

Théorie des langages et de la compilation

Thierry Massart
Université Libre de Bruxelles
Département d'Informatique

Septembre 2007

Remerciements

Je tiens à vivement remercier
Gilles Geeraerts, Sébastien Collette, Camille Constant et Thomas De Schamphelleire
pour leur relecture attentive du présent syllabus.

Thierry Massart

①	Introduction	5
②	Les langages réguliers et automates finis	44
③	L'analyse lexicale (scanning)	140
④	Les grammaires	163
⑤	Les grammaires régulières	187
⑥	Les grammaires context free	199
⑦	Les automates à pile et propriétés des langages context-free	235
⑧	L'analyse syntaxique (parsing)	275
⑨	Les parseurs LL(k)	296
⑩	Les parseurs LR(k)	363
⑪	L'analyse sémantique	445
⑫	La génération de code	481
⑬	Les machines de Turing	526
⑭	Eléments de décidabilité	560

- J. E. Hopcroft, R. Motwani, and J. D. Ullman ; *Introduction to Automata Theory, Languages, and Computation*, Second Edition, Addison-Wesley, New York, 2001.
- Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers : Principles, Techniques and Tools*, Addison-Wesley, 1986

Autres références :

- John R. Levine, Tony Mason, Doug Brown, *Lex & Yacc*, O'Reilly ed, 1992.
- Reinhard Wilhelm, Dieter Maurer, *Compiler Design*, Addison-Wesley, 1995. (parle des P-machines)
- Pierre Wolper, *Introduction à la Calculabilité*, InterEditions, 1991.

Chapitre 1 : Introduction

- 1 But du cours
- 2 Ordre des chapitres
- 3 Qu'est-ce que la théorie des langages ?
- 4 Qu'est-ce qu'un compilateur ?
- 5 Phases de la compilation
- 6 Quelques rappels et notations mathématiques

But du cours

- Ordre des chapitres
- Qu'est-ce que la théorie des langages ?
- Qu'est-ce qu'un compilateur ?
- Phases de la compilation
- Quelques rappels et notations mathématiques

Plan

- 1 But du cours
- 2 Ordre des chapitres
- 3 Qu'est-ce que la théorie des langages ?
- 4 Qu'est-ce qu'un compilateur ?
- 5 Phases de la compilation
- 6 Quelques rappels et notations mathématiques

Que va-t-on apprendre dans ce cours ? (entre autres)

- ① Comment définir formellement un modèle pour décrire
 - un langage (de programmation ou autre)
 - un système (informatique)
- ② Comment déduire des propriétés sur ce modèle
- ③ Ce qu'est
 - un compilateur
 - un outil de traitement de données
- ④ La notion d'outil permettant de construire d'autres outils
Exemple : générateur de (partie de) compilateur
- ⑤ Comment construire un compilateur ou un outil de traitement de données
 - en programmant
 - en utilisant ces outils
- ⑥ Quelques notions de décidabilité

But du cours

Ordre des chapitres

Qu'est-ce que la théorie des langages ?

Qu'est-ce qu'un compilateur ?

Phases de la compilation

Quelques rappels et notations mathématiques

Démarche

Montrer la démarche scientifique et de l'ingénieur qui consiste à

- ① Comprendre les outils (mathématiques / informatiques) disponibles pour résoudre le problème
- ② Apprendre à manipuler ces outils
- ③ Concevoir un système à partir de ces outils
- ④ Implémenter ce système

Les outils ici sont

- les formalismes pour définir un langage ou modéliser un système
- les générateurs de parties de compilateurs

But du cours

Ordre des chapitres

Qu'est-ce que la théorie des langages ?

Qu'est-ce qu'un compilateur ?

Phases de la compilation

Quelques rappels et notations mathématiques

Plan

1 But du cours

2 Ordre des chapitres

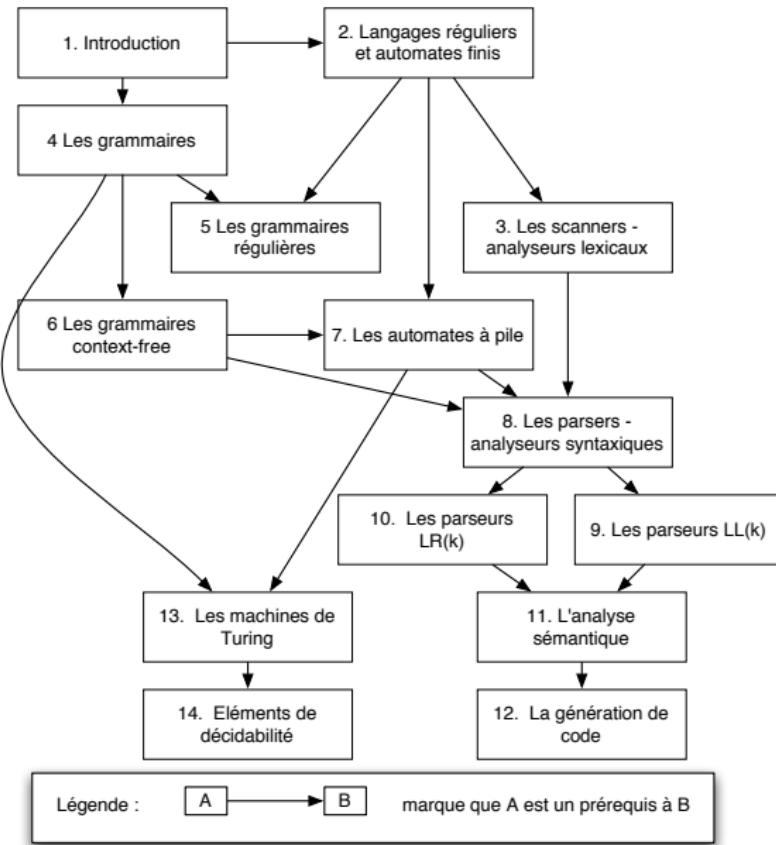
3 Qu'est-ce que la théorie des langages ?

4 Qu'est-ce qu'un compilateur ?

5 Phases de la compilation

6 Quelques rappels et notations mathématiques

Ordre des chapitres



Exemples de séquences de lecture

- Pour voir l'ensemble de la matière : 1-14 en séquence
- Pour ne voir que la matière “compilateur” : 1-2-3-4-6-7-8-9-10-11-12
- Pour ne voir que la matière “théorie des langages” : 1-2-4-5-6-7-13-14

Plan

- 1 But du cours
- 2 Ordre des chapitres
- 3 Qu'est-ce que la théorie des langages ?
- 4 Qu'est-ce qu'un compilateur ?
- 5 Phases de la compilation
- 6 Quelques rappels et notations mathématiques

Le monde de la théorie des langages

Objectifs de la théorie des langages

Comprendre le fonctionnement des langages (formel), vus comme moyen de communication, d'un point de vue mathématique.

Définition (Langage - mot)

- *Un langage est un ensemble de mots.*
- *Un mot (ou lexème ou chaîne de caractères ou string en anglais) est une séquence de symboles (signes) élémentaires dans un alphabet donné.*

Alphabets, mots, langages

Exemple (alphabets, mots et langages)

Alphabet	Mots	Langages
$\Sigma = \{0, 1\}$	$\epsilon, 0, 1, 00, 01$	$\{00, 01, 1, 0, \epsilon\}, \{\epsilon\}, \emptyset$
{a, b, c, ..., z}	bonjour, ca, va	{bonjour, ca, va, ϵ }
{"héron", "petit", "pas"}	"héron" "petit" "pas"	{ ϵ , "héron" "petit" "pas"}
$\{\alpha, \beta, \gamma, \delta, \mu, \nu, \pi, \sigma, \tau\}$	$\tau\alpha\gamma\alpha\delta\alpha$	{ $\epsilon, \tau\alpha\gamma\alpha\delta\alpha$ }
{0, 1}	$\epsilon, 01, 10$	{ $\epsilon, 01, 10, 0011, 0101, \dots$ }

Notations

Nous utiliserons habituellement des notations conventionnelles :

- Alphabet : Σ (exemple : $\Sigma = \{0, 1\}$)
- Mots : x, y, z, \dots (exemple : $x = 0011$)
- Langages : L, L_1, \dots (exemple : $L = \{\epsilon, 00, 11\}$)

Le monde de la théorie des langages (suite)

Notions générales étudiées ici

- La notion de **grammaire** (formelle) qui définit (la **syntaxe** d') un langage,
- La notion d'**automate** permettant de déterminer si un mot fait partie d'un langage (et donc permettant également de définir un langage (l'ensemble des mots acceptés)),
- La notion d'**expression régulière** qui permet de dénoter un langage.

Motivations et applications

Applications pratiques de la théorie des langages

- la définition formelle de la syntaxe et sémantique de langages de programmation,
- la construction de compilateurs,
- la modélisation abstraite d'un système informatique, électronique, biologique, ...

Motivations théoriques

Liées aux théories de

- la **calculabilité** (qui détermine en particulier quels problèmes sont solubles par un ordinateur)
- la **complexité** qui étudie les ressources (principalement temps et espace mémoire) requises pour résoudre un problème donné

Plan

- 1 But du cours
- 2 Ordre des chapitres
- 3 Qu'est-ce que la théorie des langages ?
- 4 Qu'est-ce qu'un compilateur ?**
- 5 Phases de la compilation
- 6 Quelques rappels et notations mathématiques

Définition (Compilateur)

Un compilateur est un programme informatique de traduction $C_{L_S \rightarrow L_O}^{L_C}$ avec

- ① L_C le langage avec lequel est écrit le compilateur
- ② L_S le langage source à compiler
- ③ L_O le langage cible ou langage objet

Exemple (pour $C_{L_S \rightarrow L_O}^{L_C}$)

L_C	L_S	L_O
C	Assembleur RISC	Assembleur RISC
C	C	Assembleur P7
C	java	C
Java	$L^A T_E X$	HTML
C	XML	PDF

Si $L_C = L_S$: peut nécessiter un **bootstrapping** pour compiler $C_{L_C \rightarrow L_O}^{L_C}$

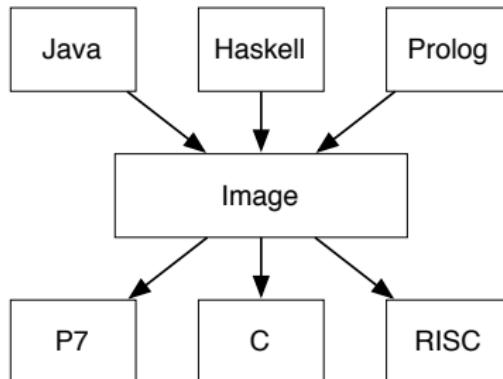
Structure générale d'un compilateur

Généralement un langage intermédiaire L_I est utilisé.

Le compilateur est formé :

- d'un **front-end** $L_S \rightarrow L_I$
- d'un **back-end** $L_I \rightarrow L_O$

Facilite la construction de nouveaux compilateurs



Caractéristiques des compilateurs

- Efficacité
- Robustesse
- Portabilité
- Fiabilité
- Code debuggable
- À une passe
- À n passes (70 pour un compilateur PL/I !)
- Optimisant
- Natif
- Cross-compilation

Compilateur vs. Interpréteur

Interpréteur = outil qui **analyse**, **traduit**, mais aussi **exécute** un programme écrit dans un langage informatique.

L'interpréteur se charge donc également de l'exécution au fur et à mesure de son interprétation.

Plan

- 1 But du cours
- 2 Ordre des chapitres
- 3 Qu'est-ce que la théorie des langages ?
- 4 Qu'est-ce qu'un compilateur ?
- 5 **Phases de la compilation**
- 6 Quelques rappels et notations mathématiques

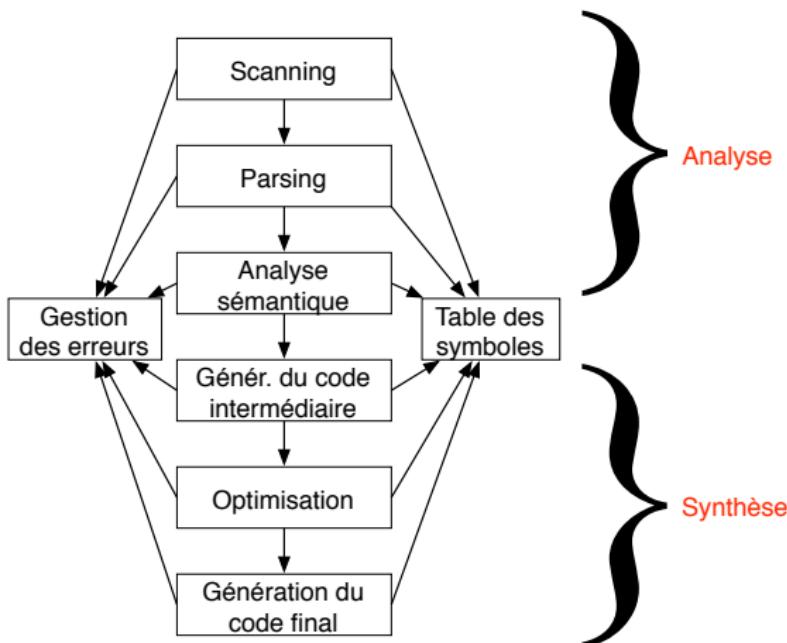
Un petit programme à compiler

Exemple (de programme à compiler)

```
int main()
// Conjecture De Syracuse
// Hypothèse : N > 0
{
    long int N;

    cout << "Entrer Un Nombre : ";
    cin >> N;
    while (N != 1)
    {
        if (N%2 == 0)
            N = N/2;
        else
            N = 3*N+1;
    }
    cout << N << endl; //Imprime 1
}
```

6 phases de compilation : 3 d'analyse - 3 de synthèse



Etapes de la compilation

Compilation découpée en 2 étapes

- ① L'**analyse** décompose et identifie les éléments et relations du programme source et construit son **image** (représentation hiérarchique du programme avec ses relations),
- ② La **synthèse** qui construit à partir de l'*image* un programme en langage cible

Contenu de la table des symboles

Un enregistrement par identificateur du programme à compiler contenant les valeurs des attributs pour le décrire.

Remarque

En cas d'erreur, le compilateur peut essayer de se resynchroniser pour éventuellement essayer de reporter d'autres erreurs

Analyse lexicale (scanning)

- Un programme peut être vu comme une “phrase” ; le rôle principal de l’analyse lexicale est d’identifier les “mots” de la phrase.
- Le scanner décompose le programme en **lexèmes** en identifiant les **unités lexicales** de chaque lexème.

Exemple (de découpe en tokens)

```
int main ()  
// Conjecture De Syracuse  
// Hypothèse : N > 0  
{  
    long int N ;  
  
    cout << "Entrer Un Nombre : " ;  
    cin >> N ;  
    while ( N != 1 )  
    {  
        if ( N % 2 == 0 )  
            N = N / 2 ;  
        else  
            N = 3 * N + 1 ;  
    }  
    cout << N << endl ; //Imprime 1  
}
```

Analyse lexicale (scanning)

Définition (Unité lexicale (ou token))

Type générique d'éléments lexicaux (correspond à un ensemble de strings ayant une sémantique proche).

Exemple : identificateur, opérateur relationnel, mot clé "begin"...

Définition (Lexème (ou string))

Occurrence d'une unité lexicale.

Exemple : `N` est un lexème dont l'unité lexicale est identificateur

Définition (Modèle (pattern))

Règle décrivant une unité lexicale

En général un modèle est donné sous forme d'expression régulière (voir ci-dessous)

Relation entre lexème, unité lexicale et modèle

$$\text{unité lexicale} = \{ \text{lexème} \mid \text{modèle(lexème)} \}$$

Exemples introductifs de la notion d'expression régulière

Opérateurs des expressions régulières :

- **.** : concaténation (généralement omis)
- **+** : union
- ***** : répétition (0,1,2, ... fois) = (fermeture de Kleene (prononcé Klayni !))

Exemple (Quelques expressions régulières)

- chiffre = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
- nat-nb = chiffre chiffre*
- operateur = << + != + == + ...
- par-ouvrante = (
- par-fermante =)
- lettre = a + b + ... + z
- identificateur = lettre (lettre + chiffre)*

Résultat du scanning

Exemple (d'unités lexicales et de lexèmes)

unité lexicale	lexème
identificateur	int
identificateur	main
par-ouvrante	(
par-fermante)
...	

Autres buts du scanning

Autres buts du scanning

- Met (éventuellement) les identificateurs (non prédéfinis) et littéraux dans la table des symboles^a
- Produit le listing / lien avec un éditeur intelligent
- Nettoie le programme source (supprime les commentaires, espaces, tabulations, ...)

^apeut se faire lors d'une phase ultérieure d'analyse

Analyse syntaxique (parsing)

- Le rôle principal de l'analyse syntaxique est de trouver la structure de la “phrase” (le programme) : càd de construire une représentation interne au compilateur et facilement manipulable de la structure syntaxique du programme source.
- Le **parser** construit l'**arbre syntaxique** correspondant au code.

L'ensemble des arbres syntaxiques possibles pour un programme est défini grâce à une grammaire (context-free).

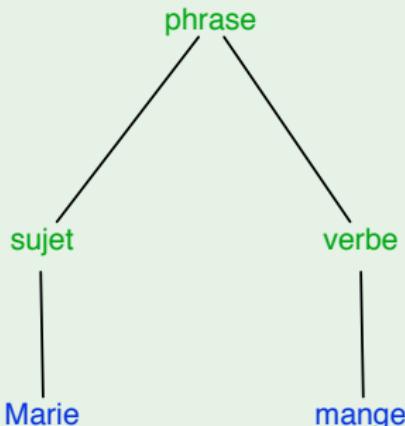
Exemple de grammaire

Exemple (Grammaire d'une phrase)

- phrase = sujet verbe
- sujet = “Jean” | “Marie”
- verbe = “mange” | “parle”

peut donner

- **Jean mange**
- **Jean parle**
- **Marie mange**
- **Marie parle**



Arbre syntaxique de la phrase
Marie mange

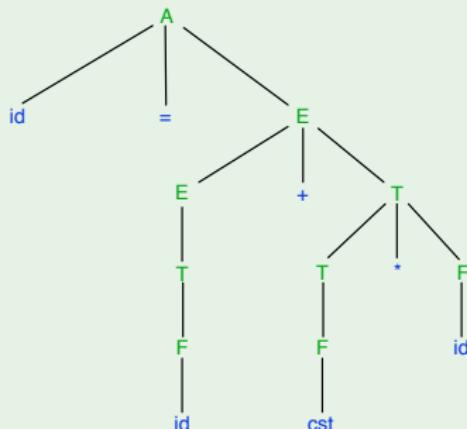
Exemple de grammaire (2)

Exemple (Grammaire d'une expression)

- $A = "id" \ "=\> E$
- $E = T \mid E \ "+" \ T$
- $T = F \mid T \ "*" \ F$
- $F = "id" \mid "cst" \mid "(" \ E \ ")"$

peut donner :

- $id = id$
- $id = id + cst * id$
- ...



Arbre syntaxique de la phrase
 $id = id + cst * id$

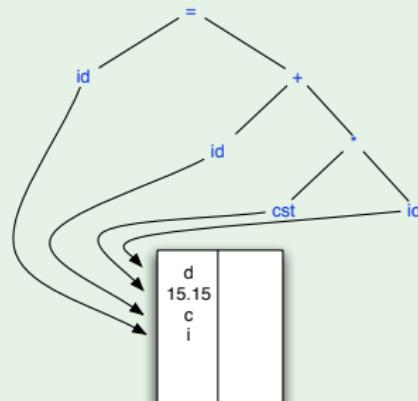
Exemple de grammaire (2 (suite))

Exemple (Grammaire d'une expression)

- $A = "id" "=" E$
- $E = T \mid E "+" T$
- $T = F \mid T "*" F$
- $F = "id" \mid "cst" \mid "(" E ")"$

peut donner :

- $id = id$
- $id = id + cst * id$
- ...



Arbre syntaxique abstrait
avec références à la table des
symboles de la phrase $i = c + 15.15 * d$

Analyse sémantique

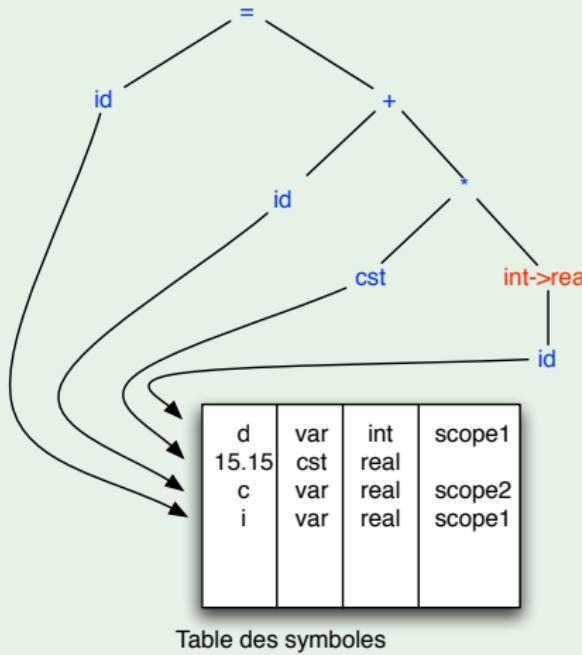
Rôle de l'analyse sémantique

Pour un langage impératif, l'**analyse sémantique** (appelée aussi **gestion de contexte**) s'occupe des relations *non locales* ; elle s'occupe ainsi :

- ① du **contrôle de visibilité** et du lien entre les définitions et utilisations des identificateurs (en utilisant/construisant la table des symboles)
- ② du **contrôle de type** des “objets”, nombre et type des paramètres de fonctions
- ③ du **contrôle de flot** (vérifie par exemple qu'un goto est licite - voir exemple plus bas)
- ④ de construire un **arbre syntaxique abstrait complété** avec des informations de type et un **graphe de contrôle de flot** pour préparer les phases de synthèse.

Exemple de résultat de l'analyse sémantique

Exemple (pour l'expression $i = c + 15.15 * d$)



Arbre syntaxique abstrait modifié
avec références à la table des
symboles de la phrase $i = c + 15.15 * d$

Synthèse

Phases de synthèse

Pour un langage impératif, la **synthèse** comporte les 3 phases

- ① **Génération de code intermédiaire** sous forme de langage universel qui
 - utilise un adressage symbolique
 - utilise des opérations standard
 - effectue l'allocation mémoire (résultat dans des variables temporaires...)
- ② **Optimisation du code**
 - supprime le code “mort”
 - met certaines instructions hors des boucles
 - supprime certaines instructions et optimise les accès à la mémoire
- ③ **Production du code final**
 - Allocation de mémoire physique
 - gestion des registres

Exemple de synthèse

Exemple (pour le code $i = c + 15.15 * d$)

① Génération de code intermédiaire

```
temp1 <- 15.15
temp2 <- Int2Real(id3)
temp2 <- temp1 * temp2
temp3 <- id2
temp3 <- temp3 + temp2
id1    <- temp3
```

② Optimisation du code

```
temp1 <- Int2Real(id3)
temp1 <- 15.15 * templ
id1 <- id2 + templ
```

③ Production du code final

```
MOVF  id3,R1
ITOR  R1
MULF  15.15,R1,R1
ADDF  id2,R1,R1
STO   R1,id1
```

Plan

- 1 But du cours
- 2 Ordre des chapitres
- 3 Qu'est-ce que la théorie des langages ?
- 4 Qu'est-ce qu'un compilateur ?
- 5 Phases de la compilation
- 6 Quelques rappels et notations mathématiques

Notations utilisées

- Σ : Alphabet du langage
- x, y, z, t, x_i (lettre à la fin de l'alphabet) : string de symboles de Σ (exemple $x = abba$)
- ϵ : mot vide
- $|x|$: longueur du string x ($|\epsilon| = 0, |abba| = 4$)
- $a^i = aa...a$ (string formé de i fois le caractère a)
- $x^i = xx...x$ (string formé de i fois le string x)
- L, L', L_i, A, B : langages

Opérations sur les strings

- **concaténation** : ex : `lent.gage = lentgage`
 - $\epsilon W = W = W\epsilon$
- w^R : **image miroir** de w (ex : $abbd^R = dbba$)
- **préfixe** de w . ex : si $w=abbc$
 - les préfixes sont : ϵ , a, ab, abb, abbc
 - les préfixes **propres** sont ϵ , a, ab, abb
- **suffixe** de w . ex : si $w=abbc$
 - les suffixes sont : ϵ , c, bc, bbc, abbc
 - les suffixes **propres** sont ϵ , c, bc, bbc

Opérations sur les langages

Définition (Langage sur l'alphabet Σ)

Ensemble de strings sur cet alphabet

Les opérations sur les langages sont donc des opérations ensemblistes

- $\cup, \cap, \setminus, A \times B, 2^A$
- **concaténation ou produit** : ex : $L_1 \cdot L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$
 - $L^0 =_{def} \{\epsilon\}$
 - $L^i = L^{i-1} \cdot L$
- La fermeture de Kleene : $L^* =_{def} \bigcup_{i \in \mathbb{N}} L^i$
- La fermeture positive : $L^+ =_{def} \bigcup_{i \in \mathbb{N}^+} L^i$
- le complément : $\bar{L} = \{w \mid w \in \Sigma^* \wedge w \notin L\}$

Relations

Définition (Équivalence)

Relation :

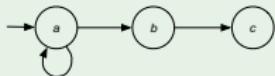
- *réflexive*
- *symétrique*
- *transitive*

Définition (Fermeture de relations)

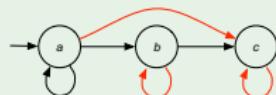
Soit \mathcal{P} un ensemble de propriétés d'une relation \mathbf{R} , la \mathcal{P} -fermeture de \mathbf{R} est la plus petite relation \mathbf{R}' qui inclut \mathbf{R} et possède les propriétés \mathcal{P}

Exemple (de fermeture réflexo-transitive)

La fermeture transitive \mathbf{R}^+ , la fermeture reflexo-transitive \mathbf{R}^*



donne pour \mathbf{R}^*



Propriété de fermeture d'une classe de langages

Définition (Fermeture d'une classe de langages)

*Une classe de langages \mathcal{C} est **fermée** pour une opération op , si le langage résultant de cette opération sur n'importe quel(s) langage(s) de \mathcal{C} reste dans cette même classe de langages \mathcal{C} .*

Exemple : en supposant que op soit binaire

\mathcal{C} est fermé pour op

ssi

$$\forall L_1, L_2 \in \mathcal{C} \Rightarrow L_1 \text{ } op \text{ } L_2 \in \mathcal{C}$$

Cardinalités

Définition (Même cardinalité)

Deux ensembles ont la même cardinalité s'il existe une bijection entre eux.
Notons

- \aleph_0 la cardinalité des ensembles infinis dénombrables (comme \mathbb{N})
- \aleph_1 la cardinalité des ensembles infinis continus (comme \mathbb{R})

Nous supposons que les ensembles non dénombrables sont infinis continus.

Cardinalité de Σ^* et 2^{Σ^*}

Soit un alphabet fini non vide Σ ,

- Σ^* : l'ensemble des strings de Σ , est infini dénombrable
- $\mathcal{P}(\Sigma^*)$ dénoté aussi 2^{Σ^*} : l'ensemble des langages de Σ , est infini continu

Chapitre 2 : Les langages réguliers et automates finis

- 1 Les langages réguliers et expressions régulières
- 2 Les automates finis
- 3 Équivalence entre FA et RE
- 4 Autres types d'automates finis
- 5 Propriétés des langages réguliers
- 6 Équivalence et minimisation d'automates finis

Les langages réguliers et expressions régulières

Les automates finis

Équivalence entre FA et RE

Autres types d'automates finis

Propriétés des langages réguliers

Équivalence et minimisation d'automates finis

Plan

1 Les langages réguliers et expressions régulières

2 Les automates finis

3 Équivalence entre FA et RE

4 Autres types d'automates finis

5 Propriétés des langages réguliers

6 Équivalence et minimisation d'automates finis

Les langages réguliers et expressions régulières

Les automates finis

Équivalence entre FA et RE

Autres types d'automates finis

Propriétés des langages réguliers

Équivalence et minimisation d'automates finis

Introduction

Motivation

- Les expressions régulières permettent aisément de dénoter des langages (réguliers)
- Par exemple, **UNIX** utilise intensivement les expressions régulières étendues dans ses shells
- Elles sont également utilisées pour définir les unités lexicales d'un langage de programmation

Définition des langages réguliers

Remarque préliminaire

- Tout langage fini peut être énuméré (même si cela peut être long).
- Pour les langages infinis une énumération exhaustive n'est pas possible
- La classe des langages réguliers (définie ci-dessous) comprend tous les langages finis et certains langages infinis

Définition (classe des langages réguliers)

L'ensemble \mathcal{L} des langages réguliers sur un alphabet Σ est le plus petit ensemble satisfaisant :

- ① $\emptyset \in \mathcal{L}$
- ② $\{\epsilon\} \in \mathcal{L}$
- ③ $\forall a \in \Sigma, \{a\} \in \mathcal{L}$
- ④ *si $A, B \in \mathcal{L}$ alors $A \cup B, A.B, A^* \in \mathcal{L}$*

Notation des langages réguliers

Définition (ensemble des expressions régulières (RE))

L'ensemble des expressions régulières (RE) sur un alphabet Σ est le plus petit ensemble contenant :

- ① \emptyset : dénote l'ensemble vide,
- ② ϵ : dénote l'ensemble $\{\epsilon\}$,
- ③ $\forall a \in \Sigma, a$: dénote l'ensemble $\{a\}$,
- ④ ayant r et s qui dénotent resp. R et S :
 $r + s$, rs et r^* dénotent resp. $R \cup S$, $R.S$ et R^*

On suppose $$ < $.$ < $+$ et on rajoute des $()$ si nécessaire*

Exemple (d'expressions régulières)

- 00
- $(0 + 1)^*$
- $(0 + 1)^*00(0 + 1)^*$
- 0^410^4 notation pour 000010000
- $(01)^* + (10)^* + 0(10)^* + 1(01)^*$
- $(\epsilon + 1)(01)^*(\epsilon + 0)$

Propriétés des langages réguliers

Théorème (correspondance expressions et langages réguliers)

Un langage est régulier ssi il est dénoté par une expression régulière.

Preuve

Par induction structurelle

Propriétés de ϵ et \emptyset

- $\epsilon w = w = w\epsilon$
- $\emptyset w = \emptyset = w\emptyset$
- $\emptyset + r = r = r + \emptyset$
- $\epsilon^* = \epsilon$
- $\emptyset^* = \epsilon$
- $(\epsilon + r)^* = r^*$

Théorème (cardinalité des langages réguliers)

L'ensemble des langages réguliers est infini dénombrable

Plan

- 1 Les langages réguliers et expressions régulières
- 2 **Les automates finis**
- 3 Équivalence entre FA et RE
- 4 Autres types d'automates finis
- 5 Propriétés des langages réguliers
- 6 Équivalence et minimisation d'automates finis

Automates

Présentation informelle

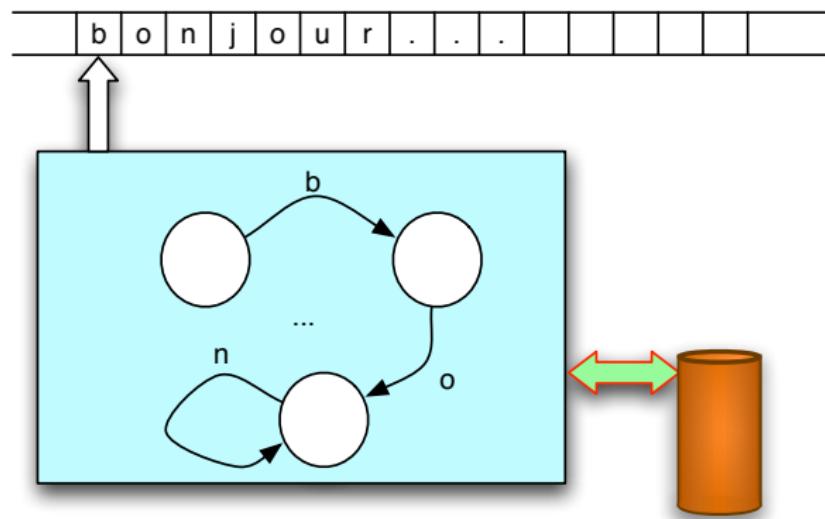
Un **automate** M est un **modèle** mathématique d'un **système** avec des **entrées** et des **sorties** discrètes.

Il contient généralement

- des états de contrôle (M se trouve dans un de ces états)
- un ruban de données contenant des symboles
- une tête de lecture
- une mémoire

Automates

Présentation informelle



Exemple : un protocole d'e-commerce avec e-money

Exemple (Evénements possibles :)

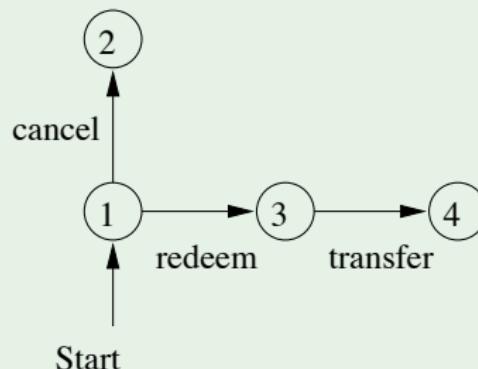
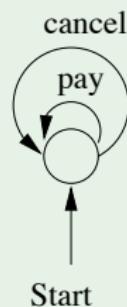
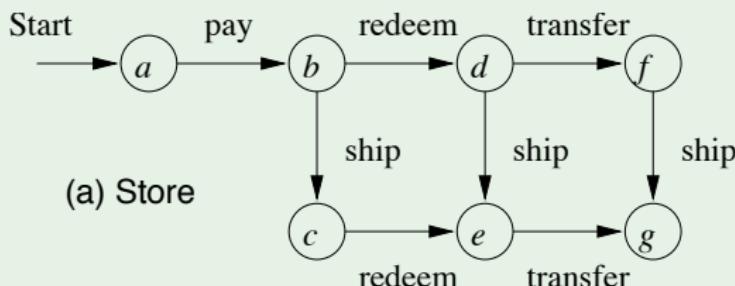
- ① **pay** : le client paye le magasin
- ② **cancel** : le client stoppe la transaction
- ③ **ship** : le magasin envoie le bien
- ④ **redeem** : le magasin demande l'argent à la banque
- ⑤ **transfer** : la banque transfert l'argent au magasin

Remarque

L'exemple sera formalisé par un (des) automate(s) fini(s) (voir plus bas)

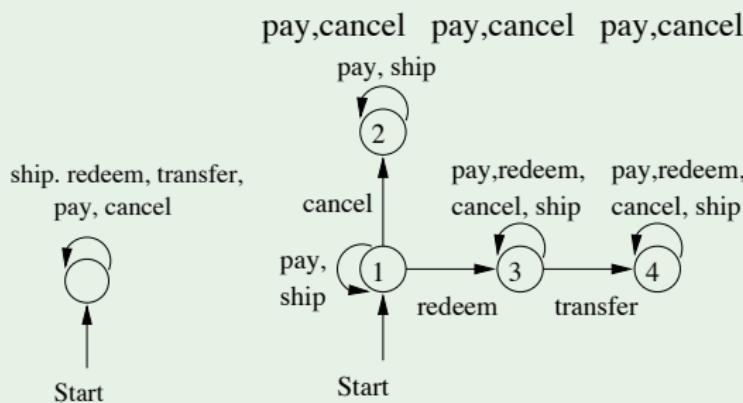
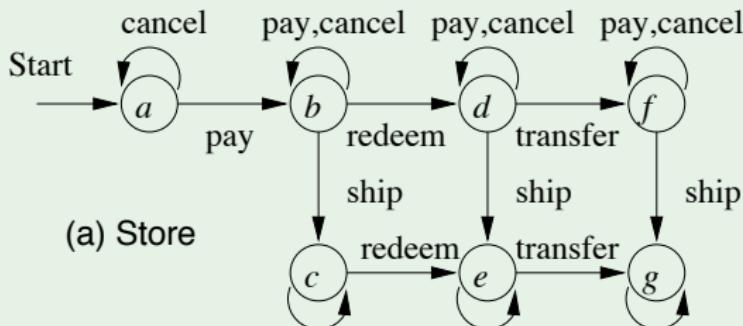
Exemple : un protocole d'e-commerce avec e-money (2)

Exemple (Protocole pour chaque participant)



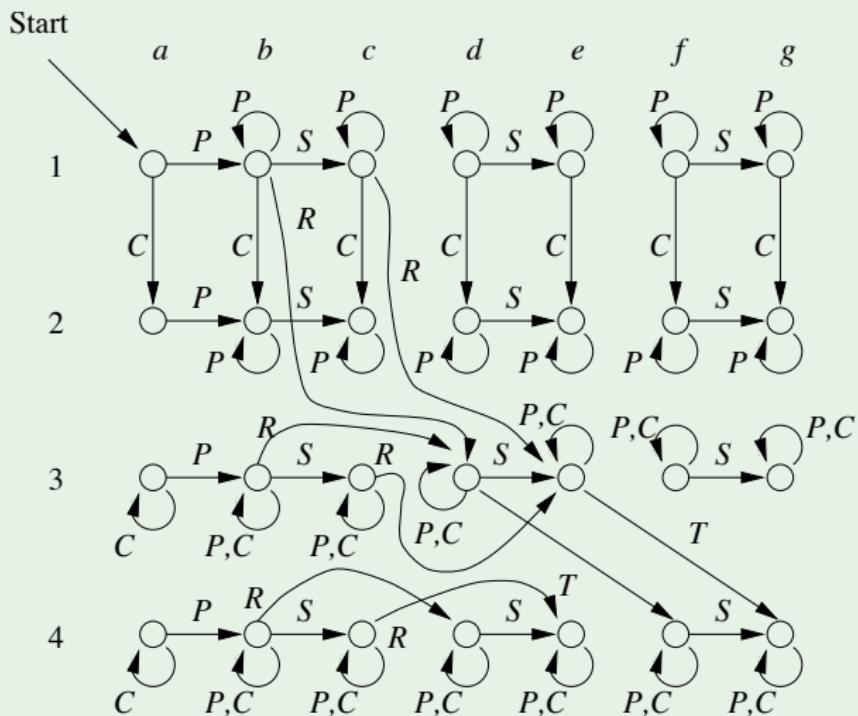
Exemple : un protocole d'e-commerce avec e-money (2)

Exemple (Protocole complet)



Exemple : un protocole d'e-commerce avec e-money (2)

Exemple (Système complet)



Mise en garde

Dans ce cours, nous examinerons les automates comme formalisme acceptant un ensemble de strings et donc définissant un langage (tous les strings acceptés)

Restriction des automates finis

Un automate fini (FA) :

- n'a pas de mémoire
- ne peut que lire sur le ruban
- la tête de lecture se déplace toujours vers la droite

On distingue

- Les **automates finis déterministes (DFA)**
- Les **automates finis non déterministes (NFA)**
- Les **automates finis non déterministes avec transitions spontannées (ϵ -NFA)**

-> Ajoutons un symbole ϵ pour décrire ces transitions spontannées

Automate fini : définition formelle

Définition (Automate fini)

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

avec

- ① Q : ensemble fini des états
- ② Σ : alphabet (symboles permis à l'entrée)
- ③ δ : **fonction de transition**
- ④ q_0 : état initial
- ⑤ $F \subseteq Q$: ensemble des états accepteurs

δ est défini pour

- M **DFA** : $\delta : Q \times \Sigma \rightarrow Q$
- M **NFA** : $\delta : Q \times \Sigma \rightarrow 2^Q$
- M **ϵ -NFA** : $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$

Exemples d'automates finis

Exemple

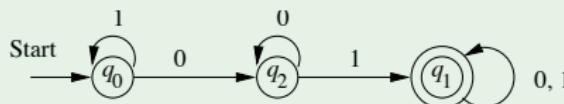
Un automate déterministe A qui accepte $L = \{x01y : x, y \in \{0, 1\}^*\}$

$$A = \langle \{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\} \rangle$$

avec δ :

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

graphiquement (diagramme de transition) :



String accepté

Un string $w = a_1 a_2 \dots a_n$ est accepté par le FA s'il existe un **chemin** dans le diagramme de transition qui **débute à l'état initial, se termine dans un état accepteur et a une séquence de labels $a_1 a_2 \dots a_n$**

Configuration et langage accepté

Définition (Configuration d'un FA)

Couple $\langle q, w \rangle \in Q \times \Sigma^*$

- Configuration initiale : $\langle q_0, w \rangle$ où w est le string à accepter
- Configuration finale (qui accepte) : $\langle q, \epsilon \rangle$ avec $q \in F$

Définition (Changement de configuration)

$\langle q, aw \rangle \underset{M}{\vdash} \langle q', w \rangle$ si

- $\delta(q, a) = q'$ pour un DFA
- $q' \in \delta(q, a)$ pour un NFA
- $q' \in \delta(q, a)$ pour un ϵ -NFA avec $a \in \Sigma \cup \{\epsilon\}$

Langage de M : $L(M)$

Définition ($L(M)$)

$$L(M) = \{w \mid w \in \Sigma^* \wedge \exists q \in F . \langle q_0, w \rangle \underset{M}{\overset{*}{\vdash}} \langle q, \epsilon \rangle\}$$

où

$\underset{M}{\overset{*}{\vdash}}$ est la fermeture réflexo-transitive de $\underset{M}{\vdash}$

Définition (Équivalence d'automates)

M et M' sont équivalents s'ils définissent le même langage ($L(M) = L(M')$)

Exemple de DFA

Exemple

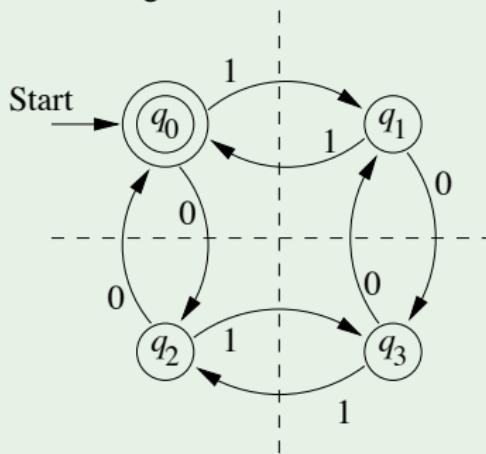
Le DFA M accepte l'ensemble des strings sur l'alphabet $\{0, 1\}$ avec un nombre pair de 0 et de 1.

$$M = \langle \{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\} \rangle$$

avec δ

et le diagramme de transition :

	0	1
$\star \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2



Exemple de NFA

Exemple

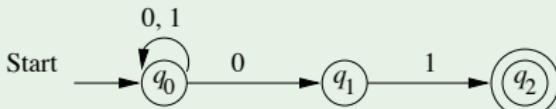
Le NFA M accepte l'ensemble des strings sur l'alphabet $\{0, 1\}$ qui se terminent par 01.

$$M = \langle \{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\} \rangle$$

avec δ

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset

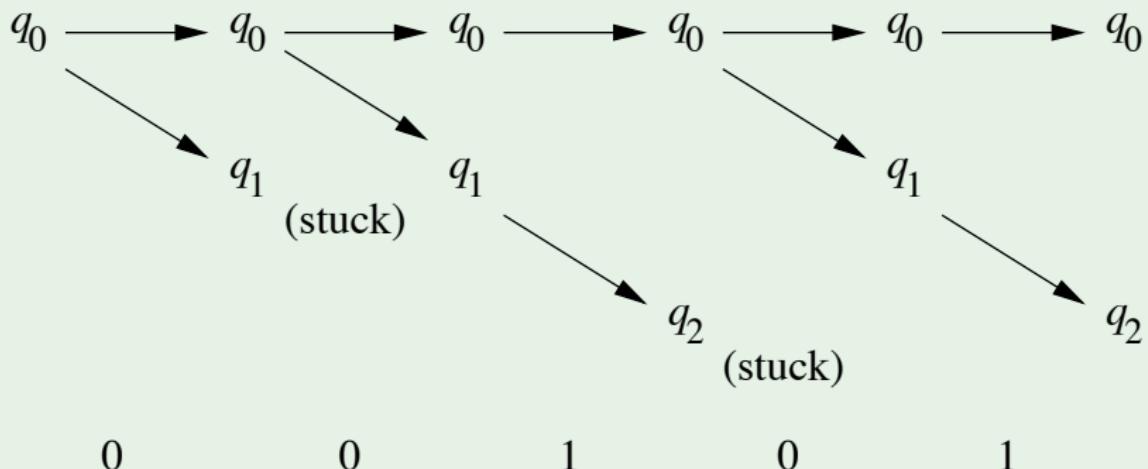
et le diagramme de transition :



Exemple de NFA (suite)

Exemple

Pour le string 00101 les chemins possibles sont :



Exemple de ϵ -NFA

Exemple

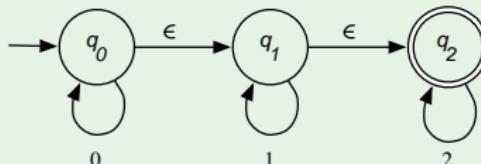
Le ϵ -NFA M accepte l'ensemble des strings sur l'alphabet $\{0, 1, 2\}$ qui correspond à l'expression régulière $0^*1^*2^*$.

$$M = \langle \{q_0, q_1, q_2\}, \{0, 1, 2\}, \delta, q_0, \{q_2\} \rangle$$

avec δ

	0	1	2	ϵ
$\rightarrow q_0$	$\{q_0\}$	\emptyset	\emptyset	$\{q_1\}$
q_1	\emptyset	$\{q_1\}$	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset	$\{q_2\}$	\emptyset

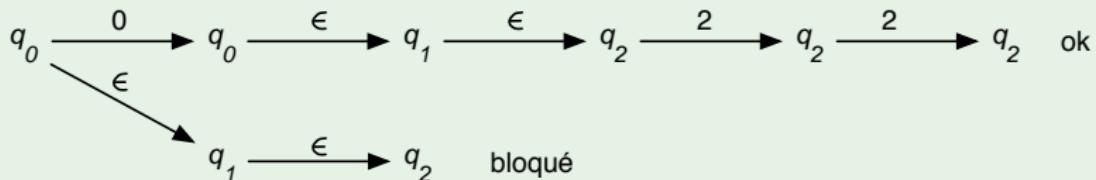
et le diagramme de transition :



Exemple de ϵ -NFA (suite)

Exemple

Pour le string 022 les chemins possibles sont :



Définition constructive de $L(M)$

Définition ($\hat{\delta}$: Extension de la fonction de transition)

Si on définit pour un ensemble d'états S : $\delta(S, a) = \bigcup_{p \in S} \delta(p, a)$

Pour les DFA : $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

- base : $\hat{\delta}(q, \epsilon) = q$
- ind. : $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$

$$L(M) = \{w \mid \hat{\delta}(q_0, w) \in F\}$$

Pour les NFA : $\hat{\delta} : Q \times \Sigma^* \rightarrow 2^Q$

- base : $\hat{\delta}(q, \epsilon) = \{q\}$
- ind. : $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$

$$L(M) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Pour les ϵ -NFA : $\hat{\delta} : Q \times \Sigma^* \rightarrow 2^Q$

- base : $\hat{\delta}(q, \epsilon) = \text{eclose}(q)$
- ind. : $\hat{\delta}(q, xa) = \text{eclose}(\delta(\hat{\delta}(q, x), a))$

avec $\text{eclose}(q) = \bigcup_{i \in \mathbb{N}} \text{eclose}^i(q)$

$$L(M) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

- $\text{eclose}^0(q) = \{q\}$
- $\text{eclose}^{i+1}(q) = \delta(\text{eclose}^i(q), \epsilon)$

Plan

- 1 Les langages réguliers et expressions régulières
- 2 Les automates finis
- 3 Équivalence entre FA et RE
- 4 Autres types d'automates finis
- 5 Propriétés des langages réguliers
- 6 Équivalence et minimisation d'automates finis

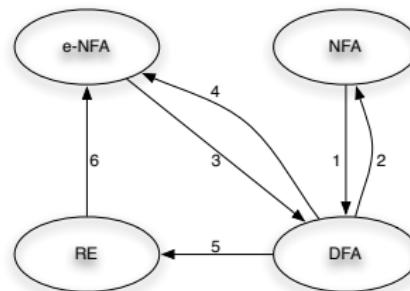
Équivalences entre automates finis (FA) et expressions régulières (RE)

Nous allons montrer que pour tout

- DFA
- NFA
- ϵ -NFA
- RE

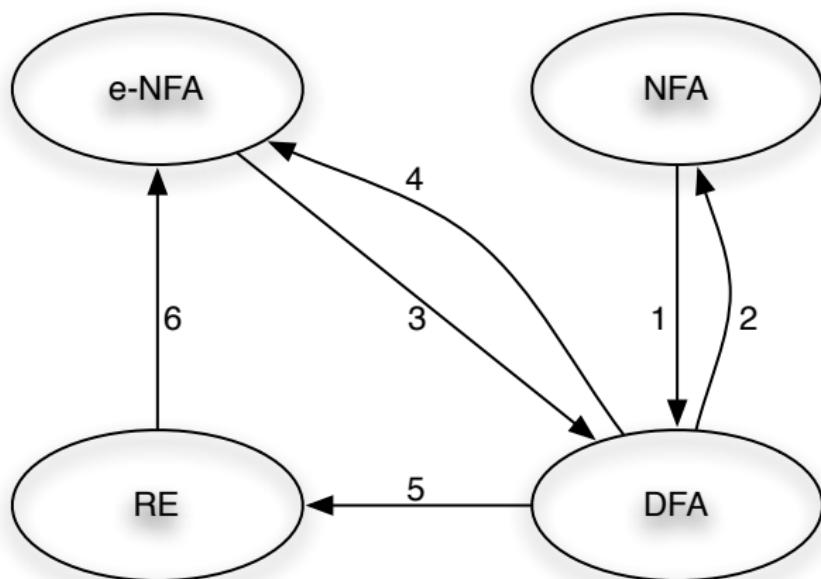
il est possible de le traduire dans les autres formalismes.

