La hiérarchie de Noam Chomsky

Noam Chomsky s'est attaché à classifier les différents systèmes informatiques. La seule solution pour réaliser ce travail est évidemment de les classer selon leur structure interne, qui correspond aussi à leur puissance de traitement. Il a placé, au sommet de cette hiérarchie, la machine universelle de Turing (qui peut effectuer tout traitement logique), et lui a affecté le coefficient 0. Ensuite, quand les machines ou systèmes décrits baissent en puissance, Chomsky incrémente l'index de leur type.

type	***Classe du langage***	***Restriction sur les règles***	***Reconnu par :***	***Exemple / Commentaires***
type 0	Récursivement dénombrable	Aucune restriction	Machine de Turing	Langage naturel. Réseau de transition augmenté. Contexte global : les actionneurs agissent sur le monde.(effets de bord)
type 1	Grammaire contextuelle Sensible au contexte (qui transporte un contexte)	Aucune règle limitant la longueur	Automate linéairement borné	Maintient d'un contexte local. Langage de programmation structurée (pure).
type 2	Grammaire hors contexte Règles libres de tout contexte (ne transporte pas de contexte)	Le côté gauche doit être un unique non terminal (il doit être sans contexte) Le côté droit doit être une string de symboles terminaux ou non $A \rightarrow \gamma$	Automate à piles	Réseau de transition récursif Les règles peuvent expanser un symbole en N autres, ce qui équivaut à un appel de sous-programme, une expansion de but en N sous-buts
type 3	Langage régulier Expressions régulières Grammaires régulières	Le côté gauche doit être un unique non terminal Le côté droit doit être : - Soit un unique terminal - Soit un unique terminal suivi ou précédé d'un unique non terminal $A \rightarrow a$ $A \rightarrow aB$ $A \rightarrow Ba$ $A \rightarrow \phi$	Automate à états finis	Réseau de transition Les règles ne peuvent pas expanser un symbole en plusieurs autres.

Systèmes de règles de production : lemmes et pré-requis

Conventions de notation :

- a, b, c... Un symbole en minuscule est un symbole terminal. (qui appartient à l'ensemble des symboles terminaux)
- A, B, C.. Un symbole en majuscule est un symbole non-terminal (n'appartient pas à cet ensemble de symboles terminaux)
- α, β, γ... Un symbole grec est un symbole quelconque, non contraint, il peut être terminal ou non.

Attribution d'un type selon la hiérarchie de Chomsky

Le type d'un langage correspond à celui de la grammaire la moins puissante qui le reconnaît.
De même, un langage est du type le plus restreint de la grammaire qui est adéquate pour le décrire.

Pré-requis : définition de la notion de grammaire

Si on utilise un système de règles de production pour faire de l'analyse syntaxique du langage naturel, on obtient des règles de grammaire, ou tout simplement une grammaire.

Les séquenceurs (de type 4)

Les séquenceurs sont si simples et si peu puissants qu'ils n'entrent même pas dans la classification de Chomsky. Disons alors rapidement qu'ils sont de type 4 (Mais c'est pour simplifier l'exposé).

Le passage depuis un séquenceur vers un système de règles de production est très simple : il suffit de transformer la table en règle. Nous avons implicitement utilisé cette technique dans le cadre de ce cours.

Ainsi, par exemple, à droite, la table de 4 adresses se traduit par les 4 règles suivantes :
Paris → Chartres
Chartres → Le_Mans
Le_Mans → Angers
Angers → Nantes

Les grammaires formelles de type 3

Déjà vu

Dans le cadre de ce cours, nous avons abondamment traité les grammaires formelles de type 3 au moyen des automates finis.

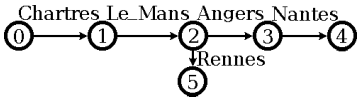
Automates à état finis (de type 3 dans la hiérarchie de Chomsky)

Introduction

La structure d'automate à états finis introduit la bifurcation dans un graphe. Dans un ou plusieurs nœuds du graphe, on se retrouve devant une patte d'oie. Il faut choisir entre une branche ou l'autre. Cette structure correspond un peu à l'alternative, au SAS (Si Alors Sinon) de l'informatique.

Reprenons l'exemple précédent pour le compléter : arrivé au Mans, il nous faut maintenant choisir entre Angers et Rennes. Pour décrire cette nouvelle possibilité, depuis les 4 règles précédentes, il faut passer par deux étapes :

villes\états	***0***	***1***	***2***	***3***	***4***	***5***
Paris						
Chartres	1					
Le_Mans		2				
Angers			3			
Nantes				4		
Rennes			5			



Utiliser une tête de règle à deux paramètres *état TêteDuRuban*

Ainsi les 4 règles précédentes prennent la forme :
État TêteDuRuban → État et elles deviennent :
0 Chartres → 1
1 Le_Mans → 2
2 Angers → 3
3 Nantes → 4

Au Mans, il suffit de rajouter la bifurcation vers Rennes

Pour cela il suffit d'ajouter la règle suivante :
2 Rennes → 5

Systèmes de règles de production : grammaires formelles de type 2 (i.e. hors contexte)

Systèmes de règles de production : un jeu de règles de grammaire hors contexte (de type 2)

Définition de la notion de grammaire dans le domaine informatique

Si on utilise un système de règles de production pour analyser syntaxiquement le langage naturel, on obtient des règles de grammaire, ou tout simplement une grammaire.

Définition de la notion de G.H.C. - grammaire hors contexte (context free grammar)

Ce type de grammaire est libre de contexte

Elle n'utilise pas de contexte. Ainsi, lors de l'analyse terme à terme de la communication échangée (groupe de mots, phrase, discours ou récit), on n'utilise pas de variable qui serait ensuite identifiée, instanciée et mémorisée. L'échange ne transporte pas d'environnement, pas de contexte. On n'y trouve pas de déictique, de pronom. Le genre et le nombre des entités (matières, l'objet, outils ou agents) ne sont pas vérifiés au moyen d'une propagation de contexte.

