Compilation

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Lectures recommandées - livres

1. « Engineering a Compiler » ©, K. D. Cooper & L. Torczon, Morgan Kaufmann 2004
2. « Introduction to Automata Theory Languages and Computation » ©, J. E. Hopcroft, R. Motwani, J. D.Ullman, 2001, Addison Wesley
3. « Compilers and Compiler Generators » ©, P.D. Terry, 2000
4. « Compilateurs avec C++ », Jacques Menu, 1994, Addison Wesley
5. « *Bottom-Up Parsing* », Maggie Johnson & Julie Zelenski, 2008
6. Cours « Machines virtuelles », Samuel Tardieu, ENST
7. Cours « Compilation et interprétation », Danny Dubé 2006
8. Cours « Compilation », C. Paulin (Université Paris Sud), 2009-2010

Plan du cours

1. Module initial Motivation & Définitions
2. Module 1 Analyseur lexical *(langages réguliers et théorie des automates)*
3. Module 2 Analyseur syntaxique *(langages hors contexte)*
4. Module 3 Analyseur sémantique *(grammaires attribuées)*
5. Module 4 Représentations Intermédiaires
6. Module 5 Génération de code intermédiaire
7. Module 6 Optimisation de code
8. Module 7 Génération de code
9. Module final Conclusion et perspectives

Compilation

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Lectures recommandées – livres & cours

1. Livres
   1. « Engineering a Compiler » ©, K. D. Cooper & L. Torczon, Morgan Kaufmann 2004
   2. « Introduction to Automata Theory Languages and Computation » ©, J. E. Hopcroft, R. Motwani, J. D.Ullman, 2001, Addison Wesley
   3. « Compilers and Compiler Generators » ©, P.D. Terry, 2000
   4. « Compilateurs avec C++ », Jacques Menu, 1994, Addison Wesley
   5. « *Bottom-Up Parsing* », Maggie Johnson & Julie Zelenski, 2008
2. Cours
   1. « Machines virtuelles », Samuel Tardieu, ENST
   2. « Compilation et interprétation », Danny Dubé 2006
   3. « Compilation », C. Paulin (Université Paris Sud), 2009-2010

Plan du cours

1. **Module initial** Motivation & Définitions
2. Module 1 Analyseur lexical *(langages réguliers et théorie des automates)*
3. Module 2 Analyseur syntaxique *(langages hors contexte)*
4. Module 3 Analyseur sémantique *(grammaires attribuées)*
5. Module 4 Représentations Intermédiaires
6. Module 5 Génération de code intermédiaire
7. Module 6 Optimisation et Génération de code
8. Conclusion et perspectives

Module initial
Motivation & Définitions

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Pourquoi la compilation ?

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Introduction

- Les micro-processeurs deviennent indispensables et sont embarqués dans tous les appareils que nous utilisons dans la vie quotidienne
  - Transport
    - Véhicules, Systèmes de navigation par satellites (GPS), Avions, …
  - Télécom
    - Téléphones portables, Smart Phones, …
  - Électroménager
    - Machine à laver, Micro-ondes, Lave vaisselles, …
  - Loisir
    - e-book, PDA, Jeux vidéo, Récepteurs, Télévision, TNT, Home Cinéma…
  - Espace
    - Satellites, Navettes, Robots d'exploration, …

- Pour fonctionner, ces micro-processeurs nécessitent des programmes spécifiques à leur architecture matérielle

Compilateur : Motivation

- Programmation en langages de bas niveau (proches de la machine)
  - **très difficile** (complexité)
    - Courbe d'apprentissage très lente
    - Débuggage fastidieux
  - **très coûteuse en temps** (perte de temps)
    - Tâches récurrentes
    - Tâches automatisables
  - **très coûteuse en ressource humaine** (budget énorme)
    - Tâches manuelles
    - Maintenance
  - **très ingrate** (artisanat)
    - Centrée sur les détails techniques et non sur les modèles conceptuels
  - **n'est pas à 100% bug-free** (fiabilité)
    - Le programmeur humain n'est pas parfait

Compilateur : Motivation

- Besoin continu de
  1. Langages de haut niveau
     - avec des structures de données, de contrôles et des modèles conceptuels proches de l'humain
  2. Logiciels de **génération** des **langages bas niveau** (propres aux microprocesseurs) à partir de **langages haut niveau** et vice versa
     - **Compilateurs**
     - **Interpréteurs**
     - **Pré-processeurs**
     - **Dé-compilateur**
     - **etc.**

Langages de programmation (∃ *plus de 150 langages*)

DATE	NOM	DESCRIPTION
1951	ASSEMBLEUR	Programmation Microprocesseur
1952	AUTOCODE	
1954	FORTRAN	
1955	FLOW-MATIC	
1957	COMTRAN	
1958	LISP	Programmation Fonctionnelle (& Déclarative)
1958	ALGOL 58	
1959	FACT	
1959	COBOL	
1962	APL	
1962	Simula	
1964	BASIC	
1964	PL/I	
1970	Pascal	
1970	Forth	Programmation à Pile
1972	C	Programmation Fonctionnelle
1972	Smalltalk	Précurseur de la Programmation MVC
1972	Prolog	Programmation Logique (& Déclarative)
1973	ML	

Langages de programmation (∃ *plus de 150 langages*)

•	**1978** SQL	Programmation Relationnelle (& Déclarative)
•	1983 Ada	Programmation Multi-thread
•	**1983** C++	Programmation Orientée Objet
•	1985 Eiffel	
•	1987 Perl	
•	1989 FL (Backus)	
•	1990 Haskell	Programmation Fonctionnelle Pure
•	1991 Python	
•	**1991** Java	
•	**1993** Ruby	
•	1993 Lua	
•	1994 Common Lisp	
•	1995 JavaScript	
•	**1995** PHP	Programmation CGI
•	**2000** C#	
•	2008 JavaFX Script	
•	2009 Go	
•	2009 Myrtryl	
•	…	

Etude de cas : UNIX

- UNIX est né aux *Laboratoire Bell* (1969)
- *K. Thompson* l'écrit d'abord en **Langage Machine**
- *J. Kernighan* le réécrit en **Assembleur**
- *D. Ritchie* implante le **Langage C** (1972)
- UNIX est réécrit en **Langage C** (1973) et reste très intimement lié a ce langage

Qu'est ce que la compilation ?

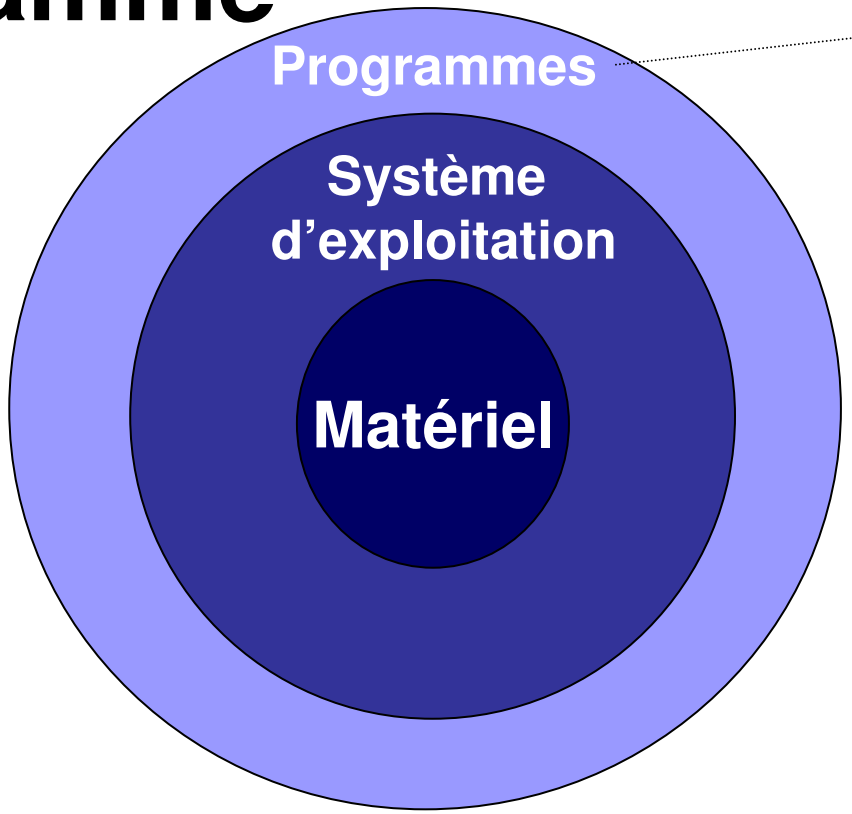
Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Positionnement des compilateurs

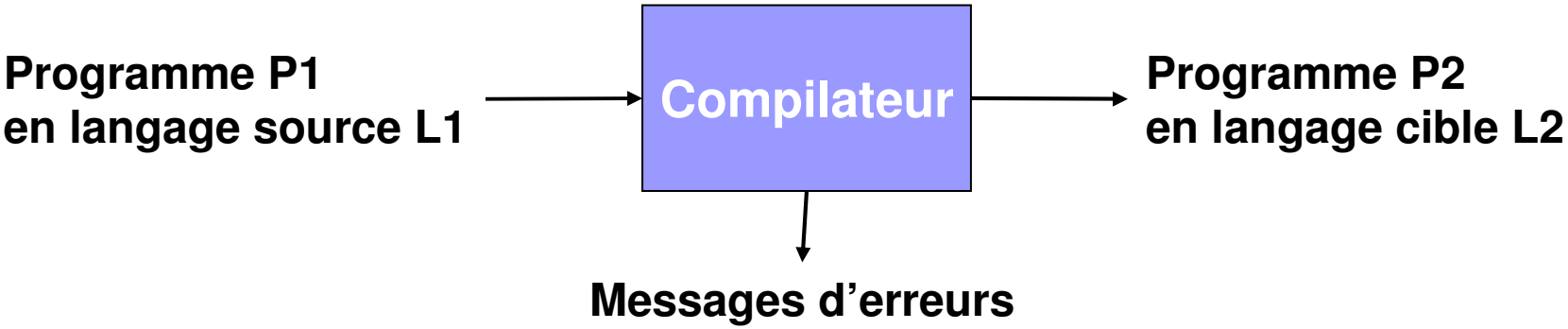
- **Un compilateur est un programme**



Utilitaires, BDD, Compilateurs, etc.

Compilateur - définition

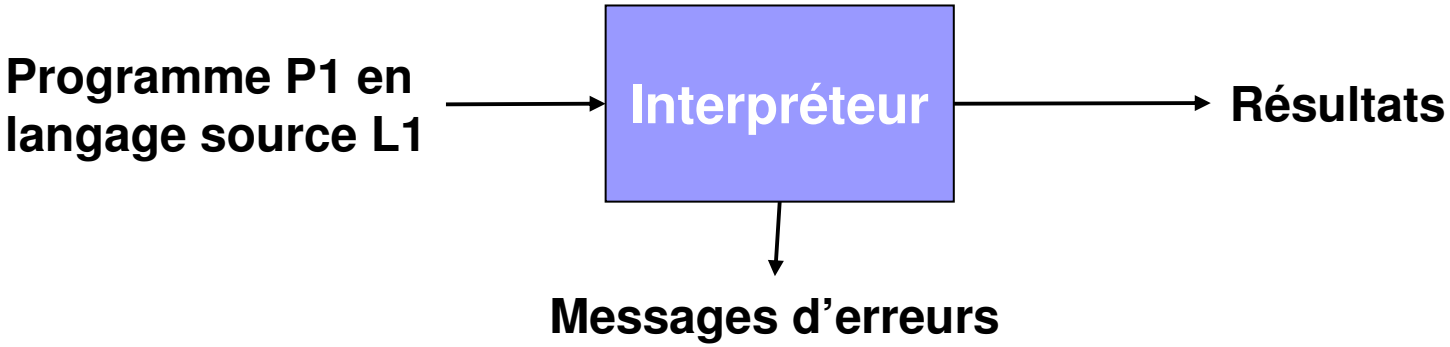
- Un compilateur est un logiciel (une *fonction*) qui prend en entrée un programme **P1** dans un langage source L1 et produit en sortie programme équivalent **P2** dans un langage L2



- Exemple de compilateurs :
  - C pour Motorola, Ada pour Intel, C++ pour Sun, doc vers pdf, ppt vers Postscript, pptx vers ppt, Latex vers Postscript, etc.

Interpréteur - définition

- Un interpréteur est logiciel qui prend en entrée un programme **P1** dans un langage source L1 et produit en sortie les **Résultats** de l'exécution de ce programme



- Exemples d'interpréteurs :
  - Batch DOS, Shell Unix, Prolog, PL/SQL, Lisp/Scheme, Basic, Calculatrice programmable, etc.

Dé-compilateur

- Dé-Compilateur est un compilateur dans le sens inverse d'un compilateur (depuis le langage bas niveau, vers un langage haut niveau)
- Applications :
  - Récupération de vieux logiciels
    - Portage de programmes dans de nouveaux langages
    - Recompilation de programmes vers de nouvelles architectures
  - Piratage
    - Compréhension du code d'un algorithme
    - Compréhension des clefs d'un algorithme de sécurité

Comment fonctionne un compilateur ?

Question

- Quels sont les constituants d'un langage naturel ?

- Si l'on veut programmer un traducteur de l'Anglais vers le Français que proposeriez-vous comme algorithme ?

Quelques éléments de réponses

- Quelques constituants d'un langage naturel ?
  - Vocabulaire (lexique), Syntaxe (grammaire), Sémantique (sens selon le contexte)

- Un macro-algorithme pour traduire de l'Anglais vers le Français que proposeriez-vous comme ?
  1. Analyser les mots selon le dictionnaire Anglais
  2. Analyser la forme des phrases selon la grammaire de l'Anglais
  3. Analyser le sens des phrases selon le contexte des mots dans la phrase anglaise
  4. Traduire le sens des phrases dans la grammaire et le vocabulaire du Français

Étapes et Architecture d'un compilateur

1. Compréhesion des entrées (**Analyse**)
   - analyse des éléments lexicaux (lexèmes : mots) du programme : **analyseur lexical**
   - analyse de la forme (structure) des instructions (phrases) du programme : **analyseur syntaxique**
   - analyse du sens (cohérence) des instructions du programme : **analyseur sémantique**

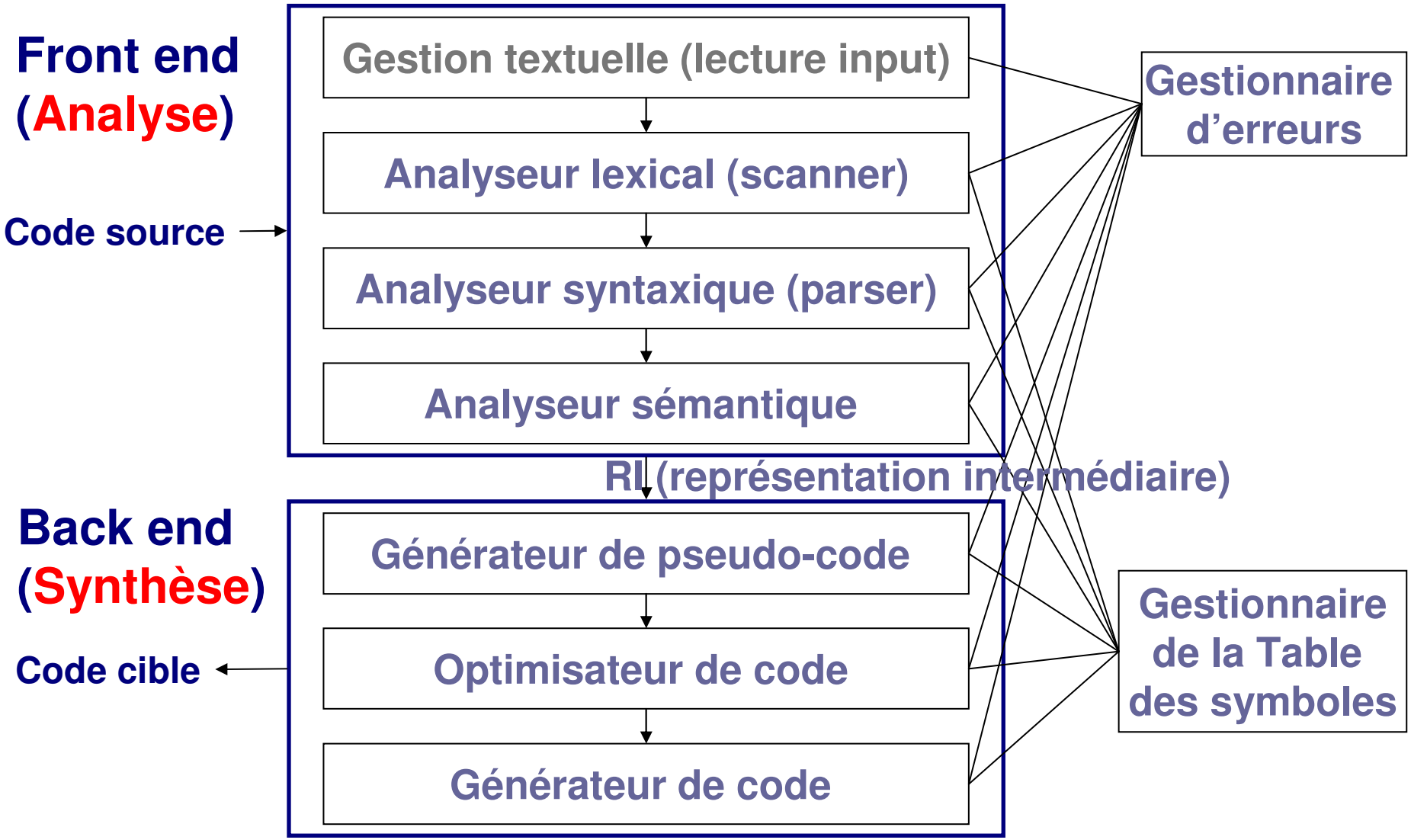
2. Préparation de la génération (**Synthèse**)
   - génération d'un code intermédiaire (nécessitant des passes d'optimisation et de transcription en code machine) : **générateur de pseudo-code**
   - optimisation du code intermédiaire : **optimisateur de code**

3. Génération finale de la sortie (**Synthèse suite**)
   - génération du code cible à partir du code intermédiaire : **générateur de code**