⇒ les 4 formalismes sont équivalents et définissent la même classe de langages : les **langages réguliers**



$$\mathcal{C}(\text{NFA}) \subseteq \mathcal{C}(\text{DFA})$$

Flèche 1 :



Équivalence entre DFA et NFA

- Écrire un NFA supprime la contrainte du déterminisme
- mais nous montrons que pour tout NFA N on peut construire un DFA D équivalent (càd $L(D) = L(N)$) et vice-versa.
- on utilise la technique de la **construction de sous-ensembles** : chaque état de D correspond à un ensemble d'états de N

$$\mathcal{C}(\text{NFA}) \subseteq \mathcal{C}(\text{DFA})$$

Théorème (Pour tout NFA N , il existe un DFA D avec $L(N) = L(D)$)

Preuve :

Soit un NFA N :

$$N = \langle Q_N, \Sigma, \delta_N, q_0, F_N \rangle$$

définissons (construisons) le DFA D :

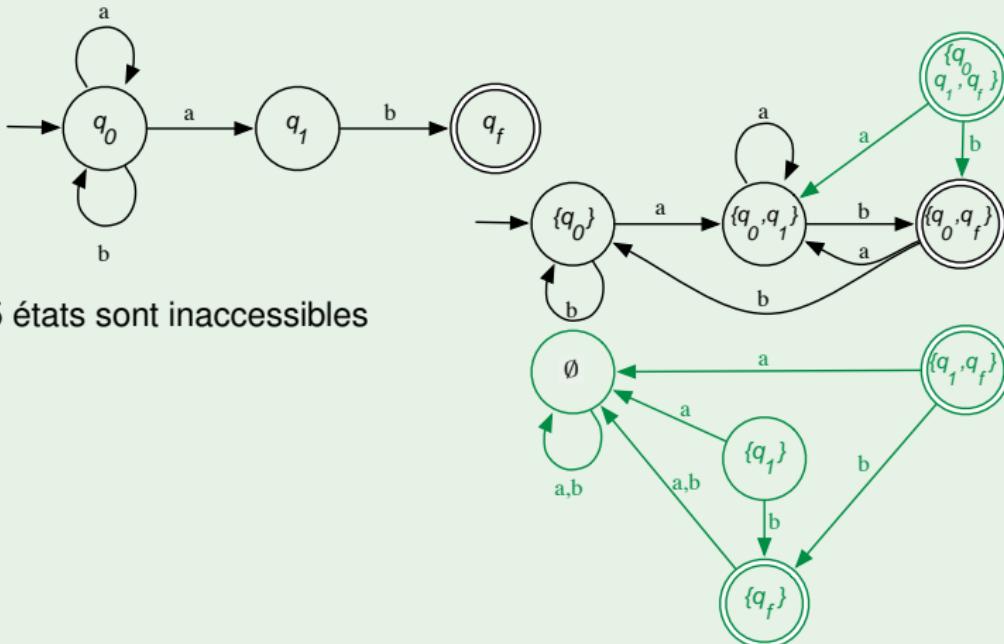
$$D = \langle Q_D, \Sigma, \delta_D, \{q_0\}, F_D \rangle$$

avec

- $Q_D = \{S \mid S \subseteq Q_N\}$ (càd $Q_D = 2^{|Q_N|}$)
- $F_D = \{S \subseteq Q_N \mid S \cap F_N \neq \emptyset\}$
- Pour tout $S \subseteq Q_N$ et $a \in \Sigma$,

$$\delta_D(S, a) = \delta_N(S, a) (= \bigcup_{p \in S} \delta_N(p, a))$$

Notons que $|Q_D| = 2^{|Q_N|}$ (bien qu'en général, beaucoup d'états soient inutiles car inaccessibles)

Exemple (NFA N et DFA D équivalents)

5 états sont inaccessibles

$\mathcal{C}(\text{NFA}) \subseteq \mathcal{C}(\text{DFA})$ (fin)

Théorème (Pour tout NFA N , il existe un DFA D avec $L(N) = L(D)$)

Preuve (suite) :

On va montrer que $L(D) = L(N)$

Il suffit de montrer par induction que :

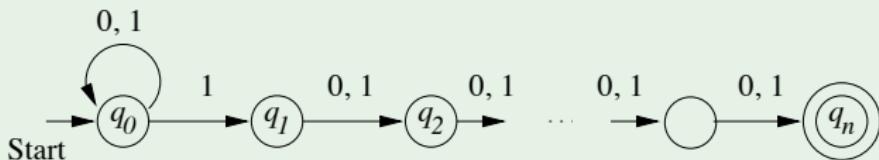
$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

- *Base : ($w = \epsilon$) : OK par définition des $\hat{\delta}$*
- *Induction : ($w = xa$)*

$$\begin{aligned}
 \hat{\delta}_D(\{q_0\}, xa) &\stackrel{\text{def}}{=} \delta_D(\hat{\delta}_D(\{q_0\}, x), a) \\
 &\stackrel{h.i}{=} \delta_D(\hat{\delta}_N(q_0, x), a) \\
 &\stackrel{cst}{=} \delta_N(\hat{\delta}_N(q_0, x), a) \\
 &\stackrel{\text{def}}{=} \hat{\delta}_N(q_0, xa)
 \end{aligned}$$

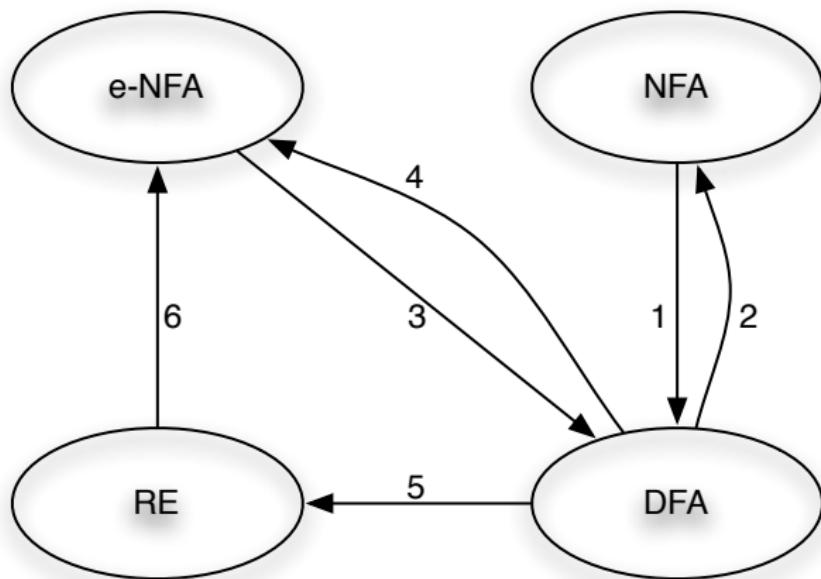
$\mathcal{C}(\text{NFA}) \subseteq \mathcal{C}(\text{DFA})$ (suite)

Exemple (NFA N avec $n + 1$ états qui a un DFA D équivalent à 2^n états)



$$\mathcal{C}(\text{NFA}) = \mathcal{C}(\text{DFA})$$

+Flèches 1 + 2 :



$$\mathcal{C}(\text{NFA}) = \mathcal{C}(\text{DFA})$$

Théorème (Un langage L est accepté par un DFA si et seulement si L est accepté par un NFA)

Preuve :

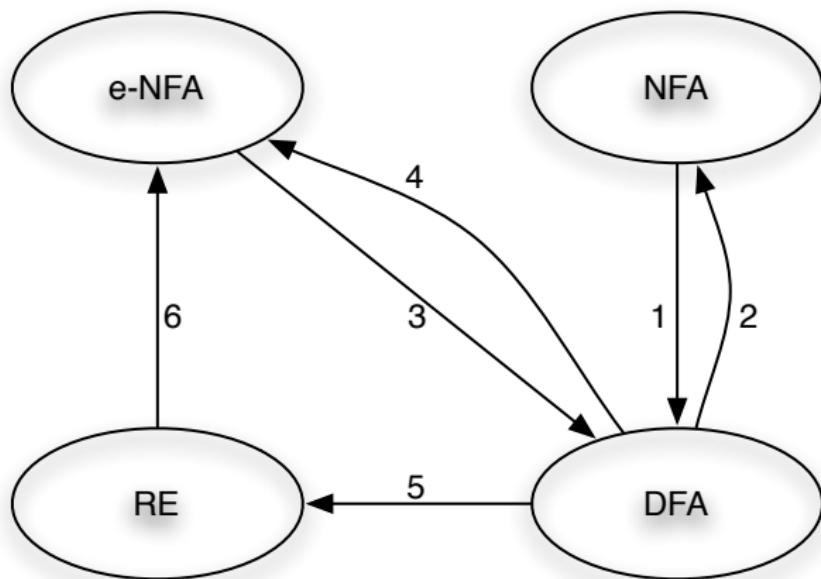
- *si* \Leftarrow voir théorème précédent
- *seulement si* \Rightarrow

Il suffit à partir du DFA D de construire un NFA N avec

$$\text{si } \delta_D(q, a) = p \text{ alors } \delta_N(q, a) = \{p\}$$

$$\mathcal{C}(\epsilon\text{-NFA}) \subseteq \mathcal{C}(\text{DFA})$$

+ Flèche 3 :



$$\mathcal{C}(\epsilon\text{-NFA}) \subseteq \mathcal{C}(\text{DFA})$$

Théorème (Pour tout ϵ -NFA E , il existe un DFA D avec $L(E) = L(D)$)

Preuve :

Soit un ϵ -NFA E :

$$E = \langle Q_E, \Sigma, \delta_E, q_0, F_E \rangle$$

définissons (*construisons*) le DFA D :

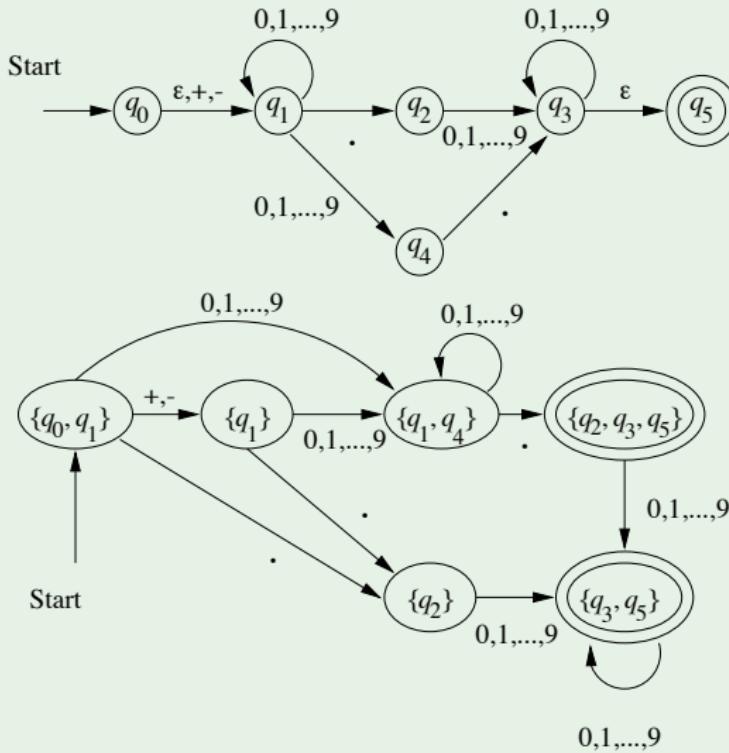
$$D = \langle Q_D, \Sigma, \delta_D, q_D, F_D \rangle$$

avec :

- $Q_D = \{S | S \subseteq Q_E \wedge S = \text{eclose}(S)\}$
- $q_D = \text{eclose}(q_0)$
- $F_D = \{S | S \in Q_D \wedge S \cap F_E \neq \emptyset\}$
- Pour tout $S \in Q_D$ et $a \in \Sigma$,

$$\delta_D(S, a) = \text{eclose}(\delta_E(S, a))$$

Exemple (ϵ -NFA E et DFA D équivalent)



$$\mathcal{C}(\epsilon\text{-NFA}) \subseteq \mathcal{C}(\text{DFA})$$

Théorème (Pour tout ϵ -NFA E , il existe un DFA D avec $L(E) = L(D)$)

Preuve (suite) :

On va montrer que $L(D) = L(E)$. Il suffit de montrer par induction que :

$$\hat{\delta}_E(\{q_0\}, w) = \hat{\delta}_D(q_D, w)$$

- *Base* : ($w = \epsilon$) :

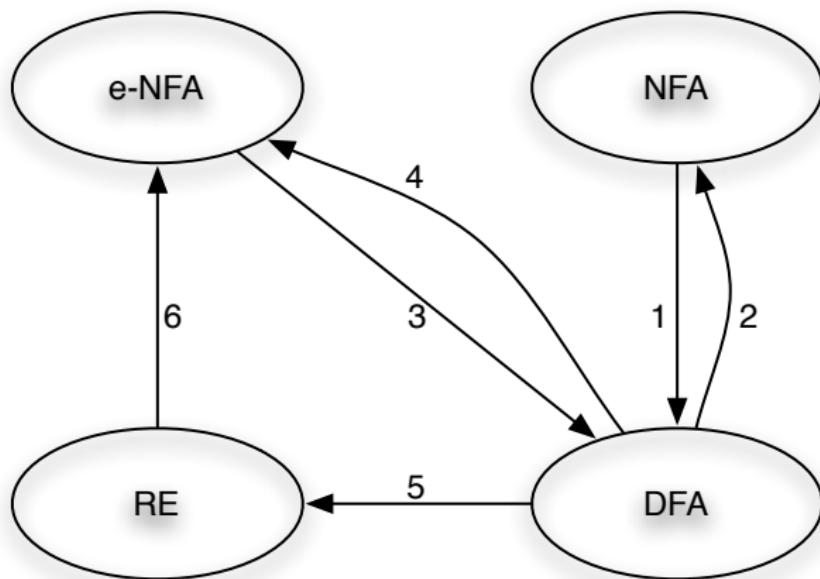
$$\hat{\delta}_E(\{q_0\}, \epsilon) = \text{eclose}(q_0) = q_D = \hat{\delta}_D(q_D, \epsilon)$$

- *Induction* : ($w = xa$)

$$\begin{aligned}\hat{\delta}_E(\{q_0\}, xa) &\stackrel{\text{def de } \hat{\delta}_E}{=} \text{eclose}(\delta_E(\hat{\delta}_E(\{q_0\}, x), a)) \\ &\stackrel{\text{h.i.}}{=} \text{eclose}(\delta_E(\hat{\delta}_D(q_D, x), a)) \\ &\stackrel{\text{cst}}{=} \delta_D(\hat{\delta}_D(q_D, x), a) \\ &\stackrel{\text{def de } \hat{\delta}_D}{=} \hat{\delta}_D(q_D, xa)\end{aligned}$$

$$\mathcal{C}(\epsilon\text{-NFA}) = \mathcal{C}(\text{DFA})$$

+Flèches 3 + 4 :



$$\mathcal{C}(\epsilon\text{-NFA}) = \mathcal{C}(\text{DFA})$$

Théorème (Un langage L est accepté par un DFA si et seulement si L est accepté par un ϵ -NFA)

Preuve :

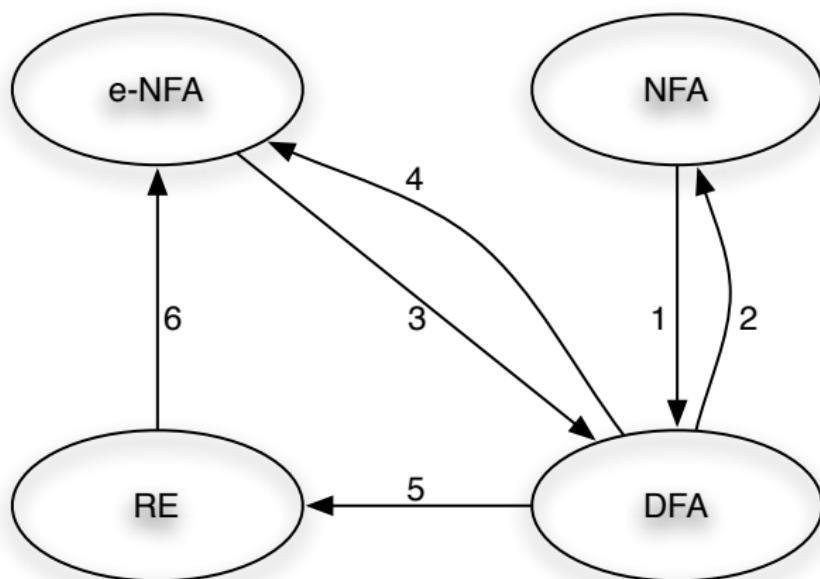
- *si* \Leftarrow voir théorème précédent
- *seulement si* \Rightarrow

Il suffit à partir du DFA D de construire un ϵ -NFA E avec

$$\text{si } \delta_D(q, a) = p \text{ alors } \delta_E(q, a) = \{p\}$$

$$\mathcal{C}(\text{DFA}) \subseteq \mathcal{C}(\text{RE})$$

+ Flèche 5 :



$$\mathcal{C}(\text{DFA}) \subseteq \mathcal{C}(\text{RE})$$

Théorème (Pour tout DFA D , il existe une RE R avec $L(R) = L(D)$)

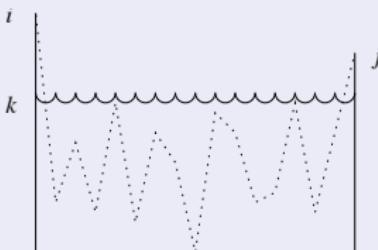
Preuve :

Soit le DFA $D = \langle Q_D, \Sigma, \delta, q_1, F \rangle$ avec :

- $Q_D = \{q_1, q_2, \dots, q_n\}$

On définit l'expression régulière R_{ij}^k décrivant l'ensemble des labels correspondants à tous les *chemins* de D

- partant de l'état q_i
- aboutissant à l'état q_j
- et ne traversant que des états de l'ensemble $\{q_1, \dots, q_k\}$



$$\mathcal{C}(\text{DFA}) \subseteq \mathcal{C}(\text{RE})$$

Théorème (Pour tout DFA D , il existe une RE R avec $L(R) = L(D)$)

Preuve : (suite)

On va définir R_{ij}^k de façon inductive. Notons que

$$L\left(\bigcup_{q_j \in F} R_{ij}^n \right) = L(D)$$

Cas de base :

- *Cas 1 : $i \neq j$*

$$R_{ij}^0 = \bigcup_{\{a \in \Sigma : \delta(q_i, a) = q_j\}} a$$

- *Cas 2 : $i = j$*

$$R_{ii}^0 = \bigcup_{\{a \in \Sigma \mid \delta(q_i, a) = q_i\}} a + \epsilon$$

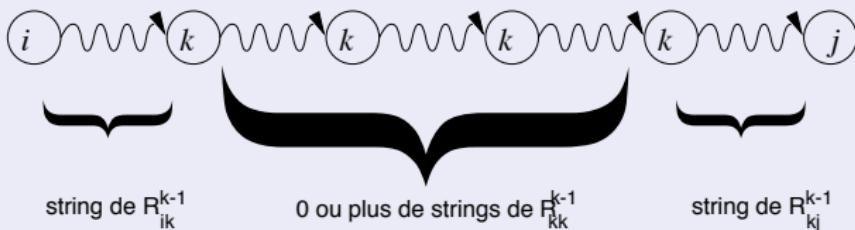
$$\mathcal{C}(\text{DFA}) \subseteq \mathcal{C}(\text{RE})$$

Théorème (Pour tout DFA D , il existe une RE R avec $L(R) = L(D)$)

Preuve : (suite 2)

Induction :(de la définition de R_{ij}^k)

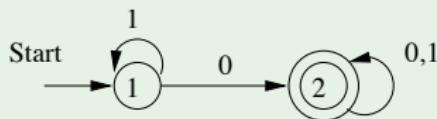
$$R_{ij}^k = R_{ij}^{k-1} + R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1}$$



$$\mathcal{C}(\text{DFA}) \subseteq \mathcal{C}(\text{RE})$$

Exemple (Pour le DFA A , construction de l'RE R avec $L(R) = L(A)$)

$$L(A) = \{x0y : x \in \{1\}^* \text{ and } y \in \{0, 1\}^*\}$$



$R_{11}^{(0)}$	$\epsilon + 1$
$R_{12}^{(0)}$	0
$R_{21}^{(0)}$	\emptyset
$R_{22}^{(0)}$	$\epsilon + 0 + 1$

$$\mathcal{C}(\text{DFA}) \subseteq \mathcal{C}(\text{RE})$$

Quelques règles de simplification des RE

- $(\epsilon + R)^* = R^*$
- $R + RS^* = RS^*$
- $\emptyset R = \emptyset = R\emptyset$
- $\emptyset + R = R + \emptyset = R$

$$\mathcal{C}(\text{DFA}) \subseteq \mathcal{C}(\text{RE})$$

Exemple (Pour le DFA A , construction de l'RE R avec $L(R) = L(A)$)

$R_{11}^{(0)}$	$\epsilon + 1$
$R_{12}^{(0)}$	0
$R_{21}^{(0)}$	\emptyset
$R_{22}^{(0)}$	$\epsilon + 0 + 1$

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}(R_{11}^{(0)})^*R_{1j}^{(0)}$$

	Par substitution directe	Simplifié
$R_{11}^{(1)}$	$\epsilon + 1 + (\epsilon + 1)(\epsilon + 1)^*(\epsilon + 1)$	1^*
$R_{12}^{(1)}$	$0 + (\epsilon + 1)(\epsilon + 1)^*0$	1^*0
$R_{21}^{(1)}$	$\emptyset + \emptyset(\epsilon + 1)^*(\epsilon + 1)$	\emptyset
$R_{22}^{(1)}$	$\epsilon + 0 + 1 + \emptyset(\epsilon + 1)^*0$	$\epsilon + 0 + 1$

$$\mathcal{C}(\text{DFA}) \subseteq \mathcal{C}(\text{RE})$$

Exemple (Pour le DFA A , construction de l'RE R avec $L(R) = L(A)$)

	Simplifié
$R_{11}^{(1)}$	1^*
$R_{12}^{(1)}$	1^*0
$R_{21}^{(1)}$	\emptyset
$R_{22}^{(1)}$	$\epsilon + 0 + 1$

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)} (R_{22}^{(1)})^* R_{2j}^{(1)}$$

	Par substitution directe
$R_{11}^{(2)}$	$1^* + 1^*0(\epsilon + 0 + 1)^*\emptyset$
$R_{12}^{(2)}$	$1^*0 + 1^*0(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$
$R_{21}^{(2)}$	$\emptyset + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*\emptyset$
$R_{22}^{(2)}$	$\epsilon + 0 + 1 + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$

$$\mathcal{C}(\text{DFA}) \subseteq \mathcal{C}(\text{RE})$$

Exemple (Pour le DFA A , construction de l'RE R avec $L(R) = L(A)$)

	Par substitution directe
$R_{11}^{(2)}$	$1^* + 1^*0(\epsilon + 0 + 1)^*\emptyset$
$R_{12}^{(2)}$	$1^*0 + 1^*0(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$
$R_{21}^{(2)}$	$\emptyset + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*\emptyset$
$R_{22}^{(2)}$	$\epsilon + 0 + 1 + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$

	Simplifié
$R_{11}^{(2)}$	1^*
$R_{12}^{(2)}$	$1^*0(0 + 1)^*$
$R_{21}^{(2)}$	\emptyset
$R_{22}^{(2)}$	$(0 + 1)^*$

La RE finale pour A est :

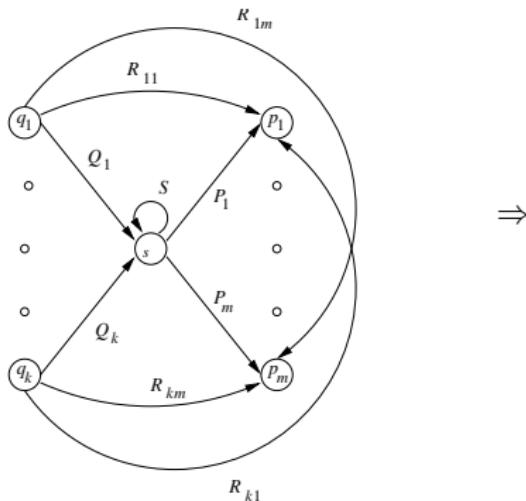
$$R_{12}^{(2)} = 1^*0(0 + 1)^*$$

$\mathcal{C}(\text{DFA}) \subseteq \mathcal{C}(\text{RE})$: Observations

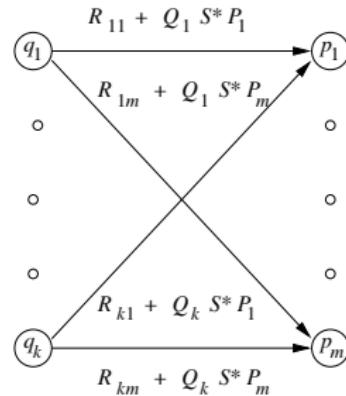
- La méthode fonctionne aussi avec des NFA ou ϵ -NFA
 - Il y a n^3 expressions R_{ij}^k
 - Chaque pas inductif quadruple la taille de l'expression
 - R_{ij}^n peut avoir une taille 4^n
 - Pour tout $\{i, j\} \subseteq \{1, \dots, n\}$, R_{ij}^k utilise R_{ij}^{k-1}
donc on doit écrire n^2 fois l'expression R_{kk}^{k-1}
- ⇒ Il faut trouver une méthode plus efficace : la **technique d'élimination d'états**

Technique :

- ① remplacer les symboles labellisant le FA en expression régulière
- ② supprimer des états (*s*)

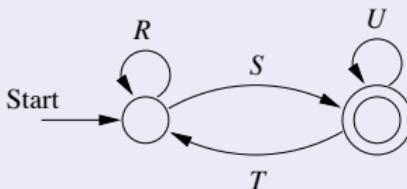


\Rightarrow



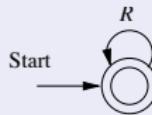
Méthode

- On construit pour chaque état accepteur q , un automate à 2 états q_0 et q en éliminant les autres
- Pour chaque $q \in F$ on obtient
 - soit A_q :



avec la RE correspondante : $E_q = (R + SU^* T)^* SU^*$

- soit A_q :



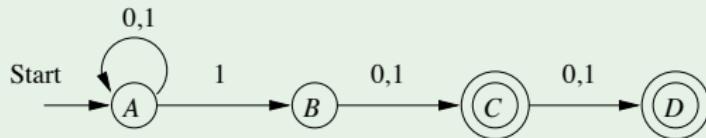
avec la RE correspondante : $E_q = R^*$

- la RE finale est : $\bigcup_{q \in F} E_q$

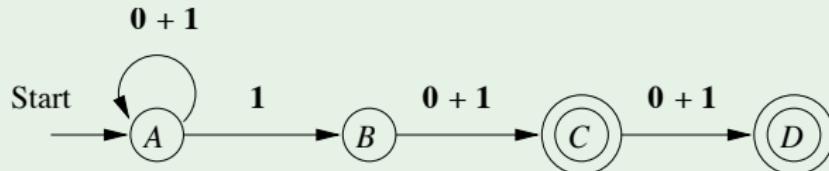
$\mathcal{C}(\text{FA}) \subseteq \mathcal{C}(\text{RE})$: par élimination des états

Exemple (trouvons une RE du NFA A par élimination d'états)

NFA \mathcal{A}



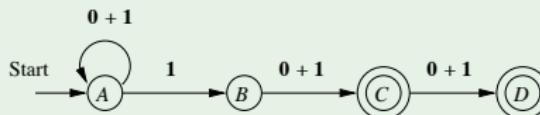
Transformation de \mathcal{A} en automate labellé par des RE :



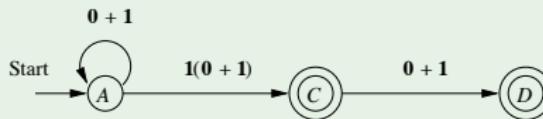
$\mathcal{C}(\text{FA}) \subseteq \mathcal{C}(\text{RE})$: par élimination des états

Exemple (suite)

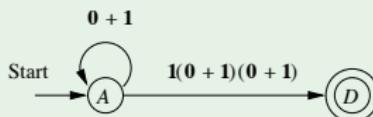
\mathcal{A} transformé :



Élimination de l'état B



Élimination de l'état C pour obtenir \mathcal{A}_D

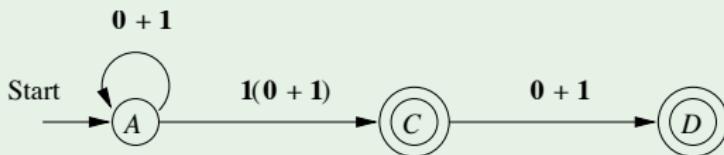


RE correspondante : $(0 + 1)^*1(0 + 1)(0 + 1)$

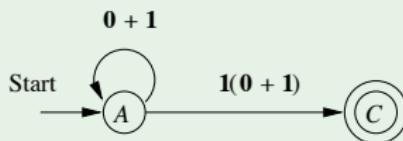
$\mathcal{C}(\text{FA}) \subseteq \mathcal{C}(\text{RE})$: par élimination des états

Exemple (trouvons une RE du FA A par élimination d'états)

A partir de l'automate avec B supprimé :



Élimination de l'état D pour obtenir \mathcal{A}_C

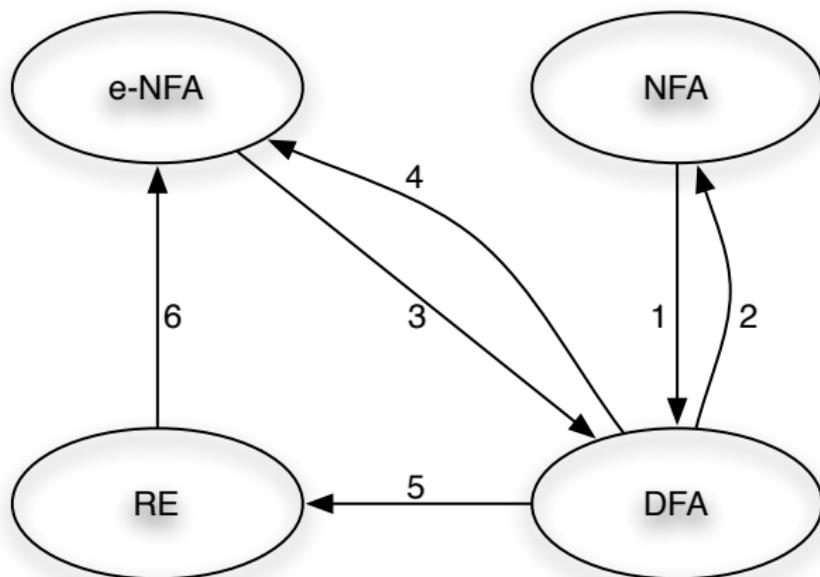


RE correspondante : $(0 + 1)^*1(0 + 1)$

RE finale : $(0 + 1)^*1(0 + 1)(0 + 1) + (0 + 1)^*1(0 + 1)$

$$\mathcal{C}(\text{RE}) \subseteq \mathcal{C}(\epsilon\text{-NFA}) = \mathcal{C}(\text{FA})$$

+ Flèche 6 :

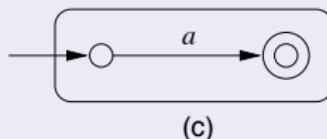
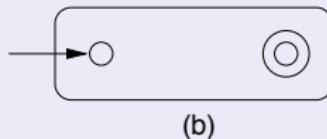
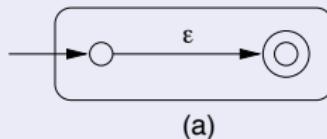


$$\mathcal{C}(\text{RE}) \subseteq \mathcal{C}(\epsilon\text{-NFA})$$

Théorème (Pour toute RE r , il existe un ϵ -NFA R avec $L(R) = L(r)$)

Preuve : par induction.

- *Cas de base : automates pour ϵ , \emptyset et a :*

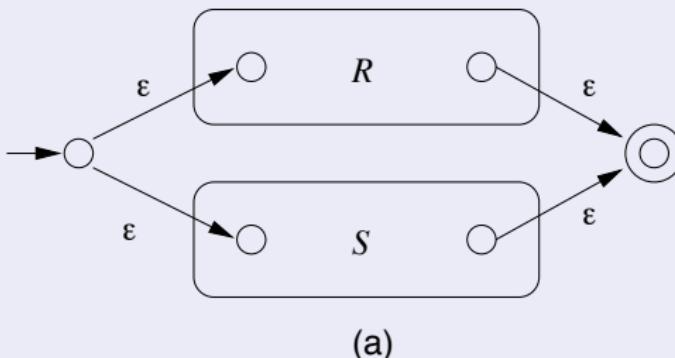


$\mathcal{C}(\text{RE}) \subseteq \mathcal{C}(\epsilon\text{-NFA})$ (suite)

Théorème (Pour toute RE r , il existe un ϵ -NFA R avec $L(R) = L(r)$)

Preuve : (suite)

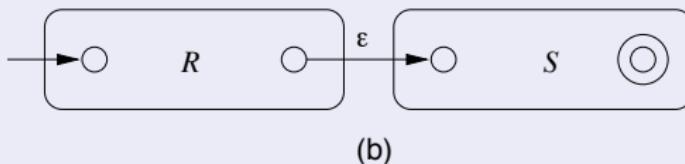
- *Induction : automate pour $r + s$:*



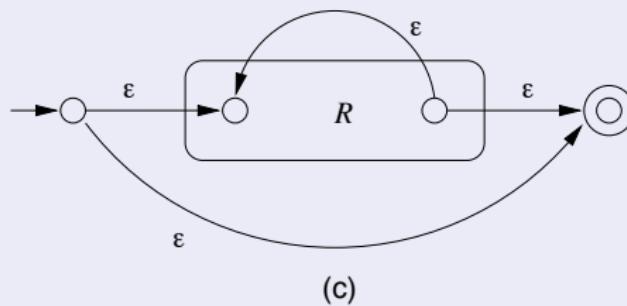
Théorème (Pour toute RE r , il existe un ϵ -NFA R avec $L(R) = L(r)$)

Preuve : (suite)

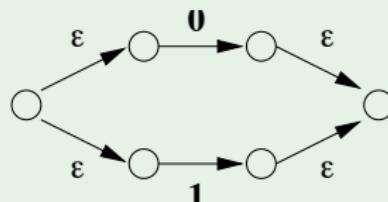
- *Induction : automates pour rs , et r^* :*



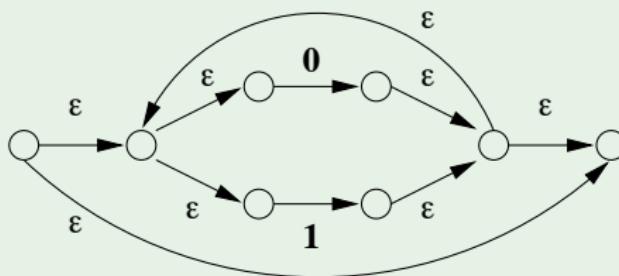
(b)



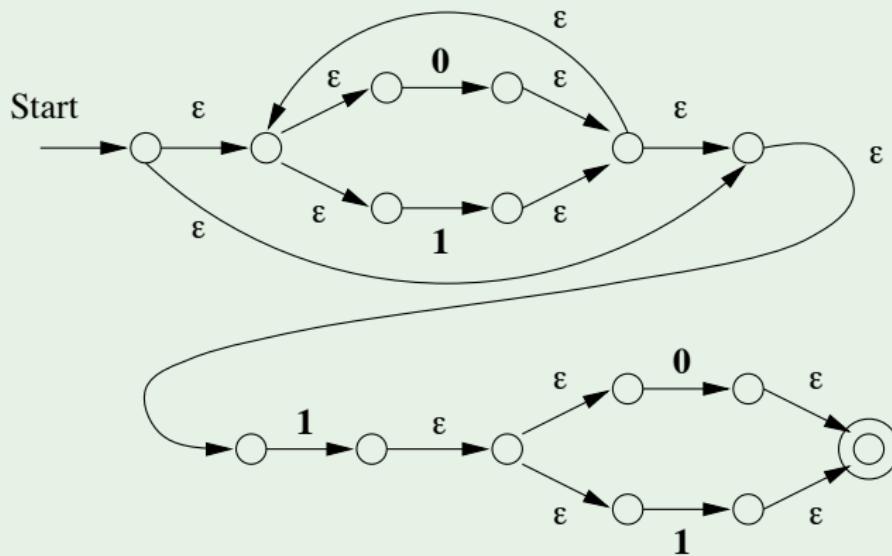
(c)

Exemple (ϵ -NFA correspondant à $(0 + 1)^*1(0 + 1)$)

(a)

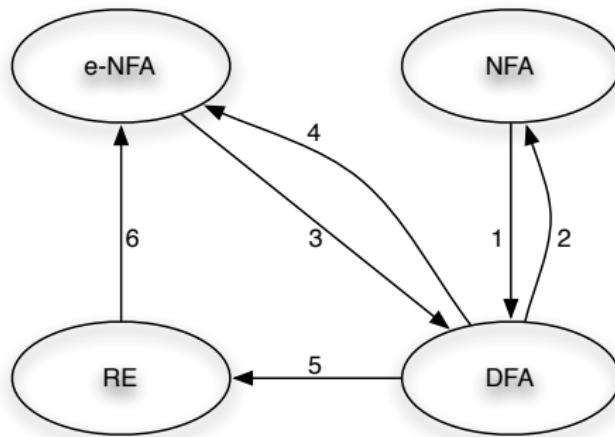


(b)

$\mathcal{C}(\text{RE}) \subseteq \mathcal{C}(\epsilon\text{-NFA})$ (suite)Exemple (ϵ -NFA correspondant à $(0 + 1)^*1(0 + 1)$ (suite))

(c)

$$\mathcal{C}(\text{RE}) = \mathcal{C}(\epsilon\text{-NFA}) = \mathcal{C}(\text{NFA}) = \mathcal{C}(\text{DFA})$$



En conclusion,

- les 4 formalismes sont équivalents et définissent la classe des langages réguliers
- on peut passer de l'un à l'autre par des traductions automatiques

Plan

- 1 Les langages réguliers et expressions régulières
- 2 Les automates finis
- 3 Équivalence entre FA et RE
- 4 Autres types d'automates finis
- 5 Propriétés des langages réguliers
- 6 Équivalence et minimisation d'automates finis

Machines avec output (actions)

On a souvent envie de **modéliser** un système plus “réactif” qui effectue des actions ou produit des outputs lors du traitement.

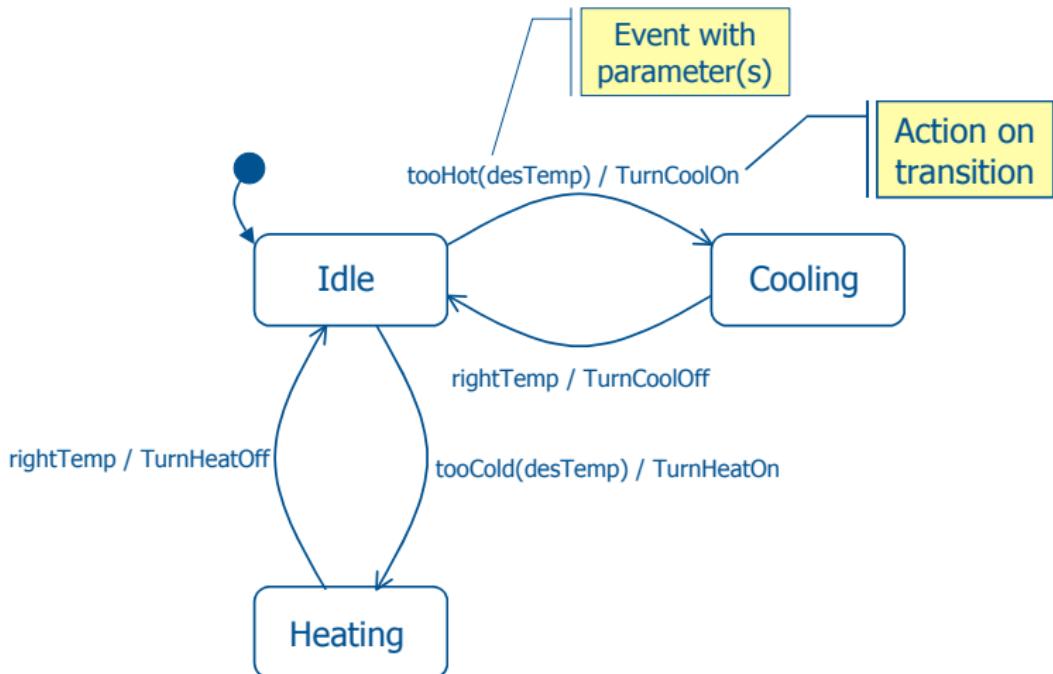
En théorie 2 types standards d'automates finis avec output ont été définis.

- les **machines de Moore** qui ont un output associé à chaque état de contrôle
- les **machines de Mealy** qui ont un output associé à chaque transition

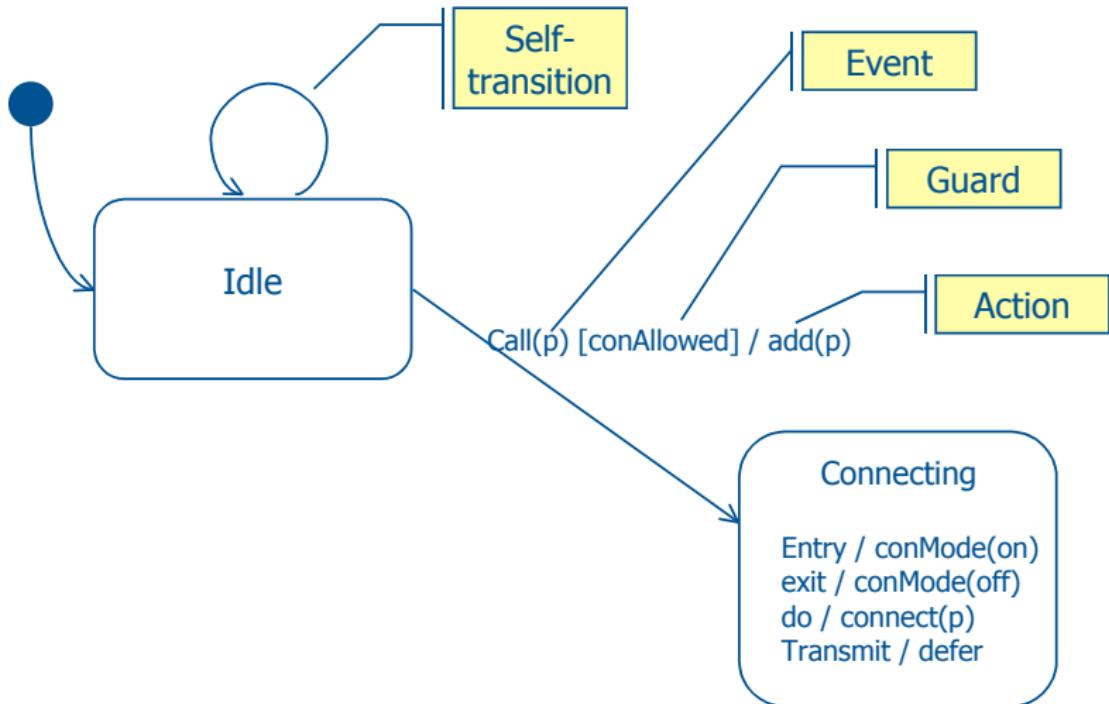
UML a retenu et mélangé tous ces concepts dans ses

- **diagrammes d'état (statecharts)**
- **diagrammes d'activité**

Exemple de statechart UML



Exemple de statechart UML (2)



Plan

- 1 Les langages réguliers et expressions régulières
- 2 Les automates finis
- 3 Équivalence entre FA et RE
- 4 Autres types d'automates finis
- 5 **Propriétés des langages réguliers**
- 6 Équivalence et minimisation d'automates finis

Questions qu'on peut se poser sur des langages L , L_1 , L_2

- L est-il régulier ?
- Pour quels opérateurs les langages réguliers sont-ils fermés ?
- $w \in L$?
- L est-il vide, fini, infini ?
- $L_1 \subseteq L_2$, $L_1 = L_2$?

L est-il régulier ?

Pour montrer que L est régulier

Il faut :

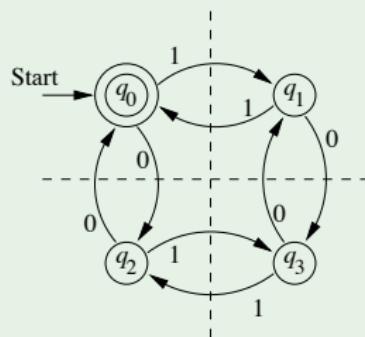
- trouver un FA (ou une RE) M et
- montrer que $L = L(M)$ càd
 - que $L \subseteq L(M)$ (tout string de L est accepté par M)
 - que $L(M) \subseteq L$ (tout string accepté par M est dans L)

L est-il régulier ?

Exemple

Pour montrer que L est régulier $L = \{w \mid w \text{ a un nombre pair de } 0 \text{ et de } 1\}$

On définit par exemple le DFA M et démontre (généralement par induction) que $L = L(M)$



L est-il régulier ?

Pour montrer que L n'est pas régulier

Parfois on peut le montrer grâce au **pumping lemma (lemme de pompage)** des langages réguliers

Théorème (Pumping lemma des langages réguliers)

*Si L est régulier
alors*

$\exists n \forall w (w \in L \wedge |w| \geq n) \Rightarrow \exists x, y, z$

- ① $w = xyz \wedge$
- ② $|xy| \leq n \wedge$
- ③ $y \neq \epsilon \wedge$
- ④ $\forall k \geq 0 \Rightarrow xy^kz \in L$

Preuve du pumping lemma pour les langages réguliers

Théorème (Pumping lemma des langages réguliers)

Preuve :

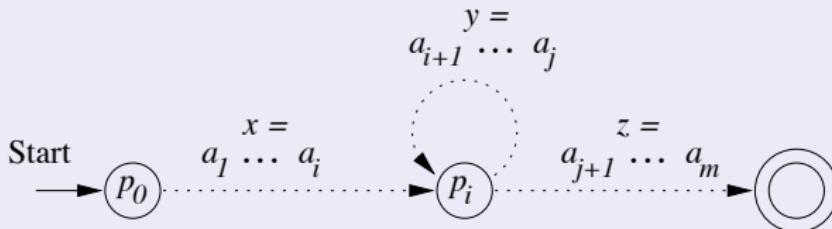
Supposons L régulier

L est reconnu par un DFA M . Soit n le nombre d'états de M

Prenons $w = a_1 a_2 \dots a_m \in L$ avec $m \geq n$

Dénotons $p_i = \hat{\delta}(q_0, a_1 a_2 \dots a_i)$ (et $p_0 = q_0$)

$\Rightarrow \exists i < j : p_i = p_j$!! (on prend $p_i =$ le premier état accédé pour la seconde fois et donc $j \leq n$) !!



Et donc $w = xyz$ et pour x, y, z on peut facilement vérifier que toutes les propriétés du pumping lemma sont bien vérifiées.

Comment le pumping lemma permet de dire que L n'est pas régulier

Pumping lemma = condition nécessaire pour que L régulier

L régulier $\Rightarrow L$ satisfait au PL

L non régulier $\Leftarrow L$ ne satisfait pas au PL

Négation du pumping lemma

$L \neg$ -régulier \Leftarrow

$\neg \exists n \forall w ((w \in L \wedge |w| \geq n) \Rightarrow (\exists x, y, z$

- ① $w = xyz \wedge$
- ② $|xy| \leq n \wedge$
- ③ $y \neq \epsilon \wedge$
- ④ $\forall k \geq 0 \Rightarrow xy^k z \in L))$

$\equiv \forall n \exists w \neg((w \in L \wedge |w| \geq n) \Rightarrow (\exists x, y, z$

- ① $w = xyz \wedge$
- ② $|xy| \leq n \wedge$
- ③ $y \neq \epsilon \wedge$
- ④ $\forall k \geq 0 \Rightarrow xy^k z \in L))$

Négation du pumping lemma (suite)

$\equiv \forall n \exists w (w \in L \wedge |w| \geq n) \wedge \neg(\exists x, y, z$

- ① $w = xyz \wedge$
- ② $|xy| \leq n \wedge$
- ③ $y \neq \epsilon \wedge$
- ④ $\forall k \geq 0 \Rightarrow xy^kz \in L)$

$\equiv \forall n \exists w (w \in L \wedge |w| \geq n) \wedge \forall x, y, z ($

- ① $w \neq xyz \vee$
- ② $|xy| > n \vee$
- ③ $y = \epsilon \vee$
- ④ $\exists k \geq 0 \wedge xy^kz \notin L)$

Comment le pumping lemma permet de dire que L n'est pas régulier

Négation du pumping lemma (suite)

$\equiv \forall n \exists w (w \in L \wedge |w| \geq n) \wedge \forall x, y, z$

- ① $(w = xyz \wedge$
- ② $|xy| \leq n \wedge$
- ③ $y \neq \epsilon)$
- ④ $\Rightarrow (\exists k \geq 0 \wedge xy^k z \notin L)$

Exemple (Montrons que $L = \{0^i 1^i | i \in \mathbb{N}\}$ n'est pas régulier)

Montrons que L satisfait la négation des conditions du pumping lemma.

Pour tout n , pour $w = 0^n 1^n \in L \wedge |w| \geq n$

et $\forall x, y, z (w = xyz \wedge |xy| \leq n \wedge y \neq \epsilon : x = 0^k, y = 0^\ell (\ell \neq 0))$
 $\Rightarrow xz \notin L$

$$w = \underbrace{000 \cdots}_{x} \cdots \underbrace{0}_{y} \underbrace{0111 \cdots 11}_{z}$$

Comment le pumping lemma permet de dire que L n'est pas régulier

Exemple (Montrons que $L = \{0^{i^2} \mid i \in \mathbb{N}\}$ n'est pas régulier)

Montrons que L satisfait la négation des conditions du pumping lemma.

Pour tout n , pour $w = 0^{n^2} \in L \wedge |w| \geq n$
et $\forall x, y, z (w = xyz \wedge |xy| \leq n \wedge y \neq \epsilon)$
 $\Rightarrow xy^2z \notin L$ car $n^2 < |xy^2z| \leq n^2 + n < (n+1)^2$

Pour quels opérateurs les langages réguliers sont-ils fermés ?

Théorème (Si L et M sont réguliers alors, sont réguliers)

- *Union* : $L \cup M$
- *Concaténation* : $L.M$
- *Fermeture de Kleene* : L^*
- *Complément* : \bar{L}
- *Intersection* : $L \cap M$
- *Difference* : $L \setminus M$
- *Image miroir* : L^R

Pour quels opérateurs les langages réguliers sont-ils fermés ?

Théorème (Si L et M sont réguliers alors, sont réguliers)

Preuve :

- Union : $L \cup M$ (*Par définition*)
- Concaténation : $L.M$ (*Par définition*)
- Fermeture de Kleene : L^* (*Par définition*)

Pour quels opérateurs les langages réguliers sont-ils fermés ?

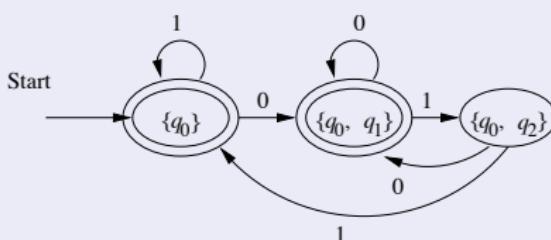
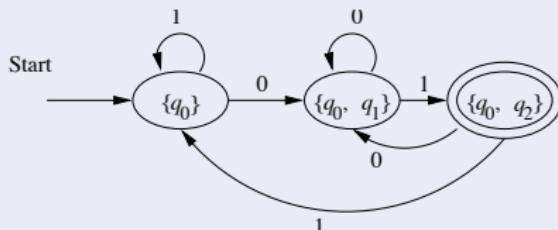
Théorème (Si L est régulier alors \bar{L} est régulier)

Preuve :

Si L est reconnu par le DFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$

On a $\bar{L} = L(B)$ pour $B = \langle Q, \Sigma, \delta, q_0, Q \setminus F \rangle$

Exemple :



Pour quels opérateurs les langages réguliers sont-ils fermés ?

Théorème (Si L et M sont réguliers alors $L \cap M$ est régulier)

Preuve :

- grâce à la loi de De Morgan $L \cap M = \overline{\overline{L} \cup \overline{M}}$
- et le fait que les langages réguliers sont fermés pour l'union et le complément

Pour quels opérateurs les langages réguliers sont-ils fermés ?

Théorème (Si L et M sont réguliers alors $L \cap M$ est régulier)

Autre preuve : (directe et constructive)

- Ayant pour L le DFA A_L avec $L(A_L) = L : A_L = \langle Q_L, \Sigma, \delta_L, q_L, F_L \rangle$
- et pour M le DFA A_M avec $L(A_M) = M : A_M = \langle Q_M, \Sigma, \delta_M, q_M, F_M \rangle$
- on définit le DFA $A_{L \cap M}$

$$A_{L \cap M} = \langle Q_L \times Q_M, \Sigma, \delta_{L \cap M}, (q_L, q_M), F_L \times F_M \rangle$$

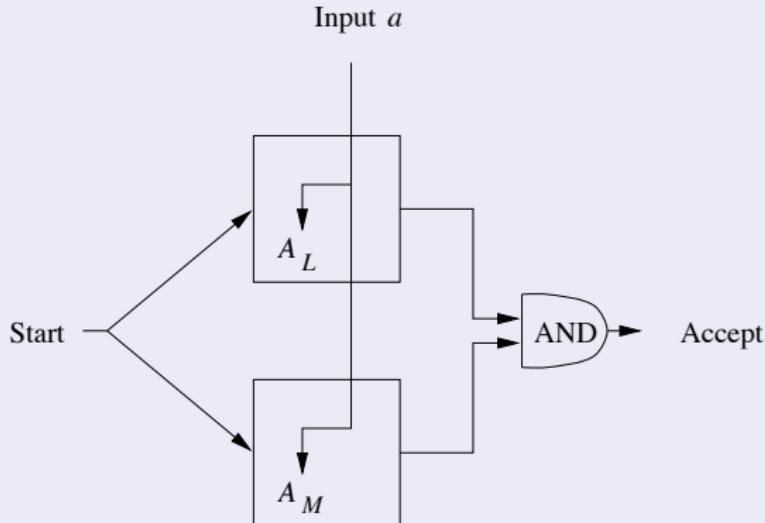
avec

$$\delta_{L \cap M}((p, q), a) = (\delta_L(p, a), \delta_M(q, a))$$

On peut démontrer que $A_{L \cap M}$ accepte $L \cap M$

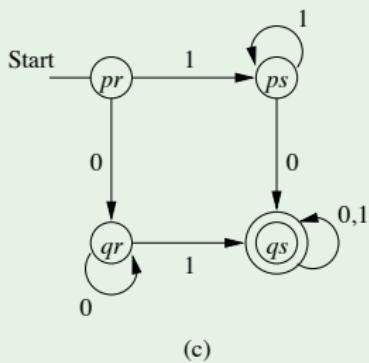
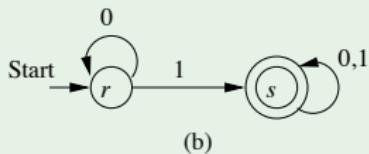
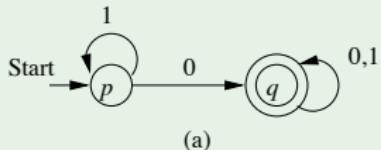
Pour quels opérateurs les langages réguliers sont-ils fermés ?

Théorème ($L \cap M$ par composition de L et M)



Pour quels opérateurs les langages réguliers sont-ils fermés ?

Exemple $((c) = (a) \times (b))$



Les langages réguliers et expressions régulières
 Les automates finis
 Équivalence entre FA et RE
Autres types d'automates finis
Propriétés des langages réguliers
Équivalence et minimisation d'automates finis

Pour quels opérateurs les langages réguliers sont-ils fermés ?

Théorème (Si L et M sont réguliers alors $L \setminus M$ est régulier)

Preuve :

$$L \setminus M = L \cap \overline{M}$$

Pour quels opérateurs les langages réguliers sont-ils fermés ?

Théorème (Si L est régulier alors L^R est régulier)

Preuve : L est reconnu par un FA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$

À partir de A on peut construire un FA $B = \langle Q \cup \{p_0\}, \Sigma, \delta', p_0, F' \rangle$ avec

- p_0 un nouvel état
- δ' est construit à partir de δ en inversant tous les arcs
- rajoutant $\delta(p_0, \epsilon) = F$
- $F' = \{q_0\}$

On peut prouver que $L(B) = L^R$

$w \in L$, L vide, fini, infini ?, $L_1 \subseteq L_2$, $L_1 = L_2$

Comment tester que :

- $w \in L$? : voir si un DFA A avec $L = L(A)$ accepte w
- L vide ? ($L = \emptyset$) : voir si un état accepteur est accessible à partir de l'état initial de A
- $L = \Sigma^*$? : voir si l'ensemble des états accepteurs du DFA qui reconnaît L = l'ensemble des états accessibles.
- L infini ? : voir si à partir de l'état initial on peut atteindre un état accepteur en passant par un état dans un cycle
- $L_1 \subseteq L_2 \iff L_1 \cap \overline{L_2} = \emptyset$
- $L_1 = L_2 \iff L_1 \subseteq L_2 \wedge L_2 \subseteq L_1$

Plan

- 1 Les langages réguliers et expressions régulières
- 2 Les automates finis
- 3 Équivalence entre FA et RE
- 4 Autres types d'automates finis
- 5 Propriétés des langages réguliers
- 6 Équivalence et minimisation d'automates finis

Équivalence d'états

Soit un DFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ et $\{p, q\} \subseteq Q$.

