Université Paris Ouest - Nanterre - la Défense

Licence mention Sciences du langage
2009 - 2010

MÉTHODES DE PROGRAMMATION

Résumé de cours

Marcel Cori

1 Qu'est-ce que le TAL ?

1.1 Une définition

Le *traitement automatique des langues* (désormais *TAL*) est encore appelé *traitement automatique du langage* ou *traitement automatique du langage naturel* (*Natural Language Processing* en anglais).

On parle aussi d'*industries de la langue* ou d'*ingénierie linguistique*, quand on met l'accent sur les produits destinés à des publics plutôt que sur les fondements des méthodes.

Traitement automatique, cela signifie traitement par des ordinateurs, traitement informatique. Or, pour caractériser un traitement informatique, il faut en premier lieu définir :

- ce qui *entre*, autrement dit les données qu'un *utilisateur* communiquera à une machine (*the input data*) ;

- ce qui *sort*, autrement dit, en simplifiant, ce qui sera affiché sur l'écran de l'ordinateur et qui fournira la résultat attendu par l'utilisateur (*the output data*).

Les traitements sont effectués grâce à des programmes, écrits par des *programmeurs*.

S'inscrivent dans le domaine du TAL les traitements qui obéissent aux deux critères qui suivent :

(1) ce qui entre est constitué de productions langagières (orales ou écrites) [1] ;

(2) le traitement tient compte des spécificités des langues humaines.

1.2 Cinq types de traitements

Dans ce qui suit nous évoquons, sans prétention à l'exhaustivité, différents types de traitements possibles qui se rattachent au TAL. Ces traitements sont distingués du point de vue de ce qu'ils réalisent, et non du point de vue des méthodes employées.

1.2.1 Correction

Tous les logiciels de traitement de texte incluent actuellement un outil de *correction orthographique* et/ou de *correction grammaticale*.

Ce qui entre est du texte, et le correcteur :

(1) informe l'utilisateur, d'une manière ou d'une autre, qu'il a détecté certaines erreurs. Par exemple, le correcteur inclus dans *Word 97* souligne en rouge les mots qu'il juge mal orthographiés et en vert les fautes de grammaire présumées.

(2) produit un diagnostic du type d'erreur qu'il a détectée.

1. Dans ce qui suit, on emploiera le mot *texte* au lieu de *production langagière*, pour des motifs de brièveté.

1.2.2 Traduction

Dès l'apparition des ordinateurs, on a songé à faire de la traduction automatique. Les projets de traduction automatique ont été réellement lancés durant les années 1950, dans le contexte de la Guerre froide : il fallait, notamment pour les Américains, être capables de traduire rapidement des textes techniques russes.

Mais, dans les années 1960, on a constaté la grande difficulté, voire l'impossibilité de résoudre le problème.

Les projets de traduction automatique ont refait surface à la fin des années 1970, en raison des progrès faits par les matériels informatiques et des nécessités résultant de l'accroissement de la mondialisation des échanges.

Enfin, ces dernières années, avec le développement du Web, chacun est mis en présence de textes en des langues diverses, et il n'est pas possible qu'un particulier fasse appel à chaque fois à un traducteur professionnel !

Actuellement, il existe de nombreux logiciels qui font soit de la traduction automatique, soit de la *traduction assistée par ordinateur* (ou de l'*aide à la traduction*). Dans ce dernier cas, la machine propose à l'utilisateur, qui est un professionnel de la traduction, un premier jet, et l'utilisateur produit une version acceptable par des lecteurs humains.

Il est clair que, comme pour la correction orthographique ou grammaticale, les logiciels actuels sont souvent pris en défaut.

1.2.3 Compréhension

On dira, peut-être par abus de langage, qu'il y a *compréhension* quand, à partir d'un texte, on produit une *action*. Et que l'action est celle que l'on est en droit d'attendre. Les actions peuvent être de types très divers, comme nous allons le voir ci-dessous.

La compréhension est définie ici selon un point de vue très pratique, celui de l'utilisateur qui bénéficie d'un service.

Ce qui différencie la compréhension de la correction ou de la traduction, c'est que deux textes très différents superficiellement peuvent produire une même action.

Nous allons examiner à présent trois exemples de traitements que nous rattacherons à la compréhension.

(1) L'*interrogation* « en langue naturelle », qu'elle soit orale ou écrite.

Il existe ainsi des standards téléphoniques automatiques, qui peuvent vous mettre en correspondance avec la personne ou le service compétent pour résoudre le problème que vous vous posez.

L'interrogation de bases de données est aussi une application possible.

(2) L'*analyse de réponses* en enseignement assisté par ordinateur (EAO).

En EAO, on pose des questions à un utilisateur-élève. Celui-ci fournit des réponses, qui peuvent être « en langue naturelle ». La machine doit alors analyser ces réponses afin

de produire des messages tels que: *votre réponse est exacte*, *votre réponse est inexacte* ou *votre réponse n'est pas assez précise*. Dans ce cas encore deux réponses d'élève peuvent être exactes tout en étant exprimées de manières différentes.

(3) La commande d'un robot, ou toute commande « en langue naturelle ».

1.2.4 Extraction

A partir d'un texte, on extrait des parties du texte, ou des informations concernant ce texte. Dans ce cadre, on peut classer :

(1) L'indexation automatique : il s'agit de déterminer une liste de mots clés susceptibles de décrire un texte (par exemple un article scientifique, ou une page Web), afin que celui-ci puisse être retrouvé par quelqu'un s'intéressant aux thèmes évoqués par le texte.

(2) La réalisation automatique de lexiques ou de dictionnaires à partir d'ensembles de textes (c'est-à-dire de corpus).

(3) La détermination de propriétés statistiques des textes.

(4) La réalisation de résumés automatiques de textes.

1.3 La chaîne des traitements

On se place dans le cas où la donnée d'entrée est du texte, qui se rencontre dans la quasi-totalité des situations de TAL. Il existe alors des traitements qui sont communs, indépendamment des perspectives pratiques.

1.3.1 La saisie du texte

Le texte est transmis à l'ordinateur de diverses façons :

(1) Les sons émis par une personne peuvent être captés et analysés à l'aide d'outils techniques. Ce qui peut donner lieu à une transcription dans un alphabet phonétique, puis en une suite de caractères[2].

(2) Un texte imprimé, ou même un texte manuscrit, peut être transmis à l'ordinateur à l'aide d'un scanner, en tant qu'image (à deux dimensions). Un outil logiciel est susceptible de traduire ensuite l'image en une suite de caractères (qui n'a plus qu'une dimension), le problème étant évidemment plus délicat pour les manuscrits.

(3) Les suites de caractères sont obtenues directement quand les textes sont saisis manuel-

2. On précisera plus loin, page 12 et suivantes, ce que l'on entend précisément par caractère.

lement.

Dans ce qui suit, on supposera que la donnée d'entrée des traitements est une suite de caractères. Les problèmes ne seraient pas fondamentalement différents pour de l'oral transcrit.

1.3.2 La segmentation

Partant d'une suite de caractères, il s'agit d'obtenir un découpage en unités, de l'ordre du morphème ou du mot[3]. A chacune des unités doivent être associées des informations linguistiques, notamment la catégorie associée au mot, ainsi que des traits.

Par exemple, si la donnée d'entrée est `"une femme chantait"`, on pourrait souhaiter obtenir comme sortie :

```
une          déterminant      féminin singulier
femme        nom              féminin singulier
chantait     verbe            3ème personne singulier imparfait
```

Plusieurs stratégies de découpage, qui peuvent d'ailleurs être combinées, sont susceptibles d'être choisies.

(1) On peut se servir des espaces[4] et des signes de ponctuation afin d'obtenir un découpage en mots.

Mais cela n'est pas sans poser de problèmes. Tout d'abord, certains signes de ponctuation, comme le trait d'union ou l'apostrophe, peuvent être internes ou externes aux mots: *chou-fleur, peut-être, l'animal, viendrez-vous aujourd'hui?*

Inversement, il ne serait pas possible de reconnaître les mots composés écrits sans traits d'union: *pomme de terre, chou rouge,...*

Enfin, on ne pourrait isoler les morphèmes à l'intérieur des mots.

(2) L'utilisation d'un *lexique*, ou *dictionnaire*, semble indispensable. Deux hypothèses, toutefois, sont possibles :

(i) Le dictionnaire peut être minimal, c'est-à-dire contenir uniquement un nombre limité de « mots grammaticaux » (déterminants, prépositions, etc.).

Des procédures peuvent alors être mises en œuvre afin de déterminer la catégorie des mots ne figurant pas dans le dictionnaire. Ces procédures peuvent se fonder sur la position par rapport aux mots grammaticaux, sur la connaissance des désinences verbales, etc.

(ii) Le dictionnaire peut contenir un maximum de mots ou de morphèmes d'une langue donnée, tout en n'étant évidemment pas exhaustif.

En ce cas, le découpage peut être guidé par l'appartenance des segments au dictionnaire.

3. Nous faisons appel à la notion intuitive de mot, notion qui est à discuter.
4. Les espaces étant a priori des caractères comme les autres, cf. plus loin, page 13.

Le choix d'un dictionnaire maximal est presque obligatoire pour nombre d'applications, dont la correction orthographique ou la traduction.

(3) L'*analyse morphologique* permet de se servir des règles de combinaison entre morphèmes afin d'obtenir des unités d'un niveau supérieur, et de construire les informations linguistiques concernant ces unités.

L'utilisation d'un analyseur morphologique présente les deux avantages suivants :

- la réduction de la taille des dictionnaires : si, en français, on répertoriait toutes les formes fléchies des verbes, cela multiplierait très fortement le nombre d'entrée du dictionnaire ;

- la possibilité de reconnaître des mots non répertoriés dans les dictionnaires (*recogner, affichable,...*), éventuellement en construisant un sens pour ces mots.