Étapes et Architecture d'un compilateur

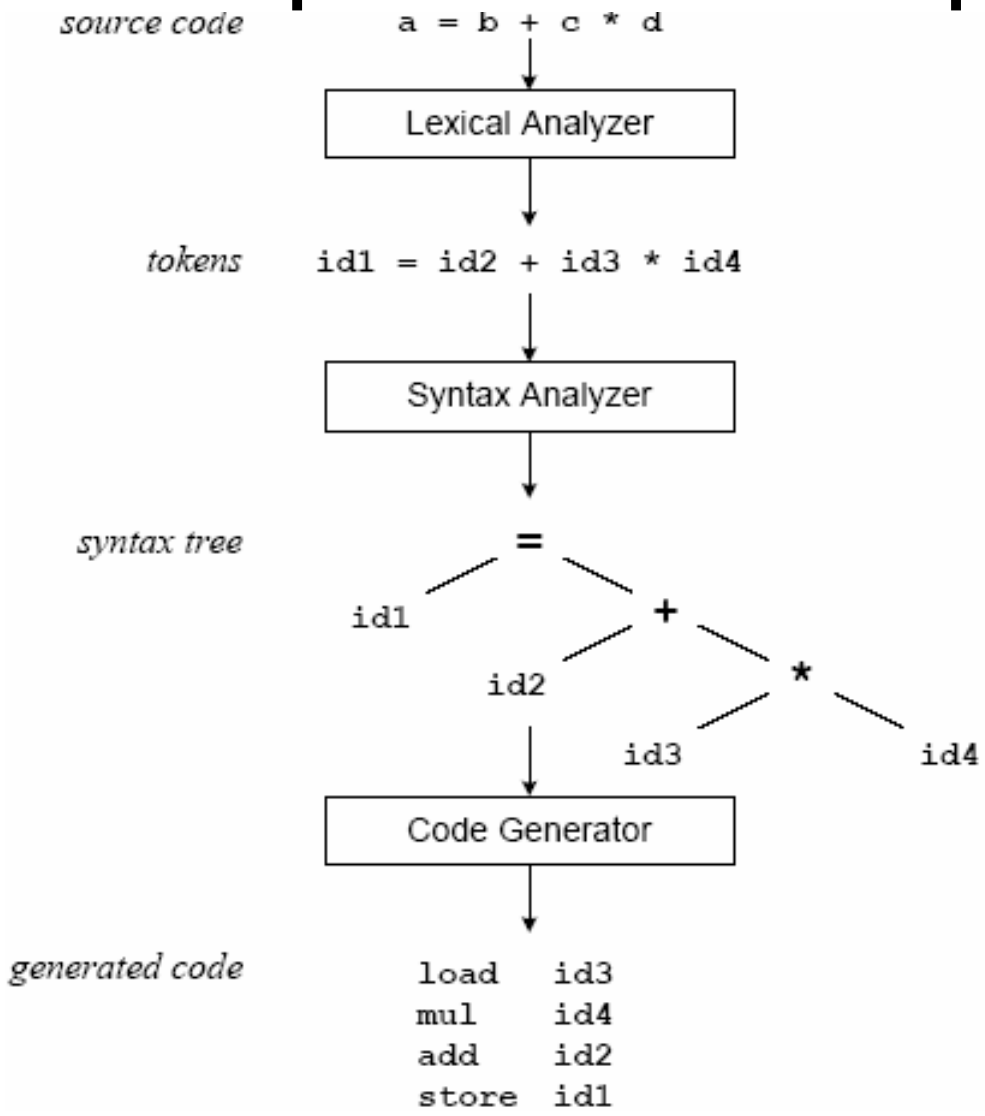
4. Fonctionnalités et briques annexes
   - gestion les erreurs et guide le programmeur pour corriger son programme : **gestionnaire d'erreurs**
   - gestion du dictionnaire des données du compilateur (symboles réservés, noms des variables, constantes) : **gestionnaire de la table des symboles**

Architecture d'un compilateur



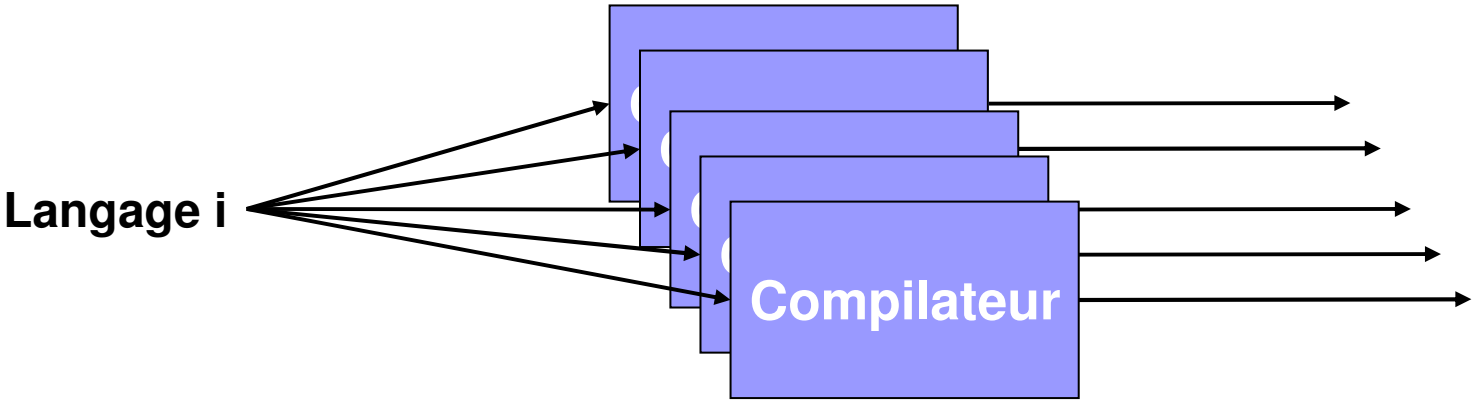
NB. Les langages de programmations restent nettement plus simple à traiter que les langages naturels !!

Exemple simplifié de compilation



Compilateur - réflexion

- Pour le même langage avancé, il faut n compilateurs différents, n étant le nombre de microprocesseurs différents



- Vue la stabilité des langages avancés, et vus les progrès continus en technologie matérielle, n ne sera presque jamais borné !! Il y aura toujours des compilateurs à écrire ;-)
- La structure, lexicale-syntaxique-sémantique du langage source ne changeant pas, les compilateurs ne différeront pas dans leur composants d'analyse, mais plutôt dans leur composants de synthèse

Compilation

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Lectures recommandées - livres

1. « Engineering a Compiler » ©, K. D. Cooper & L. Torczon, Morgan Kaufmann 2004
2. « Introduction to Automata Theory Languages and Computation » ©, J. E. Hopcroft, R. Motwani, J. D.Ullman, 2001, Addison Wesley
3. « Compilers and Compiler Generators » ©, P.D. Terry, 2000

Plan du cours

1. Module initial Motivation & Définitions
2. **Module 1** Analyseur lexical *(langages réguliers et théorie des automates)*
3. Module 2 Analyseur syntaxique *(langages hors contexte)*
4. Module 3 Analyseur sémantique *(grammaires attribuées)*
5. Module 4 Représentations Intermédiaires
6. Module 5 Génération de code intermédiaire
7. Module 6 Optimisation et Génération de code
8. Conclusion et perspectives

Module 1
Analyseur lexical

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Analyseur lexical - Plan

- Introduction
- Concepts
  - Alphabets
  - Langages
  - Expressions régulières : ER
  - Automates – déterministes : DFA, non déterministes : NFA et ε-NFA
- Transformations
  - DFA en programme
  - NFA en DFA et ε-NFA en DFA (déterminisation)
  - ER en ε-NFA (algorithme de Thompson)
  - ER en DFA (algorithme de Glushkov)
  - DFA en DFA minimal (minimisation par séparation des états)
  - ER en DFA minimal (minimisation par calcul des résiduels à gauche)
  - DFA en ER (par recherche de chemin, par équations linéaires)

Analyseur lexical - définitions

- **L'analyseur lexical (ou scanner)** fusionne les caractères lus du code source en groupes de lettres qui forment logiquement des **unités lexicales (**ou **tokens) du** langage
  - Symboles : identificateurs, chaînes, constantes numériques,
  - Mots clefs : while, if, then
  - Opérateurs : <=, :=, =

- Certains tokens sont simplement représentés par l'analyseur lexical, d'autres nécessitent une association avec d'autres propriétés telles leur nom, valeur, etc.

Analyseur lexical - exemple

- **Input** de l'analyseur lexical (code source en langage Modula-2)
  - WHILE A > 3 * B DO A := A - 1 END

- **Output** de l'analyseur lexical

Token (lexème)	***Type du Token (unité lexicale)***	***Propriété***
WHILE	*keyword*	
A	*identifier*	*name A*
>	*operator*	*comparison*
3	*constant literal*	*value 3*
*****	*operator*	*multiplication*
B	*identifier*	*name B*
DO	*keyword*	
A	*identifier*	*name A*
:=	*operator*	*assignment*
A	*identifier*	*name A*
-	*operator*	*substraction*
1	*constant literal*	*value 1*
END	*keyword*	

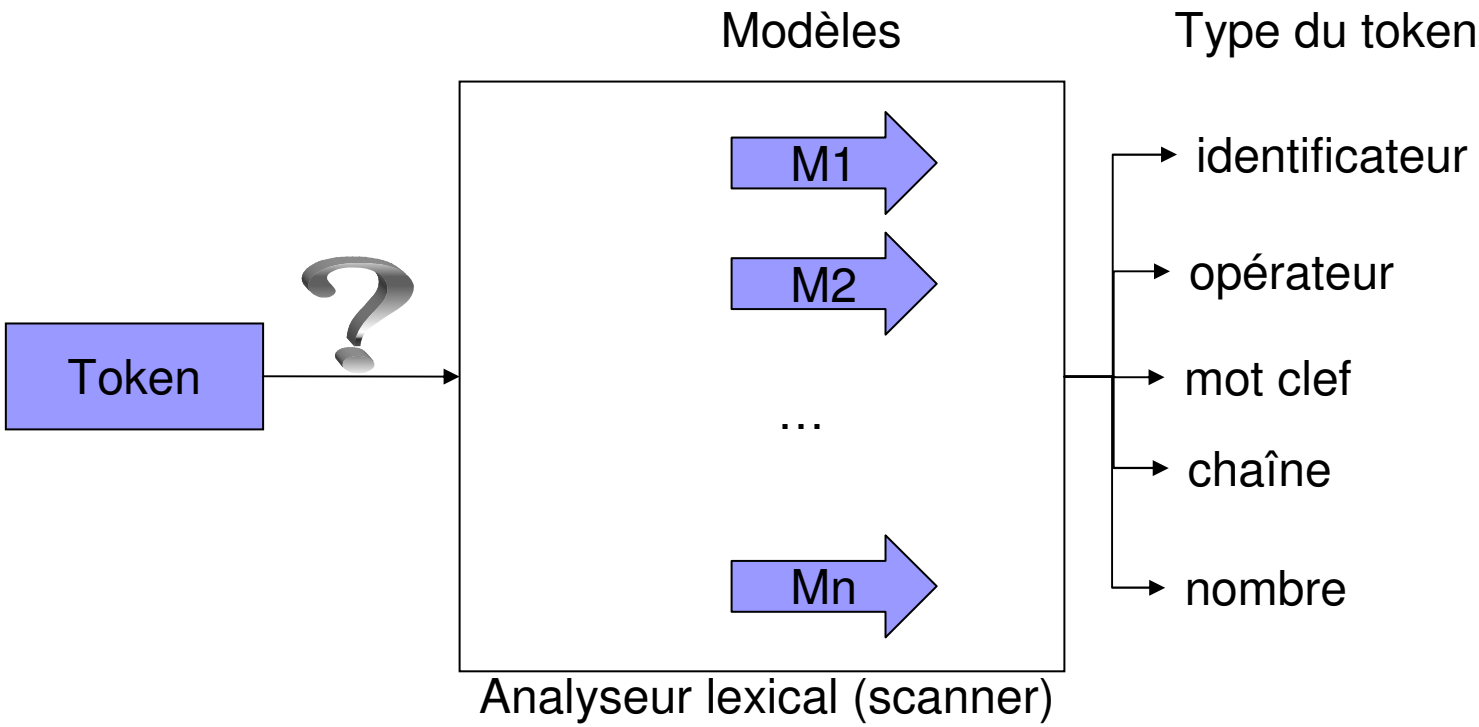
Analyseur lexical – table des symboles

- **La table des symboles résultante**

Numéro de symbole	*Token*	*Type du Token*	*Propriété*
10	**A**	*identifier*	*name A*
11	**B**	*identifier*	*name B*
…			
100	**>**	*operator*	*comparison*
101	**:=**	*operator*	*assignment*
102	**-**	*operator*	*substraction*
103	*****	*operator*	*multiplication*
…			
1000	**WHILE**	*keyword*	
1001	**DO**	*keyword*	
1002	**END**	*keyword*	
…			
5000	**1**	*constant literal*	*value 1*
5002	**3**	*constant literal*	*value 3*

- Ainsi l'instruction peut être réécrites ainsi :
  - 1000, 10, 100, 5002, 1001, 10, 101, 10, 102, 5000, 1002

Analyseur lexical - fonctionnement



Analyseur lexical - fonctionnement

- Les meilleurs modèles qui existent pour identifier les types lexicaux de tokens sont les expressions régulières
  - Alphabétique : (‘a’ | ‘b’ | … | ‘z’ | ‘A’ | ‘B’ | … | ‘Z’)
  - Numérique : (0 | …| 9)
  - Opérateurs : (+ | - | / | * | = | <= | >= | < | >)
  - Naturel : Numérique+
  - Entier : (+ | - | ε) Naturel
  - Identificateur : (Alphabétique(Alphabétique|Numérique)*)
  - ChaîneAlphaNumérique : " (Alphabétique | Numérique)+ "

Langages Réguliers et Théorie des automates - Rappels

Prof. K. Baïna 2010 ©, 7ème édition

Alphabet

- On appelle ***alphabet*** Σ un ensemble fini non vide. Les éléments d'un alphabet sont appelés ***lettres***

- Exemple d'alphabets
  - Σ = {a, b,.., z}
  - Σ = {α, β, δ, .., φ, +, *, /, =}
  - Σ = {if, then, else, while, begin, end}

Langage

- Soit *n* un entier naturel, on appelle **mot de longueur *n*** sur l'alphabet Σ toute application de *{1, 2,…n}* dans Σ.

- **Le mot vide est noté** ε

- **l'ensemble de tous les mots** sur un alphabet Σ est noté Σ*

Alphabet Σ	Mots de Σ*
Σ = {a, b,.., z}	« abab », « surf », « cascade », …
Σ = {α, β, δ, .., φ, +, *, /, =}	« α = β*2+φ », « +α δ=* », …
Σ = {if, then, else, while, begin, end}	(if,then,else), (begin,begin,end)

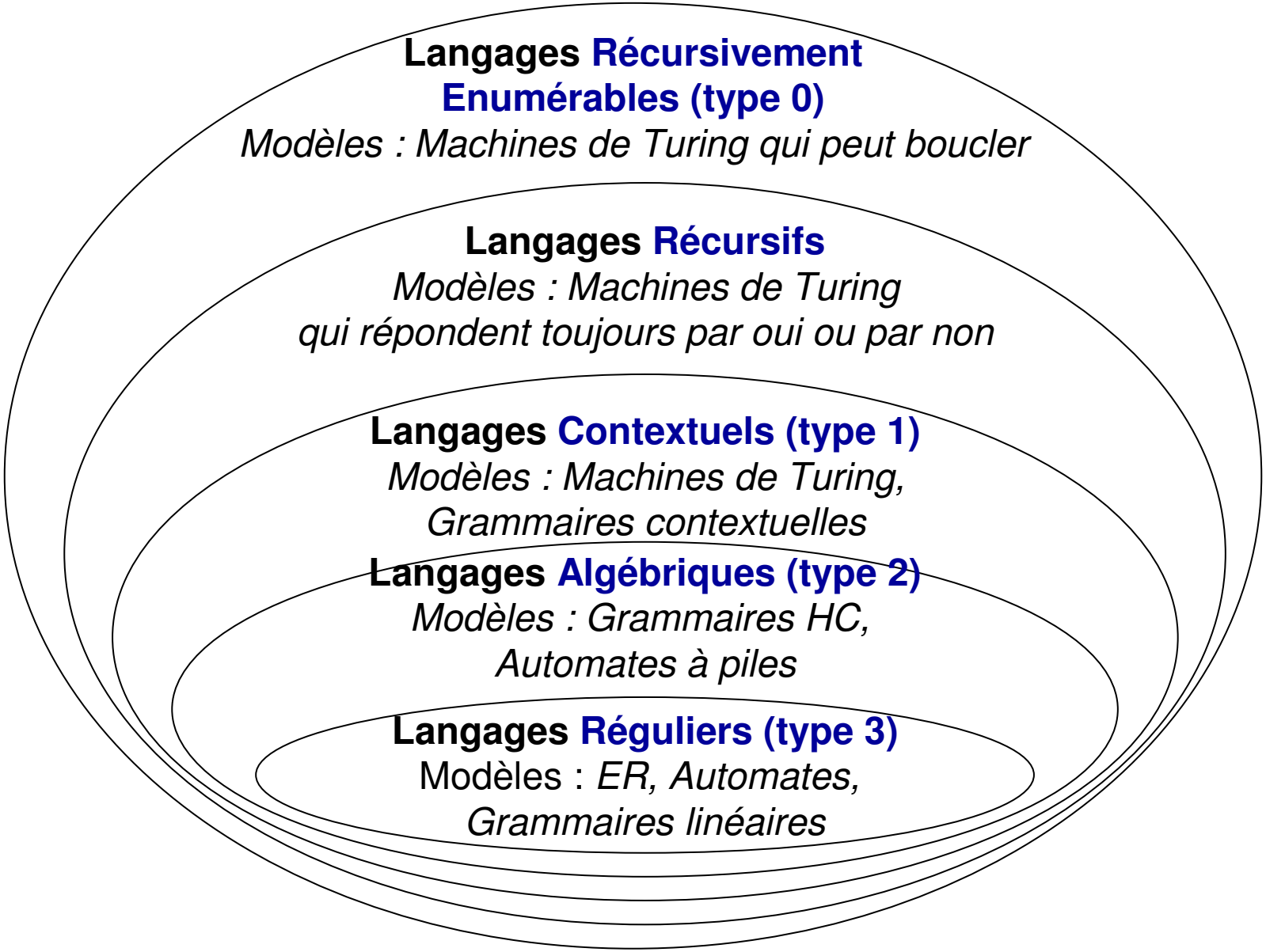
Langage

- Un **langage** sur Σ, noté $L(\Sigma)$ est une partie de Σ^* : $L(\Sigma) \subseteq \Sigma^*$

- Exemple de langages sur $\Sigma = \{a, b, .., z\}$
  - $L0 = \emptyset$, $L1 = \{\varepsilon\}$, $L2 = \{awa \mid w \in \Sigma^*\}$,
  - $L3 = \{waw \mid w \in \Sigma^*\}$,
  - $L4 = \{w \in \Sigma^* \mid |w|_a \leq 10\}$

  - La fonction $|.|_{x \in \Sigma} : \Sigma^* \rightarrow \mathbb{N}$
    - calcule le nombre d'occurrence de la lettre x de Σ dans un mot de Σ^*

Types de Langage



Lexique et expressions régulières

- Les meilleurs modèles définissant les unités lexicales (*types tokens*) auxquelles appartiennent les lexèmes (*tokens*) sont des langages particuliers (langages réguliers)

- Il existe plusieurs manières de décrire les langages réguliers
  - les expressions régulières
  - Les automates d'états finis
  - Les grammaires linéaires

Expressions régulières (ER)

- Soit Σ un alphabet, une expression régulière est définie comme suit :
  - **(1)** les éléments de Σ, **(2)** ε (mot vide) et **(3)** $\varnothing$ (ensemble vide) sont des expressions régulières.
  - Si α et β sont des expressions régulières, alors
    - **(4)** $(\alpha \mid \beta)$ représentant l'union de α **et** β,
    - **(5)** $(\alpha\ \beta)$ représentant la concaténation de α **et** β
    - **(6)** α^* représentant la fermeture de Kleene (répétition un nombre de fois éventuellement égal à 0)
    - **Sont des** expressions régulières
  - Nulle autre expression n'est une expression régulière

Expressions régulières (ER)

- associativité
  - $\alpha\ (\beta\gamma) = (\alpha\beta)\ \gamma$
  - $\alpha \mid (\beta\mid\gamma) = (\alpha \mid \beta) \mid \gamma$
- distributivité
  - $\alpha\ (\beta \mid \gamma) = \alpha\beta \mid \alpha\gamma$
- Autres propriétés
  - $\alpha \mid \varnothing = \varnothing \mid \alpha = \alpha$ (élément neutre par rapport à l'union)
  - $(\varnothing\ \alpha) = (\alpha\ \varnothing) = \varnothing$ ($\varnothing$ est l'élément absorbant pour la concaténation)
  - $(\varepsilon\ \alpha) = (\alpha\ \varepsilon) = \alpha$ (ε est l'élément neutre pour la concaténation)
  - $(\alpha \mid \alpha) = \alpha$
  - $\varnothing^* = \varepsilon$
  - $\varepsilon \mid \alpha\alpha^* = \alpha^*$
- Ecritures
  - $\alpha \mid \beta = \alpha + \beta$
  - Fermeture positive : l'écriture $\alpha+ = \alpha\ \alpha^*$ (répétition un nombre de fois au moins égal à 1)

Expressions régulières (ER)

- $\alpha(\beta\alpha)^* = (\alpha\beta)^*\alpha$
- $a(ba)^* = (ab)^*b$

- $(\alpha+\beta)^* = (\alpha^*+\beta^*)^* = (\alpha^*\beta^*)^* = (\alpha^*\beta)^*\alpha^* = \alpha^*(\beta\alpha^*)^*$

- À prouver en exercice !

