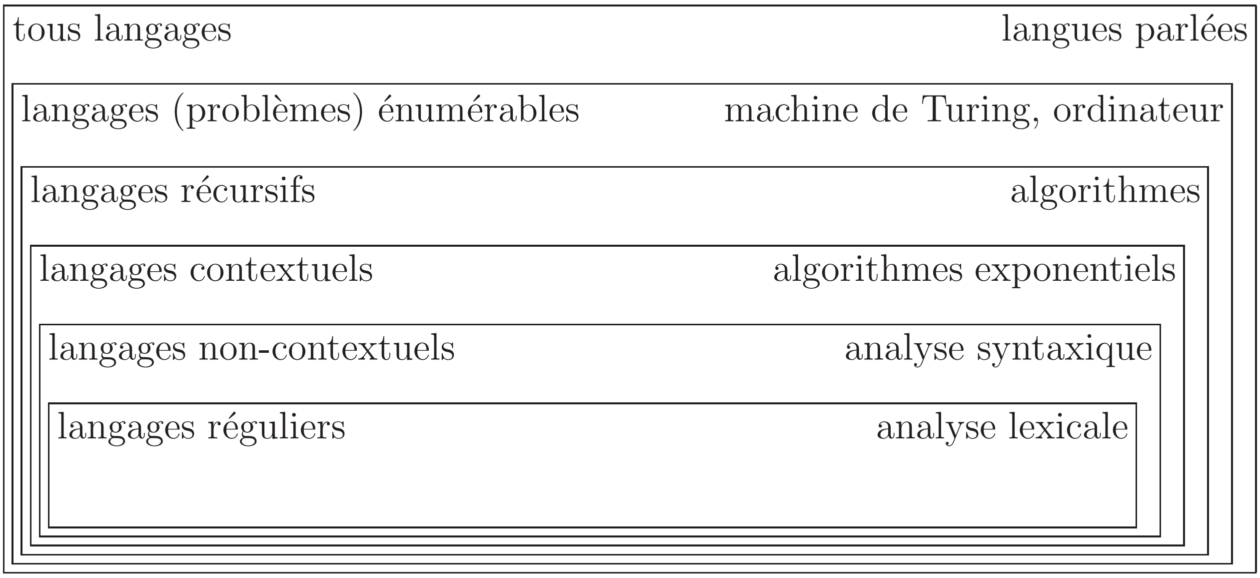
Langages et Grammaires

Dans les années 50, le linguiste Noam Chomsky se mettait à développer une théorie formelle des langues parlées.

Ces résultats ont révolutionné l'informatique. Elles ont, entre autres, donné un formalisme ainsi que des techniques pour réaliser des compilateurs.

Pour les langues parlées, ses études ont montré qu'il est impossible de les formaliser en entier. A l'époque (l'époque de l'IA), ce fut ressenti comme un affront, mais pourquoi tout formaliser et automatiser?

Hiérarchie de Chomsky



Survol de cette période

- Langages et grammaires en théorie
- Langages et grammaires en compilation

Alphabets

Un **alphabet** est un ensemble fini. Les éléments ne sont pas forcément des caractères.

Exemples

$$\begin{aligned} A_1 &= \{\ a, b\ \} \\ A_2 &= \{\ a, b, c, \ldots, z, A, B, \ldots, Z\ \} \\ A_3 &= \{\ le, grand, petit, chien, chat, mulot, chasse, mange\ \} \end{aligned}$$

Et en compilation?

Lors de l'**analyse lexicale**, l'alphabet est constitué de caractères:

$$A_l = \{\ \text{“}a\text{”}, \ldots, \text{“}z\text{”}, \text{“}A\text{”}, \ldots, \text{“}Z\text{”}, \text{“}0\text{”}, \ldots, \text{“}9\text{”}, \text{“}+\text{”}, \text{“}-\text{”}, \ldots\ \}$$

Lors de l'**analyse syntaxique**, il est constitué de lexèmes (les non-terminaux de l'analyse lexicale):

$$A_s = \{\ LPAREN, RPAREN, IF, THEN, ELSE, \ldots, ID, NUM\ \}$$

Mots, Phrases

Un **mot** (ou une **phrase**) $m \in A^*$ est un enchaînement fini (pas de flux!) d'éléments de l'alphabet. Le mot ϵ représente le **mot (la phrase) vide**.

Exemples

mots sur A_1	$\epsilon, a, abb, ba, \ldots$
mots sur A_2	$\epsilon, a, abb, ba, xya, chasse, \ldots$
mots sur A_3	ϵ, *chasse*, *le petit chien*, *le mulot chat grand*, ...

A noter: Les éléments de l'alphabet A_3 sont des mots sur l'alphabet A_2.