Il y a plusieurs façons de pratiquer l'analyse morphologique :

- soit on découpe d'abord suivant les espaces et les signes de ponctuation, puis on redécoupe en morphèmes les unités ainsi obtenues ;

- soit on découpe directement en morphèmes, et on reconstitue des unités d'un ordre supérieur grâce à la connaissance des règles morphologiques.

On notera que les règles de morphologie doivent être spécialement définies pour le traitement par ordinateur. Ainsi, on pourra considérer que la racine du verbe *manger* est *mang*, et que la désinence de la première personne du pluriel est *eons*. Ce qui fait que *manger* ne se conjugue pas comme *chanter*. De même, le verbe *aller* pourra admettre plusieurs (trois) « racines » : *all*, *ir* et *v*.

Il est clair, enfin, que le choix d'utiliser ou pas un analyseur morphologique est lié à la langue sur laquelle on travaille. Un tel analyseur est presque indispensable pour l'allemand, beaucoup moins pour l'anglais.

(4) L'ambiguïté lexicale.

Prenons un énoncé tel que *le boucher lit son livre*. Chacun des mots de cet énoncé peut recevoir plusieurs catégories, ou être ambigu du point de vue des traits.

le	déterminant	masculin singulier
	pronom	masculin singulier
boucher	nom	masculin singulier
	verbe	infinitif
lit	nom	masculin singulier
	verbe	3ème personne singulier présent
son	déterminant	masculin singulier
	nom	masculin singulier
livre	nom	masculin singulier
	nom	féminin singulier
	verbe	1ère personne singulier présent
	verbe	3ème personne singulier présent

Des ambiguïtés supplémentaires se présentent à l'oral[5].

C'est pourquoi nous définissons la notion de *forme lexicale*. Une forme lexicale est un objet superficiel, observable, qui peut recevoir un ou plusieurs ensembles d'informations linguistiques. Ainsi, *livre* est une forme lexicale.

Il existe des techniques de désambiguïsation, fondées sur la proximité. Cela peut inclure des outils statistiques.

Mais ces techniques, même si elles donnent des résultats relativement satisfaisants sur le plan pratique, sont discutables d'un point de vue linguistique. On se sert en fait de règles qui ne sont que le reflet superficiel de la structure syntaxique.

Il peut donc être préférable de réaliser une véritable analyse syntaxique, fondée sur un modèle syntaxique du langage.

1.3.3 L'analyse syntaxique

Dans les données d'entrée de l'analyse syntaxique figure la sortie de la segmentation, ou la sortie de la désambiguïsation.

Cela peut être par conséquent une suite de catégories (au cas où on ne conserve pas d'ambiguïté lexicale), ou une suite d'ensembles de catégories, ou encore une suite de formes lexicales, accompagnée de procédures permettant d'obtenir les catégories qui leur sont associées[6].

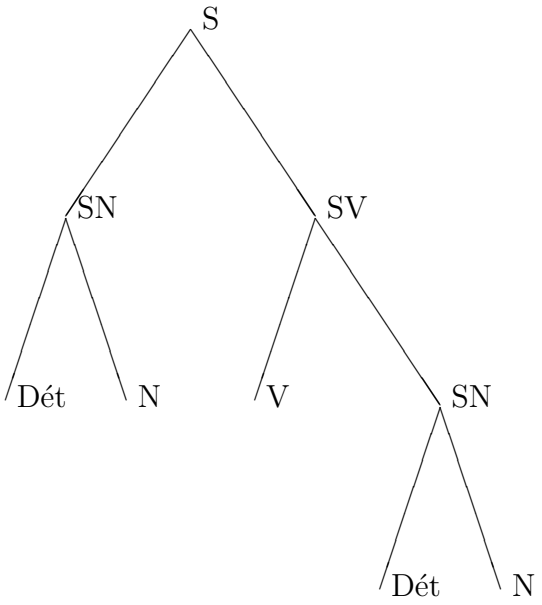
En sortie, on attend :

- un étiquetage plus précis (autrement dit, la résolution d'un certain nombre d'ambiguïtés lexicales et l'étiquetage des mots sur lesquels a échoué le recours au dictionnaire ou à l'analyse morphologique) ;
- la construction d'une structure syntaxique.

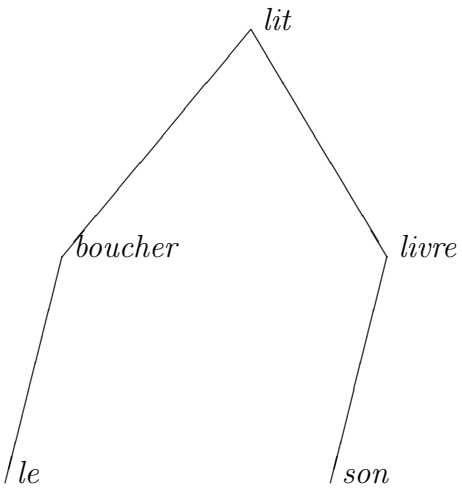
La structure syntaxique que l'on construit dépend du modèle que l'on adopte. Cela peut être un arbre : un arbre syntagmatique, mais aussi par exemple un arbre de dépendances. Ainsi, l'énoncé *le boucher lit son livre* pourrait donner lieu à la construction de l'un des deux arbres suivants :

5. En revanche, des ambiguïtés qui se rencontreraient à l'écrit sont levées à l'oral: *les poules du couvent couvent, les fils.*

6. Quand on parle ici de catégorie, on pense à des catégories accompagnées d'ensembles de traits.



arbre syntagmatique



arbre de dépendances

Mais le résultat sera une structure de traits, si on se réfère à un des formalismes les plus récents.

En fait, il arrive très souvent que pour un énoncé donné on obtienne plusieurs structures syntaxiques (à moins que l'analyseur ne soit bâti de manière à trancher dans tous les cas). On ne peut exclure non plus que l'analyseur se trouve dans la situation où il ne parvienne à construire aucune structure syntaxique.

1.3.4 L'insuffisance de la syntaxe

On s'aperçoit, dans la pratique, qu'un analyseur fondé uniquement sur des critères syntaxiques produit beaucoup de solutions non correctes.

Par exemple, on obtiendrait les deux mêmes arbres syntagmatiques pour les deux énoncés qui suivent :

Marie lit le livre
Marie lit la nuit

Le problème de l'*attachement des syntagmes prépositionnels* est bien connu. Il est illustré par les exemples suivants :

Jean mange une glace à la vanille
Jean mange une glace à la terrasse
Jean mange une glace au café
la statue de marbre de Rodin du duc d'Aumale de retour d'Afrique

De même, il n'est pas toujours évident de déterminer à quel nom ou groupe nominal

se rattachent certains adjectifs :

J'ai rencontré une directrice de société enrhumée
J'ai rencontré un possesseur de voiture électrique
J'ai trouvé une pièce de monnaie ancienne

Les ambiguïtés qui précèdent pourraient être levés à l'aide de critères *sémantiques*. Mais, cela n'est pas toujours possible. Considérons l'exemple suivant :

Connais-tu la sœur de Paul à qui Jean est marié ?

Il faut connaître les lois du pays pour savoir si le mariage homosexuel est autorisé ou pas. C'est de critères *pragamatiques* que l'on se sert alors pour désambiguïser. Comme pour :

Enlève le bonnet du bébé et mets-le à la machine à laver
Enlève le bonnet du bébé et mets-le au lit

On peut estimer que l'établissement du lien anaphorique, sur lequel porte ici l'ambiguïté, est extra-syntaxique. Il n'empêche que la connaissance de ce lien est indispensable pour certaines applications, comme la traduction automatique du français à l'anglais, où il faut savoir si on traduit le pronom *le* par *it* ou *him*.

De même pour déterminer si oui ou non il y a une faute d'orthographe dans l'énoncé qui suit, il faut savoir à quoi réfère le pronom relatif *que*.

Connais-tu la sœur de Paul que Jean a épousé ?

Considérons enfin les exemples suivants :

Pierre a apporté des fruits. Paul a mangé les cerises. Marie, quant à elle, a mangé les mûres.
Pierre a apporté des pommes. Paul a jeté les vertes. Marie, quant à elle, a mangé les mûres.

On s'aperçoit que la dernière phrase, commune aux deux énoncés, reçoit des interprétations différentes (et ne serait par conséquent pas traduite de la même manière): dans le premier énoncé *mûres* est un nom, alors que c'est un adjectif dans le deuxième.

C'est le contexte, formé par les deux premières phrases des énoncés, qui permet de trancher. Cela entraîne que l'on ne peut se contenter d'une analyse syntaxique phrase par phrase, mais qu'il est nécessaire d'aller vers une *analyse du discours*.

1.3.5 Les traitements ultérieurs

Les traitements ultérieurs dépendent crucialement des objectifs que l'on se fixe: on n'agit pas de la même manière quand on désire obtenir une traduction automatique ou analyser une réponse en EAO.

2 La représentation informatique des données

2.1 Le codage binaire de l'information

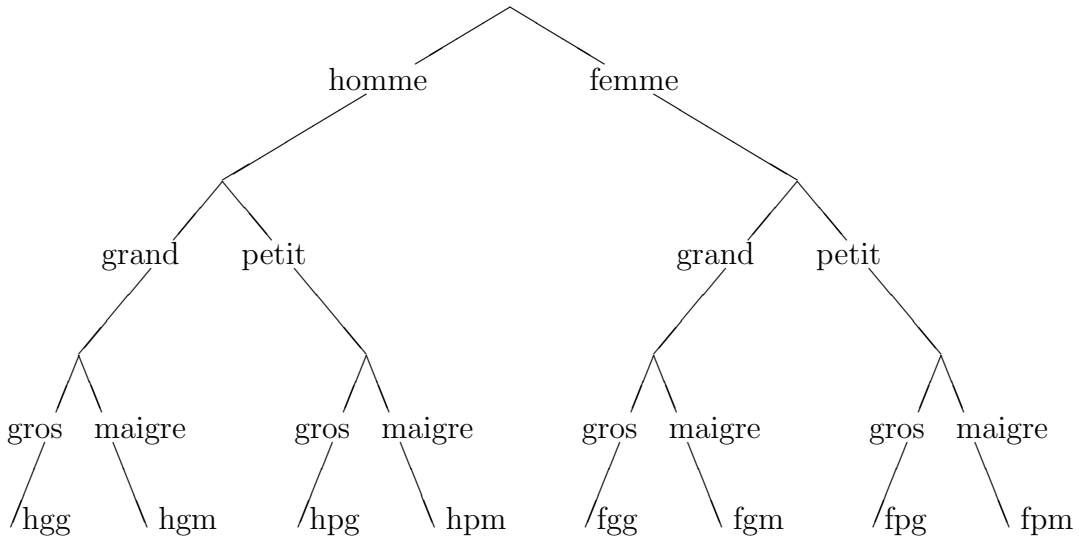
2.1.1 Des informations élémentaires aux informations complexes

Pour des raisons techniques (courant qui passe ou qui ne passe pas, interrupteur dans l'une ou l'autre position), les informations sont représentées sur les supports informatiques sous une forme *binaire*.

