Grammaires non contextuelles

28

- ER : « notation » pour décrire les chaînes d'un langage
  => spécification d'un langage
- AF : « machines » pour reconnaître les chaînes d'un langage
  => reconnaissance d'un langage
- grammaires : règles de réécriture pour produire les chaînes d'un langage
  => génération d'un langage

29

Expressivité des grammaires

- Il existe différentes formes de grammaires qui décrivent des classes différentes de langages
- Rappel de la Classification de Chomsky :

Classes de langages	Types de machines	Types de grammaires
Réguliers	Automates finis	Type 3 : régulières
Non contextuels	Automates à pile	Type 2 : non contextuelles
Contextuels		Type 1 : contextuelles
Récursivement énumérables	Machines de Turing	Type 0 : sans restriction

30

Ce que peut exprimer une grammaire non contextuelle et que ne peuvent pas exprimer les ER ou les AF :
la structure récursive des phrases.

Exemple :
si I1 et I2 sont des instructions
et E est une expression
alors si E alors I1 sinon I2 est une instruction

Inst → si Expr alors Instr sinon Instr

=> Les grammaires vont être parfaitement adaptées pour exprimer la syntaxe des langages de programmation

31

Présentation informelle

Les grammaires permettent d'exprimer des définitions récursives

Exemple 1 : $L = \{a^n b^n \mid n \in N\}$
Déf. récursive : base : $\varepsilon \in L$
récur : si $w \in L$ ($w = a^k b^k$)
alors $awb \in L$ ($awb = a^{k+1}b^{k+1}$)

Grammaire correspondante :
(1) ⟨mot⟩ → ε
(2) ⟨mot⟩ → a ⟨mot⟩ b

32

Exemple 2 : expressions arithmétiques (EA)

formées avec des constantes numériques, les opérateurs + et *, et des parenthèses

- Définition récursive:

base : une constante numérique est une EA
récur : si e1 et e2 sont des EA
alors e1 + e2
e1 * e2
(e1) sont des EA

- Grammaire non contextuelle correspondante :
  (1) ⟨expr⟩ → const
  (2) ⟨expr⟩ → ⟨expr⟩ + ⟨expr⟩
  (3) ⟨expr⟩ → ⟨expr⟩ * ⟨expr⟩
  (4) ⟨expr⟩ → (⟨expr⟩)

33

Définition

Une grammaire non contextuelle (GNC) est un quadruplet $G = (V_T, V_N, S, P)$ où

- V_T est un ensemble fini de symboles terminaux (ou alphabet)
- V_N est un ensemble fini de symboles non terminaux, ou catégories syntaxiques
- $S \in V_N$ est le symbole non terminal initial (start), ou axiome de la grammaire
- P est un ensemble de productions de la forme $A \rightarrow \alpha$ avec $A \in V_N$ et $\alpha \in (V_T \cup V_N)^*$

Abréviation : Un ensemble de règles ayant même tête : $A \rightarrow \alpha_1, \ldots, A \rightarrow \alpha_n$ peut s'écrire $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$

Exemple : affectation (simplifiée)

P = { Inst → ident ':=' Expr
Expr → ident | nbr | Expr + Expr }

V_T = {ident, nbr, :=, +}

V_N = {Inst, Expr}

S = Inst

Inst ⇒ ident ':=' Expr
⇒ ident ':=' Expr + Expr
⇒ ident ':=' nbr + Expr
⇒ ident ':=' nbr + nbr

Langage défini par une grammaire

- C'est l'ensemble des chaînes que l'on peut obtenir à partir du symbole initial en appliquant les règles
- Principe
  - On part du symbole initial (la chaîne en construction = S)
  - On applique une règle : on remplace la partie gauche de la règle (sa tête) par sa partie droite (son corps) dans la chaîne en construction
  - On continue à appliquer des règles jusqu'à ce que la chaîne ne contienne plus que des symboles terminaux
- Ce processus consistant à appliquer des règles pour construire des chaînes s'appelle dérivation

Dérivations

- Si $A \rightarrow \beta$ est une production alors on dit que $\alpha_1 A \alpha_2$ se dérive en 1 étape en $\alpha_1 \beta \alpha_2$ ce que l'on note $\alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$
- On étend la relation $\Rightarrow$ pour exprimer les dérivations en zéro ou plusieurs étapes, ce que l'on note $\Rightarrow^*$