Expressions régulières (ER)

- Exemples d'ER :
  - Alphabétique = ('a' | 'b' | … | 'z' | 'A' | 'B' | … | 'Z')
  - Numérique = (0 | …| 9)
  - Opérateurs = (+ | - | / | * | = | <= | >= | < | >)
  - Naturel = Numérique+
  - Entier = (+ | - | ε) Naturel
  - Identificateur = (Alphabétique(Alphabétique|Numérique)*)

Autres Applications des ER

- **Recherche** de information
  - textuelle, arborescente, relationnelle, hypertextuelle
- **Codage** / **décodage** de l'information
  - *Théorie des codes*
  - *Cryptage* de texte
  - *Génétique* (Décodage du génôme humain)
- **Compression** de information
- etc.

Prouver qu'un Langage est irrégulier

- **Lemme de l'étoile (pumping lemma):**
  - **Si** un langage ***L*** sur un alphabet Σ ($L \subseteq \Sigma^*$) est **régulier**,
    - $\Rightarrow \exists\, n \in IN^*$ (qui dépend de ***L***) tel que
    - $\forall\, u \in L$ avec $|u| \geq n$, il existe une décomposition de ***u*** en trois parties ***u = fgh*** telle que :
      1. $g \neq \varepsilon$
      2. $|fg| \leq n$
      3. $\forall\, k \geq 0,\ fg^k h \in L$

Prouver qu'un Langage est irrégulier

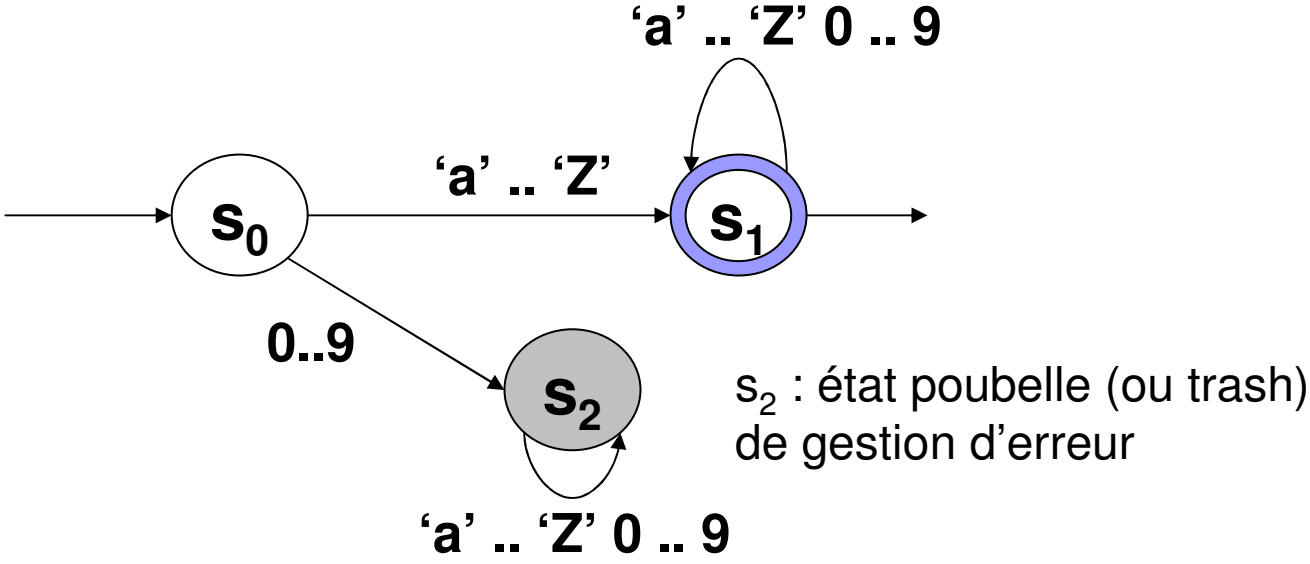
- **Soit l'alphabet** $\Sigma = \{a, b\}$
- **Le langage L = a+ est régulier (par construction) alors**
  - $\forall u \in L, |u| \geq 1 \Rightarrow$
    - $u = fgh, f=a^*, g=a, h=a^*, \ g \neq \varepsilon \wedge \forall k \geq 0, fg^kh = a^*(a)^ka^* \in L$
- **Le langage L = a*b* est régulier (par construction) alors**
  - $\forall u \in L, |u| \geq 1 \Rightarrow$
    - $u = fgh, f=a^*, g=a^i, h=a^*b^*, \ g \neq \varepsilon \wedge \forall k \geq 0, fg^kh = a^*(a^i)^ka^*b^* \in L$
- **Le langage L = a^mb^m, m $\in$ IN n'est pas régulier, car**
  - $\forall u \in L, \forall n \geq 2, |u| \geq n \Rightarrow$
    - $u = fgh, f=a^{m-i}, g=a^i, h=b^m, \ g \neq \varepsilon \wedge \exists k \geq 0, fg^kh = a^{m-i}(a^i)^kb^m \notin L$
    - $u = fgh, f=a^{m-i}, g=a^ib^i, h=b^{m-j}, \ g \neq \varepsilon \wedge \exists k \geq 0, fg^kh = a^{m-i}(a^ib^j)^kb^{m-j} \notin L$
    - $u = fgh, f=a^m, g=b^j, h=b^{m-j}, \ g \neq \varepsilon \wedge \exists k \geq 0, fg^kh = a^m(b^j)^kb^{m-j} \notin L$

Expressions régulières (ER)

- Les expressions régulières sont beaucoup plus puissantes que ce dont on a besoin lorsqu'on fait de l'analyse lexicale
- Chaque expression régulière ***R*** correspond à une machine abstraite qui reconnaît ***L*(*R*)** le langage définit par ***R***
- On appelle ces machines *automates d'états finis*

Automates et ER

- Alphabétique =(‘a’ | … | ‘z’ | ‘A’ | … | ‘Z’)
- Numérique =(0 | …| 9)
- Identificateur = (Alphabétique(Alphabétique|Numérique)*)



s_2 : état poubelle (ou trash) de gestion d’erreur

Automate correspondant aux ***identificateurs***

Automates d'états finis DFA - Deterministic Finite Automata

- Un automate d'états finis A est défini comme un 5-uplet **A = <S, Σ, δ, s_0, F >** où :
  - **S** est un ensemble d'états,
  - Σ est un alphabet (un ensemble de caractères),
  - δ **: S** $\times \Sigma \rightarrow$ **S** est la fonction de transition de l'automate,
  - s_0 est l'état initial de l'automate,
  - **F** $\subseteq$ **S** est l'ensemble des états finaux de l'automate et

- Les automates sont des graphes où les états sont les noeuds du graphe et les arcs représentent la fonction de transition.

- *L'ensemble des mots* pouvant être acceptés par un automate ***A*** est appelé le *langage reconnu par* ***A***, noté ***L*(*A*)**.

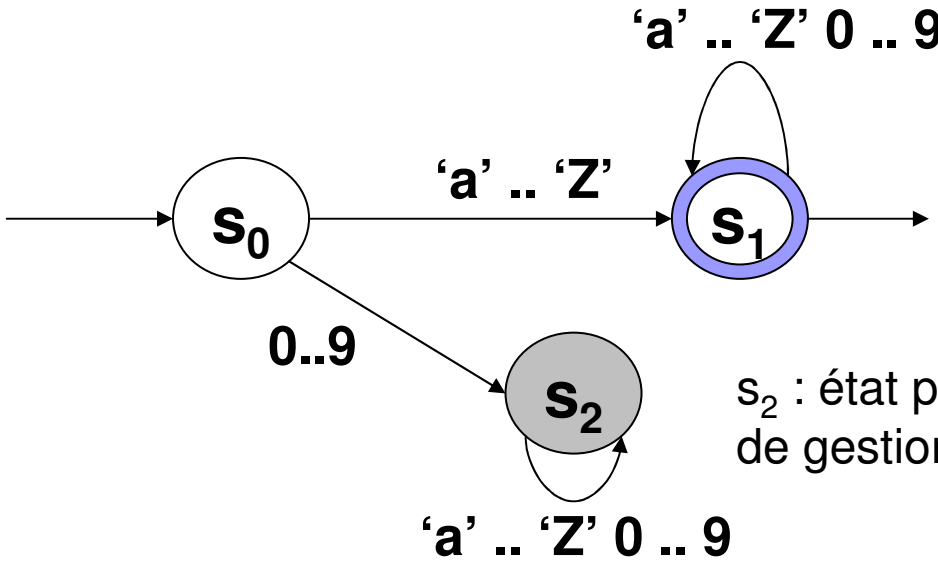
Automates d'états finis DFA - Deterministic Finite Automata

- On dit qu'un automate d'états finis A = **<S, Σ, δ, s_0, F >** accepte le mot x, composé des caractères **$x_1x_2x_3 \ldots x_n$** ssi
  $\delta(\delta(\ldots \delta(\delta(s_0, x_1), x_2), x_3), \ldots), x_{n-1}), x_n) \in F$

- Intuitivement **$x_1x_2x_3 \ldots x_n$** correspond aux étiquettes des arcs formant un chemin à travers les transitions de l'automate, commençant par **s_0** et se terminant par **$s_k \in$** F.
- A chaque étape, **x_i** correspond à l'étiquette du ième arc du chemin.

Exercice

- Écrire un programme qui implante l'automate de reconnaissance du langage des identificateurs



s_2 : état poubelle (ou trash)
de gestion d'erreur aussi noté ∅

De l'ER à l'analyseur lexical (1ère méthode)

- **void automate();**
- **typedef enum {state0, state1, state2} State;**
- **int main(char *argv){**
  - ***// 1- traitement intial***
  - **State state = state0;**
  - **printf("state0 ");**
-
  - **automate(state);**
-
  - **return 0;**
- **}**

- **void automate(State current_state){**
  - State state = current_state;
  - char c = getchar();
  - if (c != '$') {
  - *// 2- traitement récursif*
  - switch (state){
    - case state0 :
      - if (((c >= 'a') && (c <='z')) || ((c >= 'A') && (c <='Z'))){
        - state = state1; printf("state0 --> state1\n");
      - }else state = state2; printf("state0 --> state2\n");
      - break;
    - case state1 :
      - if (((c >= 'a') && (c <='z')) || ((c >= 'A') && (c <='Z')) || ((c >= '0') && (c <= '9')))
        - state = state1; printf("state1 --> state1\n");
      - break;
    - case state2 :
      - state = state2; printf("state2 --> state2\n");
    - }
  - automate(state);
  - }else{
    - *// 3- traitement final*
    - if (state == state1) printf("mot accepté par l'automate des identificateurs\n");
    - else printf("mot refusé par l'automate des identificateurs\n");}}

Test d’appartenance au langage de l’automate

- % ./automate.out
- state0 **a123456**$
- state0 --> state1
- state1 --> state1
- state1 --> state1
- state1 --> state1
- state1 --> state1
- state1 --> state1
- state1 --> state1
- mot accepté par l'automate des identificateurs

Test d’appartenance au langage de l’automate

- % ./automate.out
- state0 **12345**$
- state0 --> state2
- state2 --> state2
- state2 --> state2
- state2 --> state2
- state2 --> state2
- mot refusé par l'automate des identificateurs

De l'ER à l'analyseur lexical (2ème méthode)

- *On définit les fonctions suivantes*
  - *T* : *state* → *{start, normal, final, error}*
  - *δ* : *state* × *character* → *state*

δ

S	**'a' .. 'Z'**	**0 .. 9**
s_0	s_1	s_2
s_1	s_1	s_1
s_2	s_2	s_2

S	**T**
s_0	start
s_1	final
s_2	error

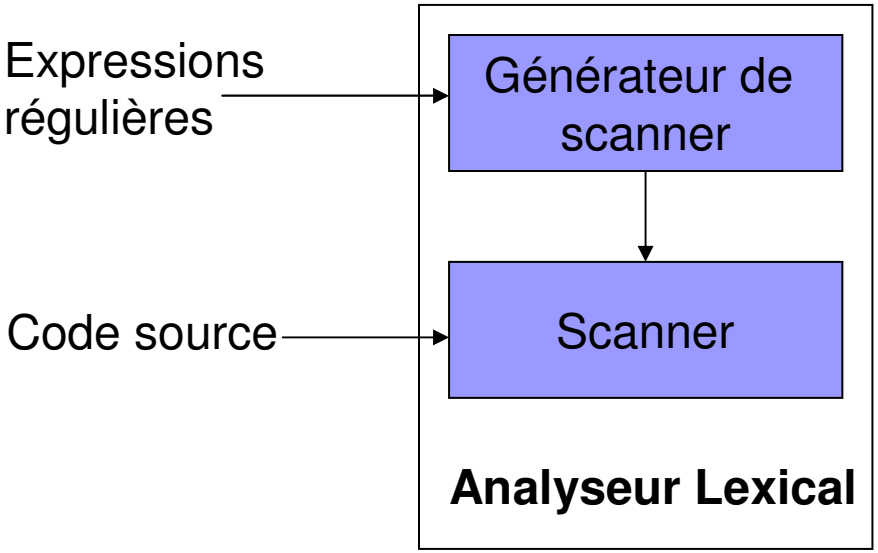
De l'ER à l'analyseur lexical

```
scanner()
    // 1- traitement intial
    char ← next_character();
    state ← s0;
    action(char,state);

    // 2- traitement itératif
    while ( char != eof )
            state ← δ(state, char);
            action(char , state);
            char ← next_character();

    // 3- traitement final
    if (T [state] == final)
            report_acceptance();
    else
            report_failure();
```

```
action(char, state)
        switch (T [state])
        case start:
                word ← char;
                break;
        case normal:
                word ← word + char;
                break;
        case final:
                word ← word + char;
                break;
        case error:
                print_error_message();
                break;
```

Génération automatique de scanner

- Il existe des algorithmes et outils permettant de générer le code implantant l'automate d'états finis (scanner) correspondant à une expression régulière (e.g. LEX)

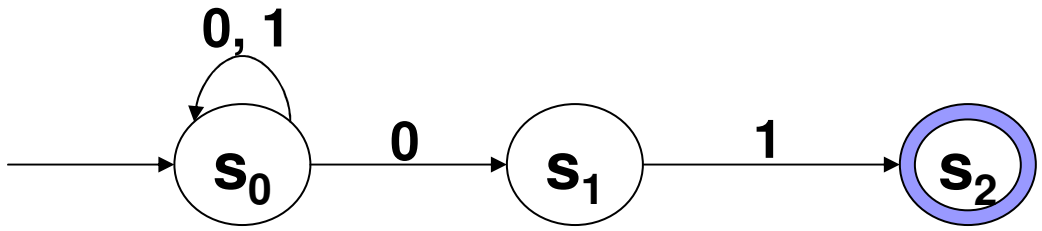


Automates non-déterministes NFA : Nondeterministic Finite Automata

- Il est difficile d'écrire simplement les automates correspondant à une ER compliquée, l'automate non-déterministe simplifie cette tâche

- Un automate non-déterministe **A = <S, Σ, δ, s_0, F >** est un automate qui présente la particularité suivante :
  - Il possède une Fonction de transition étendue $\delta : \mathbf{S} \times \Sigma \cup \{\varepsilon\} \rightarrow \wp(\mathbf{S})$
    1. plusieurs arcs reconnaissant le même caractère peuvent « sortir » du même état
    2. Il peut y avoir des transitions, appelées (ε-transitions), qui ne correspondent à une aucune reconnaissance de caractères.

