Compilation

CM1 - Introduction & Analyse syntaxique

ISTIC, Université de Rennes 1
Sebastien.Ferre@irisa.fr

COMP, M1 info

Plan

Plan

Pourquoi étudier la compilation ?

- Un compilateur utilise une grande quantité de techniques
  - de *beaux algorithmes*, utilisés dans de nombreux autres contextes
- Résultat d'une réflexion de 60 ans
  - dans le sens de toujours plus d'*automatisation*
  - un problème crucial et bien posé
    - *«traduire des programmes d'un langage dans un autre en préservant la sémantique et en étant le plus efficace possible»*
  - les compilateurs sont des programmes très complexes et en même temps très sûrs
  - rôle important de la *formalisation*

Pourquoi étudier la compilation ?

- Un compilateur utilise une grande quantité de techniques
  - de *beaux algorithmes*, utilisés dans de nombreux autres contextes
- Résultat d'une réflexion de 60 ans
  - dans le sens de toujours plus d'*automatisation*
  - un problème crucial et bien posé
    - *«traduire des programmes d'un langage dans un autre en préservant la sémantique et en étant le plus efficace possible»*
  - les compilateurs sont des programmes très complexes et en même temps très sûrs
  - rôle important de la *formalisation*

Pourquoi étudier la compilation ?

- Approcher l'intimité des langages de programmation
  - mieux les comprendre pour mieux programmer ?
  - percevoir les limites de ce qui est calculable
    - délimitation de classes de problèmes et leurs solutions
- La jonction entre le génie logiciel et le système
  - le système est responsable du chargement des programmes (ex., édition des liens)
  - le compilateur est responsable de la production de programmes chargeables (ex., tables de symboles)
- Ne concerne pas que les *grands* langages de programmation universels
  - nombreux *petits* langages de script ou d'échange de données (ex., XML)
  - langages de bases de données (ex., SQL)
  - les mêmes techniques sont à l'œuvre !

Pourquoi étudier la compilation ?

- Approcher l'intimité des langages de programmation
  - mieux les comprendre pour mieux programmer ?
  - percevoir les limites de ce qui est calculable
    - délimitation de classes de problèmes et leurs solutions
- La jonction entre le génie logiciel et le système
  - le système est responsable du chargement des programmes (ex., édition des liens)
  - le compilateur est responsable de la production de programmes chargeables (ex., tables de symboles)
- Ne concerne pas que les *grands* langages de programmation universels
  - nombreux *petits* langages de script ou d'échange de données (ex., XML)
  - langages de bases de données (ex., SQL)
  - les mêmes techniques sont à l'œuvre !

Pourquoi étudier la compilation ?

- Approcher l'intimité des langages de programmation
  - mieux les comprendre pour mieux programmer ?
  - percevoir les limites de ce qui est calculable
    - délimitation de classes de problèmes et leurs solutions
- La jonction entre le génie logiciel et le système
  - le système est responsable du chargement des programmes (ex., édition des liens)
  - le compilateur est responsable de la production de programmes chargeables (ex., tables de symboles)
- Ne concerne pas que les *grands* langages de programmation universels
  - nombreux *petits* langages de script ou d'échange de données (ex., XML)
  - langages de bases de données (ex., SQL)
  - les mêmes techniques sont à l'œuvre !

Ce que fait un compilateur

1. Analyse du programme source
   - trouver la structure du programme
   - signaler les éventuelles erreurs de syntaxe
   - formalisation par des règles de grammaire
     - structure = arbre de dérivation
   - dépasse largement le cadre de la compilation
     - concerne toute application qui lit des entrées formatées selon une grammaire
2. Production de sa traduction
   - dans un langage cible
   - avec préservation de la sémantique (impératif)
   - et efficacité du code produit (autant que possible)