Le *bit* (*bi*nary dig*it*) est la quantité minimum d'information. Il correspond à une question à laquelle on peut répondre par *oui* ou *non*, ou par *vrai* ou *faux*.

Toute information peut être représentée par une suite de questions auxquelles on répond par *oui* ou par *non*, autrement dit par une suite de bits.

Ainsi, avec 3 bits on peut représenter 8 objets différents, ou 8 informations différentes, comme on le voit ci-dessous.

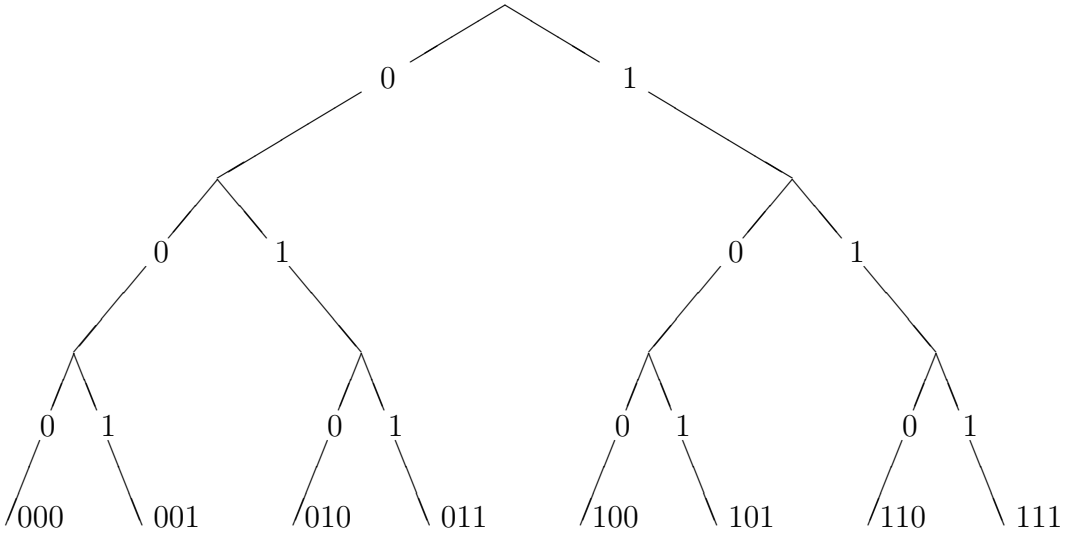


Avec 1 bit, on peut représenter 2 informations différentes ; avec 2 bits, on peut représenter $2 \times 2 = 4$ informations différentes ; avec 3 bits, on peut représenter $4 \times 2 = 8$ informations différentes ; avec 4 bits, on peut représenter $8 \times 2 = 16$ informations différentes, etc.

Plus généralement, avec N bits on peut représenter $2 \times 2 \times \cdots \times 2$ informations différentes, c'est-à-dire 2^N informations.

2.1.2 L'utilisation des nombres écrits en base 2

La représentation des deux valeurs possibles prises par une information s'effectue avec des 0 et des 1. Ce qui se traduit sous la forme arborescente suivante :



Le « chemin d'accès » à l'information obtenue à l'issue des trois choix peut s'écrire sous la forme d'une suite de `0` et de `1`. C'est donc un nombre écrit en base `2`, c'est-à-dire en binaire.

À chaque information complexe correspond ainsi un nombre, comme on le voit dans le tableau ci-dessous :

000	0	homme grand et gros
001	1	homme grand et maigre
010	2	homme petit et gros
011	3	homme petit et maigre
100	4	femme grande et grosse
101	5	femme grande et maigre
110	6	femme petite et grosse
111	7	femme petite et maigre

2.1.3 La traduction entre nombres binaires et nombres décimaux

Contrairement aux ordinateurs, nous manipulons plus facilement les nombres en base 10 (nombres décimaux) que les nombres écrits dans la base 2 (nombres binaires). C'est pourquoi il est utile de connaître des procédures permettant de passer des uns aux autres.

(1) Calcul de la valeur décimale d'un nombre écrit en binaire

En partant de la droite, on considère tous les `1`, et leur position dans la suite des chiffres. Ils correspondent respectivement aux valeurs décimales : `1, 2, 4, 8, 16, 32,` ...

Par exemple, `111` a pour valeur `1 + 2 + 4 = 7`, `1000` a pour valeur `8`, et `11010101` a pour valeur `1 + 4 + 16 + 64 + 128 = 213`.

(2) Écriture binaire d'un nombre décimal

On divise le nombre par `2`, puis on divise le quotient obtenu par `2`, et ainsi de suite. En écrivant de droite à gauche les restes obtenus au fur et à mesure, on obtient le nombre dont on est parti écrit en base `2`.

Par exemple :
`37` divisé par `2` donne `18` et il reste `1` ;
`18` divisé par `2` donne `9` et il reste `0` ;
`9` divisé par `2` donne `4` et il reste `1` ;
`4` divisé par `2` donne `2` et il reste `0` ;
`2` divisé par `2` donne `1` et il reste `0` ;
`1` divisé par `2` donne `0` et il reste `1`.

`37` s'écrit donc `100101` en base `2`.

2.2 Le codage des caractères

2.2.1 Le problème

Pour coder les 26 lettres de l'alphabet, on a besoin de 5 bits ($2^5 = 32$). Si on veut distinguer les lettres majuscules des lettres minuscules, ce sont 6 bits qui deviennent nécessaires ($2^6 = 64$). Mais si on ajoute les 10 chiffres, ainsi que plus de deux signes de ponctuation, ou des symboles qu'on trouve sur les claviers tels que « + », « = », etc., il nous faut 7 bits ($2^7 = 128$).

2.2.2 Le code ASCII

Rien n'obligeait au départ les fabricants d'ordinateurs et les concepteurs de logiciels à adopter le même codage pour les caractères. Un organisme, l'ISO (*International Organization for Standardization*), a été constitué afin de proposer des normes.

C'est ainsi qu'a été défini le code ASCII (*American Standard Code for Information Interchange*), ou norme ISO 646.

Ce code utilise 7 bits, autrement dit les nombres qui vont de 0000000 (zéro) à 1111111 (127). Les principaux codes qui ont été définis par la suite, et qui utilisent plus que 7 bits, englobent le code ASCII.

Les 32 premiers nombres binaires, autrement dit les codes qui vont de 0000000 à 0011111, ne représentent pas à proprement parler des caractères, mais ce qu'on appelle des *caractères de contrôle*[7]. Les 96 codes[8] restants représentent les caractères ci-dessous :

7. Nous donnons plus loin (page 15) un exemple d'utilisation des caractères de contrôle.

8. On remarquera qu'on utilise le terme *code* à la fois pour désigner une convention de codage (le code ASCII) et pour désigner le nombre binaire qui représente un caractère donné (le code 0100001 qui représente le caractère A).

32 :	33 : !	34 : "	35 : #	36 : $	37 : %	38 : &	39 : ’
40 : (	41 :)	42 : *	43 : +	44 : ,	45 : -	46 : .	47 : /
48 : 0	49 : 1	50 : 2	51 : 3	52 : 4	53 : 5	54 : 6	55 : 7
56 : 8	57 : 9	58 : :	59 : ;	60 : <	61 : =	62 : >	63 : ?
64 : @	65 : A	66 : B	67 : C	68 : D	69 : E	70 : F	71 : G
72 : H	73 : I	74 : J	75 : K	76 : L	77 : M	78 : N	79 : O
80 : P	81 : Q	82 : R	83 : S	84 : T	85 : U	86 : V	87 : W
88 : X	89 : Y	90 : Z	91 : [	92 : \	93 :]	94 : ^	95 : _
96 : ‘	97 : a	98 : b	99 : c	100 : d	101 : e	102 : f	103 : g
104 : h	105 : i	106 : j	107 : k	108 : l	109 : m	110 : n	111 : o
112 : p	113 : q	114 : r	115 : s	116 : t	117 : u	118 : v	119 : w
120 : x	121 : y	122 : z	123 : {	124 : |	125 : }	126 : ~	

On notera que l’espace est un caractère comme un autre, dont le code est 32, c’est-à-dire 0100000.

On notera également que les chiffres, en tant que caractères, ont un code qui est différent du nombre qu’ils représentent. Par exemple le caractère 1 a pour code ASCII 49.

2.2.3 Les codes ultérieurs

Le code ASCII ne prend pas en compte les accents ou autres signes diacritiques qui sont utilisés en français ou dans d’autres langues. C’est pourquoi ont été définis les codes *ISO-LATIN* (norme ISO 8859), *ISO-LATIN-1* pour le français.

ISO-LATIN-1 est défini sur 8 bits, et permet par conséquent la représentation de 256 caractères.

