des littéraux de type `float`
  - `"Bonjour"`, `"Bonne chance"` sont des littéraux de type `string`
- un nom de variable est une expression. Ce nom permet de trouver le contenu de la variable. La valeur de cette expression est le contenu de la variable. Il faut que la variable ait été déclarée dans la partie déclaration.
  Exemples: `X`, `Nombre`
- l'ensemble composé de deux expressions séparées par un opérateur binaire est une expression.
  Exemples:
  - avec deux littéraux: `1 + 125`
  - avec deux variables: `X * Y`
  - avec une variable et un autre opérateur: `X + Y * 12`
  - ...
- une expression placée entre parenthèses est une expression.
  Exemples: `(125)`, `(Nombre)`, `(12 + 5)`, `(12 * (Y + 5))`
- un appel de fonction est une expression.
  Exemple: `abs(-125)` (la fonction `abs` est une fonction prédéfinie d'Ada permettant de calculer la valeur absolue d'un entier.

Les expressions ne doivent apparaître qu'à l'intérieur d'une instruction. En effet, on calcule une valeur pour la stocker en mémoire ou l'afficher à l'écran. Sinon, cela ne sert à rien. Essayez de compiler les programmes suivants qui ne respectent pas cette règle (instruction se dit *statement* en anglais):

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Teste_Expr is
   X25: Integer;
   2 + 56;
begin
   X25 := abs(-15);
   Put(X25);
end Teste_Expr;

with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Teste_Expr is
   X25: Integer;
begin
```

```
   X25 := abs(-15);
   2 + 56;
   Put(X25);
end Teste_Expr;
```

N'importe quel type d'expression peut apparaître en n'importe quel endroit où une expression est attendue:

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Teste_Expr is
   X25: Integer;
   Y: Integer;
begin
   X25 := 10;
   Put(10);
   Put(X25);
   Put(X25 * 2);
   Put((25));
   Put(X25 + (12 * X25));
   Y := 10;
   Y := X25;
   Y := X25 * 2;
   Y := (25);
   Y := X25 + (12 * X25);
end Teste_Expr;
```

De même qu'une expression ne doit pas être là où il faut une instruction, une instruction ne doit pas être là où il faut une expression:

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Teste_Instr is
   X25: Integer;
begin
   X25:=(Put(25));
end Teste_Instr;

with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Teste_Instr is
   X25: Integer;
begin
   X25 := 6 * (Put(25));
end Teste_Instr;

with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Teste_Instr is
   X25: Integer;
begin
   Put(6 * (Put(25)));
end Teste_Instr;
```

Un petit truc pour distinguer les instructions des expressions: les instructions se terminent par un point-virgule. Pas les expressions. Mais parfois, comme pour l'affectation, il n'y a rien entre la fin de l'expression et le point-virgule qui termine l'instruction qui la contient.

Typage

Nous avons vu la semaine dernière qu'un type en Ada est un ensemble de valeurs. Nous avons vu plusieurs de ces types: les entiers, les flottants, les booléens, les caractères et les chaînes de caractères.

Ces ensembles de valeur sont utilisés dans les déclaration. Quand une variable est déclarée d'un certain type, cela signifie qu'elle contiendra à tout moment une valeur appartenant à cet ensemble.

Par exemple, quand on déclare:
`X: integer;`
on veut dire qu'à tout moment dans le programme, il y aura un entier dans X.

Cette déclaration permet de réserver pour X une place en mémoire suffisante pour stocker un entier quel qu'il soit. Dans beaucoup de cas, ce sera 4 octets, c'est à dire 32 bits. Pour nous programmeur, il n'est généralement pas très important de savoir combien de bits il y a. Mais quel que soit le nombre de bits de mémoire réservés pour X, il faut bien comprendre que chaque bit est soit à 0 soit à 1 et qu'il n'est pas possible que X ne contiennent rien. Il y a toujours un entier dans X. Mais entre la déclaration et le moment où on met une valeur dans X par une affectation ou un Get, il y a un certain entier qu'on ne connait pas.

Le compilateur vérifie que les déclarations et les instructions sont cohérentes du point de vue du type. Par exemple, si on déclare que X est un integer et qu'on essaie de mettre un boolean dedans, cela provoque une erreur à la compilation.

Vous pouvez tester le programme suivant:

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Typage is
   X: Integer;
begin
   X := True;
end Typage;
```

Pour effectuer cette vérification, le compilateur utilise des règles de typage des instructions.

Règles de typage

- Affectation: l'affectation a deux parties, une partie gauche avec un nom de variable et une partie droite avec une expression. Les deux parties sont séparées par le signe `:=`.

  L'affectation est bien typée si la variable est déclarée avec un type qui est le même que le type de l'expression:

  ```
  ...
    X: integer;
  ...
  begin
    ...
    X := 12 * 3;
    ...
  ```

  est correct parce que X est de type integer et que `12 * 3` est de type integer.

- une instruction Put est correcte si l'expression donnée entre les parenthèses a un type `integer` ou `string` ou `character`.
- une instruction Get est correcte si un nom de variable est donné entre les parenthèses.

Les règles de typage des expressions sont assez simples et naturelles: une expression est correcte si les opérandes données aux opérateurs sont du bon type. Le type de l'expression est le type de la valeur calculée par l'expression.

Nous n'allons pas donner ici les règles formelles, mais des exemples.

L'opérateur + veut deux opérandes numériques. De plus, en Ada, il veut deux opérandes de même type. On peut faire + de deux integer ou de deux float, mais pas d'un integer et d'un float.

`45 + 35` et `45.3 + 12.6` sont corrects, mais pas `45.3 + 6` ni `true + 5`. Le type de l'expression `45 + 35` est integer et celui de `45.3 + 12.6` est float.

Certains opérateurs ont des opérandes d'un type et un résultat d'un autre type. Par exemple < permet de comparer deux entiers et rend un résultat boolean (c'est à dire une valeur de vérité, vrai ou faux, true or false).

`12 > 3` est une expression correcte. Son type est boolean.

Exécution détaillée d'une boucle

Nous avons vu au chapitre précédent qu'une exécution d'un programme pouvait se décrire au moyen d'un tableau résumant l'état de la mémoire et des entrées-sorties écran-clavier à un moment donné. Nous allons faire un tel tableau pour retracer pas à pas l'exécution de boucles, en commençant par une boucle `For`. Voici le programme avec les lignes numérotées.

```
1   with Text_Io; use Text_Io;
2   with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3   procedure Voila_For_5 is
4   begin
5      Put("debut");
6      New_Line;
7      for Nb in 11..13 loop
8         Put(Nb);
9         New_Line;
10     end loop;
11     Put("fin");
12     New_Line;
13  end Voila_For_5;
```

numéro	état mémoire	écran	commentaire
5	-	debut	
6	-	(saut de ligne)	
7	Nb vaut 11		Nb <= 13 vaut true
8	Nb vaut 11	11	
9	Nb vaut 11	(saut de ligne)	
7	Nb vaut 12		incrémentation de Nb, Nb <= 13 vaut true
8	Nb vaut 12	12	
9	Nb vaut 12	(saut de ligne)	

7	Nb vaut 13		incrémentation de Nb, Nb <= 13 vaut true
8	Nb vaut 13	13	
9	Nb vaut 13	(saut de ligne)	
7	Nb vaut 14		incrémentation de Nb, Nb <= 13 vaut false
11	-	fin	Nb n'existe plus en mémoire
12	-	(saut de ligne)	

A noter que nous ne faisons pas apparaître dans le tableau des lignes du programme telles que 4, 10, 12 qui sont juste des marque de début ou de fin mais qui ne correspondent à aucun calcul effectif, à aucun changement d'état ni aucune entrée-sortie.