Définition ($p \equiv q$ (p équivalent à q))

$$p \equiv q \iff (\forall w \in \Sigma^* : \hat{\delta}(p, w) \in F \iff \hat{\delta}(q, w) \in F)$$

Si $p \not\equiv q$ on dit que p et q sont distinguables

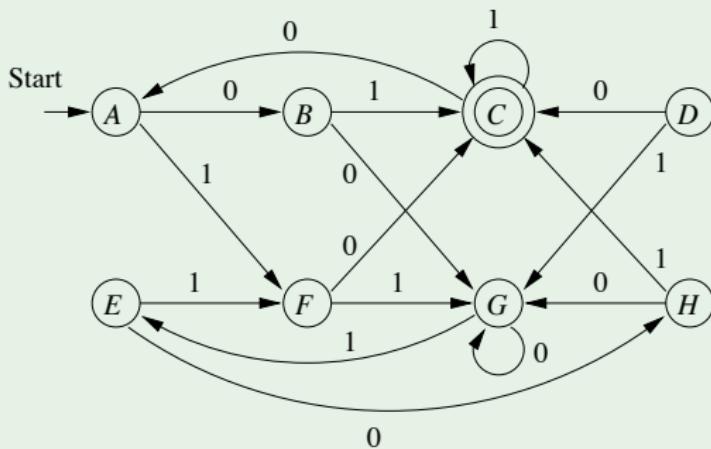
On a $p \not\equiv q \iff$

$$\exists w : \hat{\delta}(p, w) \in F \iff \hat{\delta}(q, w) \notin F \text{ ou vice-versa}$$

Intuitivement, on accepte le même langage à partir de p et de q

Équivalence d'états

Exemple (Équivalence d'états (pour le DFA $M = \langle Q, \Sigma, \delta, A, F_M \rangle$))



- $\hat{\delta}(C, \epsilon) \in F_M \wedge \hat{\delta}(G, \epsilon) \notin F_M \Rightarrow C \not\equiv G$
- $\hat{\delta}(A, 01) = C \in F_M \wedge \hat{\delta}(G, 01) = E \notin F_M \Rightarrow A \not\equiv G$
- on peut, par exemple, voir que $A \equiv E$

Équivalence d'états

Algorithme de remplissage de table pour calculer les pairs distinctes

Base : $p \in F \wedge q \notin F \iff \text{DISTINCT}[p, q] \leftarrow \text{Vrai}$

Induction : si $\exists a \in \Sigma : \text{DISTINCT}[\delta(p, a), \delta(q, a)]$
alors $\text{DISTINCT}[p, q] \leftarrow \text{Vrai}$

Exemple : pour le DFA M

B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x
	A	B	C	D	E	F	G

Équivalence d'états

Théorème (Si p et q ne sont pas distingués par l'algorithme de remplissage de table alors $p \equiv q$)

Preuve : (par l'absurde)

On suppose le contraire : il existe une paire $\{p, q\}$ telle que :

- ① $\exists w \in \Sigma^* : \hat{\delta}(p, w) \in F \wedge \hat{\delta}(q, w) \notin F$
- ② *l'algorithme de remplissage de table ne distingue pas p de q*

(C) Soit $w = a_1 a_2 \dots a_n$ *le string le plus court qui différentie une paire $\{p, q\}$ que l'algorithme de remplissage ne distingue pas*

- $w \neq \epsilon$ sinon l'algorithme distinguerait p de q (donc $n \geq 1$)
- Prenons $r = \delta(p, a_1)$ et $s = \delta(q, a_1)$
 $\{r, s\}$ ne sont pas équivalents avec $x = a_2 \dots a_n$ plus court que w et par (C) l'algorithme doit les distinguer.
Mais alors l'algorithme doit distinguer $\{p, q\}$ dans sa partie inductive.
Contradiction

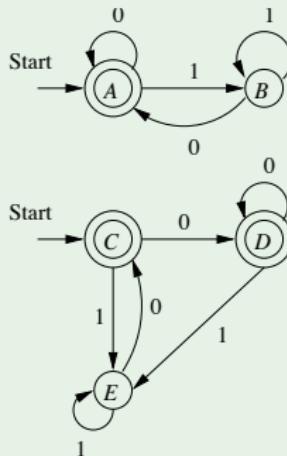
Équivalence de languages (algorithme direct)

Algorithme pour déterminer si 2 Langages réguliers sont équivalents

- Construire des DFAs pour L et pour M (leurs ensembles d'états étant disjoints)
- Regarder le DFA formé de l'union des états de L et de M (l'état initial n'est pas important)
- Faire tourner l'algorithme de remplissage de table
- Si les 2 états initiaux sont équivalents, $L \equiv M$

Equivalence d'états

Exemple (Les 2 Langages réguliers sont équivalents)



B	x			
C		x		
D		x		
E	x	x	x	
	A	B	C	D

$A \equiv C$ et donc $L \equiv M$

Minimisation de DFAs

- On peut facilement voir que la relation \equiv est une relation d'équivalence.
- Notons p/\equiv la classe d'équivalence de \equiv qui contient l'état p
- et Q/\equiv (resp. F/\equiv) l'ensemble des classes d'équivalence de Q (resp. F) pour la relation \equiv

Algorithme de minimisation d'un DFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$

- ➊ Faire fonctionner l'algorithme de remplissage de table
- ➋ fusionner les états équivalents : on construit $B = \langle Q/\equiv, \Sigma, \gamma, q_0/\equiv, F/\equiv \rangle$ avec

$$\gamma(p/\equiv, a) = \delta(p, a)/\equiv$$

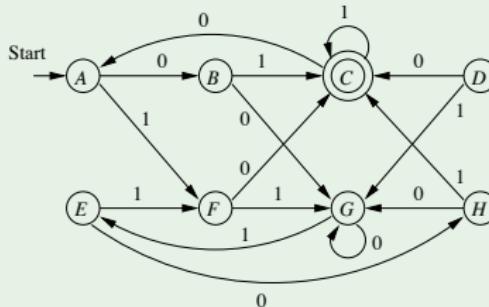
Pour que B soit bien défini, il faut montrer que

$$\text{si } p \equiv q \text{ alors } \forall a, \delta(p, a) \equiv \delta(q, a)$$

Sinon l'algorithme de remplissage de table aurait conclu que $p \not\equiv q$

Minimisation de DFAs

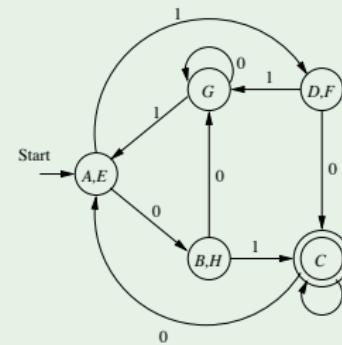
Exemple (de minimisation)



est minimisé en

B	x
C	x x
D	x x x
E	x x x
F	x x x x
G	x x x x x
H	x x x x x x

A B C D E F G



Pourquoi la minimisation de DFAs ne peut être battue

- Prenons B le DFA obtenu grâce à l'algorithme de minimisation à partir de A
- On sait que $L(A) = L(B)$
- Supposons C plus petit que B avec $L(C) = L(B)$
- Faisons fonctionner l'algorithme de remplissage de table sur $B \cup C$
- Comme $L(B) = L(C)$ on doit avoir $q_0^B \equiv q_0^C$ et $\forall a, \delta(q_0^B, a) \equiv \delta(q_0^C, a)$
- Étant donné qu'aucun état de B n'est inaccessible
on peut montrer que pour tout état p de B il existe un état q de C tel que $p \equiv q$
- si C plus petit que B il doit exister des états r et s de B , t de C avec $r \equiv t \equiv s$
- Mais alors $r \equiv s$ et l'algorithme aurait dû les fusionner (**contradiction**)

Chapitre 3 : L'analyse lexicale (scanning)

- 1 Rôles et place de l'analyse lexicale (scanning)
- 2 Éléments à traiter
- 3 Extension des expressions régulières
- 4 Construction d'un analyseur lexical à la main
- 5 Construction d'un analyseur lexical avec (f)lex

Rôles et place de l'analyse lexicale (scanning)

Éléments à traiter

Extension des expressions régulières

Construction d'un analyseur lexical à la main

Construction d'un analyseur lexical avec (f)lex

Plan

1 Rôles et place de l'analyse lexicale (scanning)

2 Éléments à traiter

3 Extension des expressions régulières

4 Construction d'un analyseur lexical à la main

5 Construction d'un analyseur lexical avec (f)lex

Rôle de l'analyse lexicale (scanning)

- ① Identifie les **lexèmes** et les **unités lexicales** correspondantes (**Rôle principal**)
- ② Met (éventuellement) les identificateurs (non prédéfinis) et littéraux dans la table des symboles¹
- ③ Produit le listing / est lié à un éditeur intelligent
- ④ Nettoie le programme source (supprime les commentaires, espaces, tabulations, majuscules, ...) : joue le rôle de **filtre**

¹peut se faire lors de d'une phase ultérieure d'analyse

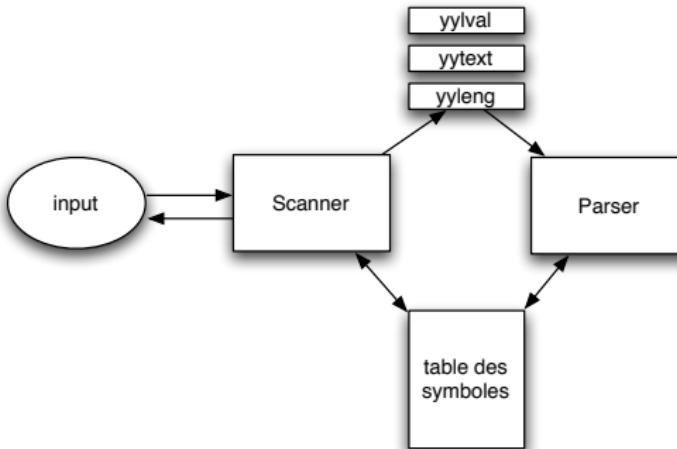
Lexème, unité lexicale, modèle

Définitions

- **Unité lexicale (ou token)** : Type générique d'éléments lexicaux (correspond à un ensemble de strings ayant une sémantique proche).
Exemple : identificateur, opérateur relationnel, mot clé "begin"...
- **Lexème (ou string)** : Occurrence d'une unité lexicale.
Exemple : N est un lexème dont l'unité lexicale est *identificateur*
- **Modèle (pattern)** : Règle décrivant une unité lexicale
Exemple : identificateur = lettre (lettre + chiffre)*

Relation entre lexème, unité lexicale et modèle

unité lexicale = { lexème | modèle(lexème) }



- travaille avec l'input ⇒ il faut optimiser la lecture (bufferisation) pour ne pas y passer trop de temps
- co-routine du parser qui lui demande des tokens et reçoit :
 - ➊ l'unité lexicale reconnue
 - ➋ des informations (nom du lexème correspondant) dans la table des symboles
 - ➌ des valeurs dans des variables globales spécifiques (ex : yyval, yytext, yylen)

Frontière entre scanning et parsing

La frontière entre scanning et parsing est parfois floue

- Du point de vue logique :
 - Au scanning : on reconnaît des lexèmes / unités lexicales
 - Au parsing : on construit l'arbre syntaxique
- Du point de vue technique :
 - Au scanning : on manipule des expressions régulières et l'analyse est locale
 - Au parsing : on manipule des grammaires context free et l'analyse est globale

Remarques :

- Parfois le scanning compte les parenthèses (lien avec un éditeur intelligent)
- Exemple compliqué pour le scanning : en FORTRAN
`DO 5 I = 1,3` différent de `DO 5 I = 1.3`
⇒ nécessite une lecture anticipée

Rôles et place de l'analyse lexicale (scanning)

Éléments à traiter

Extension des expressions régulières

Construction d'un analyseur lexical à la main

Construction d'un analyseur lexical avec (f)lex

Plan

1 Rôles et place de l'analyse lexicale (scanning)

2 Éléments à traiter

3 Extension des expressions régulières

4 Construction d'un analyseur lexical à la main

5 Construction d'un analyseur lexical avec (f)lex

Eléments à traiter

1 Les unités lexicales : règles générales :

- Le scanner reconnaît le lexème le plus long reconnaissable :
 - Pour <= le scanner ne doit pas s'arrêter à <
 - Pour x36estune variable le scanner ne doit pas s'arrêter à x
- Les "mots clés/réservés" (if, then, while) sont dans le modèle "identificateur"
⇒ le scanner doit reconnaître en priorité les mots clés (if36x doit bien sûr être reconnu comme un identificateur)

2 Les séparateurs : (espaces, tabulation, <CR>), sont soit écartés, soit traités comme tokens vides (reconnus comme tokens par le scanner mais non transmis)

3 Les erreurs : le scanner peut essayer de se resynchroniser pour détecter d'autres erreurs éventuelles (mais aucun code ne sera généré)

Rôles et place de l'analyse lexicale (scanning)

Éléments à traiter

Extension des expressions régulières

Construction d'un analyseur lexical à la main

Construction d'un analyseur lexical avec (f)lex

Plan

1 Rôles et place de l'analyse lexicale (scanning)

2 Éléments à traiter

3 Extension des expressions régulières

4 Construction d'un analyseur lexical à la main

5 Construction d'un analyseur lexical avec (f)lex

En Lex ou UNIX

Symbole	Signification
x	Le caractere 'x'
.	N'importe quel caractere sauf \n
[xyz]	Soit x, soit y, soit z
[^bz]	Tous les caracteres, SAUF b et z
[a-z]	N'importe quel caractere entre a et z
[^a-z]	Tous les caracteres, SAUF ceux compris entre a et z
R*	Zero R ou plus, ou R est n'importe quelle expression reguliere
R+	Un R ou plus
R?	Zero ou un R (c'est-a-dire un R optionnel)
R{2,5}	Entre deux et cinq R
R{2,}	Deux R ou plus
R{2}	Exactement deux R
"[xyz\"foo"	La chaine '[xyz"foo'
{NOTION}	L'expansion de la notion NOTION definie plus haut
\X	Si X est un 'a', 'b', 'f', 'n', 'r', 't', ou 'v', represente l'interpretation ANSI-C de \X.
\0	Caractere ASCII 0
\123	Caractere ASCII dont le numero est 123 EN OCTAL
\x2A	Caractere ASCII en hexadecimal
RS	R suivi de S
R S	R ou S
R/S	R, seulement s'il est suivi par S
^R	R, mais seulement en debut de ligne
R\$	R, mais seulement en fin de ligne
<<EOF>>	Fin de fichier

Exemple de définition d'unités lexicales

Exemple (d'expressions régulières étendues)

```
blancs      [\t\n ]+
lettre      [A-Za-z]
chiffre    [0-9]      /* base 10 */
chiffre16  [0-9A-Fa-f]  /* base 16 */
mot-cle-if  if
identificateur {lettre} (_|{lettre}|{chiffre})*
entier     {chiffre}+
exposant   [eE][+-]?{entier}
reel       {entier} ("."{entier})?{exposant}?
```

Toutes ces expressions étendues peuvent être traduites en expressions régulières de base (et en FA)

Plan

- 1 Rôles et place de l'analyse lexicale (scanning)
- 2 Éléments à traiter
- 3 Extension des expressions régulières
- 4 **Construction d'un analyseur lexical à la main**
- 5 Construction d'un analyseur lexical avec (f)lex

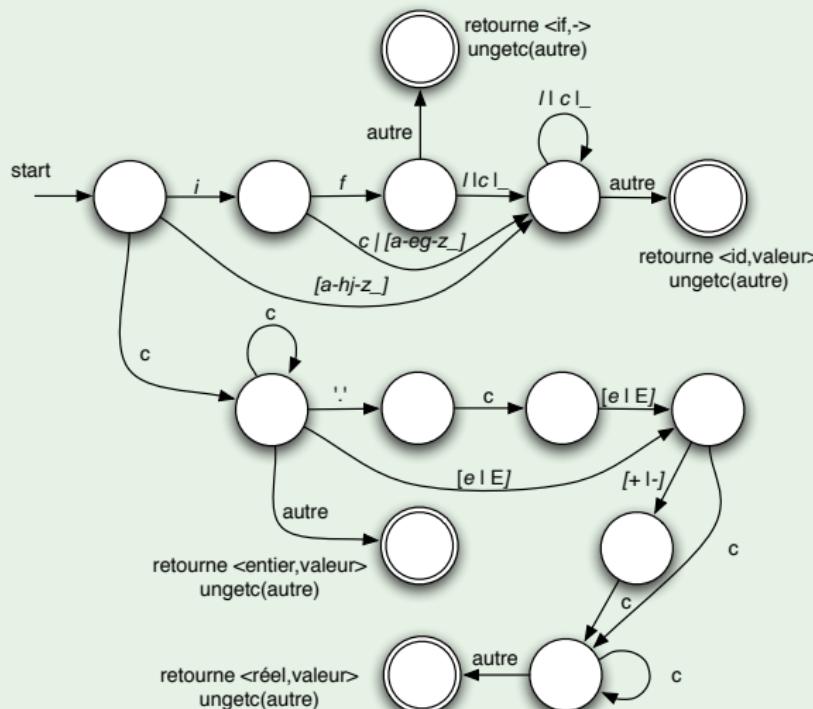
Construction d'un analyseur lexical à la main

Principe de la construction du scanner

- On part des descriptions sous forme d'expressions régulières étendues
- On "traduit" sous forme d'automate fini "déterministe"
- Cet automate est enrichi d'actions (renvoi de résultats) et de retours en arrière éventuels au dernier état accepteur retenu.

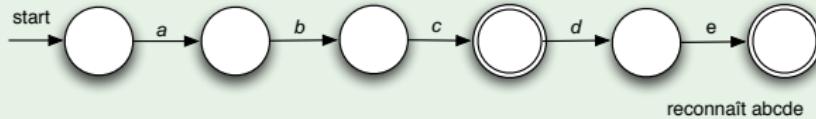
Exemple

Exemple (scanner qui reconnaît : if, identificateur, entier et réel)



Exemple où il faut retenir la dernière configuration finale

Exemple (scanner pour $abc|abcde|\dots$)

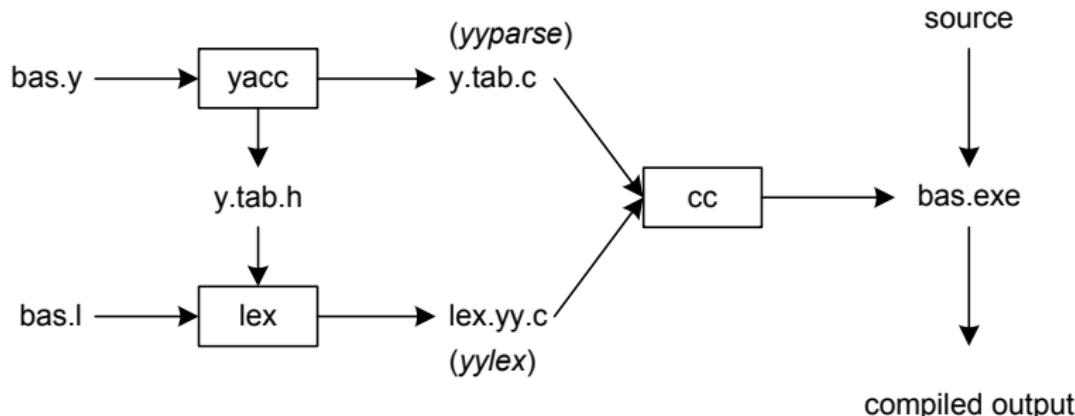


Pour le string $abcdx$, il faut accepter abc et relire dx

Plan

- 1 Rôles et place de l'analyse lexicale (scanning)
- 2 Éléments à traiter
- 3 Extension des expressions régulières
- 4 Construction d'un analyseur lexical à la main
- 5 **Construction d'un analyseur lexical avec (f)lex**

Procédure générale pour l'utilisation de Lex (Flex) et Yacc (Bison)



Compilation :

```
yacc -d bas.y          # crée y.tab.h et y.tab.c
lex bas.l              # crée lex.yy.c
cc lex.yy.c y.tab.c -ll -obas.exe # compile et fait l'édition
                                # crée bas.exe
```

Spécification Lex

définitions

%%

règles

%%

code additionnel

L'analyseur lexical résultant (`yylex()`) cherche à reconnaître des lexèmes
Il peut utiliser des variables globales :

Nom	fonction
<code>char *yytext</code>	pointeur sur le lexème reconnu
<code>yylen</code>	longueur du lexème
<code>yyval</code>	valeur de l'unité lexicale

Variables globales prédéfinies

Exemple Lex (1)

Exemple (1 d'utilisation de Lex)

```
% {  
    int yylineno;  
}%  
  
%%  
  
^(.* )\n    printf ("%4d\t%s", ++yylineno, yytext);  
  
%%  
  
int main(int argc, char *argv[]) {  
    yyin = fopen(argv[1], "r");  
    yylex();  
    fclose(yyin);  
}
```

Remarque :

Dans cet exemple, le scanner (`yylex()`) s'exécute jusqu'à la fin de la lecture complète du fichier

Exemple Lex (2)

Exemple (2 d'utilisation de Lex)

```
digit      [0-9]
letter     [A-Za-z]
%{
    int count;
%}

%%
/* match identifier */
{letter}({letter}{digit})*    count++;

%%
int main(void)  {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

Exemple Lex (3)

Exemple (3 d'utilisation de Lex)

```
%{  
    int nchar, nword, nline;  
}%  
  
%%  
  
\n        { nline++; nchar++; }  
[^ \t\n]+ { nword++; nchar += yystrlen; }  
.        { nchar++; }  
  
%%  
  
int main(void) {  
    yylex();  
    printf("%d\t%d\t%d\n", nchar, nword, nline);  
    return 0;  
}
```

Exemple Lex (4)

Exemple (4 : scanner et évaluateur d'expressions simples)

```
/* calculateur avec '+' et '-' */  
/* Thierry Massart - 28/09/2005 */  
%{  
#define NUMBER 1  
int yyval;  
%}  
  
%%  
  
[0-9]+    {yyval = atoi(yytext); return NUMBER;}  
[ \t]      ;          /* ignore les espaces et tabulation */  
\n        return 0; /* permet l'arrêt à l'eol */  
.        return yytext[0];
```

Exemple Lex (4 (suite))

Exemple (4 (suite))

```
%%
int main() {
    int val;
    int tot=0;
    int signe=1;