La compatibilité avec le code ASCII se vérifie en ce sens qu’en supprimant le 0 par lequel commencent les 128 premiers codes, on obtient un code ASCII qui représente le même caractère. Par exemple, le caractère a est représenté par 01100001 en ISO-LATIN-1 ou 1100001 en ASCII, ce qui correspond toujours au nombre 97.

Dans le tableau ci-dessous sont indiqués certains des codages, qui concernent notamment les lettres accentuées.

192 : À	194 : Â	196 : Ä	198 : Æ	199 : Ç	200 : È	201 : É	202 : Ê
203 : Ë	206 : Î	207 : Ï	209 : Ñ	212 : Ô	217 : Ù	219 : Û	220 : Ü
224 : à	226 : â	230 : æ	231 : ç	232 : è	233 : é	234 : ê	235 : ë
238 : î	239 : ï	241 : ñ	244 : ô	249 : ù	251 : û	252 : ü	255 : ÿ

On remarque toutefois l’absence de codes pour les caractères œ, Œ et Ÿ.

Sous Windows, on utilise un autre code (CP 1252), qui a une compatibilité avec ISO-LATIN-1 mais n'est pas identique. Ce code permet notamment les représentations suivantes :

Œ	140
œ	156
Ÿ	159
euro	128

Il existe aussi un code ISO 859-15, plus complet que le code ISO 8859-1.

En 1989 a été définie une nouvelle norme, la norme ISO 10646, qui, utilisant 32 bits, autorise le codage de plus de quatre milliards de caractères.

Un sous-ensemble de cette norme est le code *UNICODE* qui utilise 16 bits et autorise le codage de 65536 caractères. UNICODE est compatible avec ISO-LATIN-1.

2.2.4 Le codage des caractères non ASCII sur 7 bits

Il reste que tous les systèmes informatiques n'admettent pas de travailler sur 32, 16 ou même 8 bits. Ainsi, les transferts de données par Internet sous le protocole SMTP (*Simple Mail Transfert Protocol*) prennent en compte des caractères de 7 bits.

Il existe bien des procédures de codage/décodage entre 8 bits et 7 bits, comme la procédure MIME (*Multi-Purpose Internet Mail Extensions*), utilisée par des logiciels de courrier électronique.

Mais il est plus sûr pour éviter les risques de distorsions de ne transmettre que des caractères codés sur 7 bits (et donc n'utilisant que le code ASCII).

Les documents créés sous le format Latex ou le format HTML n'utilisent que les caractères du code ASCII. Ces documents ont par conséquent la propriété d'être plus que les autres indépendants des machines et des systèmes sur lesquels ils sont lus.

Les caractères non ASCII sont codés par une suite de caractères ASCII. Par exemple, le `é` est codé `\'e` en Latex et `&eacute;` en HTML.

2.3 Des fichiers aux documents

2.3.1 Fichiers textes

On peut considérer que les fichiers qui contiennent des données linguistiques sont constitués de suites de caractères. On les appellera *fichiers textes*[9]. On supposera dans ce qui suit que les fichiers textes sont formés de suites d'*octets* (*bytes*), un octet étant une suite de 8 bits.

9. Ils diffèrent par exemple des fichiers qui contiennent des images, ou des fichiers qui contiennent des programmes exécutables.

Cela signifie que le fichier est organisé de manière linéaire, selon une dimension unique : on ne retrouve pas les deux dimensions de l'écran, d'une page imprimée ou d'une page manuscrite.

2.3.2 Les ruptures de ligne

Quand on tape un texte sous un logiciel de *traitement de texte* comme *Word*, on pratique la frappe *au kilomètre*, c'est-à-dire qu'on ne se préoccupe pas des passages à la ligne. C'est le logiciel qui, au cours même de la frappe, décide automatiquement d'aller à la ligne. On ne frappe sur la touche « Entrée » que lorsqu'on veut forcer le passage à la ligne, c'est-à-dire en fin de paragraphe.

Sous d'autres logiciels, qui sont aussi des *éditeurs* de fichiers textes, comme le bloc-notes de *Windows*, il est obligatoire de frapper sur la touche « Entrée » pour aller à la ligne.

À ce moment-là, le logiciel insère un (ou deux) caractère(s) particuliers, ainsi les caractères de contrôle de codes ASCII respectifs 10 et 13. Le fichier reste organisé de manière linéaire, mais les logiciels qui en gèrent l'affichage à l'écran ont les moyens de le structurer selon deux dimensions.

2.3.3 La mise en forme des documents

De nombreuses autres spécifications permettent de mettre en forme les documents afin d'obtenir des textes imprimables ayant la meilleure présentation possible.

Par exemple, on peut jouer sur la taille des caractères, avoir des caractères gras, soulignés ou en italique, obtenir des marges plus ou moins grandes, plusieurs colonnes, la justifcation ou le centrage des lignes, des notes en bas de page, la numérotation des pages, etc[10].

10. On ne fera pas ici une liste exhaustive des possibilités offertes par les logiciels de traitement de texte.

3 Généralités sur la programmation

3.1 Les personnages

En informatique, il faut distinguer :

- les *utilisateurs*, qui se servent d'un logiciel de traitement de textes ou d'un « tableur », qui accèdent à des informations grâce à Internet, qui font des jeux sur ordinateur ou communiquent grâce au courrier électronique,

- de ceux qui élaborent les algorithmes et les programmes, que nous appellerons ici les *programmeurs*[11].

C'est pourquoi on peut considérer grossièrement trois « personnages » en informatique : le programmeur, l'utilisateur et la machine.

Le programmeur est celui qui s'adresse à la machine afin qu'elle exécute des tâches qui servent à l'utilisateur.

Les machines existantes, les ordinateurs, ont les caractéristiques suivantes :

- elles sont très puissantes, c'est-à-dire qu'elles ont une mémoire très importante et qu'elles sont très rapides ;

- elles sont infaillibles, c'est-à-dire que, mises plusieurs fois dans les mêmes conditions, elles réagissent d'une manière parfaitement identique ;

- elles sont très obéissantes ;

- elles sont très bêtes.

La question fondamentale qui se pose au programmeur est de savoir exprimer des ordres de telle manière que la machine agisse comme il le souhaite : ce sont ces ordres qui constitueront les *programmes*.

3.2 Algorithmes

Le travail du programmeur comporte deux phases essentielles : (1) la définition très précise de la tâche, c'est-à-dire de la classe d'actions, que la machine aura à effectuer ; (2) la décomposition, si nécessaire, de cette tâche en tâches plus réduites.

Ces deux phases peuvent être réalisées indépendamment de la machine et du langage qui seront finalement utilisés.

3.2.1 La formulation des problèmes

Le programmeur doit tout d'abord savoir lui-même très exactement ce qu'il attend de la machine, quelle tâche il désire que celle-ci exécute. Ce qui suppose au minimum et en

11. Dans certains cas, en particulier dans les situations d'apprentissage, le programmeur et l'utilisateur se confondent. Mais en fait, c'est parce qu'une même personne joue alternativement les deux rôles.

premier lieu de définir quelles seront les *données externes* qui seront fournies à la machine (par un utilisateur) et ce que la machine devra en sortir.

Par exemple, si je veux que la machine fasse une traduction automatique, je peux définir le problème de la manière suivante :

- la donnée externe sera une phrase en français ;

- le résultat sera la traduction de cette phrase en anglais.

Ce qui donne lieu à des question très ardues, telles que : qu'est-ce qu'une phrase en français? Quels sont les critères qui permettent de déterminer si une traduction est acceptable?

On le voit : avant même de passer à la conception des algorithmes, on se trouve face à des difficultés majeures. D'une certaine manière, on peut dire que la phase cruciale de l'informatisation d'un domaine réside dans la *formulation des problèmes*. Cette phase est interne au domaine donné, mais c'est l'informatisation qui la rend indispensable.

3.2.2 La conception des algorithmes

Dans une deuxième étape, le programmeur doit pouvoir trouver de quelle façon l'ordinateur exécutera une tâche qui aura été définie : *la méthode*.

Plus exactement, quand on conçoit un algorithme, on commence par déterminer grossièrement comment la tâche sera effectuée, puis on décompose la résolution de son problème en tâches plus élémentaires, cette décomposition pouvant être réitérée.

On obtient finalement une succession d'actions à exécuter. Ces actions seront exprimées en terme d'*instructions* adressées à l'ordinateur.

3.2.3 Définition des algorithmes

Un *algorithme* est constitué d'une suite finie d'instructions décrivant une action ou une classe d'actions.

Un algorithme a un degré de généralité d'autant plus élevé que la classe d'actions qu'il décrit est plus large.

Ainsi, si on reprend l'exemple ci-dessus, l'algorithme sera d'autant plus général que le vocabulaire qu'il saura prendre en compte sera étendu. Il sera évidemment encore plus général si d'autres langues que le français et l'anglais sont admises. Etc.

3.2.4 Caractéristique des algorithmes

L'ordinateur étant très obéissant et très bête, il fait *tout* ce qu'on lui dit de faire, mais il ne fait *que* ce qu'on lui dit de faire. Il faut par conséquent, dans un algorithme, ne rien omettre et prévoir toutes les situations possibles.

Par ailleurs, l'ordinateur ne prend aucune décision. Il ne peut trancher si on lui transmet un ordre ambigu. On doit donc tout expliciter, ne rien laisser sous-entendu.

3.3 Programmes et langages de programmation

3.3.1 Programmes

Un ordinateur ne peut effectuer qu'un nombre très limité de tâches élémentaires, même s'il peut effectuer et enchaîner ces tâches très rapidement et sans se tromper.

Pour pouvoir transmettre des ordres à l'ordinateur, il faut donc connaître l'ensemble des tâches élémentaires que celui-ci sait exécuter. A chcune de ces tâches élémentaires correspondra une instruction élémentaire.

Nous appellerons *programme* un algorithme écrit dans un langage immédiatement compréhensible par une machine, autrement dit dans un langage de programmation.