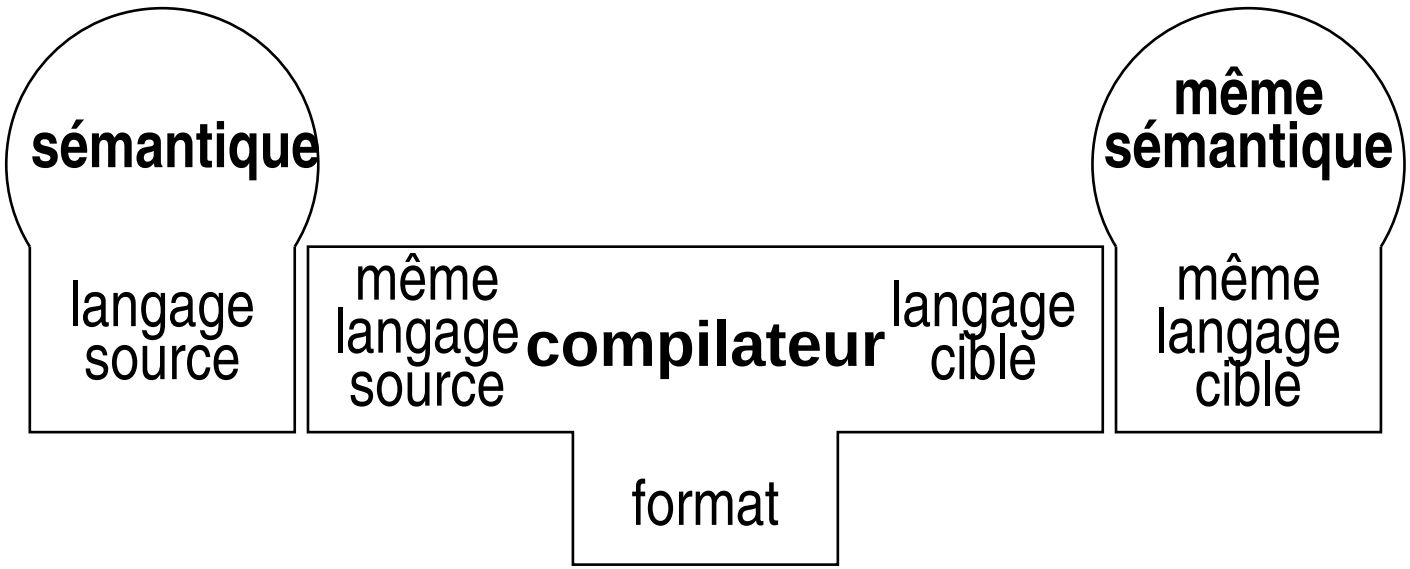
Donnée = Programme

Les données sont des programmes, et réciproquement

- point particulier des compilateurs
  - aussi à l'oeuvre dans les systèmes d'exploitation (gestion des processus)
- distinguer le programme «textuel» et le programme «en actes» (fonction)
- sémantique = lien entre le texte et la fonction

Diagramme en T

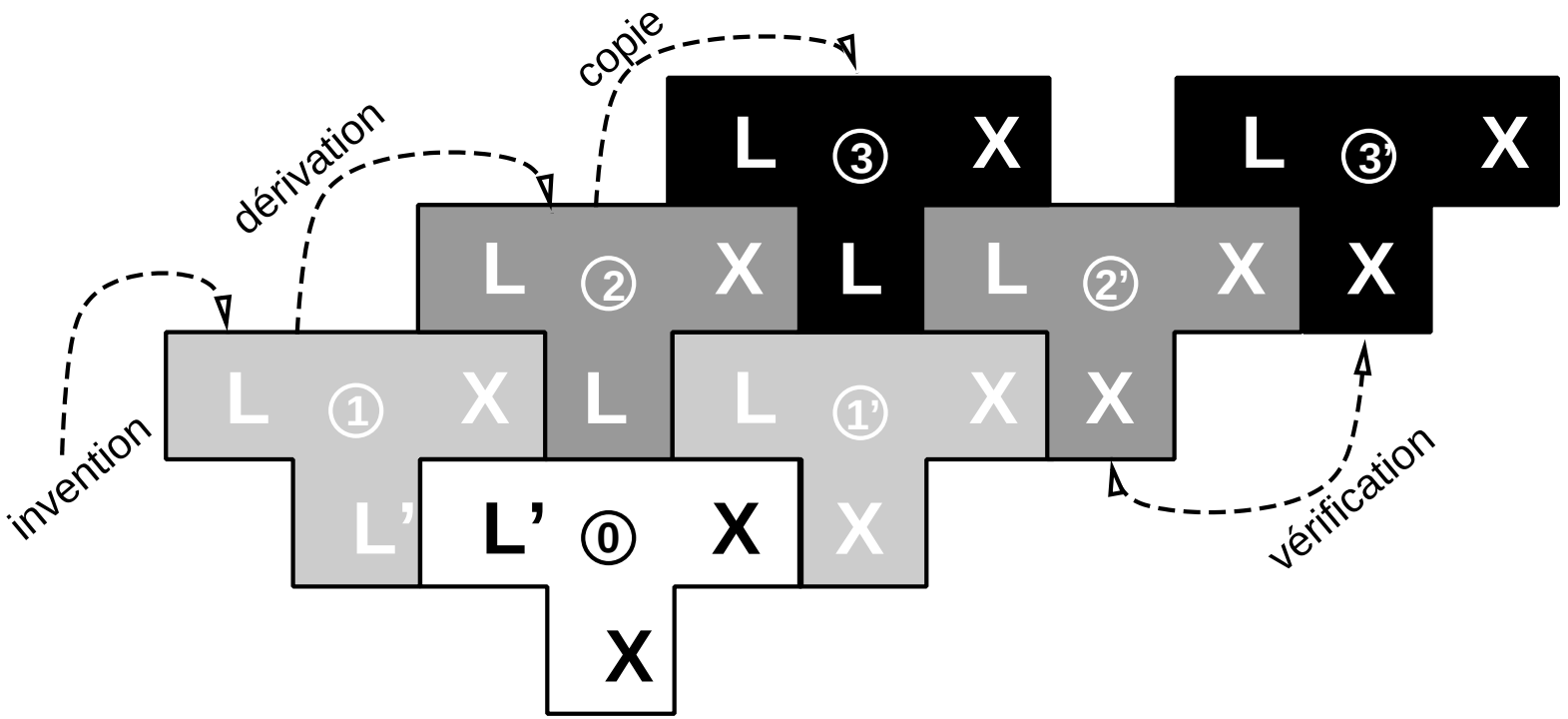
Relation entre compilateur, source et cible



Auto-compilation (*bootstraping*)