Voyons à présent une boucle loop.

```
1    with Text_Io; use Text_Io;
2    with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3    procedure Voila_Loop_3 is
4       Nombre: Integer;
5       Mult: Integer;
6    begin
7       Nombre := 5;
8       Mult := 1;
9       loop
10         Put(Nombre * Mult);
11         New_Line;
12         Mult := Mult+1;
13         exit when Mult = 5;
14      end loop;
15      Put("sont des multiples de 5");
16   end Voila_Loop_3;
```

Essayez de voir à la lecture ce que fait ce programme. A noter ligne 10, un `Put` qui affiche le résultat du calcul: le paramètre du `Put` est une expression qui est calculée et c'est le résultat qui est affiché.

numéro	état mémoire	écran	commentaire
	Nombre vaut ?, Mult vaut ?		ceci est l'état initial
7	Nombre vaut 5, Mult vaut ?		
8	Nombre vaut 5, Mult vaut 1		
10	Nombre vaut 5, Mult vaut 1	5	
11	Nombre vaut 5, Mult vaut 1	(saut de ligne)	
12	Nombre vaut 5, Mult vaut 2		
13	Nombre vaut 5, Mult vaut 2		`Mult=5` vaut false, on continue
10	Nombre vaut 5, Mult vaut 2	10	
11	Nombre vaut 5, Mult vaut 2	(saut de ligne)	
12	Nombre vaut 5, Mult vaut 3		
13	Nombre vaut 5, Mult vaut 3		`Mult=5` vaut false, on continue
10	Nombre vaut 5, Mult vaut 3	15	
11	Nombre vaut 5, Mult vaut 3	(saut de ligne)	
12	Nombre vaut 5, Mult vaut 4		

13	Nombre vaut 5, Mult vaut 4		`Mult=5` vaut false, on continue
10	Nombre vaut 5, Mult vaut 4	20	
11	Nombre vaut 5, Mult vaut 4	(saut de ligne)	
12	Nombre vaut 5, Mult vaut 5		
13	Nombre vaut 5, Mult vaut 5		`Mult=5` vaut true, on arrête
15	Nombre vaut 5, Mult vaut 5	sont des multiples de 5	
	Nombre vaut 5, Mult vaut 5		état final

Nous voyons qu'une boucle se traduit par une répétition de l'exécution de certaines instructions (10,11,12,13 dans le dernier tableau). Mais comme cette exécution se fait avec un état différent, le résultat de l'exécution peut être différent à chaque tour de boucle. Ici le nombre affiché par la ligne 10 est à chaque fois différent.

Exercice 3.1: syntaxe

Question 1

Dans la liste suivante, qu'est ce qui est un identificateur (c'est à dire qu'est-ce qui peut servir de nom):

- `X25`
- `2X5`
- `Xy_25`
- `Integer_Text_IO`
- `Text_`
- `joli nom`
- `Text-IO`
- `Text___IO`
- `ab_23_cd`
- `"ab_cd"`

Répondez d'abord à la question en réfléchissant, puis pour chaque cas, vérifiez la réponse en compilant le petit programme de test:

```
with TEXT_IO; use TEXT_IO;
with Ada.Integer_TEXT_IO; use Ada.Integer_TEXT_IO;
procedure Teste_Ident is
   X25: Integer;
begin
   X25 := abs(-15);
   Put(X25);
end Teste_Ident;
```

(vous remplacerez `X25` par la suite de caractères que vous voulez tester).

Question 2

Dans un environnement où on a déclaré:

```
X: integer;
B: boolean;
C: character;
```

lesquelles des instructions suivantes sont correctes, et pour toutes les expressions correctes, donner leur type:

```
X := 25;
X := 25 + (12 * X);
B := X;
B := X < 25;
C := 'd';
B := 'c' = C;
B := 'c' := C;
X := B > C;
B := B and (c < 'f');
Put(X);
Put(C);
Put(B);
Put("X");
Put( 25 + (12 * X));
Get(X);
Get(C);
Get(B);
```

Pour certaines expressions, vous pouvez avoir des doutes, vu que toute l'information nécessaire ne vous a pas été donnée.

Essayez de deviner. Pour trancher vos doutes ou conforter vos certitudes, vous pouvez faire un programme de test et le compiler.

Question 3

Dans le programme suivant, comptez le nombre d'expressions et le nombre d'instructions.

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Carre is
   Nombre: Integer;
   Resultat: Integer;
begin
   Put("Entrez un nombre entier ");
   Get(Nombre);
   New_Line;
   Resultat := Nombre * Nombre;
   Put("le carre de ");
   Put(Nombre);
   Put(" est ");
   Put(Resultat);
   New_Line;
end Carre;
```

Exercice 3.2: division entière

Dans le cours a été évoqué l'existence d'opérateurs qui peuvent s'appliquer à des objets de plusieurs types, mais à la condition que les deux opérandes soient du même type.

C'est le cas des opérateurs arithmétiques comme + qui peuvent s'appliquer à des integer ou à des float, mais pas à un opérande integer et l'autre float. C'est le cas aussi des opérateurs de comparaison =, <, >, qui permettent de comparer des valeurs de beaucoup de types (tous ceux que l'on a vus pour l'instant) mais seulement deux valeurs de même type.

Il n'est pas possible de comparer un integer et un float (par exemple `1.2 > 5`). On apprendra plus tard qu'on peut transformer une valeur float en une valeur integer et réciproquement. On pourra alors faire la comparaison après transformation.

Dans la plupart des cas, ce que fait un opérateur est assez simple à comprendre. Mais il y a une exception: l'opérateur de division.

Comme les autres opérateurs arithmétiques, la division peut s'appliquer à deux integer ou à deux float. Le cas de la division de deux float ne pose pas de souci: c'est la division habituelle.

Dans le cas de la division de deux integer, le résultat est un integer et l'opération effectuée est la division entière, une opération mathématique assez peu fréquente dans la vie quotidienne.

Le résultat de la division `X / Y` est le nombre `d` tel que `X = (Y * d) + r` avec `r < d`. `d` est appelé diviseur de X et Y, et `r` est appelé reste de la division.

Exemples:
`9 = (2 * 4) + 1` et `1<2`, donc `9/2=4`
`8 = (2 * 4) + 0` et `0<2`, donc `8/2=4`
`125 = (36 * 3) + 17` et `17<36`, donc `125/36=3`

Il existe egalement un opérateur permettant d'obtenir le reste de la division entière: il s'agit de l'opérateur `mod`.

Exemples:
`9 = (2 * 4) + 1` et `1<2`, donc `9 mod 2 = 1`
`8 = (2 * 4) + 0` et `0<2`, donc `8 mod 2 = 0`
`125 = (36 * 3) + 17` et `17<25`, donc `125 mod 36 = 17`

Vous pouvez constater le fonctionnement de l'opérateur mod graâce au petit programme de test:

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Teste_mod is
begin
   Put(9/2);
   New_Line;
   Put(125/36);
   New_Line;
   Put(9 mod 2);
   New_Line;
   Put(125 mod 36);
   New_Line;
end Teste_mod;
```

Question

Ecrire un programme qui lit deux nombres tapés au clavier et affiche le résultat de la division et le reste de la division avec un message du type:
dans 125 il y a 3 fois 36 et il reste 17

Ce n'est pas très difficile, mais pour ceux qui auraient du mal, vous pouvez trouver ici un indice, à savoir une structure de programme à compléter.