3.3.2 Langages de programmation

Il serait trop ardu et fastidieux pour le programmeur de s'exprimer en « langage machine » (qui est un langage binaire, c'est-à-dire composé de suites de 0 et de 1).

Le programmeur ne peut non plus s'adresser à la machine dans une langue naturelle, ambiguë et trop imprécise.

Les langages de programmation sont des langages intermédiaires entre la langue naturelle et le langage de la machine.

La sémantique de ces langages est très rigide. Le langage correspond très exactement aux actions élémentaires que la machine sait effectuer. En définissant le langage, on définit sa sémantique.

3.3.3 La traduction des programmes

L'ordinateur, face à un programme écrit en un langage qui n'est pas le sien, se trouve devant deux possibilités :

- ou bien il traduit ce programme au fur et à mesure qu'il agit : dès qu'il a compris une instruction, il l'exécute. C'est le cas pour les langages *interprétés* ;
- ou bien il traduit tout le programme avant de commencer à agir. C'est le cas pour les langages *compilés*.

3.3.4 Les erreurs de syntaxe

Si l'ordinateur ne parvient pas à traduire tout le message qui lui a été communiqué, cela voudra dire qu'il y aura eu, de la part du programmeur, une *erreur de syntaxe* dans le langage de programmation : la machine ne comprend pas les ordres qu'on lui donne.

Si l'on a affaire à un langage compilé, lorsque la machine rencontre une erreur de syntaxe, elle ne peut rien effectuer de ce qu'on lui a demandé de faire.

Un des avantages des langages interprétés est que, même en cas d'erreur de syntaxe, un programme peut être exécuté au moins partiellement puisqu'il ne s'arrêtera qu'au moment où il rencontrera l'erreur. Le défaut est que l'ordinateur est obligé de retraduire les ordres

qu'on lui donne à chaque fois qu'il doit les exécuter, alors que dans le cas des langages compilés il lui suffit de traduire ces ordres une fois pour toutes, d'où un gain de temps.

3.3.5 Langages d'algorithmes

La décomposition d'un problème qui induit la conception d'un algorithme peut s'effectuer à un plus ou moins grand degré de précision. Toutefois, on peut se donner *a priori* un jeu d'instructions élémentaires dans lequel on voudra que soient écrits tous les algorithmes.

Ces instructions constitueront alors un *langage d'algorithmes*.

3.4 Exécution des algorithmes

3.4.1 Un processus temporel

L'*exécution* d'un algorithme est le processus déclenché par l'algorithme sur la machine. Ce processus est temporel, en ce sens que la réalité (et notamment la réalité interne de la machine) est modifiée au cours du temps.

On notera que l'exécution d'un algorithme peut être simulée par un être humain. Ainsi, le programmeur, pour savoir si un algorithme écrit sur une feuille de papier est correct, peut le « faire tourner à la main ».

3.4.2 Les différentes exécutions

Un algorithme donné, invariable, peut connaître plusieurs exécutions différentes.

Ce sont les données externes de l'algorithme, fournies en général par l'utilisateur, qui varient d'une exécution à l'autre.

L'ensemble des jeux de données externes possibles définit la classe d'actions que peut effectuer l'algorithme.

3.4.3 Erreurs à l'exécution

Il se peut qu'un programme, syntaxiquement correct, donne lieu à des *erreurs à l'exécution* : soit la machine se bloque et ne parvient à aucun résultat si ce n'est un message d'erreur, soit elle fournit des résultats qui ne correspondent pas à ce que le programmeur a cru lui avoir demandé de faire, soit la machine semble « tourner » indéfiniment. Ces erreurs ne proviennent bien entendu pas de l'ordinateur (qui fait exactement ce qu'on lui dit de faire), mais du programmeur. Plusieurs hypothèses sont alors possibles :

- l'exécution du programme nécessite un temps trop élevé, ce temps pouvant éventuellement être infini ;
- le programme comporte des instructions dont la succession est incohérente ;
- ce que le programmeur a écrit est parfaitement cohérent, mais ne correspond pas à ce qu'il voulait que l'ordinateur fasse ;
- ce que le programmeur a écrit ne voulait rien dire (tout en étant syntaxiquement correct) ;

- l'utilisateur fournit des données externes qui sortent de ce que le programmeur avait envisagé.

Dans ce dernier cas, cela ne signifie pas que l'algorithme est erroné. Mais que la classe des problèmes traités n'inclut pas certaines données externes.

4 Algorithmes élémentaires

4.1 Constantes et variables

4.1.1 Données internes

Considérons l'algorithme suivant :

Algorithme 4.1

```
écrire("quel est votre prénom")
lire (prénom)
écrire ("bonjour")
écrire (prénom)
écrire ("Nous vous souhaitons la bienvenue")
```

On constate que l'instruction `écrire` s'emploie de deux manières différentes :
- ou bien `écrire("bonjour")` quand il s'agit de faire apparaître à l'écran une suite de caractères *constante*, suite qui est déterminée par le programmeur ;
- ou bien `écrire(prénom)` quand il s'agit de faire apparaître à l'écran une suite de caractères non connue à l'avance par le programmeur. Une telle suite, dans le cas présent, sera fournie par l'utilisateur à chaque exécution de l'algorithme. Cette suite n'étant pas à chaque fois la même, on dira qu'il s'agit d'une *variable*.

Les constantes comme les variables sont des *données internes*.

4.1.2 Type, nom, valeur

Une donnée interne a un *type*, une *valeur* et éventuellement un *nom*. Le nom est fixe, alors que la valeur peut varier d'une exécution à l'autre de l'algorithme ou au cours d'une exécution de l'algorithme.

4.1.2.1 Cas des constantes

Les constantes sont des données internes dont la valeur ne change ni d'une exécution à l'autre de l'algorithme, ni au cours d'une exécution de l'algorithme.

Par exemple `"bonjour"` est une constante de type `suite de caractères`. D'autres exemples de type sont `caractère`, `nombre entier` ou `nombre réel`.

Ainsi, `"e"` est une constante de type `caractère`, `15` est une constante de type `nombre entier` et `20.66` est une constante de type `nombre réel`.

4.1.2.2 Cas des variables

Les variables sont des données internes dont la valeur peut varier. Une variable est désignée dans un algorithme par son nom. Dans l'algorithme 4.1, le nom de la variable utilisée est `prénom`.

Le nom de la variable est fixé par le programmeur. Par convention, le nom d'une variable commence obligatoirement par une lettre (et pas par un chiffre).

Ceci permet que les constantes de type nombre ne soient pas écrites entre guillemets. En revanche, on accepte des noms comprenant des chiffres en dehors de la première place, tels que `a1` ou `h2o`.

Une variable, pour être utilisée, doit posséder une *valeur*. Dans le cas présent, la valeur est communiquée à la machine par l'utilisateur (à l'aide de l'instruction `lire`). La valeur de la variable peut différer d'une exécution à l'autre de l'algorithme. Mais elle peut également varier au cours d'une même exécution d'un algorithme.

On peut assimiler le nom d'une variable à un *contenant*, et la valeur au *contenu*. Toutefois, quand un nom apparaît dans un algorithme, il renvoie, selon la place qu'il occupe, soit au contenant, soit au contenu.

4.1.3 L'initialisation des variables

A l'exécution d'un algorithme, il faut que la première fois que l'on rencontre un nom désignant une variable, ce soit en tant que contenant, autrement dit que la variable qu'il désigne soit *initialisée*.

L'initialisation s'effectue par exemple grâce à une instruction `lire`, comme dans notre exemple.

Si on fait allusion à un contenu sans qu'il ait été initialisé, cela provoque une erreur (erreur à l'exécution due au fait que les instructions ne se succèdent pas comme il le faudrait). Plus précisément, deux conséquences sont possibles : soit la machine se bloque, soit elle affecte une valeur arbitraire à la variable non initialisée.

4.2 La communication entre la machine et l'utilisateur

4.2.1 Ecriture

L'instruction `écrire` permet à la machine de communiquer à l'utilisateur une valeur désignée dans l'algorithme par une constante, un nom (ou une expression comme on le verra plus loin).

Elle a la forme suivante :

```
écrire(<valeur>)
```

On étend les possibilités de cette intruction en permettant que soient communiquées plusieurs valeurs à l'utilisateur à l'aide d'une seule instruction `écrire`. Par exemple :

```
écrire("bonjour ", prénom)
```

4.2.2 Lecture

L'instruction `lire` permet à la machine de prendre connaissance d'une donnée externe qui lui est communiquée par l'utilisateur.

Elle a la forme suivante :

`lire(<nom>)`

A l'exécution, si l'utilisateur communique à la machine une donnée externe incompatible avec le type du nom, il se produit une erreur.

4.3 Tests et instructions conditionnelles

4.3.1 Exemple introductif

Considérons l'algorithme suivant, écrit dans une perspective d'enseignement assisté par ordinateur (EAO) : l'utilisateur-élève doit répondre à une question qui lui est posée par la machine. Selon cette réponse, la machine lui dit s'il s'est trompé ou pas.

Algorithme 4.2

```
écrire ("Quelle est la capitale de la France?")
lire (réponse)
si réponse = "Paris" alors
     écrire ("c'est exact")
     sinon écrire ("ce n'est pas la bonne réponse")
```

4.3.2 Tests

Cet algorithme contient un *test* :

`réponse = "Paris"`

Un test est une question à laquelle la machine doit pouvoir répondre par oui ou par non. Nous considérons pour l'instant que la forme d'un test est la suivante :

⟨`valeur`⟩ ⟨`opérateur conditionnel`⟩ ⟨`valeur`⟩

les opérateurs conditionnels étant : "`=`", "$\neq$" (différent), "$<$" (inférieur), "$\leq$" (inférieur ou égal), "$>$" (supérieur), "$\geq$" (supérieur ou égal).