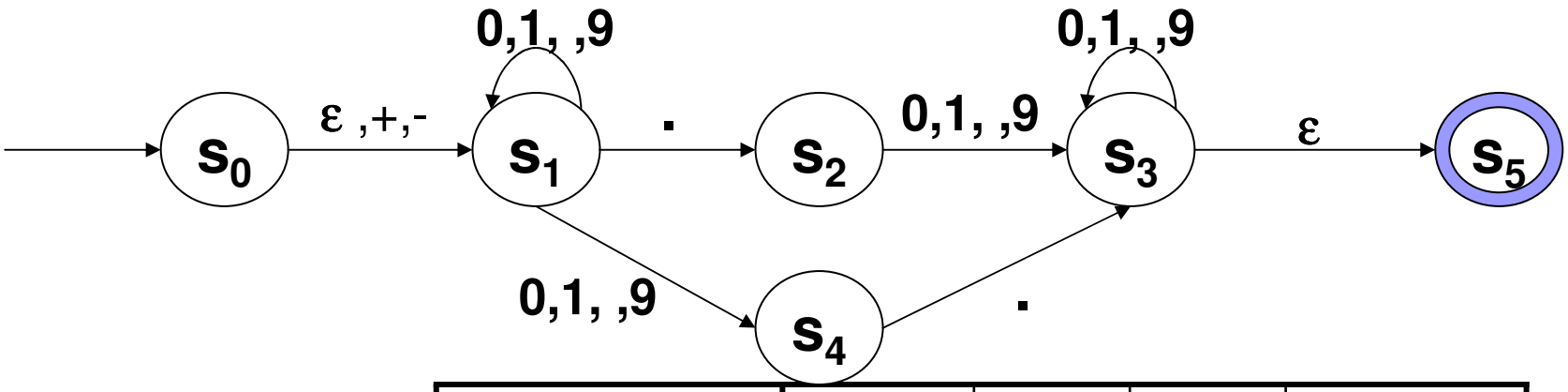
Exemple de NFA (sans ε-transitions)



δ

S	0	1
s_0	$\{s_0, s_1\}$	$\{s_0\}$
s_1	$\varnothing$	$\{s_2\}$
s_2	$\varnothing$	$\varnothing$

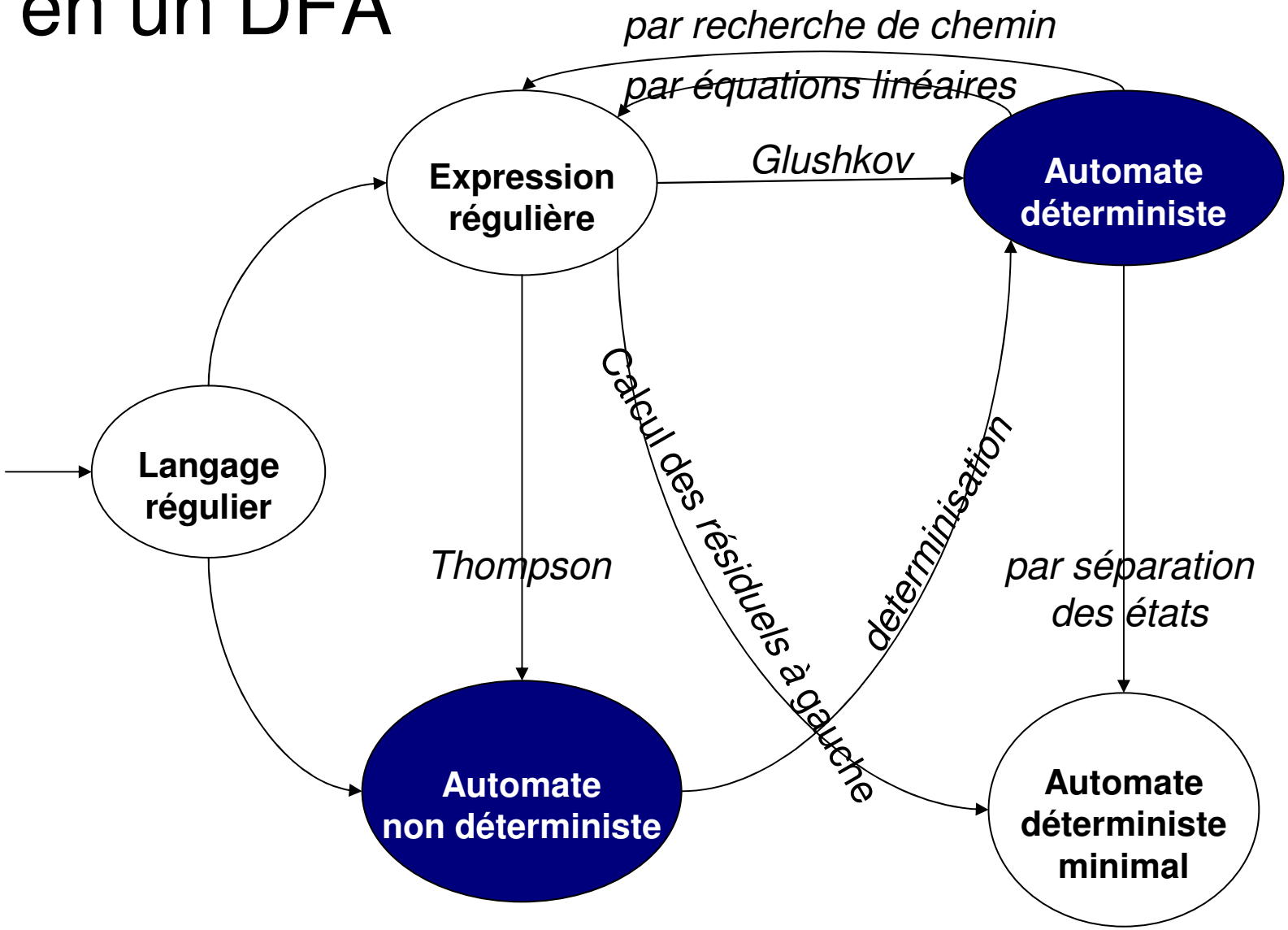
Exemple de NFA (avec ε-transitions)



δ

S	ε	+,-	.	0,1, ,9
s_0	$\{s_1\}$	$\{s_1\}$	∅	∅
s_1	∅	∅	$\{s_2\}$	$\{s_1, s_4\}$
s_2	∅	∅	∅	$\{s_3\}$
s_3	$\{s_5\}$	∅	∅	$\{s_3\}$
s_4	∅	∅	$\{s_3\}$	∅
s_5	∅	∅	∅	∅

Algorithme de transformation d'un NFA en un DFA



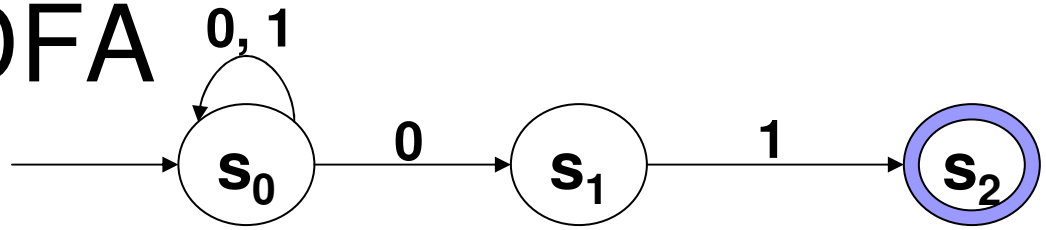
Algorithme de transformation d'un NFA en un DFA

- Deux cas
  - 1er cas : l'automate non-déterministe est sans ε-transition
  - 2ème cas : l'automate non-déterministe contient des ε-transitions

Algorithme de transformation d'un NFA (sans ε-transitions) en un DFA

- **Soit $A_N = <S_N, \Sigma_N, \delta_N, s_{N0}, F_N>$ un automate non-déterministe sans** ε-transitions **(NFA)**, il existe un automate déterministe (DFA) **$A_D = <S_D, \Sigma_D, \delta_D, s_{D0}, F_D>$ tel que $L(A_N) = L(A_D)$**
  - **$S_D \subseteq \wp(S_N)$**
    - tous les éléments de $\wp(S_N)$ ne seront accessibles depuis l'état initial de S_D
  - **$\Sigma_D = \Sigma_N$**
  - **$\delta_D(s_D \in S_D, a \in \Sigma_D) = \cup_{s_N \in s_D} \delta_N(s_N, a)$**
  - **$s_{D0} = \{s_{N0}\}$**
  - **$F_D = \{s \in S_D \mid (s \cap F_N) \neq \emptyset\}$**

Exemple : NFA (sans ε-transitions) vers DFA

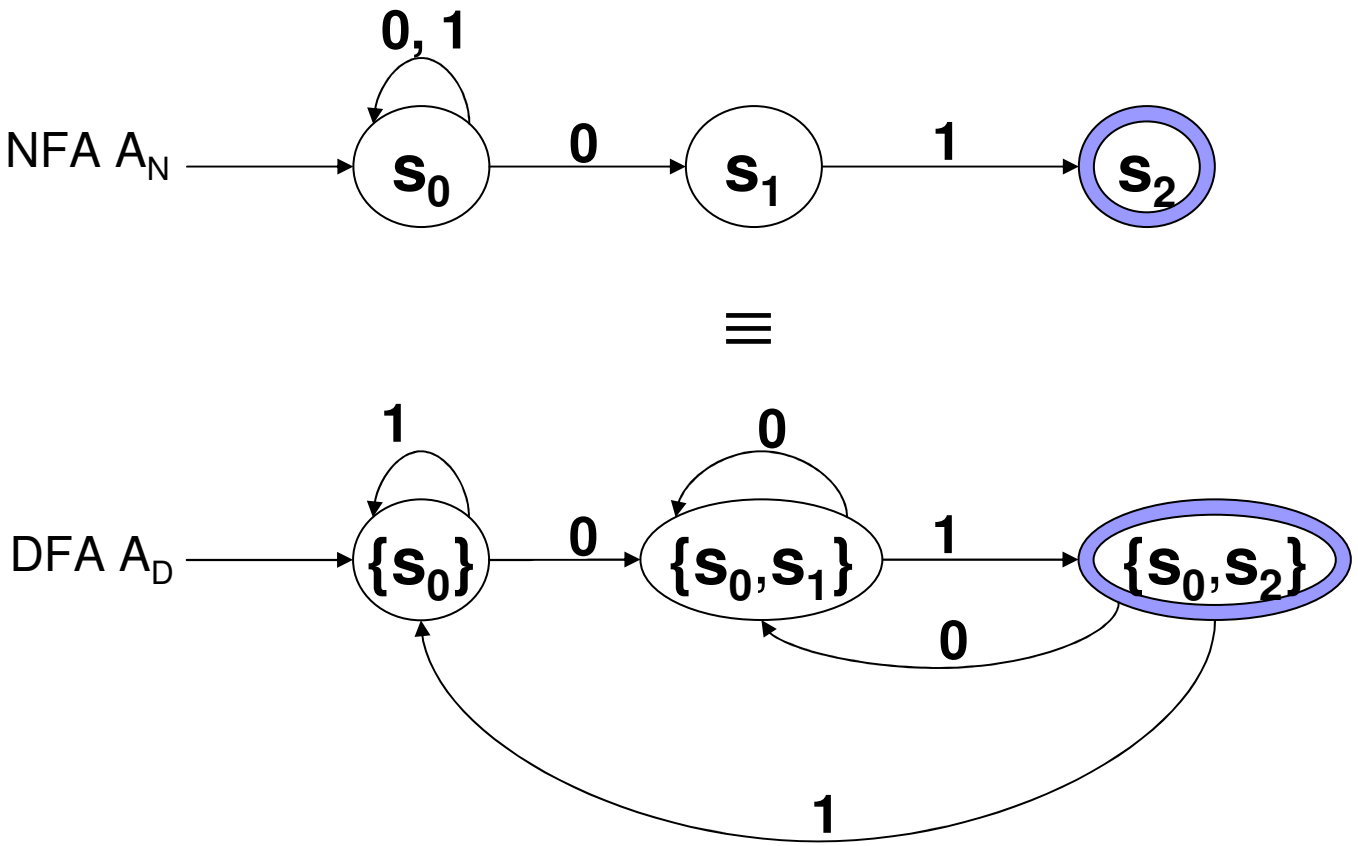


δ_D

$\wp(S_N)$	0	1
∅	∅	∅
$S_{D0} = \{s_0\}$	$\{s_0,s_1\}$	$\{s_0\}$
$\{s_1\}$	∅	$\{s_2\}$
$\{s_2\}$	∅	∅
$\{s_0,s_1\}$	$\{s_0,s_1\}$	$\{s_0,s_2\}$
$F_D=\{s_0,s_2\}$	$\{s_0,s_1\}$	$\{s_0\}$
$\{s_1,s_2\}$	∅	$\{s_2\}$
$\{s_0,s_1,s_2\}$	$\{s_0,s_1\}$	$\{s_0,s_2\}$

S : état accessible depuis S_{D0}

Exemple : NFA (sans ε-transitions) vers DFA

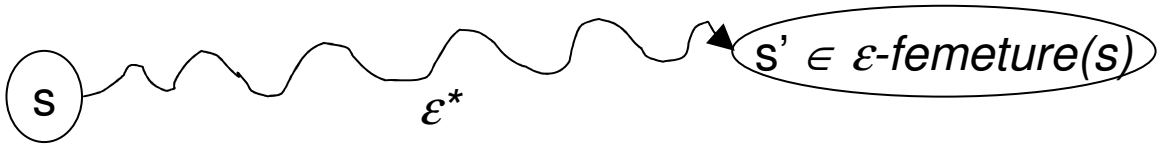


- **$L(A_N) = L(A_D) = \{w=(0|1)^*01\}$**

Algorithme de transformation d'un NFA (avec ε-transitions) en un DFA

- *On définit la fonction* ***ε-fermeture :*** $\mathbf{S} \rightarrow \wp(\mathbf{S})$ *comme suit:*
  1. $s \in \varepsilon\text{-fermeture}(s)$
  2. Si $s_1 \in \delta(s,\varepsilon)$ alors $s_1 \in \varepsilon\text{-fermeture}(s)$
  3. Si $s_1 \in \varepsilon\text{-fermeture}(s) \wedge s_2 \in \varepsilon\text{-fermeture}(s_1)$ alors $s_2 \in \varepsilon\text{-fermeture}(s)$

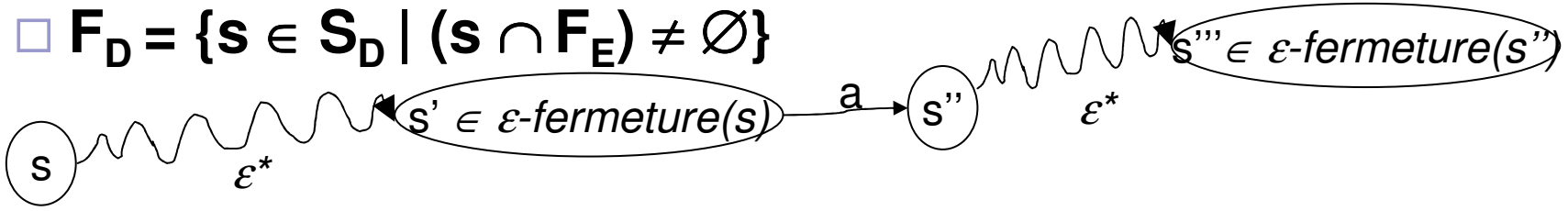
  - On peut définit par extention ***ε-fermeture :*** $\wp(\mathbf{S}) \rightarrow \wp(\mathbf{S})$ telle que
    - ***ε-fermeture(S'*** $\subseteq$ S) = $\cup_{s' \in S'}$ ***ε-fermeture(s')***



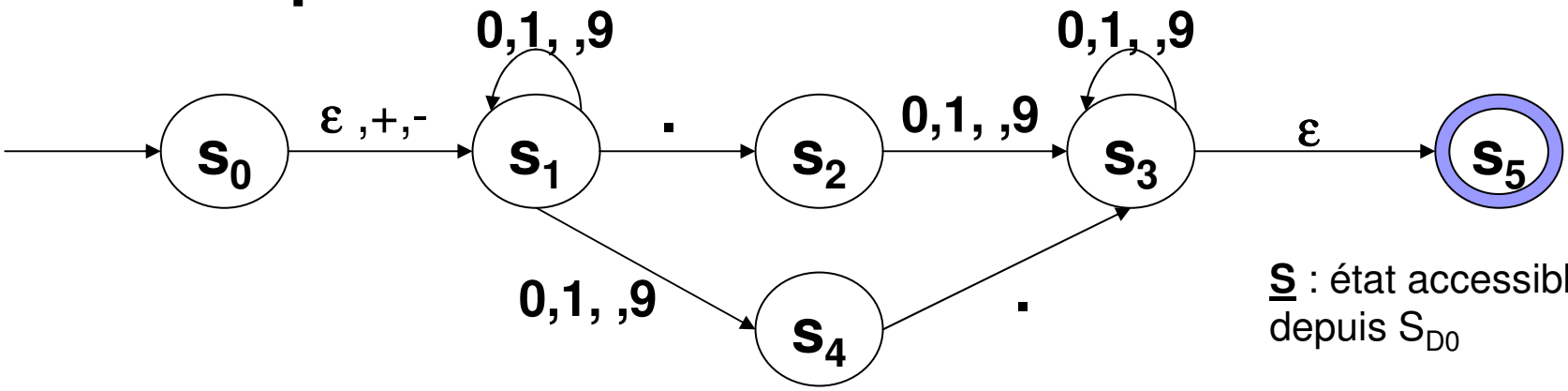
  - *ε-fermeture*(*s*) regroupe tous les états accessibles depuis s sur des chemins constitués seulement de ε-transitions

Algorithme de transformation d'un NFA (avec ε-transitions) en un DFA

- **Soit $A_E = <S_E, \Sigma_E, \delta_E, s_{E0}, F_E>$ un automate non-déterministe** avec ε-transitions **(ε-NFA)**, il existe un automate déterministe (DFA) **$A_D = <S_D, \Sigma_D, \delta_D, s_{D0}, F_D>$ tel que $L(A_E) = L(A_D)$**
  - **$S_D \subseteq$** *ε-fermeture(*$\wp$ **(S_E)***)*
    - **tous les éléments de** *ε-fermeture(*$\wp$ **(S_E)***)* **ne seront accessibles depuis l'état initial de S_D**
  - **$\Sigma_D = \Sigma_E$**
  - **$\delta_D(s_D \in S_D, a \in \Sigma_D) =$** *ε-fermeture(* $\cup_{s_E \in s_D}$ **$\delta_E(s_E, a)$***)*
  - **$s_{D0} =$ {***ε-fermeture(***s_{E0}***)***}**
  - **$F_D = \{s \in S_D \mid (s \cap F_E) \neq \emptyset\}$**