    val = yylex();
    while(val !=0) {
        if(val=='-')  signe *= -1;
        else if (val != '+') /* nombre */
        {
            tot += signe*yyval;
            signe = 1;
        }
        val=yylex();
    }
    printf("%d\n",tot);
    return 0;
}
```

Chapitre 4 : Les grammaires

- 1 Rôle des grammaires
- 2 Exemples informels de grammaires
- 3 Grammaire : définition formelle
- 4 La hiérarchie de Chomsky

Plan

- 1 Rôle des grammaires
- 2 Exemples informels de grammaires
- 3 Grammaire : définition formelle
- 4 La hiérarchie de Chomsky

Pourquoi utilise-t-on des grammaires ?

Pourquoi utilise-t-on des grammaires ?

- Beaucoup de langages que l'on désire définir/manipuler ne sont pas réguliers
- Les langages "context free" sont utilisés depuis les années 1950 pour définir la syntaxe de langages de programmation
- En particulier la syntaxe BNF (Backus Naur Form) est basée sur la notion de grammaire (context free)
- La plupart des langages formels sont définis grâce aux grammaires (exemple : XML).

Plan

- 1 Rôle des grammaires
- 2 Exemples informels de grammaires
- 3 Grammaire : définition formelle
- 4 La hiérarchie de Chomsky

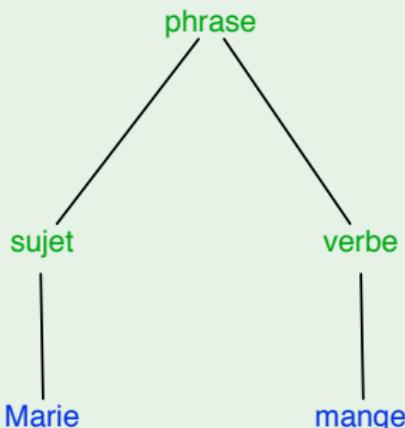
Exemple de grammaire

Exemple (Grammaire d'une phrase)

- phrase = sujet verbe
- sujet = “Jean” | “Marie”
- verbe = “mange” | “parle”

peut donner

- **Jean mange**
- **Jean parle**
- **Marie mange**
- **Marie parle**



Arbre syntaxique de la phrase
Marie mange

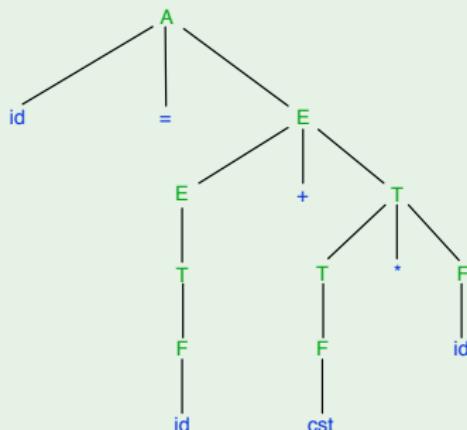
Exemple de grammaire (2)

Exemple (Grammaire d'une expression)

- $A = "id" \ "=\> E$
- $E = T \mid E \ "+" \ T$
- $T = F \mid T \ "*" \ F$
- $F = "id" \mid "cst" \mid "(" \ E \ ")"$

peut donner :

- $id = id$
- $id = id + cst * id$
- ...



Arbre syntaxique de la phrase
 $id = id + cst * id$

Autre exemple

Exemple (Le langage des palindômes)

Soit $L_{pal} = \{w \in \Sigma^* | w = w^R\}$

Par exemple (en faisant abstraction des majuscules/minuscules et des espaces) :

Ressasser

Hannah

Et la marine va, papa, venir a Malte

A Cuba, Anna a bu ça

A Laval elle l'avalà

Aron, au Togo, tua Nora

SAT ORA REPO TENETO PERARO ... TAS

Le dernier exemple est le carré magique (sacré) latin :



Traduction littérale possible : Le semeur subrepticement tient l'oeuvre dans la rotation (des temps)

Quelques interprétations possibles :

- Le laboureur à sa charrue ou en son champ dirige les travaux
- Le semeur (christ) à sa charrue (croix) retient son oeuvre (sacrifice)
- Dieu dirige la création, le travail de l'homme et le produit de la terre

Autre exemple de grammaire

Exemple (Le langage des palindrômes)

Limitons nous à $\Sigma = \{0, 1\}$. La grammaire suit le raisonnement inductif :

- **base** : ϵ , 0 et 1 sont des palindrômes
- **induction** : soit w un palindrome : $0w0$ et $1w1$ sont des palindrômes

- ➊ $P \rightarrow \epsilon$
- ➋ $P \rightarrow 0$
- ➌ $P \rightarrow 1$
- ➍ $P \rightarrow 0P0$
- ➎ $P \rightarrow 1P1$

Autre exemple de grammaire

Terminaux et variables

Dans l'exemple précédent :

- 0 et 1 sont les **terminaux** (symboles de l'alphabet terminal)
- P est une **variable** ou **non terminal** (symbole supplémentaire utilisé pour définir le langage)
- P est aussi le **symbole de départ** (ou l'axiome ou racine)
- 1-5 sont les **règles de production** de la grammaire

Plan

- 1 Rôle des grammaires
- 2 Exemples informels de grammaires
- 3 Grammaire : définition formelle
- 4 La hiérarchie de Chomsky

Grammaire : définition formelle

Définition (Grammaire)

Quadruplet :

$$G = \langle V, T, P, S \rangle$$

où

- V est un ensemble fini de **variables**
- T est un ensemble fini de **terminaux**
- P est un ensemble fini de **règles de production** de la forme $\alpha \rightarrow \beta$ avec

$$\alpha \in (V \cup T)^* V (V \cup T)^* \text{ et } \beta \in (V \cup T)^*$$

- S est une variable ($\in V$) appelée **symbole de départ**

Formellement P est une relation $P : (V \cup T)^* V (V \cup T)^* \times (V \cup T)^*$

Remarque :

Les exemples précédents utilisent des **grammaires context free** c'ât une sous-classe où les règles de production ont la forme $A \rightarrow \beta$ avec $A \in V$

Définition formelle de l'ensemble des palindrômes sur $\{0, 1\}$

Exemple (Le langage des palindrômes)

$$G = \langle \{A\}, \{0, 1\}, P, A \rangle$$

avec $P = \{A \rightarrow \epsilon, A \rightarrow 0, A \rightarrow 1, A \rightarrow 0A0, A \rightarrow 1A1\}$

On note les 5 règles de façon compacte en regroupant celles ayant la même partie gauche (ici toutes les règles !) :

- $A \rightarrow \epsilon \mid 0 \mid 1 \mid 0A0 \mid 1A1$

Définition (A-production)

L'ensemble des règles dont la partie gauche est la variable A est appelée l'ensemble des A-productions

(Relation de) dérivation

Définition (Dérivation)

Ayant une grammaire $G = \langle V, T, P, S \rangle$ Alors

$$\gamma \xrightarrow[G]{} \delta$$

ssi

- $\exists \alpha \rightarrow \beta \in P$
- $\gamma \equiv \gamma_1 \alpha \gamma_2$ pour $\gamma_1, \gamma_2 \in (V \cup T)^*$
- $\delta \equiv \gamma_1 \beta \gamma_2$

Remarques :

- Les grammaires sont donc des **systèmes de réécriture : la dérivation**
 $\gamma \equiv \gamma_1 \alpha \gamma_2 \xrightarrow[G]{} \gamma_1 \beta \gamma_2 \equiv \delta$ est une réécriture de la partie α en β dans le string γ qui dérive en δ
- Quand G est clairement identifiée on écrit plus simplement : $\gamma \Rightarrow \delta$
- $\xrightarrow{*}$ est la fermeture reflexo-transitive de \Rightarrow
- $\alpha \xrightarrow{i} \beta$ est une notation pour une dérivation de longueur i entre α et β
- tout string α dérivable à partir du symbole de départ ($S \xrightarrow{*} \alpha$) est appelé **forme phrasée** ou **protophrase**

Dérivation (suite)

Avec $G = \langle \{E, T, F\}, \{i, c, +, *, (,)\}, P, E \rangle$ et P :

- $E \rightarrow T \mid E + T$
- $T \rightarrow F \mid T * F$
- $F \rightarrow i \mid c \mid (E)$

On a

$$E \xrightarrow{*} i + c * i$$

Plusieurs dérivations sont possibles : exemples :

- ① $E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow i + T$
 $\Rightarrow i + T * F \Rightarrow i + F * F \Rightarrow i + c * F \Rightarrow i + c * i$
- ② $E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * i$
 $\Rightarrow E + F * i \Rightarrow E + c * i \Rightarrow T + c * i \Rightarrow F + c * i \Rightarrow i + c * i$
- ③ $E \Rightarrow E + T \Rightarrow T + T \Rightarrow T + T * F \Rightarrow T + F * F$
 $\Rightarrow T + c * F \Rightarrow F + c * F \Rightarrow F + c * i \Rightarrow i + c * i$
- ④ ...

Langage de G

Définition (Langage d'une grammaire $G = \langle V, T, P, S \rangle$)

$$L(G) = \{w \in T^* \mid S \xrightarrow{*} w\}$$

Définition ($L(A)$)

Pour une grammaire $G = \langle V, T, P, S \rangle$ avec $A \in V$

$$L(A) = \{w \in T^* \mid A \xrightarrow{*} w\}$$

Inférence récursive

Inférence récursive

Un autre type de raisonnement **bottom-up** qui permet de montrer qu'un string appartient à $L(G)$ est de montrer inductivement :

- **base** : ce qui peut être généré en une dérivation
- **induction** : sachant ce qui peut être généré en au plus i dérivations, de voir ce qui peut l'être en une dérivation supplémentaire

Exemple (d'inférence récursive)

Avec les
règles :

- ➊ $E \rightarrow T + E$
- ➋ $E \rightarrow T$
- ➌ $T \rightarrow F.T$
- ➍ $T \rightarrow F$
- ➎ $F \rightarrow F^*$
- ➏ $F \rightarrow 0$
- ➐ $F \rightarrow 1$
- ➑ $F \rightarrow (E)$

On a par ex. les inférences
récursives :

	Chaînes	Prod.	Racine	utilise
(i)	0	6	F	—
(ii)	1	7	F	—
(iii)	0*	5	F	(i)
(iv)	1*	5	F	(ii)
(v)	0	4	T	(i)
(vi)	1	4	T	(ii)
(vii)	0*	4	T	(iii)
(viii)	1*	4	T	(iv)
(ix)	0.0	3	T	(i), (v)
(x)	0.1	3	T	(i), (vi)
(xi)	1.0	3	T	(ii), (v)
(xii)	1.1	3	T	(ii), (vi)
(xiii)	0*.0	3	T	(iii), (v)
(xiv)	0	2	E	(v)
...

Plan

- 1 Rôle des grammaires
- 2 Exemples informels de grammaires
- 3 Grammaire : définition formelle
- 4 La hiérarchie de Chomsky

Noam Chomsky

Noam Chomsky (www.chomsky.info) (born December 7, 1928) is Institute Professor Emeritus of linguistics at the Massachusetts Institute of Technology. Chomsky is credited with the creation of the theory of generative grammar, often considered the most significant contribution to the field of theoretical linguistics of the 20th century. He also helped spark the cognitive revolution in psychology through his review of B. F. Skinner's *Verbal Behavior*, which challenged the behaviorist approach to the study of mind and language dominant in the 1950s. His naturalistic approach to the study of language has also impacted the philosophy of language and mind (see Harman, Fodor).

He is also credited with the establishment of the so-called Chomsky hierarchy, a classification of formal languages in terms of their generative power. Chomsky is also widely known for his political activism, and for his criticism of the foreign policy of the United States and other governments. Chomsky describes himself as a libertarian socialist, a sympathizer of anarcho-syndicalism.



JOHN SOARS

La hiérarchie de Chomsky

Définition (La hiérarchie de Chomsky)

Cette hiérarchie définit 4 classes de grammaires (et de langages).

- *Type 0 : les grammaires non restreintes*

C'est la définition la plus générale donnée plus haut (sans restriction)

- *Type 1 : les grammaires context-sensitive (sensibles au contexte)*

Grammaires où toutes les règles sont de la forme :

- $S \rightarrow \epsilon$ *auquel cas S n'apparaît dans aucun membre de droite de règles*
- $\alpha \rightarrow \beta$ *avec $|\alpha| \leq |\beta|$*

La hiérarchie de Chomsky

Définition (La hiérarchie de Chomsky (suite))

- **Type 2 : les grammaires context-free (hors contexte)**

Grammaires où toutes les règles sont de la forme :

- $A \rightarrow \alpha$ avec $\alpha \in (T \cup V)^*$

- **Type 3 : les grammaires régulières**

Classe de grammaires formées des 2 sous-classes suivantes :

- ① les grammaires linéaires droites, où toutes les règles sont de la forme :

$$A \rightarrow wB \quad \text{avec } A, B \in V \wedge w \in T^*$$
$$A \rightarrow w$$

- ② les grammaires linéaires gauches, où toutes les règles sont de la forme :

$$A \rightarrow Bw \quad \text{avec } A, B \in V \wedge w \in T^*$$
$$A \rightarrow w$$

La hiérarchie de Chomsky

Remarques - propriétés

- Définition **équivalente** des grammaires context-sensitive : G est context-sensitive si ses règles sont de la forme :

$$\alpha A \gamma \rightarrow \alpha \beta \gamma \quad \text{avec } \alpha, \beta, \gamma \in (V \cup T)^* \wedge A \in V$$

- Cette autre définition des grammaires context-sensitive explique le nom de ce type de grammaire (dans le contexte $\alpha - \gamma$, la variable A peut se réécrire en β)
- La grammaire suivante **n'est PAS** régulière :

- ① $S \rightarrow \epsilon$
- ② $S \rightarrow A0$
- ③ $A \rightarrow 1S$

La règle 2 étant linéaire gauche et la règle 3 linéaire droite (mélange des 2 types de règles !)

La hiérarchie de Chomsky

Remarques - propriétés (suite)

- On dit qu'un langage est de type n s'il existe une grammaire de type n qui le définit.
- Nous verrons que $\text{type } 3 \subset \text{type } 2 \subset \text{type } 1 \subset \text{type } 0$
- De ce fait si une grammaire est de type n et $n + 1$, le fait qu'elle soit de type $n + 1$ est plus précis

Chapitre 5 : Les grammaires régulières

- 1 Rappel (définition)
- 2 Équivalence entre grammaires régulières et langages réguliers

Rappel (définition)

Équivalence entre grammaires régulières et langages réguliers

Plan

1 Rappel (définition)

2 Équivalence entre grammaires régulières et langages réguliers

Grammaire régulière (définition)

Définition (Grammaire régulière - type 3 dans la hiérarchie de Chomsky)

Classe de grammaires formées des 2 sous-classes suivantes :

- ① *les grammaires linéaires droites, où toutes les règles sont de la forme*

$$\begin{array}{ll} A \rightarrow wB & \text{avec } A, B \in V \wedge w \in T^* \\ A \rightarrow w & \end{array}$$

- ② *les grammaires linéaires gauches, où toutes les règles sont de la forme*

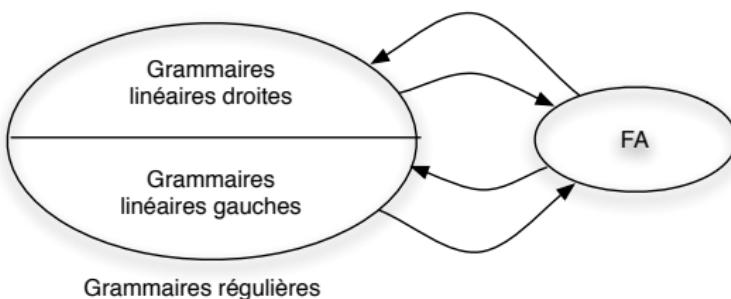
$$\begin{array}{ll} A \rightarrow Bw & \text{avec } A, B \in V \wedge w \in T^* \\ A \rightarrow w & \end{array}$$

Plan

- 1 Rappel (définition)
- 2 Équivalence entre grammaires régulières et langages réguliers

On peut montrer que

- ① Tout langage généré par une grammaire linéaire droite est régulier
- ② Tout langage généré par une grammaire linéaire gauche est régulier
- ③ Tout langage régulier est généré par une grammaire linéaire droite
- ④ Tout langage régulier est généré par une grammaire linéaire gauche



Ce qui implique que la classe des langages générés par une grammaire linéaire droite est la même que celle générée par une grammaire linéaire gauche càd la classe des langages réguliers

Équivalence entre grammaires régulières et langages réguliers (suite)

Théorème (Tout langage généré par une grammaire linéaire droite est régulier)

Principe de la construction / preuve :

Ayant une grammaire linéaire droite $G = \langle V, T, P, S \rangle$, construire un ϵ -NFA M équivalent : $M = \langle Q, T, \delta, [S], \{[\epsilon]\} \rangle$ qui simule les dérivation de G

- $Q = \{[\alpha] \text{ avec } \alpha = S \text{ ou } \alpha = \text{suffixe}(\beta) \text{ où } A \rightarrow \beta \in P \text{ et } \beta \in T^* V \cup T^*\}$
- $\delta :$
 - ① $\delta([A], \epsilon) = \{[\beta] \mid A \rightarrow \beta \in P\}$
 - ② $\delta([a\alpha], a) = \{[\alpha]\} \text{ avec } a \in T \wedge \alpha \in (T^* \cup T^* V)$

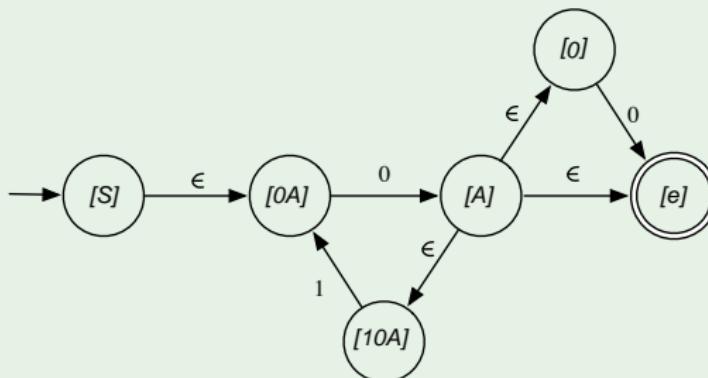
Équivalence entre grammaires régulières et langages réguliers (suite)

Exemple (Une grammaire linéaire droite et le ϵ -NFA équivalent)

La grammaire linéaire droite G génère le langage $0(10)^*(0 + \epsilon)$

$$\begin{aligned}S &\rightarrow 0A \\A &\rightarrow 10A \mid 0 \mid \epsilon\end{aligned}$$

ϵ -NFA équivalent :



Équivalence entre grammaires régulières et langages réguliers (suite)

Théorème (Tout langage généré par une grammaire linéaire gauche est régulier)

Principe de la construction / preuve :

Ayant une grammaire linéaire gauche $G = \langle V, T, P, S \rangle$,

- ➊ construire la grammaire linéaire droite $G' = \langle V, T, P', S \rangle$ avec $L(G') = L(G)^R$.

Pour cela il suffit de définir P' en faisant l'image miroir de toutes les parties droites de règles :

$$A \rightarrow \alpha \in P \iff A \rightarrow \alpha^R \in P'$$

- ➋ construire un ϵ -NFA M' équivalent à G' : $M' = \langle Q, T, \delta', [S], \{[\epsilon]\} \rangle$ qui simule les dérivation de G'
- ➌ construire M avec $L(M) = L(M')^R$: inverser toutes les transitions de M' et permuter état initial et accepteur : $M = \langle Q, T, \delta, [\epsilon], \{[S]\} \rangle$ avec

$$\forall a \in T \cup \{\epsilon\} : \delta(p, a) = q \iff \delta'(q, a) = p$$

Équivalence entre grammaires régulières et langages réguliers (suite)

Exemple (Une grammaire linéaire gauche et le ϵ -NFA équivalent)

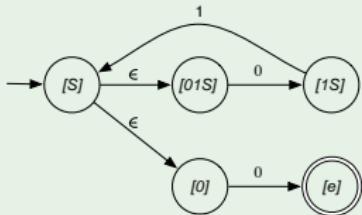
La grammaire linéaire gauche G génère le langage $0(10)^*$

$$S \rightarrow S10 \mid 0$$

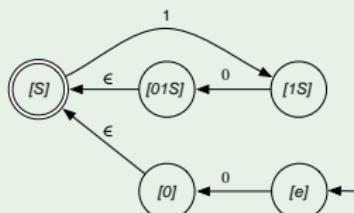
1. Les productions renversées nous donnent :

$$S \rightarrow 01S \mid 0$$

2. Qui correspond au ϵ -NFA suivant :



3. Où on inverse les flèches et permute l'état initial \leftrightarrow accepteur :



Équivalence entre grammaires régulières et langages réguliers (suite)

Théorème (Tout langage régulier est généré par une grammaire linéaire droite)

Principe de la construction / preuve :

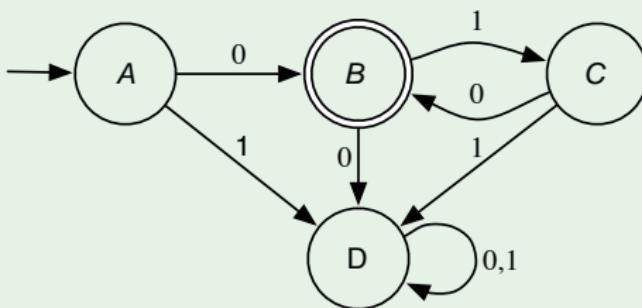
Soit le DFA $M = \langle Q, T, \delta, q_0, F \rangle$

- si $q_0 \notin F$: on définit $G = \langle Q, T, P, q_0 \rangle$ équivalent avec
 - $p \rightarrow aq \in P \iff \delta(p, a) = q$
 - $p \rightarrow a \in P \iff \delta(p, a) \in F$
- si $q_0 \in F$: on fait comme le cas précédent + on rajoute la variable S et
 - $S \rightarrow q_0 \mid \epsilon$

Équivalence entre grammaires régulières et langages réguliers (suite)

Exemple (Un DFA et la grammaire linéaire droite équivalente)

Soit le DFA pour $0(10)^*$:



La grammaire linéaire droite correspondante est :

$$\begin{aligned} A &\rightarrow 0B \mid 1D \mid 0 \\ B &\rightarrow 0D \mid 1C \\ C &\rightarrow 0B \mid 1D \mid 0 \\ D &\rightarrow 0D \mid 1D \end{aligned}$$

Après élimination de D inutile :

$$\begin{aligned} A &\rightarrow 0B \mid 0 \\ B &\rightarrow 1C \\ C &\rightarrow 0B \mid 0 \end{aligned}$$

Équivalence entre grammaires régulières et langages réguliers (suite)

Théorème (Tout langage régulier est généré par une grammaire linéaire gauche)

Principe de la construction / preuve :

A partir du DFA M ,

- ① Prenons le DFA M' pour L^R
- ② construisons la grammaire G' linéaire droite correspondante
- ③ renversons les parties droites des règles de production pour obtenir une grammaire linéaire gauche G avec $L(G) = L(M)$

Chapitre 6 : Les grammaires context free

- 1 Rappels et définitions
- 2 Arbre de dérivation
- 3 Nettoyage et simplification des grammaires context-free
- 4 La forme normale de Chomsky

Rappels et définitions

Arbre de dérivation

Nettoyage et simplification des grammaires context-free

La forme normale de Chomsky

Plan

- 1 Rappels et définitions
- 2 Arbre de dérivation
- 3 Nettoyage et simplification des grammaires context-free
- 4 La forme normale de Chomsky

Grammaire context-free (CFG) (définition)

Définition (Grammaire context-free (hors contexte) - type 2 dans la hiérarchie de Chomsky (CFG))

Grammaire où toutes les règles sont de la forme :

$$A \rightarrow \alpha \quad \text{avec } \alpha \in (T \cup V)^* \wedge A \in V$$

Définition (Langage context free (CFL))

L est un CFL si $L = L(G)$ pour une CFG G

Exemples de grammaire context-free

Exemple (Le langage des palindromes sur l'alphabet $\{0, 1\}$)

$$G = \langle \{P\}, \{0, 1\}, A, P \rangle$$

avec $A = \{P \rightarrow \epsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1\}$

Exemple (Langage d'expressions arithmétiques)

$G = \langle \{E, T, F\}, \{i, c, +, *, (,)\}, P, E \rangle$ avec P :

- $E \rightarrow T \mid E + T$
- $T \rightarrow F \mid T * F$
- $F \rightarrow i \mid c \mid (E)$

Dérivations

Ayant $G = \langle \{E, T, F\}, \{i, c, +, *, (,)\}, P, E \rangle$ et

- $E \rightarrow T \mid E + T$
- $T \rightarrow F \mid T * F$
- $F \rightarrow i \mid c \mid (E)$

On a

$$E \xrightarrow{*} i + c * i$$

Plusieurs dérivations sont possibles : exemples :

- ① $E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow i + T$
 $\Rightarrow i + T * F \Rightarrow i + F * F \Rightarrow i + c * F \Rightarrow i + c * i$
- ② $E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * i$
 $\Rightarrow E + F * i \Rightarrow E + c * i \Rightarrow T + c * i \Rightarrow F + c * i \Rightarrow i + c * i$
- ③ $E \Rightarrow E + T \Rightarrow T + T \Rightarrow T + T * F \Rightarrow T + F * F$
 $\Rightarrow T + c * F \Rightarrow F + c * F \Rightarrow F + c * i \Rightarrow i + c * i$
- ④ ...

Définition (dérivation gauche (resp. droite))

Dérivation de la grammaire G qui réécrit d'abord la variable la plus à gauche (resp. droite) dans la forme phrasée.

- la dérivation 1. (de l'exemple) est gauche (on écrit $S_G \xrightarrow{*} \alpha$)
- la dérivation 2. est droite (on écrit $S \xrightarrow{*_G} \alpha$)

Plan

- 1 Rappels et définitions
- 2 Arbre de dérivation
- 3 Nettoyage et simplification des grammaires context-free
- 4 La forme normale de Chomsky

Arbre de dérivation

Pour une grammaire context-free G , on peut montrer que $w \in L(G)$ grâce à un **arbre de dérivation**.

Exemple (arbre de dérivation)

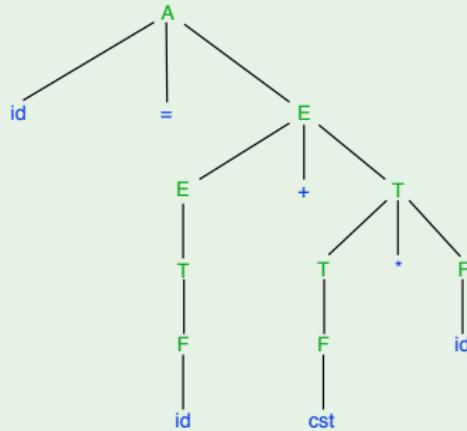
avec l'arbre de dérivation :

La grammaire avec les règles :

- $A = \text{"id"} \ "=\text{"} E$
- $E = T \mid E \ "+" \ T$
- $T = F \mid T \ "\text{**}" \ F$
- $F = \text{"id"} \mid \text{"cst"} \mid \text{("} E \text{")}$

peut donner :

- $\text{id} = \text{id} + \text{cst} * \text{id}$



Construction d'un arbre de dérivation

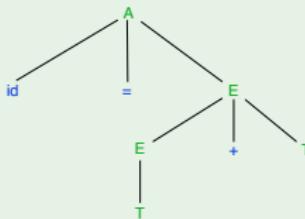
Définition (Arbre de dérivation)

Soit une CFG $G = \langle V, T, P, S \rangle$. Un **arbre de dérivation pour G** est tel que :

- ① chaque **noeud intérieur** est **labellé par une variable**
- ② chaque **feuille** est **labellée par un terminal, une variable ou ϵ** .
Chaque feuille ϵ est le seul fils de son père
- ③ Si un noeud intérieur est labellé **A** et ses fils (de gauche à droite) sont labellés X_1, X_2, \dots, X_k alors $A \rightarrow X_1 X_2 \dots X_k \in P$

Exemple (d'arbre de dérivation)

Pour la grammaire du slide 205



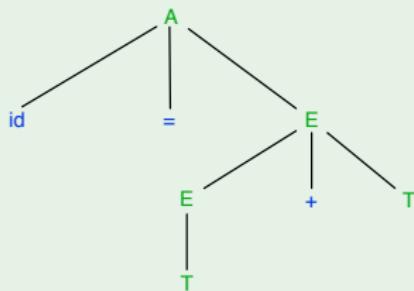
Production (yield) d'un arbre de dérivation

Définition (Production de l'arbre de dérivation)

String formé de la concaténation des labels des feuilles dans l'ordre Gauche - Droit. (Correspond à la forme phrasée dérivée)

Exemple (de production d'un arbre de dérivation)

La production de l'arbre de dérivation suivant :



est

$$id = T + T$$

Arbre de dérivation complet et A-arbre de dérivation

Définition (Arbre de dérivation *complet*)

Soit une CFG $G = \langle V, T, P, S \rangle$. Un *arbre de dérivation complet pour G* est un arbre de dérivation tel que :

- ① La *racine* est labellé par le symbole (la variable) de départ
- ② chaque *feuille* est labellée par un terminal ou ϵ (pas une variable).

Exemple (d'arbre complet)

Voir slide 205

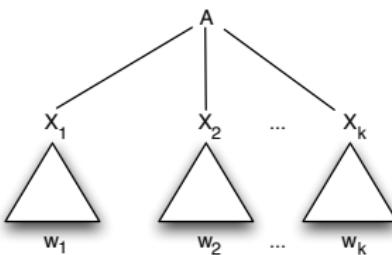
Définition (A-arbre de dérivation)

Arbre de dérivation dont la racine est labellée par la variable A

Pourquoi on appelle ces grammaires **context free**

Grâce aux arbres de dérivation, on voit clairement pourquoi on parle de grammaire context free.

Si $A \Rightarrow X_1 X_2 \dots X_k \stackrel{*}{\Rightarrow} w$ l'arbre correspondant à la forme :



qui montre que chaque X_i est dérivé indépendamment des autres et donc $X_i \stackrel{*}{\Rightarrow} w_i$

Les dérivation quelconques mélangent les dérivation des différents X_i . Par contre, les dérivation gauches ou droites dérivent chaque partie séquentiellement.

Équivalences : dérивations - arbre de dérivation - inférence récursive

Théorème (Équivalences : dérивations - arbre de dérivation - inférence récursive)

Soit une CFG $G = \langle V, T, P, S \rangle$, et $A \in V$. Les 5 propositions suivantes sont équivalentes :

- ① On peut par induction récursive montrer que w est dans le langage de A
- ② $A \xrightarrow{*} w$
- ③ $A \xrightarrow[G]{*} w$
- ④ $A \xrightarrow[G]{*} w$
- ⑤ Il existe un A -arbre de dérivation ayant la production w

Équivalences : dérивations - arbre de dérivation - inférence récursive

Dérivation gauche / droite - arbre de dérivation

Tout arbre de dérivation correspond à

- ① une dérivation gauche
- ② une dérivation droite

et vice versa

Plan

- 1 Rappels et définitions
- 2 Arbre de dérivation
- 3 Nettoyage et simplification des grammaires context-free
- 4 La forme normale de Chomsky

Grammaire context-free ambigu

Grammaire context-free ambiguë

- Pour une CFG G tout string w de $L(G)$ a au moins un arbre de dérivation pour G .
- $w \in L(G)$ peut avoir plusieurs arbres de dérivation pour G : dans ce cas on dira que la grammaire est ambiguë.
- Idéalement, pour permettre le parsing, une grammaire ne doit pas être ambiguë. En effet, l'arbre de dérivation détermine le code généré par le compilateur.
- On essayera donc de modifier la grammaire pour supprimer ses ambiguïtés.
- Il n'existe pas d'algorithme pour supprimer les ambiguïtés d'une grammaire context-free.
- Certains langages context-free sont ambigus de façon inhérente.

Grammaire context-free ambiguë

Exemple (de langage context free ambigu de façon inhérente)

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

Exemple de CFG pour L

$$\begin{aligned} S &\rightarrow AB \mid C \\ A &\rightarrow aAb \mid ab \\ B &\rightarrow cBd \mid cd \\ C &\rightarrow aCd \mid aDd \\ D &\rightarrow bDc \mid bc \end{aligned}$$

Avec G , pour tout $i \geq 0$ $a^i b^i c^i d^i$ a 2 arbres de dérivation. On peut démontrer que toute autre CFG pour L est ambiguë

Suppression de l'ambiguïté

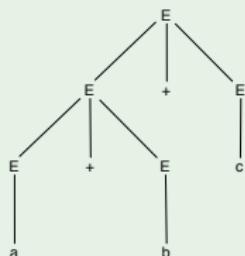
Priorité et associativité

Lorsque le langage définit des strings composés d'instructions et d'opérations, l'arbre syntaxique (qui va déterminer le code produit par le compilateur) doit refléter

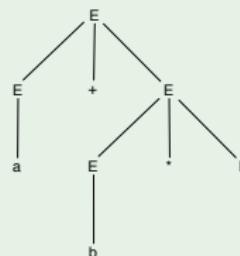
- les priorités et
- associativités

Exemple (d'arbres associés à des expressions)

$$a + b + c$$



$$a + b * c$$



Suppression de l'ambiguïté

Priorité et associativité

Pour respecter l'associativité à gauche, on n'écrit pas

$$E \rightarrow E + E \mid T$$

Mais

$$E \rightarrow E + T \mid T$$

Pour respecter les priorités on définit plusieurs niveaux de variables / règles (le symbole de départ ayant le niveau 0) : les opérateurs les moins prioritaires sont définis à un niveau plus petit (plus proche du symbole de départ) que les plus prioritaires
On n'écrit pas

$$\begin{aligned} E &\rightarrow T + E \mid T * E \mid T \\ T &\rightarrow id \mid (E) \end{aligned}$$

mais en 2 niveaux :

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow F * T \mid F \\ F &\rightarrow id \mid (E) \end{aligned}$$

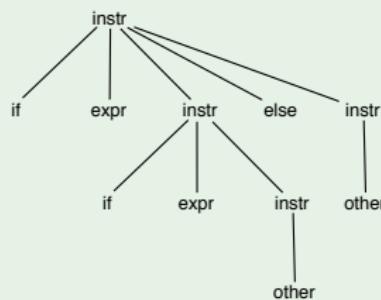
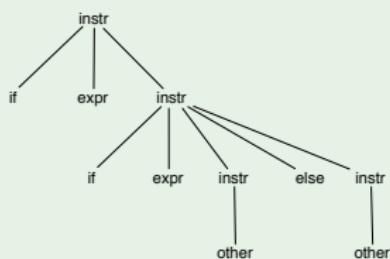
Suppression de l'ambiguïté

Exemple (Associativité de l'instruction if)

La grammaire :

$$instr \rightarrow if\ expr\ instr \mid if\ expr\ instr\ else\ instr \mid other$$

est ambiguë



Dans les langages impératifs habituels c'est l'arbre de gauche qu'il faut générer

Suppression de l'ambiguïté

Exemple (Associativité de l'instruction if)

On peut par exemple transformer la grammaire en :

$$\begin{aligned} instr &\rightarrow open \mid close \\ close &\rightarrow if\ expr\ close\ else\ close \mid other \\ open &\rightarrow if\ expr\ instr \\ open &\rightarrow if\ expr\ close\ else\ open \end{aligned}$$

Suppression des symboles inutiles

Définition (Symboles utiles / inutiles)

Pour une grammaire $G = \langle V, T, P, S \rangle$,

- un symbole X est **utile** s'il existe une dérivation

$$S \xrightarrow[G]{*} \alpha X \beta \xrightarrow[G]{*} w$$

pour des strings α, β et un string de terminaux w

Sinon X est **inutile**.

- un symbole X **produit quelque chose** si $X \xrightarrow[G]{*} w$ pour un string w de terminaux
- un symbole X est **accessible** si $S \xrightarrow[G]{*} \alpha X \beta$ pour des strings α, β

Théorème (Équivalence symboles utiles - (accessibles + produisent quelque chose))

Dans une CFG,

tous les symboles sont accessibles et produisent quelque chose



tous les symboles sont utiles

Suppression des symboles inutiles

Théorème (Suppression des symboles inutiles d'une CFG)

Ayant une CFG $G = \langle V, T, P, S \rangle$.

Si $G' = \langle V', T', P', S \rangle$ est la grammaire obtenue après les 2 étapes suivantes :

- ① Eliminer les symboles ne produisant rien et les règles où ils apparaissent. On obtient $G_I = \langle V_I, T_I, P_I, S \rangle$
- ② Eliminer de G_I les symboles non accessibles et les productions où ils apparaissent.

alors G' est équivalente à G et n'a plus de symboles inutiles.

Preuve :

On va montrer que :

- ① G' n'a plus de symboles inaccessibles ou qui ne produisent rien
- ② $L(G') = L(G)$

Suppression des symboles inutiles

Théorème (Suppression des symboles inutiles d'une CFG (suite))

Preuve de 1. : on va montrer que G' n'a plus de symboles inaccessibles ou qui ne produisent rien

- $\forall X \in T' \cup V'$
- $\exists \alpha, \beta : S \xrightarrow[G']{*} \alpha X \beta$ (tout symbole de G' est accessible)
- Donc $S \xrightarrow[G_I]{*} \alpha X \beta$ (les règles de G' sont dans G_I)
- Donc puisque les symboles de $\alpha X \beta$ n'ont pas été éliminés dans G_I ,
 $\alpha X \beta \xrightarrow[G_I]{*} w$
- Donc X est accessible et produit quelque chose dans G_I
- Mais tous les symboles utilisés dans $\alpha X \beta \xrightarrow[G_I]{*} w$ sont aussi accessibles
dans G_I (puisque $S \xrightarrow[G_I]{*} \alpha X \beta$) et n'ont donc pas été supprimés (ni leurs
règles) dans G'
- Donc X produit aussi quelque chose dans G'
- En résumé, X est accessible et produit quelque chose dans G'

Suppression des symboles inutiles

Théorème (Suppression des symboles inutiles d'une CFG (suite))

Preuve de 2. : on va montrer que $L(G') = L(G)$

- Comme $P' \subseteq P$, on a $L(G') \subseteq L(G)$
- Montrons que $L(G) \subseteq L(G_I) \subseteq L(G')$
- $\forall w \in L(G) : S \xrightarrow[G]{*} w$ et tous les symboles utilisés dans cette dérivation génèrent quelque chose
- Donc les symboles et règles utilisées sont conservés dans G_I et $S \xrightarrow[G_I]{*} w$
- Les symboles utilisés dans $S \xrightarrow[G_I]{*} w$ sont accessibles et donc ils sont conservés avec leurs règles dans $G' : S \xrightarrow[G']{*} w$

Algorithme pour calculer l'ensemble des symboles qui produisent

Algorithme pour calculer l'ensemble $g(G)$ des symboles qui produisent de
 $G = \langle V, T, P, S \rangle$

- **Base :** $g(G) \leftarrow T$
- **Induction :** Si $\alpha \in (g(G))^*$ et $X \rightarrow \alpha \in P$ alors $g(G) \stackrel{\cup}{\leftarrow} \{X\}$

Exemple (de calcul de $g(G)$)

Soit G avec les règles $S \rightarrow AB \mid a, A \rightarrow b$

- ① Au départ $g(G) \leftarrow \{a, b\}$
- ② $S \rightarrow a$ donc $g(G) \stackrel{\cup}{\leftarrow} \{S\}$
- ③ $A \rightarrow b$ donc $g(G) \stackrel{\cup}{\leftarrow} \{A\}$
- ④ Finalement $g(G) = \{S, A, a, b\}$

Théorème (A saturation, $g(G)$ contient l'ensemble des symboles qui produisent quelque chose)

Algorithme pour calculer l'ensemble des symboles accessibles

Algorithme pour calculer l'ensemble $r(G)$ des symboles accessibles de $G = \langle V, T, P, S \rangle$

- **Base :** $r(G) \leftarrow \{S\}$
- **Induction :** Si $A \in r(G)$ et $A \rightarrow \alpha \in P$ alors
 $r(G) \stackrel{\cup}{\leftarrow} \{X \mid \exists \alpha_1, \alpha_2 : \alpha = \alpha_1 X \alpha_2\}$

Exemple (de calcul de $r(G)$)

Soit G avec les règles $S \rightarrow AB \mid a, A \rightarrow b$

- ① Au départ $r(G) \leftarrow \{S\}$
- ② $S \rightarrow AB$ donc $r(G) \stackrel{\cup}{\leftarrow} \{A, B\}$
- ③ $S \rightarrow a$ donc $r(G) \stackrel{\cup}{\leftarrow} \{a\}$
- ④ $A \rightarrow b$ donc $r(G) \stackrel{\cup}{\leftarrow} \{b\}$
- ⑤ Finalement $r(G) = \{S, A, B, a, b\}$

Théorème (À saturation, $r(G)$ contient l'ensemble des symboles accessibles)

Suppression de la récursivité à gauche

Définition (Récursivité à gauche)

Une CFG G est récursive à gauche s'il existe une dérivation $A \xrightarrow[G]{*} A\alpha$

Note :

Nous verrons, dans les chapitres sur le parsing, que la récursivité à gauche est un problème pour *l'analyse syntaxique descendante*. Dans ce cas on la remplace par un autre type de récursivité

Suppression de la récursivité à gauche

Algorithme pour supprimer la récursivité directe à gauche

Ayant

$$A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_r$$

l'ensemble des A -productions directement récursives à gauche et

$$A \rightarrow \beta_1 \mid \cdots \mid \beta_s$$

les autres A -productions.

Toutes ces productions sont remplacées par :

$$A \rightarrow \beta_1 A' \mid \cdots \mid \beta_s A'$$

$$A' \rightarrow \alpha_1 A' \mid \cdots \mid \alpha_r A' \mid \epsilon$$

où A' est une nouvelle variable

Suppression de la récursivité à gauche

Algorithme général pour supprimer la récursivité à gauche

Ayant $V = \{A_1, A_2, \dots, A_n\}$.

Pour $i = 1$ jusque n faire

 Pour $j = 1$ jusque $i - 1$ faire

 Pour chaque production de la forme $A_i \rightarrow A_j \alpha$ faire

 Supprimer $A_i \rightarrow A_j \alpha$ de la grammaire

 Pour chaque production de la forme $A_j \rightarrow \beta$ faire

 Ajouter $A_i \rightarrow \beta \alpha$ à la grammaire

 Fpour

 Fpour

 Fpour

 Éliminer la récursivité gauche directe des A_i -productions

Fpour

Suppression de la récursivité à gauche

Exemple (de suppression de la récursivité à gauche)

Soit la G avec les règles :

$$\begin{array}{lcl} A & \rightarrow & Ab \mid a \mid Cf \\ B & \rightarrow & Ac \mid d \\ C & \rightarrow & Bg \mid Ae \mid Cc \end{array}$$

Traitement de A :

$$\begin{array}{lcl} A & \rightarrow & aA' \mid CfA' \\ A' & \rightarrow & bA' \mid \epsilon \\ B & \rightarrow & Ac \mid d \\ C & \rightarrow & Bg \mid Ae \mid Cc \end{array}$$

Traitement de B :

$$\begin{array}{lcl} A & \rightarrow & aA' \mid CfA' \\ A' & \rightarrow & bA' \mid \epsilon \\ B & \rightarrow & aA'c \mid CfA'c \mid d \\ C & \rightarrow & Bg \mid Ae \mid Cc \end{array}$$

Traitement de C :

$$\begin{array}{lcl} A & \rightarrow & aA' \mid CfA' \\ A' & \rightarrow & bA' \mid \epsilon \\ B & \rightarrow & aA'c \mid CfA'c \mid d \\ C & \rightarrow & \color{red}{aA'cg \mid dg \mid aA'e \mid CfA'cg \mid CfA'e \mid Cc} \end{array}$$

Traitement de C (récursivité directe) :

$$\begin{array}{lcl} A & \rightarrow & aA' \mid CfA' \\ A' & \rightarrow & bA' \mid \epsilon \\ B & \rightarrow & aA'c \mid CfA'c \mid d \\ C & \rightarrow & aA'cgC' \mid dgC' \mid aA'eC' \\ C' & \rightarrow & fA'cgC' \mid fA'eC' \mid cC' \mid \epsilon \end{array}$$

Factorisation à gauche

Définition (Règles factorisables)

Dans une CFG G, plusieurs A-productions sont factorisables à gauche si elles ont la forme

$$A \rightarrow \alpha\beta_1 \mid \cdots \mid \alpha\beta_n$$

avec un préfixe commun $\alpha \neq \epsilon$

Remarque : il peut y avoir d'autres A-productions.

Note :

Pour les analyseurs descendants, on verra qu'il faut factoriser les règles factorisables à gauche

Algorithme de factorisation à gauche

Remplacer :

$$A \rightarrow \alpha\beta_1 \mid \cdots \mid \alpha\beta_n$$

par

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \cdots \mid \beta_n \end{aligned}$$

Suppression des productions vides

Définition (Production vide)

Un production vide est une production :

$$A \rightarrow \epsilon$$

Note :

Il existe un algorithme pour supprimer les productions vides.
Ceci n'est pas utile pour la construction des analyseurs syntaxiques.
Nous ne verrons donc pas cet algorithme.

Suppression des productions unitaires

Définition (Production unitaire)

Un production unitaire est une production :

$$A \rightarrow B$$

avec $B \in V$

Note :

Les productions unitaires sont vues comme des règles qui ne font pas “progresser” la dérivation : il vaut donc mieux les éliminer

Algorithme pour éliminer les productions unitaires

Pour tout $A \xrightarrow{*} B$ en n'utilisant que des productions unitaires, et $B \rightarrow \alpha$ production non unitaire

Ajouter :

$$A \rightarrow \alpha$$

A la fin, supprimer toutes les productions unitaires.

Plan

- 1 Rappels et définitions
- 2 Arbre de dérivation
- 3 Nettoyage et simplification des grammaires context-free
- 4 La forme normale de Chomsky

Grammaire en forme normale de Chomsky

Théorème (Grammaire en forme normale de Chomsky)

Tout langage context free ne contenant pas ϵ peut être défini par une grammaire en forme normale de Chomsky c'est-à-dire une CFG où les règles de production ont les formes possibles

$$\begin{aligned}A &\rightarrow AB \\A &\rightarrow a\end{aligned}$$

Théorème (Grammaire en forme normale de Chomsky)

Tout langage context free contenant ϵ peut être défini par une grammaire $G' = \langle V \cup \{S'\}, T, P', S' \rangle$ où

$$P' = P \cup \{S' \rightarrow S \mid \epsilon\}$$

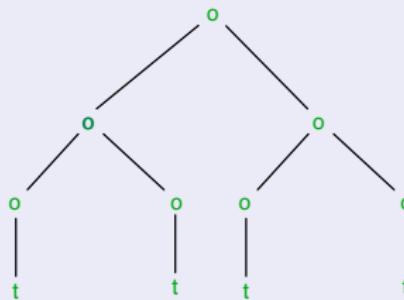
et $G = \langle V, T, P, S \rangle$ est en forme normale de Chomsky.

Grammaire en forme normale de Chomsky

Propriété des arbres de dérivation des CFG en forme normale de Chomsky

Soit G une CFG en forme normale de Chomsky, et $w \in L(G)$, alors l'arbre (ou les arbres) de dérivation de w dans G est tel que

- la hauteur de cet arbre $\geq \log_2(|w|) + 1$
- pour un arbre de hauteur k , “génère” un string de taille $\leq 2^{(k-1)}$



exemple : arbre de taille 3 qui génère un string de longueur 4

Note :

On utilisera cette propriété pour démontrer le lemme de pompage des langages context-free

Chapitre 7 : Les automates à pile et les propriétés des langages context-free

- 1 Les automates à pile (PDA)
- 2 Équivalence entre PDA et CFG
- 3 Propriétés des langages context free

Plan

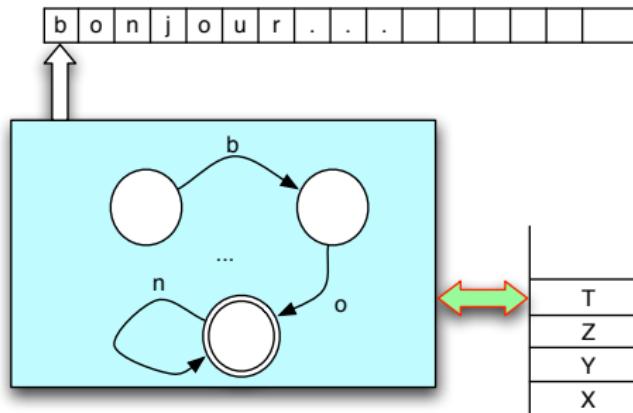
- 1 Les automates à pile (PDA)
- 2 Équivalence entre PDA et CFG
- 3 Propriétés des langages context free

Présentation informelle

Un automate à pile (PDA) est essentiellement un ϵ -NFA avec une pile (stack)

Lors d'une transition, le PDA

- ➊ Consomme un symbole d'input (ou non si c'est une ϵ -transition)
- ➋ Change d'état de contrôle
- ➌ Remplace le symbole T en sommet de la pile par un string (qui peut, en particulier, être ϵ (pop), "T" (pas de changement), "AT" (push un A))



Exemple

Exemple (PDA pour $L_{wwr} = \{ww^R \mid w \in \{0, 1\}^*\}$)

correspondant à la “grammaire” $P \rightarrow 0P0 \mid 1P1 \mid \epsilon$.

On peut construire un PDA équivalent à 3 états qui fonctionne comme suit :

En 0 Il peut supposer (guess) lire w : **push** le symbole sur la pile

En 0 Il peut supposer être au milieu de l'input : va dans l'état 1

En 1 Il compare ce qui est lu et ce qui est sur la pile : s'ils sont identiques, la comparaison est correcte, il pop le sommet de la pile et continue (sinon bloque)

En 1 S'il retombe sur le symbole initial de la pile, va dans l'état 2 (accepteur).

0 , $Z_0 / 0 Z_0$

1 , $Z_0 / 1 Z_0$

0 , $0 / 0 0$

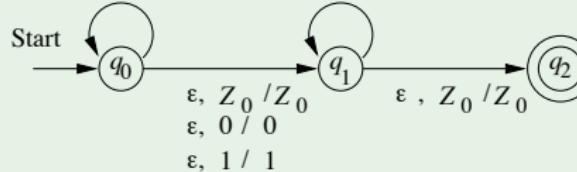
0 , $1 / 0 1$

1 , $0 / 1 0$

1 , $1 / 1 1$

0 , $0 / \epsilon$

1 , $1 / \epsilon$



PDA : définition formelle

Définition

Un PDA est un 7-tuple :

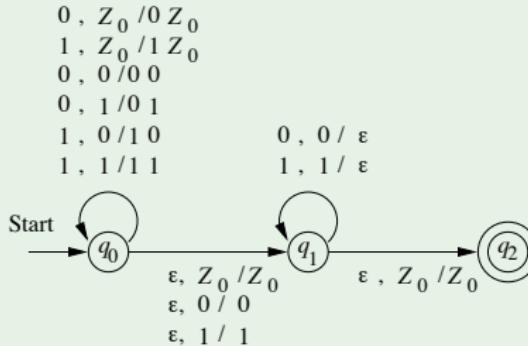
$$P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$$

où

- Q est un ensemble fini d'états
- Σ est l'alphabet d'entrée
- Γ est l'alphabet de la pile
- $\delta : Q \times (\Sigma \cup \{\epsilon\} \times \Gamma) \rightarrow 2^{Q \times \Gamma^*}$ est la fonction de transition
- q_0 est l'état initial
- Z_0 est le symbole initial de la pile
- $F \subseteq Q$ est l'ensemble des états accepteurs

Exemple de PDA

Exemple (de définition formelle du PDA suivant :)



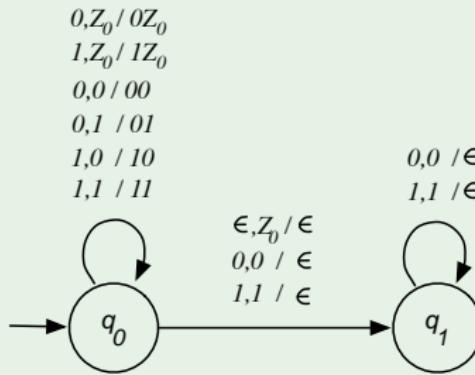
$$P = \langle \{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\} \rangle$$

δ	$(0, Z_0)$	$(1, Z_0)$	$(0, 0)$	$(0, 1)$	$(1, 0)$	$(1, 1)$
$\rightarrow q_0$	$\{(q_0, 0Z_0)\}$	$\{(q_0, 1Z_0)\}$	$\{(q_0, 00)\}$	$\{(q_0, 01)\}$	$\{(q_0, 10)\}$	$\{(q_0, 11)\}$
q_1	\emptyset	\emptyset	$\{(q_1, \epsilon)\}$	\emptyset	\emptyset	$\{(q_1, \epsilon)\}$
$*q_2$	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset

δ	(ϵ, Z_0)	$(\epsilon, 0)$	$(\epsilon, 1)$
$\rightarrow q_0$	$\{(q_1, Z_0)\}$	$\{(q_1, 0)\}$	$\{(q_1, 1)\}$
q_1	$\{(q_2, Z_0)\}$	\emptyset	\emptyset
$*q_2$	\emptyset	\emptyset	\emptyset

Autre exemple de PDA

Exemple (de définition formelle du PDA P' suivant :)



$$P = \langle \{q_0, q_1\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \emptyset \rangle$$

δ	$(0, Z_0)$	$(1, Z_0)$	$(0, 0)$	$(0, 1)$
$\rightarrow q_0$	$\{(q_0, 0Z_0)\}$	$\{(q_0, 1Z_0)\}$	$\{(q_0, 00), (q_1, \epsilon)\}$	$\{(q_0, 01)\}$
q_1	\emptyset	\emptyset	$\{(q_1, \epsilon)\}$	\emptyset

δ	$(1, 0)$	$(1, 1)$	(ϵ, Z_0)
$\rightarrow q_0$	$\{(q_0, 10)\}$	$\{(q_0, 11), (q_1, \epsilon)\}$	$\{(q_1, \epsilon)\}$
q_1	\emptyset	$\{(q_1, \epsilon)\}$	$\{(q_1, \epsilon)\}$

Critère d'acceptation

Un string w est accepté :

- **Par pile vide** : le string est totalement lu et la pile est vide.
- **Par état final** : le string est totalement lu et le PDA est dans un état accepteur.

Remarques :

- Pour un PDA P , **2 langages (a priori différents)** sont définis :
 - $N(P)$ (acceptation par pile vide) et
 - $L(P)$ (acceptation par état final)
- $N(P)$ n'utilise pas F et n'est donc pas modifié si l'on définit $F = \emptyset$

Configuration et langage accepté

Définition (Configuration d'un PDA $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$)

Triple $\langle q, w, \gamma \rangle \in Q \times \Sigma^* \times \Gamma^*$

- Configuration initiale : $\langle q_0, w, Z_0 \rangle$ où w est le string à accepter
- Configuration finale avec acceptation par pile vide : $\langle q, \epsilon, \epsilon \rangle$ (q quelconque)
- Configuration finale avec acceptation par état final : $\langle q, \epsilon, \gamma \rangle$ avec $q \in F$ ($\gamma \in \Gamma^*$ quelconque)

Définition (Changement de configuration)

$$\langle q, aw, X\beta \rangle \underset{P}{\vdash} \langle q', w, \alpha\beta \rangle$$

ssi

$$\langle q', \alpha \rangle \in \delta(q, a, X)$$

Langages de P : $L(P)$ et $N(P)$

Définition ($L(P)$ et $N(P)$)

$$L(P) = \{w \mid w \in \Sigma^* \wedge \exists q \in F, \gamma \in \Gamma^* : \langle q_0, w, Z_0 \rangle \xrightarrow[P]{*} \langle q, \epsilon, \gamma \rangle\}$$

$$N(P) = \{w \mid w \in \Sigma^* \wedge \exists q \in Q : \langle q_0, w, Z_0 \rangle \xrightarrow[P]{*} \langle q, \epsilon, \epsilon \rangle\}$$

où

$\xrightarrow[P]{*}$ est la fermeture réflexo-transitive de $\xrightarrow[P]$

Exemple de séquences d'exécution

Exemple (PDA P avec $L(P) = L_{wwr} = \{ww^R \mid w \in \{0, 1\}^*\}$ et exécutions possibles pour le string 1111)

0 , $Z_0 / 0 Z_0$

1 , $Z_0 / 1 Z_0$

0 , $0 / 0 0$

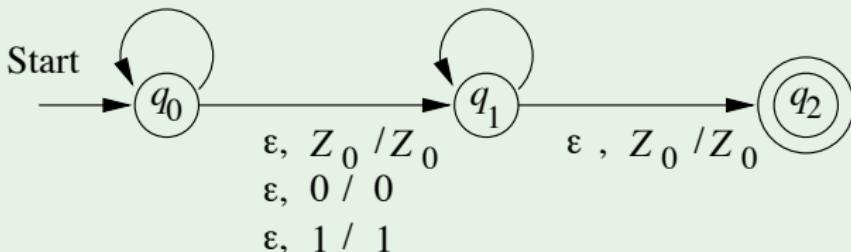
0 , $1 / 0 1$

1 , $0 / 1 0$

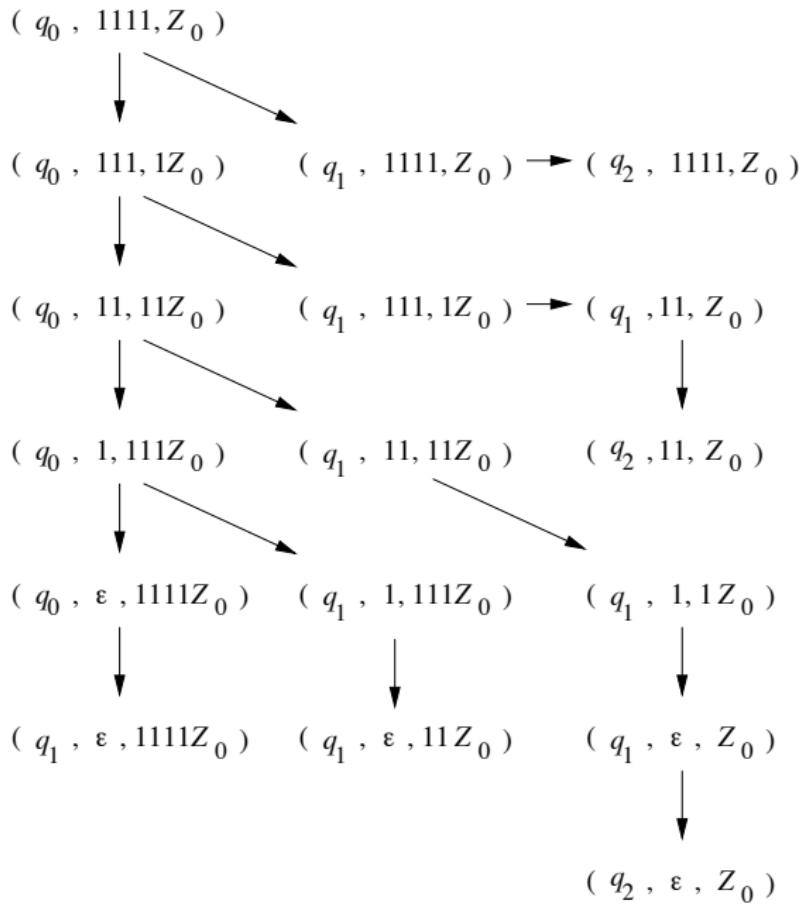
0 , $0 / \epsilon$

1 , $1 / 1 1$

1 , $1 / \epsilon$



séquences (slide suivant)



Exemple de langages acceptés par un PDA

Exemple (Le PDA P du slide 240)

$$L(P) = \{ww^R \mid w \in \{0,1\}^*\} \quad N(P) = \emptyset$$

Exemple (PDA P' du slide 241)

$$L(P') = \emptyset \quad N(P') = \{ww^R \mid w \in \{0,1\}^*\}$$

PDA déterministe (DPDA)

Définition (PDA déterministe (DPDA))

Un PDA $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ est **déterministe** ssi

- ① $\delta(q, a, X)$ est toujours soit vide soit un singleton (pour $a \in \Sigma \cup \{\epsilon\}$)
- ② Si $\delta(q, a, X)$ est non vide alors $\delta(q, \epsilon, X)$ est vide

Exemple (PDA P déterministe avec $L(P) = \{wcw^R \mid w \in \{0, 1\}^*\}$)

0 , $Z_0 / 0 Z_0$

1 , $Z_0 / 1 Z_0$

0 , $0 / 0 0$

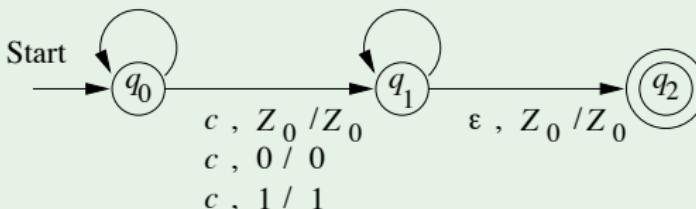
0 , $1 / 0 1$

1 , $0 / 1 0$

0 , $0 / \epsilon$

1 , $1 / 1 1$

1 , $1 / \epsilon$



PDA déterministe

Théorème (La classe des langages définis par un PDA déterministe est strictement incluse dans la classe des langages définis par un PDA (général))

Preuve :

On peut montrer que le langage L_{wwr} défini au slide 240 ne peut être défini par un PDA déterministe

Plan

- 1 Les automates à pile (PDA)
- 2 Équivalence entre PDA et CFG
- 3 Propriétés des langages context free

Équivalence PDA - CFG

On va montrer les inclusions suivantes (chaque flèche montrant une inclusion)



ce qui prouvera que :

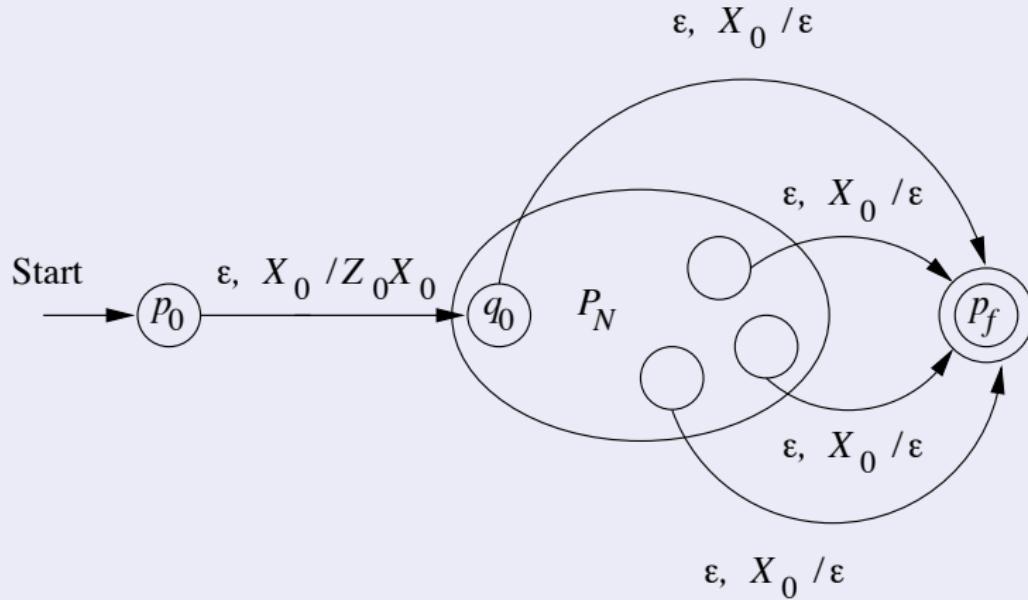
Théorème (Les trois classes de langages suivants sont équivalents)

- *Les langages définis par une CFG (càd les CFL)*
- *Les langages définis par un PDA avec acceptation par pile vide*
- *Les langages définis par un PDA avec acceptation par état final*

Équivalence état final - pile vide

Théorème (Pour tout PDA $P_N = \langle Q, \Sigma, \Gamma, \delta_N, q_0, Z_0, \emptyset \rangle$ on peut définir un PDA P_F avec $L(P_F) = N(P_N)$)

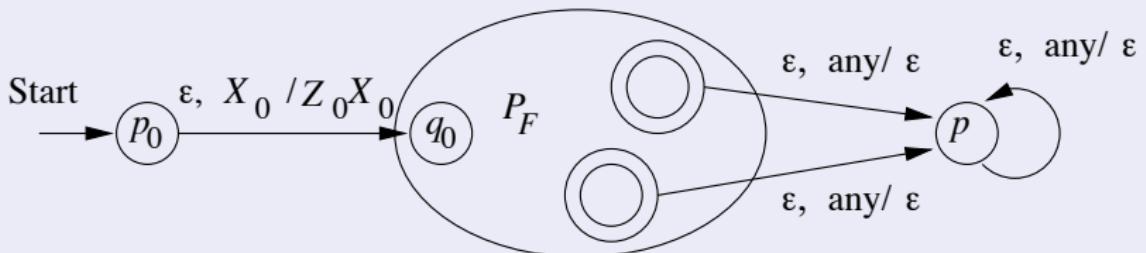
construction :



Équivalence état final - pile vide (2)

Théorème (Pour tout PDA $P_F = \langle Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F \rangle$ on peut définir un PDA P_N avec $N(P_N) = L(P_F)$)

construction :



Équivalence entre PDA et CFG

Théorème (Pour toute CFG G on peut définir un PDA M avec $L(G) = N(M)$)

Principe de la preuve : Ayant $G = \langle V, T, P, S \rangle$ une CFG, on construit un PDA M à un seul état qui simule les dérivation gauches de G

$$\begin{aligned} P &= \langle \{q\}, T, V \cup T, \delta, q, S, \emptyset \rangle \text{ avec} \\ \forall A \rightarrow X_1 X_2 \dots X_k \in P : &\langle q, X_1 X_2 \dots X_k \rangle \in \delta(q, \epsilon, A) \\ \forall a \in T : &\delta(q, a, a) = \{\langle q, \epsilon \rangle\} \end{aligned}$$

- Au départ le symbole de départ S est sur la pile
- Toute variable A au sommet de la pile avec $A \rightarrow X_1 X_2 \dots X_k \in P$ peut être remplacée par sa partie droite $X_1 X_2 \dots X_k$ avec X_1 au sommet de la pile
- Tout terminal au sommet de la pile qui est égal au prochain symbole d'input est **matché** avec l'input (on lit l'input et pop le symbole)
- À la fin, si la pile est vide, le string est accepté

Équivalence entre PDA et CFG

Théorème (Pour tout PDA P on peut définir une CFG G avec $L(G) = N(P)$)

Plan

- 1 Les automates à pile (PDA)
- 2 Équivalence entre PDA et CFG
- 3 Propriétés des langages context free

Questions qu'on peut se poser sur des langages L , L_1 , L_2

- L est-il context-free ?
- Pour quels opérateurs les langages context-free sont-ils fermés ?
- $w \in L$?
- L est-il vide, fini, infini ?
- $L_1 \subseteq L_2$, $L_1 = L_2$?

L est-il context-free ?

Pour montrer que L est context-free

Il faut :

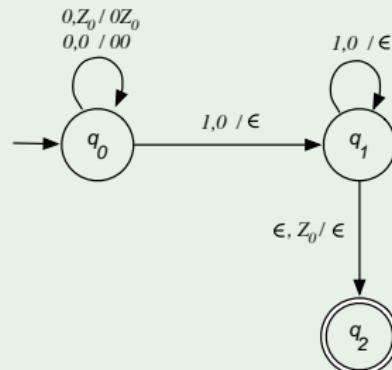
- trouver une CFG (ou un PDA) G et
- montrer que $L = L(G)$ (ou $L = N(G)$ si G est un PDA avec acceptation par pile vide) càd
 - que $L \subseteq L(M)$ (tout string de L est accepté par M)
 - que $L(M) \subseteq L$ (tout string accepté par M est dans L)

L est-il context-free ?

Exemple

Pour montrer que L est context-free $L = \{0^i 1^i \mid i \in \mathbb{N}\}$

On définit par exemple le PDA P et démontre (par induction) que $L = L(P)$



L est-il context-free ?

Pour montrer que L n'est pas context-free

Parfois on peut le montrer grâce au **pumping lemma (lemme de pompage)** des langages context-free

Théorème (Pumping lemma des langages context-free)

*Si L est context-free
alors*

$\exists n \forall z (z \in L \wedge |z| \geq n) \Rightarrow \exists u, v, w, x, y$

- ① $z = uvwxy \wedge$
- ② $|vwx| \leq n \wedge$
- ③ $vx \neq \epsilon \wedge$
- ④ $\forall i \geq 0 : uv^iwx^iy \in L$

Théorème (Pumping lemma des langages context-free)

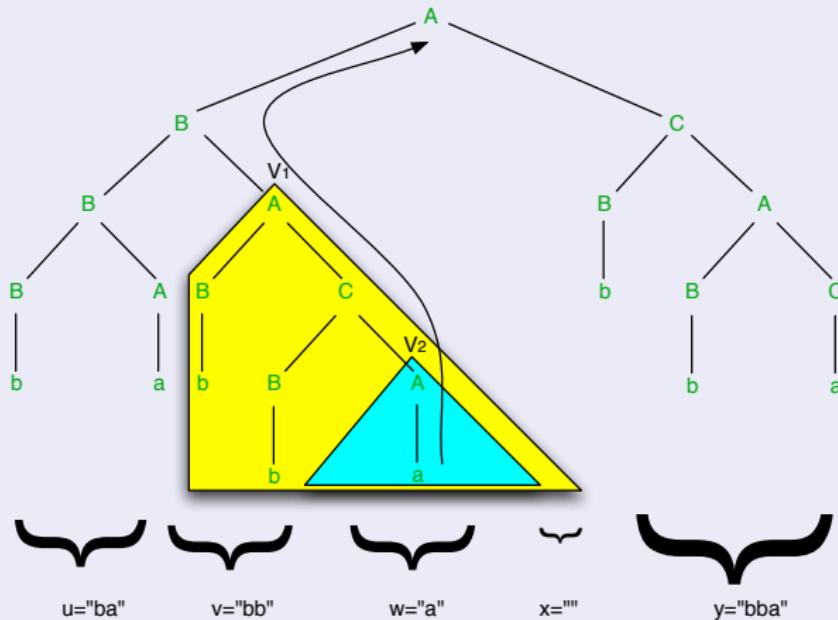
Preuve :

- Soit L un CFL
- L (ou $L \setminus \{\epsilon\}$) est reconnu par une CFG en forme normale de Chomsky G (càd avec les règles $A \rightarrow BC$ ou $A \rightarrow a$ ($a \in T \wedge A, B, C \in V$)).
- Soit k le nombre de variables de G
- Prenons $n = 2^k$ et $|z| \geq n$
- Prenons *un des plus longs chemins* de l'arbre de dérivation de z : plusieurs noeuds du chemin ont le même label (une variable)
- Dénotons v_1 et v_2 2 sommets sur ce chemin labellés par la même variable avec v_1 ancêtre de v_2 et on choisi v_1 "le plus bas possible"^a
- Découpons z comme indiqué sur la figure

^ala technique pour trouver v_1 et v_2 consiste à partir du noeud feuille du chemin choisi et remonter sur ce chemin jusqu'au moment où l'on retombe sur un sommet déjà rencontré (on les labelle v_1 et v_2 (v_2 étant sous v_1))

Preuve du pumping lemma pour les langages context-free

Théorème (Pumping lemma des langages context-free)



et donc $z = uvwxy$

Preuve du pumping lemma pour les langages context-free

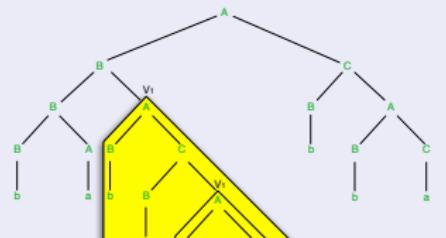
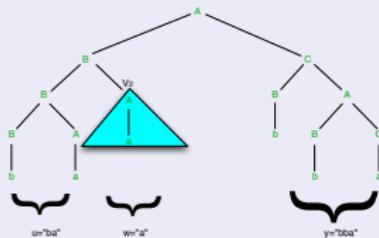
Théorème (Pumping lemma des langages context-free (suite))

Preuve (suite) :

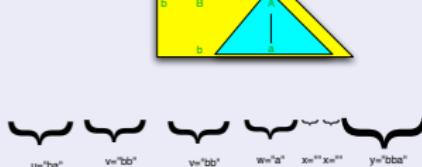
- La hauteur maximale de l'arbre de racine v_1 est $k + 1$ et génère un string vwx avec $|vwx| \leq 2^k = n$
- De plus comme G est en forme normale de Chomsky, v_2 est soit dans le fils droit, soit dans le fils gauche de l'arbre de racine v_1
- De toute façon, soit $v \neq \epsilon$ soit $x \neq \epsilon$
- On peut enfin voir que $uv^iwx^iy \in L(G)$
 - Si on remplace le sous arbre de racine v_1 par le sous arbre de racine v_2 , on obtient $uw y \in L$
 - On peut plutôt remplacer le sous arbre de racine v_2 par une nouvelle instance de sous arbre de racine v_1 , on obtient $uv^2wx^2y \in L$
 - On peut répéter l'opération précédente avec l'arbre obtenu et on obtient que $\forall i : uv^iwx^iy \in L$.

Preuve du pumping lemma pour les langages context-free

Théorème (Pumping lemma des langages context-free (suite))



...



Et donc $\forall i : uv^iwx^iy \in L$

Comment ce pumping lemma permet de dire que L n'est pas context-free

Pumping lemma des CFL = condition nécessaire pour que L context-free

L context-free $\Rightarrow L$ satisfait au PL des CFL

L non context-free $\Leftarrow L$ ne satisfait pas au PL des CFL

Négation du pumping lemma

$\equiv \forall n \exists z (z \in L \wedge |z| \geq n) \wedge \forall u, v, w, x, y$

- ① $(z = uvwxy \wedge$
- ② $|vwx| \leq n \wedge$
- ③ $vx \neq \epsilon)$
- ④ $\Rightarrow (\exists i \geq 0 \wedge uv^iwx^iy \notin L)$

Exemple (Montrons que $L = \{a^i b^j c^i \mid i \in \mathbb{N}\}$ n'est pas context-free)

Montrons que L satisfait la négation des conditions du pumping lemma des CFL.

Pour tout n , pour $z = a^n b^n c^n \in L \wedge |z| \geq n$

et $\forall u, v, w, x, y : (z = uvwxy \wedge |vwx| \leq n \wedge vx \neq \epsilon)$

\Rightarrow

$uwy \notin L$

Comment le pumping lemma permet de dire que L n'est pas context-free

Exemple (Montrons que $L = \{a^i b^j c^i d^j | i, j \geq 1\}$ n'est pas context-free)

Montrons que L satisfait la négation des conditions du pumping lemma des CFL.

Pour tout n , pour $z = a^n b^n c^n d^n \in L \wedge |z| \geq n$

et $\forall u, v, w, x, y : (z = uvwxy \wedge |vwx| \leq n \wedge vx \neq \epsilon)$

$\Rightarrow uw \notin L$

Pour quels opérateurs les langages context-free sont-ils fermés ?

Théorème (Si L_1 et L_2 sont context-free alors $L_1 \cup L_2$, $L_1.L_2$ et L_1^* sont context-free)

Preuve :

Soit $G_1 = \langle V_1, T_1, P_1, S_1 \rangle$ et $G_2 = \langle V_2, T_2, P_2, S_2 \rangle$

2 grammaires n'ayant aucune variable en commun et $S \notin V_1 \cup V_2$:

- Union : Pour $G_{\cup} = \langle V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P, S \rangle$ avec

$$P = P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}$$

$$L(G_{\cup}) = L(G_1) \cup L(G_2)$$

- Concaténation : $L_1.L_2$: Pour $G_{concat} = \langle V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P, S \rangle$ avec

$$P = P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}$$

$$L(G_{concat}) = L(G_1).L(G_2)$$

- Fermeture de Kleene : L_1^* : Pour $G_* = \langle V_1 \cup \{S\}, T_1, P, S \rangle$ avec

$$P = P_1 \cup \{S \rightarrow S_1 S \mid \epsilon\}$$

$$L(G_*) = L(G_1)^*$$

Pour quels opérateurs les langages context-free sont-ils fermés ?

Théorème (Si L est context-free alors L^R est context-free)

Preuve :

L est reconnu par une CFG $G = \langle V, T, P, S \rangle$

À partir de G on peut construire une CFG $G^R = \langle V, T, P^R, S \rangle$ avec

$$P^R = \{A \rightarrow \alpha^R \mid A \rightarrow \alpha \in P\}$$

On peut prouver que $L(G^R) = L^R$

Pour quels opérateurs les langages context-free ne sont-ils pas fermés ?

Théorème (Si L et M sont context-free alors $L \cap M$ peut ne pas être context-free)

Preuve : il suffit de donner un contre exemple !

- $L_1 = \{a^\ell b^\ell c^m \mid \ell, m \in \mathbb{N}\}$ est un langage context-free (A démontrer !)
- $L_2 = \{a^\ell b^m c^m \mid \ell, m \in \mathbb{N}\}$ est un langage context-free (A démontrer !)
- Mais on a vu que $L = L_1 \cap L_2$ n'est pas context-free

Pour quels opérateurs les langages context-free sont-ils fermés ?

Théorème (Si L est context-free et R est régulier alors $L \cap R$ est context-free)

Preuve : (directe et constructive)

- Ayant le PDA A_L avec $L(A_L) = L : A_L = \langle Q_L, \Sigma, \Gamma, \delta_L, q_L, Z_0, F_L \rangle$
- et le DFA A_R avec $L(A_R) = R : A_R = \langle Q_R, \Sigma, \delta_R, q_R, F_R \rangle$
- on définit le PDA $A_{L \cap R}$

$$A_{L \cap R} = \langle Q_L \times Q_R, \Sigma, \Gamma, \delta_{L \cap R}, (q_L, q_R), Z_0, F_L \times F_R \rangle$$

avec

$$\delta_{L \cap R}((p, q), a) = \{(p', q') \mid p' \in \delta_L(p, a) \wedge q' = \delta_R(q, a)\} \quad (a \in \Sigma \cup \{\epsilon\})$$

On peut démontrer que $L(A_{L \cap R}) = L \cap R$

Pour quels opérateurs les langages context-free ne sont-ils pas fermés ?

Théorème (Si L est context-free alors \bar{L} peut ne pas être context-free)

Preuve :

Une des lois de De Morgan est :

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Donc comme l'intersection n'est pas fermée pour les langages context-free mais que l'union l'est, le complément n'est pas non plus fermé pour les CFL.

Pour quels opérateurs les langages context-free ne sont-ils pas fermés ?

Théorème (Si L et M sont context-free alors $L \setminus M$ peut ne pas être context-free)

Preuve :

$$\overline{L} = \Sigma^* \setminus L$$

et Σ^* est régulier

$w \in L$, L vide, fini, infini ?

Comment tester que :

- $w \in L$? : voir si une CFG en forme normale de Chomsky génère w
Il existe un algorithme en $\mathcal{O}(n^3)$ où n est la longueur de w
- L vide ? ($L = \emptyset$) : voir si dans G avec $L(G) = L$, le symbole de départ produit quelque chose
- L infini ? : voir si une CFG en forme normale de Chomsky qui définit L est récursive

problèmes indécidables pour les CFL

Les problèmes suivants sur les CFL sont indécidables (il n'existe pas d'algorithme pour les résoudre de façon générale)

- la CFG G est-elle ambiguë ?
- le CFL L est-il ambigu de façon inhérente ?
- l'intersection de 2 CFL est-elle vide ?
- Le CFL $L = \Sigma^*$?
- $L_1 \subseteq L_2$?
- $L_1 = L_2$?
- $\overline{L(G)}$ est-il un CFL ?
- $L(G)$ est-il déterministe ?
- $L(G)$ est-il régulier ?

Chapitre 8 : Le parsing - analyse syntaxique

- 1 Rôles et place de l'analyse syntaxique (parsing)
- 2 Analyseurs descendants (top-down)
- 3 Analyseurs ascendants (bottom up)

Rôles et place de l'analyse syntaxique (parsing)

Analyseurs descendants (top-down)

Analyseurs ascendants (bottom up)

Plan

1 Rôles et place de l'analyse syntaxique (parsing)

2 Analyseurs descendants (top-down)

3 Analyseurs ascendants (bottom up)

Rôle et place du parser

Rôle principal du parser

- Vérifier que la structure de la chaîne de tokens fournie en entrée (typiquement le programme) fait partie du langage (généralement défini par une grammaire context-free)
- Construire l'**arbre syntaxique** correspondant à cette chaîne de tokens
- Jouer le rôle de chef d'orchestre (programme principal) du compilateur (on parle de compilateur orienté syntaxe)

Place du parser

Entre l'analyseur lexical et l'analyseur sémantique :

- Il appelle le scanner pour lui demander des tokens et
- Il appelle l'analyseur sémantique et ensuite le générateur de code pour terminer l'analyse et générer le code correspondant

Token = terminal

Note :

Dans ce qui suit, nous symbolisons chaque token par un terminal de la grammaire.

\$ en fin de string

Dans la grammaire définissant la syntaxe, on rajoutera systématiquement un nouveau symbole de départ S' , un nouveau terminal \$ (symbolisant la fin du string)^a et la règle de production $S' \rightarrow S\$$ où S est l'ancien symbole de départ.

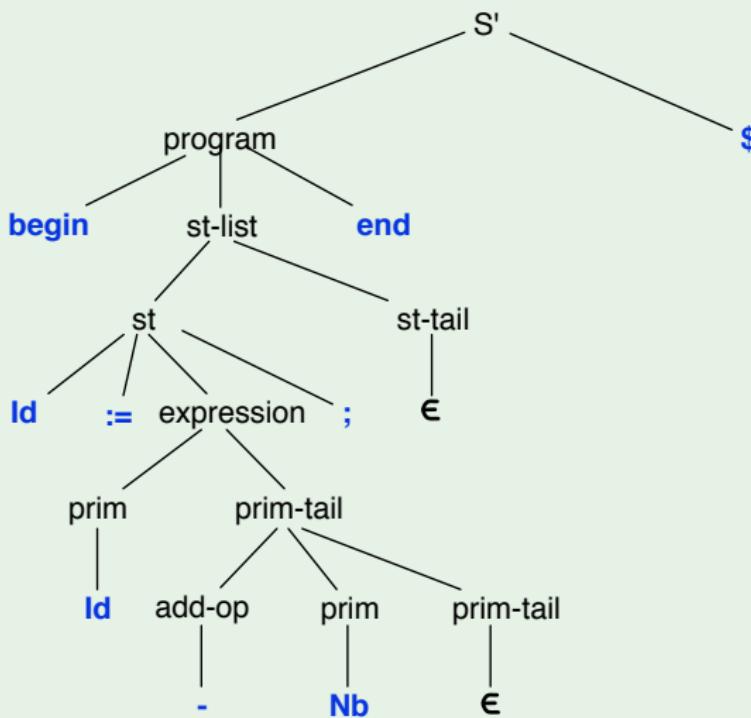
^aOn suppose que ni S' ni \$ ne sont utilisés par ailleurs

Exemple (Grammaire d'un langage très simple)

Règles	Règles de production	
0	S'	\rightarrow program \$
1	program	\rightarrow begin st-list end
2	st-list	\rightarrow st st-tail
3	st-tail	\rightarrow st st-tail
4	st-tail	\rightarrow ϵ
5	st	\rightarrow Id := expression ;
6	st	\rightarrow read (id-list) ;
7	st	\rightarrow write (expr-list) ;
8	id-list	\rightarrow Id id-tail
9	id-tail	\rightarrow , Id id-tail
10	id-tail	\rightarrow ϵ
11	expr-list	\rightarrow expression expr-tail
12	expr-tail	\rightarrow , expression expr-tail
13	expr-tail	\rightarrow ϵ
14	expression	\rightarrow prim prim-tail
15	prim-tail	\rightarrow add-op prim prim-tail
16	prim-tail	\rightarrow ϵ
17	prim	\rightarrow (expression)
18	prim	\rightarrow Id
19	prim	\rightarrow Nb
20 21	add-op	\rightarrow + -

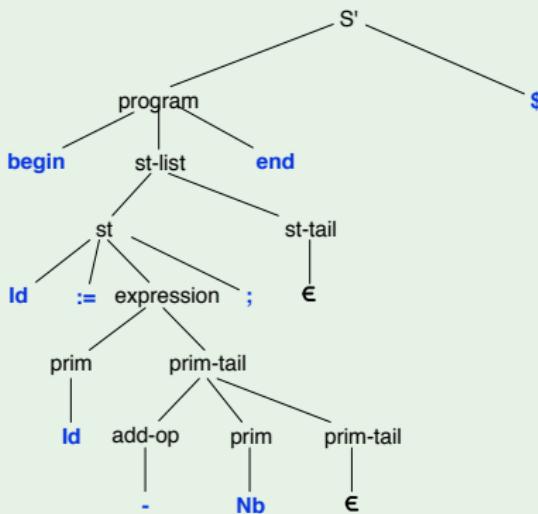
Exemple d'arbre syntaxique

Exemple (Arbre syntaxique correspondant à **begin Id := Id - Nb ; end \$**)



Exemple d'arbre syntaxique et de dérivation gauche / droite

Exemple (Dérivation gauche et droite correspondant à l'arbre syntaxique)



Dérivation gauche $S'_G \xrightarrow{*} \text{begin Id := Id - Nb ; end } \$:$

0 1 2 5 14 18 15 21 19 16 4

Dérivation droite $S'_G \xrightarrow{*} \text{begin Id := Id - Nb ; end } \$:$

0 1 2 4 5 14 15 16 19 21 18

Dérivation gauche

Exemple (Dérivation gauche complète correspondante)

Règle	plus grand préfixe $\in T^*$	suite de la protophrase	
0		S'	\Rightarrow
1	begin	program \$	\Rightarrow
2	begin	st-list end\$	\Rightarrow
5	begin Id :=	st st-tail end\$	\Rightarrow
14	begin Id :=	expression ; st-tail end\$	\Rightarrow
18	begin Id := Id	prim prim-tail ; st-tail end\$	\Rightarrow
15	begin Id := Id	prim-tail ; st-tail end\$	\Rightarrow
21	begin Id := Id -	add-op prim prim-tail ; st-tail end\$	\Rightarrow
19	begin Id := Id - Nb	prim prim-tail ; st-tail end\$	\Rightarrow
16	begin Id := Id - Nb ;	prim-tail ; st-tail end\$	\Rightarrow
4	begin Id := Id - Nb ; end \$	st-tail end\$	\Rightarrow

Dérivation droite

Exemple (Dérivation droite complète correspondante)

Règle	protophrase	
0	S'	⇒
1	program \$	⇒
2	begin st-list end \$	⇒
4	begin st end \$	⇒
5	begin Id := expression ; end \$	⇒
14	begin Id := prim prim-tail ; end \$	⇒
15	begin Id := prim add-op prim prim-tail ; end \$	⇒
16	begin Id := prim add-op prim ; end \$	⇒
19	begin Id := prim add-op Nb ; end \$	⇒
21	begin Id := prim - Nb ; end \$	⇒
18	begin Id := Id - Nb ; end \$	⇒

Structure générale d'un parser

Parser = Algorithme qui reconnaît la structure du programme et construit l'arbre syntaxique correspondant

Comme le langage à reconnaître est context-free, un parser aura le fonctionnement d'un automate à pile (PDA)

Nous verrons 2 grandes classes de parsers :

- Les analyseurs **descendants (top-down)**
- Les analyseurs **ascendants (bottom-up)**

Structure générale d'un parser

Un parser doit, en plus de reconnaître le string, produire un output, qui peut être :

- l'arbre syntaxique correspondant
- des appels aux procédures de l'analyseur sémantique - du générateur de code
- ...

Remarque :

Dans la présentation qui suit, l'output = la séquence des règles de production utilisées dans la dérivation.

Si on sait par ailleurs, que c'est une dérivation **gauche** (resp.**droite**), cette séquence identifie la dérivation et l'arbre correspondant (et permet donc de la construire facilement).

Rôles et place de l'analyse syntaxique (parsing)

Analyseurs descendants (top-down)

Analyseurs ascendents (bottom up)

Plan

1 Rôles et place de l'analyse syntaxique (parsing)

2 Analyseurs descendants (top-down)

3 Analyseurs ascendants (bottom up)

Rappel : construction d'un PDA équivalent à une CFG donnée

Au chapitre précédent nous avons vu :

Construction à partir d'une CFG G d'un PDA M avec $L(G) = N(M)$

Principe de la construction : Ayant $G = \langle V, T, P, S \rangle$ une CFG, on construit un PDA M à un seul état qui simule les dérivations gauches de G

$$P = \langle \{q\}, T, V \cup T, \delta, q, S, \emptyset \rangle \text{ avec}$$

$$\forall A \rightarrow X_1 X_2 \dots X_k \in P : \langle q, X_1 X_2 \dots X_k \rangle \in \delta(q, \epsilon, A)$$

$$\forall a \in T : \delta(q, a, a) = \{\langle q, \epsilon \rangle\}$$

- Au départ le symbole de départ S est sur la pile
- Toute variable A au sommet de la pile avec $A \rightarrow X_1 X_2 \dots X_k \in P$ peut être remplacée par sa partie droite $X_1 X_2 \dots X_k$ avec X_1 au sommet de la pile
- Tout terminal au sommet de la pile qui est égal au prochain symbole d'input est **matché** avec l'input (on lit l'input et pop le symbole)
- A la fin, si la pile est vide, le string est accepté

Dans cette construction, la règle $S' \rightarrow S\$$ n'avait pas été rajoutée

Ébauche de structure d'un parser descendant

Ébauche de structure d'un parser descendant : PDA à un état et avec output

Au départ S' est sur la pile

Le PDA peut effectuer 4 types d'actions :

- **Produce** : la variable A au sommet de la pile est remplacée par la partie droite d'une de ses règles (numérotée i) et le numéro i est écrit sur output
- **Match** : le terminal a au sommet de la pile correspond au terminal en input ; on pop ce terminal et passe au symbole d'input suivant
- **Accept** : Correspond à un Match du terminal $\$$: le terminal au sommet de la pile est $\$$ et correspond au terminal en input ; on termine l'analyse avec succès
- **Erreur** : Si aucun match ni produce n'est possible

Dérivation gauche

Exemple (Dérivation gauche complète correspondante)

Sur la pile	Input restant	Action	Output
S' \dashv	begin Id := Id - Nb ; end \$	P0	ϵ
program \$ \dashv	begin Id := Id - Nb ; end \$	P1	0
begin st-list end\$ \dashv	begin Id := Id - Nb ; end \$	M	0 1
st-list end\$ \dashv	Id := Id - Nb ; end \$	P2	0 1
st st-tail end\$ \dashv	Id := Id - Nb ; end \$	P5	0 1 2
Id := expression ; st-tail end\$ \dashv	Id := Id - Nb ; end \$	M	0 1 2 5
:= expression ; st-tail end\$ \dashv	:= Id - Nb ; end \$	M	0 1 2 5
expression ; st-tail end\$ \dashv	Id - Nb ; end \$	P14	0 1 2 5
prim prim-tail ; st-tail end\$ \dashv	Id - Nb ; end \$	P18	0 1 2 5 14
Id prim-tail ; st-tail end\$ \dashv	Id - Nb ; end \$	M	0 1 2 5 14 18
prim-tail ; st-tail end\$ \dashv	- Nb ; end \$	P15	0 1 2 5 14 18
add-op prim prim-tail ; st-tail end\$ \dashv	- Nb ; end \$	P21	0 1 2 5 14 18 15
- prim prim-tail ; st-tail end\$ \dashv	- Nb ; end \$	M	0 1 2 5 14 18 15 21
prim prim-tail ; st-tail end\$ \dashv	Nb ; end \$	P19	0 1 2 5 14 18 15 21
Nb prim-tail ; st-tail end\$ \dashv	Nb ; end \$	M	0 1 2 5 14 18 15 21 19
prim-tail ; st-tail end\$ \dashv	; end \$	P16	0 1 2 5 14 18 15 21 19
	; end \$	M	0 1 2 5 14 18 15 21 19 16
	st-tail end\$ \dashv		
	end \$ \dashv	P4	0 1 2 5 14 18 15 21 19 16
	end \$ \dashv	M	0 1 2 5 14 18 15 21 19 16 4
	\$ \dashv	A	0 1 2 5 14 18 15 21 19 16 4

où :

P_i : Produce avec la règle *i*

M : Match

A : Accept (correspond à un Match du symbole \$)

E : Erreur (ou blocage qui demande un backtracking) (pas dans l'exemple)

Points à améliorer dans l'ébauche de parser descendant

Critique de l'ébauche de parser descendant

- Tel quel, ce parser est extrêmement inefficace car il doit effectuer du **backtracking** pour explorer toutes les possibilités.
- Dans ce type de parser, un **choix doit avoir lieu au moment de l'action "Produce"**
- Si plusieurs choix sont possibles et qu'aucun critère dans la méthode ne permet de choisir, on parlera de **conflit Produce/Produce**
- Sans guide au moment de ce choix, il faudrait éventuellement explorer tous les Produce possibles : on aurait donc un parser qui prendrait un temps exponentiel (typiquement dans la longueur de l'input) ce qui est inadmissible !
- On présentera des techniques de parsing descendant efficaces au chapitre suivant.

Plan

- 1 Rôles et place de l'analyse syntaxique (parsing)
- 2 Analyseurs descendants (top-down)
- 3 Analyseurs ascendants (bottom up)

Ébauche de structure d'un parser ascendant

PDA à un état et avec output.

Il part du string d'input et construit l'arbre de bas en haut. Pour cela, il opère par :

- ① Mise de symbole d'input sur la pile jusqu'à identification d'une partie droite α (handle) de règle $A \rightarrow \alpha$
- ② "Réduction" : remplacement de la α par A^a

Au départ la pile est vide.

Le PDA peut effectuer 4 types d'actions :

- **Shift** : lecture d'un symbole d'input et push de ce symbole sur la pile
- **Reduce** : le sommet de la pile α correspond au **handle** (la partie droite d'une règle numéro i : $A \rightarrow \alpha$), est remplacé par A sur la pile et le numéro de règle utilisée i est écrit sur output
- **Accept** : Correspond à un Reduce de la règle $S' \rightarrow S\$$ (qui montre que l'input a été complètement lu et analysé) ; on termine l'analyse avec succès
- **Erreur** : Si aucun shift ni reduce n'est possible

^aCorrespond formellement à $|\alpha|$ pops suivis d'un push de A

Ebauche de structure d'un parser ascendant

Remarque :

- On peut remarquer que l'analyse correspond à une **analyse droite à l'envers** : on part du string et remonte dans les dérivation vers le symbole de départ. L'analyse est faite à l'envers étant donné qu'on lit l'input de gauche à droite.
- L'output sera donc construit à l'envers (tout nouvel output se met avant ce qui a déjà été produit) pour obtenir cette dérivation droite.

Dérivation droite

Exemple (Dérivation droite complète correspondante)

Sur la pile	Input restant	Act	Output
⊤	begin Id := Id - Nb ; end \$	S	€
⊤ begin	Id := Id - Nb ; end \$	S	€
⊤ begin Id	:= Id - Nb ; end \$	S	€
⊤ begin Id :=	Id - Nb ; end \$	S	€
⊤ begin Id := Id	- Nb ; end \$	R18	€
⊤ begin Id := prim	- Nb ; end \$	S	18
⊤ begin Id := prim -	Nb ; end \$	S	18
⊤ begin Id := prim -	Nb ; end \$	R21	18
⊤ begin Id := prim add-op	Nb ; end \$	S	21 18
⊤ begin Id := prim add-op Nb	; end \$	R19	21 18
⊤ begin Id := prim add-op prim	; end \$	R16	19 21 18
⊤ begin Id := prim add-op prim prim-tail	; end \$	R15	16 19 21 18
⊤ begin Id := prim prim-tail	; end \$	R14	15 16 19 21 18
⊤ begin Id := expression	; end \$	S	14 15 16 19 21 18
⊤ begin Id := expression ;	end \$	R5	14 15 16 19 21 18
⊤ begin st	end \$	R4	5 14 15 16 19 21 18
⊤ begin st st-tail	end \$	R2	4 5 14 15 16 19 21 18
⊤ begin st-list	end \$	S	2 4 5 14 15 16 19 21 18
⊤ begin st-list end	\$	R1	2 4 5 14 15 16 19 21 18
⊤ program	\$	S	1 2 4 5 14 15 16 19 21 18
⊤ program \$	€	A	1 2 4 5 14 15 16 19 21 18
⊤ S'	€		0 1 2 4 5 14 15 16 19 21 18

Où :

S : Shift

Ri : Reduce avec la règle *i*

A : Accept (correspond à un Reduce avec la règle 0)

E : Erreur (ou blocage qui demande un backtracking)

Points à améliorer dans l'ébauche de parser ascendant

Critique de l'ébauche de parser ascendant

- Tel quel, ce parser est extrêmement inefficace car il doit effectuer du **backtracking** pour explorer toutes les possibilités.
- Dans ce type de parser, un **choix** doit avoir lieu au moment de l'action "Reduce" et "Shift"
- Si plusieurs choix sont possibles et qu'aucun critère dans la méthode ne permet de choisir, on parlera de **conflit Shift/Reduce** ou **Reduce/Reduce**
- Sans guide au moment de ce choix, il faudrait éventuellement explorer tous les Shift et Reduce possibles : on aurait donc un parser qui prendrait un temps exponentiel (typiquement dans la longueur de l'input) ce qui est inadmissible !
- On présentera des techniques de parsing ascendant efficaces dans un chapitre ultérieur.

Chapitre 9 : Les parseurs $LL(k)$

- 1 Principe du parsing descendant
- 2 Analyseurs prédictifs - $First^k$ - $Follow^k$
- 3 CFG LL(k)
- 4 Analyseurs $LL(1)$
- 5 Analyseurs fortement $LL(k)$ ($k > 1$)
- 6 Analyseurs $LL(k)$ ($k > 1$)
- 7 Gestion des erreurs et resynchronisations
- 8 Analyseurs $LL(k)$ récursifs

Plan

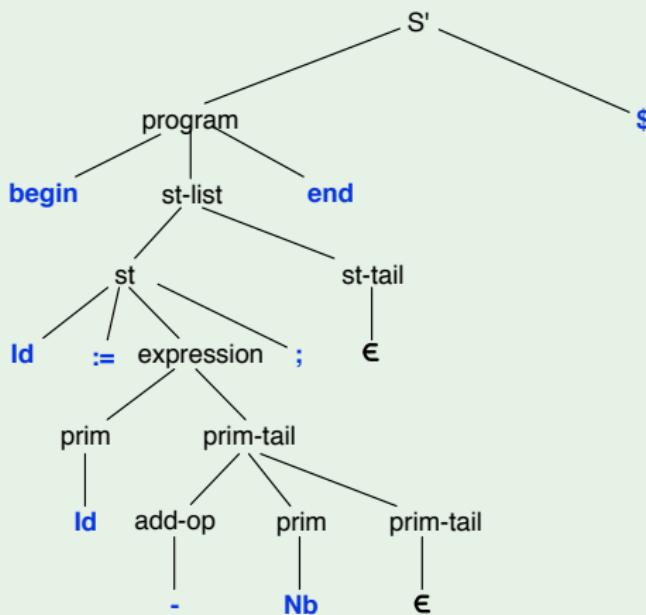
- 1 Principe du parsing descendant
- 2 Analyseurs prédictifs - $First^k$ - $Follow^k$
- 3 CFG LL(k)
- 4 Analyseurs $LL(1)$
- 5 Analyseurs fortement $LL(k)$ ($k > 1$)
- 6 Analyseurs $LL(k)$ ($k > 1$)
- 7 Gestion des erreurs et resynchronisations
- 8 Analyseurs $LL(k)$ récursifs

Exemple (CFG d'un langage très simple)

Règles	Règles de production	
0	$S' \rightarrow$	program \$
1	$program \rightarrow$	begin st-list end
2	$st-list \rightarrow$	st st-tail
3	$st-tail \rightarrow$	st st-tail
4	$st-tail \rightarrow$	ϵ
5	$st \rightarrow$	Id := expression ;
6	$st \rightarrow$	read (id-list) ;
7	$st \rightarrow$	write (expr-list) ;
8	$id-list \rightarrow$	Id id-tail
9	$id-tail \rightarrow$, Id id-tail
10	$id-tail \rightarrow$	ϵ
11	$expr-list \rightarrow$	expression expr-tail
12	$expr-tail \rightarrow$, expression expr-tail
13	$expr-tail \rightarrow$	ϵ
14	$expression \rightarrow$	prim prim-tail
15	$prim-tail \rightarrow$	add-op prim prim-tail
16	$prim-tail \rightarrow$	ϵ
17	$prim \rightarrow$	(expression)
18	$prim \rightarrow$	Id
19	$prim \rightarrow$	Nb
20 21	$add-op \rightarrow$	+ -

Exemple d'arbre syntaxique et de dérivation gauche

Exemple (Dérivation gauche correspondant à l'arbre syntaxique)



Dévolution gauche $S'_G \xrightarrow{*} \text{begin Id } \coloneqq \text{Id } - \text{Nb} ; \text{end } \$:$
0 1 2 5 14 18 15 21 19 16 4

Dérivation gauche

Exemple (Dérivation gauche complète correspondante)

Règle	plus grand préfixe $\in T^*$	suite de la protophrase	
0		S'	\Rightarrow
1	begin	program \$	\Rightarrow
2	begin	st-list end\$	\Rightarrow
5	begin Id :=	st st-tail end\$	\Rightarrow
14	begin Id :=	expression ; st-tail end\$	\Rightarrow
18	begin Id := Id	prim prim-tail ; st-tail end\$	\Rightarrow
15	begin Id := Id	prim-tail ; st-tail end\$	\Rightarrow
21	begin Id := Id -	add-op prim prim-tail ; st-tail end\$	\Rightarrow
19	begin Id := Id - Nb	prim prim-tail ; st-tail end\$	\Rightarrow
16	begin Id := Id - Nb ;	prim-tail ; st-tail end\$	\Rightarrow
4	begin Id := Id - Nb ; end \$	st-tail end\$	\Rightarrow

Ébauche de structure d'un parser descendant

Ébauche de structure d'un parser descendant : PDA à un état et avec output

Au départ S' est sur la pile

Le PDA peut effectuer 4 types d'actions :

- **Produce** : la variable A au sommet de la pile est remplacée par la partie droite d'une de ses règles (numérotée i) et le numéro i est écrit sur output
- **Match** : le terminal a au sommet de la pile correspond au terminal en input ; on pop ce terminal et passe au symbole d'input suivant
- **Accept** : Correspond à un Match du terminal $\$$: le terminal au sommet de la pile est $\$$ et correspond au terminal en input ; on termine l'analyse avec succès
- **Erreur** : Si aucun match ni produce n'est possible

Dérivation gauche

Exemple (Dérivation gauche complète correspondante)

Sur la pile	Input restant	Action	Output
S' \dashv	begin Id := Id - Nb ; end \$	P0	ϵ
program \$ \dashv	begin Id := Id - Nb ; end \$	P1	0
begin st-list end\$ \dashv	begin Id := Id - Nb ; end \$	M	0 1
st-list end\$ \dashv	Id := Id - Nb ; end \$	P2	0 1
st st-tail end\$ \dashv	Id := Id - Nb ; end \$	P5	0 1 2
Id := expression ; st-tail end\$ \dashv	Id := Id - Nb ; end \$	M	0 1 2 5
:= expression ; st-tail end\$ \dashv	:= Id - Nb ; end \$	M	0 1 2 5
expression ; st-tail end\$ \dashv	Id - Nb ; end \$	P14	0 1 2 5
prim prim-tail ; st-tail end\$ \dashv	Id - Nb ; end \$	P18	0 1 2 5 14
Id prim-tail ; st-tail end\$ \dashv	Id - Nb ; end \$	M	0 1 2 5 14 18
prim-tail ; st-tail end\$ \dashv	- Nb ; end \$	P15	0 1 2 5 14 18
add-op prim prim-tail ; st-tail end\$ \dashv	- Nb ; end \$	P21	0 1 2 5 14 18 15
- prim prim-tail ; st-tail end\$ \dashv	- Nb ; end \$	M	0 1 2 5 14 18 15 21
prim prim-tail ; st-tail end\$ \dashv	Nb ; end \$	P19	0 1 2 5 14 18 15 21
Nb prim-tail ; st-tail end\$ \dashv	Nb ; end \$	M	0 1 2 5 14 18 15 21 19
prim-tail ; st-tail end\$ \dashv	; end \$	P16	0 1 2 5 14 18 15 21 19
	; end \$	M	0 1 2 5 14 18 15 21 19 16
	st-tail end\$ \dashv		
	end \$ \dashv	P4	0 1 2 5 14 18 15 21 19 16
	end \$ \dashv	M	0 1 2 5 14 18 15 21 19 16 4
	\$ \dashv	A	0 1 2 5 14 18 15 21 19 16 4

où :

P_i : Produce avec la règle *i*

M : Match

A : Accept (correspond à un Match du symbole \$)

E : Erreur (ou blocage qui demande un backtracking) (pas dans l'exemple)

Points à améliorer dans l'ébauche de parser descendant

Critique de l'ébauche de parser descendant

- Tel quel, ce parser est extrêmement inefficace car il doit effectuer du **backtracking** pour explorer toutes les possibilités.
- Dans ce type de parser, un **choix doit avoir lieu au moment de l'action "Produce"**
- Si plusieurs choix sont possibles et qu'aucun critère dans la méthode ne permet de choisir, on parlera de **conflit Produce/Produce**
- Sans guide au moment de ce choix, il faudrait éventuellement explorer tous les Produce possibles : on aurait donc un parser qui prendrait un temps exponentiel (typiquement dans la longueur de l'input) ce qui est inadmissible !

Plan

- 1 Principe du parsing descendant
- 2 Analyseurs prédictifs - $First^k$ - $Follow^k$
- 3 CFG LL(k)
- 4 Analyseurs $LL(1)$
- 5 Analyseurs fortement $LL(k)$ ($k > 1$)
- 6 Analyseurs $LL(k)$ ($k > 1$)
- 7 Gestion des erreurs et resynchronisations
- 8 Analyseurs $LL(k)$ récursifs

Introduction des analyseurs prédictifs

Motivation

- Lors d'une analyse descendante, les choix que le parser doit effectuer ont lieu lorsque l'action à réaliser est un **Produce** et la variable concernée (au sommet de la pile) a plusieurs règles.
- Dans ce cas, l'input qui reste à analyser (non encore Matché) peut servir de "guide"
- Dans l'exemple, si on doit faire un produce avec la variable **st**, selon que l'input restant est **Id**, **read** ou **write**, il est clair que le parser devra faire un **Produce 5, 6 ou 7**

Analyseur prédictif

Les analyseurs $LL(k)$ sont **prédictifs** et ont k symboles de prévision (k est un nombre naturel) : lorsqu'une variable est au sommet de la pile, le produit effectué dépendra,

- ① de la variable
- ② des (au plus) k premiers symboles à l'entrée

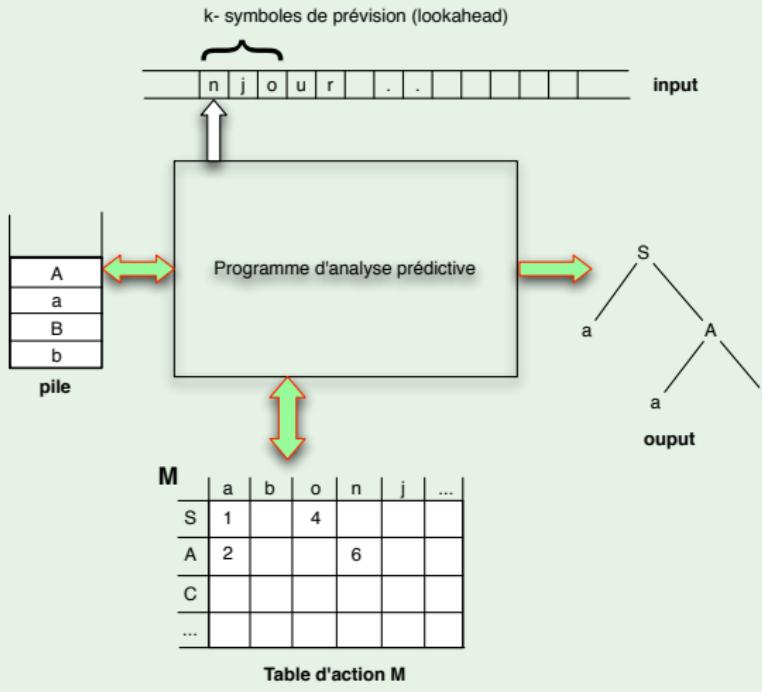
Exemple (Table d'action M)

Un analyseur $LL(k)$ aura une table à 2 dimensions M où $M[A, u]$ détermine le numéro de règle du produit à effectuer quand A est la variable à développer (au sommet de la pile) et u est la prévision.

- Pour un analyseur $LL(1)$ la prévision sera un symbole terminal a .
- Pour un analyseur $LL(k)$ avec k un entier supérieur à 1, on aura une table $M[A, u]$ où u sera un string de taille inférieure ou égale à k (limité au \$ final)

Analyseurs prédictifs $LL(k)$

Exemple (Analyseur prédictif $LL(k)$)



Remarque : Toute combinaison non prévue dans **M** donne une erreur lors de l'analyse

Analyseurs prédictifs $LL(k)$

Algorithme général du fonctionnement d'un analyseur prédictif $LL(k)$ pour $G = \langle V, T, P, S' \rangle$ et les règles de la forme $A \rightarrow \alpha_i$

On suppose que la table M est construite

Parser-LL-k() :=

Initialement : $\text{Push}(S')$

Tant que (pas Error ni Accept)

$X \leftarrow \text{Top}()$

$u \leftarrow \text{Prévision de l'input}$

Si ($X = A \in V$ et $M[A, u] = i$) : Produce(i) ;

Sinon si ($X = a \neq \$ \in T$) et $u = av$ ($v \in T^*$) : Match() ;

Sinon si ($X = u = \$$) : Accept() ;

Sinon : /* rien n'est prévu */ Erreur() ;

FProc

Produce(i) := Pop() ; Push(α_i) ; Fproc

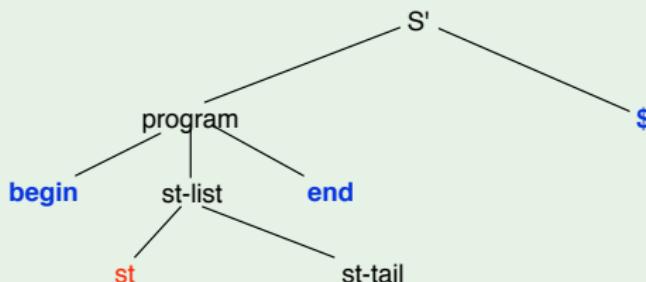
Match() := Pop() ; Avance à l'input suivant ; Fproc

Accept() := Informe du succès de l'analyse ; Fproc

Erreur() := Informe d'une erreur dans l'analyse ; Fproc

Comment peut-on prédire (c'est-à-dire remplir M) ?

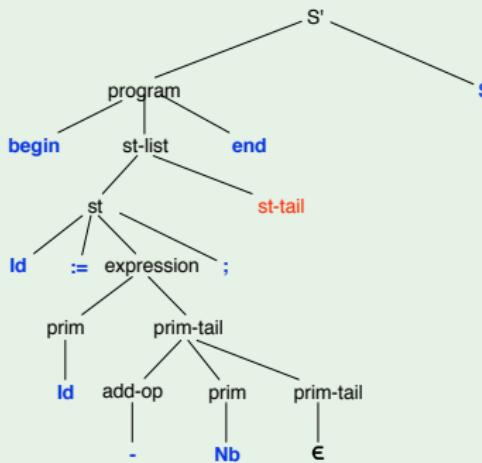
Exemple (1 : une étape du parsing descendant pour $S'_G \xrightarrow{*} \text{begin Id := Id - Nb ; end \$}$)



- Dérivation gauche déjà effectuée $S'_G \xrightarrow{*} \text{begin st st-tail end \$} : 0\ 1\ 2$
- Input restant non encore matché : $\text{Id := Id - Nb ; end \$}$
- ⇒ Il faut faire un Produit de $st \rightarrow \text{Id := expression}$; qui débute par Id et correspond donc à l'input

Comment peut-on prédire (c'est-à-dire remplir M) ?

Exemple (2 : une étape du parsing descendant pour $S' \xrightarrow{*} \text{begin Id := Id - Nb ; end \$}$)



- Dérivation gauche déjà effectuée $S' \xrightarrow{*} \text{begin Id := Id - Nb ; st-tail end} \mathbb{S} : 0 1 2 5 14 18 15 21 19 16$
- Input restant non encore matché : **end \$**
- ⇒ Il faut faire un Produit de $st-tail \rightarrow \epsilon$ car ce qui suit $st-tail$ débute par **end** qui correspond à l'input

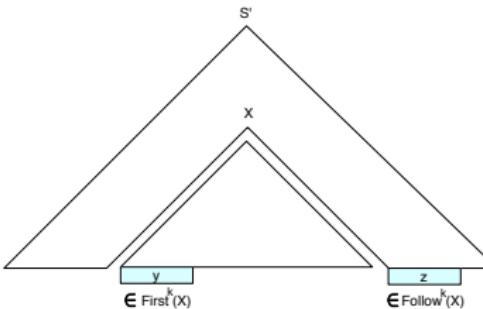
$First^k(X) - Follow^k(X)$

Pour un symbole X , on voudra donc connaître :

- $First^k(X)$: l'ensemble des strings de terminaux de longueur maximale mais limitée à k caractères qui peuvent **débuter** un string généré à partir de X
- $Follow^k(X)$: l'ensemble des strings de terminaux de longueur maximale mais limitée à k caractères qui peuvent **suivre** un string généré à partir de X

Dans la figure suivante on a :

- $y \in First^k(X)$
- $z \in Follow^k(X)$:



Principe du parsing descendant
Analyseurs prédictifs - $First^k$ - $Follow^k$
CFG LL(k)
Analyseurs $LL(1)$
Analyseurs fortement $LL(k)$ ($k > 1$)
Analyseurs $LL(k)$ ($k > 1$)
Gestion des erreurs et resynchronisations
Analyseurs $LL(k)$ récursifs

$First^k$ - $Follow^k$

La construction de la table d'actions M utilise les fonctions $First^k$ et $Follow^k$ définies pour une CFG $G = \langle V, T, P, S' \rangle$ donnée :

Définition ($First^k(\alpha)$). Pour la CFG G , un entier positif k et $\alpha \in (T \cup V)^*$

$First^k(\alpha)$ est l'ensemble des strings de terminaux de longueur maximale mais limitée à k caractères qui peuvent **débuter** un string généré à partir de α

Mathématiquement :

$$First^k(\alpha) = \left\{ w \in T^{\leq k} \mid \exists x \in T^*: \alpha \xrightarrow{*} wx \wedge \begin{array}{l} ((|w| = k) \vee \\ (|w| < k \wedge x = \epsilon)) \end{array} \right\}$$

où : $T^{\leq k} = \bigcup_{i=0}^k T^i$

Principe du parsing descendant
Analyseurs prédictifs - $First^k$ - $Follow^k$
CFG LL(k)
Analyseurs $LL(1)$
Analyseurs fortement $LL(k)$ ($k > 1$)
Analyseurs $LL(k)$ ($k > 1$)
Gestion des erreurs et resynchronisations
Analyseurs $LL(k)$ récursifs

$First^1(\alpha)$ (ou $First(\alpha)$)

Définition ($First^1(\alpha)$ (ou $First(\alpha)$))

$First^1(\alpha)$, dénoté plus simplement $First(\alpha)$, est l'ensemble des symboles terminaux qui peuvent commencer un string généré à partir de α , union ϵ si α peut générer ϵ

Mathématiquement :

$$First(\alpha) = \{ a \in T \mid \exists x \in T^* : \alpha \xrightarrow{*} ax \} \cup \{ \epsilon \mid \alpha \xrightarrow{*} \epsilon \}$$

Calcul de $First^k(\alpha)$

$First^k(\alpha)$ avec $\alpha = X_1 X_2 \dots X_n$

$$First^k(\alpha) = First^k(X_1) \oplus^k First^k(X_2) \oplus^k \dots \oplus^k First^k(X_n)$$

avec

$$L_1 \oplus^k L_2 =$$

$$\left\{ w \in T^{\leq k} \mid \exists x \in T^*, y \in L_1, z \in L_2 : wx = yz \wedge \begin{aligned} & ((|w| = k) \vee \\ & (|w| < k \wedge x = \epsilon)) \end{aligned} \right\}$$

Calcul de $First^k(X)$

Calcul de $First^k(X)$ avec $X \in (T \cup V)$

Algorithme glouton (greedy) : on augmente des ensembles $First^k(X)$ jusqu'à stabilisation.

Base :

$$\begin{aligned}\forall a \in T : First^k(a) &= \{a\} \\ \forall A \in V : First^k(A) &= \emptyset\end{aligned}$$

Induction : boucle jusqu'à stabilisation :

$$\forall A \in V : First^k(A) \stackrel{\cup}{\leftarrow} \{x \in T^* \mid A \rightarrow Y_1 Y_2 \dots Y_n \wedge x \in First^k(Y_1) \oplus^k First^k(Y_2) \oplus^k \dots \oplus^k First^k(Y_n)\}$$

Calcul de $First(X)$

Exemple (de calcul de $First(A)$) ($\forall A \in V$) avec $G = \langle V, T, P, S' \rangle$)

où P contient :

- $S' \rightarrow E\$$
- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid id$

Au départ

① $\forall A \in V : First(A) = \emptyset$

Étape 1 :

① $First(E') \leftarrow \{\epsilon\}$

② $First(T') \leftarrow \{\epsilon\}$

③ $First(F) \leftarrow \{id\}$

Étape 2 :

① $First(T) \leftarrow \{id\}$

② $First(T') \leftarrow \{*\}$

Étape 3 :

① $First(E) \leftarrow \{id\}$

② $First(E') \leftarrow \{+\}$

Étape 4 :

① $First(S') \leftarrow \{id\}$

② $First(F) \leftarrow \{()\}$

Étape 5 :

① $First(T) \leftarrow \{()\}$

Étape 6 :

① $First(E) \leftarrow \{()\}$

Étape 7 :

① $First(S') \leftarrow \{()\}$

Étape 8 : stabilisation

Calcul de $First(X)$

Exemple (de calcul de $First(A)$) ($\forall A \in V$) avec $G = \langle V, T, P, S' \rangle$)

où P contient :

- $S' \rightarrow E\$$
- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $First(S') = \{id, ()\}$
- $First(E) = \{id, ()\}$
- $First(E') = \{+, \epsilon\}$
- $First(T) = \{id, ()\}$
- $First(T') = \{*, \epsilon\}$
- $First(F) = \{id, ()\}$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid id$

Exemple (de calcul de $First^2(A)$ ($\forall A \in V$) avec la même grammaire G)

Au départ

① $\forall A \in V : First^2(A) = \emptyset$

Étape 1 :

① $First^2(E') \stackrel{\cup}{\Leftarrow} \{\epsilon\}$

② $First^2(T') \stackrel{\cup}{\Leftarrow} \{\epsilon\}$

③ $First^2(F) \stackrel{\cup}{\Leftarrow} \{id\}$

Étape 2 :

① $First^2(T) \stackrel{\cup}{\Leftarrow} \{id\}$

② $First^2(T') \stackrel{\cup}{\Leftarrow} \{*id\}$

Étape 3 :

① $First^2(E) \stackrel{\cup}{\Leftarrow} \{id\}$

② $First^2(E') \stackrel{\cup}{\Leftarrow} \{+id\}$

③ $First^2(T) \stackrel{\cup}{\Leftarrow} \{id*\}$

Étape 4 :

① $First^2(S') \stackrel{\cup}{\Leftarrow} \{id\$ \}$

② $First^2(E) \stackrel{\cup}{\Leftarrow} \{id+, id*\}$

③ $First^2(F) \stackrel{\cup}{\Leftarrow} \{(id\}$

Étape 5 :

① $First^2(S') \stackrel{\cup}{\Leftarrow} \{id+, id*\}$

② $First^2(T) \stackrel{\cup}{\Leftarrow} \{(id\}$

③ $First^2(T') \stackrel{\cup}{\Leftarrow} \{*(\}$

Étape 6 :

① $First^2(E) \stackrel{\cup}{\Leftarrow} \{(id\}$

② $First^2(E') \stackrel{\cup}{\Leftarrow} \{+(\}$

Étape 7 :

① $First^2(S') \stackrel{\cup}{\Leftarrow} \{(id\}$

② $First^2(F) \stackrel{\cup}{\Leftarrow} \{((\}$

Étape 8 :

① $First^2(T) \stackrel{\cup}{\Leftarrow} \{((\}$

Étape 9 :

① $First^2(E) \stackrel{\cup}{\Leftarrow} \{((\}$

Étape 10 :

① $First^2(S') \stackrel{\cup}{\Leftarrow} \{((\}$

Étape 11 : stabilisation

Calcul de $First^2(X)$

Exemple (de calcul de $First^2(A)$ ($\forall A \in V$) avec $G = \langle V, T, P, S' \rangle$)

dont les règles sont :

- $S' \rightarrow E\$$
- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid id$

- $First^2(S') = \{id\$, id+, id*, (id, ((\})\}$
- $First^2(E) = \{id, id+, id*, (id, ((\})\}$
- $First^2(E') = \{\epsilon, +id, +(}\}$
- $First^2(T) = \{id, id*, (id, ((\})\}$
- $First^2(T') = \{\epsilon, *id, *(}\}$
- $First^2(F) = \{id, (id, ((\})\}$

$Follow^k$

Pour déterminer les prévisions

Définition ($Follow^k(\beta)$) Pour la CFG G , un entier positif k et $\beta \in (T \cup V)^*$

$Follow^k(\beta)$ est l'ensemble des strings de terminaux de longueur maximale mais limitée à k caractères qui peuvent suivre un string généré à partir de β

Mathématiquement :

$$Follow^k(\beta) = \{ w \in T^{\leq k} \mid \exists \alpha, \gamma \in (T \cup V)^* : S' \xrightarrow{*} \alpha\beta\gamma \wedge w \in First^k(\gamma) \}$$

Calcul de $Follow^k(X)$

Il ne nous sera nécessaire que de calculer le Follow de variables.

Calcul de $Follow^k(A)$ avec $A \in (T \cup V)$

Algorithme glouton : On augmente les ensembles $Follow^k(B)$ initialement vide, jusqu'à stabilisation.

Base :

$$\forall A \in V : Follow^k(A) = \emptyset$$

Induction :

$$\forall A \in V, A \rightarrow \alpha B \beta \in P \quad (B \in V; \alpha, \beta \in (V \cup T)^*) :$$

$$Follow^k(B) \leftarrow First^k(\beta) \oplus^k Follow^k(A)$$

Jusqu'à stabilisation.

Calcul de $Follow(A)$

Exemple (de calcul de $Follow(A)$) ($\forall A \in V$) avec $G = \langle V, T, P, S' \rangle$)

dont les règles sont :

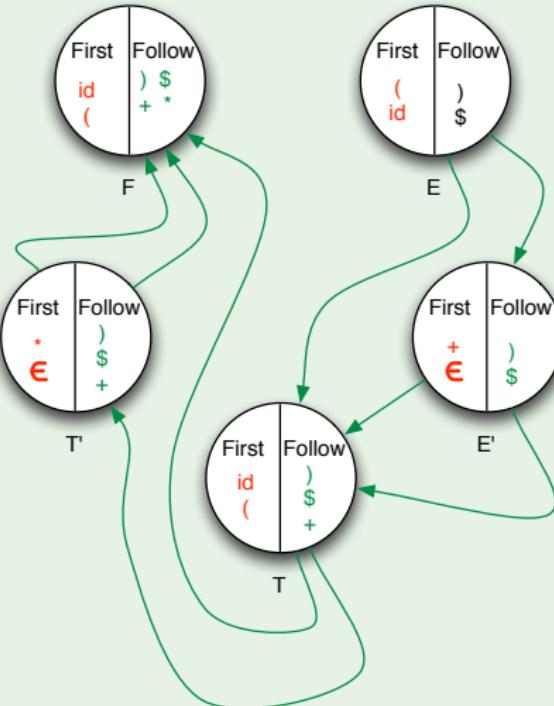
- $S' \rightarrow E\$$
- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid id$

On obtient en appliquant l'algorithme (voir figure) :

- $Follow(S') = \{\epsilon\}$
- $Follow(E) = \{()\}, \$\}$
- $Follow(E') = \{()\}, \$\}$
- $Follow(T) = \{()\}, \$, +\}$
- $Follow(T') = \{()\}, \$, +\}$
- $Follow(F) = \{()\}, \$, +, *\}$

Calcul de $Follow(A)$

Exemple (de calcul de $Follow(A)$) ($\forall A \in V$) avec $G = \langle V, T, P, S' \rangle$



Plan

- 1 Principe du parsing descendant
- 2 Analyseurs prédictifs - $First^k$ - $Follow^k$
- 3 **CFG LL(k)**
- 4 Analyseurs $LL(1)$
- 5 Analyseurs fortement $LL(k)$ ($k > 1$)
- 6 Analyseurs $LL(k)$ ($k > 1$)
- 7 Gestion des erreurs et resynchronisations
- 8 Analyseurs $LL(k)$ récursifs

CFG $LL(k)$

CFG $LL(k)$

$LL(k)$ signifie

- Left scanning
- Leftmost derivation
- k lookahead symbols

Définition (La CFG $G = \langle V, T, P, S' \rangle$ est $LL(k)$ (k un nombre naturel fixé) si)

$\forall w, x_1, x_2 \in T^*$:

- $S' \xrightarrow{G}^* wA\gamma \xrightarrow{G}^* w\alpha_1\gamma \xrightarrow{G}^* wx_1$
- $S' \xrightarrow{G}^* wA\gamma \xrightarrow{G}^* w\alpha_2\gamma \xrightarrow{G}^* wx_2 \Rightarrow \alpha_1 = \alpha_2$
- $First^k(x_1) = First^k(x_2)$

Ce qui implique que pour la forme phrasée $wA\gamma$, si on connaît $First^k(x_1)$ (ce qui reste à analyser après w), on peut déterminer avec la prévision de k symboles, la règle $A \rightarrow \alpha_i$ à appliquer

Problème

En théorie pour déterminer si G (en supposant qu'il génère un langage infini) est $LL(k)$, il faut vérifier une infinité de conditions

Propriété 1 sur les CFG $LL(k)$

Théorème (1 sur les CFG $LL(k)$)

Une CFG G est $LL(k)$

\iff

$\forall A \in V : S' \xrightarrow{*} wA\gamma, A \rightarrow \alpha_i \in P \ (i = 1, 2 : \alpha_1 \neq \alpha_2) :$
 $First^k(\alpha_1\gamma) \cap First^k(\alpha_2\gamma) = \emptyset$

Problème

En théorie pour déterminer si la propriété est satisfaite sur G (en supposant qu'il génère un langage infini), il faut vérifier une infinité de conditions : vu que seuls les k premiers symboles nous intéressent, on peut trouver un algorithme qui fait cela en un temps fini (voir plus loin))

Théorème (1 sur les CFG $LL(k)$)

$$\begin{gathered} \text{Une CFG } G \text{ est } LL(k) \\ \iff \\ \forall S' \xrightarrow{*} wA\gamma, A \rightarrow \alpha_i \in P \ (i = 1, 2 : \alpha_1 \neq \alpha_2) : \\ First^k(\alpha_1\gamma) \cap First^k(\alpha_2\gamma) = \emptyset \end{gathered}$$

Preuve : Par l'absurde

\Rightarrow : On suppose G $LL(k)$ et la propriété non vérifiée.

- $\exists S' \xrightarrow{*} wA\gamma, A \rightarrow \alpha_i \in P \ (i = 1, 2 : \alpha_1 \neq \alpha_2) :$
 $First^k(\alpha_1\gamma) \cap First^k(\alpha_2\gamma) \neq \emptyset$
- Donc $\exists x_1, x_2 :$
- $S' \xrightarrow{*} wA\gamma \Rightarrow w\alpha_1\gamma \Rightarrow^* wx_1$
- $S' \xrightarrow{*} wA\gamma \Rightarrow w\alpha_2\gamma \Rightarrow^* wx_2$
- $First^k(x_1) = First^k(x_2) \wedge \alpha_1 \neq \alpha_2$

Ce qui contredit le fait que G soit $LL(k)$

Propriété 1 sur les CFG $LL(k)$

Théorème (1 sur les CFG $LL(k)$)

$$\begin{array}{c} \text{Une CFG } G \text{ est } LL(k) \\ \iff \\ \forall S' \xrightarrow{*} wA\gamma, A \rightarrow \alpha_i \in P \ (i = 1, 2 : \alpha_1 \neq \alpha_2) : \\ First^k(\alpha_1\gamma) \cap First^k(\alpha_2\gamma) = \emptyset \end{array}$$

Preuve (suite) : Par l'absurde

\Leftarrow : *On suppose la propriété vérifiée et G pas $LL(k)$*

- $\forall S' \xrightarrow{*} wA\gamma, A \rightarrow \alpha_i \in P \ (i = 1, 2 : \alpha_1 \neq \alpha_2) :$
 $First^k(\alpha_1\gamma) \cap First^k(\alpha_2\gamma) = \emptyset \wedge G \text{ n'est pas } LL(k)$
- *Donc puisque G n'est pas $LL(k)$:* $\exists w, x_1, x_2 \in T^* :$

$$\begin{aligned} S' &\xrightarrow{G} wA\gamma \xrightarrow{G} w\alpha_1\gamma \xrightarrow{G} wx_1 \wedge \\ S' &\xrightarrow{G} wA\gamma \xrightarrow{G} w\alpha_2\gamma \xrightarrow{G} wx_2 \wedge \\ First^k(x_1) &= First^k(x_2) \wedge \\ \alpha_1 &\neq \alpha_2 \end{aligned}$$

Or

- $First^k(x_1) \in First^k(\alpha_1\gamma)$
- $First^k(x_2) \in First^k(\alpha_2\gamma)$

Ce qui contredit la propriété.

Principe du parsing descendant
Analyseurs prédictifs - $First^K$ - $Follow^K$
CFG LL(k)
Analyseurs $LL(1)$
Analyseurs fortement $LL(k)$ ($k > 1$)
Analyseurs $LL(k)$ ($k > 1$)
Gestion des erreurs et resynchronisations
Analyseurs $LL(k)$ récursifs

Propriété 2 sur les CFG $LL(k)$

Théorème (sur les CFG $LL(1)$)

Une CFG G est $LL(1)$
 \iff
 $\forall A \rightarrow \alpha_i \in P \ (i = 1, 2 : \alpha_1 \neq \alpha_2) :$
 $First(\alpha_1 Follow(A)) \cap First(\alpha_2 Follow(A)) = \emptyset$

Avantage

Pour déterminer si G est $LL(1)$, il suffit de vérifier un nombre fini de conditions

Propriété 3 sur les CFG $LL(k)$

Théorème (2 sur les CFG $LL(k)$)

Une CFG G est $LL(k)$

\Leftarrow

$\forall A \rightarrow \alpha_i \in P \ (i = 1, 2 : \alpha_1 \neq \alpha_2) :$

$First^k(\alpha_1 Follow^k(A)) \cap First^k(\alpha_2 Follow^k(A)) = \emptyset$

Définition (CFG fortement $LL(k)$)

Une CFG qui satisfait la propriété ci-dessus est fortement $LL(k)$ (strong $LL(k)$)

$LL(1) = \text{fortement } LL(1)$

D'après la propriété sur les langages $LL(1)$ et la définition des langages fortement $LL(k)$, on déduit que tout langage $LL(1)$ est fortement $LL(1)$
Pour $k > 1$ on a : strong $LL(k) \Rightarrow LL(k)$

Avantage

Pour déterminer si G est fortement $LL(k)$, il suffit de vérifier un nombre fini de conditions

A partir de $k = 2$: "fortement $LL(k)$ " $\subset LL(k)$

Exemple (de CFG G $LL(2)$ mais pas fortement $LL(2)$)

dont les règles sont :

- $S' \rightarrow S\$$
- $S \rightarrow aAa$
- $S \rightarrow bABA$

- $A \rightarrow b$
- $A \rightarrow \epsilon$
- $B \rightarrow b$
- $B \rightarrow c$

G est $LL(2)$

- $S' \xrightarrow{1} S\$: First^2(aAa\$) \cap First^2(bABA\$) = \emptyset$
- $S' \xrightarrow{2} aAa\$: First^2(ba\$) \cap First^2(a\$) = \emptyset$
- $S' \xrightarrow{2} bABA\$: First^2(bBa\$) \cap First^2(Ba\$) = \emptyset$
- $S' \xrightarrow{3} bbBa\$: First^2(ba\$) \cap First^2(ca\$) = \emptyset$
- $S' \xrightarrow{3} bBa\$: First^2(ba\$) \cap First^2(ca\$) = \emptyset$

G n'est pas fortement $LL(2)$

- Pour $S : First^2(aAa\dots) \cap First^2(bABA\dots) = \emptyset$
- Pour $B : First^2(b\dots) \cap First^2(c\dots) = \emptyset$
- Mais pour A : $First^2(bFollow^2(A)) \cap First^2(\epsilon Follow^2(A)) \neq \emptyset$
 $(\{ba, bb, bc\} \cap \{a\$, ba, ca\} \neq \emptyset)$

Principe du parsing descendant
Analyseurs prédictifs - $First^K$ - $Follow^K$
CFG LL(k)
Analyseurs $LL(1)$
Analyseurs fortement $LL(k)$ ($k > 1$)
Analyseurs $LL(k)$ ($k > 1$)
Gestion des erreurs et resynchronisations
Analyseurs $LL(k)$ récursifs

Grammaires qui ne sont pas $LL(k)$

Toute CFG G ambiguë n'est pas $LL(k)$ (quelque soit k)

Preuve : évident d'après les définitions de grammaire ambiguë et CFG $LL(k)$

Toute CFG G (où tous les symboles sont utiles), récursive à gauche n'est pas $LL(1)$

Preuve :

Si G est récursive à gauche, on a pour une certaine variable A (utile par hypothèse) de G ,

- $A \rightarrow \alpha \mid \beta$ (donc $First(\beta) \subseteq First(A)$)
 - $A \Rightarrow \alpha \xrightarrow{*} A\gamma$ (donc $First(A) \subseteq First(\alpha)$)
- $\Rightarrow First(\beta) \subseteq First(\alpha)$

Donc G n'est pas $LL(1)$

Grammaires qui ne sont pas $LL(k)$

Remarque :

Généralement, une CFG G récursive à gauche n'est pas $LL(k)$ quelque soit k

Toute CFG G avec 2 règles $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \in P$ et $\alpha \xrightarrow{*} x \wedge |x| \geq k$ n'est pas $LL(k)$

Nettoyage de G

Plus k est grand, plus l'analyseur $LL(k)$ est complexe. On essaye donc que la CFG soit $LL(k)$ avec le k le plus petit possible (si possible 1). Pour cela on :

- Supprime les éventuelles ambiguïtés dans G
- Supprime les récursivités à gauche
- Factorise à gauche

Plan

- 1 Principe du parsing descendant
- 2 Analyseurs prédictifs - $First^k$ - $Follow^k$
- 3 CFG LL(k)
- 4 Analyseurs $LL(1)$
- 5 Analyseurs fortement $LL(k)$ ($k > 1$)
- 6 Analyseurs $LL(k)$ ($k > 1$)
- 7 Gestion des erreurs et resynchronisations
- 8 Analyseurs $LL(k)$ récursifs

Construction d'un analyseur $LL(1)$

Algorithme de construction de la table des Actions $M[A, a]$

Initialisation : $\forall A, a : M[A, a] = \emptyset$

$\forall A \rightarrow \alpha \in P$ (règle numéro i) :

$\forall a \in First(\alpha Follow(A))$

$M[A, a] \leftarrow i$

Note :

La grammaire est $LL(1)$ si et seulement si chaque entrée de la table M a au plus une valeur.

Sinon, cela signifie que des conflits Produce/Produce ne sont pas résolus

Par exemple pour la CFG G avec les règles (1) et (2) : $S \rightarrow aS \mid a$
 $M[A, a] = \{1, 2\}$

Construction de la table d'action d'un analyseur $LL(1)$

Exemple (Construction de M pour G)

dont les règles
sont :

$$\begin{array}{ll} S' \rightarrow E\$ & (0) \\ E \rightarrow TE' & (1) \\ E' \rightarrow +TE' & (2) \\ E' \rightarrow \epsilon & (3) \\ T \rightarrow FT' & (4) \\ T' \rightarrow *FT' & (5) \\ T' \rightarrow \epsilon & (6) \\ F \rightarrow (E) & (7) \\ F \rightarrow id & (8) \end{array}$$

$$\begin{array}{lll} First(E\$) = \{id, ()\} & Follow(S') = \{\epsilon\} \\ First(TE') = \{id, ()\} & Follow(E) = \{\}, \$\} \\ First(+TE') = \{+\} & Follow(E') = \{\}, \$\} \\ First(FT') = \{id, ()\} & Follow(T) = \{\}, \$, +\} \\ First(*FT') = \{* \} & Follow(T') = \{\}, \$, +\} \\ First((E)) = \{(\} & Follow(F) = \{\}, \$, +, *\} \\ First(id) = \{id\} & \end{array}$$

M	id	+	*	()	\$
S'	0			0		
E	1			1		
E'		2			3	3
T	4			4		
T'		6	5		6	6
F	8			7		

Analyse d'un string avec l'analyseur $LL(1)$

Exemple (Analyse de $a * (b + c) \$$)

Sur la pile	Input restant	Action	Output
$S \dashv$	$a * (b + c) \$$	P0	ϵ
$E \$ \dashv$	$a * (b + c) \$$	P1	0
$TE' \$ \dashv$	$a * (b + c) \$$	P4	0 1
$FT'E' \$ \dashv$	$a * (b + c) \$$	P8	0 1 4
$\text{id} T'E' \$ \dashv$	$*(b + c) \$$	M	0 1 4 8
$T'E' \$ \dashv$	$*(b + c) \$$	P5	0 1 4 8
$*FT'E' \$ \dashv$	$*(b + c) \$$	M	0 1 4 8 5
$FT'E' \$ \dashv$	$(b + c) \$$	P7	0 1 4 8 5
$(E)T'E' \$ \dashv$	$(b + c) \$$	M	0 1 4 8 5 7
$E)T'E' \$ \dashv$	$b + c) \$$	P1	0 1 4 8 5 7
$TE')T'E' \$ \dashv$	$b + c) \$$	P4	0 1 4 8 5 7 1
$FT'E')T'E' \$ \dashv$	$b + c) \$$	P8	0 1 4 8 5 7 1 4
$\text{id} T'E')T'E' \$ \dashv$	$b + c) \$$	M	0 1 4 8 5 7 1 4 8
$T'E')T'E' \$ \dashv$	$+c) \$$	P6	0 1 4 8 5 7 1 4 8
$E')T'E' \$ \dashv$	$+c) \$$	P2	0 1 4 8 5 7 1 4 8 6
$+TE')T'E' \$ \dashv$	$+c) \$$	M	0 1 4 8 5 7 1 4 8 6 2
$TE')T'E' \$ \dashv$	$c) \$$	P4	0 1 4 8 5 7 1 4 8 6 2
$FT'E')T'E' \$ \dashv$	$c) \$$	P8	0 1 4 8 5 7 1 4 8 6 2 4
$\text{id} T'E')T'E' \$ \dashv$	$c) \$$	M	0 1 4 8 5 7 1 4 8 6 2 4 8
$T'E')T'E' \$ \dashv$	$) \$$	P6	0 1 4 8 5 7 1 4 8 6 2 4 8
$E')T'E' \$ \dashv$	$) \$$	P3	0 1 4 8 5 7 1 4 8 6 2 4 8 6
$)T'E' \$ \dashv$	$) \$$	M	0 1 4 8 5 7 1 4 8 6 2 4 8 6 3
$T'E' \$ \dashv$	$\$$	P6	0 1 4 8 5 7 1 4 8 6 2 4 8 6 3
$E' \$ \dashv$	$\$$	P3	0 1 4 8 5 7 1 4 8 6 2 4 8 6 3 6
$\$ \dashv$	$\$$	A	0 1 4 8 5 7 1 4 8 6 2 4 8 6 3 6 3

Plan

- 1 Principe du parsing descendant
- 2 Analyseurs prédictifs - $First^k$ - $Follow^k$
- 3 CFG LL(k)
- 4 Analyseurs $LL(1)$
- 5 Analyseurs fortement $LL(k)$ ($k > 1$)**
- 6 Analyseurs $LL(k)$ ($k > 1$)
- 7 Gestion des erreurs et resynchronisations
- 8 Analyseurs $LL(k)$ récursifs

Construction d'un analyseur fortement $LL(k)$

Algorithme de construction de la table des Actions $M[A, u]$ ($u \in T^{\leq k}$)

Initialisation : $\forall A, u : M[A, u] = \text{Error}$

$\forall A \rightarrow \alpha \in P$ (règle numéro i) :

$\forall u \in First^k(\alpha Follow^k(A))$

$M[A, u] \leftarrow i$

Notes :

- La grammaire est fortement $LL(k)$ si et seulement si chaque entrée de la table M a au plus une valeur.
Sinon, cela signifie que des conflits Produce/Produce ne sont pas résolus
- En pratique on stocke M de façon compacte

Plan

- 1 Principe du parsing descendant
- 2 Analyseurs prédictifs - $First^k$ - $Follow^k$
- 3 CFG LL(k)
- 4 Analyseurs $LL(1)$
- 5 Analyseurs fortement $LL(k)$ ($k > 1$)
- 6 Analyseurs $LL(k)$ ($k > 1$)**
- 7 Gestion des erreurs et resynchronisations
- 8 Analyseurs $LL(k)$ récursifs

Analyseurs $LL(k)$ pour les CFG non fortement $LL(k)$

Explication

Cela signifie que pour $A \rightarrow \alpha_1 \mid \alpha_2$

$$First^k(\alpha_1 Follow^k(A)) \cap First^k(\alpha_2 Follow^k(A)) \neq \emptyset$$

Ce qui signifie que :

- contrairement au CFG “fortement $LL(k)$ ” où les prévisions globales suffisent
- pour les CFG $LL(k)$ non fortement $LL(k)$, il faut utiliser les prévisions locales

Autrement dit, en pratique, lors de l’analyse, le sommet de la pile et la prévision ne suffisent pas pour déterminer le Produce à effectuer ; il faut aussi “retenir” dans quel contexte local on se trouve.

Déterminons le problème

Reprendons la définition :

La CFG G est $LL(k) \iff \forall w, x_1, x_2 \in T^* :$

- $S' \xrightarrow{G}^* wA\gamma \xrightarrow{G}^* w\alpha_1\gamma \xrightarrow{G}^* wx_1$
- $S' \xrightarrow{G}^* wA\gamma \xrightarrow{G}^* w\alpha_2\gamma \xrightarrow{G}^* wx_2 \Rightarrow \alpha_1 = \alpha_2$
- $First^k(x_1) = First^k(x_2)$

Ce qui signifie que dans le contexte $wA\gamma$, une prévision de k symboles permet de déterminer de façon unique la prochaine production à utiliser.

Déterminons le problème

Prenons la grammaire $LL(2)$ mais pas fortement $LL(2)$

de règles :

- $S' \rightarrow S\$$
- $S \rightarrow aAa$
- $S \rightarrow bABA$
- $A \rightarrow b$
- $A \rightarrow \epsilon$
- $B \rightarrow b$
- $B \rightarrow c$

Suivant que l'on a fait $S \rightarrow aAa$ ou $S \rightarrow bABA$, une prévision “*ba*” signifie que l'on doit faire un **produce** $A \rightarrow b$ ou $A \rightarrow \epsilon$

De façon générale il faut donc calculer le “**follow local**” de chaque variable pour chaque contexte possible.

Transformation de CFG $LL(k)$ en fortement $LL(k)$

Etant donné qu'il n'y a qu'un nombre fini de variables et de prévisions de longueur inférieure ou égale à k , on peut expliciter l'ensemble des **follow locaux** possibles.

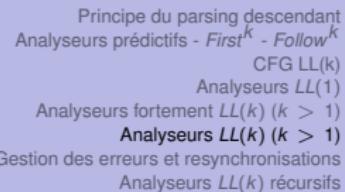
On va transformer G en G' où chaque variable A sera remplacée par un couple $[A, L]$

- le nom **A** de la variable dans G
- le follow local, c'est-à-dire l'ensemble **L** des prévisions locales possibles

Pour l'exemple précédent, la variable **A** se transforme en **$[A, \{a$\}]$** et **$[A, \{ba, ca\}]$**

Propriété

Si G est $LL(k)$ alors G' est fortement $LL(k)$



Transformation de CFG $LL(k)$ en fortement $LL(k)$

Algorithme de transformation de CFG $G\ LL(k)$ en G' fortement $LL(k)$

Ayant $G = \langle V, T, P, S' \rangle$ on construit $G' = \langle V', T, P', S'' \rangle$ comme suit :

Initialement :

$$V' \Leftarrow \{[S', \{\epsilon\}]\}$$

$$S'' = [S', \{\epsilon\}]$$

$$P' = \emptyset$$

Répéter jusqu'à ce que toutes les nouvelles variables aient leurs règles :

Ayant

$$[A, L] \in V' \wedge A \rightarrow \alpha \equiv x_0 B_1 x_1 B_2 \dots B_m x_m \in P \quad (x_i \in T^*, B_i \in V)$$

$$P' \stackrel{\cup}{\Leftarrow} [A, L] \rightarrow T(\alpha) \text{ avec}$$

$$T(\alpha) = x_0[B_1, L_1]x_1[B_2, L_2]\dots[B_m, L_m]x_m$$

$$L_i = First^k(x_i B_{i+1} \dots B_m x_m . L)$$

$$\forall 1 \leq i \leq m : V' \stackrel{\cup}{\Leftarrow} [B_i, L_i]$$

Transformation de CFG $LL(k)$ en fortement $LL(k)$

Exemple (de transformation de $G\ LL(2)$ en G' fortement $LL(2)$)

$G = \langle V, T, P, S' \rangle$ de règles :

$$S' \rightarrow S\$ \quad (0)$$

$$S \rightarrow aAa \quad (1)$$

$$S \rightarrow bABA \quad (2)$$

$$A \rightarrow b \quad (3)$$

$$A \rightarrow \epsilon \quad (4)$$

$$B \rightarrow b \quad (5)$$

$$B \rightarrow c \quad (6)$$

devient : $G' = \langle V', T, P', S'' \rangle$ avec :

Règle	numéro dans G'	numéro dans G
$[S', \{\epsilon\}] \rightarrow [S, \{\$\}] \$$	(0')	(0)
$[S, \{\$\}] \rightarrow a[A, \{a\$}] a$	(1')	(1)
$[S, \{\$\}] \rightarrow b[A, \{ba, ca\}] [B, \{a\$}] a$	(2')	(2)
$[A, \{a\$}] \rightarrow b$	(3')	(3)
$[A, \{a\$}] \rightarrow \epsilon$	(4')	(4)
$[A, \{ba, ca\}] \rightarrow b$	(5')	(3)
$[A, \{ba, ca\}] \rightarrow \epsilon$	(6')	(4)
$[B, \{a\$}] \rightarrow b$	(7')	(5)
$[B, \{a\$}] \rightarrow c$	(8')	(6)

Transformation de CFG $LL(k)$ en fortement $LL(k)$

Exemple (de transformation de $G\ LL(2)$ en G' fortement $LL(2)$)

$G' = \langle V', T, P', S'' \rangle$ avec :

Règle	numéro dans G'	numéro dans G
$[S', \{\epsilon\}] \rightarrow [S, \{\$\}]\$$	(0')	(0)
$[S, \{\$\}] \rightarrow a[A, \{a\$}\]a$	(1')	(1)
$[S, \{\$\}] \rightarrow b[A, \{ba, ca}\][B, \{a\$}\]a$	(2')	(2)
$[A, \{a\$}\] \rightarrow b$	(3')	(3)
$[A, \{a\$}\] \rightarrow \epsilon$	(4')	(4)
$[A, \{ba, ca}\] \rightarrow b$	(5')	(3)
$[A, \{ba, ca}\] \rightarrow \epsilon$	(6')	(4)
$[B, \{a\$}\] \rightarrow b$	(7')	(5)
$[B, \{a\$}\] \rightarrow c$	(8')	(6)

M	ab	aa	bb	bc	ba	ca	a\$
$[S', \epsilon]$	0'	0'	0'	0'			
$[S, \{\$\}]$	1'	1'	2'	2'			
$[A, \{a\$}\]$					3'		4'
$[A, \{ba, ca}\]$			5'	5'	6'	6'	
$[B, \{a\$}\]$					7'	8'	

Analyse avec la CFG fortement $LL(k)$ construite

Exemple (Analyse de $bba\$$)

M	ab	aa	bb	bc	ba	ca	a\$
$[S', \epsilon]$	0'	0'	0'	0'			
$[S, \{\$\}]$	1'	1'	2'	2'			
$[A, \{a\$}\}]$					3'		4'
$[A, \{ba, ca}\}]$			5'	5'	6'	6'	
$[B, \{a\$}\}]$					7'	8'	

Lors de l'analyse, on peut faire correspondre l'output aux règles de G

Sur la pile	Input	Action	Out. de G'	Out. de G
$[S', \epsilon] \dashv$	$bba\$$	P0'	ϵ	ϵ
$[S, \$] \$ \dashv$	$bba\$$	P2'	0'	0
$b [A, \{ba, ca}\][B, \{a\$}\}] a\$ \dashv$	$bba\$$	M	0' 2'	0 2
$[A, \{ba, ca}\][B, \{a\$}\}] a\$ \dashv$	$ba\$$	P6'	0' 2'	0 2
$[B, \{a\$}\}] a\$ \dashv$	$ba\$$	P7'	0' 2' 6'	0 2 4
$ba\$ \dashv$	$ba\$$	M	0' 2' 6' 7'	0 2 4 5
$a\$ \dashv$	$ba\$$	M	0' 2' 6' 7'	0 2 4 5
$\$ \dashv$	$ba\$$	A	0' 2' 6' 7'	0 2 4 5

Plan

- 1 Principe du parsing descendant
- 2 Analyseurs prédictifs - $First^k$ - $Follow^k$
- 3 CFG LL(k)
- 4 Analyseurs $LL(1)$
- 5 Analyseurs fortement $LL(k)$ ($k > 1$)
- 6 Analyseurs $LL(k)$ ($k > 1$)
- 7 **Gestion des erreurs et resynchronisations**
- 8 Analyseurs $LL(k)$ récursifs

Récupération sur erreur en analyse prédictive

Lors d'une erreur, l'analyseur peut décider

- d'informer de l'erreur et de s'arrêter.
- d'essayer de continuer (sans produire de code) pour détecter d'éventuelles erreurs ultérieures (resynchronisation).

On détecte une erreur lorsque

- le terminal au sommet de la pile ne correspond pas au prochain symbole à l'entrée
- la variable au sommet de la pile ne prévoit pas la prévision à l'entrée

Dans ce cas on peut essayer de se resynchroniser en modifiant

- la pile
- l'input

Exemple : récupération en mode panique

Gestion des erreurs en mode panique

En cas d'erreur :

- Popper les terminaux sur la pile jusqu'à la première variable
- Sauter les symboles d'input qui ne correspondent pas à une prévision ou un **symbole de resynchronisation associé à la variable**
- Si le symbole rencontré est un **symbole de synchronisation**, Popper la variable de la pile
- Continuer l'analyse (en espérant que la synchronisation ait eu lieu)

Récupération en mode panique pour un analyseur $LL(1)$

Symboles de synchronisation

Si l'entrée n'est pas déjà occupée, on ajoute à la table des actions M , pour chaque variable A , les symboles de synchronisation de A ; par exemple :

- les symboles de $Follow(A)$
- d'autres symboles bien choisis

Exemple (Table d'action pour l'analyseur $LL(1)$ de $G = \langle V, T, P, S' \rangle$)

dont les règles sont :

$$\begin{array}{lll} S' \rightarrow E & (0) & T \rightarrow FT' & (4) \\ E \rightarrow TE' & (1) & T' \rightarrow *FT' \mid \epsilon & (5) \mid (6) \\ E' \rightarrow +TE' \mid \epsilon & (2) \mid (3) & F \rightarrow (E) \mid id & (7) \mid (8) \end{array}$$

M	<i>id</i>	+	*	()	\$
S'	0			0		
E	1			1	<i>sync</i>	<i>sync</i>
E'		2			3	3
T	4	<i>sync</i>		4	<i>sync</i>	<i>sync</i>
T'		6	5		6	6
F	8	<i>sync</i>	<i>sync</i>	7	<i>sync</i>	<i>sync</i>

Analyse d'un string erroné avec l'analyseur $LL(1)$

Exemple (Analyse de $+a^*+b\$$)

Sur la pile	Input restant	Action	Output
$S' \vdash$	$+a^*+b\$$	Erreur : saute +	ϵ
$S' \vdash$	$a^*+b\$$	P0	$*$
$E \$ \vdash$	$a^*+b\$$	P1	$* 0$
$TE' \$ \vdash$	$a^*+b\$$	P4	$* 0 1$
$FT'E' \$ \vdash$	$a^*+b\$$	P8	$* 0 1 4$
$idT'E' \$ \vdash$	$a^*+b\$$	M	$* 0 1 4 8$
$T'E' \$ \vdash$	$*+b\$$	P5	$* 0 1 4 8$
$*FT'E' \$ \vdash$	$*+b\$$	M	$* 0 1 4 8 5$
$FT'E' \$ \vdash$	$+b\$$	Erreur : + est sync : pop F	$* 0 1 4 8 5$
$T'E' \$ \vdash$	$+b\$$	P6	$* 0 1 4 8 5 *$
$E' \$ \vdash$	$+b\$$	P2	$* 0 1 4 8 5 * 6$
$+TE' \$ \vdash$	$+b\$$	M	$* 0 1 4 8 5 * 6 2$
$TE' \$ \vdash$	$b\$$	P4	$* 0 1 4 8 5 * 6 2$
$FT'E' \$ \vdash$	$b\$$	P8	$* 0 1 4 8 5 * 6 2 4$
$idT'E' \$ \vdash$	$b\$$	M	$* 0 1 4 8 5 * 6 2 4 8$
$T'E' \$ \vdash$	$b\$$	P6	$* 0 1 4 8 5 * 6 2 4 8$
$E' \$ \vdash$	$\$$	P3	$* 0 1 4 8 5 * 6 2 4 8 6$
$\$ \vdash$	$\$$	A (avec erreurs)	$* 0 1 4 8 5 * 6 2 4 8 6 3$

Plan

- 1 Principe du parsing descendant
- 2 Analyseurs prédictifs - $First^k$ - $Follow^k$
- 3 CFG LL(k)
- 4 Analyseurs $LL(1)$
- 5 Analyseurs fortement $LL(k)$ ($k > 1$)
- 6 Analyseurs $LL(k)$ ($k > 1$)
- 7 Gestion des erreurs et resynchronisations
- 8 Analyseurs $LL(k)$ récursifs

Principe du parsing descendant
 Analyseurs prédictifs - $First^K$ - $Follow^K$
 CFG LL(k)
 Analyseurs $LL(1)$
 Analyseurs fortement $LL(k)$ ($k > 1$)
 Analyseurs $LL(k)$ ($k > 1$)
 Gestion des erreurs et resynchronisations
 Analyseurs $LL(k)$ récursifs

Exemple d'analyseur $LL(1)$ récursif

Un analyseur $LL(k)$ peut facilement être encodé sous forme de programme (récursif) où l'analyse de chaque variable de G est effectuée par une procédure récursive.

Le code qui suit donne un analyseur $LL(1)$ récursif pour G dont les règles sont :

$$\begin{aligned} E &\rightarrow TE' & (1) \\ E' &\rightarrow +TE' \mid \epsilon & (2) \mid (3) \end{aligned}$$

$$\begin{aligned} T &\rightarrow FT' & (4) \\ T' &\rightarrow *FT' \mid \epsilon & (5) \mid (6) \\ F &\rightarrow (E) \mid id & (7) \mid (8) \end{aligned}$$

Code récursif d'un analyseur $LL(1)$