4.3.3 Instructions conditionnelles

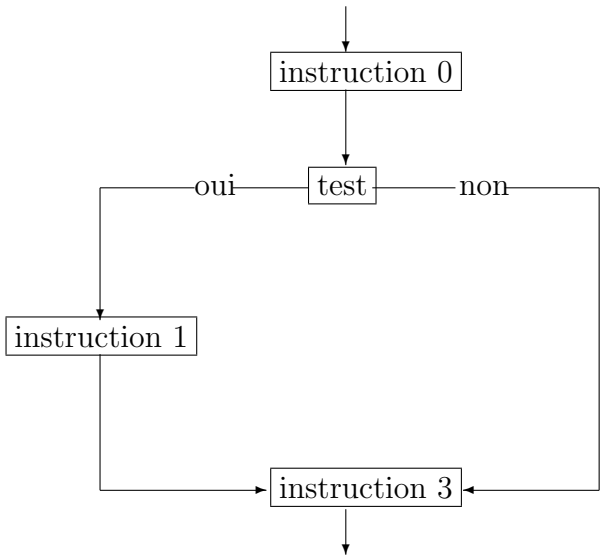
Une *instruction conditionnelle* est une instruction qui contient d'autres instructions qui seront ou ne seront pas exécutées, en fonction du résultat d'un test.

On considérera deux instructions conditionnelles différentes :

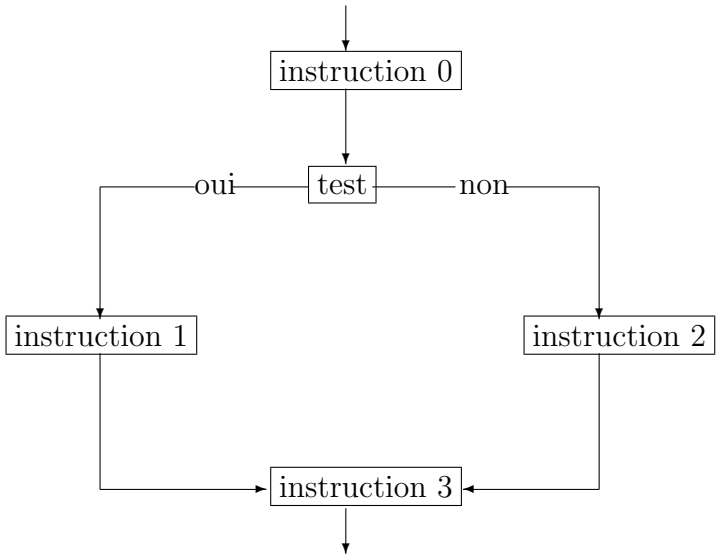
`(1) si <test> alors <instruction 1>`

`(2) si <test> alors <instruction 1> sinon <instruction 2>`

Nous explicitons ce que font ces instructions conditionnelles, dans un contexte donné, de manière graphique :




```
<instruction 0>
si <test> alors <instruction 1>
<instruction 3>
```




```
<instruction 0>
si <test> alors <instruction 1> sinon <instruction 2>
<instruction 3>
```

4.3.4 Les blocs

Considérons une modification de l'algorithme d'enseignement assisté par ordinateur ci-dessus : cette fois-ci on demande à l'utilisateur le nom du Président de la République, et au cas où le nom est exact, on lui demande le prénom du Président.

Algorithme 4.3

```
écrire ("Quel est le nom du Président de la République?")
lire (réponse)
si réponse = "Sarkozy" alors
     début
     écrire ("quel est son prénom")
     lire (réponse)
     si réponse = "Nicolas" alors
          écrire ("c'est exact")
     sinon écrire ("ce n'est pas le bon prénom")
     fin
sinon écrire ("ce n'est pas la bonne réponse")
```

Afin d'exécuter plusieurs instructions dans le cas d'une première réponse correcte, on est obligée de les enclore dans un *bloc*. Le bloc est délimité par un `début` et une `fin`.

On remarquera que dans cet exemple la variable `réponse` pourra contenir successivement deux valeurs distinctes au cours d'une même exécution de l'algorithme.

4.4 L'instruction d'affectation

On peut proposer une variante dans l'écriture de l'algorithme 4.2.

Algorithme 4.4

```
message := "ce n'est pas la bonne réponse"
écrire ("Quelle est la capitale de la France?")
lire (réponse)
si réponse = "Paris" alors
     message := "c'est exact"
écrire (message)
```

On utilise une variable (de type `suite de caractères`) appelée `message` qui va contenir le message à faire parvenir à l'élève. Au départ ce message est intialisé à la valeur `"ce n'est pas la bonne réponse"` ; ce n'est que dans le cas où l'élève donne la bonne réponse que la valeur de cette variable est modifiée.

Afin d'intialiser et de modifier cette variable, on se sert de l'instruction d'*affectation*, qui est une instruction de transfert interne.

Cette instruction a la forme générale suivante :

<nom>:=<valeur>

Il est nécessaire que les données internes situées à gauche et à droite du signe d'affectation soient de même type.

Dans le cas où la valeur est désignée par un nom, l'instruction d'affectation aura une forme telle que :

```
a := b
```

Juste avant l'exécution de l'instruction d'affectation, il faut que la variable `b` ait été initialisée, pas nécessairement la variable `a`. A l'issue de l'exécution de cette instruction, la valeur de `b` est inchangée, tandis que la valeur de `a` est changée : elle est désormais égale à la valeur de `b`.

Si on veut échanger les contenus des cases `a` et `b`, on est par conséquent tenu d'utiliser une troisième case, ou case auxiliaire, selon la méthode suivante :

```
c := a
a := b
b := c
```

5 Opérations

5.1 Les opérations et leurs notations

5.1.1 Les opérations

5.1.1.1 Définition

Une *opération* associe à un certain nombre d'objets d'un type déterminé (les *opérandes*) un objet de type également déterminé (le *résultat*).

Par exemple, l'addition de nombres entiers associe à deux objets de type `nombre entier` un troisième objet de type `nombre entier`, selon un processus bien connu.

Une opération est dite *n-aire* si elle comprend n opérandes. n est en ce cas l'*arité* de l'opération. Ainsi, l'addition est une opération 2-aire, ou binaire.

Le type des opérandes et du résultat définissent le *type de l'opération*. Nous dirons par exemple que l'addition de nombres entiers est une opération de type

⟨`nombre entier, nombre entier` → `nombre entier`⟩.

Si l'on ne respecte pas les contraintes relatives au type dans l'écriture d'une opération, il y aura une erreur de syntaxe qui sera détectée. Par exemple :

```
a := 27 / "bonjour"
```

5.1.1.2 Sémantique des opérations

Le fait de connaître le type des opérandes d'une opération ne suffit pas pour en connaître le *domaine*. Ainsi, dans la division de nombres entiers, le deuxième opérande est de type `nombre entier`, mais la division échoue si la valeur de cet opérande est zéro.

C'est à l'exécution d'un algorithme que se manifestent les contraintes relatives au domaine dans l'écriture d'une opération. Par exemple, les instructions qui suivent induisent une erreur à l'exécution :

```
b := 0
a := 27 / b
```

Il est très difficile de décrire très précisément ce que « fait » réellement une opération. La description peut être donnée :
- mathématiquement, par exemple pour les opérations sur les nombres,
- intuitivement, par une ou plusieurs phrases en langue naturelle,
- à l'aide d'exemples,
- par une combinaison de ces trois méthodes.

Mais les descriptions qui sont données ne sont pas toutes exhaustives. C'est pourquoi on ne peut pas toujours prévoir ce qui sera fait effectivement par la machine dans tous les cas.

De plus, il ne faut pas confondre l'opération théorique (l'addition, par exemple, qui permet d'obtenir des nombres si grands soient-ils) et l'opération réelle, qui fonctionne sur

des variables dont la taille est limitée par les caractéristiques physiques de la machine à laquelle on s'adresse.

5.1.2 Opérateurs

L'addition est notée habituellement à l'aide d'un *opérateur*, l'opérateur "+". De même la soustraction est notée à l'aide de l'opérateur "-". Donnons un exemple d'algorithme utilisant ces opérateurs:

Algorithme 5.1

```
année := 2009
écrire("quel sera ton âge le 31 décembre ?")
lire(âge)
naissance := année - âge
écrire("tu es né en ",naissance)
âge := âge + 1
écrire("le 31 décembre de l'année prochaine, tu auras ",âge," ans!")
```

Une opération permet de former une nouvelle valeur à partir d'autres valeurs. Par exemple, `âge + 1` représente la valeur de la variable de nom `âge` à laquelle est ajoutée la valeur constante `1`.

Cette nouvelle valeur est affectée à la variable `âge`. La variable `âge` a ainsi sa valeur qui est modifiée au cours de chaque exécution de l'algorithme.

Le même opérateur peut servir à désigner plusieurs opérations différentes. Ainsi, l'opérateur "+" désigne à la fois l'addition de nombres entiers et l'addition de nombres réels (et, comme on le verra par la suite, la concaténation de suites de caractères). C'est le type des objets auxquels s'applique l'opérateur qui permettra à la machine de savoir de quelle opération il s'agit.

On considérera également les deux autres opérations habituelles sur les nombres:
- la multiplication, désignée par l'opérateur "*";
- la division que l'on désignera par "/". On décidera dans ce qui suit, par convention, que la division de deux valeurs entières produira un résultat entier (5 divisé par 2 donnant 2), alors que la division d'une valeur entière par une valeur réelle, ou d'une valeur réelle par une valeur entière donnera une valeur réelle. Par exemple 5.0 divisé par 2 aura pour résultat 2.5.

5.1.3 Les expressions

On construit des *expressions*, qui peuvent être complexes, utilisant plusieurs opérateurs, comme par exemple:

```
a + b + c + 3 + d
```

Une valeur, par conséquent, peut être désignée dans un algorithme de trois manières différentes: par une constante, par un nom ou par une expression.