Cependant, dès le paragraphe suivant, nous verrons comment contourner cette limitation afin d'effectuer ces traitements autrement.

Étude d'un exemple : vérifier que le langage est bien capable de générer un groupe nominal

La grammaire formelle

Les pièces

On dispose des symboles de types suivants :
'GN', abréviation de : 'Groupe Nominal',
'ART', abréviation de : 'Article',
'ADJ', abréviation de : 'Adjectif'
'NOM'.
Et enfin on dispose de symboles de constantes : le, un, la, beau, petit, chat et chien.

État de départ

On peut supposer que sur le ruban on trouve le symbole 'GN'.

Les coups autorisés

	// Les 3 règles centrales décrivant des syntaxes du GN.
GN → ART NOM	// Expansion de but en sous-but : selon Chomsky, on est au moins au type 2
GN → ART ADJ NOM	
GN → ART ADJ NOM ADJ	
ART → le | un | la	// Les constantes et leur type
ADJ → beau | petit	
NOM → chat | chien	

Conventions de notation

Il faut lire le symbole *|* *comme ou bien*. Cette notation permet de condenser les règles dont la prémisse (tête de règle) est la même. Ainsi la règle *NOM → chat | chien* provient de la simplification par factorisation des deux règles suivantes : NOM → chat ; NOM → chien.

Les conditions d'arrêt

De façon classique, on arrête si le symbole du départ 'GN' a été remplacé par des constantes terminales (écrites en minuscules).

Exemple d'une session

État de départ :

GN // Il faut démontrer que cette grammaire, muni de son vocabulaire peut construire un groupe nominal.

On obtient successivement les états suivants :

GN // Le but à démontrer. Maintenant appliquons la règle : GN → ART ADJ NOM
ART ADJ NOM // maintenant appliquons la règle : ART → le
le ADJ NOM // maintenant appliquons la règle : ADJ → beau
le beau NOM // maintenant appliquons la règle : NOM → chat
le beau chat // est un état final car constitué de symboles terminaux.

Arrivée : *le beau chat*

Notre grammaire a donc bien généré le groupe nominal *le beau chat*. Si l'on avait permuté l'ordre des règles, elle aurait encore pu générer le groupe nominal : *le petit chien*, mais, par exemple *le grand chien* n'aurait pas pu être généré.

Unification d'un énoncé général avec un énoncé restreint

Affectation d'une variable au moyen d'une valeur

Dans l'exemple précédent, nous avons remplacé la variable typée générale *ADJ* par une instance (un exemple) d'adjectif : *beau*. Dans ce cas, on affecte à la variable une valeur, on lui donne un contenu. On a affecté la variable *ADJ* au moyen de la constante *beau*. On obtient donc la liaison (variable – valeur) : (ADJ - beau).

Notion de variable typée

Par exemple, le symbole *ADJ* est une variable car il peut recevoir une valeur. Mais en fait son domaine d'affectation est réduit : il ne peut être affecté que par un symbole d'un certain type, en l'occurrence ici, que par un adjectif (que par un symbole appartenant à la classe adjectif). Donc, en fait, il constitue une variable typée.

Il faut cependant préciser que nous sommes dans une GHC, qui ne mémorise par les instanciations. Et maintenant, si on nous pose la question : quel est *ADJ* (quel est l'adjectif), aucun mécanisme du système est donc capable de répondre : *C'est : beau*.

Instance et instanciation : ça s'en va et ça revient

Dans le domaine informatique, les affectations ne sont pas toujours définitives. Dans le cas du traitement avec le 'backtrack', on remet en cause des choix effectués précédemment. Par exemple, la grammaire que nous utilisons comporte la règle : *ART → le | un | la*.

Ainsi la variable *ADJ* peut être affectée par différentes valeurs. En conséquence, on appelle instanciation, l'affectation dans l'instant, i.e. provisoire d'une variable par une valeur. Cette appellation souligne qu'elle est moins solide, i.e. que sa liaison peut être cassée.

Note : une illustration pertinente de la fluctuation des instanciations est clairement donnée par l'évaluation de fonctions récursives, par exemple fact(3). Voir plus loin dans le poly, l'illustration de l'évaluation de factorielle de 3, dans le paragraphe de transition, depuis les grammaires de type 2 vers celles de type 1.

Unifier deux chaînes, c'est trouver les remplacements qui les rendent égales

Mais, dans le traitement de l'exemple précédent, pour ces trois variables *ART ADJ NOM* qui était contraintes, on a tout de même trouvé trois valeurs correspondantes (les trois constantes : *le beau chat*) qui lui conviennent. Finalement, dans le cadre de ces trois instanciations, après substitution, on a rendu le but *ART ADJ NOM* égal à la chaîne *le beau chat*. On a donc unifié les deux chaînes.

Une grammaire hors contexte pour traiter le genre dans un groupe nominal

Dans une GHC, comment transporter une information au sein d'une règle

Théoriquement, une grammaire hors contexte ne transporte pas de donnée locale, cependant le problème est parfois de transporter quelques informations contextuelles. Alors voyons comment cette mémorisation peut se faire implicitement dans la règle en cours de démonstration.

Ainsi, en traitement du langage naturel, avec une GHC, si le programmeur veut traiter le genre et le nombre, i.e. faire des vérifications de genre et de nombre sur les sujets et objets de la phrase, il doit transporter l'information dans la règle en cours, i.e. grâce aux marqueurs dans les règles. Mais ceci implique que chaque règle soit dédiée à un cas particulier : une règle pour le masculin, une règle pour le féminin. Chaque règle transporte implicitement des informations : si, dans l'application d'une règle, je viens de vérifier que le genre était masculin, et si je suis toujours en train de la traiter, alors, implicitement jusqu'à maintenant, elle n'a pas été mise en échec, et le genre est effectivement masculin.

Ensuite, si, à la grammaire, on veut rajouter un traitement du nombre, comme il faut une règle dédiée par cas, il faut doubler le nombre précédent de règles. Illustrons cela au travers d'exemples concrets.