Et en compilation?

En **analyse lexicale**, les mots comprennent les mots clés ("if", "then", "else", ...), les délimiteurs ("{", "}", ";", ...), les nombres ("42", "2002", ...) et les identificateurs ("abc", "x11", ...). Tous les mots, à l'exception de commentaires et des espaces, sont transformés en **lexèmes**, une représentation particulière.

En **analyse syntaxique**, les mots (on dirait plutôt phrases, voir plus haut) sont des combinaisons de lexèmes (provenant de l'analyse lexicale). Par exemple

$$ID\ ASSIGN\ ID\ PLUS\ NUM \qquad x := x + 1$$

serait accepté comme mot admissible, tandis que

$$NUM\ ASSIGN\ ID\ PLUS\ RBRACE \qquad 13 := x + \}$$

ne serait pas accepté.

Langages

Un **langage** est un ensemble (fini ou infini) de mots sur un alphabet.

Exemples

$$
\begin{array}{lcll}
L_1 & = & A_1^* & \text{(sur } A_1\text{, infini)} \\
L_2 & = & \emptyset & \text{(sur } A_1\text{, } A_2\text{, } A_3\text{, fini)} \\
L_3 & = & \{\, a, aa, ab \,\} & \text{(sur } A_1\text{, } A_2\text{, fini)} \\
L_4 & = & A_3 & \text{(sur } A_2\text{, fini)} \\
L_5 & = & \{\, \mathit{le\ chat\ chasse\ le\ mulot} \,\} & \text{(sur } A_3\text{, fini)}
\end{array}
$$

Il n'est naturellement pas nécessaire d'"utiliser" toutes les lettres de l'alphabet dans un langage.

Et en compilation?

En **analyse lexicale**, les langages comprennent les mots clé, les nombres, les identificateurs, etc. Ces langages sont en principe infinis (bien sûr, la restriction sur les bornes des nombres et sur la longueur des identificateurs les rend finis, mais ils restent toujours assez grands...)

$$L_l = \{\, \text{“}if\text{”}, \text{“}then\text{”}, \text{“}else\text{”}, \ldots, \text{“}42\text{”}, \ldots, \text{“}x1\text{”}, \ldots, \text{“}\{\text{”}, \text{“}\}\text{”}, \ldots \}$$

En **analyse syntaxique**,

$$L_s = \{\, CLASS\ ID\ LPAREN\ ID\ ASSIGN\ NUM\ RPAREN\ \ldots \}$$

Evidemment, il faut un autre schéma que l'énumération pour décrire les langages (de programmation)...

Grammaires

Une **grammaire** (N, T, S, R) est un outil pour décrire un langage.

N symboles non-terminaux; $S \in N$ symbole de départ;

T symboles terminaux;

$R \subseteq (N \cup T)^+ \times (N \cup T)^*$ ensemble fini de règles de dérivation.

Exemples

$G_1 = (\{ S \}, A_1, S, \{ S \to Sa, S \to Sb, S \to \epsilon \})$ décrit L_1.

$G_2 = (\{ S \}, A_1, S, \emptyset)$ décrit le langage L_2 sur A_1.

$G_3 = (\{ S, A \}, A_1, S, \{ S \to aA, A \to a, A \to b, A \to \epsilon \})$ décrit L_3.

$G_4 = (\{ S \}, A_2, S, \{ S \to le, S \to grand, S \to petit, \dots \})$ décrit L_4.

$G_5 = (\{ S \}, A_3, S, \{ S \to le\ chat\ chasse\ le\ mulot \})$ décrit L_5.

Remarques

Il n'y a pas de grammaire unique pour décrire un langage donné.

Une grammaire peut être **ambiguë**: il existe plusieurs façons de dériver un mot (voir exemple, page 10). Nous allons discuter cette problématique dans le cadre de l'analyse syntaxique.

Il existe d'autres moyens pour décrire des langages (par exemples des automates ou des calculs logiques).

Les grammaires sont souvent regardées sous l'aspect constructif (générer des mots/phrases), tandis que les automates sont plutôt vus sous l'aspect analytique (accepter des mots/phrases).

Dérivations

Pour montrer qu'un mot appartient au langage défini par une grammaire, on donne une dérivation, c-à-d une chaîne d'applications des règles de la grammaire sur le symbole de départ.

Pour montrer qu'un mot n'appartient pas à un langage, il faut analyser toutes les productions possibles, et généralement appliquer un argument de monotonie (par exemple, le nombre de terminaux dans un mot ne diminue pas au cours d'une dérivation).