Exemple : ε-NFA vers DFA

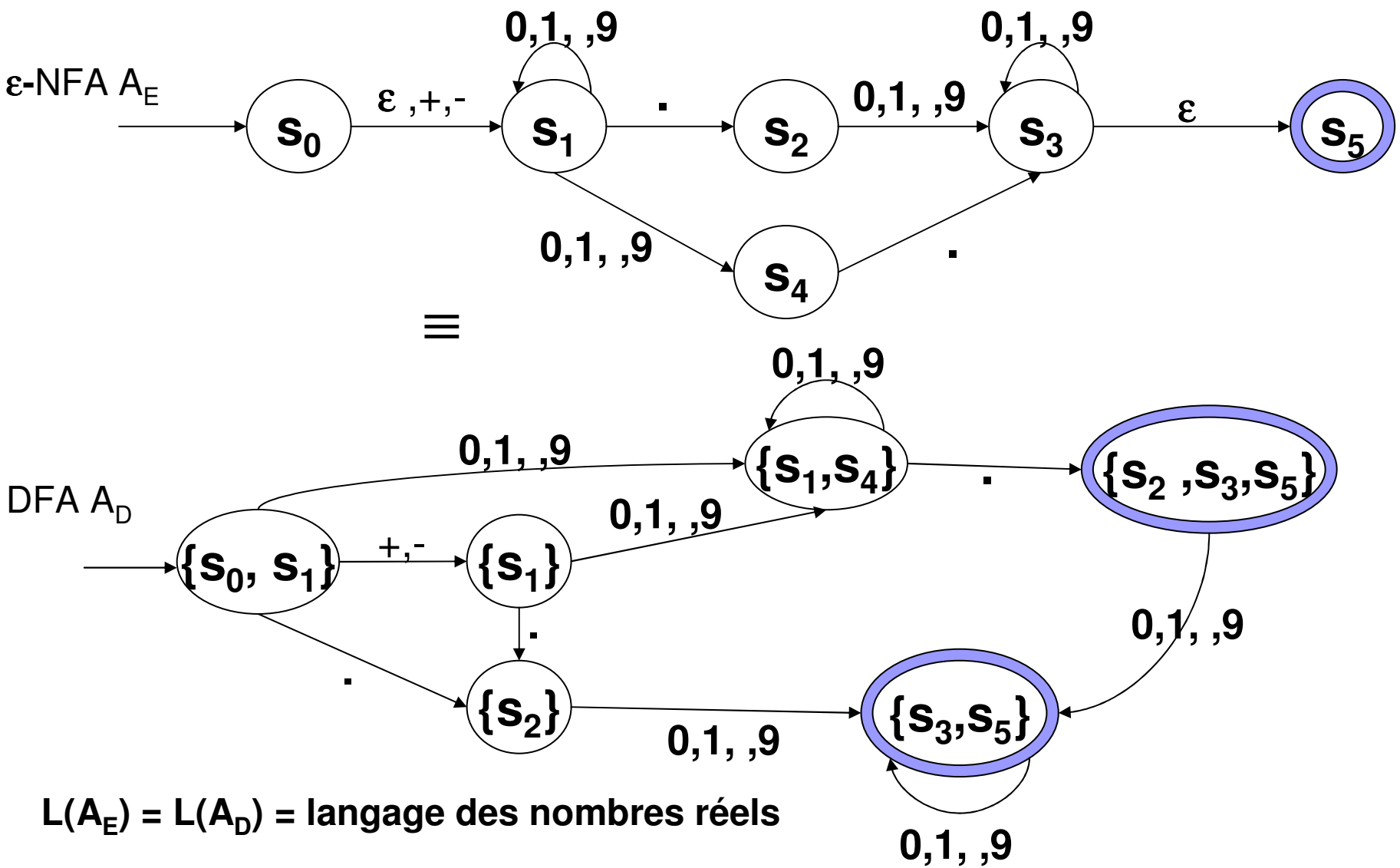


S : état accessible depuis S_{D0}

δ_D

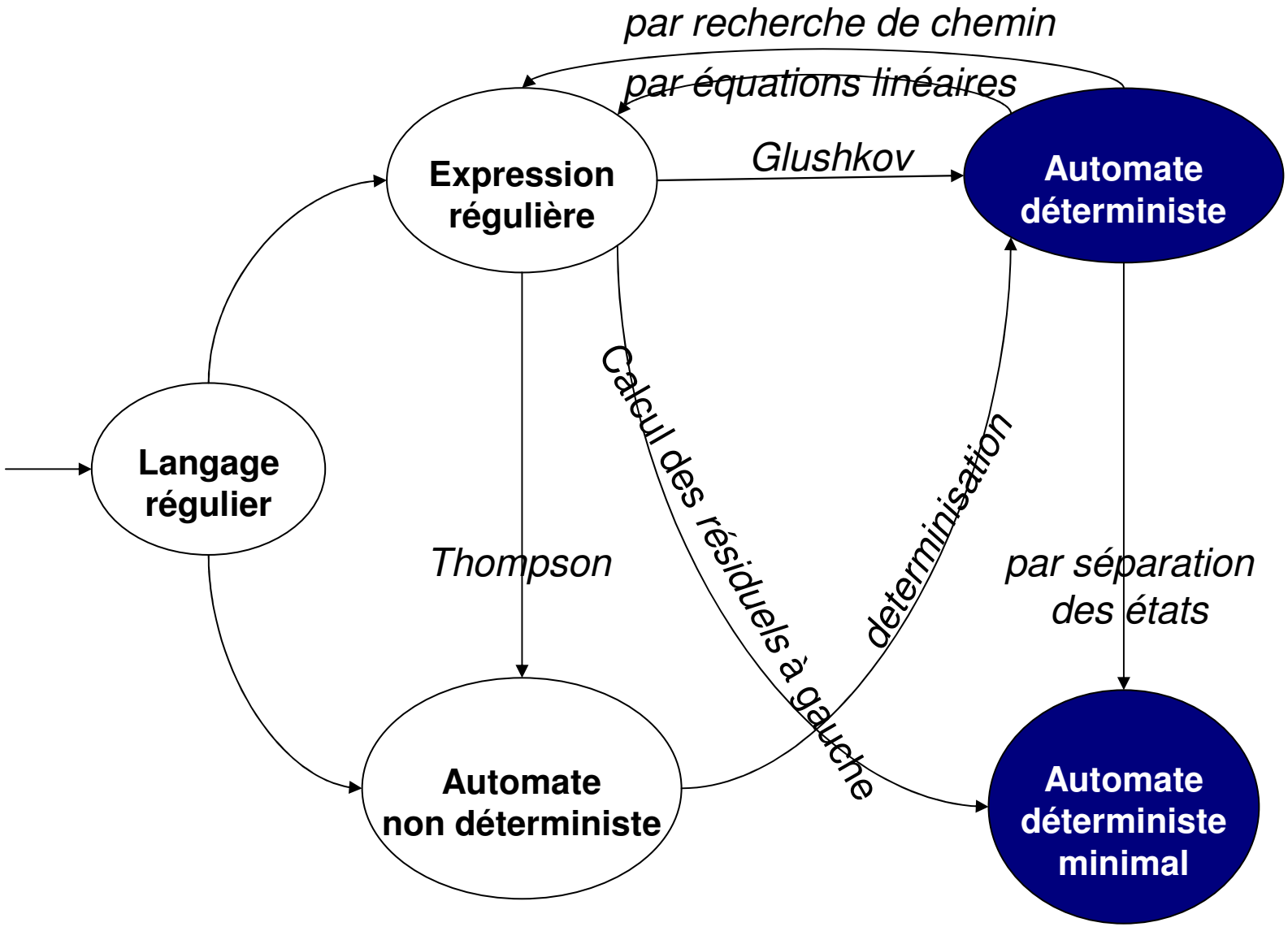
S_E ⇨ ε-fermeture	+,- ⇨ ε-fermeture	. ⇨ ε-fermeture	0,1, ,9 ⇨ ε-fermeture
s_0	s_1	s_2	s_1, s_4
$S_{D0}=\{s_0, s_1\}$	$\{s_1\}$	$\{s_2\}$	**$\{s_1, s_4\}$**
s_1	∅	s_2	s_1, s_4
$\{s_1\}$	∅	$\{s_2\}$	$\{s_1, s_4\}$
s_2	∅	∅	s_3
$\{s_2\}$	∅	∅	$\{s_3, s_5\}$
s_3	∅	∅	s_3
$F_{D1}=\{s_3, s_5\}$	∅	∅	$\{s_3, s_5\}$
s_4	∅	s_3	∅
$\{s_4\}$	∅	s_3, s_5	∅
s_5	∅	∅	∅
$\{s_5\}$	∅	∅	∅
	∅	s_2, s_3	s_1, s_4
$\{s_1, s_4\}$	∅	**$\{s_2, s_3, s_5\}$**	$\{s_1, s_4\}$
	∅	∅	s_3
$F_{D2}=\{s_2, s_3, s_5\}$	∅	∅	$\{s_3, s_5\}$

Exemple : ε-NFA vers DFA



$L(A_E) = L(A_D)$ = langage des nombres réels

Algorithme de minimisation d'un automate déterministe *(par séparation des états)*



Algorithme de minimisation d'un automate déterministe *(par séparation des états)*

- Deux états ***p***,***q*** de ***A*** sont dits équivalents (non-distinguables) p $\approx$ q ssi
  $\forall \boldsymbol{w} \in \Sigma^{*}, \delta(\boldsymbol{p},\boldsymbol{w}) \in F \Leftrightarrow \delta(\boldsymbol{q},\boldsymbol{w}) \in F$
  1. ***w* = ε, si *p* $\in$ *F* $\wedge$ *q* $\notin$ *F* alors *p* et *q* sont distinguables (non équivalents)**
  2. ***w* $\neq$ ε, (δ(*p*,*w*)=*r* et δ(*q*,*w*)=*s*) $\wedge$ (*r* et *s* sont distinguables) alors *p* et *q* sont distinguables**

- On note *[p $\in$ S]* la classe d'équivalence de *p* (l'ensemble des états *q* qui lui sont équivalents)

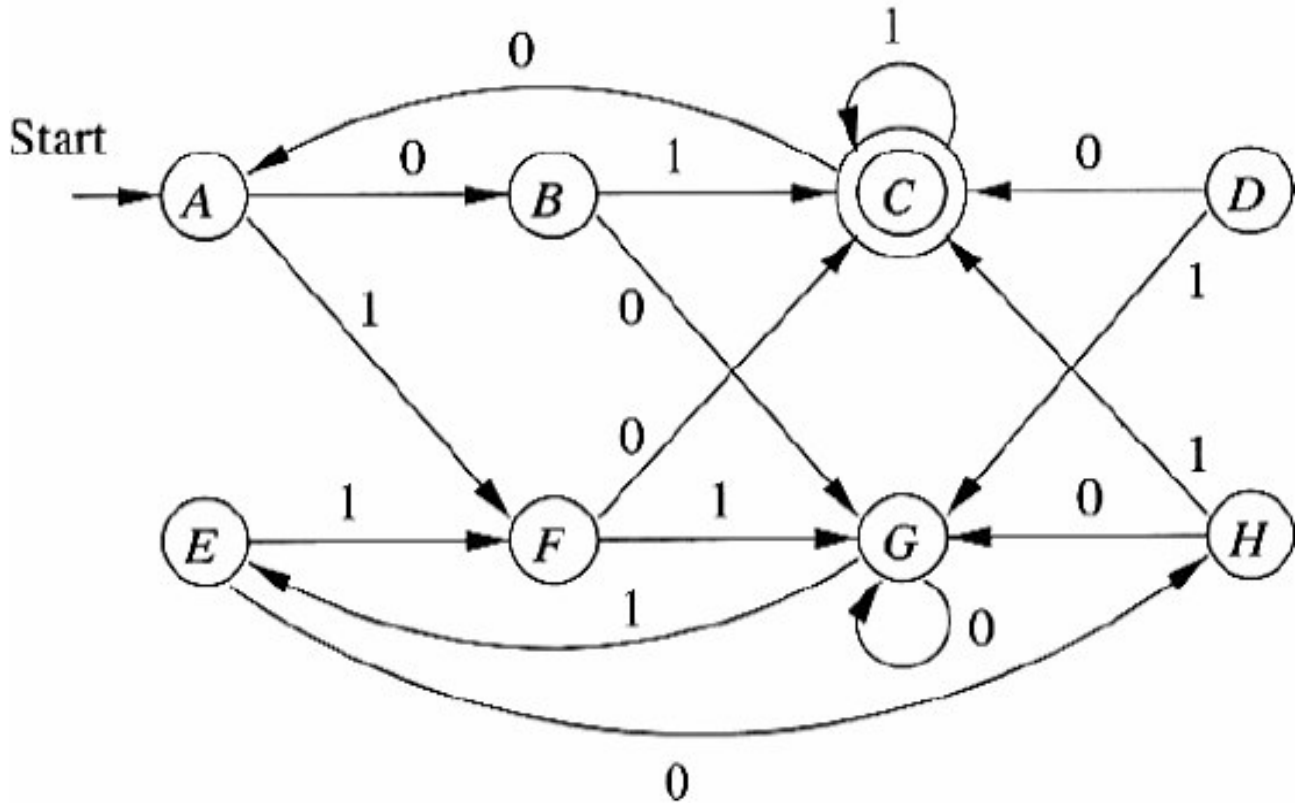
Algorithme de minimisation d'un automate déterministe *(par séparation des états)*

- ***Étape 0 :***
  - $p, q \in S, p \approx_0 q$ ***ssi***
    - $(p \in F \wedge q \in F) \vee (p \notin F \wedge q \notin F)$

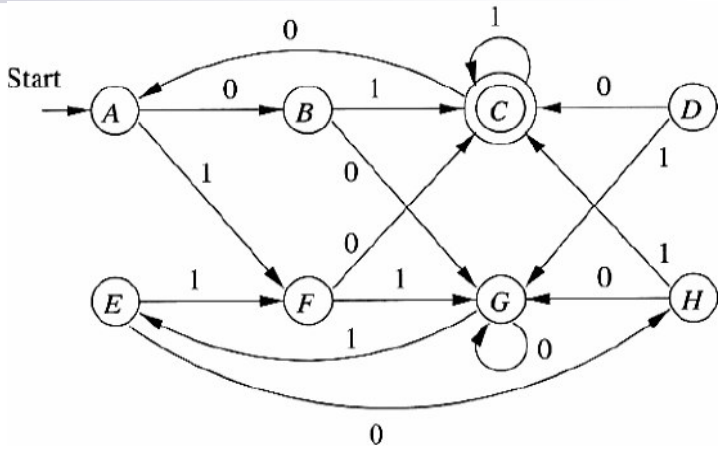
- ***Étape i :***
  - $\forall i>0, p \approx_i q$ ***ssi***
    - $p \approx_{i-1} q$
    - $\wedge \forall l \in \Sigma, \delta(p,l) \approx_{i-1} \delta(q,l)$

- ***Étape finale (arrêt des itérations) :***
  - $\approx_i = \approx_{i-1}$

Exemple de minimisation d'un DFA



Exemple de minimisation d'un DFA

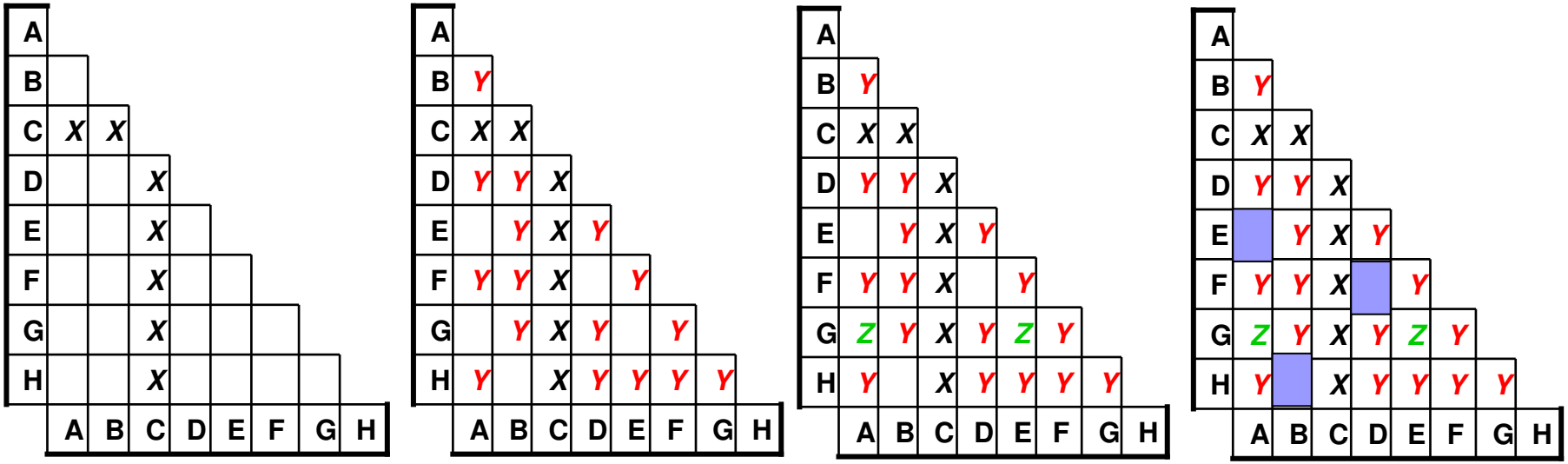


C_{01}= [A]={A,B,D,E,F,G,H}
C_{02}= [C]={C}

C_{11}=[A]={A,E,G},
C_{12}=[B]={B,H},
C_{13}=[C]={C},
C_{14}=[D]={D,F}

C_{21}=[A]={A, E},
C_{22}=[B]={B,H},
C_{23}=[C]={C},
C_{24}=[D]={D,F}
C_{25}=[G]={G}

C_{31}=[A]={A, E},
C_{32}=[B]={B,H},
C_{33}=[C]={C},
C_{34}=[D]={D,F}
C_{35}=[G]={G}



$\approx_0$ $\approx_1$ $\approx_2$ $\approx_3 = \approx_2$

- *$[p,q]_i$ = x signifie que $\neg (p \approx_i q)$ (i.e. p et q sont distinguables à la ième itération)*

Exemple de minimisation d'un DFA

	A	B	C	D	E	F	G	H
A								
B	Y							
C	X	X						
D	Y	Y	X					
E		Y	X	Y				
F	Y	Y	X		Y			
G	Z	Y	X	Y	Z	Y		
H	Y		X	Y	Y	Y	Y	

$\approx_3 = \approx_2$

$A_{min} = <S_{min}, \Sigma, \delta_{min}, S_{min0}, F_{min}>$

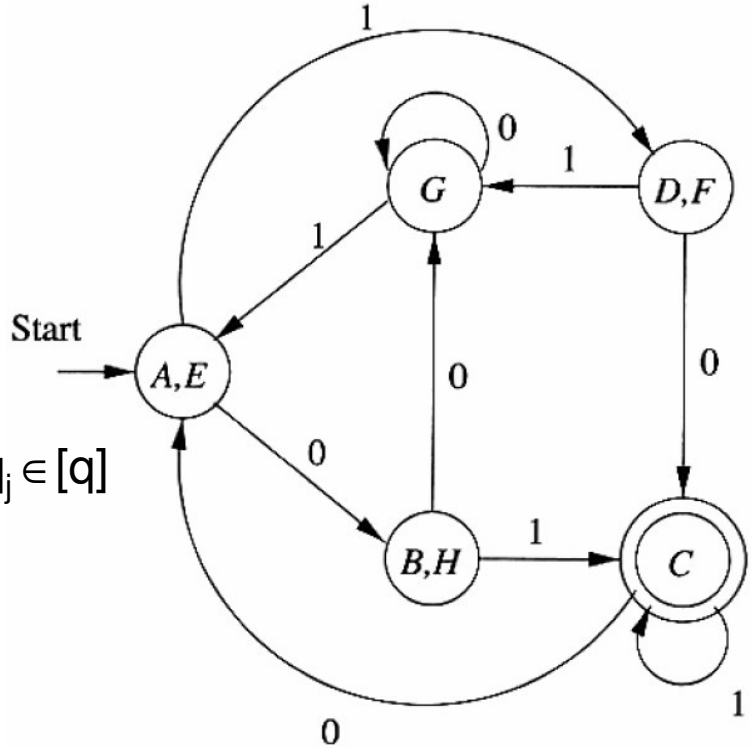
$S_{min} = \{S_j / S_j = C_{ij}\}$

$S_1 = C_{31} = [A] = \{A, E\}$,
$S_2 = C_{32} = [B] = \{B, H\}$,
$S_3 = C_{33} = [C] = \{C\}$,
$S_4 = C_{34} = [D] = \{D, F\}$
$S_5 = C_{35} = [G] = \{G\}$