La grammaire

// 2 règles hors contexte, expansant un symbole en plusieurs autres.
GN → ArtMasc NomMasc // Une règle dédiée à un GN masculin

```
GN → ArtFemi NomFemi        // Une règle dédiée à un GN féminin
                            // Puis maintenant apparaissent les symboles terminaux
ArtMasc → le                // Ils instancient des types grammaticaux (des classes de mots du langage)
ArtFemi → une
NomFemi → chatte
NonMasc → chien
```

Exemple de séquence de dérivation qui génère la chaîne *un chien* appartenant au langage

```
                            // Le but est de démonter que cette grammaire peut générer un groupe nominal.
GN                          // On commence avec le but à démontrer (selon un processus en chaînage arrière).
ArtMasc NomMasc             // Le but à démontrer est expansé en deux sous-buts (restrictifs car on teste 'masculin').
le NomMasc                  // Obtenu après application de la règle ArtMasc → le
                            // Attention, implicitement, si le programme arrive jusqu'ici, c'est que l'article est masculin.
le chien                    // Obtenu après application de la règle NonMasc → chien
```

Grammaire hors contexte (type 2) pour traiter le genre et le nombre dans un groupe nominal

C'est un exemple un peu plus complexe, qui traite le genre et le nombre. Alors on doit donc écrire 4 règles centrales.

La grammaire

```
                                    // 4 règles hors contexte, expansant un symbole en plusieurs autres.
GN → ArtMascSing NomMascSing        // Une règle dédiée à un GN masculin et singulier
GN → ArtFemiSing NomFemiSing        // Une règle dédiée à un GN féminin et singulier
GN → ArtMascPlur NomMascPlur        // Une règle dédiée à un GN masculin et pluriel
GN → ArtFemiPlur NomFemiPlur        // Une règle dédiée à un GN féminin et pluriel
                                    // Puis maintenant apparaissent les symboles terminaux

ArtMascSing → le
ArtFemiSing → la
NomFemiSing → chatte
NonMascSing → chien
ArtMascPlur → les
ArtFemiPlur → les
NomFemiPlur → chattes
NonMascPlur → chiens
```

Exemple de séquence de dérivation générant la chaîne *un chien* qui appartient au langage.

```
                            // Le but est de démonter que cette grammaire peut générer un groupe nominal.
GN                          // On commence avec le but à démonter (selon un processus en chaînage arrière).
ArtMascSing NomMascSing     // Expansion du but en 2 sous-buts (restrictifs car on teste 'masculin' et 'singulier').
le  NomMascSing             // Obtenu après application de la règle ArtMascSing → le
le chien                    // Obtenu après application de la règle NonMascSing → chien
```

Inconvénients de la grammaire hors contexte :

Les GHC ne peuvent pas transporter de contexte où on mémoriserait les précédentes instanciations. Bien sûr on peut contourner cette difficulté en le transportant implicitement au sein des règles elles-mêmes, mais cela alourdit l'écriture des règles et l'exécution du traitement.

Regardons en détail comment notre tâche se complique quand on travaille sans transporter de contexte. En effet, si on veut traiter le genre, on doit écrire deux règles (masculin et féminin), trois si on a un neutre. Cela double ou triple le nombre de règles. Mais si en plus, on veut traiter le nombre, on doit écrire deux jeux de règles (singulier pluriel), cela double le nombre de règles principales. En effet, si on veut traiter à la fois le genre et le nombre, on doit donc écrire 2x2 (ou 2x3) règles.

Mais si on voulait encore en plus filtrer un troisième critère, on doublerait à nouveau le nombre de règles. En conclusion, pour un traitement un peu complexe, l'utilisation des GHC devient rapidement rédhibitoire !

Les grammaires formelles hors contexte (de type 2) peuvent traiter la récursivité

La récursivité

Importance de la récursivité

Nous avons effleuré la récursivité à plusieurs occasions. Son introduction dans les systèmes informatiques ou mathématiques est importante, car elle engendre élégance, souplesse et universalité.

Rapide introduction à la notion de récursivité :

Le nain et la fée : un exemple sympathique pour présenter la notion de récursivité de façon imagée.

Le nain vient de rendre un grand service à la fée. Pour le remercier, elle lui promet : énonce-moi deux vœux, et je te les accorderai. Alors le nain, qui est un peu matheux, formule :
- Comme premier vœu, je voudrais un bel habit brodé d'or !
- Et comme second vœu, accorde-m'en, s'il te plaît, deux autres !
Évidemment, au moyen de ce subterfuge, le nain put obtenir de la fée une infinité de vœux.

Comment détecter une construction présentant une structure récursive ?

Quand le nain formule son désir à la fée, dans la définition du vœu, il utilise le terme vœu, ainsi sa définition se mord la queue, et la circularité de cet énoncé provient de cet artifice.

Remarquez la même chose dans la définition de factorielle

fact N = N × fact (N–1). Il faut bien voir que la définition de la fonction factorielle est basée sur la même démarche : dans la partie droite de la formulation, on fait référence à la fonction factorielle elle-même.

Définition récursive du parcours d'un arbre binaire

Pour parcourir un arbre binaire, il faut : s'il existe, parcourir son tronc vers le bas, puis parcourir sa branche de gauche, ensuite sa branche de droite, et enfin, s'il existe, remonter son tronc. Ici la récursivité est plus difficile à voir, mais il faut se souvenir, qu'un arbre c'est aussi une branche. On voit donc bien que pour définir le parcours d'un arbre, on évoque *parcourir une branche*, i.e. *parcourir un arbre*. La définition est bien récursive.

Le langage naturel est récursif

La prose que nous utilisons quotidiennement peut être récursive. En voici quelques illustrations :

Emboîtage de relatives

Au sein d'une subordonnée relative on peut trouver une autre relative. Voici quelques exemples pour illustrer cet aspect :
The house, the man, the hat … is funny, is kind, is large.
Le petit chat boit le bon lait de la bouteille qui est à côté des grandes herbes qui ondulent.