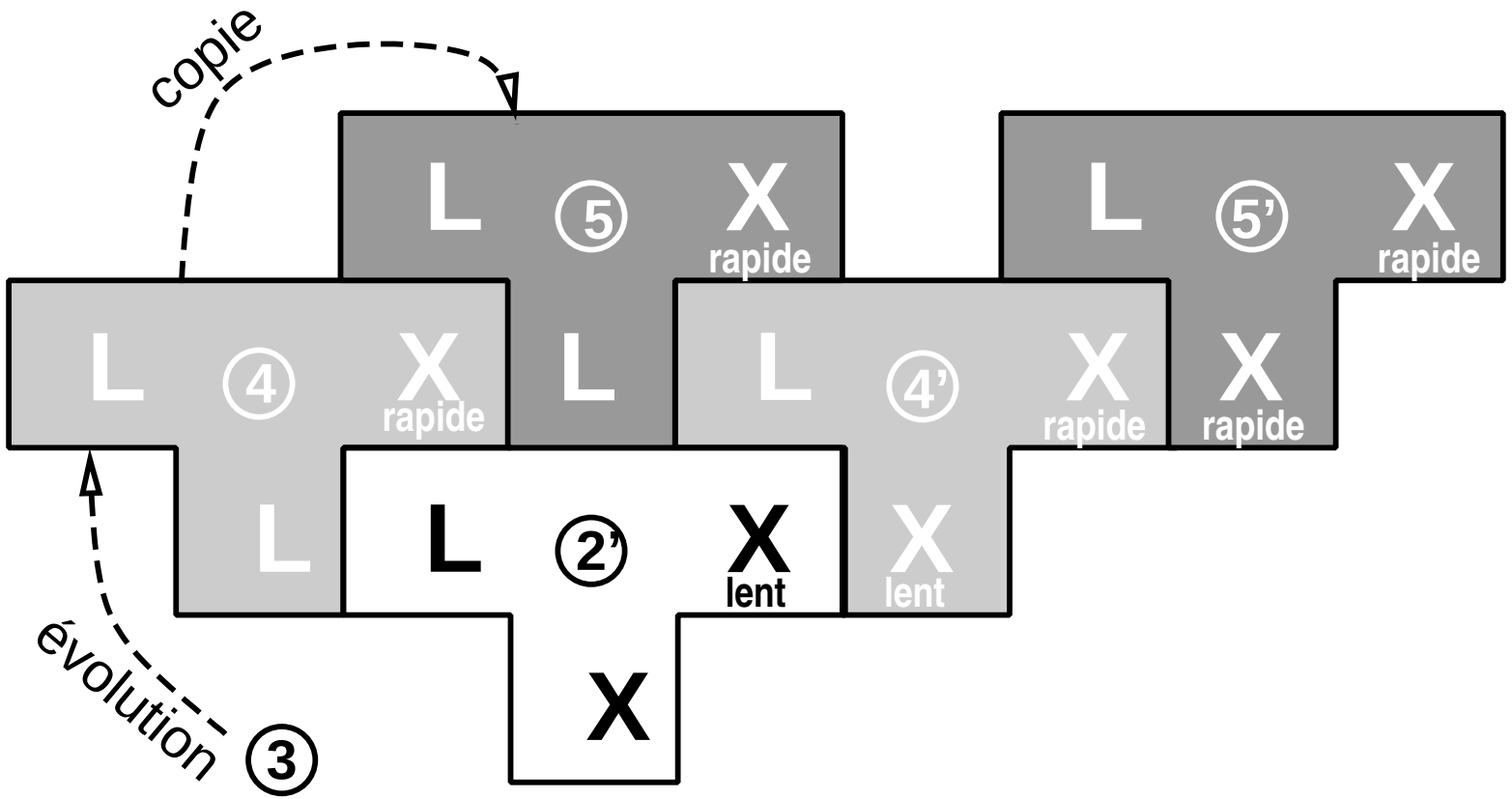
On souhaite écrire le compilateur dans son langage source.

- ex., écrire un compilateur Java en Java !
  L = Java, L' = C, X = langage machine éxécutable



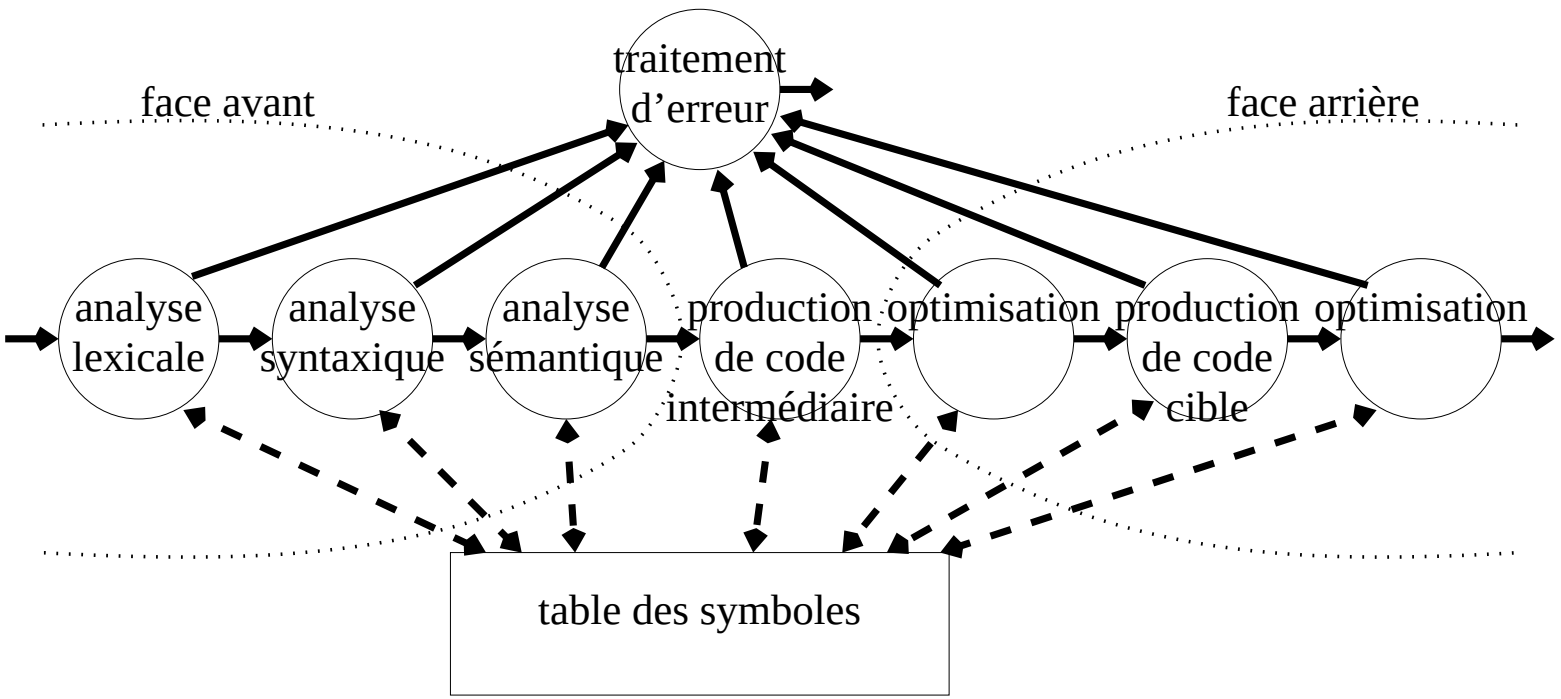
Évolution dans un schéma d'auto-compilation