Essayez d'éviter de regarder l'indice si vous n'en avez pas besoin.

Exercice 3.3

Le programme suivant a plusieurs erreurs.

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Multiplier Deux Entiers is
   X: Integer;
   Y: boolean;
   Resultat: Float;
begin
   Get("X");
   Put("Entrez le premier nombre ");
   New_Line;
   Put('entrez le deuxieme nombre ');
   Get(Y * 1);
   Resultat := XxY;
   New_Line;
   Put(Le produit des deux est );
   Put("Resultat");
end Multiplier Deux Entiers;
```

Question 1

Donnez le programme corrigé, c'est à dire un programme qui ne produit pas d'erreur à la compilation et qui fait ce qu'on attend de lui à l'exécution (à savoir, afficher le produit de deux nombres).

Question 2

Pour chaque erreur détectée, expliquez ce qui ne va pas. Dites si c'est une erreur de type ou une erreur de syntaxe.

Exercice 3.4: conditionnelle simple

Ecrire un programme qui teste si un nombre est pair ou non. La méthode à utiliser consiste à regarder le reste de la division entière par 2. Si le reste est nul, le nombre est divisible par 2, il est donc pair. Vous pouvez consulter l'exercice 3.3 sur la division entière pour voir comment calculer un reste.

Exercice 3.5: boucle simple

1. Ecrire un programme qui calcule puis affiche la somme de 10 nombres saisis au clavier.
2. Modifier votre programme pour qu'il calcule et affiche également la moyenne entière.
3. Modifiez encore une fois votre programme pour qu'il fasse le calcul sur un nombre quelconque de nombres saisis au clavier. Après chaque nombre entré, demander à l'utilisateur si il y a encore d'autres nombres à ajouter.

chapitre 4:fonctions

Fonction: notion mathématique

Voici la définition mathématique classique du mot fonction:
Définition Soient A et B deux ensembles. Une fonction f définie sur A (ou de domaine A, ou d'espace de départ A, ou de domaine de définition A) à valeurs dans B (ou de codomaine B, ou d'espace d'arrivée B, ou d'espace image B) est une correspondance qui, à tout élément x de A, fait correspondre un élément et un seul, noté f(x), de B. Cet élément f(x) est appelé résultat de l'application de f à l'élément x (parfois image de x par f).

Remarque: Il ne faut pas confondre la fonction f, qui est un élément de l'ensemble des fonctions de A dans B (en général noté A -> B), et le résultat de l'application de f à un argument x, qui est un élément de B. Dans certains cours de mathématiques, lorsque l'on ne s'intéresse pas aux fonctions en tant que telles mais seulement aux résultats de leurs applications, on parle parfois de la fonction f(x).

La notion de fonction pose plusieurs questions. La première est celle de la *calculabilité*, c'est à dire la possibilité de calculer la valeur f(x) pour une valeur x de A donnée. On peut définir certaines fonctions sans pour autant avoir de moyen de réaliser le calcul correspondant.

Par exemple, la fonction qui à un numéro de département associe le nombre de personnes présentes actuellement dans ce département. Cette fonction a un sens parfaitement compréhensible, et ce nombre de personnes existe. Simplement, il n'y a aucun moyen réaliste pour calculer ce nombre. Même l'INSEE, l'Institut National des Statistiques ne peut faire qu'un calcul approximatif.

L'informaticien s'intéresse presque exclusivement à des fonctions calculables, et parmi celles-ci, plus spécialement à celles qu'un ordinateur peut calculer dans des conditions acceptables, c'est à dire en un temps limité et avec des moyens en rapport avec les enjeux du calcul. Par exemple, si l'on fait un programme de prévision météorologique pour les cinq jours qui viennent, si on a une fonction qui calcule très précisément ces prévisions en fonction des relevés de stations météo, mais qu'il faut deux mois pour calculer cette fonction, elle n'a aucun intérêt pratique.

Une autre questions qui se pose est celle du langage que l'on emploie pour définir une fonction. Nous aborderons cette question d'abord du point de vue des mathématiques qui fournissent une base théorique pour les constructions offertes par les langages informatiques pour définir des fonctions.

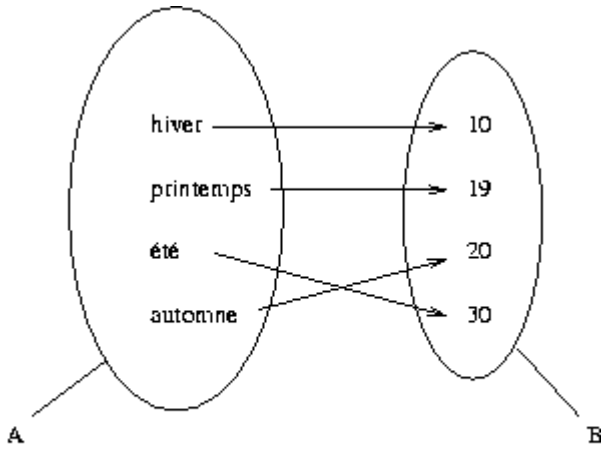
En mathématiques, il existe plusieurs méthodes pour définir une fonction. Voici celles qui sont le plus couramment employées.

Construire le graphe

Le graphe d'une fonction f de A dans B est l'ensemble des couples `(x,y)` où `x` est un élément de A, `y` est un élément de B et `y=f(x)`. Si le domaine A est fini, on peut indiquer explicitement quel est l'élément de B qui correspond à un élément donné de A. Dans ce cas, on peut définir la fonction par son graphe de manière effective.

Soit l'ensemble A = {hiver, printemps, été, automne}. On peut définir une fonction `max_temp` par le graphe suivant: `{(hiver,10),(printemps,19),(été,30),(automne,20)}`.

Un tel graphe peut aussi se représenter graphiquement, de la façon suivante:



graphe de f

Calcul du résultat de l'application

Calculer le résultat de l'application d'une fonction définie par un graphe à un argument donné `v` consiste simplement à chercher le couple `(v,y)` dans le graphe et en extraire la valeur `y` qui est la valeur de `f(v)`. Par définition de la notion de fonction, il existe un seul couple commençant par `v` dans le graphe de la fonction `f`.

Donner une expression

La fonction peut être définie par une expression dont le calcul donne la valeur de la fonction en tout point de son domaine de définition. Pour cela, il faut choisir un nom, disons x, pour désigner un élément quelconque du domaine. Ce nom est appelé variable en mathématiques.

Par exemple on peut définir la fonction `f(x)=3*x+2` Dans cette fonction, `x` est la variable. C'est un nom que l'on donne pour désigner un élément quelconque de l'ensemble A. L'expression est `3*x`2+. Cette expression contient la variable, des constantes (2 et 3) et des opérations (+ et *). Ces opérations sont elles-memes des fonctions, définies avant `f`.

Le nom de la variable n'a pas d'importance. Par exemple, les deux définitions suivantes définissent une seule et même fonction: `f(x)=x+3` et `f(y)=y+3`.

Calcul du résultat de l'application

Pour calculer la valeur d'une fonction `f` définie par une expression pour une valeur donnée de l'ensemble de définition A, il faut remplacer dans l'expression la variable par cette valeur. Cela donne une expression dans laquelle il n'y a plus d'inconnue. On peut réaliser le calcul pour obtenir une valeur de l'ensemble B.

Par exemple, pour calculer la valeur de `f` définie par `f(x)=3*x+2` pour la valeur 5, on remplace `x` par 5 dans l'expression `3*x+2`. Cela donne l'expression `3*5+2` dans laquelle il n'y a pas d'inconnue et qu'on peut calculer pour obtenir le résultat `17`. On en conclut que +f(5)=17+.

Par rapport à la définition au moyen d'un graphe, la définition de fonction par une expression a l'avantage de permettre de définir des fonctions dont le graphe est infini. Par exemple, la fonction `f(x)=3*x+2` est définie pour tout l'ensemble des entiers relatifs, qui est un ensemble infini. Il y a donc une infinité de couples dans le graphe de cette fonction.

L'ordre des calculs n'est pas important, c'est à dire que le résultat obtenu à la fin est le même quel que soit l'ordre de réalisation du calcul. Il existe d'ailleurs un certains nombres de lois définissant l'équivalence

d'expressions, permettant de réaliser les calculs plus simplement (factorisation, associativité, commutativité).

Utiliser une construction par cas

Une fonction peut aussi ne pas être définie de la même façon suivant les valeurs de la variable. On utilise différentes expressions pour différents sous-ensemble de l'ensemble de définition A. On parle de fonction définie par morceau.

Voici quelques fonctions définies à l'aide de constructions par cas, écrites de différentes manières.

1. abs(x) = si x $\leq$ 0 alors x sinon (- x)
2. `par_morceaux` (x) =
   - x + 1 si x $\leq$ 1
   - x + 4 si x > 1 et x $\leq$ 100
   - x + 2 si x > 100
4. continue (x) =
   - sin(x) / x si x $\neq$ 0
   - 1 sinon

Pour qu'une telle définition soit valide, il faut que les différents cas soient mutuellement exclusifs, c'est à dire que pour une valeur donnée, il n'y ait qu'une définition.

Calcul du résultat de l'application

Pour calculer le résultat de l'application d'une fonction définie par morceau pour une valeur `v` donnée, il faut d'abord déterminer quel cas s'applique à cette valeur puis effectuer le calcul de l'expression donnée pour ce cas.

Utiliser la récursion

Les moyens déjà vus ne sont pas suffisants pour décrire par exemple la fonction factorielle de N dans N. La suite des valeurs de factorielle est souvent décrite comme suit :