Les compléments de noms

Au sein d'un complément de nom, on peut trouver plusieurs fois un autre complément de nom. Voici un exemple[1], certes un peu excessif, mais qui illustre bien cet aspect :

Le bout du poil du bout de la queue de la puce du bout du poil du bout de la queue du chien de la fille de Madame Gaspard s'en va à Montréal.

Groupe nominal

Dans un groupe nominal, un nom peut être introduit par plusieurs adjectifs : *le grand beau jeune homme*. Dans ce cas, la grammaire qui reconnaît ce groupe, peut comporter cette règle récursive :

RES_GN → ADJ RES_GN // Cette règle est récursive, car pour définir un *reste de GN*, on évoque un *reste de GN*.

Les grammaires de type 2 sont reconnues par les automates à pile

Selon la classification de Noam Chomsky, les grammaires formelles de types 2 sont reconnues par les automates à piles qui sont capables de traiter des R T R (réseaux de transition récursifs). Elles permettent donc de traiter des domaines récursifs.

Étude d'un exemple récursif traité au moyen d'une grammaire hors contexte

Une grammaire formelle récursive avec des variables

- État de départ : GN

- Les conditions d'arrêt :

On arrête si le symbole du départ 'GN' a été entièrement remplacé par des symboles terminaux.

- Les coups autorisés
GN → ART RES_GN // Deux règles expansives
RES_GN → ADJ RES_GN

1 Cet exemple est tiré du folklore populaire français.

RES_GN → NOM
ART → le // Les symboles terminaux
ART → un
ART → la
ADJ → beau
ADJ → petit
ADJ → gentil
NOM → chat
NOM → chien

Exemple d'une session

On obtient successivement les états suivants :

GN // D'abord on commence à expanser le but en deux sous-buts avec : GN → ART RES_GN
ART RES_GN // Déjà on fait apparaître l'article : ART → le
le RES_GN // Ici, c'est le premier rappel récursif au moyen de la règle : RES_GN → ADJ RES_GN
le ADJ RES_GN // On peut déjà faire apparaître un premier adjectif avec la règle : ADJ → gentil
le gentil RES_GN // Ici, c'est un deuxième rappel récursif au moyen de la règle : RES_GN → ADJ RES_GN
le gentil ADJ RES_GN // On peut ensuite faire apparaître un deuxième adjectif avec la règle : ADJ → beau
le gentil beau RES_GN // Maintenant appliquons la règle : RES_GN → ADJ RES_GN
le gentil beau ADJ RES_GN // On peut ensuite faire apparaître le dernier adjectif avec : *ADJ → petit*
le gentil beau petit RES_GN // Maintenant appliquons la règle : RES_GN → NOM
le gentil beau petit NOM // Maintenant appliquons la règle : NOM → chat
le gentil beau petit chat // Et c'est fini

La récursivité de cette grammaire provient de la règle :

RES_GN → ADJ RES_GN

Dans sa partie gauche, cette règle annonce 'RES_GN' : elle s'attache à définir le reste d'un groupe nominal, et, dans sa partie droite, elle définit de quoi est fait un reste de groupe nominal, mais pour ce faire, elle évoque encore le 'RES_GN'.

En épistémologie, si pour définir un mot nouveau, on évoque à nouveau ce mot dans la définition, cela pose un problème de cohérence, car on tourne en rond et on dit que la définition est circulaire.

En informatique ou en mathématiques, si pour définir une fonction on évoque encore cette fonction, on dit que la définition est récursive.

Un exemple plus conséquent du langage naturel traité en chaînage arrière

La grammaire à utiliser

phrase → gr_nominal_étendu reste_de_phrase // La règle centrale pour construire une phrase
gr_nominal_étendu → gn relative // La règle pour construire un groupe nominal étendu
gn → article adjectif nom // Deux règles pour créer un groupe nominal
gn → article nom
reste_de_phrase → verbe_transitif adverbe gr_nominal_étendu
reste_de_phrase → verbe_intransitif adverbe
adverbe → lentement
adverbe → nil
relative → qui reste_de_phrase
relative → nil
article → le
nom → chat | lait
verbe_transitif → boit
verbe_intransitif → nourrit | grandit

Grammaire récursive

Cette grammaire est récursive, mais la récursivité est cachée profondément. Elle passe par le chemin suivant :
gr_nominal_étendu → gn relative
relative → qui reste_de_phrase
reste_de_phrase → verbe_transitif adverbe gr_nominal_étendu

Générer la séquence :

Avec cette grammaire, peut-on générer la séquence : *le petit chat qui boit le lait qui nourrit grandit lentement* ?

Résumé, un peu raccourci, de ce traitement

```
phrase
gr_nominal_étendu                                                                                    reste_de_phrase
gn          relative                                                                                 reste_de_phrase
:           qui       reste_de_phrase                                                                reste_de_phrase
:           :         verbe_transitif   adverbe   gr_nominal_étendu                                   reste_de_phrase
:           :         :                 :         gn          relative                               reste_de_phrase
:           :         :                 :         :           qui       reste_de_phrase              reste_de_phrase
:           :         :                 :         :           :         verbe_intransitif  adverbe   reste_de_phrase
:           :         :                 :         :           :         :                  :         verbe_intransitif  adverbe
:           :         :                 :         :           :         :                  :         grandit            lentement
:           :         :                 :         :           qui       nourrit            nil
:           :         :                 :         le lait
:           :         boit              nil
:           qui
:
article     adjectif  nom
le          petit     chat
```

Un exemple plus conséquent du langage naturel traité en chaînage avant (analyse montante)

Introduction

Dans ce court exemple, il s'agit d'évoquer une autre stratégie d'application des règles : le chaînage avant, qui permet ici une analyse montante. Cependant, précisons que, lors du traitement de cet exemple, notre but sera d'illustrer son fonctionnement, tout en prenant soin de ne pas rentrer dans les détails théoriques ou d'application.

Reconnaître :

Supposons qu'on dispose d'un système de production capable d'appliquer le jeu de règles décrit plus loin. Le but est démontrer en chaînage avant que la séquence : *le petit chat de la belle chatte de la maison boit le bon lait* est bien une phrase.

