



**Prérequis :** théorie des langages et logique, théorie des graphes, calculabilité - lire, apprendre, comprendre.

**Pliage :** recto visible (autre face), traits gris rentrants, traits rouges saillants. Découper selon le trait rouge entre les deux ●, puis achever le pliage.

## ① Théorie des langages et ses propres prérequis

La **théorie des langages**, avec ses propre prérequis, est bien sûr un prérequis à l'étude de la compilation, même si il ne faut **pas réduire la seconde à une application de la première**.

En particulier, connaître la théorie des **langages rationnels** et celles des **langages algébriques** est indispensable. Comprendre le rôle et le fonctionnement des **analyseurs syntaxiques** l'est aussi, ainsi que savoir évaluer la qualité d'une grammaire relativement à la relation **syntaxe-sémantique**, aux possibles **ambiguïtés**, et aux limitations d'une **stratégie** d'analyse syntaxique que l'on ne choisit pas toujours.

On se rappellera donc que la grammaire

$Expr \rightarrow Expr '+' Expr \mid Expr 'x' Expr \mid Num \mid '(' Expr ')'$

est **ambiguë** et qu'une façon de supprimer cette ambiguïté est de lui substituer la grammaire suivante :

$Expr \rightarrow Prod '+' Expr \mid Prod$   
 $Prod \rightarrow Fact 'x' Prod \mid Fact$   
 $Fact \rightarrow Num \mid '(' Expr ')'$

De même, on se rappellera que la grammaire

$Diff \rightarrow Diff '-' Elem \mid Elem$

ne convient pas à un analyseur LL ou par **descente récursive**, parce qu'elle est **récursive à gauche**, mais qu'elle engendre la même chose que

$Diff \rightarrow Elem RestDiff$   
 $RestDiff \rightarrow '-' Elem RestDiff \mid \epsilon$

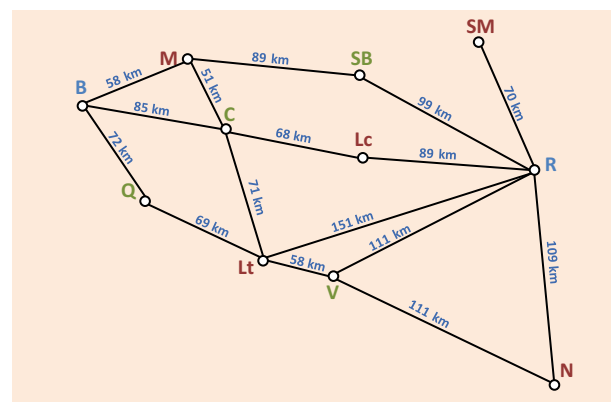
qui convient très bien (forme normale de Greibach).

Dans les deux cas, la nouvelle grammaire produira des **arbres de dérivation** qui ne sont **pas de la forme que la grammaire initiale naïve aurait produite**. On peut donc avoir intérêt à reconstituer l'arbre de dérivation attendu **avant tout autre opération**.

## N'oubliez jamais !

Un **modèle** est toujours **imparfait**, parfois **utile**, et c'est tout ce qu'on peut lui demander (d'après George Box 1978, voir aussi a contrario les cartes parfaites et inutiles de Jorge Luis Borges 1946 et Lewis Carroll 1893, cartes à l'échelle 1/1).

L'informatique ne travaille que sur la **représentation** des choses ; il lui faut des capteurs et des actionneurs pour toucher les choses. Par contre, travailler sur des représentations de représentations de ... est courant. Par exemple, **440** est une représentation d'un entier qui peut être la représentation de la mesure d'une fréquence, tout comme **1B8**, et **La<sub>3</sub>** pourraient être d'autres représentations de la même fréquence. Il faut toujours se demander ce qui a du sens par rapport à ce qui est modélisé.



Un graphe non-orienté pondéré et colorié

## ② Arbres

Un graphe non-orienté sans cycle est appelé un **arbre**. Dans sa représentation graphique, de chaque sommet d'un arbre partent des ramifications qui ne se rejoignent jamais, et chaque sommet peut jouer le rôle de point de départ de l'ensemble des ramifications. On peut désigner un sommet particulier pour être la **racine** d'un arbre. On parle alors d'un arbre **enraciné**. En pratique, la plupart des arbres sont enracinés et on omettra de le spécifier.

Un point de vue alternatif qui ne s'appuie pas sur la théorie des graphes, mais sur celle des ensembles, est de définir un arbre par un **ensemble de sommets**, parmi lesquels un **sommet racine** est distingué, et les autres sommets sont **partitionnés en n sous-arbres** numérotés de 1 à n.

En général, on dessine un arbre enraciné en plaçant la racine en bordure du dessin plutôt qu'au centre, mais le choix de la bordure est une affaire de tradition. Par exemple, sont souvent représentés **racine en haut** les arbres de dérivation, les arbres de décision (parfois racine à gauche !), les arbres d'indexation en base de données, les arbres structures de données (ex. ceux de l'algorithme **UNION-FIND**, ou d'un **système de fichiers**, parfois aussi racine en haut à gauche). Par contre, les arbres de preuve sont souvent représentés **racine en bas**, et dans ce cas la racine est la conclusion de la preuve. Remarque : pour les arbres les plus concrets, ex. structures de données, on parle plus facilement de **nœud** (*node* en anglais) que de **sommet**.