On souhaite améliorer le compilateur (code plus rapide).



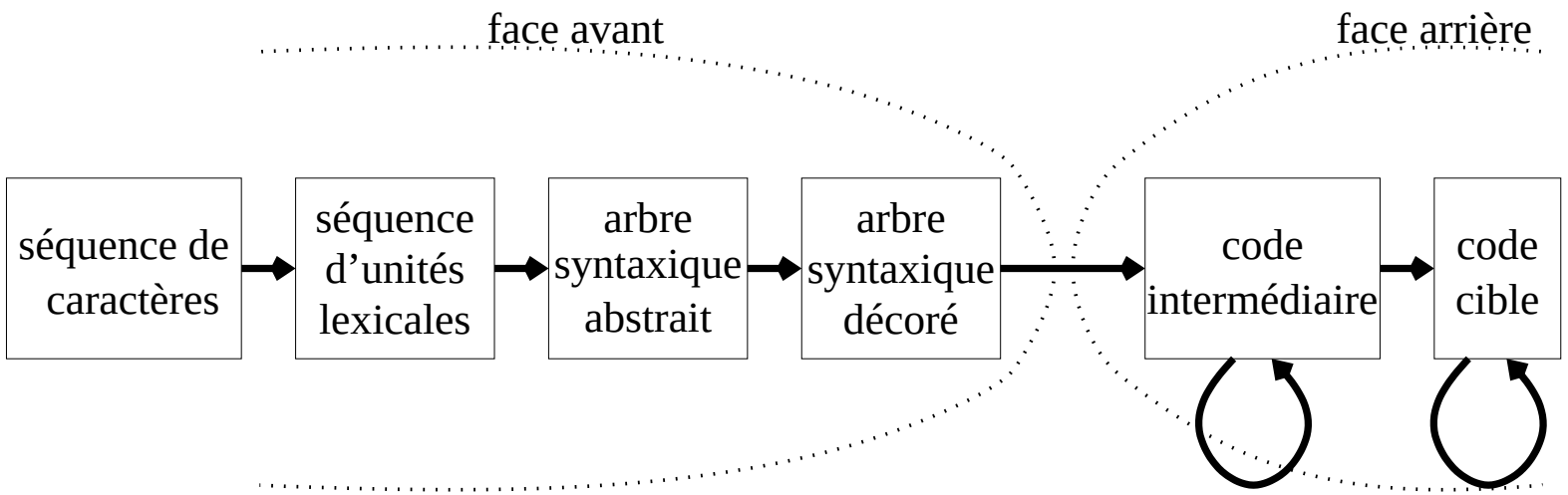
Les phases de la compilation

- elles forment un pipeline
- pas nécessairement en séquence stricte



Les représentations du programme

La représentation du programme évolue d'une phase à l'autre.



Analyses lexicales et syntaxiques

- Analyse lexicale
  - des lettres aux mots
  - langages formels de type 3
  - expressions régulières et automates finis
- Analyse syntaxique
  - des mots aux phrases (structure du programme)
  - langages formels de type 2
  - grammaires algébriques et automates à pile
  - + vérifications contextuelles (hors grammaire)
    - ex. : variables déclarées avant d'être utilisées

Analyses sémantiques

- détection des erreurs non-syntaxiques
- analogie : vérification des accords en nombre et en genre en français
- essentiellement : analyse de types
  - vérification vs inférence
  - typage *fort* vs *faible*
  - défini formellement dans les langages de type ML (ex., Objective Caml)

Production de code intermédiaire

- entre face avant et face arrière
- format relativement indépendant des langages source *et* cible
  - réutilisation de la face avant pour un autre langage cible (ex., une autre architecture)
  - réutilisation de la face arrière pour un autre langage source
  - factorisation des optimisations indépendantes de la cible

Optimisations et production de code

- la production du code intermédiaire est locale
  - traduction de chaque trait du langage source
  - compositionalité
  - code inefficace
- les optimisations ont souvent besoin d'une vision globale
  - ex. : savoir si une variable est utilisée après tel point de programme
  - techniques : analyse de flots de données, interprétation abstraite
- production du code cible
  - très lié à l'architecture (code binaire)
  - et au système d'exploitation (format des exécutables)

Structure du cours

- 8 CMs et 8 TDs
  1. Rappels sur les langages formels
  2. Analyse syntaxique descendante
     - rapidement : ascendante et tabulée
  3. Grammaires attribuées
  4. Vérification de types
  5. Génération de code (expressions et instructions)
  6. Génération de code (structures de données et adressage)
  7. Optimisations (bref)
  8. Analyse de flots de données