```
/* main.c */
/* E -> TE'; E' -> +TE' | e ; T -> FT' ;
T' -> *FT' | e; F -> (E) | id */

#define NOTOK 0
#define OK 1
char Phrase[100];
int CurToken;
int Res;

int Match(char t)
{
    if (Phrase[CurToken]==t)
    {
        CurToken++;
        return OK;
    }
    else
    {
        return NOTOK;
    }
}

void Send_output(int no)
{
    printf("%d  ",no);
}
```

Code récursif d'un analyseur $LL(1)$

```
int E(void)
{
    Send_output(1);
    if(T()==OK)
        if(E2()==OK)
            return(OK);
        return(NOTOK);
}
```

Code récursif d'un analyseur $LL(1)$

```
int E2(void)
{
    switch (Phrase[CurToken])
    {
        case '+':
            Send_output(2);
            Match('+');
            if(T()==OK)
                if(E2()==OK)
                    return(OK);
            break;
        case '$':
        case ')':
            Send_output(3);
            return(OK);
    }
    return(NOTOK);
}
```

Code récursif d'un analyseur $LL(1)$

```
int T(void)
{
    Send_output(4);
    if(F() == OK)
        if(T2() == OK)
            return(OK);
        return(NOTOK);
}
```

Code récursif d'un analyseur $LL(1)$

```
int T2(void)
{
    switch (Phrase[CurToken])
    {
        case '*':
            Send_output(5);
            Match('*');
            if (F() == OK)
                if (T2() == OK)
                    return (OK);
            break;
        case '+':
        case '$':
        case ')':
            Send_output(6);
            return (OK);
    }
    return (NOTOK);
}
```

Code récursif d'un analyseur $LL(1)$

```
int F(void)
{
    switch (Phrase[CurToken])
    {
        case '(':
            Send_output(7);
            Match('(');
            if (E() == OK)
                if (Match(')') == OK)
                    return (OK);
                break;
        case 'n':
            Send_output(8);
            Match('n');
            return (OK);
            break;
    }
    return (NOTOK);
}
```

Code récursif d'un analyseur $LL(1)$