Exemples

Le mot "aa" appartient au langage $L(G_3)$. Il peut être dérivé par

$$\begin{array}{rcll} S & \Rightarrow & aA & \qquad\qquad \text{(par } S \rightarrow aA\text{)} \\ & \Rightarrow & aa & \qquad\qquad \text{(par } A \rightarrow a\text{)} \end{array}$$

Le mot "bb" n'appartient pas au langage $L(G_3)$. La seule règle par laquelle on peut commencer une dérivation est $S \rightarrow aA$, ce qui produit déjà un "a". Comme aucune des règles n'efface de terminaux, celui-ci ne peut plus disparaître. Pour effacer des terminaux, il faudrait des règles comme par exemple "$aA \rightarrow b$".

Arbres

Chaque dérivation définit un arbre (de dérivation), avec le symbole de départ comme racine, et les lettres du mot comme feuilles. Les branches de l'arbre décrivent la structure de la dérivation. Une grammaire est ambiguë, s'il existe plus d'un arbre pour un mot (voir exemple page 10).

Hiérarchie

Une grammaire définit exactement un langage, tandis qu'il existe de nombreuses grammaires décrivant un langage.

Nous sommes intéressés pour le cas suivant:

Une grammaire est dite **régulière** si toutes les règles ont la forme "$A \to abc$" ou "$A \to Bab$" (non-terminal à gauche, un ou plusieurs terminaux à droite, éventuellement précédés ou suivis par un seul non-terminal). Pour symbole de départ $S \to \epsilon$ admissible.

Une grammaire est dite **non-contextuelle** si toutes les règles ont la forme "$A \to w$" avec $w \in (N \cup T)^+$ (non-terminal à gauche, un ou plusieurs terminaux et/ou non-terminaux à droite). Pour symbole de départ $S \to \epsilon$ admissible.

Un langage est **régulier/non-contextuel** si elle peut être décrite par une grammaire régulière/non-contextuelle.

Evidemment, tout langage fini est régulier.

Les grammaires G_i ($i \in \{ 1, \ldots, 5 \}$), et donc aussi les langages qu'elles définissent, sont non-contextuelles et régulières.

Le langage des palindromes est non-contextuel mais pas régulier (voir exercice 2). La raison est qu'il y a un rapport entre des nombres illimités de caractères à des endroits éloignés. Pour décrire cela, il est nécessaire de mettre des non-terminaux entre les terminaux en relation.

Et en compilation?

En **analyse lexicale**, on utilise une collection de grammaires régulières. L'ensemble des symboles de départ (les lexèmes) représente l'alphabet de l'analyse syntaxique.

En **analyse syntaxique**, on utilise des grammaires non-contextuelles.

Exercice 1

Pour commencer ce genre d'exercice, nous effectuons quelques exemples:

$$S \Rightarrow EaE \Rightarrow Ea \Rightarrow a$$
$$S \Rightarrow EaE \Rightarrow Ea \Rightarrow Eba \Rightarrow ba$$
$$S \Rightarrow EaE \Rightarrow Ea \Rightarrow Eba \Rightarrow Ebba \Rightarrow bba$$
$$S \Rightarrow EaE \Rightarrow Ea \Rightarrow Eba \Rightarrow Ebba \Rightarrow Ebbba \Rightarrow bbba$$
$$S \Rightarrow EaE \Rightarrow Ea \Rightarrow Eba \Rightarrow Ebba \Rightarrow Ebbba \Rightarrow Ebbbba \Rightarrow bbbba$$
$$\ldots$$
$$S \Rightarrow EaE \Rightarrow aE \Rightarrow a$$
$$S \Rightarrow EaE \Rightarrow aE \Rightarrow aEb \Rightarrow ab$$
$$S \Rightarrow EaE \Rightarrow aE \Rightarrow aEb \Rightarrow aEbb \Rightarrow abb$$
$$S \Rightarrow EaE \Rightarrow aE \Rightarrow aEb \Rightarrow aEbb \Rightarrow aEbbb \Rightarrow abbb$$
$$\ldots$$

On constate que les deux côtés du "a" sont indépendants; il peut y avoir un nombre quelconque de "b" à gauche et à droite. Le langage que produit la grammaire est donc

$$L_1 \quad = \quad \{\ b^n a b^m \mid n, m \in I\!N\ \}$$

La grammaire n'est pas régulière, mais la grammaire équivalente (produisant le même langage) avec les règles suivantes l'est:

$$\begin{array}{lcll} S & \rightarrow & Sb & \text{production des "}b\text{" à droite} \\ S & \rightarrow & Aa & \text{production du "}a\text{"} \\ S & \rightarrow & a & \text{pas de "}b\text{" à gauche} \\ A & \rightarrow & Ab & \text{production des "}b\text{" à gauche} \\ A & \rightarrow & b & \text{dernier "}b\text{" à gauche} \end{array}$$

Exercice 1 (suite)

Le mot "*abb*" appartient au langage, c-à-d. $abb \in L_1$.