Structure du cours

- 1 DM (devoir à la maison)
  - en binôme, 1/4 de la note de CC
  - approfondissement du cours et préparation des TPs
- 8 TPs
  - TP1 (2 séances) : conversion de RDF/Turtle en RDF/Ntriples (1/4 note CC)
  - TP2 (6 séances) : réalisation d'un compilateur de VSL+ (1/2 note CC)
    - face avant : vérification de type et génération de code 3 adresses
    - face arrière fournie

Environnement pour les TPs

2 propositions, au choix :

1. Java + ANTLR + face arrière "maison"
   - ANTLR = ANother Tool for Language Recognition
   - librairies Java "maison" pour la représentation intermédiaire et la production de code cible
2. OCaml + stream parsers + LLVM
   - suppose une connaissance de OCaml (enseigné en L3)
   - face arrière : LLVM = Low-Level Virtual Machine
   - suppose plus d'autonomie car ne colle pas exactement au cours (mais il y a un très bon tutoriel pour OCaml)

À choisir dès le DM, mais changement encore possible au TP1.

Plan

Analyse syntaxique

- 1ère phase de la compilation
- fonction : *vérification de l'appartenance du programme source au langage source*
  - sinon, traduire le programme n'a pas de sens
- cela suppose de pouvoir spécifier la syntaxe concrète du langage source

  $\rightarrow$ théorie des langages formels (phase très bien formalisée)

Théorie des langages formels

fondée sur les correspondances entre 3 entités :

- les langages ($\mathcal{L}$)
- les grammaires (G)
- les automates (M)

et entre 3 questions portant sur un mot m :

- m appartient à un langage $\mathcal{L}$?
- m est engendré par une grammaire G ?
- m est reconnu par un automate M ?

Rôle des grammaires

Comment spécifier un langage $\mathcal{L}$ (un ens. de mots) ?

- si $\mathcal{L}$ est fini : possible par énumération
- si $\mathcal{L}$ est infini : possible par un ensemble de règles syntaxiques
  - Phrase $\rightarrow$ Sujet Verbe Complément
  - Expression $\rightarrow$ Expression + Expression

Definition (grammaire)

Une grammaire G

- est un ensemble de règles syntaxiques
- engendre un langage $\mathcal{L}(G)$

Une grammaire G engendre un unique langage $\mathcal{L}(G)$, mais plusieurs grammaires peuvent engendrer un même langage $\mathcal{L}$.

Familles de grammaires

- il existe plusieurs formalismes/familles de grammaires
- qui engendrent différentes familles de langages
- imbriquées les unes dans les autres selon la hiérarchie de Chomsky
  - type 3 : grammaires/langages réguliers ou rationnels (*regular*)
  - type 2 : grammaires/langages hors-contextes ou sans contexte (*context-free*)
  - type 1 : grammaires/langages contextuels ou avec contexte (*context-sensitive*)
  - type 0 : grammaires/langages récursivement énumérables (*recursively enumerable*)
- ATTENTION : Une grammaire G_2 de type 2 peut engendrer un langage de type 3, si il existe une grammaire G_3 de type 3 qui engendre le même langage ($\mathcal{L}(G_3) = \mathcal{L}(G_2)$)

Rôle des automates

La décision $m \in \mathcal{L}(G)$ peut être automatisée.

Definition

On dit qu'un automate M reconnaît un langage $\mathcal{L}$ s'il reconnaît tous les mots du langage et aucun autre.

- $\mathcal{L}(M)$ dénote la langage reconnu par un automate M
- il existe une hiérarchie de formalismes/familles d'automates parallèle à celle des grammaires/langages
  - type 3 : automates à nombre fini d'états (*finite state automaton*)
  - type 2 : automates à pile (*pushdown automaton*)
  - type 1 : automates linéairement bornés (*linear bounded automaton*)
  - type 0 : machine de Turing (*Turing machine*)

Articulation entre langage, grammaire et automate

Le jeu en compilation consiste à

1. spécifier un langage $\mathcal{L}$ par une grammaire G
   - c-à-d. tel que $\mathcal{L}(G) = \mathcal{L}$
   - le langage et sa grammaire peuvent aussi être conçus de concert
2. dériver de la grammaire G un automate M qui reconnaisse le même langage
   - c-à-d. $\mathcal{L}(M) = \mathcal{L}(G) = \mathcal{L}$
   - des outils permettent d'automatiser cette dérivation (traduction)
     - ex., Lex/Yacc, JavaCC, ANTLR (pour le type 2)
     - ce sont des compilateurs d'analyseurs syntaxiques

Plan

- Rappels et notations
  - alphabets
  - langages
- langages de type 3
  - expressions régulières
  - automates finis
- langages de type 2
  - grammaires hors-contexte
  - automates à pile
  - automates des items non-contextuels
  - analyse syntaxique descendante
  - analyse syntaxique ascendante (bref)
  - analyse syntaxique tabulée (bref)

Alphabet, symboles et mots

- alphabet V : ensemble fini de *symboles* (caractères ou unités lexicales)
- mot x sur V : une suite finie de symboles de l'alphabet V
  - $x = x_1 x_2 ... x_n$ pour un mot de longueur n
  - x_i est le i-ième symbole du mot
  - $|x|$ est la longueur du mot x
- mot vide ϵ : mot de longueur 0
- V^n : ensemble des mots sur V de longueur n
- $V^* = \bigcup_{n \geq 0} V^n$: tous les mots (finis) sur V
- $V^+ = V^* \setminus \epsilon$: tous les mots sauf le mot vide