```
0! = 1
n! = 1 * 2 * 3 * ... * (n-1) * n
```

L'écriture ci-dessus, même si elle est évocatrice, n'est en rien effective. Il est impossible d'écrire un algorithme contenant des points de suspension ! Pour définir une fonction calculant effectivement la factorielle d'un entier, il faut autoriser l'emploi du nom de la fonction en cours de définition dans l'expression qui la définit. Une telle définition sera dite récursive. Nous avons déjà rencontré des définitions récursives dans les descriptions de syntaxes. Voici une définition effective de la fonction factorielle : fact (n) = si n=0 ou n=1 alors 1 sinon n * fact (n-1)

Suivons le calcul de fact(3) en utilisant le symbole $\longrightarrow$ pour abréger la phrase ``se simplifie en".
fact(3) $\longrightarrow$ `3 * fact(2)` $\longrightarrow$ `3 * 2 * fact(1)` $\longrightarrow$ `3 * 2 * 1` $\longrightarrow$ 6 Le calcul a pu être mené à bien parce que le calcul de fact(1) est fait directement, sans appel de la fonction fact.

Pour être correcte, toute définition récursive de fonction, disons f, doit utiliser au moins une construction par cas avec au moins un cas dans lequel l'expression ne comporte pas la fonction f, de façon à permettre l'arrêt des calculs. Ce cas est appelé cas de base de la récursion.

Un autre exemple bien classique de fonction récursive est la fonction de Fibonnacci, définie par:

fib (n) = 1 si n =0 ou n=1
fib(n -1) + fib(n-2) sinon

Un autre exemple tout aussi célèbre est celui de la fonction d'Ackerman définie sur N $\times$ N par:

Ack(m, p) = p+1 si m=0
Ack (m-1,1) si p = 0
Ack(m-1, Ack(m,p-1)) sinon

Le lecteur est invité à calculer les valeurs Ack(m,p), pour les premières valeurs de m et p.

Fonction en Ada

Les fonctions servent à calculer des valeurs. On peut utiliser une fonction dans une expression.

On utilise une fonction:

- lorsque le calcul à faire est complexe. Le fait d'écrire une fonction permet de séparer la description du calcul de l'utilisation qui est faite du résultat. C'est un moyen de découper le problèmes en deux parties plus simples.
- lorsqu'un calcul revient à plusieurs endroits du programme, une fonction permet de ne décrire ce calcul qu'une fois et de l'effectuer plusieurs fois.
- l'utilisation d'une fonction, si son nom est bien choisi, peut faciliter la compréhension du programme.

Il y a deux temps distincts dans la vie d'une fonction dans un programme:

- la définition de la fonction: c'est la description du calcul que va faire la fonction. Elle apparait dans la partie déclaration du programme (c'est à dire entre `procedure machin is` et `begin`.
- l'utilisation de la fonction: pour réaliser le calcul. Cela apparait dans une expression, dans la partie instruction du programme (entre `begin` et `end machin`). Le terme technique pour désigner une utilisation de fonction est *appel de fonction*.

La déclaration d'une fonction comprend:

- un nom pour la fonction
- une liste de paramètres ou arguments de la fonctions: ce sont des valeurs que la fonction doit utiliser pour réaliser le calcul. Chaque paramètre a un nom et un type.
- le type du résultat calculé par la fonction.
- ce qu'on appelle le coprs de la fonction: une suite d'instructions qui permet de réaliser le calcul.

Parmi ces instructions, il y en a une sp{ecifique aux fonctions qui permet de terminer l'exécution de la fonction en renvoyant le résultat du calcul. Il s'agit de l'instruction `return` suivie d'une expression.

Exemple de définition de fonction: la valeur absolue d'un entier.