Un arbre inversé

Soit le dessin de l'arbre syntaxique de droite. Conventionnellement, on le représente inversé : sa racine est en haut et les branches tombent. On convient de dire que le mot 'phrase' est posé à son sommet, et que les feuilles pendent à sa base.

Expliquons la terminologie : *Analyse montante*

Ainsi, une analyse montante part de la séquence *le petit chat de la belle chatte de la maison boit le bon lait* constituée de feuilles, et remonte dans l'arbre pour conclure, à son sommet, qu'elle constitue une phrase.

La grammaire à appliquer :

Notez que pour faire de l'analyse montante, l'ordre de chaque règle est changé. Alors dans certaines règles, il existe des têtes de règles comportant plusieurs prémisses. Implicitement, entre chacune de ces conditions, il existe un connecteur logique ET. En effet, pour que la règle se déclenche, il faut que toutes les prémisses/conditions soient satisfaites.

Notez aussi que, volontairement, le fonctionnement interne du système de production capable d'appliquer ces règles n'est pas précisé.

ARTICLE ADJECTIF NOM → GN

GN de GN → GN

GN VERBE_TRANSITIF GN → PHRASE

le → ARTICLE
la → ARTICLE
petit → ADJECTIF
belle → ADJECTIF
bon → ADJECTIF
chat → NOM
chatte → NOM
maison → NOM
lait → NOM

Voici le résumé d'une session

(convention de notation : quand une séquence à été abrégée elle est matérialisée par des pointillés à la fin de la ligne)
le petit chat de la belle chatte de la grande maison boit le bon lait
ARTICLE petit chat de la belle chatte de la grande maison boit le bon lait
ARTICLE ADJECTIF chat de la belle chatte de la grande maison boit le bon lait
ARTICLE ADJECTIF NOM de la belle chatte de la grande maison boit le bon lait
GN de la belle chatte de la grande maison boit le bon lait...
GN de ARTICLE ADJECTIF NOM de la grande maison boit le bon lait...
GN de GN de la grande maison boit le bon lait
GN de la grande maison boit le bon lait...
GN de GN boit le bon lait
GN boit le bon lait
GN VERBE_TRANSITIF le bon lait...
GN VERBE_TRANSITIF GN
PHRASE

Transition depuis les grammaires de type 2 jusqu'aux grammaires de type 1

Pré-requis : retour sur la notion de contexte dans le cas d'une grammaire contextuelle

Dans le cadre d'une *communication langagière* entre deux individus, le locuteur, pour raccourcir son message, ne répète pas certaines informations déjà dites et inférables, il se contente de les évoquer rapidement. L'auditeur, qui traite cette information (groupe de mots, phrase, discours ou récit), les identifie et les mémorise : au fil de l'échange, un contexte se crée, composé de déictiques, de pronoms ou de genres et de nombres.

Notion de contexte

Dans le cadre d'un échange entre deux locuteurs, le contexte correspond à l'information qu'on évoque, mais ne répète pas. Elle est mémorisée chez les deux partenaires et éclaire, oriente la suite du traitement.

En langage naturel, ce sont les informations qu'on a inférées sur le discours ou le récit.

Quelques formes de contexte

En informatique, la forme la plus simple est le drapeau (qui prend les valeurs 0 ou 1). Au cours d'un traitement de l'information, il peut être testé (par l'alternative), et ainsi commuter le comportement de l'agent entre deux fonctionnements opposés.

Une forme similaire mais plus complexe peut être le marqueur : il peut prendre N valeurs. Au cours d'un traitement de l'information, il peut être testé par un switch, et ainsi commuter le comportement entre N fonctionnements distincts.

En langage naturel, on trouve aussi un contexte qui passe des informations de phrase en phrase. Ce peut être, les instanciations des pronoms. Par exemple, la phrase : *Jim donne à Jeanne*, à la lumière d'un contexte qui préciserait que l'actant est Jim, et qu'on parle de Jeanne, se simplifie en : *Il lui donne*.

En programmation, le contexte s'appelle aussi environnement, c'est la liste des affectations / instanciations, i.e. la liste des variables (locales ou globales) et de leur valeur. Ci-contre, dans l'illustration de l'évaluation de factorielle (3), à mesure que l'ordinateur s'enfonce dans la récursion, on voit les contextes / environnements qui s'emboîtent : (n=3 (n=2 (n=1))).

Les formes les plus complexes de contexte peuvent aller jusqu'à des listes de prédicats (représentions des connaissances ou des intentions d'un agent) :
- Paul sait que (il fait beau – Jo est dans la maison).
- Jo projette (Jo va maison – Jo prend panier).

Importance et effet du traitement contextuel :

Donner de la cohérence au procès

En informatique le contexte permet la communication, et donc l'échange de données, entre processus.

En langage naturel, le contexte, quand il est transporté de mot à mot au sein d'une phrase permet les vérifications de genre et de nombre. Il permet aussi de traiter les déictiques et d'identifier les pronoms. Il est transporté de phrase en phrase, afin de les lier et produire un discours ou récit.

Simplifier l'écriture des règles

Nous avons déjà évoqué ce problème : en traitement du langage naturel, si on veut traiter le genre et le nombre au moyen des grammaires hors contexte (G.H.C.), i.e. faire des vérifications de genre et de nombre sur les sujets et objets de la phrase, le programmeur doit biaiser et donc transporter l'information dans l'état du monde, grâce aux marqueurs utilisés

au sein des règles, i.e. au moyen des noms qu'il leur donne. Chacune est dédiée à un cas particulier : une règle pour le masculin, une règle pour le féminin. Ensuite, si on veut rajouter un traitement du nombre, puisqu'avec les G.H.C. il faut une règle dédiée par cas, il faut doubler le nombre précédent de règles.

Avec les grammaires contextuelles, la difficulté disparaît, car justement, une règle est capable de transporter un contexte. Quand le programmeur doit rajouter un traitement à un jeu de règles, il rajoute un ou deux termes (variable et/ou constante). Certes, elle devient un peu plus longue, mais ainsi, il évite de doubler son nombre de règles principales.