Concaténation et découpage des mots

- $x.y$ ou xy : la concaténation des mots x et y
  - définition : $xy = x_1 ... x_n y_1 .. y_m$, avec $x \in V^n$ et $y \in V^m$
  - élément neutre ϵ : $\epsilon.x = x.\epsilon = x$
  - associativité : $x.(y.z) = (x.y).z = x.y.z = xyz$
  - non commutativité : $x.y \neq y.x$
- soit $w = xyz$ un mot sur V
  - x est un préfixe ou facteur gauche de w (propre si $yz \neq \epsilon$)
  - z est un suffixe ou facteur droit de w (propre si $xy \neq \epsilon$)
  - y est un sous-mot ou facteur de w (propre si $x \neq \epsilon \neq z$)

Langages formels

Un langage formel $\mathcal{L}$ est un ensemble de mot, donc un sous-ensemble de V^*.

- opérations ensemblistes
  - union : $\mathcal{L}_1 \cup \mathcal{L}_2$
  - intersection : $\mathcal{L}_1 \cap \mathcal{L}_2$
  - complémentation : $\overline{\mathcal{L}} = V^* \setminus \mathcal{L}$
- opérations rationnelles
  - produit : $\mathcal{L}_1\mathcal{L}_2 = \{xy \mid x \in \mathcal{L}_1, y \in \mathcal{L}_2\}$
  - répétition : $\mathcal{L}^n = \{x_1 ... x_n \mid \forall i \in 1..n : x_i \in \mathcal{L}\}$
  - fermeture : $\mathcal{L}^* = \bigcup_{n \geq 0} \mathcal{L}^n$

Plan

- langages de type 3
  - expressions régulières
  - automates finis
- langages de type 2
  - grammaires hors-contexte
  - automates à pile
  - automates des items non-contextuels
  - analyse syntaxique descendante
  - analyse syntaxique ascendante (bref)
  - analyse syntaxique tabulée (bref)

Langages de type 3/rationnels/réguliers

- particulièrement favorables à une mise en œuvre informatique
- peuvent être décrits par des expressions régulières, qui jouent ici le rôle de grammaire

Definition (Expression régulière)

$r \to \mathcal{L}(r)$ une expression régulière engendre un langage

- $\underline{\{\}} \to \{\}$ 0 mot
- $\underline{\epsilon} \to \{\epsilon\}$ 1 mot de 0 caractere
- $\underline{a} \to \{a\}$, pour tout $a \in V$ 1 mot de 1 caractère
- $\underline{(}r_1\underline{|}r_2\underline{)} \to \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$ union
- $\underline{(}r_1 r_2\underline{)} \to \mathcal{L}(r_1)\mathcal{L}(r_2)$ produit
- $\underline{(}r_1\underline{)^*} \to \mathcal{L}(r_1)^*$ fermeture

Automates finis

Le formalisme d'automates correspondant au type 3 est celui des automates finis

Definition (Automates finis)

$M = (Q, V, \Delta, q_0, F)$ est un automate fini où :

- V (fini) est l'alphabet d'entrée,
- Q est un ensemble fini d'états,
- q_0 est l'état initial,
- $F \subseteq Q$ est l'ensemble des états finaux,
- $\Delta \subset Q \times V \times Q$ est la relation de transition.

Fonctionnement d'un automate fini

analogie avec une machine

- automate fini : $M = (Q, V, \Delta, q_0, F)$ machine
- configuration : (q, w) avec $q \in Q$, $w \in V^*$ mémoire
  - initiale : (q_o, w) où w est le mot à reconnaître
  - finale : (q_f, ϵ) où $q_f \in F$
- transition entre 2 configurations : loi de fonctionnement
  $(q, aw) \rightarrow_M^a (p, w)$ ssi $(q, a, p) \in \Delta$
- suite de transitions :
  $(q, xw) \rightarrow_M^* (p, w)$
  $= (q, x_1 \dots x_n w) \rightarrow_M^{x_1} (p_1, x_2 \dots x_n w) \dots \rightarrow_M^{x_n} (p, w)$

Fonctionnement d'un automate fini

analogie avec une machine

- acceptation : M accepte (reconnait) un mot $w \in V^*$
  ssi $(q_o, w) \rightarrow_M^* (q_f, \epsilon)$ avec $q_f \in F$
  ssi il existe une suite de transitions de l'état initial vers un état final qui consomme le mot w

Definition (langage accepté)

$\mathcal{L}(M) = \{w \in V^* \mid (q_o, w) \rightarrow_M^* (q_f, \epsilon), q_f \in F\}$

Représentation par un graphe d'un automate fini

Rappel : graphe = noeuds + arcs

- état $\rightarrow$ noeud . . . . . .
  - état initial $\rightarrow$ noeud avec flèche entrante . . . . . .
  - état final $\rightarrow$ noeud doublement cerclé . . . . . .
- transition $\rightarrow$ arc étiqueté par le symbole lu . . . . . .

Example : . . . . . .

Lecture du graphe représentant un automate

- Un w-chemin est un chemin dans le graphe tel que w est la concaténation des étiquettes des arcs.
- Un mot w est reconnu s'il existe un w-chemin de l'état inital à un état final.
- Exemple : . . . . . .

Automates non-déterministes

L'exemple d'automate ci-dessus n'est pas déterministe :

- plusieurs transitions possibles dans certaines configurations : ex., (4,bw) ou (0,aw)
- besoin de représenter un ensemble de configurations possibles
- besoin de plus de mémoire

Déterminisation d'automates finis

Theorem

Tout automate M peut être déterminisé, càd, il existe un automate déterministe M' tel que $\mathcal{L}(M') = \mathcal{L}(M)$ (qui reconnait le même langage).