$\alpha \Rightarrow^* \alpha'$
se lit
« α se dérive en 0, 1 ou ++ étapes en α'»

Phrases et Langages

- Une chaîne qui peut être dérivée à partir de S est appelée:
  - une phrase si elle est constituée uniquement de symboles terminaux
  - Une protophrase si elle contient au moins un symbole non terminal

Soit $G = (V_T, V_N, S, P)$ une GNC

- Le langage engendré par G, noté L(G), est l'ensemble des chaînes de symboles terminaux ω qui peuvent être dérivées à partir de S :

$$L(G) = \{\omega \in V_T^* / S \Rightarrow^* \omega\}$$

Stratégies de dérivation

À chaque étape de dérivation, il faut faire 2 choix :

- Quel symbole NT remplacer ?
- Une fois choisi le symbole NT, quelle production utiliser ?

Une dérivation gauche (resp. droite) est une dérivation où, à chaque étape de dérivation, c'est le symbole non terminal le plus à gauche (resp. droite) qui est remplacé

Si α se dérive en β par une dérivation gauche (resp. droite), on le note $\alpha \Rightarrow_g^* \beta$ (resp. $\alpha \Rightarrow_d^* \beta$)

Important : La stratégie de dérivation ne change pas les phrases que l'on peut produire

Arbre d'analyse ou Arbre de dérivation

- C'est une représentation graphique d'une dérivation où on ne précise pas l'ordre de remplacement des symboles non terminaux
- Soit $G = (V_T, V_N, S, P)$ une GNC,
  un arbre d'analyse pour G est tel que :
  - La racine est le symbole initial S
  - Chaque feuille est soit ε, soit un symbole terminal
  - Chaque nœud interne est un symbole non terminal X et ses fils sont les symboles de la partie droite d'une règle $X \rightarrow X_1X_2 \ldots X_n$

  Les feuilles de l'arbre, lues de gauche à droite, forment une phrase ω de L(G)

40

- L'ordre de remplacement des symboles NT n'étant pas précisé, un arbre d'analyse peut représenter plusieurs dérivations possibles d'une même chaîne
- Par contre, il représente une unique dérivation gauche (ou droite)

=> Un arbre d'analyse est équivalent à une dérivation gauche :

> Un arbre $\rightarrow$ une dérivation gauche unique
> Une dérivation gauche $\rightarrow$ un arbre unique

Mais il peut arriver qu'une chaîne soit obtenue par ++ dérivations gauches (avoir ++ arbres d'analyse), dans ce cas, la chaîne est dite ambiguë

41

Les formulations suivantes sont équivalentes :

- ω est engendré par G : $\omega \in L(G)$
- Il existe une dérivation pour ω : $S \Rightarrow^* \omega$
- Il existe une dérivation gauche pour ω : $S \Rightarrow_g^* \omega$
- Il existe un arbre d'analyse pour ω

42

Langages non contextuels

- Les langages générés par les GNC sont appelés les langages non contextuels

43

exercice

Pour chacun des langages suivants, donner une grammaire et dire à quelle classe appartient le langage (régulier, non contextuel, ou au delà) :

1. a^*b^*
2. $\{\omega \in \{a,b\}^* / |\omega|_a \bmod 2 = 0\}$
3. $\{\omega\omega^R / \omega \in \{a,b\}^*\}$
4. $\{a^n b^p c^p d^n / n,p \geq 1\}$
5. $\{\omega\omega / \omega \in \{a,b\}^*\}$

44

Construction d'une GNC à partir d'un AF

Soit $M = (\Sigma, Q, q_0, F, \delta)$ un AFD (ou un AFN)
On construit $G = (V_T, V_N, S, P)$ une GNC
telle que L(G)=L(M) :

- $V_T = \Sigma$
- $V_N = Q$
- $S = q_0$
- P est constitué des productions :

  $q_i \rightarrow a\ q_j$ dès que $\delta(q_i,a) = q_j$

  et $q_i \rightarrow \varepsilon$ si $q_i \in F$

45

Langages réguliers et langages non contextuels

- Pour tout AF, on peut construire une GNC qui génère le même langage
  => Les GNC sont au moins aussi puissantes que les AF
- Mais sont-elles plus puissantes que les AF ?
  => Oui, car il existe des langages non contextuels (pour lesquels il existe une GNC) qui ne sont pas réguliers (pour lesquels il n'existe pas d'ER ou d'AF)
  Exemple : $L = \{a^n b^n \mid n \in N\}$
  il existe une GNC pour L, mais il n'existe pas d'ER ni d'AF