Utilisation des variables au sein d'un contexte local (dans les grammaires de type 1)

Rappels sur quelques points de vocabulaire : notion de variable typée

Rappels sur la notion intuitive de variable

Soit la description de l'action suivante : *quelqu'un qui est quelque part va quelqu'autre part*. Dans cet énoncé, on trouve trois variables : '*quelqu'un*', '*quelque part*' et '*quelqu'autre part*'. Nous avons déjà constaté, avec un intérêt certain, que non seulement le langage naturel inclut la notion de variable, mais qu'en plus il les marque au moyen du préfixe '*quelque*'. Ainsi '*quelqu'un*', '*quelquepart*' et '*quelqu'autrepart*' sont des variables au sens mathématique et/ou informatique.

Ces variables ne sont pas totalement libres : ce sont des variables typées, i.e. avec une restriction de type

- La variable '*quelqu'un*' ne peut prendre qu'une valeur de personne.
- Les variables '*quelque part*' et '*quelqu'autre part*' ne peuvent prendre que des valeurs de lieu.
- La variable '*quelque chose*' ne peut prendre qu'une valeur d'objet (une entité appartenant à la classe des choses).

Rappels sur quelques points de vocabulaire : notion d'affectation ou d'instanciation

Donner une valeur à une variable, c'est l'affecter pour toujours ou l'instancier pour un moment

Dans l'exemple précédent, l'action de déduire que : *quelqu'un c'est Paul*, et plus généralement de donner une valeur à une variable, s'appelle *affecter une valeur à cette variable* (si l'opération est définitive). Sinon, elle s'appelle une instanciation.

À la lumière de ces définitions, maintenant reformulons, en termes informatiques, nos trois phrases :

- La variable 'quelqu'un ne peut être affectée/instanciée que par une personne.
- Les variables 'quelquepart et quelqu'autrepart ne peuvent être affectées/instanciées que par un lieu.
- La variable quelque chose ne peut être affectée/instancié que par un objet.

Introduction intuitive à la notion d'unification

Unification de deux énoncés

Le mécanisme d'unification détermine les affectations et/ou instanciations nécessaires pour qu'un énoncé général (qui comporte des variables) s'unifie, (en franglais matche) avec un énoncé comportant des constantes.

Exemple d'unification au moyen d'une fonction lisp :

Pour ce faire, utilisons un énoncé en langage naturel

Si d'abord j'énonce :*Quelqu'un qui est quelque part va quelqu'autre part*et si ensuite je précise : *Paul qui est dans le jardin va dans la maison*, l'auditeur rapidement fait le rapprochement et infère : *quelqu'un c'est Paul, quelque part c'est le jardin, et quelqu'autre part, c'est la maison*. Ainsi, on peut dire qu'il a unifié les deux énoncés.

En lisp on peut écrire une fonction telle que, si on lui commande :

```
(unifie                                  // Le nom de la fonction lisp
'(_qqun qui est _qqpart va _qqautrepart) // L'énoncé général portant des variables
'(Paul qui est jardin va maison))        // L'énoncé instancié avec les constantes
```

Alors l'interprète retourne l'environnement suivant :

((_qqun Paul) (_qqpart jardin) (_qqautrepart maison))

Cette liste de listes est un contexte, une liste d'instanciations, i.e. de liaisons *Variable→Valeur*.

Introduction intuitive à la notion de propagation d'un contexte dans un énoncé

Illustration de la notion de ***propagation de contexte*** dans le cadre de la vie quotidienne

Si ensuite, dans ce même contexte, j'énonce : *Ce quelqu'un qui est quelque part prend quelque chose* alors l'auditeur, puisqu'il sait que *quelqu'un c'est Paul, quelque part c'est le jardin, et quelqu'autre part, c'est la maison*, il est capable d'inférer, de comprendre que maintenant : *Paul qui est dans la maison prend quelque chose*

En lisp on peut écrire une fonction de propagation de contexte telle que, si on énonce la commande :

```
(propage                                                    // Le nom de la fonction lisp
 '(_qqun qui est _qqautrepart prend _qqchose),              // L'énoncé général portant des variables
 '((_qqun Paul) (_qqpart jardin) (_qqautrepart maison)) )   // L'environnement/contexte pour instancier l'énoncé général
```

Alors l'interprète lisp retourne l'énoncé suivant :

```
'(Paul qui est maison prend _qqchose)      // Paul qui est dans la maison prend quelque chose
```

Définition de la propagation d'un environnement dans un énoncé

Propager un environnement/contexte dans un énoncé, c'est utiliser les instanciations qu'il décrit, pour remplacer les variables contenues dans le texte de cet énoncé, par leur valeur listée dans les liaisons.

À partir de quel type de grammaire l'ordinateur propage-t-il l'environnement dans un énoncé

Les grammaires de type 2 sont libres de tout contexte. Elles ne peuvent donc pas élaborer puis propager un environnement dans un énoncé. Mais les grammaires de type 1 et de type 0 le peuvent car elles sont contextuelles.

Systèmes de règles de production : grammaires formelles contextuelles de type 1

Introduction : les grammaires contextuelles de type 1 possèdent un contexte local

Dans les grammaires de type 1, pour traiter un problème, d'abord le système de production gère une liste de sous-buts. Ensuite il élabore progressivement le contexte local correspondant et le gère. Pour ce faire, à chaque fois qu'il identifie une variable, il l'ajoute dans sa liste d'instanciations. Puis, à la lumière de cette nouvelle information, il actualise le reste de la liste de sous-buts en y propageant cette instanciation.

Exemple d'une grammaire contextuelle pour traiter le genre dans un groupe nominal

La grammaire décrite ci-dessous est une grammaire de type 1 car elle est contextuelle : on remarque particulièrement son utilisation de la variable GENRE, qu'elle instancie et propage au travers de tout le traitement.