Bonne nouvelle : il existe un algorithme de déterminisation.

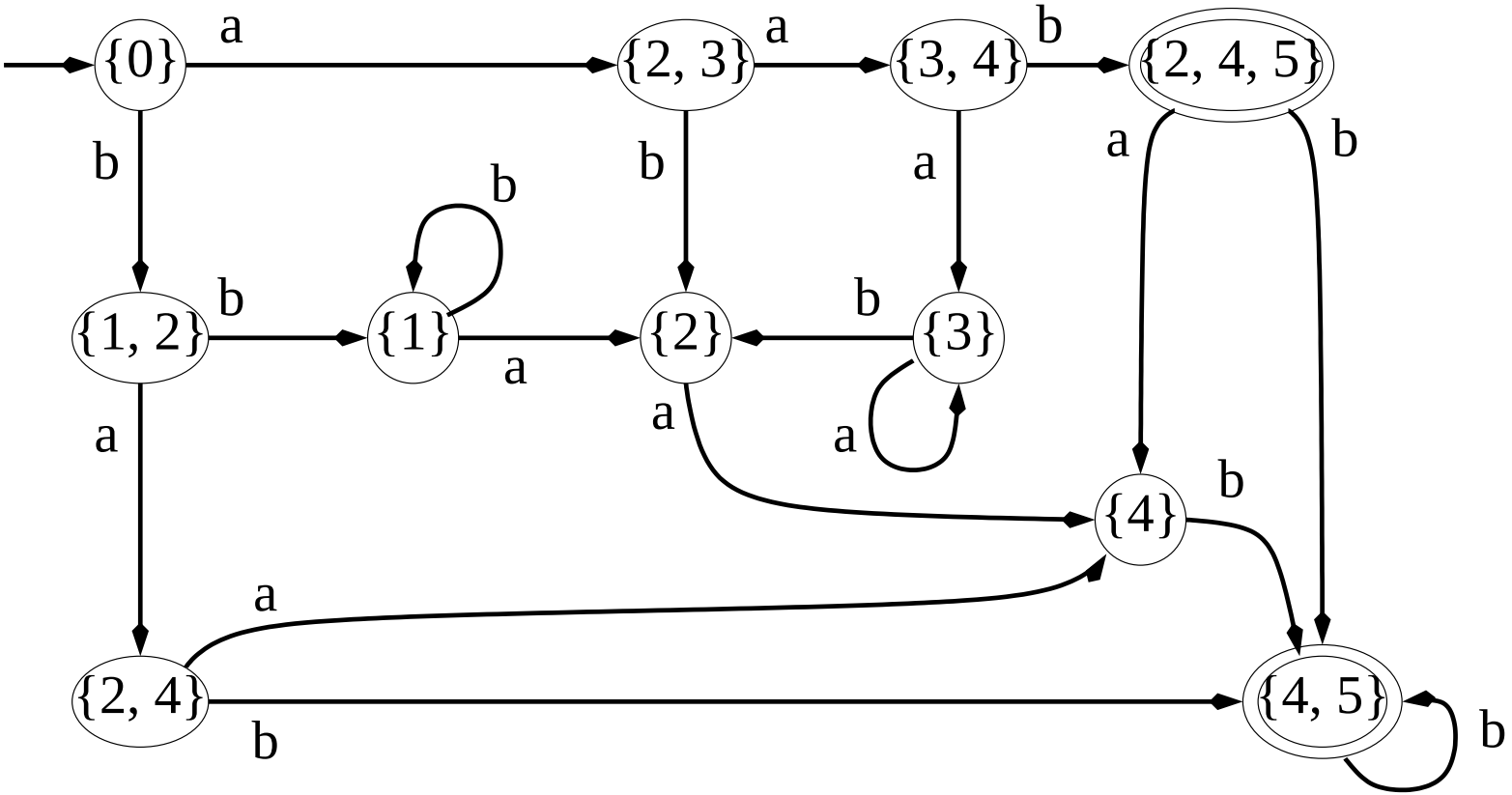
Algorithme de déterminisation d'automates

Connaissant un automate fini $M = \{Q, V, \Delta, q_0, F\}$, l'automate déterministe qui accepte le même langage est complètement caractérisé de la façon suivante :
$M^{det} = (Q', V, \delta, q'_0, F')$ avec

- $Q' \subset \mathcal{P}(Q)$,
- $q'_0 = \{q_0\}$,
- $F' = \{S \in Q' \mid S \cap F \neq \{\}\}$,
- $\delta(S, a) = \{p \mid (q, a, p) \in \Delta, q \in S\}$ pour tout $a \in V$.

La déterminisation peut entraîner une explosion du nombre d'états (exponentiel dans le pire cas).

Automate déterminisé



Limites et usages des langages rationnels

Ils ne rendent pas compte des structures imbriquées :

- ex : parenthésage des expressions
- ex : blocs imbriqués
- ex : langage $\{a^n b^n \mid n \in \mathbb{N}\}$

En conséquence

- pas adaptés aux langages de programmation
  $\Rightarrow$ langages de type 2
- mais adapté à la description des éléments de ces langages
  - ex : identificateurs, nombres, commentaires
  - $\Rightarrow$ utilisé pour l'analyse lexicale (découpage de la chaine d'entrée en suite d'unités lexicales)

Langages de type 2

Definition

Les langages de type 2 sont ceux engendrés par les grammaires hors-contexte.