```
int main()
{
    scanf("%s", Phrase);
    if (E() != OK)
    {
        printf("error1\n");
    }
    else
    {
        if (Match('$') == OK)
        {
            printf("\n");
        }
        else
        {
            printf("error2\n");
        }
    }
}
```

Chapitre 10 : Les parseurs LR(k)

- 1 Principe de parsing ascendant
- 2 CFG LR(k)
- 3 Parser LR(0)
- 4 Parser LR(1)
- 5 Parser SLR(1)
- 6 Parser LALR(1)
- 7 Classes LL vs LR
- 8 L'outil Yacc (Bison)

Plan

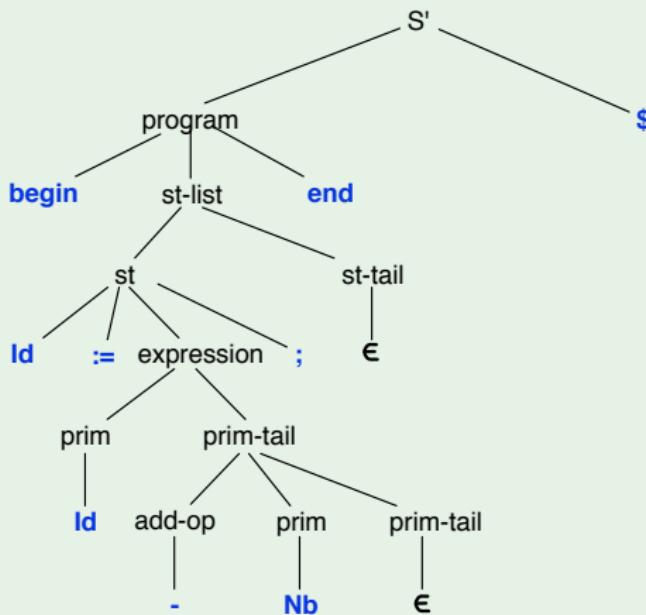
- 1 Principe de parsing ascendant
- 2 CFG LR(k)
- 3 Parser LR(0)
- 4 Parser LR(1)
- 5 Parser SLR(1)
- 6 Parser LALR(1)
- 7 Classes LL vs LR
- 8 L'outil Yacc (Bison)

Exemple (Grammaire d'un langage très simple)

Règles	Règles de production	
0	S'	\rightarrow program \$
1	program	\rightarrow begin st-list end
2	st-list	\rightarrow st st-tail
3	st-tail	\rightarrow st st-tail
4	st-tail	\rightarrow ϵ
5	st	\rightarrow Id := expression ;
6	st	\rightarrow read (id-list) ;
7	st	\rightarrow write (expr-list) ;
8	id-list	\rightarrow Id id-tail
9	id-tail	\rightarrow , Id id-tail
10	id-tail	\rightarrow ϵ
11	expr-list	\rightarrow expression expr-tail
12	expr-tail	\rightarrow , expression expr-tail
13	expr-tail	\rightarrow ϵ
14	expression	\rightarrow prim prim-tail
15	prim-tail	\rightarrow add-op prim prim-tail
16	prim-tail	\rightarrow ϵ
17	prim	\rightarrow (expression)
18	prim	\rightarrow Id
19	prim	\rightarrow Nb
20 21	add-op	\rightarrow + -

Exemple d'arbre syntaxique et de dérivation droite

Exemple (Dérivation droite correspondant à l'arbre syntaxique)



Dérivation droite $S'_G \xrightarrow{*} \text{begin Id := Id - Nb ; end :}$
0 1 2 4 5 14 15 16 19 21 18

Dérivation droite

Exemple (Dérivation droite complète correspondante)

Règle	protophrase	
0	S'	⇒
1	program \$	⇒
2	begin st-list end \$	⇒
4	begin st end \$	⇒
5	begin Id := expression ; end \$	⇒
14	begin Id := prim prim-tail ; end \$	⇒
15	begin Id := prim add-op prim prim-tail ; end \$	⇒
16	begin Id := prim add-op prim ; end \$	⇒
19	begin Id := prim add-op Nb ; end \$	⇒
21	begin Id := prim - Nb ; end \$	⇒
18	begin Id := Id - Nb ; end \$	⇒

Ébauche de structure d'un parser ascendant

PDA à un état et avec output. Au départ la pile est vide.

Le PDA part du string d'input et construit l'arbre de bas en haut. Pour cela, il opère par :

- ① Mise de symbole d'input sur la pile jusqu'à identification d'une partie droite α (handle) de règle $A \rightarrow \alpha$
- ② "Réduction" : remplacement^a de α par A

Donc, le PDA peut effectuer 4 types d'actions :

- **Shift** : lecture d'un symbole d'input et push de ce symbole sur la pile
- **Reduce** : le sommet de la pile α correspond au **handle** (la partie droite d'une règle numéro $i : A \rightarrow \alpha$), est remplacée par A sur la pile et le numéro de règle utilisée i est écrit sur output
- **Accept** : Correspond à un Reduce de la règle $S' \rightarrow S\$$ (qui montre que l'input a été complètement lu et analysé) ; on termine l'analyse avec succès
- **Erreur** : Si aucun shift ni reduce n'est possible

^aCorrespond formellement à $|\alpha|$ pops suivis d'un push de A

Ébauche de structure d'un parser ascendant

Remarque :

- On peut remarquer que l'analyse correspond à une **analyse droite à l'envers** : on part du string et remonte dans les dérivations vers le symbole de départ. L'analyse est faite à l'envers étant donné qu'on lit l'input de gauche à droite.
- L'output sera donc construit à l'envers (tout nouvel output se met avant ce qui a déjà été produit) pour obtenir cette dérivation droite.
- Lors d'une analyse correcte, la pile contient toujours **un prefixe viable** c'est-à-dire un préfixe de forme phrasée "**qui ne dépasse pas le handle**" c'est à dire formellement que si le handle à trouver est $A \rightarrow \alpha$ et correspond à la dérivation $\gamma A x \Rightarrow_G \gamma \alpha x$, la pile ne peut contenir qu'un préfixe de $\gamma \alpha$.

Dérivation droite

Exemple (Dérivation droite complète correspondante)

Sur la pile	Input restant	Act	Output
⊤	begin Id := Id - Nb ; end \$	S	€
⊤ begin	Id := Id - Nb ; end \$	S	€
⊤ begin Id	:= Id - Nb ; end \$	S	€
⊤ begin Id :=	Id - Nb ; end \$	S	€
⊤ begin Id := Id	- Nb ; end \$	R18	€
⊤ begin Id := prim	- Nb ; end \$	S	18
⊤ begin Id := prim -	Nb ; end \$	S	18
⊤ begin Id := prim -	Nb ; end \$	R21	18
⊤ begin Id := prim add-op	Nb ; end \$	S	21 18
⊤ begin Id := prim add-op Nb	; end \$	R19	21 18
⊤ begin Id := prim add-op prim	; end \$	R16	19 21 18
⊤ begin Id := prim add-op prim prim-tail	; end \$	R15	16 19 21 18
⊤ begin Id := prim prim-tail	; end \$	R14	15 16 19 21 18
⊤ begin Id := expression	; end \$	S	14 15 16 19 21 18
⊤ begin Id := expression ;	end \$	R5	14 15 16 19 21 18
⊤ begin st	end \$	R4	5 14 15 16 19 21 18
⊤ begin st st-tail	end \$	R2	4 5 14 15 16 19 21 18
⊤ begin st-list	end \$	S	2 4 5 14 15 16 19 21 18
⊤ begin st-list end	\$	R1	2 4 5 14 15 16 19 21 18
⊤ program	\$	S	1 2 4 5 14 15 16 19 21 18
⊤ program \$	€	A	1 2 4 5 14 15 16 19 21 18
⊤ S'	€		0 1 2 4 5 14 15 16 19 21 18

Où :

S : Shift

Ri : Reduce avec la règle *i*

A : Accept (correspond à un Reduce avec la règle 0)

E : Erreur (ou blocage qui demande un backtracking)

Points à améliorer dans l'ébauche de parser ascendant

Critique de l'ébauche de parser ascendant

- Tel quel, ce parser est extrêmement inefficace car il doit effectuer du **backtracking** pour explorer toutes les possibilités.
- Dans ce type de parser, un **choix doit avoir lieu au moment de l'action "Reduce" et "Shift"**
- Si plusieurs choix sont possibles et qu'aucun critère dans la méthode ne permet de choisir, on parlera de **conflit Shift/Reduce** ou **Reduce/Reduce**
- Sans guide au moment de ce choix, il faudrait éventuellement explorer tous les Shift et Reduce possibles : on aurait donc un parser qui prendrait un temps exponentiel (typiquement dans la longueur de l'input) ce qui est inadmissible !

Plan

- 1 Principe de parsing ascendant
- 2 CFG LR(k)
- 3 Parser LR(0)
- 4 Parser LR(1)
- 5 Parser SLR(1)
- 6 Parser LALR(1)
- 7 Classes LL vs LR
- 8 L'outil Yacc (Bison)

CFG $LR(k)$ $LR(k)$ signifie

- Left scanning
- Rightmost derivation
- k lookahead symbols

Définition (La CFG $G' = \langle V, T, P, S' \rangle$ est $LR(k)$ (k un nombre naturel fixé) si)

- $S' \xrightarrow{*} \gamma Ax \Rightarrow_G \gamma \alpha x$ $\gamma Ax' = \delta By : c'est\text{-à}-dire$
- $S' \xrightarrow{*} \delta By \Rightarrow_G \delta \beta y = \gamma \alpha x'$ $\gamma = \delta,$ \Rightarrow $A = B,$
- $First^k(x) = First^k(x')$ $x' = y$

Intuitivement cela signifie qu'en examinant $First^k(x)$ on peut déterminer univoquement le handle $A \rightarrow \alpha$ dans $\gamma \alpha x$

Grammaires non $LR(k)$

Exemple (de grammaire ni $LR(0)$ ni $LR(1)$)

La grammaire G' suivante n'est pas $LR(0)$ ni $LR(1)$

$$\begin{aligned} S' &\rightarrow S\$ \\ S &\rightarrow Sa \mid a \mid \epsilon \end{aligned}$$

- $S' \xrightarrow[G]{2} Sa\$ \Rightarrow_G a\$$
- $S' \xrightarrow[G]{2} Sa\$ \Rightarrow_G aa\$$
- $First(a\$) = First(aa\$)$

$$\begin{array}{lll} A = S & \gamma = \epsilon & \alpha = \epsilon \\ x = a\$ & x' = aa\$ & B = S \\ \delta = \epsilon & \beta = a & y = a\$ \end{array}$$

Mais $\delta By = Sa\$ \neq Saa\$ = \gamma Ax'$

Notons que G' est aussi ambiguë.

Théorème (Toute CFG G' ambiguë n'est pas $LR(k)$ quelque soit k)

Types d'analyseurs $LR(k)$ étudiés

Type d'analyseurs ascendants étudiés ici

Nous allons voir 3 types d'analyseurs $LR(k)$

- Les analyseurs “*LR canoniques*” les plus puissants mais coûteux
- Les analyseurs “*Simple LR*” (*SLR*) : qui sont moins coûteux mais moins puissants
- Les analyseurs “*LALR*” plus puissants que *SLR(1)* (un peu moins puissants que *LR(1)*) et moins coûteux que les *LR* canoniques

Fonctionnement d'un analyseur ascendant

Ces 3 types d'analyseurs ascendants utilisent

- une **pile**
- une **table Action** qui en fonction du sommet de la pile et de la prévision détermine s'il faut faire un **Shift** ou un **Reduce *i*** où *i* est le numéro de la règle à utiliser
- une **table Successeur** qui détermine ce qui va être mis sur la pile (voir plus loin)

Plan

- 1 Principe de parsing ascendant
- 2 CFG LR(k)
- 3 Parser LR(0)**
- 4 Parser LR(1)
- 5 Parser SLR(1)
- 6 Parser LALR(1)
- 7 Classes LL vs LR
- 8 L'outil Yacc (Bison)

Analyse $LR(0)$

Principe de la construction de l'analyseur

- Le principe d'un analyseur **LR** est de déterminer un **handle** et d'effectuer la réduction
- On va construire un automate fini déterministe qui reconnaît tous les préfixes viables et détermine quand on arrive au handle

Notion d'item $LR(0)$

- On utilise dans cette construction la notion de **$LR(0)$ -item**
- Un $LR(0)$ -item est une règle de production avec un \bullet quelque part dans la partie droite de la règle
- Pour la règle $A \rightarrow \alpha : A \rightarrow \alpha_1 \bullet \alpha_2$ avec $\alpha = \alpha_1 \alpha_2$ signifie **qu'il est possible** que l'on soit
 - en cours d'analyse d'une règle $A \rightarrow \alpha$,
 - après l'analyse de α_1 (α_1 est sur la pile)
 - avant l'analyse de α_2

Analyse $LR(0)$

Remarque sur $A \rightarrow \alpha_1 \bullet \alpha_2$

Il s'agit de possibilités qui seront regroupées. On pourra par exemple avoir les 2 possibilités suivantes regroupées :

- $S \rightarrow a \bullet AC$
- $S \rightarrow a \bullet b$

2 types de $LR(0)$ -items sont particuliers :

- $A \rightarrow \bullet\alpha$ qui prédit que l'on peut débuter une analyse utilisant la règle $A \rightarrow \alpha$
- $A \rightarrow \alpha\bullet$ qui reconnaît que l'on a terminé l'analyse d'une règle $A \rightarrow \alpha$ (et détermine, s'il n'y a pas de conflit, que l'on peut faire le réduire correspondant)

Construction de l'automate caractéristique $LR(0)$

Automate caractéristique $LR(0)$

- On va construire un **automate fini déterministe** qui en fonction des caractères d'input avalés ou des variables déjà acceptées, détermine les débuts de **handle** possibles.
- Quand l'automate arrive dans un état “accepteur” càd contenant un $LR(0)$ -item $A \rightarrow \alpha \bullet$ (où un **handle est complet**, $A \rightarrow \alpha$ est reconnu, on peut effectuer la **réduction**, c'est à dire remplacer α par A)
- Cet automate reconnaît comme langage l'ensemble des préfixes viables de G'**

Construction de l'automate caractéristique $LR(0)$

Construction de l'automate caractéristique $LR(0)$

- Au départ on marque que l'on doit analyser S' : $S' \rightarrow \bullet S\$\text{}$
- Un $LR(0)$ -item $A \rightarrow \gamma_1 \bullet B\gamma_2$ où le \bullet est avant une variable B : signifie que l'on “prédit” que l'on devra analyser B ; en d'autres termes, une partie droite de règle $B \rightarrow \beta_j$: il faut donc rajouter $B \rightarrow \bullet\beta_j$, et ce pour toutes les B -productions $B \rightarrow \beta_j$,
- Ajouter, en utilisant le même critère, toutes les $LR(0)$ -items $B \rightarrow \bullet\beta$ jusqu'à stabilisation s'appelle **l'opération de fermeture**
- L'opération de fermeture permet d'obtenir l'ensemble des possibilités sur l'état de l'analyse de G' et constitue **un état de l'automate caractéristique $LR(0)$**
- Les transitions $s \xrightarrow{X} s'$ de cet automate fini expriment que l'analyse du symbole (terminal ou variable) X labellant la transition est terminée

Exemple d'automate caractéristique $LR(0)$

Exemple (De construction d'automate caractéristique $LR(0)$)

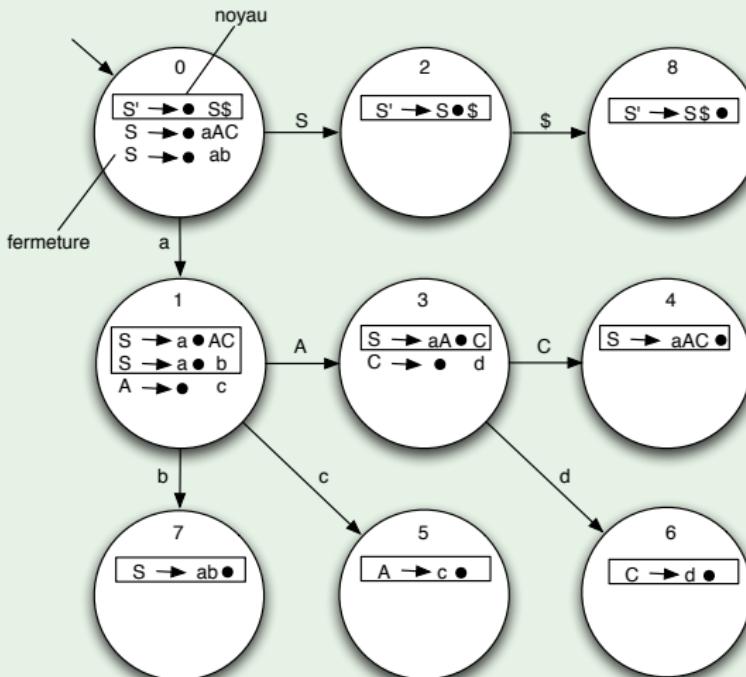
Pour G' qui a les règles suivantes :

$$\begin{array}{lcl} S' & \rightarrow & S\$ \quad (0) \\ S & \rightarrow & aAC \quad (1) \\ S & \rightarrow & ab \quad (2) \\ A & \rightarrow & c \quad (3) \\ C & \rightarrow & d \quad (4) \end{array}$$

L'automate caractéristique $LR(0)$ est représenté par la figure suivante :

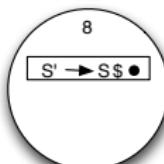
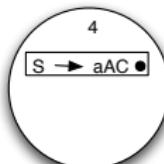
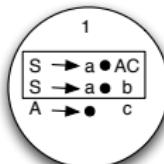
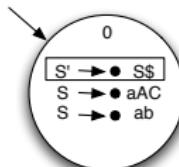
Exemple d'automate caractéristique $LR(0)$

Exemple (De construction d'automate caractéristique $LR(0)$)



Interprétation de l'automate caractéristique $LR(0)$

État de l'automate caractéristique :



État de l'analyse :

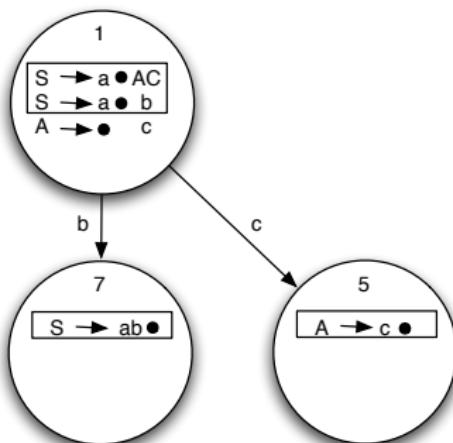
On est au début de l'analyse (état initial), il faut analyser $S\$$, ce qui correspond à analyser soit aAC soit ab

On a déjà analysé a et il faut encore analyser soit AC soit b . Pour analyser A , on doit analyser c

On a analysé aAC ; on a donc terminé l'analyse d'un S

On a analysé $S\$$; on a donc terminé l'analyse de S' (il n'y en a qu'un seul puisque $S' \rightarrow S\$$ a été rajouté à G) ; l'analyse est donc terminée avec succès.

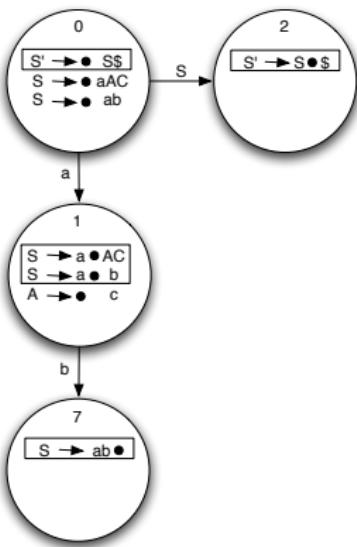
État et transition dans l'automate caractéristique :



État de l'analyse :

- Dans l'état 1 on peut (entre autres) devoir analyser un **b** ou un **c**
- On va donc faire un **Shift**
- Si l'input shifté est un **b** on va dans l'état 7
- Si l'input shifté est un **c** on va dans l'état 5
- Si l'input est tout autre caractère, c'est une **erreur** (le string analysé n'appartient pas au langage)

État et transition dans l'automate caractéristique :



État de l'analyse :

- Dans l'état 7 on a terminé une analyse de $ab \dots$
- ce qui correspond à l'analyse d'un S , débutée 2 états avant dans le chemin pris par l'analyse, càd dans l'état 0
- donc on termine l'analyse de S débutée dans l'état 0
- on va donc dans l'état 2

Ceci correspond à un **Reduce de $S \rightarrow ab$**

Note

On voit qu'un *Reduce* correspond à un *Shift* d'une variable dont on a terminé l'analyse

Parsing LR

Parsing LR

- L'analyse d'un string correspond à la détermination de handles
- On va utiliser une pile pour retenir les états du chemin pris dans l'automate caractéristique $LR(0)$
- On part de l'état 0 qui comprend $S' \rightarrow \bullet S\$$
- Il existe implicitement un état d'erreur \emptyset ; toute transition non prévue explicitement va dans cet état d'erreur

Construction des tables *Action* et *Successeur*

Les tables *Action* et *Successeur* synthétisent les informations à retenir de l'automate caractéristique $LR(0)$.

Construction de la table *Action*

La table *Action* donne pour **chaque état** de l'automate caractéristique $LR(0)$, l'action à réaliser parmi

- Shift
- Reduce i où i est le numéro de règle dans G' à utiliser pour le reduce
- Accept qui correspond au Reduce $S' \rightarrow S\$$
- Erreur si rien n'est prévu dans cet état

Construction de la table *Successeur*

La table *Successeur* représente la fonction de transition de l'automate caractéristique $LR(0)$. Elle est utilisée pour déterminer

- Lorsque l'**action est un Shift**
- Lorsque l'**action est un Reduce**

dans quel état on va (càd quel état on Push sur la pile, éventuellement (si Reduce) après avoir retiré $|\alpha|$ symboles)

Algorithmes de construction d'un analyseur $LR(0)$

Algorithme de fermeture d'un ensemble s de $LR(0)$ -items

Fermeture(s) :=

Fermeture $\Leftarrow s$

Répéter

fermeture' \Leftarrow Fermeture

si $[B \rightarrow \delta \bullet A\rho] \in$ Fermeture

$\forall A \rightarrow \gamma \in P$ (A -production de G')

Fermeture $\stackrel{\cup}{\Leftarrow} \{[A \rightarrow \bullet\gamma]\}$

f \forall

fsi

Jusqu'à ce que : Fermeture = fermeture'

Retourne : Fermeture

fproc

Algorithmes de construction d'un analyseur $LR(0)$

Algorithme de calcul de l'état suivant d'un état s pour un symbole X

Transition(s, X) :=

Transition \leftarrow Fermeture($\{[B \rightarrow \delta X \bullet \rho] \mid [B \rightarrow \delta \bullet X\rho] \in s\}$)

Retourne : Transition

fproc

Algorithme de construction de l'ensemble des états \mathcal{C} de l'automate caractéristique $LR(0)$

Note : Un état de \mathcal{C} est un ensemble de $LR(0)$ -items

Construction-automate-LR(0) :=

$$\mathcal{C} \Leftarrow \text{Fermeture}(\{[S' \rightarrow \bullet S\$]\}) \cup \{\emptyset\}$$

/* où \emptyset désigne l'état d'erreur */

Répéter

Soit $s \in \mathcal{C}$ non encore traité

$\forall X \in V \cup T$

$\text{Sucesseur}[s, X] \Leftarrow \text{Transition}(s, X)$

$\mathcal{C} \stackrel{\cup}{\Leftarrow} \text{Sucesseur}[s, X]$

fin

Jusqu'à ce que tous les s soient traités

/* Les entrées de Sucesseur vide référencent l'état d'erreur \emptyset */

fproc

Algorithmes de construction d'un analyseur $LR(0)$

Algorithme de construction de la table *Action*

Construction-table-Action() :=

$\forall s \in \mathcal{C} : Action[s] \leftarrow \emptyset$

$\forall s \in \mathcal{C}$

si $[A \rightarrow \alpha \bullet] \in s$: $Action[s] \stackrel{\cup}{\leftarrow} Reduce i$

/* où $A \rightarrow \alpha$ est la règle i */

si $[A \rightarrow \alpha \bullet a\beta] \in s$: $Action[s] \stackrel{\cup}{\leftarrow} Shift$

si $[S' \rightarrow S\$ \bullet] \in s$: $Action[s] \stackrel{\cup}{\leftarrow} Accept$

fproc

Langage accepté par l'automate caractéristique $LR(0)$

Théorème (La CFG G' est $LR(0)$ si et seulement si la table *Action* n'a au plus qu'une seule action dans chaque entrée)

Intuitivement, Action[s] résume l'état s

Théorème (Le langage accepté par l'automate caractéristique $LR(0)$ d'une CFG G' $LR(0)$, en supposant que tous les états soient accepteurs, est l'ensemble de ses préfixes viables)

Remarque

Les états de l'automate sont généralement renommés par un identificateur entier naturel (0 pour l'état initial)

Exemple (tables *Action* et *Successeur* pour G' du slide 381)

	a	b	c	d	A	C	S	\$
0	S						2	
1	S		7	5		3		
2	S							8
3	S				6		4	
4	R1							
5	R3							
6	R4							
7	R2							
8	A							

Action

Successeur

Rappel : Implicitement les entrées de la table *Successeur* non remplies renvoient vers un état d'erreur

Tables d'une CFG $LR(0)$

Notes

- Peu de grammaires sont $LR(0)$ (mais plus que de grammaires $LL(0)$)
- Si G' n'est pas $LR(0)$, on doit considérer une prévision et construire un parser $LR(k)$ avec $k \geq 1$.
- Évidemment ici aussi, plus k est grand, plus le parser sera compliqué
- On essaye de se limiter à $k = 1$.

Analyseurs prédictifs $LR(k)$

Remarques sur l'algorithme

- On donne ici l'**algorithme général de parsing $LR(k)$** pour $k \geq 0$
- Si $k = 0$, la table *Action* est un vecteur
- Si $k \geq 1$, la table *Action* a une prévision u de taille $\leq k$ comme second paramètre
- La construction des tables *Action* et *Successeur* pour $k > 0$ sera donné dans les sections suivantes.

Algorithme général du fonctionnement d'un analyseur $LR(k)$

On suppose que les tables *Action* et *Successeur* sont construites

Parser-LR-k() :=

Initialement : *Push(0)* /* État initial */

Boucle

$s \leftarrow Top()$

si *Action*[s, u] = *Shift* : *Shift(s)* /* $|u| \leq k$ et *Action* est un */

si *Action*[s, u] = *Reduce i* : *Reduce(i)* /* vecteur (pas de paramètre */

si *Action*[s, u] = \emptyset : *Erreur()* /* u si le parser est $LR(0)$ */

si *Action*[s, u] = *Accept* : *Accept()*

Fboucle

Fproc

- *Shift(s)* := $X \leftarrow$ input suivant ; *Push(Successeur[s, X])*^a Fproc
- *Reduce(i)* := (la règle $i \equiv A \rightarrow \alpha$) Pour $j = 1$ à $|\alpha|$ *Pop()* fpour ;
 $s \leftarrow Top(); Push(Successeur[s, A]); Fproc$
- *Erreur()* := Informe d'une erreur dans l'analyse ; Fproc
- *Accept()* := Informe du succès de l'analyse ; Fproc

^apeut être l'état d'erreur

Analyse LR(0)

Exemple (Analyse du string $acd\$$ pour G')

qui a les règles suivantes :

$$\begin{array}{lcl} S' & \rightarrow & S\$ \quad (0) \\ S & \rightarrow & aAC \quad (1) \\ S & \rightarrow & ab \quad (2) \\ A & \rightarrow & c \quad (3) \\ C & \rightarrow & d \quad (4) \end{array}$$

Sur la pile	Input restant	Act	Output
$\vdash 0$	$acd\$$	S	ϵ
$\vdash 01$	$cd\$$	S	ϵ
$\vdash 015$	$d\$$	R3	ϵ
$\vdash 013$	$d\$$	S	3
$\vdash 0136$	$\$$	R4	3
$\vdash 0134$	$\$$	R1	43
$\vdash 02$	$\$$	S	143
$\vdash 028$	ϵ	A	143

Analyse LR(0)

Correspondance entre l'analyse telle que présentée au début du chapitre et l'analyse LR

Bien que cela soit inutile, on peut (pour aider à la compréhension du lecteur) explicitement *Pusher* sur la pile, les symboles *Shiftés* (terminaux ou variables lors de Reduce)

Sur la pile	Input restant	Act	Output
$\vdash 0$	$acd\$$	S	ϵ
$\vdash 0a1$	$cd\$$	S	ϵ
$\vdash 0a1c5$	$d\$$	R3	ϵ
$\vdash 0a1A3$	$d\$$	S	3
$\vdash 0a1A3d6$	$\$$	R4	3
$\vdash 0a1A3C4$	$\$$	R1	43
$\vdash 0S2$	$\$$	S	143
$\vdash 0S2\$8$	ϵ	A	143

On a que :

- La pile où l'on fait abstraction des numéros d'états, concaténée avec l'input restant est à tout moment une forme phrasée d'une dérivation droite.
- La pile, où l'on fait abstraction des numéros d'état, est toujours un préfixe viable.

Exemple 2 d'automate caractéristique $LR(0)$

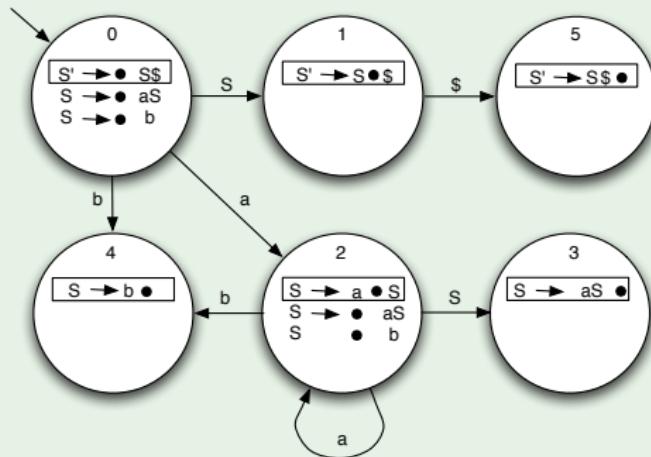
Exemple (De construction d'automate caractéristique $LR(0)$)

Pour G'_2 qui a les règles suivantes :

$$\begin{array}{ll} S' \rightarrow S\$ & (0) \\ S \rightarrow aS & (1) \\ S \rightarrow b & (2) \end{array}$$

L'automate caractéristique $LR(0)$ et les tables *Action* et *Successeur* suivants :

Exemple (Automate caractéristique $LR(0)$ et tables *Action* et *Successeur*)



	a	b	S	\$
0	S			
1	S			5
2				
3	R1			
4	R2			
5	A			

Action

	a	b	S	\$
2	4	1		
3				
4	4	3		
5				

Successeur

Analyse LR(0)

Exemple (Analyse du string $aab\$\epsilon$ pour G'_2)

qui a les règles suivantes :

$$\begin{array}{ll} S' \rightarrow S\$ & (0) \\ S \rightarrow aS & (1) \\ S \rightarrow b & (2) \end{array}$$

Sur la pile	Input restant	Act	Output
$\vdash 0$	$aab\$$	S	ϵ
$\vdash 02$	$ab\$$	S	ϵ
$\vdash 022$	$b\$$	S	ϵ
$\vdash 0224$	$\$$	R2	ϵ
$\vdash 0223$	$\$$	R1	2
$\vdash 023$	$\$$	R1	12
$\vdash 01$	$\$$	S	112
$\vdash 015$	ϵ	A	112

Note : la règle 0 n'est pas donnée en output

Exemple 3 d'automate caractéristique $LR(0)$

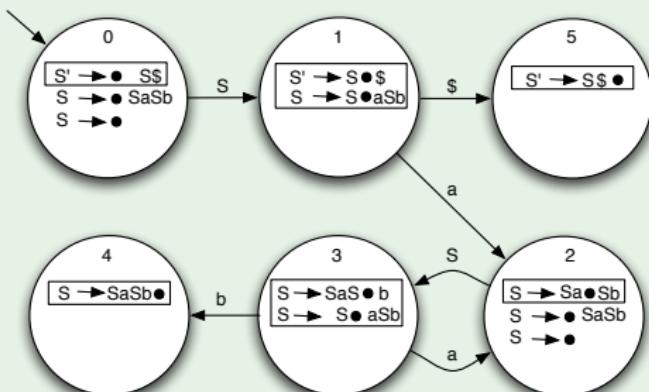
Exemple (De construction d'automate caractéristique $LR(0)$)

Pour G'_3 qui a les règles suivantes :

$$\begin{array}{ll} S' \rightarrow S\$ & (0) \\ S \rightarrow SaSb & (1) \\ S \rightarrow \epsilon & (2) \end{array}$$

L'automate caractéristique $LR(0)$ et les tables *Action* et *Successeur* suivants :

Exemple (Automate caractéristique $LR(0)$ et tables *Action* et *Successeur*)



	a	b	S	\$
0	R2		1	
1	S			5
2	R2			
3	S			
4	R1			
5	A			

Action

	a	b	S	\$
a			1	
b				5
S				3
\$	2	4		

Successeur

Analyse LR(0)

Exemple (Analyse du string $aabb\$\epsilon$ pour G'_3)

qui a les règles suivantes :

$$\begin{array}{ll} S' \rightarrow S\$ & (0) \\ S \rightarrow SaSb & (1) \\ S \rightarrow \epsilon & (2) \end{array}$$

Sur la pile	Input restant	Act	Output
$\vdash 0$	$aabb\$$	R2	ϵ
$\vdash 01$	$aabb\$$	S	2
$\vdash 012$	$abb\$$	R2	2
$\vdash 0123$	$abb\$$	S	22
$\vdash 01232$	$bb\$$	R2	22
$\vdash 012323$	$bb\$$	S	222
$\vdash 0123234$	$b\$$	R1	222
$\vdash 0123$	$b\$$	S	1222
$\vdash 01234$	$\$$	R1	1222
$\vdash 01$	$\$$	S	11222
$\vdash 015$	ϵ	A	11222

Exemple (la grammaire G'_4 n'est pas $LR(0)$)

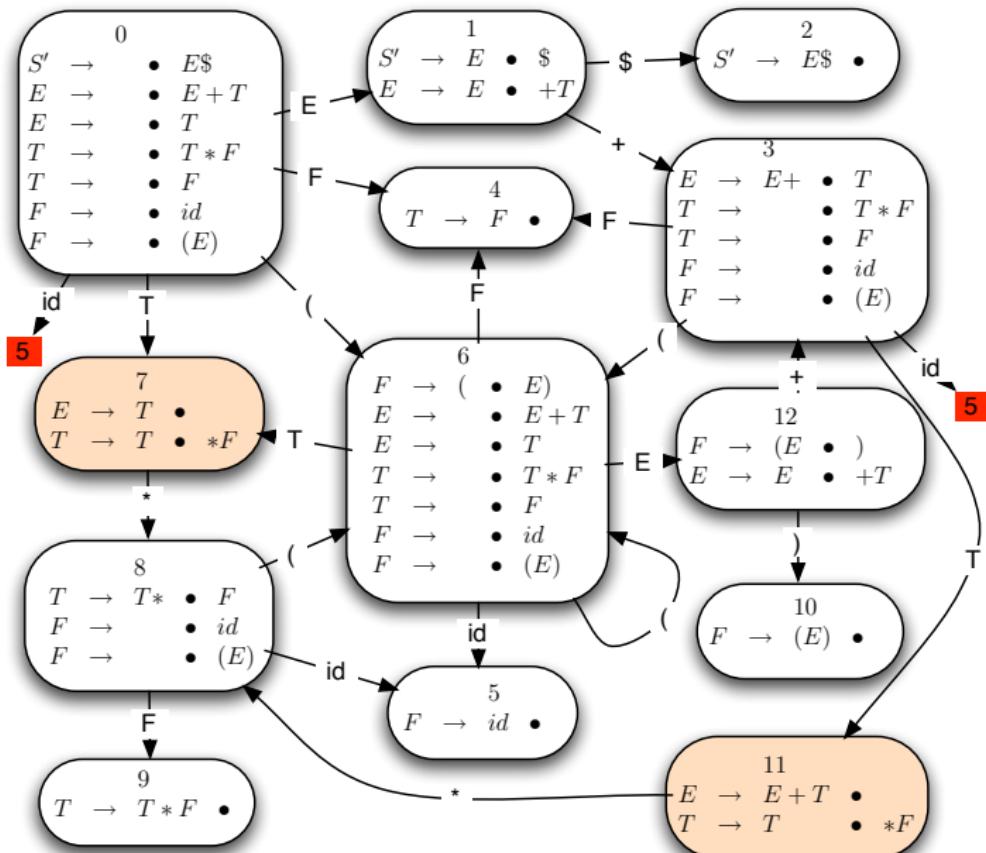
$G'_4 = \langle \{S', E, T, F\}, \{+, *, id, (,), \$\}, P_4, S' \rangle$ avec les règles P_4 suivantes :

$$\begin{array}{lll}
 S' & \rightarrow & E\$ \quad (0) \\
 E & \rightarrow & E + T \quad (1) \\
 E & \rightarrow & T \quad (2) \\
 T & \rightarrow & T * F \quad (3) \\
 T & \rightarrow & F \quad (4) \\
 F & \rightarrow & id \quad (5) \\
 F & \rightarrow & (E) \quad (6)
 \end{array}$$

n'est pas $LR(0)$ comme le montre la construction de son automate caractéristique $LR(0)$ où

- l'état 7
- l'état 11

induisent à la fois une action **Shift** et **Reduce** (conflit Shift/Reduce).
Nous verrons que G'_4 est une grammaire $LR(1)$.



Plan

- 1 Principe de parsing ascendant
- 2 CFG LR(k)
- 3 Parser LR(0)
- 4 Parser LR(1)**
- 5 Parser SLR(1)
- 6 Parser LALR(1)
- 7 Classes LL vs LR
- 8 L'outil Yacc (Bison)

- La prévision sera utile quand il est question de faire un **Reduce**
- Dans ce cas, il faut déterminer les prévisions (de longueur jusqu'à k , ici $k = 1$) possibles.
- Cela revient à calculer les **Follow locaux** aux règles (items)

Calcul des $LR(1)$ -items

- On aura des $LR(1)$ -items de la forme $[A \rightarrow \alpha_1 \bullet \alpha_2, u]$ où u est un follow local à $A \rightarrow \alpha$ ($\alpha = \alpha_1\alpha_2$)
 - Le $LR(1)$ -item “initial” est $[S' \rightarrow \bullet S\$, \epsilon]$
 - L’opération de fermeture augmente un ensemble s de $LR(1)$ -items comme suit :
 - Ayant $[B \rightarrow \delta \bullet A\rho, \ell] \in s$
 - $A \rightarrow \gamma \in P$
- $\} \Rightarrow \forall u \in First^k(\rho\ell) : s \stackrel{\cup}{\leftarrow} [A \rightarrow \bullet\gamma, u]$

Remarque

Dans ce qui suit nous donnons les algorithmes pour $LR(k)$ même si les exemples se limitent à $k = 1$.

Algorithmes de construction d'un analyseur $LR(k)$

Algorithme de fermeture d'un ensemble s de $LR(k)$ -items

Fermeture(s) :=

Fermeture $\Leftarrow s$

Répéter

fermeture' \Leftarrow Fermeture

si $[B \rightarrow S \bullet A\rho, \ell] \in$ Fermeture

$\forall A \rightarrow \gamma \in P$ (A-production de G')

$\forall u \in First^k(\rho\ell)$: Fermeture $\stackrel{\cup}{\Leftarrow} [A \rightarrow \bullet\gamma, u]$

f \forall

fsi

Jusqu'à ce que : Fermeture = fermeture'

Retourne : Fermeture

fproc

Algorithmes de construction d'un analyseur $LR(k)$

Algorithme de calcul de l'état suivant d'un état s pour un symbole X

Transition(s, X) :=

Transition \Leftarrow Fermeture($\{[B \rightarrow \delta X \bullet \rho, u] \mid [B \rightarrow \delta \bullet X\rho, u] \in s\}$)

Retourne : Transition

fproc

Algorithme de construction de l'ensemble des états \mathcal{C} de l'automate caractéristique $LR(k)$

Construction-automate-LR(0) :=

$\mathcal{C} \Leftarrow \text{Fermeture}(\{[S' \rightarrow \bullet S\$, \epsilon]\}) \cup \emptyset$

/* où \emptyset est l'état d'erreur */

Répéter

Soit $s \in \mathcal{C}$ non encore traité

$\forall X \in V \cup T$

$\text{Successeur}[s, X] \Leftarrow \text{Transition}(s, X)$

$\mathcal{C} \stackrel{\cup}{\Leftarrow} \text{Successeur}[s, X]$

f \forall

Jusqu'à ce que tous les s soient traités

/* Les entrées vides de Successeur réfèrent à l'état d'erreur \emptyset */

fproc

Algorithmes de construction d'un analyseur $LR(k)$

Algorithme de construction de la table *Action*

Construction-table-Action :=

$\forall s \in \mathcal{C}, u \in T^{\leq k} : Action[s, u] \Leftarrow \emptyset$

$\forall s \in \mathcal{C}$

si $[A \rightarrow \alpha \bullet, u] \in s : Action[s, u] \stackrel{\cup}{\Leftarrow} Reduce\ i$

/* où $A \rightarrow \alpha$ est la règle i */

si $[A \rightarrow \alpha \bullet a\beta, u] \in s \wedge u \in First^k(a\beta y) : Action[s, u] \stackrel{\cup}{\Leftarrow} Shift$

si $[S' \rightarrow S\$ \bullet, \epsilon] \in s : Action[s, \epsilon] \stackrel{\cup}{\Leftarrow} Accept$

fproc

Construction du parser $LR(1)$ pour G'_4

Exemple (Construction du parser $LR(1)$ pour G'_4)

$G'_4 = \langle \{S', E, T, F\}, \{+, *, id, (,), \$\}, P_4, S' \rangle$ avec les règles P_4 suivantes :

$$\begin{array}{lll} S' & \rightarrow & E\$ \quad (0) \\ E & \rightarrow & E + T \quad (1) \\ E & \rightarrow & T \quad (2) \\ T & \rightarrow & T * F \quad (3) \\ T & \rightarrow & F \quad (4) \\ F & \rightarrow & id \quad (5) \\ F & \rightarrow & (E) \quad (6) \end{array}$$

- Automate caractéristique $LR(1)$
 - Tables
- } Voir slides suivants

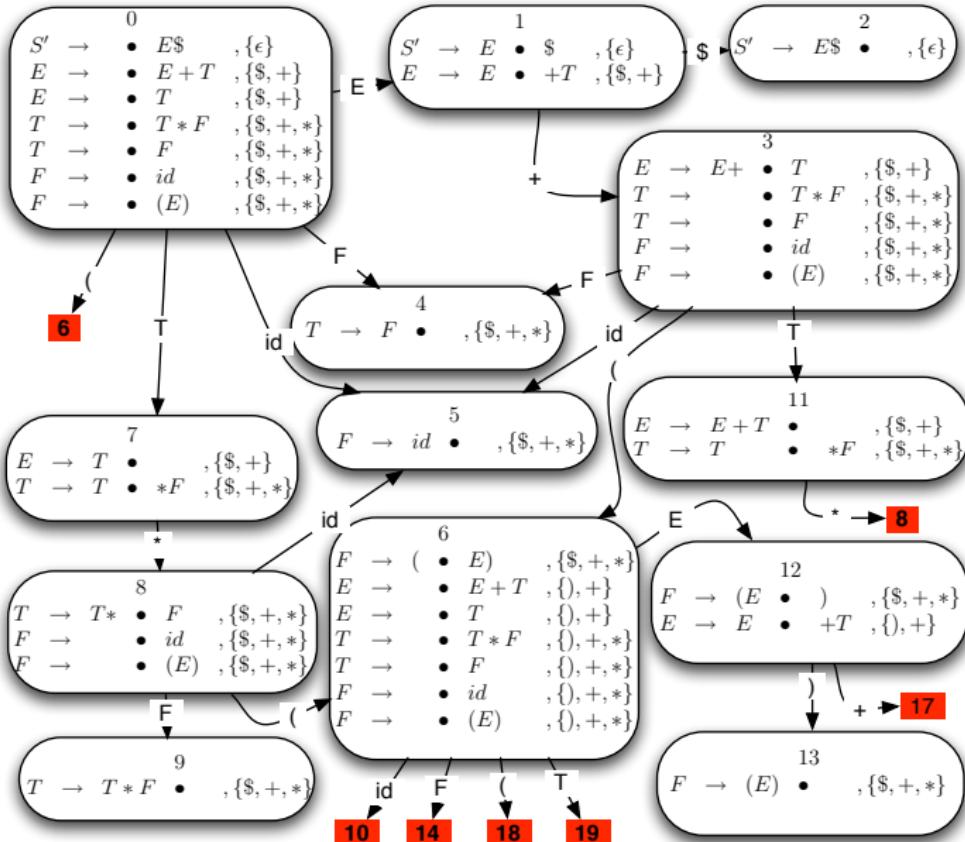
Notation : on écrit $[A \rightarrow \alpha_1 \bullet \alpha_2, \{u_1, u_2, \dots, u_n\}]$ à la place de

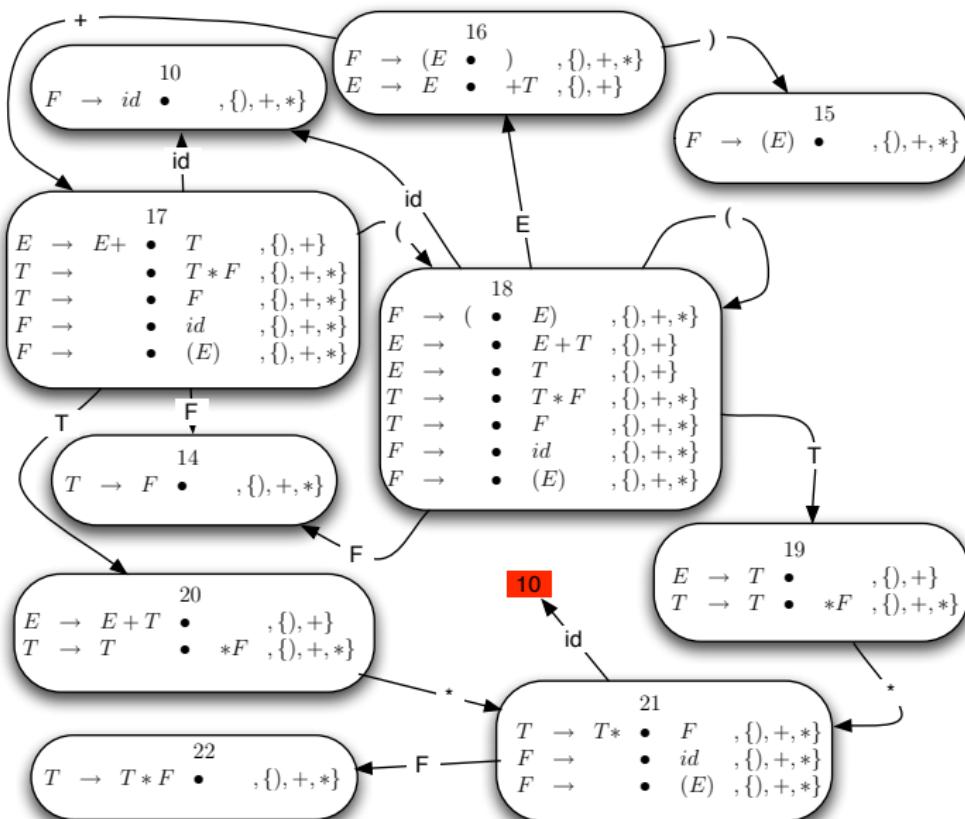
$[A \rightarrow \alpha_1 \bullet \alpha_2, u_1]$

$[A \rightarrow \alpha_1 \bullet \alpha_2, u_2]$

...

$[A \rightarrow \alpha_1 \bullet \alpha_2, u_n]$





Tables $LR(1)$

Action

État	+	*	<i>id</i>	()	\$	ϵ
0			<i>S</i>	<i>S</i>			
1	<i>S</i>					<i>S</i>	
2							<i>A</i>
3			<i>S</i>	<i>S</i>			
4	<i>R4</i>	<i>R4</i>				<i>R4</i>	
5	<i>R5</i>	<i>R5</i>				<i>R5</i>	
6			<i>S</i>	<i>S</i>			
7	<i>R2</i>	<i>S</i>				<i>R2</i>	
8			<i>S</i>	<i>S</i>			
9	<i>R3</i>	<i>R3</i>				<i>R3</i>	
10	<i>R5</i>	<i>R5</i>				<i>R5</i>	
11	<i>R1</i>	<i>S</i>				<i>R1</i>	
12	<i>S</i>					<i>S</i>	
13	<i>R6</i>	<i>R6</i>				<i>R6</i>	
14	<i>R4</i>	<i>R4</i>				<i>R4</i>	
15	<i>R6</i>	<i>R6</i>				<i>R6</i>	
16	<i>S</i>					<i>S</i>	
17			<i>S</i>	<i>S</i>			
18			<i>S</i>	<i>S</i>			
19	<i>R2</i>	<i>S</i>				<i>R2</i>	
20	<i>R1</i>	<i>S</i>				<i>R1</i>	
21			<i>S</i>	<i>S</i>			
22	<i>R3</i>	<i>R3</i>				<i>R3</i>	

Successeur

Critique de $LR(1)$ ($LR(k)$)

Critique de $LR(1)$ ($LR(k)$)

- On constate que très vite, l'analyseur (l'automate caractéristique et les tables) devient très gros.
- De ce fait, on a essayé de trouver des alternatives, plus puissantes que $LR(0)$ mais plus compactes que $LR(1)$
⇒ Les parsers $SLR(1)$ ($SLR(k)$) et $LALR(1)$ ($LALR(k)$)

Plan

- 1 Principe de parsing ascendant
- 2 CFG LR(k)
- 3 Parser LR(0)
- 4 Parser LR(1)
- 5 **Parser SLR(1)**
- 6 Parser LALR(1)
- 7 Classes LL vs LR
- 8 L'outil Yacc (Bison)

Principe de la construction des parsers SLR(1)

Le principe est très simple

- ① On construit l'automate caractéristique $LR(0)$
- ② Pour construire la table *Action*, on utilise les $LR(0)$ -items et les **follow globaux** des variables pour déterminer pour quelles prévisions faire des Reduce :

Plus précisément

- **Action**[$s, a]$ contiendra l'action **Shift** si l'état s contient $B \rightarrow \delta \bullet a\gamma$ pour une certaine variable B et des strings γ et δ ,
- **Action**[$s, a]$ contiendra l'action **Reduce i** si
 - la règle i est $A \rightarrow \alpha$
 - s contient $A \rightarrow \alpha \bullet$
 - $a \in Follow(A)$

Pour SLR(k)

Les mêmes idées sont aisément étendues pour k symboles de prévision : on utilise :

- $First^k(a\gamma Follow^k(B))$
- $Follow^k(A)$.

} Voir slide suivant

Algorithmes de construction d'un analyseur $SLR(k)$

Algorithme de construction de la table Action

Construction-table-Action() :=

$$\forall s \in \mathcal{C}, u \in T^{\leq k} : \text{Action}[s, u] \Leftarrow \emptyset$$

$\forall s \in \mathcal{C}$

si $[A \rightarrow \alpha \bullet] \in s \wedge u \in \text{Follow}^k(A) : \text{Action}[s, u] \Leftarrow \text{Reduce } i$

/* où $A \rightarrow \alpha$ est la règle i */

si $[A \rightarrow \alpha \bullet a\beta] \in s \wedge u \in \text{First}^k(a\beta\text{Follow}^k(A)) :$

$\text{Action}[s, u] \Leftarrow \text{Shift}$

si $[S' \rightarrow S\$ \bullet] \in s : \text{Action}[s, \epsilon] \Leftarrow \text{Accept}$

fproc

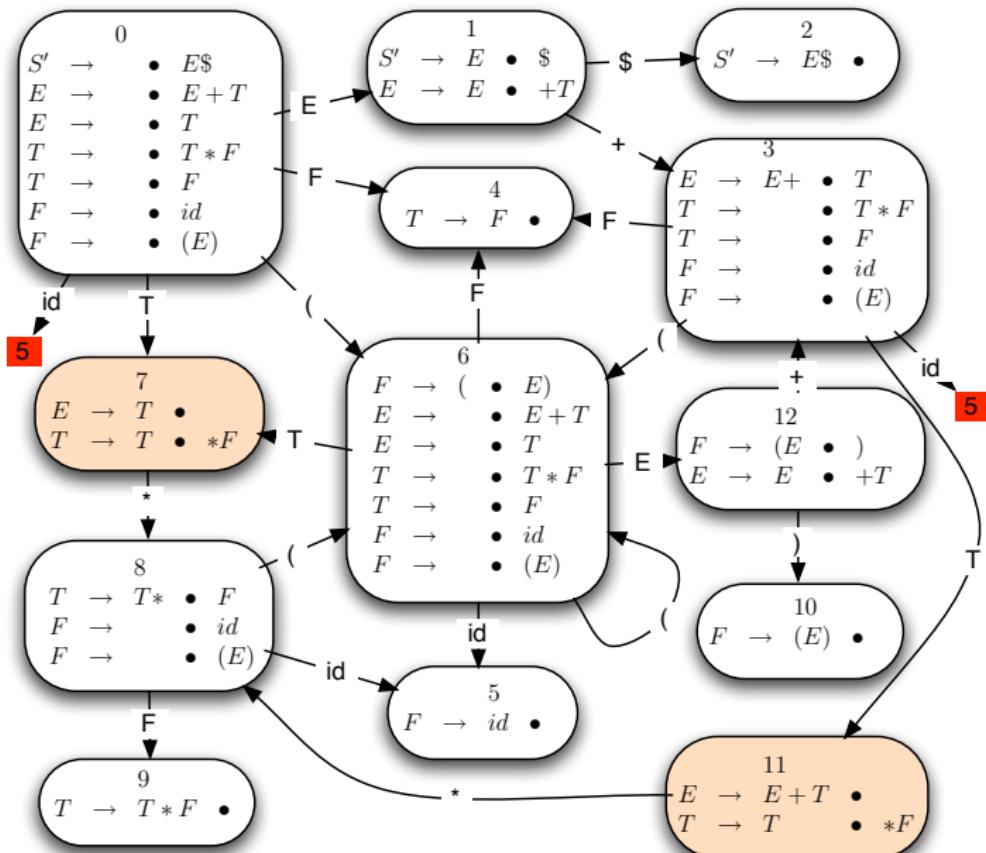
Exemple (Construction du parser $SLR(1)$ pour $G'4$)

$G'_4 = \langle \{S', E, T, F\}, \{+, *, id, (,), \$\}, P_4, S' \rangle$ avec les règles P_4 suivantes :

$$\begin{array}{lll} S' & \rightarrow & E\$ \quad (0) \\ E & \rightarrow & E + T \quad (1) \\ E & \rightarrow & T \quad (2) \\ T & \rightarrow & T * F \quad (3) \\ T & \rightarrow & F \quad (4) \\ F & \rightarrow & id \quad (5) \\ F & \rightarrow & (E) \quad (6) \end{array}$$

On a

- $Follow(E) = \{+, (), \$\}$
 - $Follow(T) = \{+, *, (), \$\}$
 - $Follow(F) = \{+, *, (), \$\}$
 - Automate caractéristique $LR(0)$
 - Tables (avec prévisions “globales”)
- } Voir slides suivants



Tables $SLR(1)$

Action

État	+	*	id	()	\$	ϵ	
0			S	S				
1	S					S		
2								A
3			S	S				
4	$R4$	$R4$			$R4$	$R4$		
5	$R5$	$R5$			$R5$	$R5$		
6			S	S				
7	$R2$	S			$R2$	$R2$		
8			S	S				
9	$R3$	$R3$			$R3$	$R3$		
10	$R6$	$R6$			$R6$	$R6$		
11	$R1$	S			$R1$	$R1$		
12	S				S			

Successeur

État	+	*	id	()	\$	E	T	F
0				5	6		1	7	4
1		3				2			
2									
3				5	6			11	4
4									
5									
6				5	6		12	7	4
7			8						
8				5	6				9
9									
10									
11			8						
12		3				10			

Notons que la table Successeur d'un analyseur $SLR(k)$ est toujours égale à celle d'un analyseur $LR(0)$

Critique de la méthode $SLR(1)$

Limitations de la méthode $SLR(1)$

La méthode $SLR(1)$ est simple et plus puissante que la méthode $LR(0)$; mais, on rencontre facilement des exemples où il reste des conflits

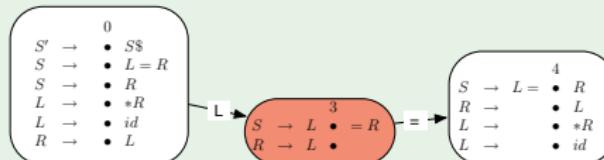
Exemple (Grammaire qui n'est ni $LR(0)$ ni $SLR(1)$)

$G'_5 = \langle \{S', E, T, F\}, \{+, *, id, (,), \$\}, P_5, S' \rangle$ avec les règles P_5 et les Follow suivants :

$$\begin{array}{lcl} S' & \rightarrow & S\$ \quad (0) \\ S & \rightarrow & L = R \quad (1) \\ S & \rightarrow & R \quad (2) \\ L & \rightarrow & *R \quad (3) \\ L & \rightarrow & id \quad (4) \\ R & \rightarrow & L \quad (5) \end{array}$$

- $Follow(S) = \{\$\}$
- $Follow(L) = \{=, \$\}$
- $Follow(R) = \{=, \$\}$

Donne une partie d'automate caractéristique suivante :



Où on voit le conflit : $Action[3, =] = \{\text{Shift, Reduce } 5\}$

Plan

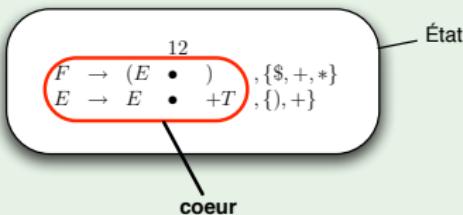
- 1 Principe de parsing ascendant
- 2 CFG LR(k)
- 3 Parser LR(0)
- 4 Parser LR(1)
- 5 Parser SLR(1)
- 6 Parser LALR(1)**
- 7 Classes LL vs LR
- 8 L'outil Yacc (Bison)

Principe de la construction des parsers $LALR(k)$

Définition (Cœur d'un état d'un automate caractéristique $LR(k)$)

C'est l'ensemble des $LR(k)$ -items de l'état desquels on a supprimé les prévisions

Exemple (de cœur d'un état s)



Principe de la construction des parsers $LALR(k)$

Le principe est très simple

- ① On construit l'automate caractéristique $LR(k)$
- ② On fusionne les états ayant le même cœur en prenant l'union de leurs $LR(k)$ -items
- ③ La construction des tables reprend ensuite l'algorithme pour $LR(k)$

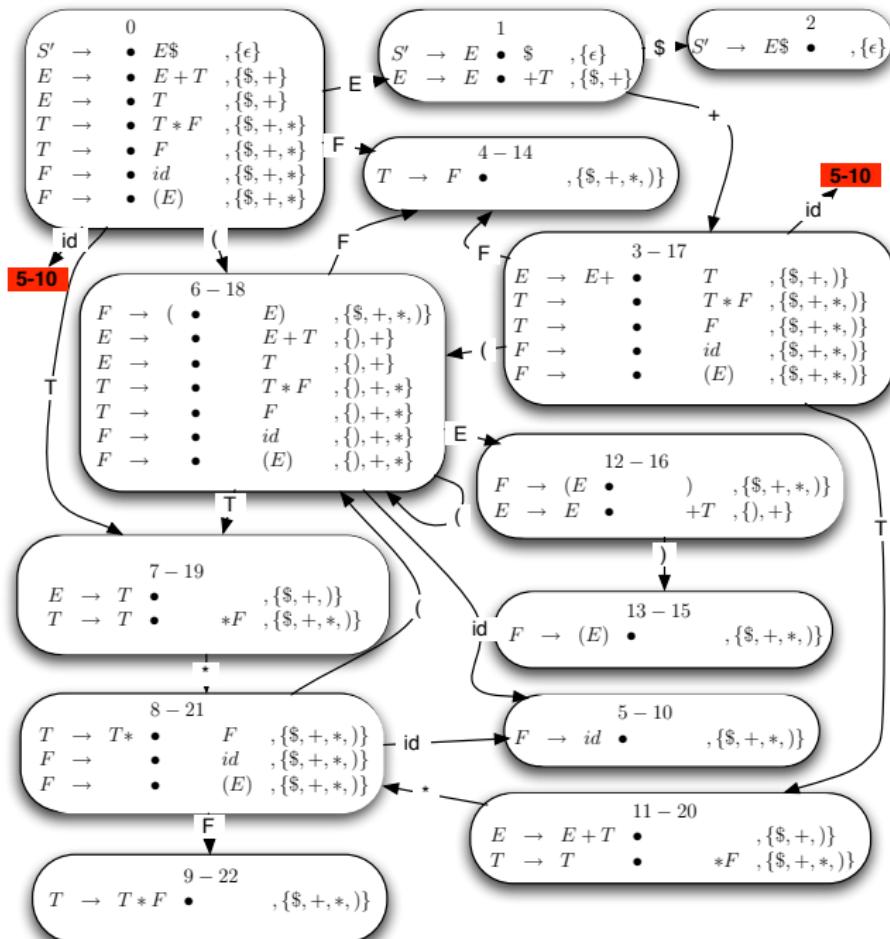
Exemple (Construction du parser *LALR*(1) pour $G'4$)

$G'_4 = \langle \{S', E, T, F\}, \{+, *, id, (,), \$\}, P_4, S' \rangle$ avec les règles P_4 suivantes :

$$\begin{array}{lll} S' & \rightarrow & E\$ \quad (0) \\ E & \rightarrow & E + T \quad (1) \\ E & \rightarrow & T \quad (2) \\ T & \rightarrow & T * F \quad (3) \\ T & \rightarrow & F \quad (4) \\ F & \rightarrow & id \quad (5) \\ F & \rightarrow & (E) \quad (6) \end{array}$$

Après construction de l'automate caractéristique $LR(1)$ les états suivants ont le même cœur :

- 3 et 17
 - 4 et 14
 - 5 et 10
 - 6 et 18
 - 7 et 19
 - Automate caractéristique $LALR(1)$
 - Tables
 - 8 et 21
 - 9 et 22
 - 11 et 20
 - 12 et 16
 - 13 et 15
- } Voir slides suivants



Tables *LALR(1)*

Action

État	+	*	<i>id</i>	()	\$	ϵ
0			<i>S</i>	<i>S</i>			
1	<i>S</i>					<i>S</i>	
2							<i>A</i>
3 – 17			<i>S</i>	<i>S</i>			
4 – 14	<i>R4</i>	<i>R4</i>			<i>R4</i>	<i>R4</i>	
5 – 10	<i>R5</i>	<i>R5</i>			<i>R5</i>	<i>R5</i>	
6 – 18			<i>S</i>	<i>S</i>			
7 – 19	<i>R2</i>	<i>S</i>			<i>R2</i>	<i>R2</i>	
8 – 21			<i>S</i>	<i>S</i>			
9 – 22	<i>R3</i>	<i>R3</i>			<i>R3</i>	<i>R3</i>	
13 – 15	<i>R6</i>	<i>R6</i>			<i>R6</i>	<i>R6</i>	
11 – 20	<i>R1</i>	<i>S</i>			<i>R1</i>	<i>R1</i>	
12 – 16	<i>S</i>				<i>S</i>		

Tables *LALR(1)*

Successeur

État	+	*	id	()	\$	E	T	F
0			5-10	6-18			1	7-19	4-14
1	3-17					2			
2									
3-17			5-10	6-18				11-20	4-14
4-14									
5-10									
6-18			5-10	6-18			12-16	7-19	4-14
7-19		8-21							
8-21			5-10	6-18					9-22
9-22									
13-15									
11-20		8-21							
12-16	3-17				13-15				

Caractéristiques de la méthode *LALR*(*k*)

Théorème (La fusion des états de l'automate caractéristique *LR*(*k*) est consistante)

C'est-à-dire si 2 états s_1 et s_2 doivent être fusionnés dans l'automate *LALR*(*k*) alors $\forall X : \text{Transition}[s_1, X]$ et $\text{Transition}[s_2, X]$ doivent également être fusionnés.

En effet, **Transition(s, X)** ne dépend que des coeurs des *LR*(*k*)-items et pas des prévisions.

Caractéristique de la méthode $LALR(k)$

- Pour toute CFG G' , si on fait abstraction des prévisions, les automates caractéristiques $LR(0)$ et $LALR(k)$ sont les mêmes.
- Il est possible de construire l'automate caractéristique $LALR(1)$ directement à partir de l'automate caractéristique $LR(0)$ sur lequel on calcule directement les prévisions : cette méthode n'est pas vue dans ce cours.
- Pour l'exemple précédent, les tables *Actions* des analyseurs $LALR(1)$ et $SLR(1)$ sont les mêmes (au nom des états près) ; ce n'est généralement pas le cas.
- Toute grammaire $SLR(k)$ est $LALR(k)$ mais l'inverse n'est pas toujours vraie.
- Par exemple, la grammaire G'_5 du slide 424 n'est pas $SLR(1)$ mais est $LR(1)$ et $LALR(1)$.

Caractéristiques de la méthode *LALR(1)*

Il se peut qu'une grammaire *LR(1)* ne soit pas *LALR(1)*. En effet :

La fusion des états peut rajouter des conflits Reduce / Reduce

Pour la grammaire dont les règles sont :

$$\begin{array}{lll} S' \rightarrow S\$ & (0) \\ S \rightarrow aAd & (1) \\ S \rightarrow bBd & (2) \\ S \rightarrow aBe & (3) \end{array} \qquad \begin{array}{lll} S \rightarrow bAe & (4) \\ A \rightarrow c & (5) \\ B \rightarrow c & (6) \end{array}$$

les états suivants sont générés dans l'automate caractéristique *LR(1)*

- $\{[A \rightarrow c\bullet, d], [B \rightarrow c\bullet, e]\}$
- $\{[A \rightarrow c\bullet, e], [B \rightarrow c\bullet, d]\}$

dont la fusion engendre un conflit Reduce 5/ Reduce 6

Caractéristiques de la méthode *LALR(1)*

La fusion des états ne peut pas rajouter des conflits Shift / Reduce

En effet, si on a dans un même état de l'automate caractéristique *LALR(1)*

- $[A \rightarrow \alpha \bullet, a]$ et
- $[B \rightarrow \beta \bullet a\gamma, b]$

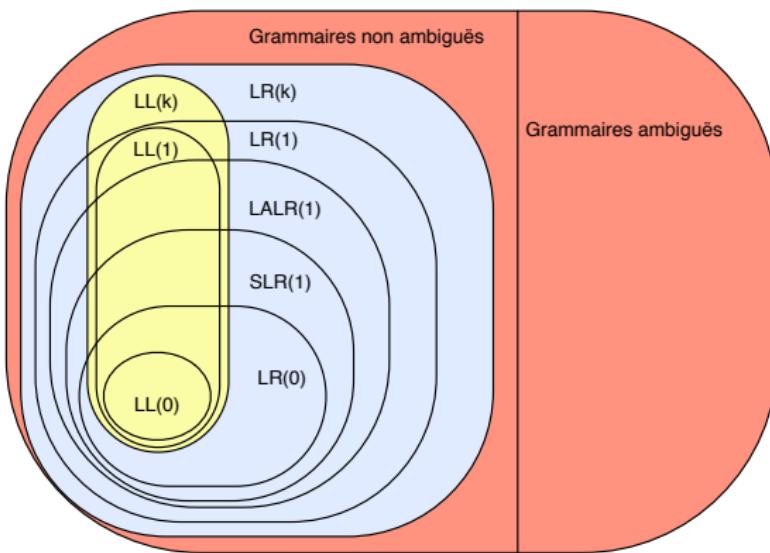
alors dans l'état de l'automate caractéristique *LR(1)* correspondant, contenant $[A \rightarrow \alpha \bullet, a]$

- on aurait forcément $[B \rightarrow \beta \bullet a\gamma, c]$ pour un certain c et
- le conflit Shift / Reduce existerait déjà au niveau *LR(1)*

Plan

- 1 Principe de parsing ascendant
- 2 CFG LR(k)
- 3 Parser LR(0)
- 4 Parser LR(1)
- 5 Parser SLR(1)
- 6 Parser LALR(1)
- 7 Classes LL vs LR**
- 8 L'outil Yacc (Bison)

Inclusion des classes de grammaires



Notes

- En pratique, la plupart des grammaires que l'on veut compiler sont $LALR(1)$
- On peut démontrer que les 3 classes des langages $LR(k)$ (càd acceptés par une grammaire $LR(k)$), $LR(1)$ et des langages acceptés par un DPDA (automate à pile déterministe) sont les mêmes.

Plan

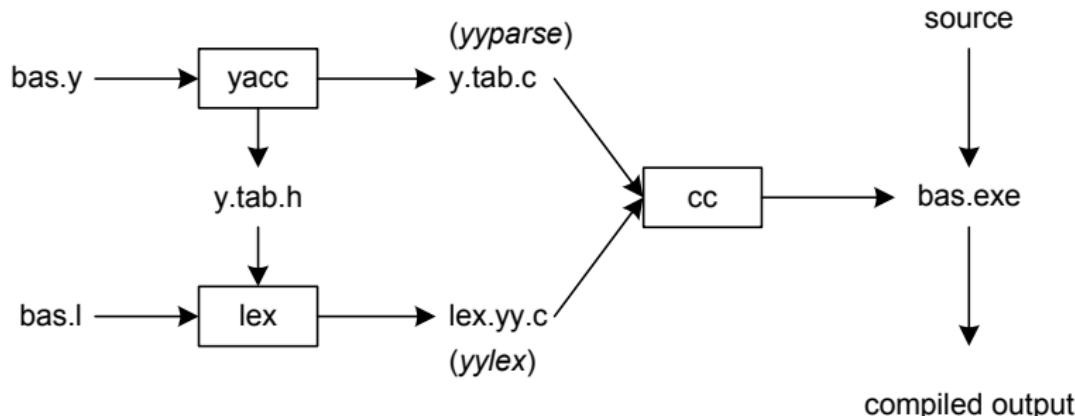
- 1 Principe de parsing ascendant
- 2 CFG LR(k)
- 3 Parser LR(0)
- 4 Parser LR(1)
- 5 Parser SLR(1)
- 6 Parser LALR(1)
- 7 Classes LL vs LR
- 8 L'outil Yacc (Bison)

Yacc = Yet Another Compiler Compiler (\equiv Bison en système GNU)

Que fait Yacc ?

- Yacc a été conçu comme générateur de parsers *LALR(1)*, et de façon plus générale, de parties de compilateur.
- En collaboration avec l'outil Lex, Yacc peut construire une grande partie ou même la totalité d'un compilateur.
- Yacc est un outil puissant pour créer des programmes traitant des langages dont la grammaire context-free est donnée.

Procédure générale pour l'utilisation de Lex (Flex) et Yacc (Bison)



Compilation :