La grammaire

```
GN → GENRE ART GENRE NOM          // Le but à démontrer s'expanse en 4 sous-buts
                                  // GENRE : variable, élément d'un contxte, qui est propagée au long du calcul
GENRE → FEMININ | MASCULIN        // Il faut noter l'utilisation de la notation sous forme de OU. Elle remplace
                                  // les deux règles GENRE → FEMININ, GENRE → MASCULIN
MASCULIN ART → le                 // MASCULIN et FEMININ constituent des constantes qui instancieront
FEMININ ART → une                 //  la variable contextuelle GENRE
FEMININ NOM → chatte
MASCULIN NOM → chien
```

Exemple de séquence de dérivation :génération de la chaîne *une chatte* qui appartient au langage

```
GN                                // Point de départ
                                  // En appliquant la règle : GN → GENRE ART GENRE NOM le but à
GENRE ART GENRE NOM               // démontrer est expansé en 4 sous-buts (et mémorisé dans la pile de buts)
FEMININ ART GENRE NOM             // La règle GENRE → FEMININ permet de réécrire GENRE par FEMININ, ainsi
                                  // on identifie la variable GENRE et on obtient le contexte ((GENRE FEMININ))
une GENRE NOM                     // Obtenu en appliquant la règle FEMININ ART → une.
une FEMININ NOM                   // Obtenu en propageant le contexte ((GENRE FEMININ)) dans la liste des s/buts
                                  // i.e. en remplaçant la variable GENRE par sa valeur : FEMININ dans cette liste.
une chatte                        // Obtenu en appliquant la règle FEMININ NOM → chatte.
```

En conclusion, on a généré la séquence *une chatte* qui constitue un GN (groupe nominal) appartenant à ce langage.

Ex : une grammaire contextuelle pour traiter le genre et le nombre dans un groupe nominal

Le propos de l'exemple ci-dessous est de montrer la puissance et la souplesse des grammaires contextuelles. Leur aptitude à propager un contexte/environnement local simplifie l'écriture des règles. Dans cette grammaire, la règle centrale teste à la fois le genre et le nombre des éléments du groupe nominal. Il faut noter que cette vérification s'effectue au sein d'une règle centrale unique : *GN → ART GENRE NOMBRE NOM GENRE NOMBRE* alors qu'avec une grammaire hors contexte, il aurait fallu en écrire au moins 4.

La grammaire utilisée dans cet exemple

GN → ART GENRE NOMBRE NOM GENRE NOMBRE // L'unique règle centrale qui porte toute la vérification
// Par contre, il faut énoncer explicitement les instanciations possibles
// des variables. Notez que cette déclaration prend une ligne de plus, c'est peu
// comparé au GHC, où elle doublerait le nombre de règles centrales
GENRE → FEMININ | MASCULIN // Les instanciations possibles de GENRE
NOMBRE → SINGULIER | PLURIEL // Les instanciations possibles de NOMBRE
// Enfin la déclaration des noms, i.e. des constantes demeure sensiblement identique aux GHC
ART MASCULIN SINGULIER → le
ART FEMININ SINGULIER → la
NOM FEMININ SINGULIER → chatte
NOM MASCULIN SINGULIER → chien
ART MASCULIN PLURIEL → les
ART FEMININ PLURIEL → les
NOM FEMININ PLURIEL → chattes
NOM MASCULIN PLURIEL → chiennes

Exemple de séquence de dérivation :

Vérifions qu'on peut générer un groupe nominal cohérent en genre et nombre, qui appartienne au langage

GN	// Point de départ (but à démontrer)
ART GENRE NOMBRE NOM GENRE NOMBRE	// expansion au moyen de la règle centrale
ART FEMININ NOMBRE NOM FEMININ NOMBRE	// Après application de la première règle sur le GENRE
ART FEMININ SINGULIER NOM FEMININ SINGULIER	// Après application de la deuxième règle sur le NOMBRE
la NOM FEMININ SINGULIER	// Après application de la seule règle déclenchable
la chatte	// GN cohérent en genre et nombre, appartenant au langage.

Grammaires de type 0 reconnues par la machine de Turing

La machine de Turing

Nous en sommes arrivés à des outils puissants comme les langages de programmation

Nous voici arrivés au domaine de la machine universelle de Turing, dont, paradoxalement nous ne parlerons guère. En effet il correspond aussi au domaine, non enviable, des anciens langages de programmation non structurés (basic, cobol et fortran). Certes ils sont omnipotents. Ils peuvent coder un programme à partir d'analyses structurées selon la forme d'un réseau (algorithme et organigramme). Certes un programme purement structuré n'est pas capable de le faire...

Attention danger ! Ils sont omnipotents...

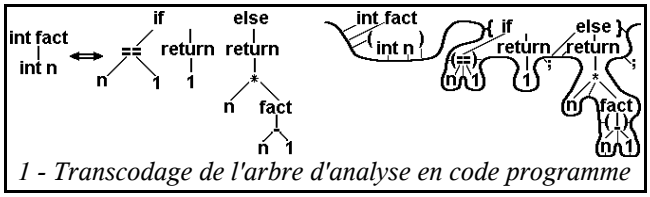
Ces vieux langages sont omnipotents, mais ils présentent de graves défauts !

Lisibilité et parcours

Ces langages sont dépassés car ils fournissent un code difficilement lisible à cause des instructions de branchement.

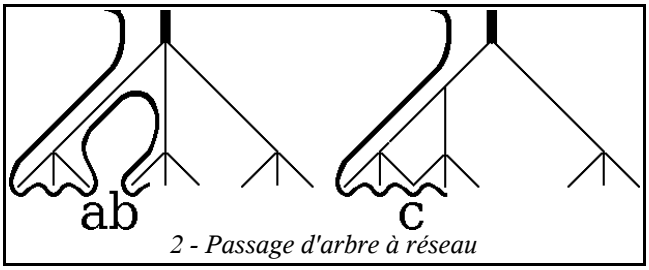
Au contraire, un programme structuré présente une structure d'arbre, nettement plus élégante et fiable. Ainsi, quand l'analyste fournit au programmeur un synoptique de programme structuré, la codification en est facilitée car l'arbre à coder est systématiquement parcourable.