$\delta_{min}([p], l) = [q]$ ssi $\exists p_i \in [p], \exists q_j \in [q]$
$\delta(p_i, l) = q_j$

$S_{min0} = [s_0] = [A]$

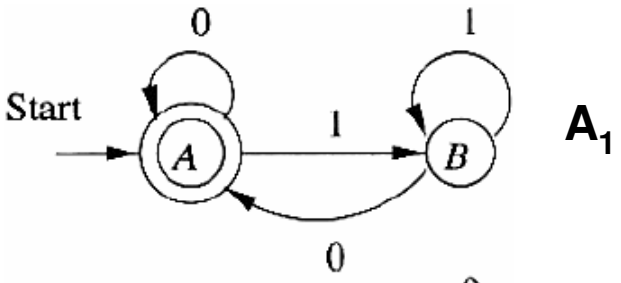
$F_{min} = \cup_{s \in F} [s] = [C]$



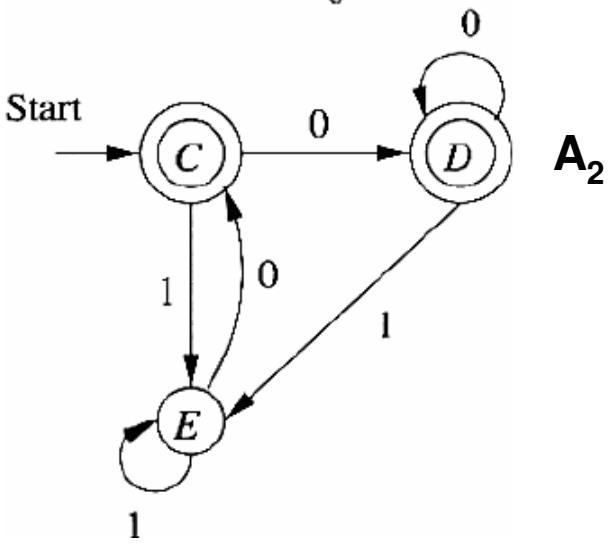
- L'automate minimal regroupe les états équivalents à l'étape finale

Exercice – minimisation d'automates

- Construire l'automate minimal équivalent à l'automate suivant (union de 2 automates A_1 et A_2)



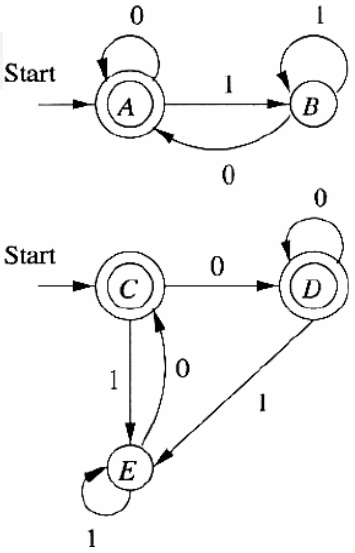
A_1



A_2

- PS. Négliger le fait qu'il existe deux états initiaux (pas important pour la minimisation)

Solution



C_{01}= [A]={A,C,D}
C_{02}= [B]={B,E}

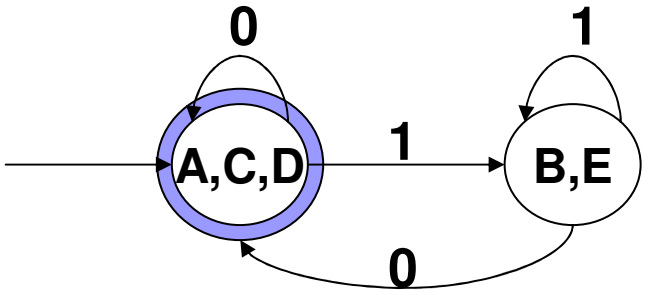
C_{11}=[A]={A,C,D},
C_{12}=[B]={B,E},

	A	B	C	D	E
B	X				
C		X			
D		X			
E	X		X	X	

$\approx_0$

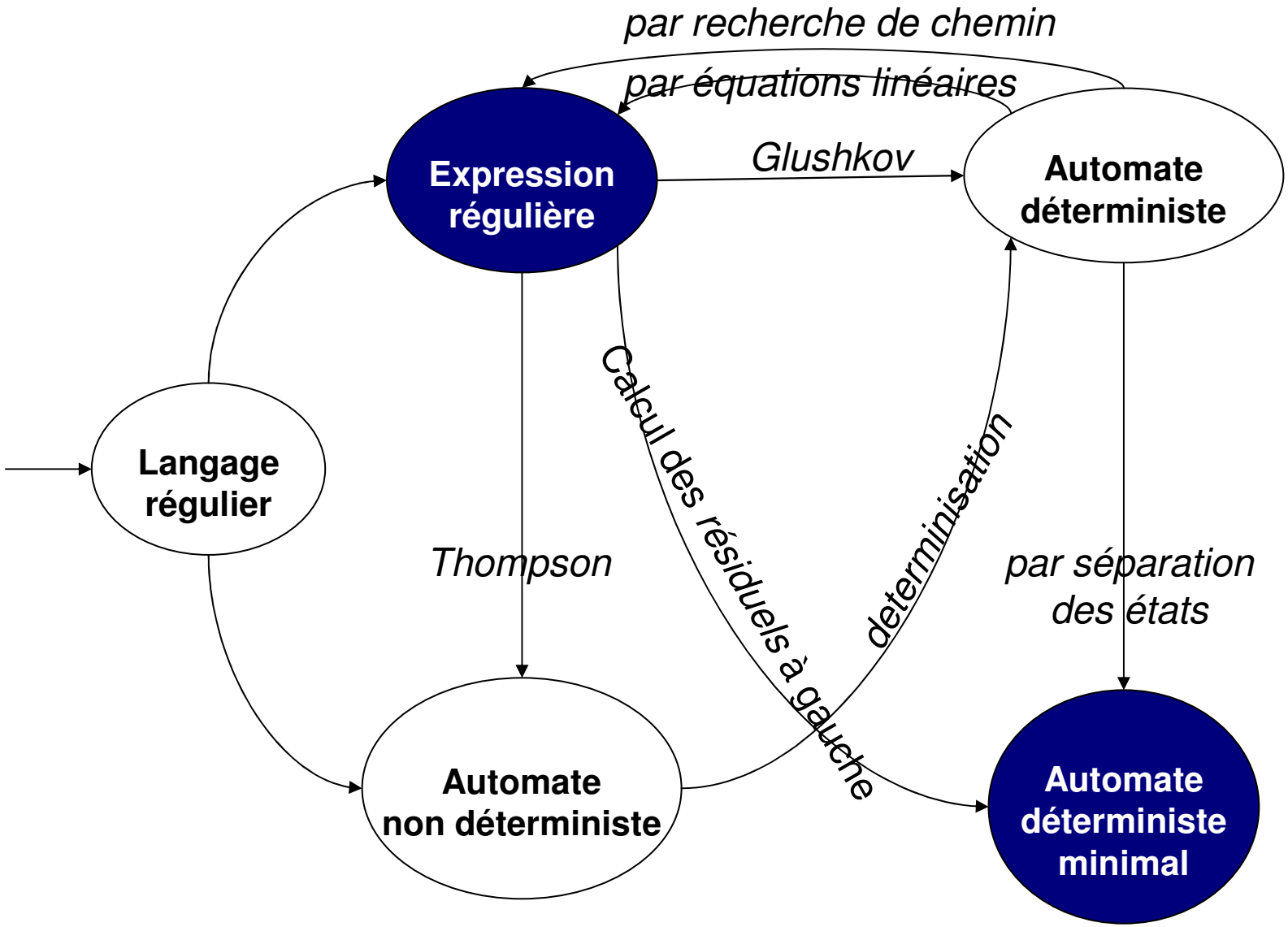
	A	B	C	D	E
B	X				
C		X			
D		X			
E	X		X	X	

$\approx_1 = \approx_0$



- *$[p,q]_i$ = x signifie que ¬ ($p \approx_i q$) (i.e. p et q sont distinguables à la ième itération)*

Algorithme de minimisation d'un automate déterministe *(par séparation des états)*



Algorithme de minimisation d'une expression régulière

- *Soit L un langage sur* un alphabet Σ, on appelle **résiduel à gauche** (*suffixe*) de L par rapport à un mot $u \in \Sigma^*$ l'ensemble des mots $v \in \Sigma^*$ *tels que* $uv \in L$ *(on note* $u^{-1}L$*)*
  - $u^{-1}L = \{v \in \Sigma^*, uv \in L\}$

- **L'ensemble des résiduels à gauche** (*suffixes*) de L est l'union pour tout $u \in \Sigma^*$ des $u^{-1}L$
  - $R(L) = \cup_{u \in \Sigma^*} u^{-1}L$

Algorithme de minimisation d'une expression régulière

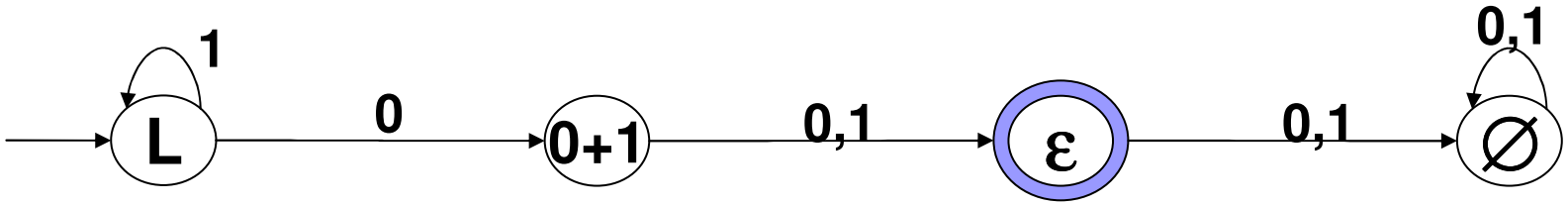
- *Soit* A= <S, Σ, δ, s_0, F> un automate, on appelle le langage associé à un état s$\in$ S (et on note $L_s(A)$) le langage reconnu par l'automate dont s est l'état initial et F l'ensemble des états finaux $L_s(A)=\{w \in \Sigma^* / \delta^*(s,w) \in F\}$ (*ensemble des suffixes de L reconnus à partir de l'état s*)

- **Proposition :** L'ensemble des résiduels à gauche (suffixes) de L est l'union des langages associés aux états de l'automate minimal le reconnaissant
  - $R(L) = \cup_{s\in S} L_s(A)$

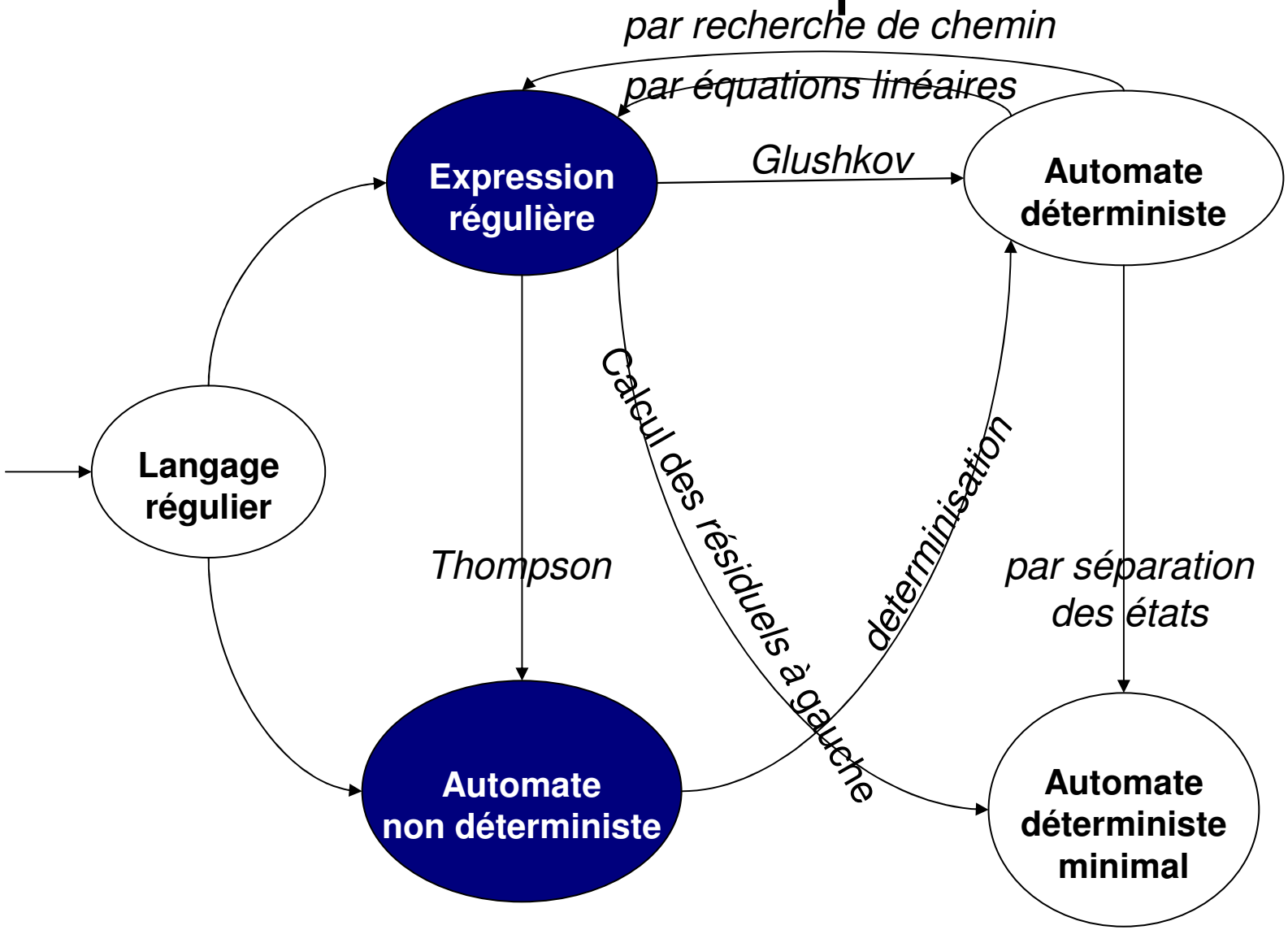
- **Conséquence :** donc calculer l'automate minimal reconnaissant une ER (ensemble des langages associés à ces états) revient à calculer R(L(ER))

Algorithme de minimisation d'une expression régulière - Exemple

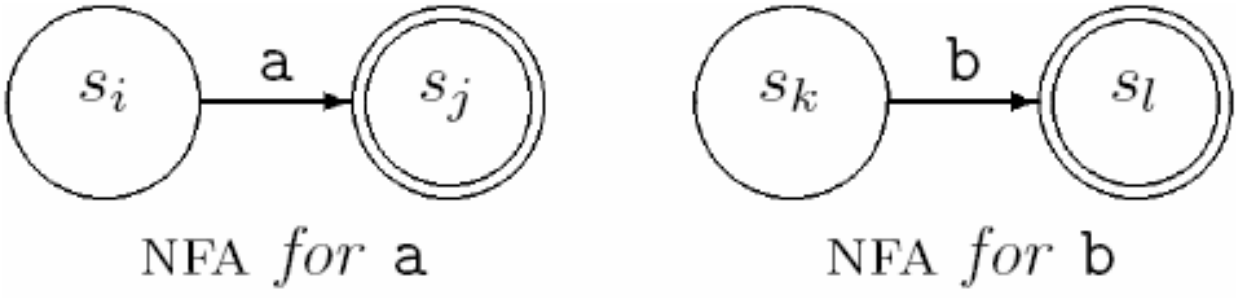
- Soit E = 1* 0 (0 + 1)
- Calculant R(L(E)) (i.e. les états de l'automate minimal)
  - $0^{-1}(L) = (\underline{0 + 1})$ *suffixe de L pour 0*
  - $1^{-1}(L) = 1^* 0 (0 + 1) = (\underline{L})$ *suffixe de L pour 1 (point fixe)*
  - $0^{-1}(0^{-1}L) = 0^{-1}(\underline{0 + 1}) = (\underline{\varepsilon})$ *suffixe d'ordre 2 de L pour 0*
  - $1^{-1}(0^{-1}L) = 1^{-1}(\underline{0 + 1}) = (\underline{\varepsilon})$ *suffixe d'ordre 2 de L pour 1*
  - $0^{-1}(0^{-1}(0^{-1}L)) = 0^{-1}(1^{-1}(0^{-1}L)) = 0^{-1}(\underline{\varepsilon}) = (\underline{\varnothing})$ *suffixe d'ordre 3 de L pour 0*
  - $1^{-1}(0^{-1}(0^{-1}L)) = 1^{-1}(1^{-1}(0^{-1}L)) = 1^{-1}(\underline{\varepsilon}) = (\underline{\varnothing})$ *suffixe d'ordre 3 de L pour 1*
  - $0^{-1}(\underline{\varnothing}) = (\underline{\varnothing})$ .....
  - $1^{-1}(\underline{\varnothing}) = (\underline{\varnothing})$ .....