```
   function Valeur_Absolue(X: Integer) return Integer is
   begin
      if X >= 0 then
         return X;
      else
         return X*(-1);
      end if;
   end Valeur_Absolue;
```

- le nom de la fonction: `Valeur_Absolue`
- nom et type des paramètres: `X: Integer`
- type du résultat de la fonction: `return Integer`
- le corps: ce qu'il y a entre `begin` et `end Valeur_Absolue`

L'appel d'une fonction a la syntaxe suivante: le nom de la fonction suivi entre parenthèses d'autant d'expressions que la fonction a de paramètres, séparés par des virgules. Ces expressions doivent avoir le bon type, c'est à dire le type déclaré dans la définition de la fonction.

Exemple: la fonction `Valeur_Absolue` a un seul paramètre de type integer. L'appel doit donc comporter une expression de type integer.

- appel avec littéral pour expression: `Valeur_Absolue(-159)`
- appel avec variable pour expression: `Valeur_Absolue(Nombre)`
- appel expression comportant un opérateur: `Valeur_Absolue(Nombre - 252)`
- enfin, un appel de fonction est lui même une expression. On peut donc avoir un appel de fonction avec pour expression un autre appel de fonction (appel d'une autre fonction ou de la même): `Valeur_Absolue(Valeur_Absolue(Nombre - 252))`

Et voici un programme complet comportant la définition et un appel de fonction:

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Absolue is
   function Valeur_Absolue(X: Integer) return Integer is
   begin
      if X >= 0 then
         return X;
      else
         return X*(-1);
      end if;
   end Valeur_Absolue;
   Nombre: Integer;
   Val_Abs_De_Nombre: Integer;
begin
   Put("Entrez un nombre ");
   Get(Nombre);
   New_Line;
   Val_Abs_De_Nombre:=Valeur_Absolue(Nombre);
   Put("la valeur absolue de ce nombre est ");
   Put(Val_Abs_De_Nombre);
   New_Line;
end Absolue;
```

L'exécution d'une fonction ne concerne que l'appel de fonction. La déclaration d'une fonction ne s'exécute pas. Elle définit ce qui se passe lors de l'exécution d'un appel.

L'exécution d'un appel de fonction se fait de la façon suivant:

1. les expressions données entre parenthèses dans l'appel sont calculées.
2. De la mémoire est allouée pour les paramètres qui commencent à exister à ce moment-là. Les valeurs précédemment calculées sont stockées dans cette mémoire.
3. les instructions entre le `begin` et le `end` de la fonction sont exécutées jusqu'à atteindre un `return`.
4. le résultat du calcul de l'expression suivant le `return` est la valeur de l'appel de la fonction.
5. La mémoire allouée aux paramètres est libérée. Les paramètres cessent d'exister.

Exemple d'exécution d'un appel: `Valeur_Absolue(32 - 252)`

1. L'expression `32 - 252` est calculée. Valeur: `-220`.
2. De la mémoire est allouée pour le paramètre `X` et `-220` est mis dans cette mémoire. `X` commence à exister.
3. les instructions entre le `begin` et le `end` de la fonction sont exécutées. Cela commence par le `if`. L'expression booléenne `X >= 0` est évaluée. Pour cela, on regarde si le contenu de `X` (`-252`) est supérieur à 0. La valeur est false. On va donc exécuter ce qui se trouve entre le `else` et le `end if`. On va exécuter `return X*(-1);`. On commence par calculer l'expression `X*(-1)` qui vaut `252`. On arrête l'exécution de la fonction et le résultat renvoyé est `252`.
4. `X` est détruit et cesse d'exister.

On voit donc que le paramètre `X` n'existe qu'à l'intérieur de la fonction, qu'on ne peut l'utiliser qu'entre `function Valeur_Absolue` et `end Valeur_Absolue`.

Fonction: exécution détaillée

Nous allons revenir sur ce qui vient d'être expliqué en mots à l'aide d'un de nos tableaux permettant de décrire l'évolution de l'état mémoire de la machine. Nous allons reprendre l'exemple donné en le simplifiant et en référençant les lignes à l'aide d'un commentaire (cf la sous-section commentaire de ce chapitre).

Les paramètres et variables locales de la fonction n'existent que pendant l'appel de la fonction, le temps de calculer sa valeur.

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Absolue is
   function Valeur_Absolue(X: Integer) return Integer is
   begin
      if X < 0 then                -- a
         return X * (-1);          -- b
      else
         return X;                 -- c
      end if;
   end Valeur_Absolue;
   Nombre: Integer;
begin
   Nombre := Valeur_Absolue(-25);  -- 1
   Put(Nombre);                    -- 2
   New_Line;                       -- 3
   Nombre := Valeur_Absolue(56);   -- 4
   Put(Nombre);                    -- 5
   New_Line;                       -- 6
end Absolue;
```

Le tableau suivant décrit l'exécution pas à pas du programme. NEP signifie n'existe pas et (saut de ligne) note un passage à la ligne à l'écran. L'appel d'une fonction est une expression à l'intérieur d'une instruction. Si on veut détailler l'évaluation de l'appel, on est obligé de détailler ce qui se passe à l'intérieur d'une instruction. C'est pourquoi dans le tableau ci-dessous on distingue entre le début et la fin de certaines instructions (ex: instruction 1), entre les deux il se passe plein de choses.

Par rapport aux tableaux donnés les semaines précédentes, il y a une petite différence de présentation: nous avons choisi de décrire l'état à l'aide d'une colonne pour chaque identificateur (nom de variable ou de paramètre) du programme au lieu d'une unique colonne intitulée *état mémoire*. Il s'agit juste d'une différence de présentation.

instr	Nombre	X	écran	commentaire
debut 1	?	NEP		
a	NEP	-25		`X < 0` est calculé. Résultat: true
b	NEP	-25		`-1 * -25` est calculé et renvoyé
fin 1	25	NEP		le résultat est stocké dans X
2	25	NEP	25	
3	25	NEP	(saut de ligne)	
debut 4	25	NEP		
a	NEP	56		`X < 0` est calculé. Résultat: false
c	NEP	56		la valeur de X (56) est renvoyée
fin 4	56	NEP		le résultat est stocké dans X
5	56	NEP	56	
6	56	NEP	(saut de ligne)	

Comme on le voit dans le tableau, `X` n'existe pas dans le corps du programme (instruction numérotées par un chiffre), alors que `Nombre` n'existe pas dans le corps de la fonction (instructions numérotées par une lettre).

If imbriqués

Dans un `if..then..else..end if`, entre le `then` et le `else` et entre le `else` et le `end if` peuvent apparaître n'importe quelle instruction, donc éventuellement un autre `if..then..else..end if`. C'est ce qu'on appelle des conditionnelles imbriquées. Il faut bien comprendre la différence entre des `if` imbriqués et des `if` successifs. Nous allons voir cette différence sur un exemple.

Nous allons voir deux programmes qui diffèrent seulement par leur structure: ils comprennent les mêmes instructions, le même nombre de `if` avec les mêmes conditions, mais dans l'un, les `if` sont imbriqués et dans l'autre pas.

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Imbrication is
   A: Integer;
   B: Integer;
   C: Integer;
begin
   Put("Donnez-moi un nombre entier ");
   Get(A);
```

```
   Put("Donnez-moi un nombre entier ");
   Get(B);
   Put("Donnez-moi un nombre entier ");
   Get(C);
   if (A = 0) then
      Put("ici");
      New_Line;
   else
      if (B = 0) then
         Put("la");
         New_Line;
      else
         if (C = 0) then
            Put("la-bas");
            New_Line;
         else
            Put("nulle part!");
         end if;
      end if;
   end if;
end Imbrication;

with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure plat is
   A: Integer;
   B: Integer;
   C: Integer;
begin
   Put("Donnez-moi un nombre entier ");
   Get(A);
   Put("Donnez-moi un nombre entier ");
   Get(B);
   Put("Donnez-moi un nombre entier ");
   Get(C);
   if (A = 0) then
      Put("ici");
      New_Line;
   end if;
   if (B = 0) then
      Put("la");
      New_Line;
   end if;
   if (C = 0) then
      Put("la-bas");
      New_Line;
   else
      Put("nulle part!");
   end if;
end plat;
```

Essayez les deux programmes avec différentes valeurs pour A, B, et C.