Partout où, dans un algorithme, on attend une valeur, on peut par conséquent avoir une expression. Aux deux dernières instructions exécutables de l'algorithme précédent, on peut ainsi substituer:

```
écrire("l'année prochaine, tu auras ", âge + 1 ," ans!")
```

5.1.4 Notation fonctionnelle

On peut représenter une opération en utilisant, au lieu d'un opérateur, la *notation fonctionnelle*: un nom désigne l'opération, et la liste des opérandes est placée à la suite, entre parenthèses. Dans la liste, les opérandes sont séparés les uns des autres par une virgule.

Toutes les opérations pourraient être écrites à l'aide de la notation fonctionnelle, ainsi

```
a + b * c
```

pourrait aussi bien s'écrire

```
plus(a,multiplie(b,c))
```

mais l'usage des opérateurs est certainement plus commode dans ce cas.

La notation fonctionnelle est néanmoins obligatoire pour les opérations d'arité supérieure à 2.

5.2 Opérations sur les suites de caractères

Il est possible d'accéder à chacun des éléments d'une suite de caractères, par l'intermédiaire d'un *indice*, c'est-à-dire de la place occupée par l'élément dans la suite :

⟨`suite`⟩ `[`⟨`valeur`⟩`]`

La première valeur possible renvoyant à un élément de la séquence, autrement dit le premier *indice* possible, est 0. C'est pourquoi si la valeur affectée à la variable `mot` est `"bonjour"`, `mot[2]` renvoie la valeur `"n"`.

Si on cherche à accéder à un élément de la séquence par un indice qui est en dehors des valeurs possibles, cela produit une erreur.

Plus exactement, l'indice donne une position intermédiaire entre deux éléments. Ce qui permet d'accéder à des sous-suites. Par exemple `mot[2:4]` renvoie la valeur `"nj"`. `mot[0:4]` renvoie `"bonj"`, `mot[0:7]` renvoie `"bonjour"`.

Nous définissons par ailleurs les opérations suivantes :

- l'opération `longueur` est une opération unaire ; elle associe à un objet de type `suite de caractères` un objet de type `nombre entier` : le nombre de caractères dont la suite est constituée. Par exemple, `longueur("bonjour")` a la valeur `7`.

- la *concaténation* est une opération binaire ; elle associe à deux suites de caractères une suite de caractères : la suite de caractères obtenue par la concaténation (mise bout à bout) des deux suites. On notera cette opération à l'aide de l'opérateur "`+`".

Par exemple si `a` a pour valeur `"bon"` et si `b` a pour valeur `"jour"`, on obtient pour `a+b` la valeur `"bonjour"`.

- l'opération `position` associe à deux suites de caractères un nombre entier ;
`position(a,b)` prend la valeur -1 si `b` n'est pas contenue dans `a`, donne la position du premier caractère de `b` dans `a` dans le cas contraire.

Par exemple `position("abcdef", "bc")` prend la valeur `1`, alors que `position("abcdef", "fa")` prend la valeur `-1`.

Remarquons que la notation `a[i]` peut représenter un contenant plutôt qu'une valeur, à gauche d'un signe d'affectation, ou dans une instruction de lecture. Par exemple :

```
a[4]:="x"
```

L'instruction ci-dessus signifie que l'on remplace le cinquième caractère de la suite `a` par le caractère constant `"x"`. Pour que cette instruction ait un sens, il faut que la suite `a` comporte au moins 5 caractères. Une telle instruction ne peut donc servir à l'initialisation de la suite.

Enfin, à propos des suites de caractères, on notera l'existence de la suite de caractères constante *vide*, qui sera notée `""`. Cette suite jouera un rôle particulier dans les différentes opérations. Ainsi, sa longueur sera nulle.

5.3 Type booléen

5.3.1 Définition

Le type logique ou booléen est un type primitif. Les objets de type `booléen` peuvent prendre l'une des deux valeurs : `vrai` ou `faux`.

Quand on examine si une condition est vérifiée ou pas, on effectue en réalité une opération dont le résultat est de type `booléen`. Ainsi, l'égalité entre suites de caractères est une opération qui à deux objets de type `suite de caractères` fait correspondre un objet de type `booléen`. C'est pourquoi, si `a` est une variable de type `booléen` et `b` une variable de type `suite de caractères`, on peut écrire :

```
a:= b = "bonjour".
```

5.3.2 Opérations sur les booléens

Il existe des opérations entre objets de type `booléen`. Ces opérations sont notées à l'aide d'opérateurs.

`non` est une opération qui à un objet de type `booléen` fait correspondre un objet de type `booléen` ; `et` et `ou` sont deux opérations qui à deux objets de type `booléen` font correspondre un objet de type `booléen`.

Les tables de ces opérations sont les suivantes :

non

a	faux	vrai
non a	vrai	faux

et

a \ b	faux	vrai
faux	faux	faux
vrai	faux	vrai

ou

a \ b	faux	vrai
faux	faux	vrai
vrai	vrai	vrai

On peut vérifier, à l'aide de ces tables, que
`non (a ou b)` est équivalent à `(non a) et (non b)` et que
`non (a et b)` est équivalent à `(non a) ou (non b)`.

Nous allons donner un exemple d'algorithme utilisant les booléens, ainsi que les opérations sur les booléens. On pose encore une question à l'utilisateur, question qui comporte deux réponses, ces réponses pouvant être données dans un ordre quelconque.

Algorithme 5.2

```
écrire ("Qui sont les jumeaux qui ont fondé Rome?")
écrire ("Premier jumeau:")
lire (réponse)
r1 := réponse = "Remus"
r2 := réponse = "Romulus"
si r1 ou r2 alors
     début
     écrire ("Deuxième jumeau:")
     lire (réponse)
     r1 := r1 ou (réponse = "Remus")
     r2 := r2 ou (réponse = "Romulus")
     fin
si r1 et r2 alors écrire("c'est bon")
     sinon écrire ("ce n'est pas la bonne réponse")
```

5.4 Priorités

5.4.1 Motivation

L'utilisation des opérateurs peut introduire une ambiguïté dans l'ordre d'exécution des opérations[12]. Par exemple, pour

```
a + b * c
```

on ne sait s'il faut d'abord effectuer l'addition ou la multiplication.

12. La notation fonctionnelle exclut, quant à elle, tout risque d'ambiguïté.

C'est pourquoi on donne un ordre de priorité entre opérateurs. Ainsi, pour nous, la multiplication sera prioritaire par rapport à l'addition. Cela signifie que la multiplication sera effectuée avant l'addition.

Par conséquent si, dans l'exemple précédent, on veut effectuer l'addition avant la multiplication, on utilise des *parenthèses* :

```
(a + b) * c
```

5.4.2 Règles de priorité

Les règles de priorité que l'on se donne ici sont les suivantes:

- les opérateurs `*` et `/` sont prioritaires par rapport aux opérateurs `+` et `-` ;

- les opérateurs arithmétiques sont prioritaires par rapport aux opérateurs de comparaison et aux opérateurs logiques ;

- les opérateurs de comparaison sont prioritaires par rapport aux opérateurs logiques ;

- l'opérateur `non` est prioritaire par rapport aux opérateurs `ou` et `et` ;

- l'opérateur `et` est prioritaire par rapport à l'opérateur `ou`.

5.4.3 Ordre d'exécution des opérations

Il faut savoir dans quel ordre seront effectuées les opérations en cas de priorités égales ou de l'utilisation multiple d'un même opérateur. Pour nous, ce sera de la gauche vers la droite.

C'est pourquoi l'expression numérique

```
8 / 4 / 2
```

prend la valeur 1 et pas la valeur 4.

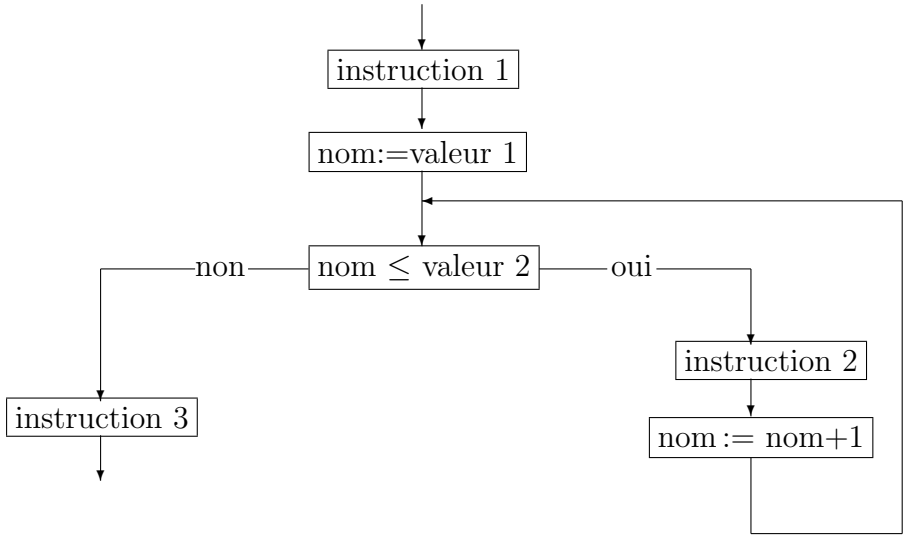
6 Algorithmes itératifs

6.1 L'instruction `pour`

Cette instruction a la forme générale suivante :

`pour` ⟨`nom`⟩ `:=` ⟨`valeur 1`⟩ `à` ⟨`valeur 2`⟩ `faire` ⟨`instruction`⟩

Dans un contexte donné, on peut représenter graphiquement l'instruction `pour` par :



Le contexte serait celui-ci :

⟨`instruction 1`⟩
`pour` ⟨`nom`⟩ `:=` ⟨`valeur 1`⟩ `à` ⟨`valeur 2`⟩ `faire` ⟨`instruction 2`⟩
⟨`instruction 3`⟩

L'instruction *pour* permet d'*itérer* une instruction, c'est-à-dire de la répéter un certain nombre de fois (ici connu à l'avance).

Nous allons donner un exemple d'utilisation de l'instruction `pour` : la recherche du nombre d'occurrences de la lettre « e » dans une phrase.