=> Les langages réguliers sont un sous-ensemble strict des langages non contextuels

46

Les différentes formes de grammaires

- Les grammaires régulières (type 3) ont des règles de la forme $A \rightarrow aB$ ou $A \rightarrow a$ ou $A \rightarrow \varepsilon$
- Les grammaires non contextuelles (type 2) ont des règles de la forme $A \rightarrow \alpha$
- Les grammaires contextuelles (type 1) ont des règles de la forme $\alpha \rightarrow \beta$ avec $\alpha \neq \varepsilon$ et $|\alpha| \leq |\beta|$
- Les grammaires contextuelles sans restriction (type 0) ont des règles de la forme $\alpha \rightarrow \beta$ avec $\alpha \neq \varepsilon$

47

Analyse syntaxique

2nde phase de la compilation : reconnaissance des « phrases »

Exemple de règles exprimant la syntaxe d'un langage de programmation :

Prog: program ident;
  Liste_déclarations
  begin Suite_instructions end.
Déclaration:
  Déclaration_procédure
  ou Déclaration_fonction
  ou Déclaration_variable
Instruction:
  Instruction_affectation
  ou Instruction_conditionnelle
  ou Instruction_tant_que
  ou begin Suite_instructions end
Etc…

49

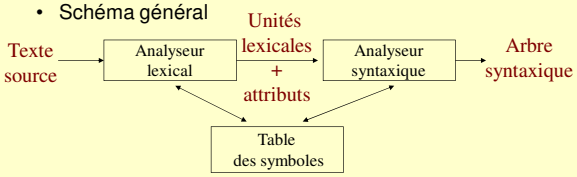
Syntaxe et grammaires

- La syntaxe d'un programme peut être décrite par une grammaire non contextuelle
- Avantages :
  - Spécification formelle précise, et facile à comprendre
  - Possibilité (souvent, mais pas toujours) de construire automatiquement un analyseur syntaxique efficace
  - Met en évidence la structure du langage, utile pour la traduction en du code correct et pour la détection d'erreurs
  - Peut facilement être complétée quand le langage évolue
- Attention : certaines choses ne peuvent pas être exprimées avec les GNC, par ex. qu'un identificateur doit être déclaré avant d'être utilisé.

50

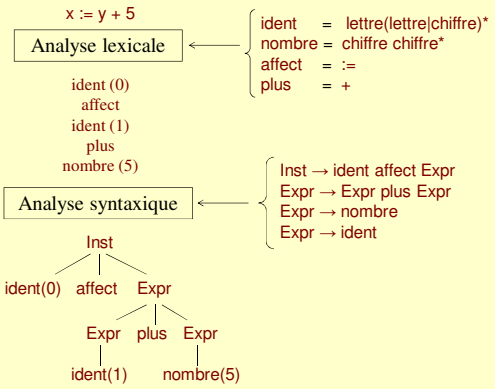
L'analyseur syntaxique

- Schéma général



- Ses tâches
  - Vérifier que la suite d'unités lexicales fournie par l'analyseur lexical peut être engendrée par la grammaire
  - Produire l'arbre d'analyse qui spécifie la structure grammaticale du programme
  - Signaler les erreurs

51



3 types généraux d'analyseurs syntaxiques

- Problématique :
  - On a : - une suite d'unités lexicales ω
    - une grammaire G
  - But : déterminer si ω peut être engendré par G et construire l'arbre d'analyse
- Méthodes de construction de l'arbre : 3 types
  - Méthodes universelles (tous types de grammaires NC) Mais inefficaces
  - Méthodes descendantes : construction de l'arbre de haut en bas => utilisées pour implémentation manuelle d'AS
  - Méthodes ascendantes : construction de l'arbre de bas en haut => utilisées par les outils de construction automatique d'AS

Ces 2 derniers types de méthodes ne fonctionnent que sur des sous-classes de grammaires

grammaires non contextuelles et analyse syntaxique

- Une grammaire NC : G = (V_T, V_N, S, P) où
  - V_T : vocabulaire terminal (alphabet) ici, V_T est **l'ensemble des UL retournées par l'AL**
  - V_N : vocabulaire non terminal
  - S : axiome de la grammaire ici, S **engendre tous les programmes bien formés** du langage
  - P : ensemble de règles de la forme $X \rightarrow \alpha$ où $X \in V_N$ et $\alpha \in (V_T \cup V_N)^*$
- On va voir quelques propriétés et quelques transformations de grammaires utiles pour l'analyse syntaxique