L'enracinement d'un arbre crée une **orientation de fait** selon que l'on s'éloigne ou qu'on se rapproche de la racine. On appelle  **fils**  d'un sommet ses voisins qui sont plus éloignés que lui de la racine. On dit qu'il en est le **sommet père**. On dit qu'un arbre est **binaire** (*n-aire*) si ses sommets ont tous **2** (*n*) fils ou aucun. On appelle **feuille** un sommet qui n'a pas de fils. On appelle **niveau** d'un sommet le nombre d'arcs qui le séparent de la racine et **hauteur** d'un arbre le niveau de sa feuille de plus grand niveau.

La représentation concrète (graphique, physique ou informatique) d'un arbre crée aussi une orientation de fait des fils de chaque sommet. On parle alors de **1<sup>er</sup>**, **2<sup>nd</sup>**, ... fils, ou de fils de **gauche**, de **droite**, etc.

Un arbre peut être parcouru de plusieurs façons différentes, illustrées ici par des algorithmes de parcours d'arbres binaires :

• En profondeur d'abord (**DF** pour **Depth First**) :

Input : un sommet ; Output : un mot formé des sommets de l'arbre enraciné en ce sommet,

$DF(s) = s \bullet \text{if } s \text{ est une feuille then } \epsilon \text{ else } DF(s.\text{left}) \bullet DF(s.\text{right}) \text{ fi}$

Correspond à la façon **polonaise préfixe** de linéariser une expression. Il s'agit de la variante **préfixe** car le **s** est produit **avant** ses fils. Il existe une variante **postfixe** où le **s** est produit **après**, et une variante **infixe** où il est produit **entre** ses fils.

• En largeur d'abord (**BF** pour **Breadth First**) :

Input : un mot formé de un sommet ; Output : un mot formé des sommets de l'arbre enraciné en ce sommet,

$BF(w) = \text{if } w = \epsilon \text{ then } \epsilon$   
 $\text{else } w.\text{first} \bullet BF(w.\text{rest}) \bullet \text{if } w.\text{first} \text{ est une feuille then } \epsilon \text{ else } (w.\text{first}.\text{left} \bullet w.\text{first}.\text{right}) \text{ fi}$

Correspond à un parcours niveau par niveau. Est utile quand la recherche d'un nœud dans un arbre s'accompagne de l'objectif de minimiser la distance à la racine. Est aussi utile quand l'arbre peut avoir des branches infinies.

## ① Théorie des graphes

Un graphe est une structure constituée d'un ensemble de **sommets**, souvent noté **V** (pour **Vertices** en anglais), et d'un ensemble **d'arcs**, souvent noté **E** (pour **Edges**). On appelle parfois les sommets des **nœuds**. On considère soit des graphes **non-orientés**, et dans ce cas chaque arc est une paire de 2 sommets  $\{e, f\}$ , parfois noté  $e - f$ , qu'on appelle les **extrémités** de l'arc (on dit aussi que **e** et **f** sont voisins), soit des graphes **orientés**, et dans ce cas chaque arc est un couple de 2 sommets  $(e, f)$ , parfois noté  $e \rightarrow f$ , qu'on appelle le sommet **initial** et le sommet **final** de l'arc. Comme partout en théorie des ensembles, les sommets peuvent être quasiment n'importe quoi.

On dessine les graphes finis de façon intuitive en représentant les sommets par des points, des icons, des textes, etc., et les arcs par des flèches ou des traits selon qu'ils sont orientés ou non. Le tout peut être décoré de différentes annotations selon l'usage qui est fait du graphe ou ce qu'il représente (voir la figure ci-dessus, ainsi que pour les exemples qui suivent).

Une fois défini un graphe, on peut s'intéresser à ses **chemins**, suites de  $e_i$  telles que  $e_i - e_{i+1}$  ou  $e_i \rightarrow e_{i+1}$ , à l'existence de chemins entre deux sommets choisis a priori, à l'existence de **composantes fortement connexes** (dont tous les éléments sont connectés à tous les autres par au moins un chemin ; le graphe ci-dessus forme une composante fortement connexe), etc. On peut associer aux arcs un **poids**, et alors qualifier les chemins par le poids total de leurs arcs (ex.  $w(B - Q - Lt - V - N) = 310 \text{ km}$ ). On peut alors s'intéresser aux paires de sommets les plus distants l'un de l'autre (ex.  $\{B, SM\}$ ). On peut aussi s'intéresser à l'existence de **cycles**, c'est-à-dire de chemins qui reviennent à leur point de départ (ex.  $B - M - C - B$ ).

On peut aussi s'intéresser aux propriétés qui sont vraies sur tous les chemins, ou seulement sur au moins un chemin, ou celles qui sont vraies sur tous ou au moins un chemin qui mène vers un sommet ou qui en part (ex. tous les chemins au départ de **SM** passent par **R**).