Algorithme de transformation d'une ER en un NFA - Thompson

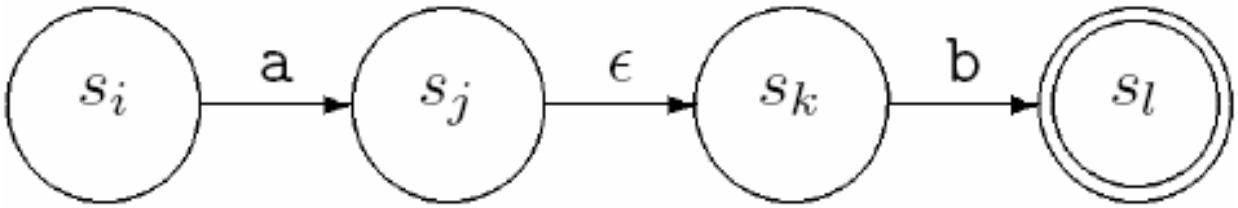


Algorithme de transformation d'une ER en un NFA - Thompson



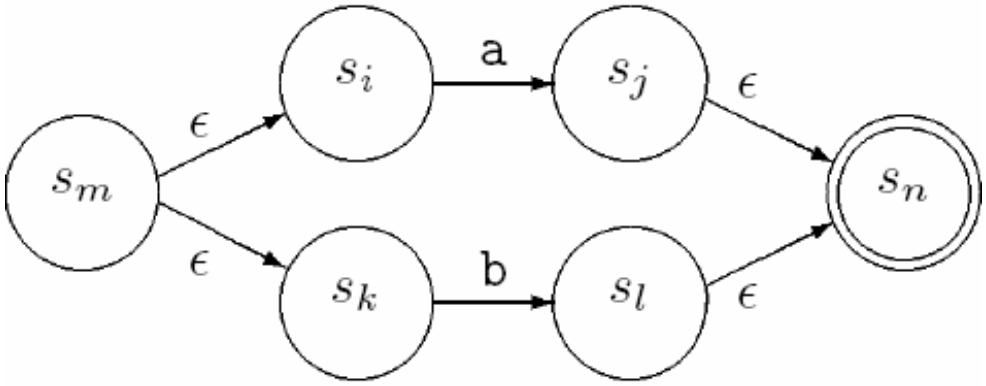
NFA *for* a

NFA *for* b

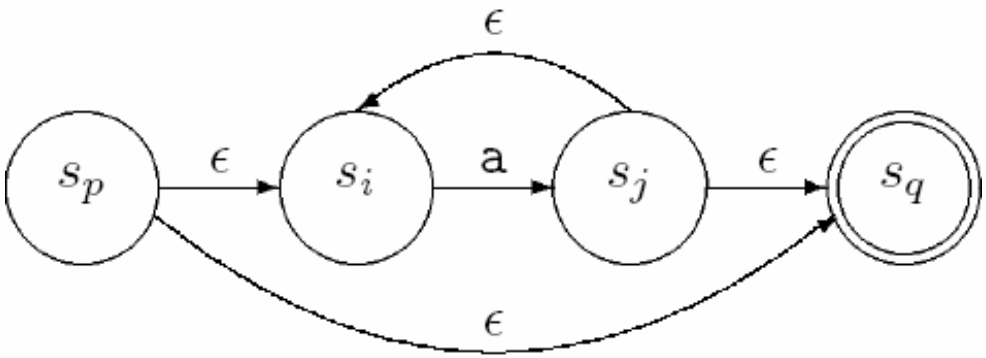


NFA *for* ab

Algorithme de transformation d'une ER en un NFA - Thompson



NFA *for* a | b



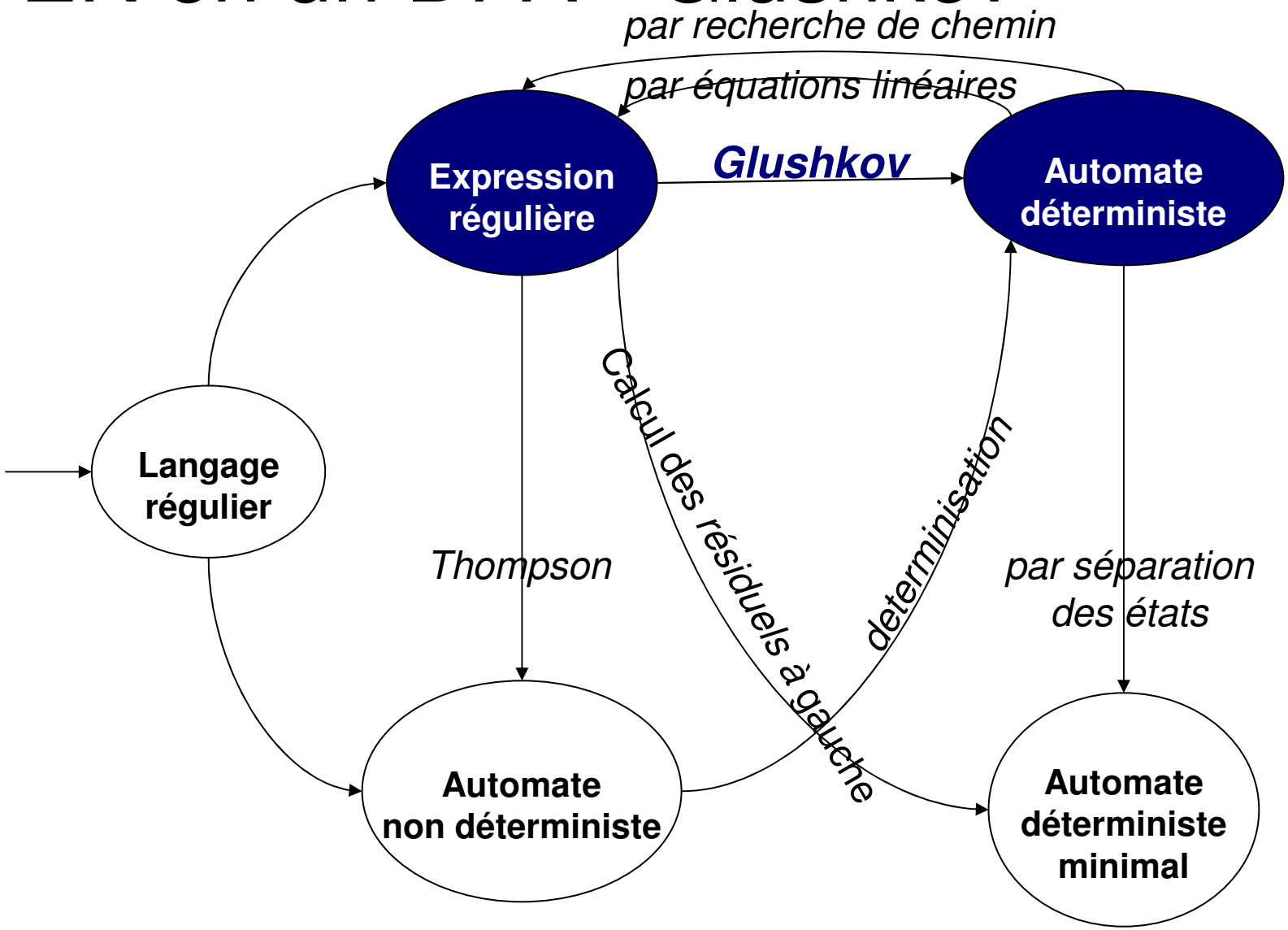
NFA *for* a*

Fermeture des langages réguliers

- Les langages réguliers sont fermés par les opérateurs union, concaténation, fermeture de Kleene, intersection, différence et complément

- Si L_1 et L_2 sont des langages réguliers alors
  - $L_1 \cup L_2$ est un langage régulier (par Thompson)
  - $L_1 . L_2$ est un langage régulier (par Thompson)
  - L_1^* est un langage régulier (par Thompson)
  - $L_1 \cap L_2$ est un langage régulier (par produit cartésien des automates)
  - $L_1 \setminus L_2$ est un langage régulier (par produit cartésien des automates)
  - $\Sigma^* \setminus L_1$ est un langage régulier (par inversion des états finaux)

Algorithme de transformation d'une ER en un DFA - Glushkov

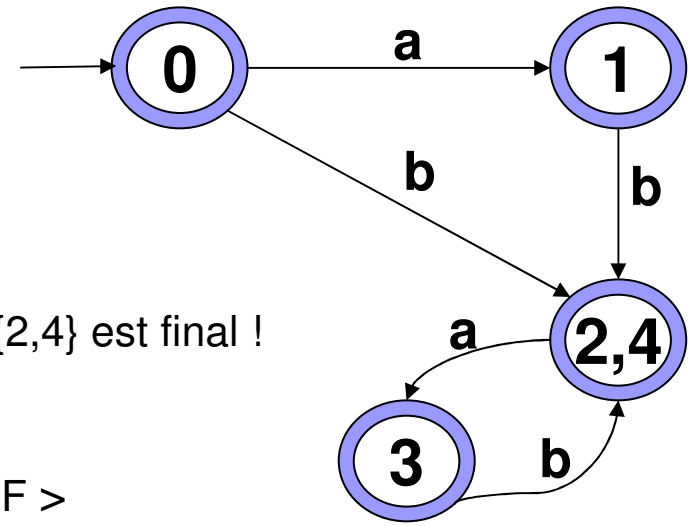


Algorithme de transformation d'une ER en un DFA - Glushkov

- **Entrée :** Soit *E* une expression régulière
  1. On transforme *E* en *E'* en indiçant toute lettre de 1 .. n (indice = état)
  2. On ajoute l'état $_0$ au début de *E'*
  3. On place une transition de 0 vers i ($\delta(0, l_j) \leftarrow i$)
     - ssi la lettre l_i peut être la première lettre d'un mot du langage L(E')
     - déterminisation systématique (*si $j \neq k$ et $\delta(0, l) = \{j,k\}$, alors l'état composite {j,k} regroupera les états atomiques j et k*)
  4. On place une transition de i vers j étiquetée par la lettre l_j ($\delta(i,l_j) \leftarrow j$)
     - ssi la lettre l_j peut suivre la lettre l_i dans un mot du langage L(E')
     - déterminisation systématique : *si $j \neq k$ et $\delta(i, l) = \{j,k\}$, alors l'état composite {j,k} regroupera les états atomiques j et k*
  5. L'état i (i > 0) est final
     - ssi l_i peut être la dernière lettre d'un mot du langage L(E')
  6. L'état 0 est final
     - ssi ε est dans le langage L(E')
- **Sortie :** $A = < S, \Sigma, \delta, s_0, F >$

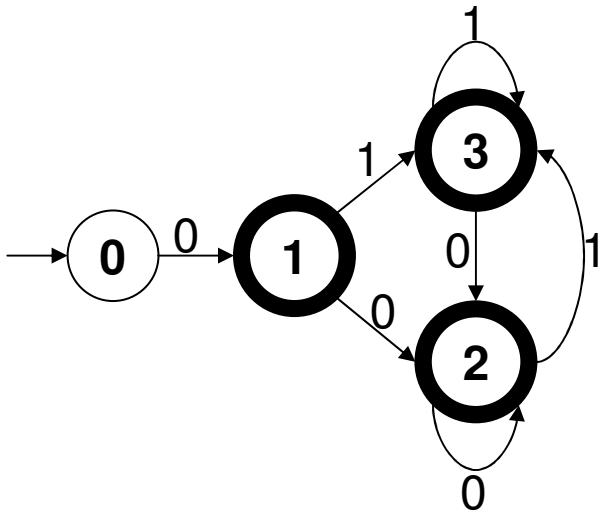
Algorithme de transformation d'une ER en un DFA - Exemple

- **Entrée :** Soit $E = (a+\varepsilon)\ (ba)^*\ (b+\varepsilon)$ une expression régulière
  1. $E' \leftarrow E_{\ Indicée}$
     - $E' \leftarrow (a_1+\varepsilon)\ (b_2a_3)^*\ (b_4+\varepsilon)$
  2. $E' \leftarrow {}_0E'$
     - $E' \leftarrow {}_0(a_1+\varepsilon)\ (b_2a_3)^*\ (b_4+\varepsilon)$
  3. Premières lettres possibles et transitions initiales
     1. $\delta(0, a) \leftarrow 1$; $\delta(0, b) \leftarrow \{2,4\}$ (* *determinisation systématique par regroupement des états*)
  4. $\delta(i,l_j) \leftarrow j$ ssi l_i peut être suivie de l_j
     1. $\delta(1, b) \leftarrow \{2,4\}$ (*)
     2. $\delta(\{2,4\}, a) \leftarrow 3$ (*)
     3. $\delta(3, b) \leftarrow \{2,4\}$ (*)
  5. Dernières lettres et transitions finales
     - $F \leftarrow \{1, 3, \{2,4\}\}$ l_4 peut être à la fin, donc {2,4} est final !
  6. L'état initial est-il final ?
     1. $F \leftarrow F \cup \{0\}$
- **Sortie :** $A = < S=\{0,1,\{2,4\},3\}, \Sigma=\{a,b\}, \delta, s_0=\{0\}, F >$

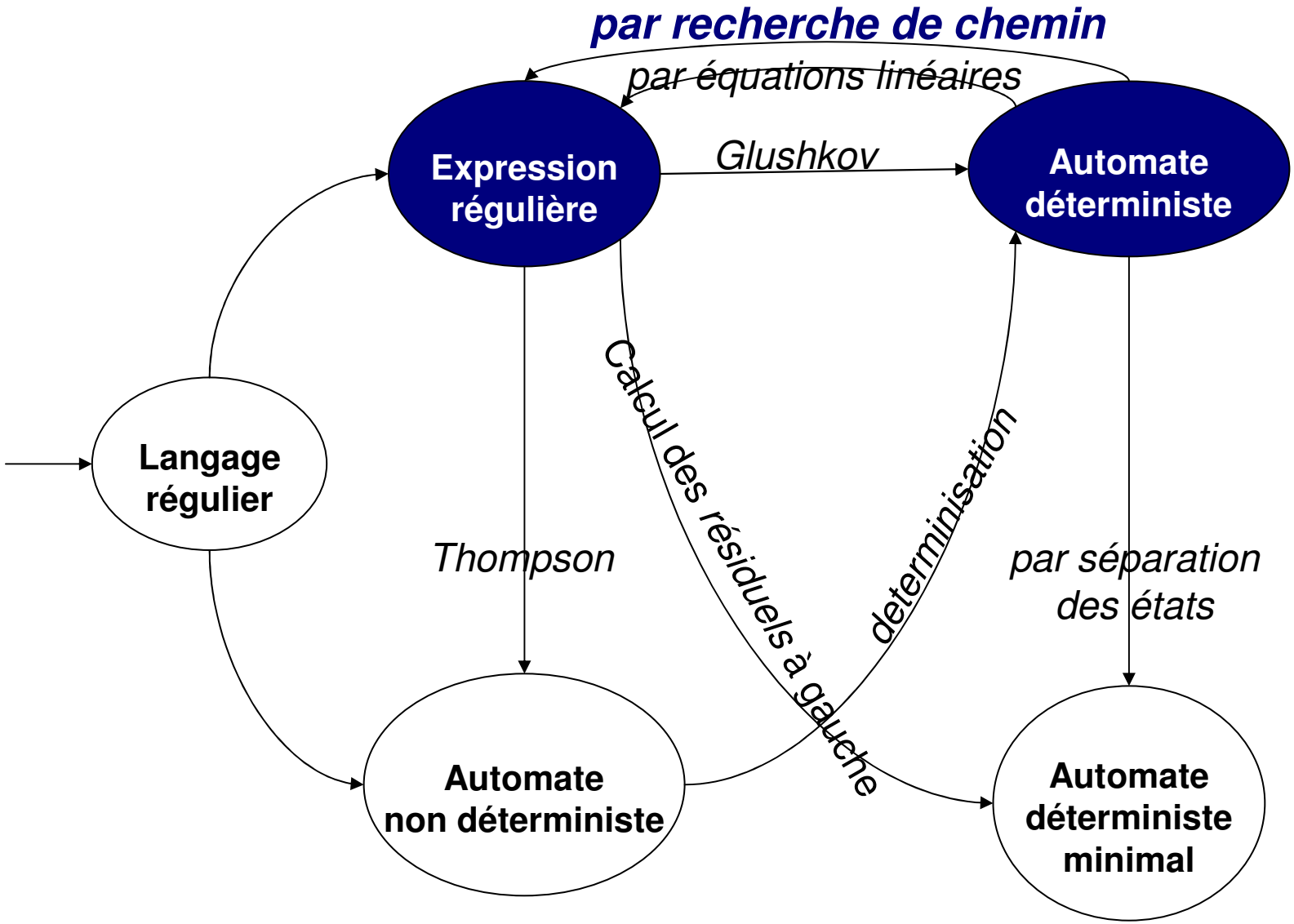


Exemple d'application

- 0(0+1)*



Algorithme de transformation d'un DFA en ER : *par recherche de chemin*

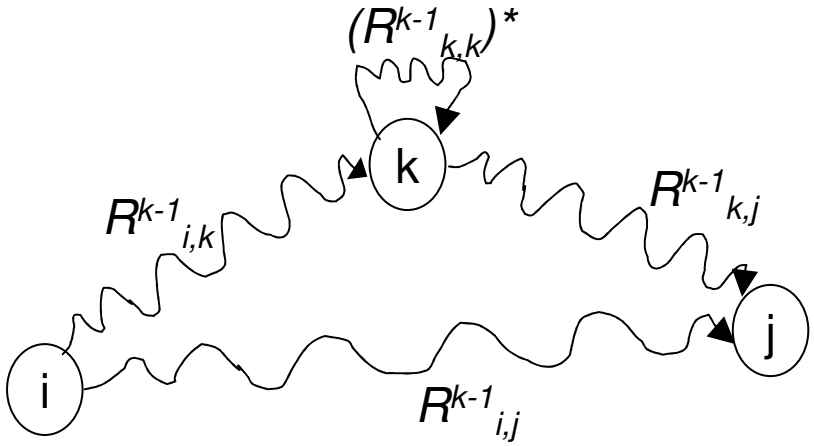


Algorithme de transformation d'un DFA en ER : *par recherche de chemin*

- Soit **A = <S, Σ, δ, s_0, F >** un automate déterministe
- Soit ***{1, 2, …, n}*** une image de S dans ***IN****
- On définit $R^k_{i,j}$ comme l'expression régulière décrivant le langage des mots ***w*** reconnus pas un chemin de l'état ***i*** à l'état ***j*** dans ***A***. Ce chemin ne contient pas d'état intermédiaire supérieur à ***k***.
- $L(R^k_{i,j})=\{w = u_1u_2\ldots u_{|w|} \in \Sigma^* \mid \delta(i, u_1) \leq k \wedge \delta(\delta(i, u_1), u_2) \leq k \wedge \delta(\delta(\delta(i, u_1), u_2), u_3) \wedge \delta(\delta(\ldots\delta(i, u_1), \ldots u_{|w|-1}), u_{|w|})=j \leq k \}$

Algorithme de transformation d'un DFA en ER : *par recherche de chemin*

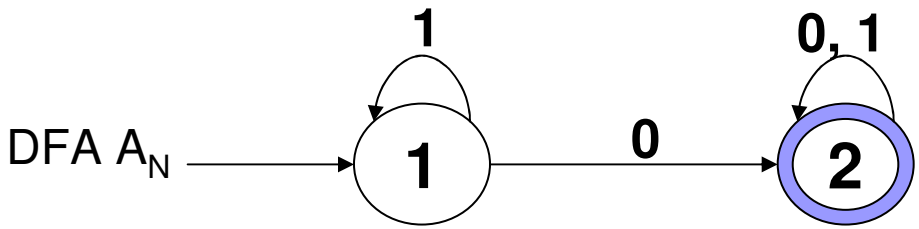
- $R^0_{i,j}=\{a \in \Sigma \mid \delta(i, a) = j\} \cup \{\varepsilon \text{ si } i = j\}$
- $\forall k \in \{1,..,n\}, R^k_{i,j} = R^{k-1}_{i,j} + R^{k-1}_{i,k} . (R^{k-1}_{k,k})^* . R^{k-1}_{k,j}$



- *ER(A) = expression régulière définissant les mots reconnus par les chemins allant de l'état initial vers l'état final et pouvant passer par les |S| états de l'automate A*
  - $ER(A) = R^{|S|}_{S_0,F}$

Exercice : Transformation DFA - ER

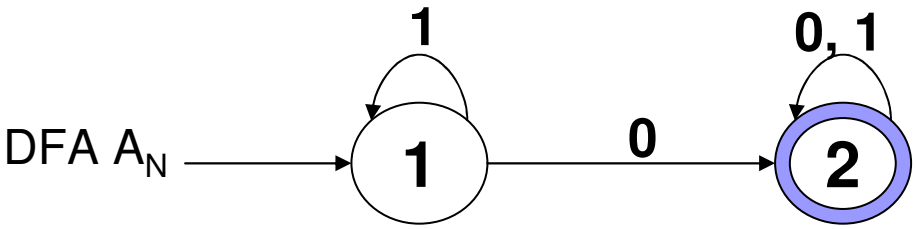
- Trouver l'expression régulière reconnu par l'automate suivant :