Grammaires ambiguës

- Une grammaire est ambiguë s'il existe $\omega \in L(G)$ qui admet plusieurs arbres d'analyse
- Problème pour l'AS : l'arbre, en donnant la structure d'une phrase, donne des indications sur sa signification. Une phrase qui admet plusieurs arbres va donc avoir plusieurs significations possibles.
- Difficultés :
  - il n'existe pas d'algo général pour déterminer si une grammaire est ambiguë ou non
  - Il n'existe pas d'algo général pour supprimer les ambiguïtés
  - Pire, certains langages NC n'admettent que des GNC ambiguës
- Rem : Yacc accepte des grammaires ambiguës (On ajoute des règles pour lever les ambiguïtés).
- Mais, d'une façon générale, les grammaires ambiguës sont proscrites.

Suppression des ambiguïtés

- On peut souvent réécrire une grammaire ambiguë pour éliminer les ambiguïtés, mais il n'existe pas de méthode systématique
- Exemple : expressions arithmétiques
  - On veut :
    - Priorité de * sur +
    - Associativité à gauche des opérateurs
  - On réécrit la grammaire de façon à
    - Dériver en 1er les opérateurs de plus faible priorité
    - Avoir toujours, à droite d'un opérateur, une expression de priorité supérieure

Récursivité à gauche

- Une grammaire est récursive à gauche si $\exists A \in V_N$ t.q. $\exists$dérivation $A \Rightarrow^+ A\alpha$ où $\alpha \in (V_N \cup V_T)^*$
- Les méthodes d'analyse descendante ne fonctionnent pas avec ce type de grammaires
- Il existe un algo général pour éliminer la récursivité à gauche

Suppression de la récursivité directe

Récursivité directe : on a des règles de la forme

$A \rightarrow A\alpha \mid \beta$

On peut les transformer en :

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

- Cas général (en supposant qu'aucun des α_i n'est ε)
  - Grouper toutes les A-productions comme suit :
    $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_p$
    où les β_i ne commencent pas par A
  - Remplacer ces A-productions par :
    $A \rightarrow \beta_1 A' \mid \ldots \mid \beta_p A'$
    $A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$

58

Suppression de toutes les récursivités à gauche

Récursivité indirecte : plusieurs étapes de dérivations sont nécessaires pour obtenir $A \Rightarrow^+ A\alpha$

- Il existe un algo pour éliminer toutes les récursivités à gauche d'une grammaire (non étudié ici)

59

Factorisation à gauche

Autre transformation des grammaires utile pour obtenir des grammaires adaptées à l'analyse descendante

- **Algo** : soit une grammaire G
  - Pour chaque non terminal A de G
    - Trouver le plus long préfixe α commun à plusieurs alternatives
    - Si $\alpha \neq \varepsilon$
      alors Remplacer toutes les A-productions
      $A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma$
      où γ représente toutes les alternatives ne commençant pas par α
      Par $A \rightarrow \alpha A' \mid \gamma$
      $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
      où A' est un nouveau NT
  - On répète cette opération jusqu'à ce qu'aucune des alternatives d'un même NT n'ait de préfixe commun

60

Symboles effaçables

- X est effaçable ssi $X \Rightarrow^* \varepsilon$
- **Algo** pour déterminer les symboles effaçables d'une grammaire G :
  - S'il y a une règle $X \rightarrow \varepsilon$
    alors X est effaçable
  - S'il y a une règle $X \rightarrow Y_1 Y_2 \ldots Y_n$
    et tous les Y_i sont effaçables
    alors X est effaçable
  - On répète l'opération jusqu'à ce qu'on ne puisse plus rien ajouter
- Rem : évidemment, ε est effaçable !

61

Grammaires propres

- Une grammaire propre est une grammaire dans laquelle tout symbole de $V_N \cup V_T$ apparaît dans au moins une dérivation d'une phrase du langage

$\Rightarrow$ Interdit tout symbole inaccessible ou improductif

- Un symbole est inaccessible s'il ne figure dans aucune protophrase de G
- Un symbole A est improductif s'il n'existe pas de dérivation $A \Rightarrow^* \alpha$ avec $\alpha \in V_T^*$

62