Si vous ne comprenez pas bien la différence entre les deux, remplissez le tableau suivant:

A	B	C	A = 0	B = 0	C = 0	imbrique	plat
0	0	0	T	T	T		
0	0	1	T	T	F		
0	1	0	T	F	T		
0	1	1	T	F	F		

1	0	0	F	T	T		
1	0	1	F	T	F		
1	1	0	F	F	T		
1	1	1	F	F	F		

Ce tableau ne prend pas en compte toutes les valeurs possibles de A, B et C, mais il prend en compte toutes les valeurs de vérité possibles pour les trois conditions des `if`. Dans les deux dernières colonnes, indiquez le résultat de l'exécution du programme correspondant. Vous essayerez de deviner d'abord, puis de faire confirmer par une execution réelle sur votre machine.

Vous pouvez vous arrêter quand vous avez compris.

Complément sur les boucles

Ces compléments sont des notions importantes que nous vous invitons à étudier sérieusement.

Compteur de boucle For

La boucle For comporte un compteur de boucle qui a des points communs avec une variable, mais ce n'est pas une variable.

Voyons la boucle suivante:

```
for X in 1..5 loop
   put(X);
   New_Line;
end loop;
```

Ici le compteur de boucle est `X`. Les points communs entre ce compteur et une variable est:

- il a un nom
- le programmeur choisi ce nom
- X a un type. Dans nos premiers exemple, ce sera généralement un integer, mais nous verrons que cela peut être autre chose.
- X contient une valeur que l'on peut utiliser. C'est d'ailleurs ce qu'on fait lorsque l'on écrit `put(X)`. Cela affiche le contenu de `X`.

Les différences sont les suivantes:

- le compteur de boucle n'est pas déclaré dans la partie déclaration du programme.
- il n'existe qu'entre le `for` et le `end loop` correspondant.
- il n'est pas possible d'en changer la valeur par une affectation.

Essai d'affecter le compteur de boucle

Si on essaie de changer la valeur du compteur de boucle à l'intérieur de la boucle, cela produit une erreur de compilation. Voyez cela en compilant ce programme:

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Boucle_For is
begin
```

```
   Put("debut");
   New_Line;
   for Truc in 1..4 loop
      Put(Truc * 5);
      New_Line;
      Truc := Truc+1;
   end loop;
   Put("fin");
   New_Line;
end Boucle_For;
```

Conclusion: dans la boucle, on peut utiliser la valeur du compteur, on ne peut pas la modifier. On peut l'utiliser comme on utiliserait une constante.

Essai d'utiliser le compteur hors de la boucle

Essayez de compiler ce programme.

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Boucle_For is
begin
   for Truc in 1..10 loop
      Put(Truc * 5);
      New_Line;
   end loop;
   Put(Truc);
end Boucle_For;
```

`Truc` n'existe qu'entre `for` et `end loop`.

Essai de changer les bornes de l'intervalle

Si on essaie de changer les bornes de l'intervalle à l'intérieur de la boucle: essayez de comprendre ce que fait ce programme, en suivant l'exécution pas à pas. Au besoin faites un tableau pour suivre les valeurs de `Truc` et de `Nombre`. Puis compilez et exécutez ce programme pour voir ce qui se passe. Tirez-en les conclusions.

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Boucle_For is
   Nombre: Integer;
begin
   Nombre := 4;
   for Truc in 1..Nombre loop
      Put(Truc);
      New_Line;
      Nombre := 10;
   end loop;
   Put(Nombre);
end Boucle_For;
```

Terminaison

Le risque des boucles: qu'on n'en sorte jamais. Par exemple, le programme suivant ne s'arrête jamais:

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Boucle_Sans_Fin is
   Nombre: Integer;
begin
   Put("Donnez-moi un nombre entier ");
   Get(Nombre);
   loop
      Put(Nombre);
      New_Line;
      Nombre := Nombre - 1;
   end loop;
   Put("Et c'est tout...");
end Boucle_Sans_Fin;
```

Si vous voulez interrompre l'exécution de ce programme, vous devez taper `control-C` dans la fenêtre dos (en appuyant simultanément sur les touches `control` (ou `ctrl`) et `C`).

Dans la plupart des cas, les programmes corrects doivent s'arrêter, les boucles sans fin sont le signe d'une erreur de programmation. Il existe des cas de programme qui effectivement ne doivent jamais s'arrêter et boucler indéfiniment, par exemple les programmes qui contrôlent des automatismes comme des feux rouges ou la sécurité des centrales nucléaires.

Le risque de non-terminaison ne concerne que les boucles `while` et `loop`. Les boucles `for` ne sont exécutées qu'un nombre fini de fois.

Pour être sûr qu'une boucle termine, il faut que deux propriétés soient vérifiées:

1. la ou les conditions d'arrêt doivent être affectées par le corps de la boucle.
2. chaque tour de boucle doit rapprocher d'une condition d'arrêt.

Prenons des exemples.

```
loop
   exit when X > 5;
   Y := Y+1;
end loop;
```

Ici, la condition porte sur la valeur de `X` mais cette valeur ne change pas dans la boucle. Conclusion: soit on sort au premier tour de boucle, soit on ne sort jamais. C'est une erreur manifeste.

```
loop
   exit when Y = 5;
   Y := Y+1;
end loop;
```

Voilà qui est mieux. La condition dépend de `Y` et `Y` est modifiée à chaque tour de boucle. La première condition est remplie. Quid de la deuxième? Eh bien, en ajoutant 1 à Y, on l'augmente. Est-ce qu'en l'augmentant, on se rapproche de la condition? En fait, cela dépend de la valeur de Y. Si elle est supérieure à 5, en l'augmentant encore, on s'éloigne du but. Si elle est inférieure à 5, en l'augmentant, on va dans le bon sens. Cette même boucle pourra donc être correcte (si on est sûr en entrant dans la boucle que Y est plus petit que 5) ou incorrecte (dans le cas inverse).

Commentaires

Il est possible d'inclure des commentaires dans un programme Ada. Ces commentaires sont du texte libre, qui expliquent des points importants du programme, à destination d'un lecteur humain. A la compilation, les commentaires sont simplement ignorés par le compilateur.