Cela se démontre en exhibant la dérivation suivante:

$$S \Rightarrow EaE \Rightarrow aE \Rightarrow aEb \Rightarrow aEbb \Rightarrow abb$$

Le mot "*aba*" n'appartient pas au langage, c-à-d. $aba \notin L_1$.

Cela se démontre en montrant qu'il n'existe aucune dérivation de S menant à "*aba*". La seule possibilité pour commencer une dérivation est d'utiliser la règle $S \Rightarrow EaE$:

$$S \Rightarrow EaE \Rightarrow \ldots$$

Cela produit un seul "a", or il n'existe aucune dérivation partant d'un E qui produise un "a". Il est donc impossible de produire un second "a" et donc de dériver "*aba*" à partir de S.

Exercice 2

Chaque fois que l'on produit un "a" à gauche/droite, on doit également produire un "a" à droite/gauche.

Il faut donc une règle "$S \rightarrow aSa$".

Chaque fois que l'on produit un "b" à gauche/droite, on doit également produire un "b" à droite/gauche.

Il faut donc une règle "$S \rightarrow bSb$".

Pour terminer, il faut effacer le symbole de départ et éventuellement ajouter un "a" ou "b".

Ensemble, cela donne:

$S \rightarrow aSa$	"a" à gauche et à droite
$S \rightarrow bSb$	"b" à gauche et à droite
$S \rightarrow \epsilon$	terminer
$S \rightarrow a$	terminer avec "a" au milieu
$S \rightarrow b$	terminer avec "b" au milieu

La grammaire, et donc le langage, est non-contextuel. La grammaire n'est pas régulière et l'on peut montrer que le langage ne l'est pas non plus (en se servant du "lemme de pompage", pour ceux qui en ont entendu parler).

Pour vous familiariser avec cette grammaire, faites quelques dérivations.

Exercice 3

L'intersection contient les mots du langage L_1 qui ont des nombres égaux de "b" à gauche et à droite de "a".

$$L_3 \;=\; \{\; b^n a b^n \mid n \in I\!N \;\}$$

Une grammaire correspondante:

$S \rightarrow bSb$	"b" à gauche et à droite
$S \rightarrow a$	terminer avec "a" au milieu

Ce langage est non-contextuel (voir la grammaire). Comme le langage des palindromes, il n'est pas régulier.

Exercice 4

Classes implémentant la grammaire

```
interface BinOp;    // interface pour les operateurs

abstract class Stm;
class CmpStm extends Stm;    // Stm  -->  Stm ";" Stm
class AssStm extends Stm;    // Stm  -->  id "::=" Exp
class PrtStm extends Stm;    // Stm  -->  "print" Exp

abstract class Exp;
class NumExp extends Exp;    // Exp  -->  num
class IdExp extends Exp;     // Exp  -->  id
class OpExp extends Exp;     // Exp  -->  Exp BinOp Exp
```

Chacune des classes a comme variables les éléments de la partie droite de sa règle. La règle est implémentée par la méthode de construction de la sous-classe. Par exemple,

```
class OpExp extends Exp {
    Exp exp1, exp2; int op;
    OpExp(Exp exp1, int op, Exp exp2) {
        this.exp1 = exp1; this.op = op; this.exp2 = exp2;
    }
}
```

Pour créer une phrase (dans la méthode `main`), on appelle les constructeurs (notez la structure en arbre):

```
Exp exp1 = new NumExp(15);
Exp exp2 = new NumExp(12);
Exp exp3 = new OpExp(exp1, BinOp.Plus, exp2);
```

Exercice 4 (suite)

L'interprète se sert d'une table de symboles simple. Pour chaque identificateur, il y a une entrée avec le nom et la valeur actuelle.

```
class Table {
    String id; int val; Table tail;
    class TableUtils {
    static Table enter(String id, int val, Table table);
    static int lookup(String id, Table table);
    }
}
```

Liste linéaire. Entrer: rajouter au début de la liste. Lire: chercher la première occurrence du nom de l'identificateur.

```
class Interpreter extends TableUtils {
    private Table table;
    public void interprete(Stm stm) { ... }

    public int interprete(Exp exp) {
        ...
        // exp op exp
        if (exp instanceof OpExp) {
            OpExp cast = (OpExp)exp;
            int res1 = interprete(cast.exp1);
            int res2 = interprete(cast.exp2);
            switch (cast.op) {
            case BinOp.Plus: return res1 + res2;
            ...
            default: throw new error("illegal operator.");
            }
       ...
    }
}
```