```
yacc -d bas.y          # crée y.tab.h et y.tab.c
lex bas.l              # crée lex.yy.c
cc lex.yy.c y.tab.c -ll -o bas.exe # compile et fait l'édition
                                    # crée bas.exe
```

Spécification Yacc

déclarations

%%

productions

%%

code additionnel

Le parser résultant (`yyparse()`) cherche à reconnaître la phrase compatible avec la grammaire.

Lors de l'analyse d'un parser généré avec Yacc, des *actions sémantiques* sous forme de code C peuvent être exécutées et des *attributs* peuvent être calculés (voir exemple et chapitre suivant).

Exemple Lex et Yacc : évaluateur d'expressions

Exemple (Partie Lex de l'évaluateur d'expressions)

```
/*      File ex3.l      */
%{
#include "y.tab.h"
#define yywrap() 1
extern int yylval;
%}
integer  [0-9] +
separator [\t]
nl        \n
%%
{integer} { sscanf(yytext, "%d", &yylval);
            return(INTEGER);
        }
[-+*/()] { return yytext[0]; }
quit      { return 0; }
{nl}       { return '\n'; }
{separator}; ;
.          { return yytext[0]; }
```

Exemple Lex et Yacc : évaluateur d'expressions

Exemple (partie Yacc de l'évaluateur d'expressions (1))

```
/*      File ex3.y      */
%{
#include <stdio.h>
%}

%token INTEGER

%%
```

Exemple Lex et Yacc : évaluateur d'expressions

Exemple (partie Yacc de l'évaluateur d'expressions (2))

```
lines:      /*empty*/
          | lines line
          ;
line:      '\n'
          | exp '\n'
                  {printf(" = %d\n", $1);}

exp:       exp '+' term           {$$ = $1 + $3;}
          | exp '-' term           {$$ = $1 - $3;}
          | term
          ;
term:      term '*' fact         {$$ = $1 * $3;}
          | term '/' fact         {$$ = $1 / $3;}
          | fact
          ;
fact:      INTEGER
          | '-' INTEGER           {$$ = - $2;}
          | '(' exp ')'          {$$ = $2;}
          ;
```

Exemple Lex et Yacc : évaluateur d'expressions

Exemple (partie Yacc de l'évaluateur d'expressions (3))

```
%%
int yyerror()
{
    printf("syntax error\n");
    return(-1);
}
main()
{
    yyparse();
    printf("goodbye\n");
}
```

Chapitre 11 : L'analyse sémantique

- 1 Rôle et phases de l'analyse sémantique
- 2 Outils pour effectuer l'analyse sémantique
- 3 Construction de l'AST
- 4 Quelques exemples d'utilisation de grammaires attribuées

Plan

- 1 Rôle et phases de l'analyse sémantique
- 2 Outils pour effectuer l'analyse sémantique
- 3 Construction de l'AST
- 4 Quelques exemples d'utilisation de grammaires attribuées

Rôle de l'analyse sémantique

Définition (Rôle de l'analyse sémantique)

Pour un langage impératif, l'**analyse sémantique** appelée aussi **gestion de contexte** s'occupe des relations non locales ; elle s'occupe ainsi :

- ① du **contrôle de visibilité** et du lien entre les définitions et utilisations des identificateurs
- ② du **contrôle de type** des “objets”, nombre et type des paramètres de fonctions
- ③ du **contrôle de flot** (vérifie par exemple qu'un goto est licite - voir exemple plus bas)

Exemple (de contrôle de flots erronné)

Le code suivant est illicite :

```
int main ()
{
    for(int i=1;i<10;++i)
        infor: cout << "iteration " << i << endl;
        goto infor;
}
```

Première phase de l'analyse sémantique

Construction de l'arbre syntaxique abstrait (et du graphe de flux de contrôle)

Définition (**AST (Abstract Syntax Tree)**) Arbre syntaxique abstrait)

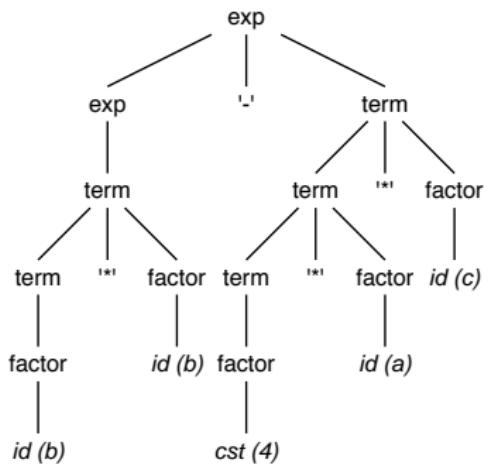
Forme résumée de l'arbre syntaxique qui ne conserve que les éléments utiles pour la suite.

Exemple (de grammaire, d'arbre syntaxique et d'AST)

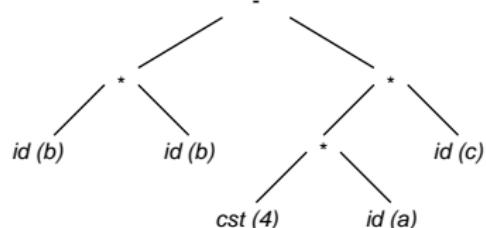
exp	\rightarrow	$exp + term$
	\rightarrow	$exp - term$
	\rightarrow	$term$
$term$	\rightarrow	$term * factor$
	\rightarrow	$term / factor$
	\rightarrow	$factor$
$factor$	\rightarrow	id cst (exp)

Soit G avec les règles de production : $term$

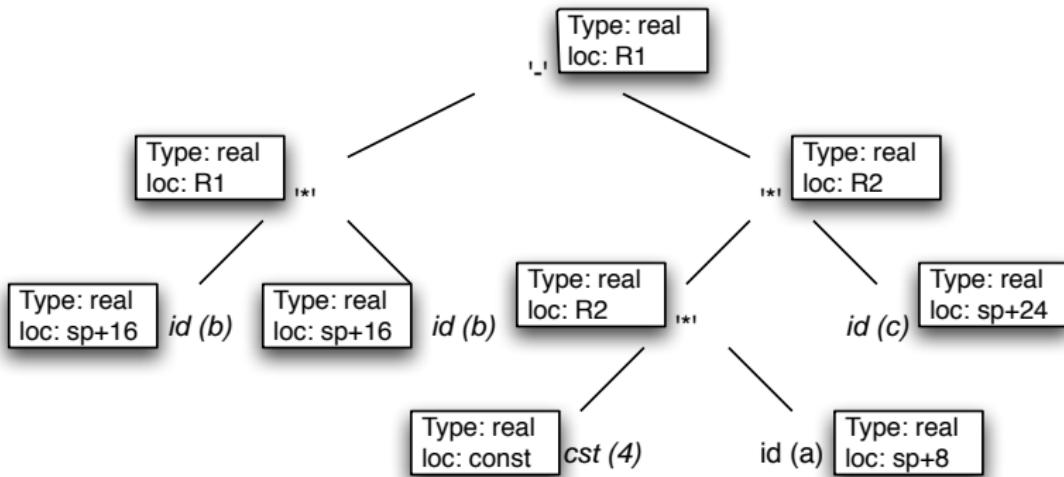
L'expression $b^*b-4^*a^*c$ donne
 l'arbre syntaxique suivant :



l'arbre abstrait (AST) suivant :



Cet AST sera aussi décoré lors des phases d'analyse sémantique et de génération de code ; par exemple :



Ceci permettra de gérer le contexte (collecte des informations sémantiques et vérification des contraintes) et ensuite de générer le code.

Deuxième phase de l'analyse sémantique

Gestion du contexte (contrôle sémantique)

Rappel sur le rôle de la gestion de contexte

La gestion de contexte des langages impératifs inclut :

- ① le **contrôle de la visibilité** et du lien entre les définitions et utilisations des identificateurs
- ② le **contrôle de type** des “objets”, nombre et type des paramètres de fonctions
- ③ le **contrôle de flot** (vérifie par exemple qu'un goto est licite)

L'analyseur sémantique manipule /calcule des attributs d'identificateurs

Attributs d'un "identificateur" :

- ① sorte (constante, variable, fonction)
- ② type
- ③ valeur initiale ou fixe
- ④ portée
- ⑤ propriétés éventuelles de "localisation" (place en mémoire : pour la génération du code)

Identification de la définition correspondant à une occurrence

Dépend de la portée (chaque langage à ses propres règles de portée)
En Pascal / C, peut être déterminée lors de l'analyse grâce à une **pile de portée** (qui peut être fusionnée avec la table des symboles) : Lors de l'analyse

- d'un début de bloc : on empile une nouvelle portée
- d'une nouvelle définition : on la met dans la portée courante
- de l'utilisation d'un élément : on trouve la définition correspondante en regardant dans le stack à partir du sommet
- de la sortie d'un bloc : on dépile la portée.

Note

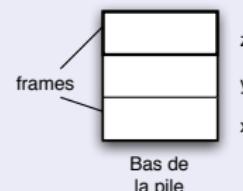
Cette technique permet également de trouver l'adresse d'une variable par exemple :

(nombre de frames statiques en dessous de la frame courante, place dans la frame)

Par exemple avec le code :

```
{  
    int x=3, y=4;  
    if(x==y)  
    {  
        int z=8;  
        y=z;  
    }  
}
```

y=z pourrait
être traduit par
var(1,4) ←
valvar(0,0)



La notion de frame est développée au chapitre \ref{chap12}

Autres problèmes d'identification ou de contrôle :

Surcharge

Plusieurs opérateurs ou fonctions ayant le même nom. Exemple : le '+' peut être

- + : double x double → double
- + : int x int → int
- const Matrix operator+(Matrix& m)
- ...

Polymorphisme

- Plusieurs fonctions ou méthodes ayant le même nom (ex : méthodes de même nom dans différentes classes).
- Polymorphisme “pur” : une fonction est polymorphe si elle peut s’appliquer de la même façon à tout type (ex : voir dans les langages fonctionnels)

Suivant le langage, le contrôle de type et en particulier la résolution du polymorphisme peut se faire

- **statiquement** c'est-à-dire lors de l'analyse sémantique, à la compilation
- **dynamiquement** c'est-à-dire lors de l'exécution, connaissant le type réel de l'objet.

Note

Dans les langages OO on parle

- d'**overloading**
 - nécessite que la signature soit différente pour déterminer quelle méthode exécuter
- d'**overriding**
 - nécessite que la signature soit la même
- de **polymorphisme**
 - sur des objets de classes différentes.

Autres problèmes d'identification ou de contrôle (suite) :

Coercition et casting

Il se peut que dans une opération le type attendu soit T_1 et que la valeur soit de type T_2 .

- Soit le langage accepte de faire une **coercition** càd une conversion de type non explicitement demandée par le programmeur.
- Soit la conversion doit être demandée explicitement grâce à un opérateur de **casting**.

Exemple (de casting et coercition)

```
{  
    double x=3.14 y=3.14;  
    int i,j;  
    i = x;           // coercition: 'int -> double'  
                  // un warning peut être émis  
    x = i;           // coercition: 'double -> int'  
                  // un warning peut être émis  
    j = (int) y;     // casting  
    y = (double) j; // casting  
}
```

Système de typage

Il faut d'abord

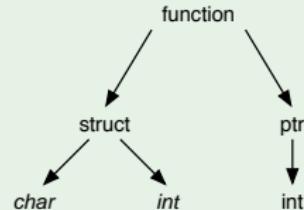
- ➊ pouvoir décrire formellement un type
- ➋ définir quels **types sont équivalents** ou compatibles

Définition (Expression de type)

Graphe orienté dont les noeuds peuvent être :

- *un type de base* : *bool, int, double,...*
- *un constructeur* :
 - *array*
 - *struct*
 - *function*
 - *pointer*
- *un nom de type* (*défini ailleurs*)
- *une variable de type* (*représentant a priori n'importe quel type*)

Exemple (d'expression de type)



Caractéristiques d'un langage

Il peut être

- de typage **statique** ou **dynamique** selon que ce typage peut être fait entièrement à la compilation ou doit être fait en partie à l'exécution.
- de typage **sain** si les types de toutes les valeurs utilisées à l'exécution sont calculées statiquement.
- **typé** si le typage impose des conditions sur les programmes.
- **fortement typé**
 - si les informations de typage sont associées aux variables et non pas aux valeurs, ou
 - si l'ensemble des types utilisés est connu à la compilation et le type des variables peut être testé à l'exécution.

²Ce terme a plusieurs définitions

Équivalence de type

Suivant le langage de programmation, l'équivalence de type peut être une équivalence de **nom** ou une équivalence de **structure**

En Pascal, C, C++

En Pascal, C et C++ on a l'**équivalence de nom** :

- V et W sont de type différent de X et Y

```
struct {int i; char c;} V,W;  
struct {int i; char c;} X,Y;
```

- idem

```
typedef struct {int i; char c;} S;  
S V,W;  
struct {int i; char c;} X,Y;
```

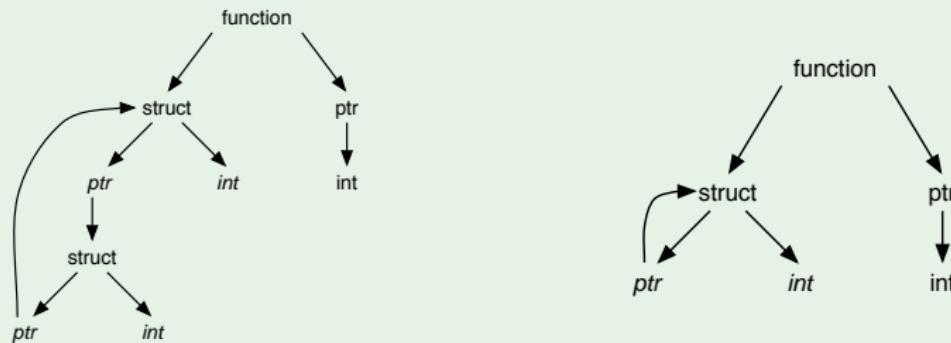
- les variables suivantes sont de types équivalents ! : pointeur constant vers un entier

```
typedef int Veccent[100];  
Veccent V,W;  
int X[100],Y[10],Z[1];
```

Exemple d'équivalence de structure

Dans d'autres langages (Algol68 par exemple), l'équivalence est déterminée par la structure.

Exemple (Expressions structurellement équivalentes)



Remarque :

L'algorithme d'unification (voir la résolution en Prolog) permet de vérifier si 2 types sont équivalents / compatibles.

Plan

- 1 Rôle et phases de l'analyse sémantique
- 2 Outils pour effectuer l'analyse sémantique
- 3 Construction de l'AST
- 4 Quelques exemples d'utilisation de grammaires attribuées

Outils pour effectuer l'analyse sémantique

Même si l'analyse sémantique n'est pas automatisée comme l'est l'analyse lexicale ou syntaxique, deux outils existent.

- Les actions sémantiques
- Les grammaires attribuées

Les actions sémantiques

Définition (Actions sémantiques)

Actions rajoutées dans la grammaire. Lors de l'analyse syntaxique, l'analyse de ces actions correspond à effectuer les actions prescrites.

Exemple (d'action sémantique)

Exemple classique : traduction d'expressions en notation algébrique directe (NAD) vers la notation polonaise inversée (NPI) :

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +T\{\text{printf('+')}\}E' \\ E' & \rightarrow & \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *F\{\text{printf('*')}\}T' \\ T' & \rightarrow & \epsilon \\ F & \rightarrow & (E) \\ F & \rightarrow & id\{\text{printf(val(id))}\} \end{array}$$

Les grammaires attribuées

Définition (Grammaire attribuées)

Il s'agit de traitements spécifiés sur la grammaire context-free décrivant la syntaxe, qui consistent à évaluer des attributs associés aux noeuds de l'arbre syntaxique

Définition (Attributs hérités et synthétisés)

Il n'existe que deux types d'attributs associés à un noeud :

- **Hérité** : dont la valeur ne peut dépendre que d'attributs de son père ou de ses frères,
- **Synthétisés** : dont la valeur ne peut dépendre que d'attributs de ses fils, ou si le noeud est une feuille, est donné par l'analyseur lexical (ou syntaxique).

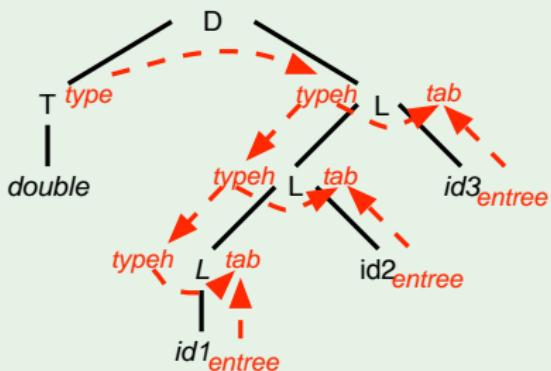
Exemple (de grammaire attribuée)

$D \rightarrow T L$	$L.typeh := T.type$
$T \rightarrow int$	$T.type := int$
$T \rightarrow double$	$T.type := double$
$L \rightarrow L_1 id$	$L_1.typeh := L.typeh$
$L \rightarrow id$	$L.tab := AjouterType(L_1.tab, id.entre, L.typeh)$ $L.tab := AjouterType(vide, id.entre, L.typeh)$

Sur l'arbre, les règles de calculs des attributs donne un **graphe de dépendances**

- $A \dashrightarrow B$ = la valeur de A est utilisée pour calculer B

Exemple :
double id1 id2 id3
Donne :



Evaluation des attributs

Soit l'ordre est fixé avant la compilation (ordre **statique**), soit il est déterminé lors de la compilation (ordre **dynamique**).

Evaluation dynamique naïve

tant que les attributs ne sont pas tous évalués
 Evaluate (Racine)

```
fonction Evaluate(S) (Règle S -> X1 X2 ... Xm) {  
    1°) propage les attributs présents dans S  
    2°) pour tout fils Xi (i=1 -> m):     Evaluate(Xi)  
    3°) pour tout attribut synthétisé A non encore évalué  
        si les attributs requis sont évalués alors     calcule A  
}
```

Autre méthode : Tri topologique

Tout attribut est dans une liste de prédécesseurs.

On donne le nombre de prédécesseurs de chacun.

On peut évaluer les attributs A sans prédécesseur et diminuer le nombre de prédécesseurs des attributs qui dépendent de A .

Note

Les attributs cycliques posent problème avec ces méthodes

Les types de grammaires attribuées 'classiques' qui peuvent être évaluées statiquement :

- **Les grammaires S-attribuées** : n'ont que des attributs synthétisés (ex : analyseur produits par YACC)
- **Les grammaires L-attribuées** : évaluables lors d'un parcourt LR de l'arbre du type parcours en profondeur gauche. Cela signifie qu'un attribut hérité ne dépend que des attributs hérités plus à gauche ou du père.

Remarques : les évaluations des attributs de ces grammaires peuvent être effectués dans une analyse LL(1) ou LALR(1).

Plan

- 1 Rôle et phases de l'analyse sémantique
- 2 Outils pour effectuer l'analyse sémantique
- 3 Construction de l'AST
- 4 Quelques exemples d'utilisation de grammaires attribuées

Construction d'un AST (ou AS-DAG) pour une expression

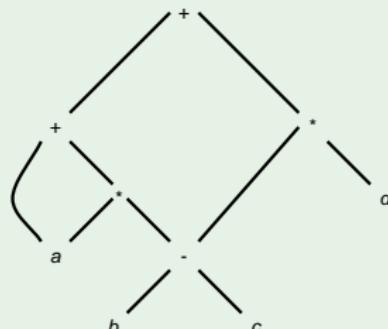
Exemple (de construction d'un AST)

$E \rightarrow E_1 + T$	$E.\text{noeud} := \text{CréerNoeud}('+', E_1.\text{noeud}, T.\text{noeud})$
$E \rightarrow E_1 - T$	$E.\text{noeud} := \text{CréerNoeud}('-', E_1.\text{noeud}, T.\text{noeud})$
$E \rightarrow T$	$E.\text{noeud} := T.\text{noeud}$
$T \rightarrow (E)$	$T.\text{noeud} := E.\text{noeud}$
$T \rightarrow id$	$T.\text{noeud} := \text{CréerFeuille}(id, id.\text{entrée})$
$T \rightarrow nb$	$T.\text{noeud} := \text{CréerFeuille}(nb, nb.\text{entrée})$

CréerFeuille et
CréerNoeud

- vérifient si le noeud existe déjà
- renvoient un pointeur vers le noeud existant ou créé.

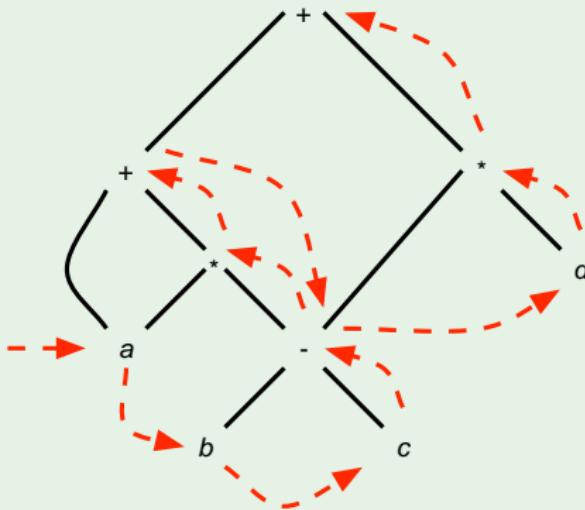
$a+a*(b-c)+(b-c)*d$
donne :



Graphe de Flot de Contrôle

En étendant ce calcul d'attributs, on peut obtenir le **Graphe de flot de contrôle** qui est la base de nombreuses optimisations du programme.

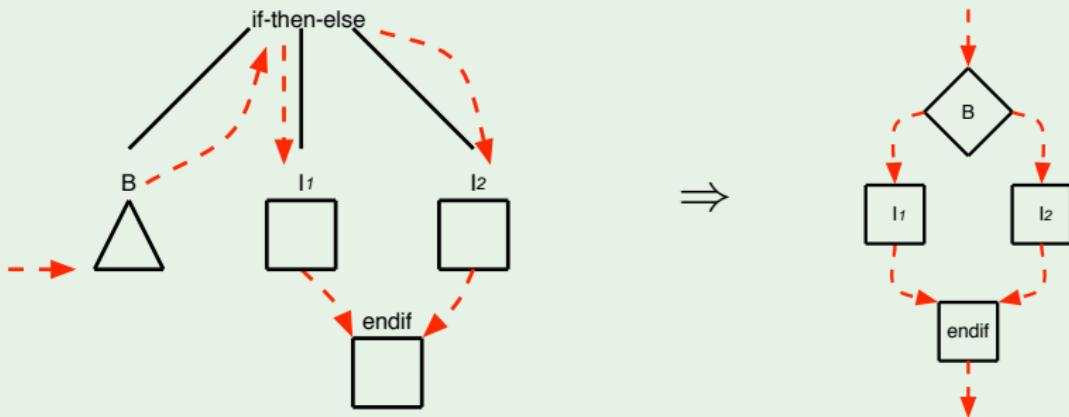
Exemple (de graphe de flux de contrôle)



Graphe de Flux de Contrôle

Exemple ((2) de graphe de flux de contrôle)

] Graphe de flot pour *if B then I₁ else I₂ endif*



Graphe de flux de contrôle

Definition (Graphe de flux de contrôle)

- C'est un organigramme donnant les flux possibles des instructions.
- Il est formé de **blocs de base**.
- Un bloc de base est une **séquence d'instructions consécutives sans arrêt ni branchement**

Plan

- 1 Rôle et phases de l'analyse sémantique
- 2 Outils pour effectuer l'analyse sémantique
- 3 Construction de l'AST
- 4 Quelques exemples d'utilisation de grammaires attribuées

Exemple (Yacc)

```
S      :   E      ;
E      :   E  '+'  E    { $$ = $1+$3 }
      |   T
      ;
T      :   T  '*'  F    { $$ = $1*$3 }
      |   F
      ;
F      :   '(' E ')'   { $$ = $2 }
      |   nb
      ;
```

Note sur Yacc

Dans cet exemple,

- la règle par défaut en YACC est $\$$ = \1
- la pile des attributs se comporte comme une pile d'évaluation postfixée

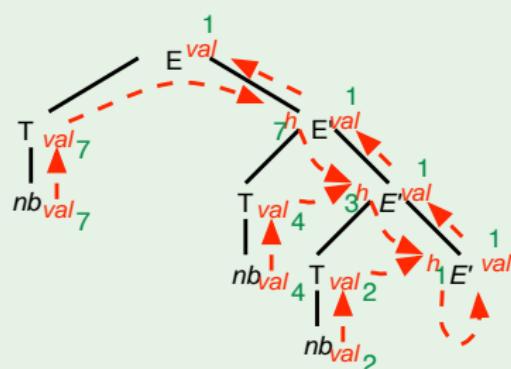
évaluation d'une expression en analyse LL(1)

Exemple (de grammaire attribuée)

Avec la grammaire attribuée :

$$\begin{array}{ll} E \rightarrow TE' & E'.h := T.val \\ & E.val = E'.val \\ E' \rightarrow -TE'_1 & E'_1.h = E'.h - T.val \\ & E'.val = E'_1.val \\ E' \rightarrow \epsilon & E'.val = E'.h \\ T \rightarrow nb & T.val = nb.val \end{array}$$

7 – 4 – 2 sera évalué :



Avec une descente LL(1) récursive, les attributs peuvent être transmis via des paramètres d'entrée ou de sortie.

Chapitre 12 : La génération de code

- 1 Note préliminaire : langages considérés
- 2 Particularités des langages impératifs (et organisation mémoire)
- 3 Code intermédiaire
- 4 Architecture du processeur
- 5 Génération du code

Plan

- 1 Note préliminaire : langages considérés
- 2 Particularités des langages impératifs (et organisation mémoire)
- 3 Code intermédiaire
- 4 Architecture du processeur
- 5 Génération du code

Restriction

Dans ce cours, nous considérons que le langage source est impératif et procédural

Types de langages

- Array languages
- Aspect Oriented Programming Languages
- Assembly languages
- Command Line Interface (CLI) languages (batch languages)
- Concatenative languages
- Concurrent languages
- Data-oriented languages
- Dataflow languages
- Data-structured languages
- Fourth-generation languages
- Functional languages
- Logical languages
- Machine languages
- Macro languages
- Multiparadigm languages
- Object-oriented languages
- Page description languages
- Procedural languages
- Rule-based languages
- Scripting languages
- Specification languages
- Syntax handling languages
- ...

Plan

- 1 Note préliminaire : langages considérés
- 2 Particularités des langages impératifs (et organisation mémoire)
- 3 Code intermédiaire
- 4 Architecture du processeur
- 5 Génération du code

Questions à se poser

- ① Les fonctions peuvent-elles être **récursives** ?
- ② Une fonction peut-elle référencer des **identificateurs non locaux** ?
- ③ Un bloc peut-il référencer des **identificateurs non locaux** ?
- ④ Quels **types de passage de paramètres** sont possibles ?
- ⑤ Des **fonctions** peuvent-elles être **passées en paramètre** ?
- ⑥ Une **fonction** peut-elle être passée **en résultat** ?
- ⑦ Le programme peut-il **alloquer dynamiquement de la mémoire** ?
- ⑧ L'espace **mémoire** doit-il être **libéré explicitement** ?

Fixons les réponses pour simplifier

- ➊ type(s) de passage de paramètre ? : par **valeur et référence**
- ➋ fonctions récursives ? : **oui**
- ➌ fonction qui fait référence à des id non locaux ? : **non**
- ➍ bloc qui fait référence à des id non locaux ? : **oui**
- ➎ fonctions passées en paramètre ? : **non**
- ➏ fonction passée en résultat ? : **non**
- ➐ allocation dynamique de mémoire ? : **oui**
- ➑ espace mémoire libéré explicitement ? : **oui**

Type(s) de passage de paramètre ? : par valeur et référence

Principaux types de passages de paramètre :

- **Par valeur** : le paramètre formel est une variable locale à la fonction ; sa valeur est initialisée avec la valeur du paramètre effectif
- **Par référence (ou par variable)** : l'adresse de la variable passée en paramètre est transmise à la fonction
- **Par copie (in / out)** : le paramètre formel est une variable locale à la fonction ; sa valeur est initialisée avec la valeur de la variable donnée en paramètre effectif ; à la fin de l'exécution de la fonction, la nouvelle valeur est copiée dans le paramètre effectif
- **Par nom** : remplacement textuel du paramètre formel par le paramètre effectif transmis.

Fonctions récursives ? : oui

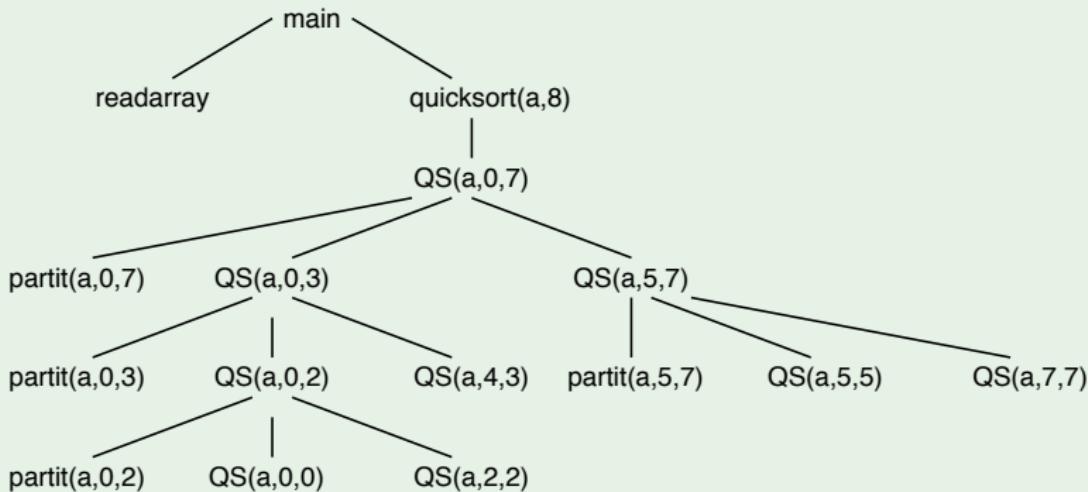
L'exécution d'un programme est schématisé par son **arbre d'activation**

Exemple (de programme et d'arbre d'activation)

```
void readarray(int a[], int aSize){...}
static int partit(int a[], int first, int last) {...}
static void QS(int a[], int first, int last) {
    if (first < last) {
        int pivotIndex = partit(a, first, last);
        QS(a, first, pivotIndex - 1);
        QS(a, pivotIndex + 1, last);
    }
}
static void quicksort(int a[], int aSize) {
    QS(a, 0, aSize - 1);
}
int main() {
    ...
    readarray(a,n);
    quicksort(a,n);
}
```

Exemple d'arbre d'activation pour quicksort

Exemple (de programme et d'arbre d'activation)



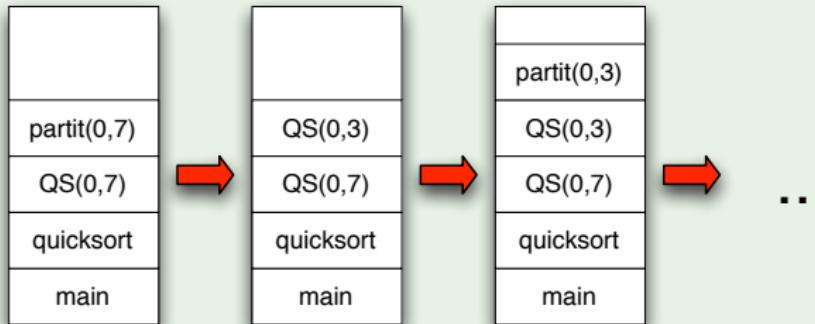
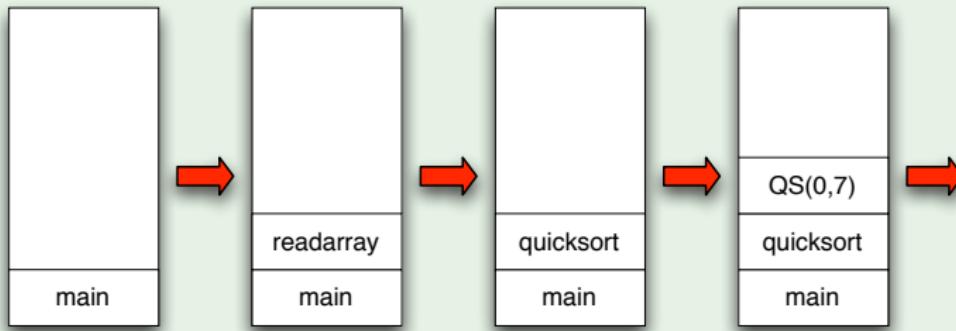
Exécution = parcours en profondeur de l'arbre d'activation

l'arbre d'activation n'est connu qu'à l'exécution

=> pile de contrôle (run-time) pour stocker les contextes et variables locales des fonctions appelantes

Evolution de la pile run-time lors de l'exécution

Exemple (d'évolution de la pile)



Bloc qui fait référence à des id non locaux ? : **oui**

Bloc d'activation (Frame)

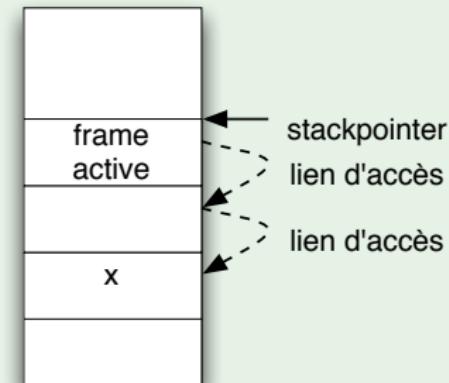
Chaque zone du stack correspondant à un bloc est nommé **bloc d'activation**.
On utilisera aussi le nom anglais **frame**.

Bloc qui fait référence à des id non locaux ? : oui

- Les références à des zones mémoires (variables par exemple) seront codées par des accès mémoire dont l'adresse est relative au début de la frame.
- Par exemple, l'accès à une variable x du bloc du grand-père correspond à un accès qui est relatif à l'antépénultième frame du stack run-time.
- Pour cela, chaque frame contient l'adresse du début de la frame directement en dessous (comme une liste simplement liée)

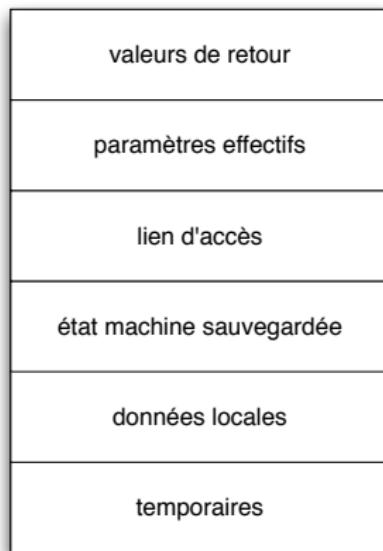
Exemple (de lien d'accès)

Si x se trouve dans un bloc “grand-père”, l'accès à x se fait à travers 2 **liens d'accès**.



Structure d'une frame (bloc d'activation) run-time

Cas le plus complet : bloc correspondant à une fonction



Remarque sur l'accès à une composante de tableau

Exemple (d'accès à une composante de tableau)

- Si V est un tableau à 3 dimensions $n_1 \times n_2 \times n_3$,
- en supposant que le langage stocke les éléments “ligne par ligne”
- et que la première composante est $V[0, 0, 0]$
- $V[i, j, k]$ est à l'adresse $*V + i.n_1.n_2 + j.n_2 + k$
- que l'on peut également écrire (par Horner) : $*V + ((i.n_1) + j).n_2 + k$
- Remarquons que pour ce calcul, n_3 n'est pas requis.

Allocation dynamique de mémoire ? : oui

Les allocations se font lors des **new**

Cela nécessite une zone mémoire supplémentaire : le **tas (heap)** qui n'a pas de structure (FIFO, LIFO, ...).

La zone mémoire contient donc

- des zones allouées au programme et
- d'autres disponibles (allouables) que les fonctions run-time gèrent

Espace mémoire libéré explicitement ? : oui (avec **delete**)

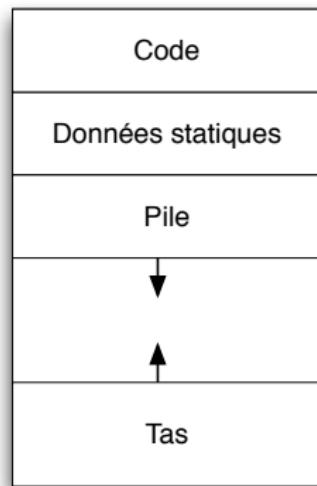
- Sinon, un **garbage collector** est nécessaire pour pouvoir récupérer les parties du tas qui ne sont plus accessibles par le programme.
- Un garbage collector est une fonction run-time qui récupère sur le tas, l'espace mémoire qui n'est plus accessible par le programme.
- Cela permet au run-time de satisfaire des demandes ultérieures d'allocation.

Remarque :

Le garbage collector peut interrompre l'exécution du programme pendant son travail
=> problème pour les systèmes temps réel

Répartition de la mémoire à l'exécution

La mémoire d'un programme qui s'exécute est donc généralement composée de 4 zones

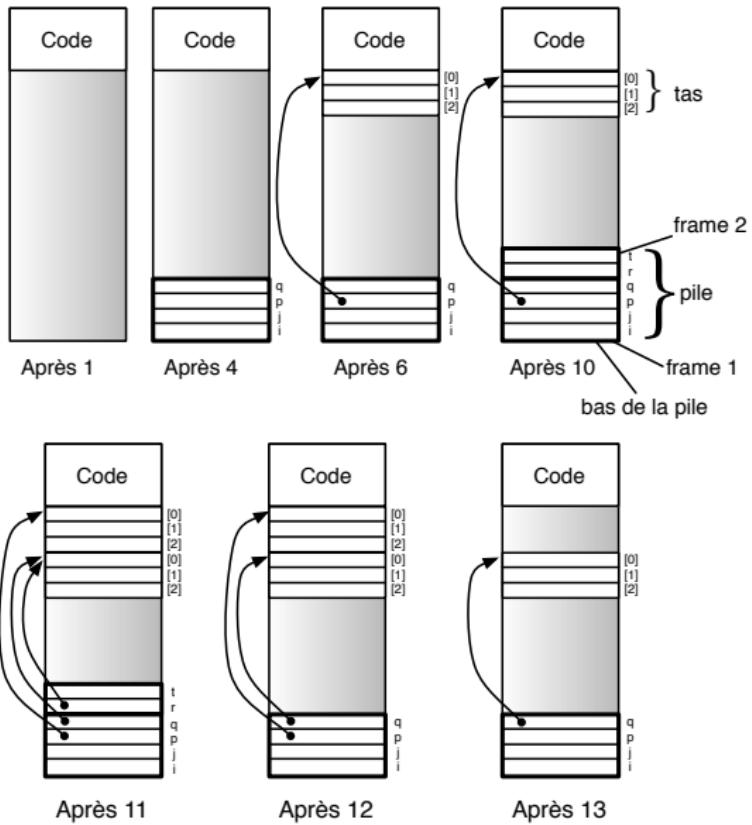


Un exemple : programme et mémoire associée

Note :

Pour simplifier le schéma, seules les variables sont représentées sur le tas