Les commentaires sont introduits par le signe `--` et ils vont jusqu'à la fin de la ligne.

```
-- ceci est une ligne de commentaire
with Ada.Text_Io; use Ada.Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Somme_2_Entiers is
   X: Integer;   -- ceci est un bout de commentaire
   Y: Integer;
   Somme: Integer;
begin     -- je peux meme insulter le compilo
          -- dans un commentaire, ca passe
          -- Gnat est un #@*%$ !
   Put("Entrez un premier nombre : ");
   Get(X);
   Put("Entrez un second nombre : ");
   Get(Y);
   Somme := X + Y;
   Put("La somme vaut : ");
   Put(Somme);
   New_Line;
end Somme_2_Entiers;
```

Il ne faut pas abuser des commentaires. Il faut les utiliser quand on ne peut pas faire autrement. Il est plus simple de donner un nom parlant à une variable que de dire ce qu'elle fait par un commentaire.

Une autre façon de rendre un programme lisible sans commentaires, c'est en soignant la présentation. Il faut mettre en évidence les phénomènes d'imbrication des instructions qui en contiennent d'autres comme le If et les boucles. Pour cela on décale à l'aide d'espaces blancs les instructions contenues dans une autre.

Par exemple, pour un if, le `if`, le `else` et le `end if` alignés l'un au-dessus de l'autre, les autres instructions légèrement décalées. C'est vital quand on a des if imbriqués.

Pour vous aider, il existe dans l'éditeur de GNAT le bouton de mise en forme du programme de GIDE qui introduit automatiquement des espaces où il faut pour que le programme soit lisible.

Vous pouvez aussi utiliser l'option `reformat` du menu `tools` (c'est la même fonction que le bouton `R`).

Faites l'essai de taper un programme sans respecter la présentation, puis cliquez sur le bouton.

Les devoirs

Ce premier devoir va sembler très facile à beaucoup d'entre vous. Je préfère commencer calmement pour voir le niveau de la classe. Des devoirs plus difficile viendront plus tard.

La plus grande difficulté est de comprendre l'énoncé.

Rendez les devoirs!

Il est important de rendre les devoirs pour plusieurs raisons:

- les notes de devoir peuvent rattrapper une note insuffisante à l'examen (une note comprise entre 7 et 10)
- si vous avez des difficultés et que vous rendez un mauvais devoir, le prof peut vous aider. Il voit les problèmes que vous avez. Si vous ne rendez rien, il ne peut pas intervenir.
- cela vous oblige à travailler
- cela vous permet de vous évaluer, d'avoir un retour sur votre niveau. Le correcteur verra des erreurs que vous ne détectez pas seuls.
- toutes les erreurs que vous faites lors des devoirs, vous êtes à peu près sûr de ne pas les refaire à l'examen.

Conclusion: rendez les devoirs, même si ils ne se compilent pas, même si ils ne marchent pas. Si vous avez fait quelque chose de nul, vous aurez 5 et c'est toujours mieux que 0. Sur l'année, c'est tout à fait rattrapable.

Généralités sur les devoirs

Les devoirs ne concernent que les élèves inscrits au cours à distance.

Le plus souvent, on vous demande d'écrire un programme; Il faut rendre le code source de votre programme tel que vous l'avez tapé. La page web permet d'*uploader* (comment dit-on en français?) votre fichier. Il y a une case de texte libre si vous avez des remarques à faire pour le correcteur.

Cette page web est dans votre zone personnelle dont l'adresse et les codes d'accès vous seront communiqués par email dans le courant de la semaine du 12 novembre (si vous avez rempli le formulaire de post-inscription).

Tant que la page web est accessible, vous pouvez remplacer le programme déposé par une nouvelle version. Après la date limite, la page web n'est plus accessible, vous ne pouvez plus rendre le devoir.

Ne trichez pas: si un même programme est rendu par plusieurs personnes, elles ont toutes zéro et en plus elles se font mal voir. Il vaut bien mieux rendre un devoir mauvais, qui ne marche pas.

Une règle: vous ne devez pas poster d'explications ni de code en rapport avec le programme sur le forum. En revanche, vous pouvez poser des questions, par exemple si vous avez du mal à comprendre ce qui est demandé. Seul le prof répond à ces questions.

Enoncé du premier devoir

Le programme suivant comporte plusieurs erreurs:

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Double d'un Entier is
   Entier: boolean;
```

```
   Resultat: Float;
begin
   Get("Entier");
   New_Line;
   Put('entrez le nombre ');
   Get(Entier * 2);
   Resultat := Entier x 2;
   New_Line;
   Put(Le double du nombre est );
   Put("Resultat");
end Doubel d'un Entier;
```

Question 1

Donnez le programme corrigé, c'est à dire un programme qui ne produit pas d'erreur à la compilation et qui fait ce qu'on attend de lui à l'exécution (à savoir, afficher le double d'un nombre).

Question 2

Pour chaque erreur détectée, expliquez ce qui ne va pas. Dites si c'est une erreur de type ou une erreur de syntaxe.

Ce devoir est à rendre au plus tard le lundi 11 novembre par la page web de votre zone personnelle prévue à cet effet.

Exercice 4.1: fonction simple

Ecrivez une fonction qui calcule le carré d'un entier.

Question 1

Ecrivez cette fonction et un petit programme simple qui l'utilise.

Question 2

Ecrivez un programme qui utilise la fonction pour calculer le carre du carre du carre d'un nombre.

Une première version du programme se fera sans boucle, puis une deuxième version avec une boucle.

Question 3

Modifiez le programme avec boucle pour qu'il calcule le carre du carre du carre du carre du carre etc d'un nombre, où sera donné le nombre de carrés qu'on veut. Par exemple, on pourra calculer le troisième carré de 52 qui sera le carré du carré du carré de 52. Mais on pourra aussi calculer le cinquième carré de 17 qui sera le carré du carré du carré du carré du carré de 17.

Exercice 4.2: fonction

Le programme suivant comporte plusieurs calculs qui se ressemblent beaucoup. Ecrivez une fonction qui réalise ce calcul et modifiez le programme pour utiliser cette fonction.

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
```

```
procedure Faut_Fonction is
   X: Integer;
begin
   X:=17*3+2*3-5;
   Put(17*X+2*X-5);
   if 17*6+2*6-5 = X+7 then
      Put("Incrivel, nao e?");
   end if;
end Faut_Fonction;
```

Exercice 4.3: conditionnelles imbriquées

On veut écrire un programme qui calcule si une année est bissextile ou non.

La règle est la suivante: une année est bissextile si elle est divisible par 4, sauf si elle est divisible par 100, par contre celles divisibles par 400 sont bissextiles. Par exemple, 1996 est bissextile, car divisible par 4, 1900 n'est pas bissextile, car divisible par 100, mais 2000 est bissextile, car divisible par 400.

La table de vérité suivante retranscrit cette règle, en disant sila propriété *être bissextile* est vraie en fonction de la table de vérité des propriétés *être divisible par 4*, *être divisible par 100*, *être divisible par 400*.

divisible par 4	divisible par 100	divisible par 400	bissextile
T	T	T	T
T	T	F	F
T	F	T	impossible
T	F	F	T
F	T	T	impossible
F	F	T	impossible
F	T	F	impossible
F	F	F	F

Les cas impossibles viennent de ce que les 3 conditions d'être divisible ne sont pas indépendantes. Tous les nombres divisibles par 100 sont aussi divisibles par 4, etc.