On peut aussi **colorier** un graphe, ex. en assignant des couleurs aux sommets de sorte que deux sommets voisins aient toujours deux couleurs différentes (comme sur l'illustration où 3 couleurs sont nécessaires au total à cause des triangles, ex.  $V - R - N - V$ ), ou bien en assignant des couleurs aux arcs de sorte que deux arcs ayant une extrémité commune soient toujours de couleurs différentes. Ce sont des problèmes **NP-complets** qui modélisent toute sorte de problèmes concrets intéressants.

## Calculabilité

La théorie de la calculabilité montre très facilement par un argument de cardinalité de l'ensemble des fonctions et de l'ensemble des programmes que beaucoup de fonctions ne sont pas réalisables par un programme ; on dit qu'elles ne sont pas **calculables**. Mais cela ne dit pas lesquelles ne le sont pas.

Le **théorème de Rice** délimite une catégorie de fonctions qui ne peuvent pas être calculables ; ce sont les fonctions qui décident des propriétés **extensionnelles** de programme. C'est-à-dire des propriétés de programme qui sont en fait des propriétés de leur **sémantique**, ou dit autrement, des propriétés telles que deux programmes qui ont la même sémantique ont forcément la même propriété. Par exemple, le nombre de lignes d'un programme n'est pas une propriété extensionnelle, car il est facile de concevoir que deux programmes de tailles différentes aient la même sémantique. Inversement, ne jamais boucler est une propriété extensionnelle, ainsi que ne jamais rendre la valeur 0, ou réaliser une fonction croissante. De même, toutes les propriétés opérationnelles mais liées à la sémantique, comme faire une division par zéro, un débordement de tableau ou une écriture dans une mémoire non réservée, sont des propriétés extensionnelles. Aucun algorithme ne peut décider à coup sûr et sans boucler si ces propriétés sont vraies pour un programme quelconque.

## ② Systèmes de déduction

Les **systèmes de déduction** sont un moyen de formaliser la sémantique des systèmes formels. Ils établissent quels jugements sont vrais ou faux. Ils décrivent d'une façon graphique comment former des **règles de déduction** et les combiner pour former des **preuves**. On va l'illustrer en prenant l'exemple des **axiomes de Armstrong** utilisés en **théorie des bases de données** pour raisonner sur les dépendances fonctionnelles :

• **transitivité** :  $X \rightarrow Y$  et  $Y \rightarrow Z$  entraînent  $X \rightarrow Z$

• **inclusion** :  $Y \subseteq X$  entraîne  $X \rightarrow Y$

• **augmentation** :  $X \rightarrow Y$  entraîne  $X, W \rightarrow Y, W$

Ici, les jugements sont des notations de dépendances fonctionnelles :  $X \rightarrow Y$ .

La transcription des axiomes de Armstrong en règles de déduction se fait de la façon suivante :

$\frac{X \rightarrow Y \quad Y \rightarrow Z}{X \rightarrow Z}$  trans       $\frac{Y \subseteq X}{X \rightarrow Y}$  incl       $\frac{X \rightarrow Y}{X, W \rightarrow Y, W}$  aug

Une règle de déduction relie des jugements **hypothèses** (au dessus de la ligne) et un jugement **conclusion** (au dessous). Ces règles sont en fait des **schémas** de règles où des lettres, ici **X, Y** et **Z**, jouent le rôle de **méta-variables** qui peuvent être remplacées par n'importe quoi de légal dans leur contexte : ici, des **ensembles de noms d'attributs de schéma relationnel**.

Certaines règles de déduction n'ont pas d'hypothèse ; on les appelle des **axiomes** (ne pas confondre les axiomes du système de déduction et les axiomes de la théorie modélisée par le système de déduction). Ici, les jugements conclusions des axiomes sont les **dépendances fonctionnelles** de l'application modélisée, ex.  $ville \rightarrow taxe$ ,  $ville \rightarrow météo$ , et  $taxe \rightarrow prix$ .

$\frac{}{ville \rightarrow taxe}$  ax       $\frac{}{ville \rightarrow météo}$  ax       $\frac{}{taxe \rightarrow prix}$  ax

On peut alors former des instances des règles de déduction en remplaçant les méta-variables par des expressions du domaine modélisé, puis former des arbres preuves en agençant les instances de règles de telle sorte qu'à une instance d'hypothèse correspond toujours une instance de conclusion ou une instance d'axiome. Voici des exemples de preuve :

$\frac{}{ville \rightarrow taxe}$  ax       $\frac{ville \rightarrow taxe}{ville \rightarrow ville, taxe}$  aug       $\frac{ville \rightarrow météo}{ville, taxe \rightarrow taxe, météo}$  aug  
 $\frac{}{ville, prix \rightarrow taxe, prix}$  aug       $\frac{}{ville \rightarrow taxe, météo}$  trans

Les règles de déduction se comportent alors comme des règles d'une grammaire d'arbre de preuve. On trouve aux feuilles de cet arbre (en haut) les jugements axiomes, et à sa racine (en bas) un jugement conclusion qui est le théorème démontré par cette preuve.