```
int main()           //1
{
    int i, j;        //2
    int *p, *q;       //3
    cin >>i;          //4
    p = new int[i];   //5
    if (i==3)         //6
    {
        int *r;       //7
        int t=4;       //8
        r=q=new int[i]; //9
    }
    delete[] p;       //10
    ...
}
```



Plan

- 1 Note préliminaire : langages considérés
- 2 Particularités des langages impératifs (et organisation mémoire)
- 3 Code intermédiaire**
- 4 Architecture du processeur
- 5 Génération du code

Langages intermédiaires

Deux approches classiques existent :

- On utilise un langage intermédiaire avec **instructions à 3 adresses** de forme générale $x := y \ op \ z$
- On utilise le byte-code de machines virtuelles (exemple JVM = Java Virtual Machine) ce qui peut éventuellement exempter de produire du code machine.

Un exemple de machine virtuelle simplifiée : la P-machine

définie dans le livre de Wilhelm et Maurer ; (adaptée au langage Pascal).

- Une pile d'évaluation et de contexte
- **SP** : registre pointeur de sommet de pile (pile : [0..*maxstr*])
- **PC** : registre pointeur vers l'instruction courante
- **EP** : registre pointeur vers l'emplacement le plus haut nécessaire pour exécuter le block courant
- **MP** : registre pointeur vers le début de la frame courante sur la pile
- **NP** : registre pointeur vers la dernière zone occupée sur le tas
- code : [0..*codemax*] ; 1 instruction par mot, init : $PC = 0$
- types
 - **i** : entier
 - **r** : réel
 - **b** : booléen
 - **a** : adresse
- notations
 - **N** signifie numérique
 - **T** signifie tout type ou adresse

Expressions

Instr	Sémantique	Cond	Result
add <i>N</i>	$STORE[SP - 1] = STORE[SP - 1] + STORE[SP]; SP --$	(<i>N</i> , <i>N</i>)	(<i>N</i>)
sub <i>N</i>	$STORE[SP - 1] = STORE[SP - 1] - STORE[SP]; SP --$	(<i>N</i> , <i>N</i>)	(<i>N</i>)
mul <i>N</i>	$STORE[SP - 1] = STORE[SP - 1] * STORE[SP]; SP --$	(<i>N</i> , <i>N</i>)	(<i>N</i>)
div <i>N</i>	$STORE[SP - 1] = STORE[SP - 1] / STORE[SP]; SP --$	(<i>N</i> , <i>N</i>)	(<i>N</i>)
neg <i>N</i>	$STORE[SP] = - STORE[SP]$	(<i>N</i>)	(<i>N</i>)
and <i>N</i>	$STORE[SP - 1] = STORE[SP - 1] \text{ and } STORE[SP]; SP --$	(<i>b</i> , <i>b</i>)	(<i>b</i>)
or <i>N</i>	$STORE[SP - 1] = STORE[SP - 1] \text{ or } STORE[SP]; SP --$	(<i>b</i> , <i>b</i>)	(<i>b</i>)
not <i>N</i>	$STORE[SP] = \text{not } STORE[SP]$	(<i>b</i>)	(<i>b</i>)
equ <i>T</i>	$STORE[SP - 1] = STORE[SP - 1] == STORE[SP]; SP --$	(<i>T</i> , <i>T</i>)	(<i>b</i>)
geq <i>T</i>	$STORE[SP - 1] = STORE[SP - 1] >= STORE[SP]; SP --$	(<i>T</i> , <i>T</i>)	(<i>b</i>)
leq <i>T</i>	$STORE[SP - 1] = STORE[SP - 1] <= STORE[SP]; SP --$	(<i>T</i> , <i>T</i>)	(<i>b</i>)
les <i>T</i>	$STORE[SP - 1] = STORE[SP - 1] < STORE[SP]; SP --$	(<i>T</i> , <i>T</i>)	(<i>b</i>)
grt <i>T</i>	$STORE[SP - 1] = STORE[SP - 1] > STORE[SP]; SP --$	(<i>T</i> , <i>T</i>)	(<i>b</i>)
neq <i>T</i>	$STORE[SP - 1] = STORE[SP - 1] != STORE[SP]; SP --$	(<i>T</i> , <i>T</i>)	(<i>b</i>)

Exemple : add i

Load et store

Instr	Sémantique	Cond	Result
ldo $T\ q$	$SP++;$ $STORE[SP] = STORE[q]$	$q \in [0..maxstr]$	(T)
ldc $T\ q$	$SP++;$ $STORE[SP] = q$	$Type(q) = T$	(T)
ind T	$STORE[SP] = STORE[STORE[SP]]$	(a)	(T)
sro $T\ q$	$STORE[q] = STORE[SP]; SP--$	(T) $q \in [0..maxstr]$	
sto T	$STORE[STORE[SP-1]] = STORE[SP]; SP = SP - 2$	(a, T)	

Utilisation

- **ldo** : met le mot d'adresse q au sommet de la pile
- **ldc** : met la constante q au sommet de la pile
- **ind** : remplace l'adresse au sommet de la pile par le contenu du mot correspondant
- **sro** : met le sommet de la pile à l'adresse q
- **sto** : met la valeur au sommet de la pile à l'adresse donnée juste en dessous sur la pile

Code de $x = y;$

Pile $\leftarrow @x;$ Pile $\leftarrow y;$ sto i

Sauts

Instr	Sémantique	Cond	Result
u jp q	$PC = q$	$q \in [0..codemax]$	
f jp q	<i>if STORE[SP] == false then $PC = q$ fi</i> $SP --$	(b) $q \in [0..codemax]$	
ixj q	$PC = STORE[SP] + q; SP --$	(i)	

Allocation de mémoire et calculs d'adresses (tableaux statiques, dynamiques)

Instr	Sémantique	Cond	Result
ixa q	$STORE[SP - 1] = STORE[SP - 1] +$ $STORE[SP] * q; SP --$	(a, i)	(a)
inc $T q$	$STORE[SP] = STORE[SP] + q$	$(T) \text{ and } type(q) = i$	(T)
dec $T q$	$STORE[SP] = STORE[SP] - q$	$(T) \text{ and } type(q) = i$	(T)
chk $p q$	<i>if</i> $(STORE[SP] < p) \text{ or } (STORE[SP] > q)$ <i>then error("value out of range") fi</i>	(i, i)	(i)
dpl T	$SP++; STORE[SP] = STORE[SP - 1]$	(T)	(T, T)
ldd q	$SP++;$ $STORE[SP] = STORE[STORE[SP - 3] + q]$	(a, T_1, T_2)	(a, T_1, T_2, i)
sli T_2	$STORE[SP - 1] = STORE[SP]; SP --$	(T_1, T_2)	(T_2)
new	<i>if</i> $(NP - STORE[SP] \leq EP)$ <i>then error("store overflow") fi</i> <i>else</i> $NP = NP - STORE[SP];$ $STORE[STORE[SP - 1]] = NP;$ $SP = SP - 2; fi$	(a, i)	

Gestion de la pile (variables, procédures,...)

Ayant par définition

$\text{base}(p, a) \equiv \text{if } (p == 0) \text{ then } a \text{ else } \text{base}(p - 1, \text{STORE}[a + 1])$

Instr	Sémantique	Commentaires
lod $T p q$	$SP++;$ $\text{STORE}[SP] = \text{STORE}[\text{base}(p, MP) + q]$	load value
lda $p q$	$SP++;$ $\text{STORE}[SP] = \text{base}(p, MP) + q$	load address
str $T p q$	$\text{STORE}[\text{base}(p, MP) + q] = \text{STORE}[SP];$ $SP--$	store
mst p	$\text{STORE}[SP + 2] = \text{base}(p, MP);$ $\text{STORE}[SP + 3] = MP;$ $\text{STORE}[SP + 4] = EP;$ $SP = SP + 5$	static link dynamic link sauve EP
cup $p q$	$MP = SP - (p + 4);$ $\text{STORE}[MP + 4] = PC;$ $PC = q$	p est la place pour les paramètres sauve l'adresse de retour branchement en q
ssp p	$SP = MP + p - 1$	p = place pour les variables statiques
sep p	$EP = SP + p;$ $\text{if } EP \geq NP$ $\text{then error("store overflow") fi}$	p est la profondeur max de la pile contrôle la collision pile / tas
ent $p q$	$SP = MP + q - 1;$ $EP = SP + p;$ $\text{if } EP \geq NP$ $\text{then error("store overflow") fi}$	q zone de données p est la profondeur max de la pile contrôle la collision pile / tas

Gestion de la pile (variables, procédures,...)

Utilisation

- **lod** : met sur la pile la valeur d'adresse (p, q) : p liens statiques, q offset dans la frame
- **lda** : idem mais on met l'adresse du mot sur la pile
- **str** : store
- **mst** : met sur la pile : lien statique, dynamique, EP
- **cup** : branchement avec sauvegarde de l'adresse de retour et mise à jour de MP
- **ssp** : allocation sur la pile de p entrées
- **sep** : contrôle si on peut augmenter le stack de p emplacements
- **ent** : exécution en séquence de **ssp** et **sep**

Liens statiques et dynamiques : voir référence Wilhelm et Maurer.

Gestion de la pile (procédures, passages des paramètres,...)

Instr	Sémantique	Commentaires
retf	$SP = MP;$ $PC = STORE[MP + 4];$ $EP = STORE[MP + 3];$ <i>if</i> $EP \geq NP$ <i>then</i> error("store overflow") <i>fi</i> $MP = STORE[MP + 2]$	résultat de la fonction sur la pile return restore EP restore dynamic link
retp	$SP = MP - 1;$ $PC = STORE[MP + 4];$ $EP = STORE[MP + 3];$ <i>if</i> $EP \geq NP$ <i>then</i> error("store overflow") <i>fi</i> $MP = STORE[MP + 2]$	procédure sans résultat return restore EP restore dynamic link

Gestion de la pile (procédures, passages des paramètres,...)

Instr	Sémantique	Cond	Résultats
movs <i>q</i>	$\text{for } (i = q - 1; i \geq 0; -- i)$ $STORE[SP + i] = STORE[STORE[SP] + i];$ <i>od</i> $SP = SP + q - 1$	(a)	
movd <i>q</i>	$\text{for } (i = 1; i \leq STORE[MP + q + 1]; ++ i)$ $STORE[SP + i] = STORE[STORE[MP + q] +$ $STORE[MP + q + 2] + i - 1];$ <i>od</i> $STORE[MP + q] = SP + 1 - STORE[MP + q + 2];$ $SP = SP + STORE[MP + q + 1];$		

Utilisation

- **movs** : copie un bloc de taille fixée de données sur la pile
- **movd** : copie un bloc de taille connue à l'exécution

Gestion de la pile (procédures, passages des paramètres,...)

Ayant $\text{base}(p, a) = \text{if } p = 0 \text{ then } a \text{ else } \text{base}(p - 1, \text{STORE}[a + 1])$

Instr	Sémantique	Commentaires
smp p	$MP = SP - (p + 4);$	set MP
cupi $p q$	$STORE[MP + 4] = PC;$ $PC = STORE[\text{base}(p, \text{STORE}[MP + 2] + q)]$	return address
mstf $p q$	$STORE[SP + 2] = STORE[\text{base}(p, MP) + q + 1];$ $STORE[SP + 3] = MP;$ $STORE[SP + 4] = EP;$ $SP = SP + 5$	dynamic link sauve EP

Note préliminaire : langages considérés
Particularités des langages impératifs (et organisation mémoire)

Code intermédiaire

Architecture du processeur
Génération du code

Label, I/O, et stop

Instr	Sémantique	Commentaires
define @i		@i = adresse de l'instruction suivante
prin	$\text{Print}(\text{STORE}[SP]); SP --$	imprime le sommet de la pile
read	$SP++; \text{STORE}[SP] = \text{integer input}$	lecture et mise sur la pile d'un entier
; stp		fin de programme

Plan

- 1 Note préliminaire : langages considérés
- 2 Particularités des langages impératifs (et organisation mémoire)
- 3 Code intermédiaire
- 4 **Architecture du processeur**
- 5 Génération du code

Types de processeurs

En gros, deux grandes classes de processeurs existent :

- les **machines à registres**

Les instructions utilisent un ensemble de registres permettant de faire les calculs. Généralement les registres sont spécialisés :

- registres généraux
- registres pour les calculs flottants
- registres des prédictats (1 bit)
- pointeur d'instruction courante (PC)
- pointeur du stack (SP)
- registres de statuts et contrôles
- registre d'application (rôles spécifiques)
- ...

Les instructions sont généralement à 1, 2 ou 3 opérandes (du type **opcode a1 a2 a3**).

Types de processeurs (suite)

- les machines à pile

Les instructions utilisent une pile d'évaluation permettant de faire un maximum d'opérations (calculs, condition de branchement, ...)
(Ex : la **Java Virtual Machine (JVM)**)

Type de processeurs (suite)

On distingue également les processeurs

- **CISC (Complex Instruction Set Computers)** qui ont un grand jeu d'instructions avec en général des registres spécialisés (ex :x86, pentium, 680xx)
- **RISC (Reduced Instruction Set Computers)** qui ont un jeu d'instructions très limité et en général de nombreux registres universels (SPARC, powerPC, nouvelles architectures, ...)

Plan

- 1 Note préliminaire : langages considérés
- 2 Particularités des langages impératifs (et organisation mémoire)
- 3 Code intermédiaire
- 4 Architecture du processeur
- 5 Génération du code

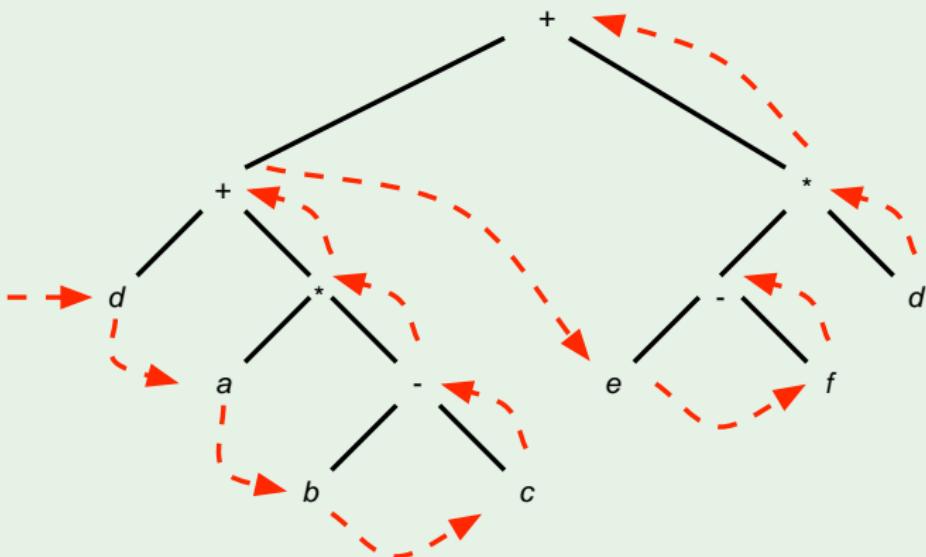
Rappels : résultats de l'analyse sémantique

L'output de l'analyseur sémantique comprend :

- un arbre syntaxique abstrait décoré (AST)
- un (embryon de) graphe de flot de contrôle
- une table des symboles structurée permettant de déterminer la portée de chaque identificateur

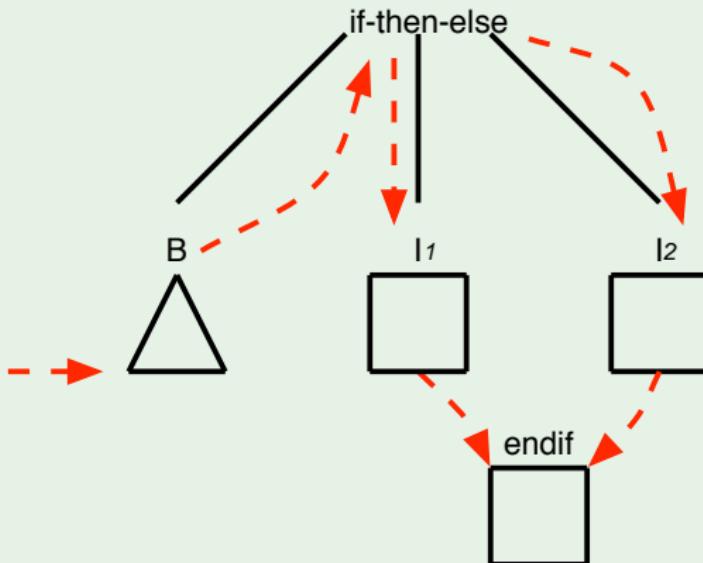
Rappels :AST et graphe de contrôle de flux

Exemple (AST décoré et graphe de flot de l'expression
 $d + a * (b - c) + (e - f) * d$)



Rappels :AST et graphe de contrôle de flux

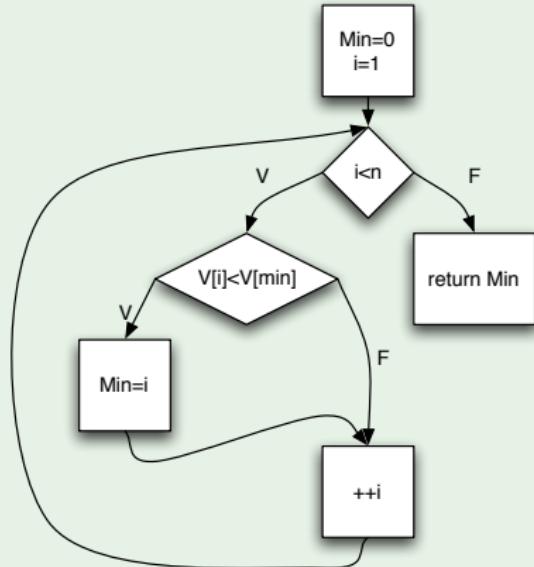
Exemple (AST décoré et graphe de flot d'une instruction **if B then I_1 else I_2 endif**)



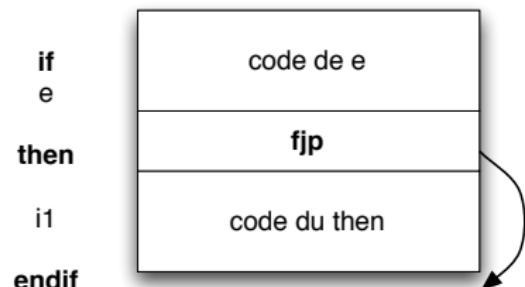
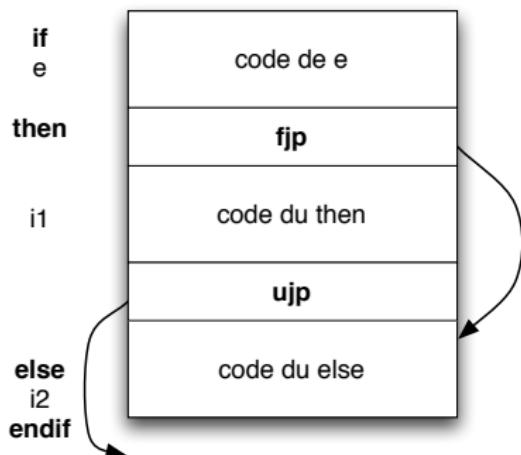
Exemple de graphe de flot de contrôle

Exemple (Graphe de flot correspondant à un simple code)

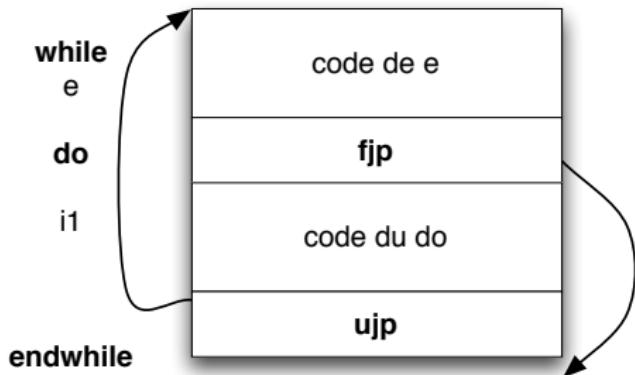
```
{  
    int Min=0;  
    for (int i=1; i<n; ++i)  
        if (V[i]<V[Min])  
            Min = i;  
    return Min;  
}
```



Code correspondant à un if



Code correspondant à un while



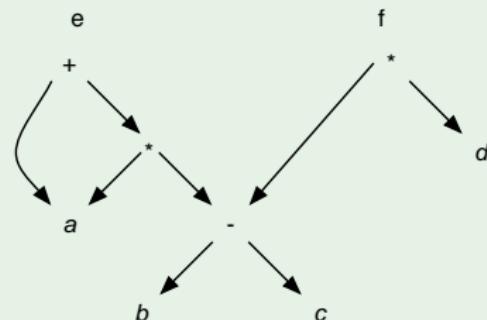
Représentation d'un bloc de base sous forme de DAG

(DAG = Directed Acyclic Graph) Le DAG contient

- un noeud par opérateur et
- une feuille par variable / constante utilisée

Exemple (code d'un bloc et DAG associé)

$e = a + a * (b - c);$
 $f = (b - c) * d;$

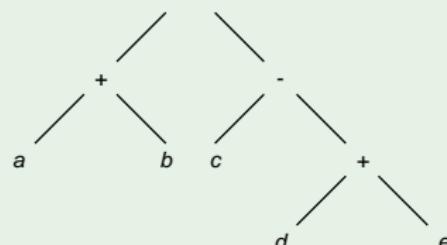


Génération du code correspondant à un bloc de base

- Grâce à l'AST (ou AS-DAG),
- des algorithmes optimisent l'utilisation de registres et
- l'ordre dans lequel les opérations sont évaluées

Exemple (Pour $(a + b) - (c - (d + e))$ avec 3 registres)

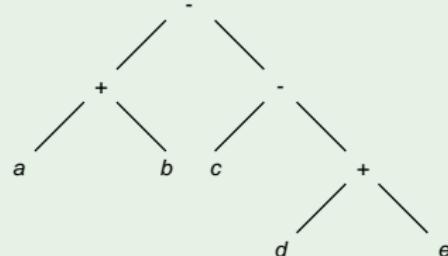
```
lw R1 , a
lw R2 , b
add R1 , R1 , R2
lw R2 , d
lw R3 , e
add R2 , R2 , R3
lw R3 , c
sub R2 , R3 , R2
sub R1 , R1 , R2
```



Code correspondant à une expression

Exemple (Pour $(a + b) - (c - (d + e))$ avec 2 registres)

```
lw R1 , a
lw R2 , b
add R1 , R1 , R2
sw R1 , T
lw R1 , d
lw R2 , e
add R1 , R1 , R2
lw R2 , c
sub R1 , R2 , R1
lw R2 , T
sub R1 , R2 , R1
```



Chapitre 13 : Les machines de Turing

- 1 Introduction
- 2 La machine de Turing (TM - 1936)
- 3 TM étendues et restreintes
- 4 Classes des langages RE et R vs classe 0 et classe 1
- 5 Propriétés des langages R et RE

Introduction

La machine de Turing (TM - 1936)

TM étendues et restreintes

Classes des langages RE et R vs classe 0 et classe 1

Propriétés des langages R et RE

Plan

1 Introduction

2 La machine de Turing (TM - 1936)

3 TM étendues et restreintes

4 Classes des langages RE et R vs classe 0 et classe 1

5 Propriétés des langages R et RE

Problèmes que l'on se pose

- ① Quel langage est définissable formellement
- ② Quel problème possède une **procédure effective (= procédure automatisable qui se termine toujours)**. En d'autres termes quel problème est-il **décidable**
- ③ Quel formalisme utilise t-on pour se poser ces questions (TM = machine de Turing)

Alan Mathison Turing (23 juin 1912 - 7 juin 1954)



était un mathématicien britannique et est considéré comme un des pères fondateurs de l'informatique moderne. Il est à l'origine de la formalisation du concept d'algorithme et de calculabilité qui ont profondément marqué cette discipline, avec la machine de Turing. Il a défini la thèse Church-Turing, maintenant largement partagée, qui postule que tout autre modèle de calcul est définissable par une machine de Turing (1936) et possède tout ou partie de sa capacité à être programmée. Durant la Seconde Guerre mondiale, il a dirigé les recherches qui ont conduit à déchiffrer le code secret généré par la machine Enigma utilisée par le camp Nazi. Après la guerre, il a travaillé sur un des tout premiers ordinateurs, puis a contribué de manière provocatrice au débat déjà houleux à cette période sur la capacité des machines à penser en établissant le test de Turing.

Plan

- 1 Introduction
- 2 **La machine de Turing (TM - 1936)**
- 3 TM étendues et restreintes
- 4 Classes des langages RE et R vs classe 0 et classe 1
- 5 Propriétés des langages R et RE

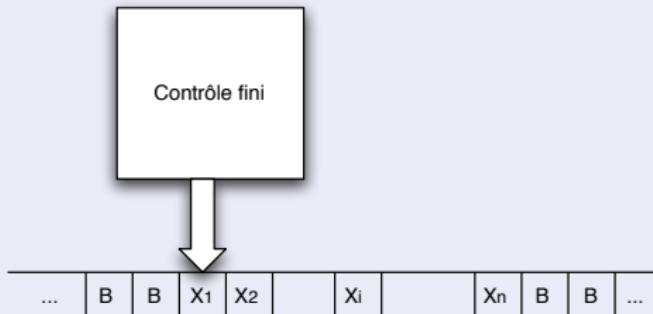
Modèle de la machine de Turing

La machine de Turing est la (une) formalisation admise actuellement pour modéliser une procédure effective (voir plus bas)

Machine de Turing : description informelle

Automate avec

- un contrôle fini
- un ruban d'entrée divisé en cellules et infini à gauche et à droite
- une tête de lecture / écriture qui examine une cellule à la fois



Modèle de la machine de Turing

Fonctionnement informel de la Machine de Turing

Initialement

- les cellules du ruban ont le symbole B ($B \notin \Sigma$)
- sauf n cellules contenant l'input ($X_1..X_n$)
- la tête de lecture est positionnée sur X_1

Un déplacement

- change l'état de contrôle
- écrit un symbole dans la cellule courante
- déplace la tête d'une position à gauche ou à droite

Définition formelle de la Machine de Turing (TM)

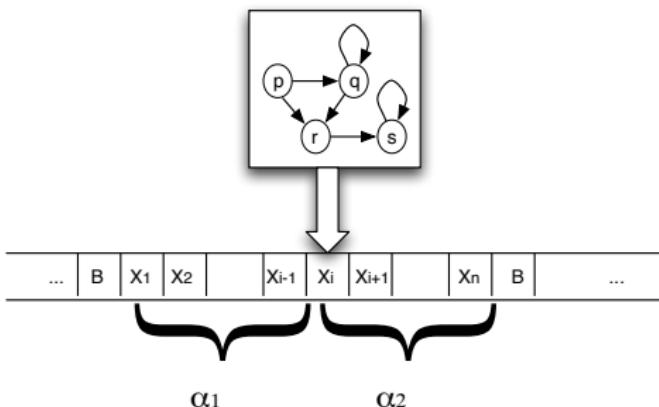
Définition (Machine de Turing (TM))

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

avec

- Q : ensemble fini d'états
- Σ : alphabet d'entrée fini
- Γ : alphabet fini du ruban avec $\Sigma \subset \Gamma$
- δ : la fonction de transition $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times D$ avec $\delta(p, X)$ pas toujours défini et $D \in \{L, R\}$ (Left, Right)
- q_0 l'état initial
- B le symbole blanc avec $B \in \Gamma \setminus \Sigma$
- F l'ensemble des états accepteurs avec $F \subseteq Q$

Description instantanée (ID = état global) d'une TM



Définition (Description instantanée (ID))

$ID = (\alpha_1, q, \alpha_2)$ ou plus simplement $\alpha_1 q \alpha_2$ avec

- q l'état courant
- α_1 , la chaîne de caractères avant le symbole courant
- α_2 la chaîne de caractères à partir du symbole courant

Pour pouvoir l'écrire, on oublie dans α_1 (resp. α_2) les B "initiaux" ("finaux")

Description instantanée (ID = état global) d'une TM

ID particuliers

Si la tête de lecture :

- est positionnée k symboles B à gauche du premier symbole $\neq B$
 $ID = qX_1X_2..X_n$ avec $X_1..X_k$ ces k symboles B
- est positionnée k symboles à droite du dernier symbole $\neq B$,
 $ID = X_1..X_{n-1}qX_n$ avec $X_{n-k+1}..X_n$ ces k symboles B

“Mouvement” d’une TM : $ID_1 \underset{M}{\vdash} ID_2$ ou $ID_1 \vdash ID_2$

Mouvement d’une TM ($ID_1 \underset{M}{\vdash} ID_2$)

Ayant une TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ avec

- $\delta(q, X_i) = (p, Y, L)$: alors

$$X_1 X_2 \dots X_{i-1} q X_i X_{i+1} \dots X_n \underset{M}{\vdash} X_1 X_2 \dots X_{i-2} p X_{i-1} Y X_{i+1} \dots X_n$$

- si $i = 1$ un B est rajouté à gauche dans ID_2 (symbole X_{i-1})

- $\delta(q, X_i) = (p, Y, R)$: alors

$$X_1 X_2 \dots X_{i-1} q X_i X_{i+1} \dots X_n \vdash X_1 X_2 \dots X_{i-1} Y p X_{i+1} \dots X_n$$

- si $i = n$ un B est rajouté à droite dans ID_2 (symbole X_{i+1})

Langage d'une TM

$$L(M) = \{w \in \Sigma^* \mid q_0 w \xrightarrow{*} \alpha p \beta \wedge p \in F\}$$

On suppose sans perte de généralité que la TM s'arrête quand elle accepte un mot.

Il est possible que pour certains mots non acceptés la TM ne s'arrête jamais

Exemple de TM

Exemple

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

State	Symbol				
	0	1	X	Y	B
q_0	(q_1, X, R)	—	—	(q_3, Y, R)	—
q_1	$(q_1, 0, R)$	(q_2, Y, L)	—	(q_1, Y, R)	—
q_2	$(q_2, 0, L)$	—	(q_0, X, R)	(q_2, Y, L)	—
q_3	—	—	—	(q_3, Y, R)	(q_4, B, R)
q_4	—	—	—	—	—

Exemple (Séquence acceptée : 0011)

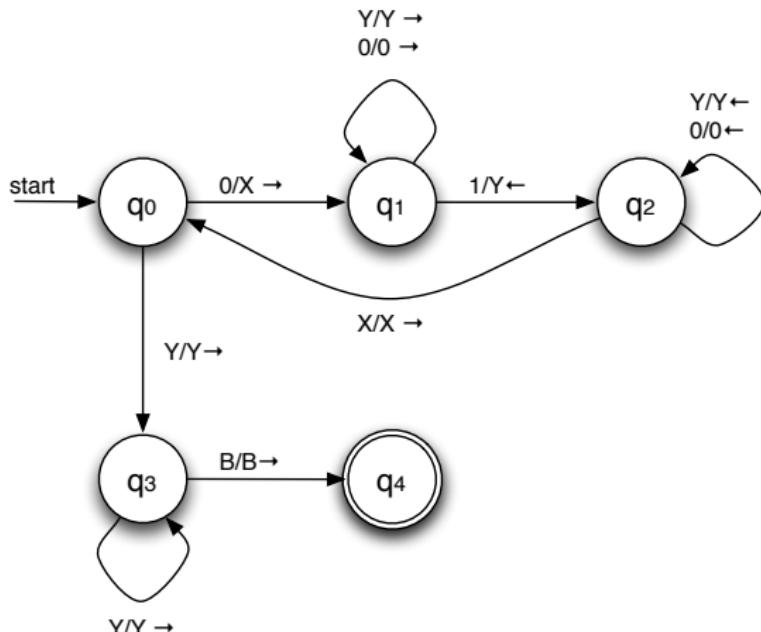
$q_0 0011 \vdash X q_1 011 \vdash X 0 q_1 11 \vdash X q_2 0 Y 1 \vdash q_2 X 0 Y 1 \vdash X q_0 0 Y 1 \vdash X X q_1 Y 1 \vdash X X Y q_1 1 \vdash X X q_2 Y Y \vdash X q_2 X Y Y \vdash X X q_0 Y Y \vdash X X Y q_3 Y \vdash X X Y Y q_3 B \vdash X X Y Y B q_4 B$

Exemple (Séquence refusée : 0010)

$q_0 0011 \vdash X q_1 010 \vdash X 0 q_1 10 \vdash X q_2 0 Y 0 \vdash q_2 X 0 Y 0 \vdash X q_0 0 Y 0 \vdash X X q_1 Y 0 \vdash X X Y q_1 0 \vdash X X Y 0 q_1 B$

Diagramme de transition d'une TM

Représentation graphique d'une TM



Langages récursivement énumérables (RE) et récursifs (R)

Définition (Langage récursivement énumérable (RE))

Langage accepté par une TM, c'est-à-dire langage formé de tous les strings acceptés par une TM

Définition (Langage récursif (R))

Langage calculé par une TM, c'est-à-dire langage formé de tous les strings acceptés par une TM qui s'arrête pour tout input.

Note

Un langage récursif peut donc être vu comme un langage tel qu'il existe un algorithme pour reconnaître ses mots

TM pour calculer une fonction entière

La fonction $-$ est définie comme suit : $m - n = \max(m - n, 0)$

Codification :

- input : $0^m 1 0^n$
- output : les 0 consécutifs donnent la réponse

Exemple

$$M = (\{q_0, q_1, \dots, q_6\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B)$$

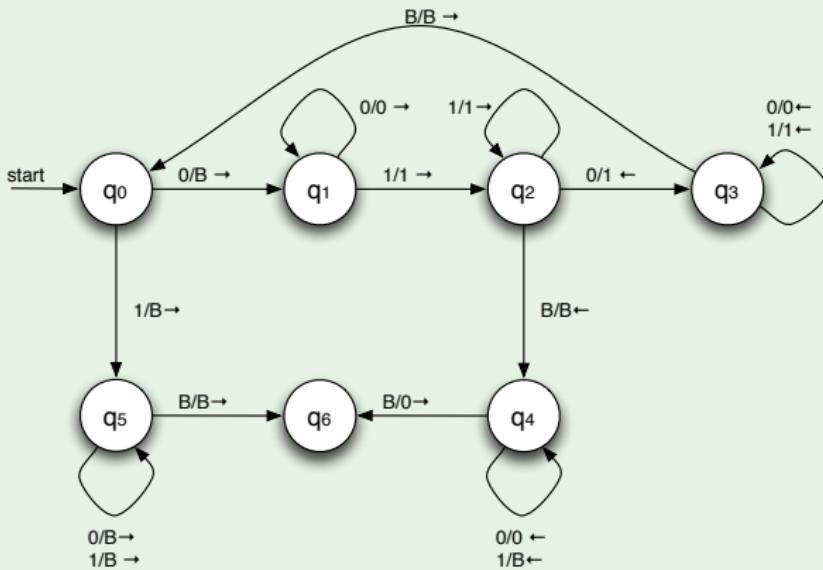
L'état accepteur est omis car la TM n'est pas utilisée comme accepteur.

δ	Symbol		
State	0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	—
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	—
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	—	—	—

TM pour calculer une fonction entière

La fonction $-$ est définie comme suit : $m - n = \max(m - n, 0)$

Exemple



Exemple d'exécution pour $4 - 2$ (codifié en 0000100)

q0	0	0	0	0	1	0	0		0	0	1	q3	1	1
	q1	0	0	0	1	0	0		0	0	q3	1	1	1
0	q1	0	0	1	0	0	0		0	q3	0	1	1	1
0	0	q1	0	1	0	0	0		q3	0	0	1	1	1
0	0	0	q1	1	0	0	0		q3	B	0	0	1	1
0	0	0	1	q2	0	0	0		q0	0	0	1	1	1
0	0	0	q3	1	1	0	0		q1	0	1	1	1	1
0	0	q3	0	1	1	0	0		0	q1	1	1	1	1
0	q3	0	0	1	1	0	0		0	1	q2	1	1	1
q3	0	0	0	1	1	0	0		0	1	1	q2	1	1
q3	B	0	0	0	1	1	0		0	1	1	1	q2	B
	q0	0	0	0	1	1	0		0	1	1	q4	1	1
	q1	0	0	1	1	0	0		0	1	q4	1	1	1
	0	q1	0	1	1	0	0		0	q4	1	1	1	1
	0	0	q1	1	1	0	0		q4	0				
	0	0	1	q2	1	0	0		q4	B	0			
	0	0	1	1	q2	0	0		0	q6	0			

Plan

- 1 Introduction
- 2 La machine de Turing (TM - 1936)
- 3 TM étendues et restreintes**
- 4 Classes des langages RE et R vs classe 0 et classe 1
- 5 Propriétés des langages R et RE

Extensions pour les TM

Si l'on restreint une TM à un ruban infini d'un seul côté, on ne diminue pas la classe de langages acceptés / calculés.

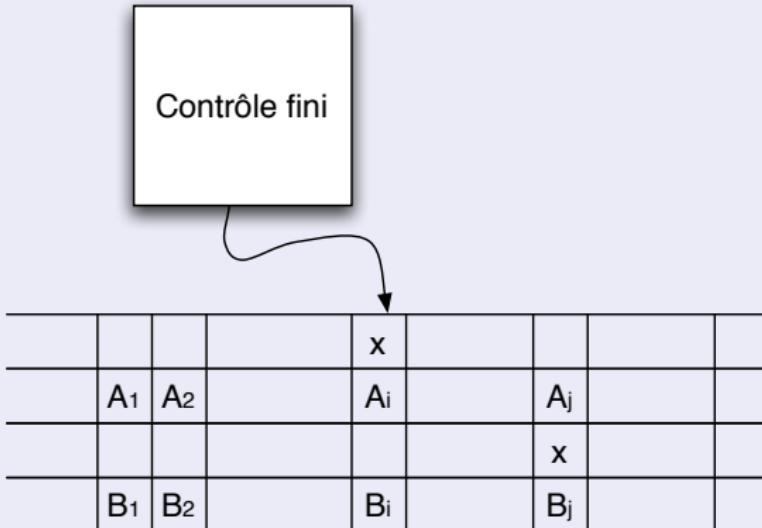
Aucune extension proposée n'augmente le pouvoir d'expressivité.

- TM avec une mémoire de stockage supplémentaire finie
- TM avec un ruban à plusieurs pistes
- TM avec sous-routines
- TM avec plusieurs rubans
- TM non déterministe

Traduction TM M_1 à k rubans en TM M_2 de base

On définit une TM à $2k$ pistes qui simule M_1 :

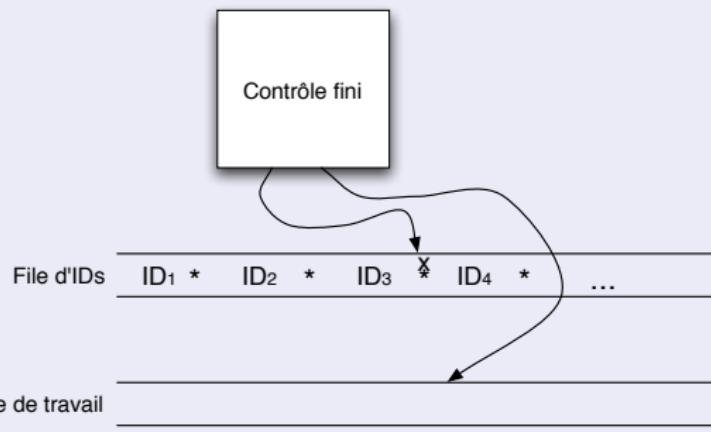
- k pistes pour mettre l'information des k rubans
- k pistes pour indiquer la position de la tête



Traduction TM M_1 non déterministe en TM M_2 de base

On définit TM_2 avec :

- une piste contenant les ID_i qu'il reste à “exécuter”
- un marqueur en plus sur cette piste pour déterminer l'ID suivant
- une piste de travail



Qui veut gagner un million de \$?

Il suffit de répondre à :

La classe des TM qui résolvent un problème en un temps polynomial dans la taille de l'input est-elle égale à la classe des TM **non déterministes** qui résolvent un problème en un temps polynomial dans la taille de l'input (\equiv problème dont on peut vérifier une solution en un temps polynomial)

$$\mathcal{P} \stackrel{?}{=} \mathcal{NP}$$

voir : http://www.claymath.org/millennium/P_vs_NP/

Actuellement on suppose que

$$\mathcal{P} \neq \mathcal{NP}$$

Plan

- 1 Introduction
- 2 La machine de Turing (TM - 1936)
- 3 TM étendues et restreintes
- 4 Classes des langages RE et R vs classe 0 et classe 1**
- 5 Propriétés des langages R et RE

Équivalence TM et langage de classe 0

Théorème (Les langages de classe 0 sont RE)

Preuve :

Ayant $G = \langle N, \Sigma, P, S \rangle$,
on construit une TM M non déterministe à 2 rubans qui accepte le même langage.

- le premier ruban contient le string d'input w
- le deuxième est utilisé pour contenir la forme phrasée α

Équivalence TM et langage de classe 0

Théorème ((suite) Les langages de classe 0 sont RE)

Fonctionnement de M :

Init Au départ on met S sur le deuxième ruban, ensuite la TM

- 1 choisit de façon non déterministe une position i dans α
- 2 choisit de façon non déterministe une production $\beta \rightarrow \gamma$ de G
- 3 si β est en position i dans α , remplace β par γ en shiftant ce qui suit (à gauche ou à droite)
- 4 compare la forme phrasée obtenue avec w sur le ruban 1 :
 - si cela correspond, w est accepté
 - sinon retour au point 1

Équivalence TM et langage de classe 0

Théorème (La classe RE est incluse dans la classe 0)

Principe de la preuve :

Ayant $M = \langle Q, \Sigma, \Gamma, \delta, q_0, B, F \rangle$,
on construit $G = \langle N, \Sigma, P, S \rangle$ avec $L(G) = L(M)$.

Inclusion stricte de la classe des langages de type 2 (context-free) dans la classe des langages de type 1 (context sensitive)

Rappels et propriétés

- Une grammaire context-sensitive a ses règles de la forme $\alpha \rightarrow \beta$ avec $|\alpha| \leq |\beta|$
- Un langage est context-sensitive s'il est défini par une grammaire context-sensitive
- Les langages context-free sont inclus dans les langages context-sensitive
- On peut montrer qu'il existe des langages context-sensitive qui ne sont pas context-free (l'inclusion est donc stricte)

Exemple ($L_{02i} = \{0^{2^i} \mid i \geq 1\}$ est context-sensitive mais pas context-free)

Une grammaire pour L_{02i} :

- | | |
|---|--|
| <ul style="list-style-type: none">① $S \rightarrow DF$② $S \rightarrow DH$③ $DH \rightarrow 00$④ $D \rightarrow DAG$ | <ul style="list-style-type: none">① $GF \rightarrow AF$② $AF \rightarrow H0$③ $AH \rightarrow H0$④ $GA \rightarrow AAG$ |
|---|--|

Langages R versus classe 1 (context sensitive)

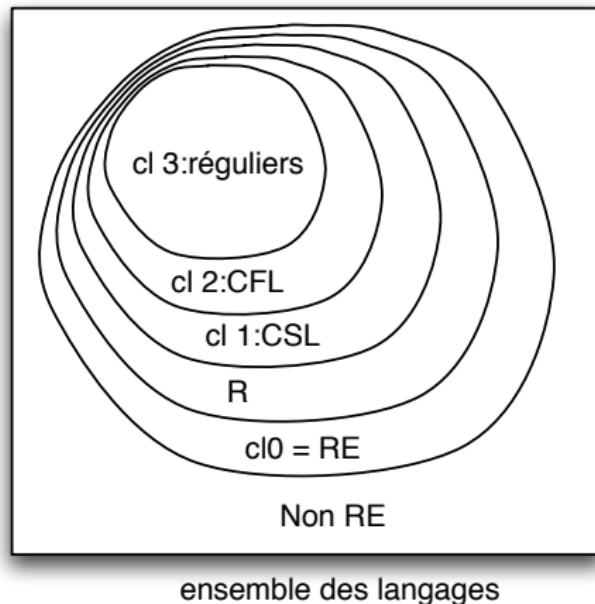
R vs context sensitive

On peut montrer que :

- Tout langage context sensitive (classe 1) est récursif.
 - Ayant $G = \langle N, \Sigma, P, S \rangle$ une grammaire avec les règles $\alpha \rightarrow \beta$ où $|\beta| \geq |\alpha|$, et w ,
 - on peut construire le graphe d'accessibilité à partir de S à toute forme phrasée de taille $\leq |w|$.
 - et déterminer l' "accessibilité" $S \xrightarrow{*} w$
- Certains langages récursifs ne sont pas de classe 1 (pas démontré ici)

Inclusion des classes de langages

En résumé nous avons vu que :

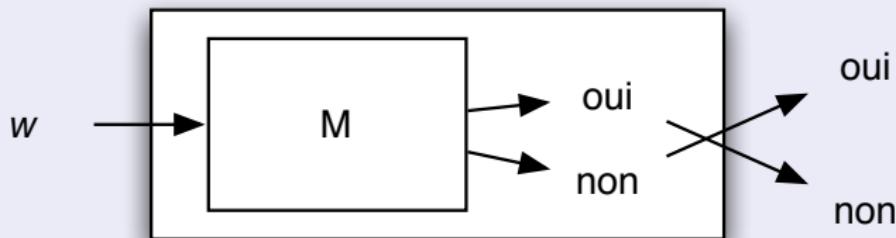


Plan

- 1 Introduction
- 2 La machine de Turing (TM - 1936)
- 3 TM étendues et restreintes
- 4 Classes des langages RE et R vs classe 0 et classe 1
- 5 Propriétés des langages R et RE

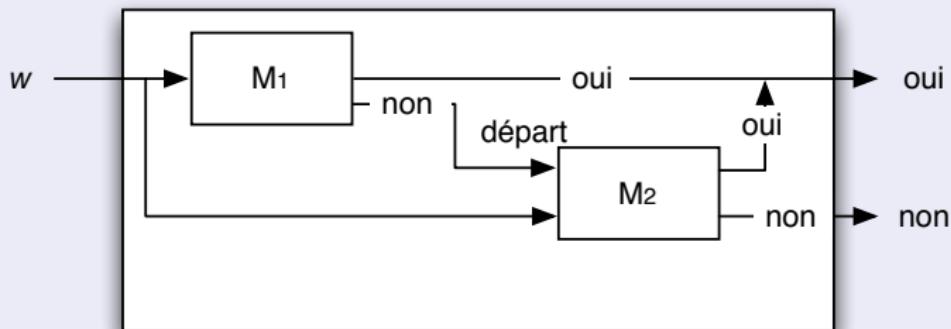
Théorème (Le complément d'un langage R est R)

Preuve :



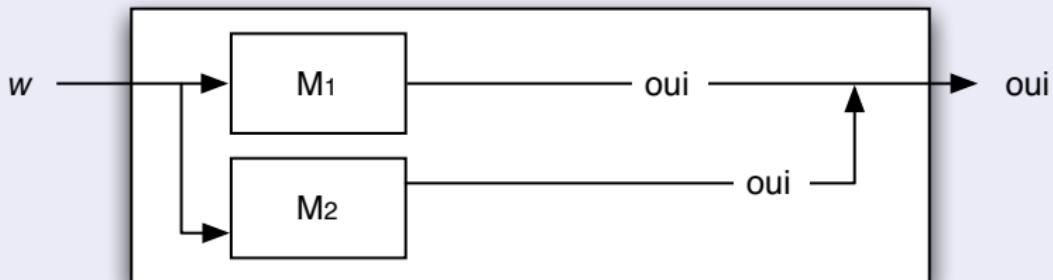
Théorème (L'union de 2 langages R est R)

Preuve :



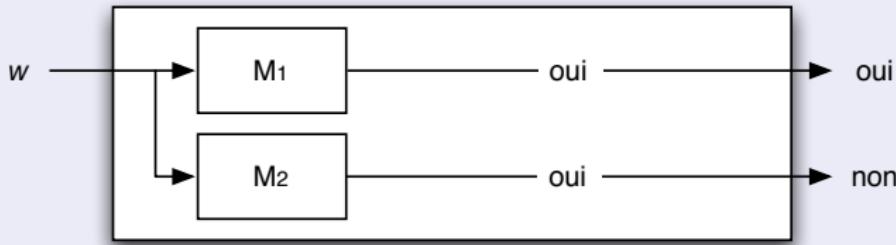
Théorème (L'union de 2 langages RE est RE)

Preuve :



Théorème (Si L et \bar{L} sont RE alors L (et \bar{L}) sont R)

Preuve :



L et \bar{L} sont :

soit tous 2 sont R

soit aucun des 2 n'est RE

soit l'un RE et l'autre non RE

Chapitre 14 : Eléments de décidabilité

- 1 Problèmes indécidables
- 2 Un exemple de problème indécidable
- 3 Technique de réduction
- 4 Codification (de Gödel) des TM et langages non RE
- 5 Langages et propriétés indécidables

Plan

- 1 Problèmes indécidables
- 2 Un exemple de problème indécidable
- 3 Technique de réduction
- 4 Codification (de Gödel) des TM et langages non RE
- 5 Langages et propriétés indécidables

Problèmes indécidables

Un exemple de problème indécidable

Technique de réduction

Codification (de Gödel) des TM et langages non RE

Langages et propriétés indécidables

Problème que les ordinateurs ne savent résoudre

Exemple (Le programme suivant imprime-t-il “hello world” ?)

```
main()
{
    printf(''hello world\n'');
}
```

Problème que les ordinateurs ne savent résoudre

Exemple (Le programme suivant imprime t-il "hello world" pour un n donné (en supposant qu'il n'y a pas de limitation de stockage))

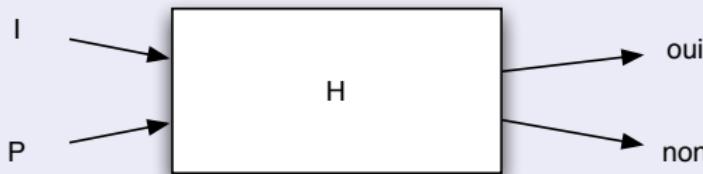
```
int exp(inti, n){  
    int ans, j;  ans = 1;  
    for (j=1;j<n; j++) ans *=i;  
    return ans;  
}  
main(){  
    int n, total, x, y, z;  scanf(``%d'', &n);  total = 3;  
    while (1){  
        for (x=1;x<=total-2;x++)  
            for (y=1;y<=total-x-1; y++) {  
                z=total-x-y;  
                if (exp(x,n)+exp(y,n)==exp(z,n))  
                    printf(``hello world\n'');  return();  
            }  
        total++;  
    }  
}
```

Problème que les ordinateurs ne savent pas résoudre

Testeur de “hello world”

Peut-on construire un programme H qui

- ayant en entrée : un programme P et un input I
- détermine si P écrit “hello world” avec I comme entrée ?



Le problème : ayant P et I déterminer si P écrit “hello world”, **est indécidable**

Définition (Problème décidable)

- *Un problème est décidable s'il existe une procédure effective (un algorithme) pour le résoudre*
- *sinon le problème est indécidable*

Pourquoi les problèmes indécidables doivent exister

Raisonnement

- On s'intéresse à des programmes qui répondent “oui” ou “non”,
- C'est donc équivalent au problème de l'appartenance d'un string à un langage,
- L'ensemble des langages pour un alphabet fini (d'au moins deux caractères) est non dénombrable,
- L'ensemble des programmes (étant aussi vu comme une string d'alphabet fini (typiquement ascii) est dénombrable,
- ⇒ Un nombre infini continu de problèmes n'ont pas de programme pour le résoudre

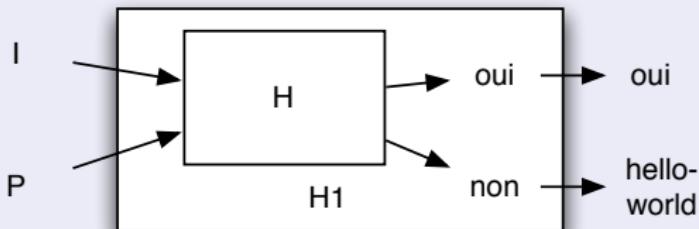
Plan

- 1 Problèmes indécidables
- 2 **Un exemple de problème indécidable**
- 3 Technique de réduction
- 4 Codification (de Gödel) des TM et langages non RE
- 5 Langages et propriétés indécidables

Le problème du testeur "hello world" est indécidable

Preuve par contradiction

Hypothèse H existe.
Alors on peut construire $H1$



H1 : reçoit P et I et répond

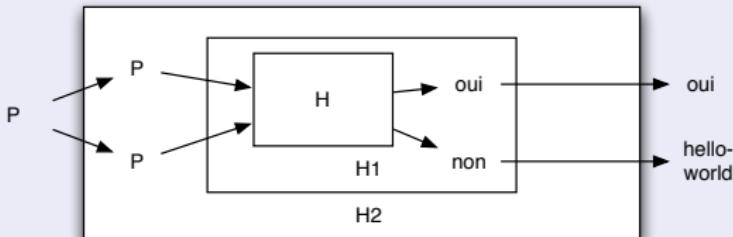
- “oui” si P avec I en entrée écrit “hello world”
- “hello world” si P avec I n’écrit pas “hello world”

Le problème du testeur "hello world" est indécidable

Preuve par contradiction

Hypothèse H existe.

Alors on peut construire $H1$ et $H2$

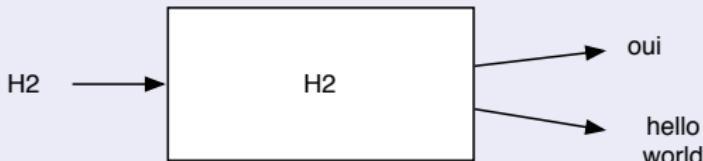


$H2$: reçoit P et répond

- “oui” si P avec P en entrée écrit “hello world”
- “hello world” si P avec P n’écrit pas “hello world”

Le problème du testeur “hello world” est indécidable

Que fait $H2$ avec en entrée $H2$?



Répond :

- “oui” si $H2$ avec en entrée $H2$ répond “hello world”
- “hello world” si $H2$ avec en entrée $H2$ ne répond pas “hello world”

D'où la contradiction : H ne peut exister

Problème décidable

Définition (Problème décidable)

*Problème qui a un **algorithme** pour le résoudre.*

Note

Un langage récursif peut donc être vu comme un langage tel qu'il existe un algorithme pour reconnaître ses mots

Hypothèse de Church ou thèse de Church-Turing

Les fonctions calculables (problèmes décidables) sont les problèmes qui peuvent être calculés par une TM, c'est-à-dire les langages récursifs.

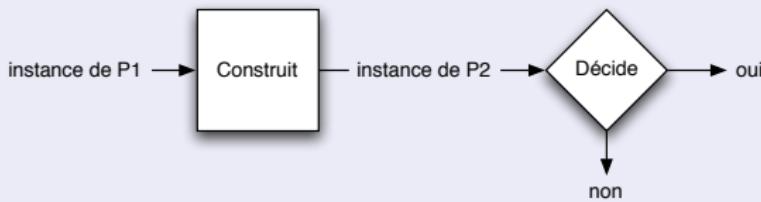
Plan

- 1 Problèmes indécidables
- 2 Un exemple de problème indécidable
- 3 **Technique de réduction**
- 4 Codification (de Gödel) des TM et langages non RE
- 5 Langages et propriétés indécidables

Réduction d'un problème à un autre

Réduction de P_1 à P_2

- On sait P_1 indécidable
- On veut prouver que P_2 est aussi indécidable



Réduction de P_1 en P_2

- On construit un programme qui transforme toute instance de P_1 en une instance de P_2
 - Si P_2 est décidable, alors P_1 aussi
- ⇒ P_2 est indécidable

Exemple de réduction

Exemple (Réduction du testeur de “hello world” en testeur de l’appel à une fonction *foo*)

- Transforme une instance de programme pour que s'il écrivait “hello world”, maintenant, il appelle la fonction *foo*
- Si le testeur d'appel à *foo* existe, le testeur “hello world” devrait exister
⇒ le testeur d'appel à *foo* est indécidable

Plan

- 1 Problèmes indécidables
- 2 Un exemple de problème indécidable
- 3 Technique de réduction
- 4 Codification (de Gödel) des TM et langages non RE
- 5 Langages et propriétés indécidables

Codification de Gödel

On suppose $M = (Q, \Sigma, \Gamma, \delta, q_1, B, F)$ avec

- $Q = \{q_1, q_2, \dots, q_r\}$
- q_1 est l'état initial q_2 est l'état accepteur (un suffit)
- $\Sigma = \{0, 1\}$
- $\Gamma = \{X_1, X_2, \dots, X_s\}$ avec $X_1 = 0$, $X_2 = 1$ et $X_3 = B$
- les directions $L = D_1$ et $R = D_2$

On code

- $\delta(q_i, X_j) = (q_k, X_\ell, D_m)$ par $C = 0^i 1 0^j 1 0^k 1 0^\ell 1 0^m$
- M par tous ses codes $111C_111C_211\dots11C_{n-1}11C_n111$

Donc

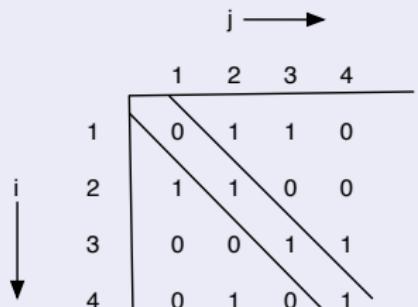
- toute TM M correspond à un string binaire $\langle M \rangle$.
- tout string binaire correspond à une TM (avec la convention que si le string ne respecte pas la codification, c'est une TM qui ne bouge pas)

Note

La codification de Gödel énumère toutes les TM, et on peut parler de M_i la $i^{\text{ème}}$ TM qui correspond à w_i le $i^{\text{ème}}$ string pour une codification déterminée.

Définition (L_d)

$$L_d = \{w_i \mid w_i \notin L(M_i)\}$$



	1	2	3	4
1	0	1	1	0
2	1	1	0	0
3	0	0	1	1
4	0	1	0	1

Diagonale $M_{ij} = 1$ ssi $w_i \in L(M_j)$

Théorème (L_d n'est pas RE)

Preuve :

- $\forall i : w_i \in L(M_i)$ ssi $w_i \notin L_d$
- *Donc pour tout $i : L(M_i) \neq L_d$*
- *Donc aucune TM n'accepte L_d*

Théorème (\bar{L}_d est RE (mais pas R))

Preuve :

- \bar{L}_d n'est pas R sinon L_d le serait aussi
- \bar{L}_d est RE car on peut construire une TM M (RE) qui simule M_i sur w_i

Plan

- 1 Problèmes indécidables
- 2 Un exemple de problème indécidable
- 3 Technique de réduction
- 4 Codification (de Gödel) des TM et langages non RE
- 5 Langages et propriétés indécidables

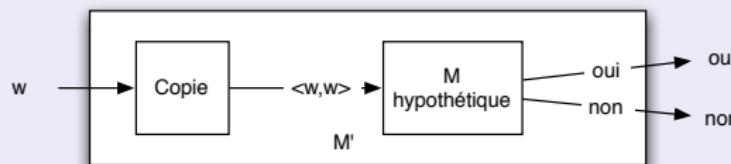
Définition (L_u)

$$L_u = \{ \langle M, w \rangle \mid M \text{ accepte } w \}$$

Théorème (L_u est RE mais pas R)

Preuve :

- L_u est RE : on peut construire une TM M_u qui simule M sur w
- L_u n'est pas R : montrons que \bar{L}_u n'est pas R (ce qui implique le résultat)
 - Hypothèse : la TM $M_{\bar{u}}$ "réursive" (càd qui s'arrête toujours) pour \bar{L}_u existe.
 - alors on peut construire M' "réursive" pour L_d (réduction de L_d vers \bar{L}_u)



⇒ contradiction de l'hypothèse que $M_{\bar{u}}$ existe.

L_e et L_{ne}

Définition (L_e et L_{ne})

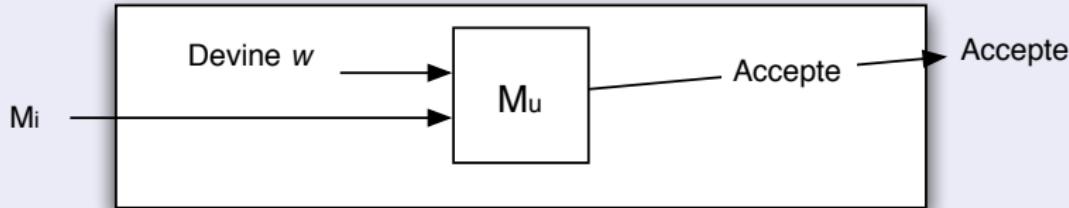
- $L_e = \{< M > \mid L(M) = \emptyset\}$
- $L_{ne} = \{< M > \mid L(M) \neq \emptyset\}$

Notons que L_e et L_{ne} sont des langages complémentaires.

Théorème (L_{ne} est RE)

Preuve :

On peut construire une NTM qui accepte L_{ne}



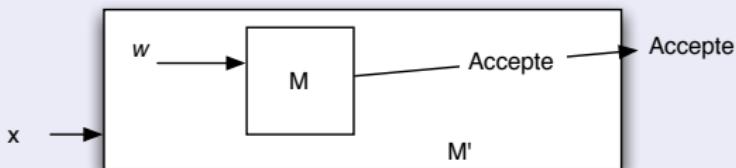
Machine pour L_{ne} utilisant la machine universelle M_u

L_e et L_{ne}

Théorème (L_{ne} n'est pas R)

Preuve par l'absurde : réduction de L_u à L_{ne}

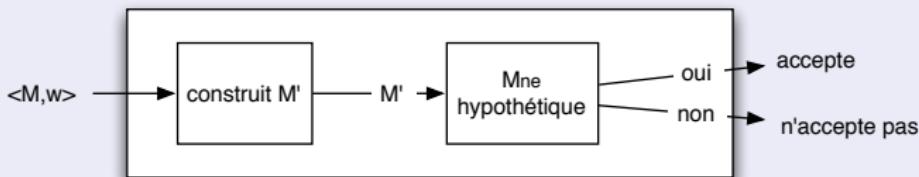
- Hypothèse : L_{ne} est récursif : une TM M_{ne} calcule L_{ne}
- Montrons que l'on peut alors construire une TM qui calcule L_u (ce qui est impossible)
- On peut construire un algorithme qui à partir de tout $\langle M, w \rangle$ construit M' avec
 - $L(M') = \Sigma^*$ si M accepte w
 - $L(M') = \emptyset$ si M n'accepte pas w



Machine M'

Théorème (L_{ne} n'est pas R (suite))

- la TM pour L_u utilisant M_{ne} est alors :



Machine hypothétique pour L_u

⇒ ce qui contredit l'hypothèse

Corollaire (L_e n'est pas RE)

Preuve :

L_{ne} est RE mais pas R

Théorème de Rice sur les langages RE

Note

L_e et L_{ne} sont indécidables = cas particulier du théorème de Rice : toute propriété non triviale sur les langages RE est indécidable càd : il est impossible de reconnaître les codes de TM dont le langage a la propriété

Définition (Propriété d'un langage RE)

Ensemble de langages RE

- Si l'ensemble des langages (ayant cette propriété) est vide ou contient tous les langages, on dit que la propriété est triviale
- sinon la propriété est non triviale

Note

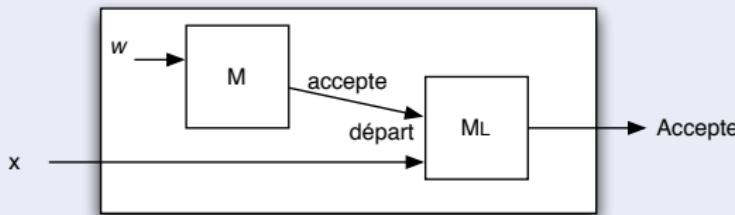
- Les TM M ont une description finie : $L(M)$ est souvent infini
- Pour une propriété \mathcal{P} sur les langages RE,
$$L_{\mathcal{P}} = \{< M > \mid L(M) \text{ a la propriété } \mathcal{P}\}$$
- “Décidabilité d'une propriété \mathcal{P} ” \equiv “décidabilité de $L_{\mathcal{P}}$ ”

Théorème de Rice

Théorème (Toute propriété \mathcal{P} non triviale des langages RE est indécidable)

Preuve : réduction de L_u à $L_{\mathcal{P}}$

- Hypothèse : \mathcal{P} est décidable
- Supposons que $\emptyset \notin \mathcal{P}$ (sinon on prend la proposition complémentaire)
- et qu'un certain langage $L \in \mathcal{P} (L \neq \emptyset)$
- Une TM M_L accepte L
- A partir de $\langle M, w \rangle$ on peut avoir un algorithme qui construit M' :



$L(M') = L$ si M accepte w et $L(M') = \emptyset$ si M n'accepte pas w .
Machine M' a la propriété ssi M accepte w

- On peut ainsi construire une TM qui calcule L_u ce qui est impossible

Conséquences du théorème de Rice

Corollaire (Propriétés indécidables sur les ensembles RE)

Si L est un langage RE les propriétés suivantes sont indécidables :

- L est vide
- L est fini
- L est régulier
- L est context free

未