Grammaire hors-contexte

Definition (Grammaire hors-contexte)

(V_N, V_T, P, S) est une grammaire hors-contexte (ou sans contexte, ou non contextuelle) où,

- V_T : alphabet de symboles terminaux
- V_N : alphabet de symboles non-terminaux
- $S \in V_N$: symbole de départ ou axiome
- $P \subset V_N \times (V_N \cup V_T)^*$: ensemble des productions (ou règles)

Exemple : . . . . . .

Dérivations

- Les productions servent de règles de réécritures
- Si $(A \to \alpha) \in P$ alors $\sigma A \tau \Rightarrow_G \sigma \alpha \tau$
- $\Rightarrow_G^*$ est la fermeture réflexive et transitive de $\Rightarrow_G$

Example : . . . . . .

Langage engendré

Definition (langage engendré)

$\mathcal{L}(G) = \{u \in V_T^* \mid S \Rightarrow_G^* u\}$

- $u \in V_T^*$ est une phrase
  - ne contient que des terminaux
  - plus de dérivations possibles
  - phrase de G si $S \Rightarrow_G^* u$
- $\alpha \in (V_T \cup V_N)^*$ est une proto-phrase
  - peut comporter des non-terminaux
  - des dérivations supplémentaires sont possibles
  - proto-phrase de G si $S \Rightarrow_G^* \alpha$

Productions récursives

Productions de la forme

- $A \to \alpha' A \alpha''$ — récursive directe
- $A \to A\alpha''$ — récursive (directe) gauche
- $A \to \alpha' A$ — récursive (directe) droite

Sinon si $A \Rightarrow_G^* \alpha' A \alpha''$ — récursivité indirecte

Example : . . . . . .

Arbres de dérivation

Definition (Arbre de dérivation)

C'est la trace d'une dérivation dans laquelle on a "oublié" l'ordre des dérivations élémentaires

- noeud A :
  - une occurence de dérivation pour une production $A \to \alpha$
  - chaque α_i est la racine d'un sous-arbre ou une feuille
- feuille a :
  - un terminal pour une phrase
  - un terminal OU un non-terminal pour une proto-phrase

Exemple : . . . . . .

Amiguite

Definition (ambiguité)

1. une phrase est ambigüe si elle admet plusieurs arbres de dérivation
2. une grammaire est ambigüe si elle engendre au moins une phrase ambigüe

Exemple : . . . . . .

Ambiguité

- L'ambiguité est une faiblesse en informatique et doit être évitée
  - responsabilité du concepteur de la grammaire
    propriété non-décidable des grammaires
  - possibilité de modifier la grammaire
    - sans changer le langage engendré
    - mais en restreignant les analyses possibles
- Exemple : associativité gauche du $-$ . . . . . .

le choix des "bons arbres" dépend de la sémantique du langage

Automates des langages de type 2

- Le formalisme d'automate correspondant au type 2 est celui des automates à pile.
- Ils vont permettre d'automatiser la reconnaissance d'un mot en se laissant guider par les éléments du mots.
- La différence avec les automates finis est dans l'utilisation d'une pile d'états dans leur configuration (au lieu d'un unique état).

Automate à pile

Definition (Automate à pile)

$M = (V, Q, \Delta, q_0, F)$ avec

- V : alphabet (fini) d'entrée
- Q : ensemble fini d'états
- q_0 : état initial
- $F \subset Q$: états finaux
- $\Delta \subset Q^+ \times V^{0|1} \times Q^*$: relation de transition
  notez bien les puissances ajoutées par rapport aux automates finis

Fonctionnement d'un automate à pile

- automate à pile : $M = (V, Q, \Delta, q_o, F)$
- configuration : $(\gamma|, w) \in Q^* \times V^*$
  - γ est la pile d'états (fond à droite)
  - config. initiale : $(q_0|, w)$ pour un mot w à reconnaitre
  - config. finale : $(q_f\gamma|, \epsilon)$ avec $q_f \in F$
- transition :
  $(\gamma_2\gamma_1|, aw) \rightarrow_M (\gamma_3\gamma_1|, w)$ si $(\gamma_2, a, \gamma_3) \in \Delta$
- suite de transitions : comme pour automates finis
- acceptation d'un mot $w \in V^*$
  $w \in \mathcal{L}(M)$ ssi $(q_o|, w) \rightarrow_M^* (q_f\gamma|, \epsilon)$ avec $q_f \in F$

Des grammaires aux automates

Problème

Comment construire un automate à pile M à partir d'une grammaire G tel que $\mathcal{L}(M) = \mathcal{L}(G)$?

- quels états (Q) ?
- quelles transitions (Δ) ?

- grammaire = spécification du langage (génération)
- automate = implémentation du langage (reconnaissance)

C'est un problème de compilation !

- langage source : grammaires
- langage cible : automates à pile
- préservation de la sémantique : $\mathcal{L}(G) = \mathcal{L}(M)$
- efficacité : automate déterministe, si possible

Des grammaires aux automates

Problème

Comment construire un automate à pile M à partir d'une grammaire G tel que $\mathcal{L}(M) = \mathcal{L}(G)$?

- quels états (Q) ?
- quelles transitions (Δ) ?

- grammaire = spécification du langage (génération)
- automate = implémentation du langage (reconnaissance)

C'est un problème de compilation !

- langage source : grammaires
- langage cible : automates à pile
- préservation de la sémantique : $\mathcal{L}(G) = \mathcal{L}(M)$
- efficacité : automate déterministe, si possible

Des grammaires aux automates

Décomposition du problème :

1. traduction : dérivation d'un automate des items non-contextuels (AINC), indéterministe
2. optimisation : réduction de l'indéterminisme par application de restrictions
   - différentes restrictions
   - différents types d'analyses pour différents types de grammaires
     ex. : descendante, ascendante, tabulée