Algorithme 6.1

```
écrire ("Donner une phrase:")
```

```
lire(phrase)
k := 0
lg := longueur(phrase)
pour i := 0 à lg-1 faire
     si phrase[i] = "e" alors k := k + 1
écrire ("il y a ",k," occurrences de e dans la phrase")
```

Dans cet algorithme, on utilise un *compteur*, `k`, c'est-à-dire une variable entière qui est initialisée (ici, à `0`), puis incrémentée (par l'instruction `k := k+1`) au fur et à mesure que c'est nécessaire.

6.2 Applications

6.2.1 Calcul d'une somme

Le calcul d'une somme d'un certain nombre de quantités s'effectue en généralisant ce qui a été fait pour les compteurs.

On initialise une variable (par exemple de nom `somme`), en général à la valeur 0. Puis, on ajoute à cette variable les différentes valeurs qu'il faut lui ajouter, à l'aide d'une instruction `tant que` ou d'une instruction `pour`.

Considérons par exemple le problème suivant : un enseignant va communiquer à la machine toutes les notes obtenues par ses 30 élèves. Il veut connaître la moyenne de ces notes.

Algorithme 6.2

```
somme := 0
pour i := 1 à 30 faire
     début
     écrire ("Donner la note numéro ", i)
     lire(note)
     somme := somme + note
     fin
écrire ("la moyenne est : ",somme/30)
```

6.2.2 Calcul d'un maximum ou d'un minimum

Soit maintenant le problème suivant: le professeur entre le nom de ses 30 élèves (dans un ordre quelconque). La machine lui renvoie le nom du premier élève dans l'ordre alphabétique.

Algorithme 6.3

```
premier := "zzzz" {Initialisation du minimum}
```

```
pour i := 1 à 30 faire
     début
     écrire ("Donner le nom de l'élève numéro ", i)
     lire(élève)
     si élève < premier {test}
          alors premier := élève {changement du premier provisoire}
     fin
écrire ("le premier par ordre alphabétique est : ",premier)
```

On a ici résolu un problème de recherche d'une valeur minimum. Le problème de la recherche d'une valeur maximum serait résolu d'une manière similaire. Dans les deux cas, on effectue des comparaisons entre valeurs prises deux par deux.

On utilise une variable qui contient à un moment donné le minimum (ou le maximum) provisoire (ici `premier`). On initialise convenablement cette variable (selon le problème). Puis on compare chacun des candidats possibles à être le minimum ou le maximum à la valeur du minimum ou du maximum provisoire. Si nécessaire, on modifie cette valeur provisoire.

En fin d'itération, cette valeur sera le minimum ou le maximum définitif, c'est-à-dire le résultat de l'algorithme.

6.3 L'instruction `tant que`

Supposons que l'on envisage le problème d'enseignement assisté en posant la question à l'élève, jusqu'à ce qu'il donne la bonne réponse.

Cela va s'effectuer de la manière suivante :

Algorithme 6.4

```
réponse := ""
écrire ("Quelle est la capitale de la France?")
tant que réponse ≠ "Paris" faire
     début
     lire (réponse)
     si réponse ≠ "Paris" alors
          écrire ("ce n'est pas la bonne réponse, recommencez")
     fin
écrire ("c'est cela")
```

Dans l'algorithme qui précède, on utilise une instruction `tant que`. Cette instruction a la forme générale suivante :

`tant que` ⟨test⟩ `faire` ⟨instruction⟩

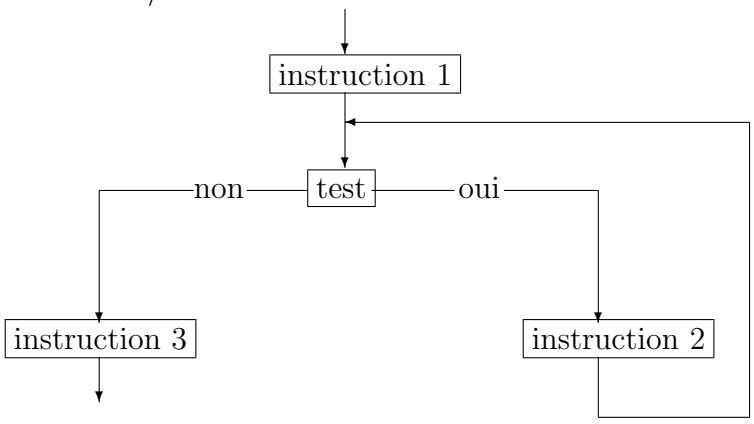
Elle permet d'*itérer* une instruction, c'est-à-dire ici de la répéter un certain nombre de fois non connu à l'avance[13].

C'est le test qui permet de savoir jusqu'à quand on répète l'instruction.

Il est clair que l'instruction qui est itérée, peut être remplacée par un bloc d'instructions[14] (ce qui est le cas dans le présent exemple).

Nous explicitons ce que fait l'instruction `tant que`, dans un contexte donné, de manière graphique:

⟨`instruction 1`⟩
`tant que` ⟨`test`⟩ `faire` ⟨`instruction 2`⟩
⟨`instruction 3`⟩



Donnons un autre exemple qui utilise cette instruction : un algorithme qui compte le nombre de mots d'une phrase. On suppose que les mots de la phrase sont séparés par des espaces uniques et qu'il n'y a d'espace ni en début ni en fin de phrase.

Algorithme 6.5

```
écrire ("Donner une phrase:")
lire(phrase)
k := 1 {Initialisation de k}
p := position(phrase," ")
tant que p ≥ 0 faire
     début
     k := k + 1 {Incrémentation de k}
     phrase := phrase[p+1:longueur(phrase)]
     p := position(phrase," ")
     fin
écrire ("il y a ",k," mots dans la phrase")
```

13. Dans le cas présent, on n'est pas certain que l'algorithme s'arrêtera un jour.
14. Voir ci-dessus page 25.

L'instruction `tant que` est plus générale que l'instruction `pour`. Toutefois, quand la machine peut savoir, au moment de commencer à exécuter une itération, combien de fois elle répétera son instruction ou son bloc d'instructions, il est préférable d'utiliser l'instruction `pour`.

6.3.1 Instructions itératives imbriquées

Il est possible que, parmi les instructions qui sont à itérer, figure une instruction d'itération (qui elle-même peut contenir une instruction d'itération, etc.).

On va donner un exemple d'imbrication d'instructions itératives : le jeu du « pendu ». La machine « pense » à un mot, et l'utilisateur doit essayer de le deviner en proposant successivement des lettres. S'il propose plus de dix lettres erronées, il a perdu. Au fur et à mesure de ses propositions, la machine affiche les lettres qu'il a trouvées dans le mot à leur bonne position.

Algorithme 6.6

```
a := "anticonstitutionnellement"
l := longueur(a)
b := ""
pour i := 1 à l faire b := b + "_"
écrire(b)
k := 0
tant que (a ≠ b) et (k ≤ 10) faire
     début
     écrire ("Proposez une lettre")
     lire(lettre)
     bon := faux
     pour i := 0 à l-1 faire
          si a[i] = lettre alors
               début
               bon := vrai
               b[i] := lettre
               fin
     si bon alors écrire(b)
          sinon k := k+1
     fin
si k > 10 alors écrire ("vous êtes pendu")
     sinon écrire("vous avez trouvé")
```

6.4 Les listes

Les listes permettent de traiter des ensembles de données. On peut avoir des listes de nombres, des listes de suites de caractères, voire des listes de listes.

Par exemple `["le", "chien", "mange", "une", "pomme"]` est une liste de suites de caractères.

Les opérations sur les listes sont les mêmes que celles sur les suites de caractères, sauf l'opération *position*.

Ainsi, il est possible d'accéder à chacun des éléments d'une liste par l'intermédiaire d'un *indice*. Par exemple si la valeur affectée à la variable `phrase` est `["le", "chien", "mange", "une", "pomme"]`, `phrase[2]` renvoie la valeur `"mange"`.

Si on cherche à accéder à un élément de la séquence par un indice qui est en dehors des valeurs possibles, cela produit une erreur.

Ici encore, l'indice donne une position intermédiaire entre deux éléments. Ce qui permet d'accéder à des sous-suites. Par exemple `phrase[2:4]` renvoie la valeur `[ "mange", "une"]`. `phrase[0:4]` renvoie `["le", "chien", "mange", "une"]`, `phrase[0:5]` renvoie `["le", "chien", "mange", "une", "pomme"]`.

Nous définissons également les opérations suivantes :

- l'opération `longueur` est une opération unaire ; elle associe à une liste un nombre entier : le nombre d'éléments dont la liste est constituée. Par exemple, `longueur(["le", "chien", "mange", "une", "pomme"])` a la valeur `5`.

- la *concaténation* est une opération binaire ; elle associe à deux listes une nouvelle liste : la liste obtenue par la concaténation (mise bout à bout) des deux listes. On notera cette opération à l'aide de l'opérateur "+".

Par exemple si `la` a pour valeur `["le", "chien", "mange"]` et si `lb` a pour valeur `[ "une", "pomme"]`, on obtient pour `la+lb` la valeur `["le", "chien", "mange", "une", "pomme"]`.

Remarquons que la notation `la[i]` peut représenter un contenant plutôt qu'une valeur, à gauche d'un signe d'affectation, ou dans une instruction de lecture. Par exemple :

```
la[4]:="poire"
```

L'instruction ci-dessus signifie que l'on remplace le cinquième élément de la liste `a` par la suite de caractères `"poire"`. Pour que cette instruction ait un sens, il faut que la liste `la` comporte au moins 5 éléments. Une telle instruction ne peut donc servir à l'initialisation de la liste.

On se donne ici également l'opération *appartient*, que l'on peut noter à l'aide de la notation mathématique $\in$. On vérifie si un élément appartient ou pas à une liste.

Enfin, la liste *vide* sera notée `[ ]`. Cette liste a une longueur nulle.