Dans l'illustration ci-contre, le graphique de gauche montre l'arbre du programme provenant de l'analyse. Le graphisme de droite montre le parcours de cet arbre, qui permet de le codifier, i.e. d'écrire le programme C correspondant. En effet, en programmation structurée, il existe une méthode systématique pour parcourir cet arbre, en relever les symboles et générer un code digeste pour le compilateur C. Évidemment, cette aptitude facilite grandement la codification.



1 - Transcodage de l'arbre d'analyse en code programme

Par contre, en langage non structuré, l'analyse d'un programme se présente sous forme d'un organigramme qui souvent possède une structure de graphe. Or un graphe n'est pas parcourable systématiquement. Il faut le découper en séquences et lui ajouter des étiquettes pour les branchements. Tout cela est source d'erreur.



2 - Passage d'arbre à réseau

Le graphique ci-contre illustre cet aspect de façon très simple. L'arbre de gauche est parcourable entre les feuilles *a* et *b*. Mais quand elles se collent au point *c*, on obtient (figure de droite), un réseau et cette structure n'est plus parcourable.

Le contexte local des langages structurés permet la modularité du code

Les langages non-structurés (assembleur, fortran, cobol, basic...) ne possèdent pas (ou peu) de contexte local, mais plutôt un important contexte global. Alors les variables utilisées sont visibles/accessibles depuis tout point du programme. Mais ce qui semble un avantage, constitue en fait un inconvénient nommé *collision de variable* qui empêche la modularité des logiciels : le programmeur qui utilise par exemple la variable i dans une boucle peut très bien en voir l'exécution boucler indéfiniment car il appelle une routine écrite par un collègue qui utilise cette même variable et donc la modifie intempestivement ! Grrrr !

Conclusion : le paradoxe de la privation de liberté

Dans cette histoire, le paradoxe est que les beaux langages de programmation (les langages pédagogiques : Lisp, algol, Pascal, Prolog, c++, Python) sont fiables, faciles à utiliser et à lire car ils restreignent la liberté du programmeur ! Par exemple, ils induisent un travail supplémentaire de compartimentation. Le programme appelant doit passer les paramètres nécessaires au sous-programme appelé. Pour un hacker qui vient de l'assembleur, c'est énervant, il considère que c'est inutile, car ces données sont déjà présentes et visibles dans l'environnement global du système !

L'avantage des langages structurés, surtout quand ils sont fortement typés, est qu'ils sont plus difficiles à compiler ! En effet 90% des erreurs de codification s'y détectent à la compilation, ce qui fait qu'ensuite, un programme exécutable est rapidement débogué. Par contre, les vieux langages non-structurés (et surtout l'assembleur) sont faciles à compiler, mais alors *bonjour la galère* quand il faut déboguer[2] !.

Le domaine du langage naturel

Nous voici arrivés dans le domaine du langage naturel, si complexe qu'il peut seulement être reconnu par des analyseurs construits au moyen de langages de programmation possédant la puissance de la machine de Turing. Puisque nous ne sommes par linguistes, nous sommes plutôt à la recherche des jolies régularités du langage, alors nous n'irons pas fouiller dans les coins, à la recherche des originalités de la langue. Ainsi nous n'aborderons pas ces aspects trop difficiles et si vastes[3].

Contexte global et opérations enrichies

Le domaine des grammaires de type 0 se caractérise par un accès au contexte global au moyen d'actions diverses et variées.

Les effets de bord de l'informatique

En informatique on nomme *effet de bord*[4] (side effects en anglais) toute action qui modifie l'environnement global, il se compose du contexte global et du monde extérieur[5].

Les opérations de sortie modifient le monde extérieur

Toutes ces actions modifient le monde extérieur (écriture à l'écran ou sur une imprimante, activation d'un actionneur qui agit sur le monde).

Les opérations internes modifient le contexte global

2 Un des enseignants que j'ai eu (Jacques Pitrat), disait que comparé à C, il faut 10 fois plus de temps pour déboguer en assembleur, et 100 fois plus dans un métalangage.

3 Un des enseignants que j'ai eu (Jacques Jayez), disait carrément que le langage naturel était un merdier. Dans la bouche de ce chercheur longiligne, élégant et distingué, la formule détonnait, mais je dois avouer que j'ai toujours apprécié ce point de vue !

4 Effets de bord : *side effects* en anglais.

5 Le langage C n'est pas un langage purement structuré, il possède aussi des instructions *inesthétiques* à effets de bord qui lui confèrent la puissance de la machine universelle de Turing.

En informatique, la chose se fait au moyen de l'affectation/effacement de variables globales par toutes sortes de données : entiers, symboles, prédicats ou listes.

Opérations d'entrée

En théorie une bonne prise d'information sur l'extérieur doit se faire sans perturber le milieu mesuré. Donc ce n'est pas dans cette direction qu'il faut chercher la modification de l'environnement global. Mais en fait ces mesures débouchent sur des opérations internes : par exemple, la lecture d'un caractère saisi au clavier ne modifie pas le milieu extérieur. Mais l'information retournée est mémorisée dans un registre tampon : le contexte global est donc modifié.

Vie artificielle

Dans les simulations de vie artificielle, ces opérations enrichies se produisent quand les actionneurs des agents agissent sur le monde. Dans ce cas il apparaît comme un environnement global.

En traitement du langage naturel : les réseaux de transition augmentés

Il faut noter qu'à la fin du polycopié à propos des automates à pile, nous avons déjà utilisé les réseaux de transition augmentés pour passer une phrase passive en mode actif. En traitement du langage naturel, quand un réseau de transition récursif est augmenté d'actions sur le contexte global, il récupère la puissance de traitement d'une machine de Turing.

Conclusion

Partis d'instruments très simples (les systèmes formels et les automates), nous avons progressivement complexifié nos outils, et maintenant nous récoltons le fruit de nos efforts, nous disposons de modèles très simples permettant de rentrer à l'intérieur des mécanismes fondamentaux de l'informatique et de l'I.A..