Question 1: écrivez un programme qui détermine si une année est bissextile ou non en utilisant trois `if` imbriqués avec comme conditions *être divisible par 4*, *être divisible par 100*, *être divisible par 400*.

Question 2: écrivez un programme qui détermine si une année est bissextile ou non en utilisant un seul if avec une seule condition.

Evidemment, les deux programmes doivent donner les mêmes résultats.

Pour savoir si un nombre est divisible par un autre, on utilise l'opérateur `mod` qui donne le reste de la division entière (cf. exercice 3.3).

Exercice 4.4: premiers nombres pairs

Ecrire un programme qui calcule et affiche les `n` premiers nombres pairs, `n` étant entré par l'utilisateur.

Par exemple, les 4 premiers nombres pairs sont: 2, 4, 6 et 8.

Corrigé: Exercice 3.2, division entière

Question

Ecrire un programme qui lit deux nombres tapés au clavier et affiche le résultat de la division et le reste de la division avec un message du type:
dans 125 il y a 3 fois 36 et il reste 17

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Division is
   X: Integer;
   Y: Integer;
   Diviseur: Integer;
   Reste: Integer;
begin
   Put("Entrez le premier nombre ");
   Get(X);
   New_line;
   Put("Entrez le deuxieme nombre ");
   Get(Y);
   New_line;
   Diviseur := X/Y;
   Reste := X mod Y;
   Put("dans ");
   Put(X);
   Put(" il y a ");
   Put(Diviseur);
   Put(" fois ");
   Put(Y);
   Put(" et il reste ");
   Put(Reste);
   New_line;
end Division;
```

Corrigé de l'exercice 3.3

Le programme suivant a plusieurs erreurs.

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Multiplier Deux Entiers is
   X: Integer;
   Y: boolean;
   Resultat: Float;
begin
   Get("X");
   Put("Entrez le premier nombre ");
   New_Line;
   Put('entrez le deuxieme nombre ');
   Get(Y * 1);
   Resultat := XxY;
   New_Line;
   Put(Le produit des deux est );
   Put("Resultat");
end Multiplier Deux Entiers;
```

Question 1

Donnez le programme corrigé, c'est à dire un programme qui ne produit pas d'erreur à la compilation et qui fait ce qu'on attend de lui à l'exécution (à savoir, afficher le produit de deux nombres).

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Multiplier_Deux_Entiers is
   X: Integer;
   Y: Integer;
   Resultat: Integer;
begin
   Put("Entrez le premier nombre ");
   Get(X);
   New_Line;
   Put("Entrez le deuxieme nombre ");
   Get(Y);
   Resultat := X*Y;
   New_Line;
   Put("Le produit des deux est" );
   Put(Resultat);
end Multiplier_Deux_Entiers;
```

Question 2

Pour chaque erreur détectée, expliquez ce qui ne va pas. Dites si c'est une erreur de type ou une erreur de syntaxe.

- erreur de syntaxe: le nom de la procédure ne peut pas comporter d'espace.
- erreurs de type: pour multiplier deux entiers, il faut deux entiers et le résultat est entier. Donc `Y` et `Res` doivent être de type `integer`.
- erreurs de syntaxe: le paramètre de `get` doit être une variable et ne peut pas être une expression . Cela concerne `get("X")` et `get(Y 1)+`.
- erreur de logique: le premier `Put` doit être avant le premier `Get`. Ce n'est ni une erreur de syntaxe ni une erreur de type. Cela n'empêche pas la compilation ni l'exécution. Simplement, le comportement n'est pas celui qu'on désire.
- erreur de syntaxe: `'entrez le deuxieme nombre '` ne peut pas être correct. Il ne peut y avoir qu'un caractère entre quotes.
- erreur de syntaxe: l'opérateur de multiplication est `*` et non `x`.
- erreur de syntaxe: `Put(Le produit des deux est )` n'est pas correct. Il doit y avoir un seul paramètre. Ici, il y a 5 identificateurs, qui de plus n'ont pas été déclarés, séparés par des espaces, ce qui ne peut jamais se produire en Ada.
- erreur de logique: `Put("Resultat");` va afficher la chaine de caractère `"Resultat"` au lieu de nous afficher le nombre que l'on vient de calculer. Ici encore, cela n'empêche pas la compilation ni l'exécution. Simplement, le comportement n'est pas celui qu'on désire.

Corrigé de l'exercice 3.4

Ecrire un programme qui teste si un nombre est pair ou non. La méthode à utiliser consiste à regarder le reste de la division entière par 2. Si le reste est nul, le nombre est divisible par 2, il est donc pair. Vous pouvez consulter l'exercice 3.3 sur la division entière pour voir comment calculer un reste.

Pour calculer un reste, on utilise l'opérateur mod.

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Est_Pair is
```

```
   Nombre: Integer;
begin
   Put("entrez un nombre ");
   Get(Nombre);
   if (Nombre mod 2) = 0 then
      Put("ce nombre est pair");
   else
      Put("ce nombre n'est pas pair");
   end if;
   New_Line;
end Est_Pair;
```

Voilà, un petit `if` et l'affaire est bouclée.

Corrigé de l'exercice 3.5

Ecrire un programme qui calcule puis affiche la somme de 10 nombres saisis au clavier.

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Somme_10_Nb is
   X: Integer;
   Somme: Integer;
begin
   Somme := 0;
   for I in 1..10 loop
      Put("entrez un nombre ");
      Get(X);
      Somme := Somme+X;
   end loop;
   Put("la somme des nombres est ");
   Put(Somme);
   New_Line;
end Somme_10_Nb;
```

Modifier votre programme pour qu'il calcule et affiche également la moyenne entière. Note: on appelle la moyenne entière une moyenne qui utilise la division entière, c'est à dire le résultat de la division entière de la somme par le nombre d'entiers.

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Somme_Moy_10_Nb is
   X: Integer;
   Somme: Integer;
begin
   Somme := 0;
   for I in 1..10 loop
      Put("entrez un nombre ");
      Get(X);
      Somme := Somme+X;
   end loop;
   Put("la somme des nombres est ");
   Put(Somme);
   New_Line;
   Put("la somme des nombres est ");
   Put(Somme/10);
end Somme_Moy_10_Nb;
```

Modifiez encore une fois votre programme pour qu'il fasse le calcul sur un nombre quelconque de nombres saisis au clavier. Après chaque nombre entré, demander à l'utilisateur si il y a encore d'autres nombres à ajouter.

Comme on ne connaît pas le nombre de tours de boucle à l'avance, on ne peux pas utiliser un `For`. On va utiliser un `Loop` (`While` aurait été possible).

```
with Text_Io; use Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Somme_X_Nb is
   X: Integer;
   Somme: Integer;
   Rep: Character;
begin
   Somme := 0;
   loop
      Put("entrez un nombre ");
      Get(X);
      Somme := Somme+X;
      Put("Y a-t-il un autre nombre a entrer (o/n)?");
      Get(Rep);
      exit when Rep='n';
   end loop;
   Put("la somme des nombres est ");
   Put(Somme);
   New_Line;
end Somme_X_Nb;
```