- Cela revient à calculer $R^{|S|}_{S_0,F} = R^2_{1,2}$

Solution

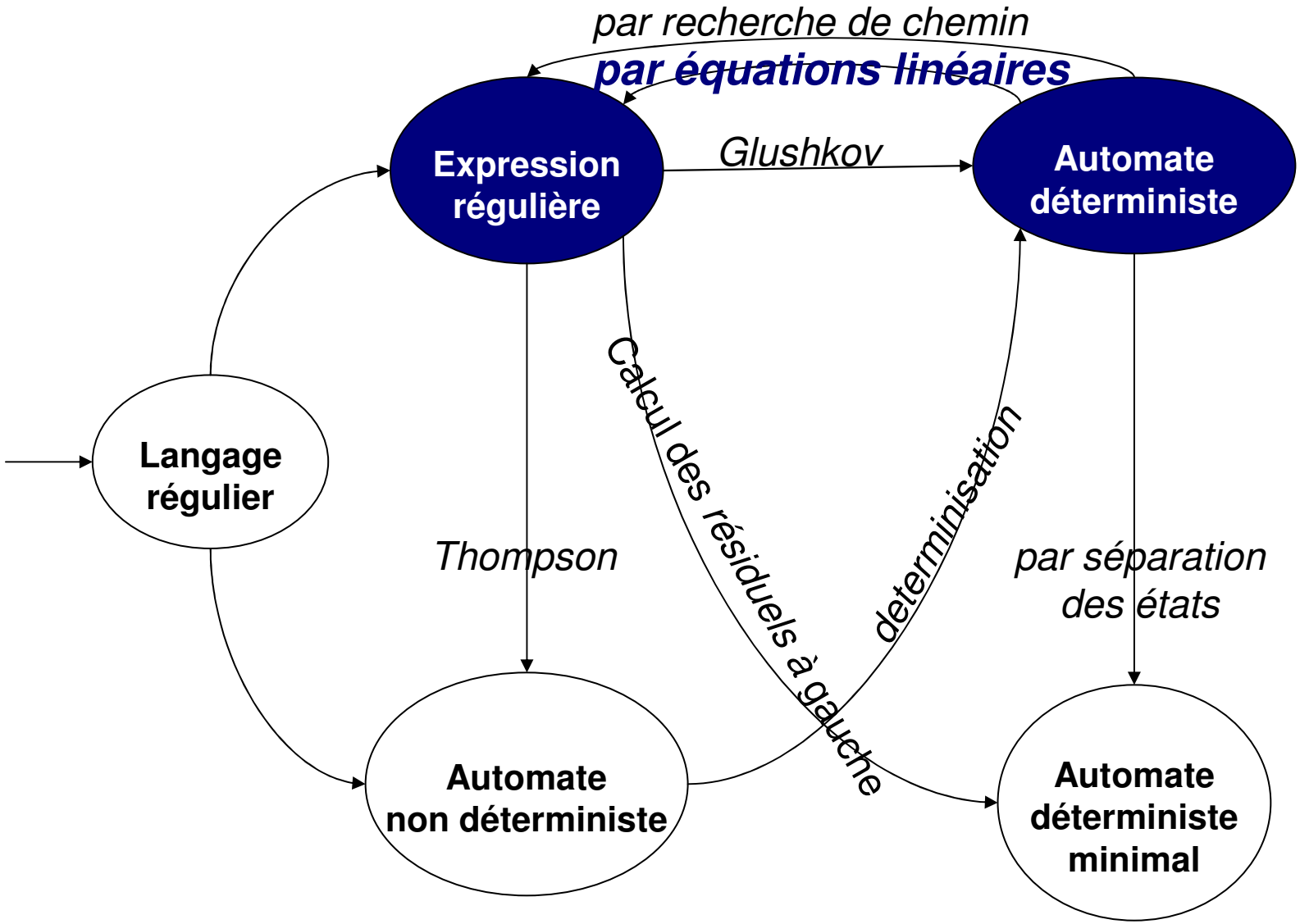
- $k = 0$
  - $R^0_{1,1} = 1 + \varepsilon$
  - $R^0_{1,2} = 0$
  - $R^0_{2,1} = \varnothing$
  - $R^0_{2,2} = 0 + 1 + \varepsilon$



- $k = 1$, $R^1_{i,j} = R^0_{i,j} + R^0_{i,1} . (R^0_{1,1})^* . R^0_{1,j}$
  - $R^1_{1,1} = (1 + \varepsilon) + (1 + \varepsilon).(1 + \varepsilon)^*.(1 + \varepsilon) = (1 + \varepsilon).(\varepsilon + (1 + \varepsilon)^+) = (1 + \varepsilon).(1 + \varepsilon)^* = (1 + \varepsilon)^+$
  - $R^1_{1,2} = 0 + (1 + \varepsilon).(1 + \varepsilon)^*.0 = 0 + (1 + \varepsilon)^+.0 = (1 + \varepsilon)^*.0$
  - $R^1_{2,1} = \varnothing + \varnothing.(1 + \varepsilon)^*.(1 + \varepsilon) = \varnothing$
  - $R^1_{2,2} = (0 + 1 + \varepsilon) + \varnothing.(1 + \varepsilon)^*.0 = (0 + 1 + \varepsilon)$

- $k = 2$,
  - $R^2_{1,2} = R^1_{1,2} + R^1_{1,2} . (R^1_{2,2})^* . R^1_{2,2}$
  - $= ((1 + \varepsilon)^*.0) + ((1 + \varepsilon)^*.0) .(0 + 1 + \varepsilon)^*.(0 + 1 + \varepsilon)$
  - $= ((1 + \varepsilon)^*.0) . (\varepsilon + ((0 + 1 + \varepsilon)^*.(0 + 1 + \varepsilon)))$
  - $= ((1 + \varepsilon)^*.0) . (0 + 1 + \varepsilon)^*$
  - $= (1^*.0) . (0 + 1)^*$
  - **$R^2_{1,2} = 1^* 0 (0 + 1)^*$**

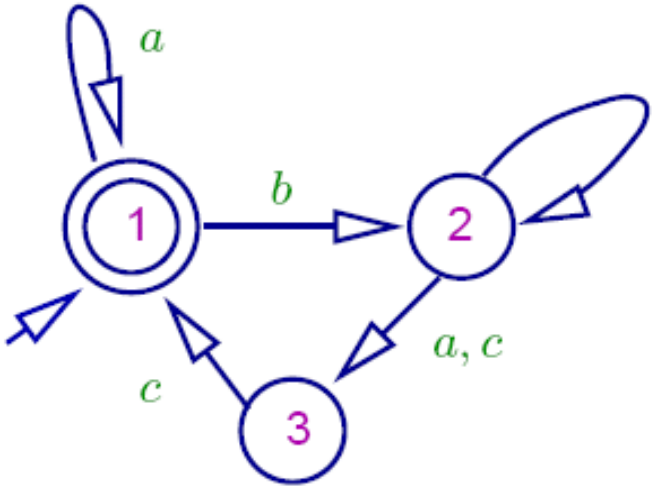
Algorithme de transformation d'un DFA en ER : *par recherche de chemin*



Algorithme de transformation d'un DFA en ER : *par équations linéaires*

- Soit A = <S, Σ, δ, s_0, F > un automate fini.
- SE(A) est le système d'équations de A
  - $X_{s \in S} = \sum_{\delta(s,a) = s'} aX_{s'}$ + (si $s \in F$ alors ε sinon ∅)

- exemple



$$\begin{cases} X_1 = aX_1 + bX_2 + \varepsilon \\ X_2 = bX_2 + (a + c)\, X_3 \\ X_3 = cX_1 \end{cases}$$

Algorithme de transformation d'un DFA en ER : *par équations linéaires*

- Théorème :
  - Soit A un automate et $(L_{s \in S})$ la plus petite solution associée à X_s dans SE(A), alors $L(A) = L_{s0}$

- Lemme
  - Soient A, B $\subseteq \Sigma^*$ des langages.
  - Le langage A*B est une solution de l'équation
    - X = AX + B

- Lemme d'Arden
  - Si $\varepsilon \notin$ A alors A*B est la solution unique de X = AX + B

Algorithme de transformation d'un DFA en ER – exemple

- Etape 0
  - $X_1 = aX_1 + bX_2 + \varepsilon$
  - $X_2 = bX_2 + (a + c)\, X_3$
  - $X_3 = cX_1$
- Etape 1
  - $X_1 = aX_1 + bX_2 + \varepsilon$
  - $X_2 = bX_2 + (a + c)\, cX_1$ (X_3 remplacée par sa valeur)
  - $X_3 = cX_1$

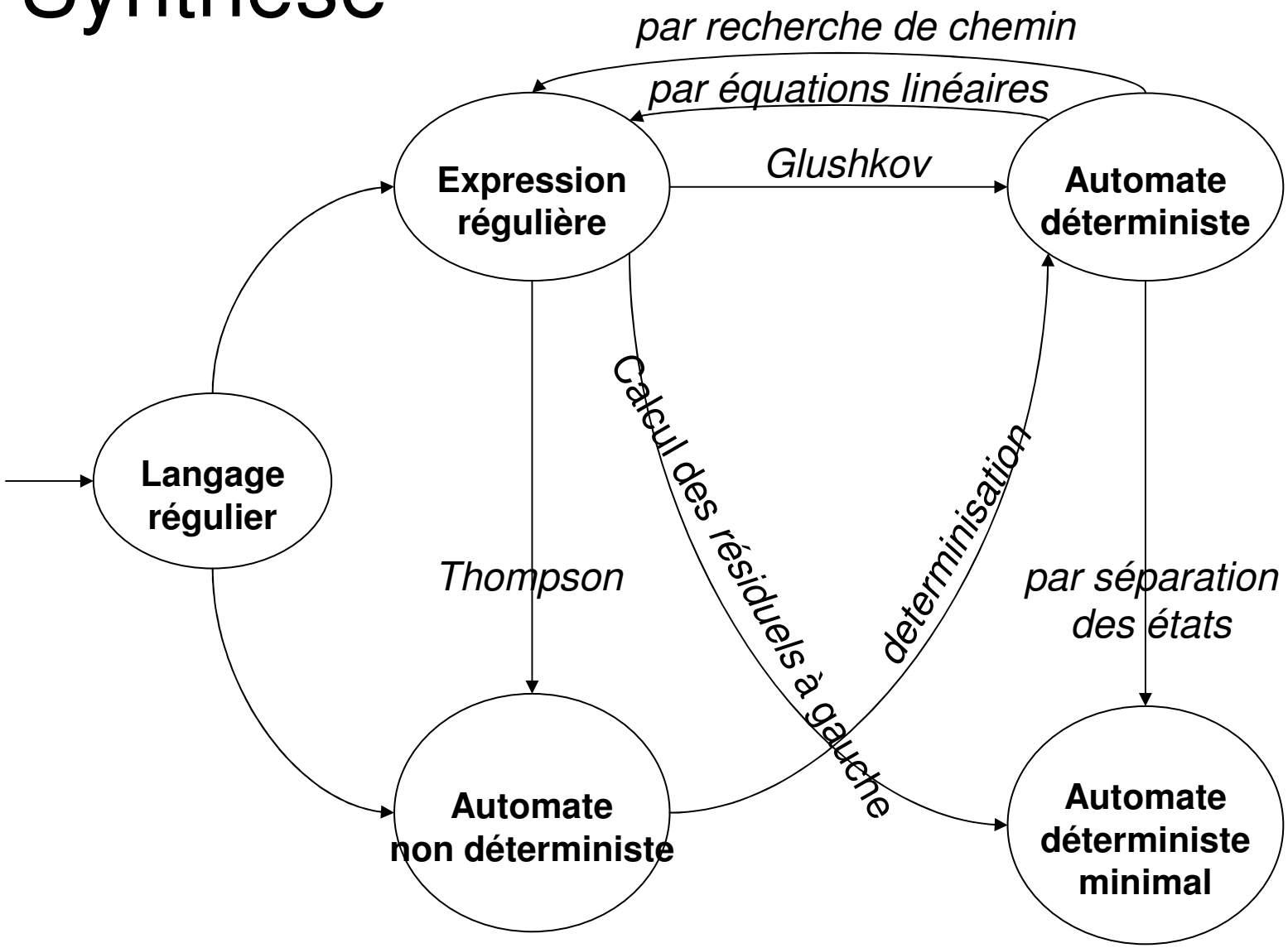
Algorithme de transformation d'un DFA en ER – exemple

- Etape 1
  - $X_1 = aX_1 + bX_2 + \varepsilon$
  - $X_2 = bX_2 + (a + c)\, cX_1$ (X_3 remplacée par sa valeur)
  - $X_3 = cX_1$
- Etape 2
  - $X_1 = aX_1 + bX_2 + \varepsilon$
  - $X_2 = b^*(a + c)cX_1$ (Résolution de X_2 par Application du lemme)
  - $X_3 = cX_1$
- Etape 3
  - $X_1 = aX_1 + b\, b^*(a + c)cX_1 + \varepsilon$ (X_2 remplacée par sa valeur)
  - $X_2 = b^*(a + c)cX_1$
  - $X_3 = cX_1$

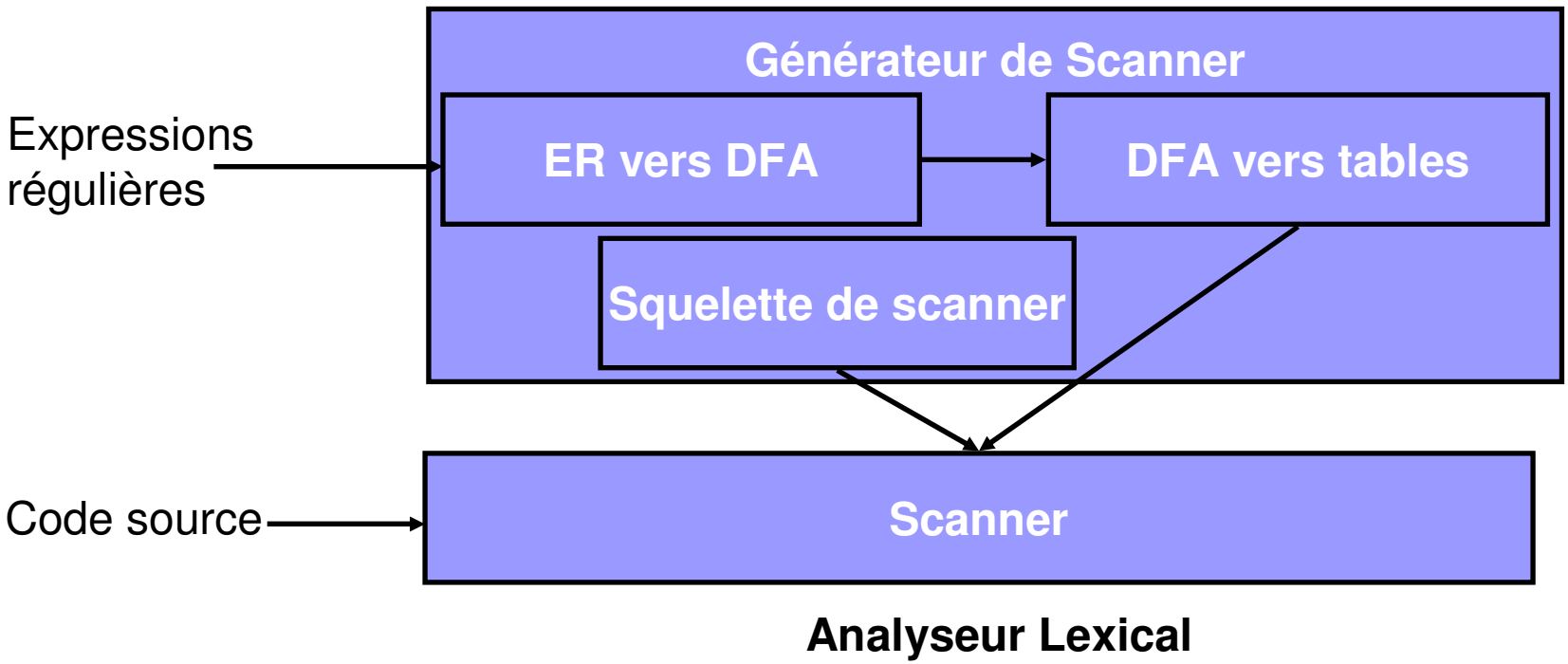
Algorithme de transformation d'un DFA en ER – exemple

- Etape 3
  - $X_1 = aX_1 + b\ b^*(a + c)cX_1 + \varepsilon$ (X_2 remplacée par sa valeur)
  - $X_2 = b^*(a + c)cX_1$
  - $X_3 = cX_1$
- Etape 4
  - $X_1 = (a + b\ b^*(a + c)\ c)X_1 + \varepsilon$ (factorisation car distributivité)
  - $X_2 = b^*(a + c)cX_1$
  - $X_3 = cX_1$
- **$L(A) = (a + b\ b^*(a + c)\ c)^*.\ \varepsilon = (a + b\ b^*(a + c)\ c)^* = (a + b^+(a + c)\ c)^*$**

Synthèse


par recherche de chemin
par équations linéaires
Expression régulière
Glushkov
Automate déterministe
Langage régulier
Thompson
Calcul des résiduels à gauche
déterminisation
par séparation des états
Automate non déterministe
Automate déterministe minimal

Génération automatique de scanner - Lex



Génération automatique de scanner – Lex/Flex

- Le générateur d'analyseur lexical (de scanner)
  1. prend en entrée un ensemble *d'expressions régulières*,
  2. pour chaque expression régulière, il *construit l'automate déterministe* (selon Glushkov),
  3. Convertit l'automate déterministe en programme exécutable en encodant les transitions en table indexée par les éléments de Σ (alphabet) et de S (états) et passe cela en paramètre au squelette de scanner qu'on a étudié auparavant.

Exemple d'utilisation de Lex

1. $cat example1.l

```
%{
#include <stdio.h>
%}
%option noyywrap
%%
[0123456789]+          printf("NUMBER");
[a–zA–Z][a–zA–Z0–9]*    printf("IDENTIFIER");
=                      printf("ASSIGNMENT");
–                      printf("MINUS");
\*                     printf("MULT");
\+                     printf("PLUS");
\"                     printf("QUOTE ");
\{                     printf("OBRACE ");
\}                     printf("EBRACE ");
;                      printf("SEMICOLON ");
[\n]+                  /* négliger caractères blancs : espaces et RC*/;
%%
```

(1) Production des ER dans un fichier Lex

2. $flex example1.l
3. $cc lex.yy.c –o example1 –lfl

(2) Génération automatique et (3) compilation d'un programme C implantant les ADF correspondant au fichier Lex

4. $./example1
   - C = A + B * A - 99
   - IDENTIFIER ASSIGNMENT IDENTIFIER PLUS IDENTIFIER MULT IDENTIFIER MINUS NUMBER

(4) Exécution du programme généré

Exercices en Analyse lexicale

Exercice – langage

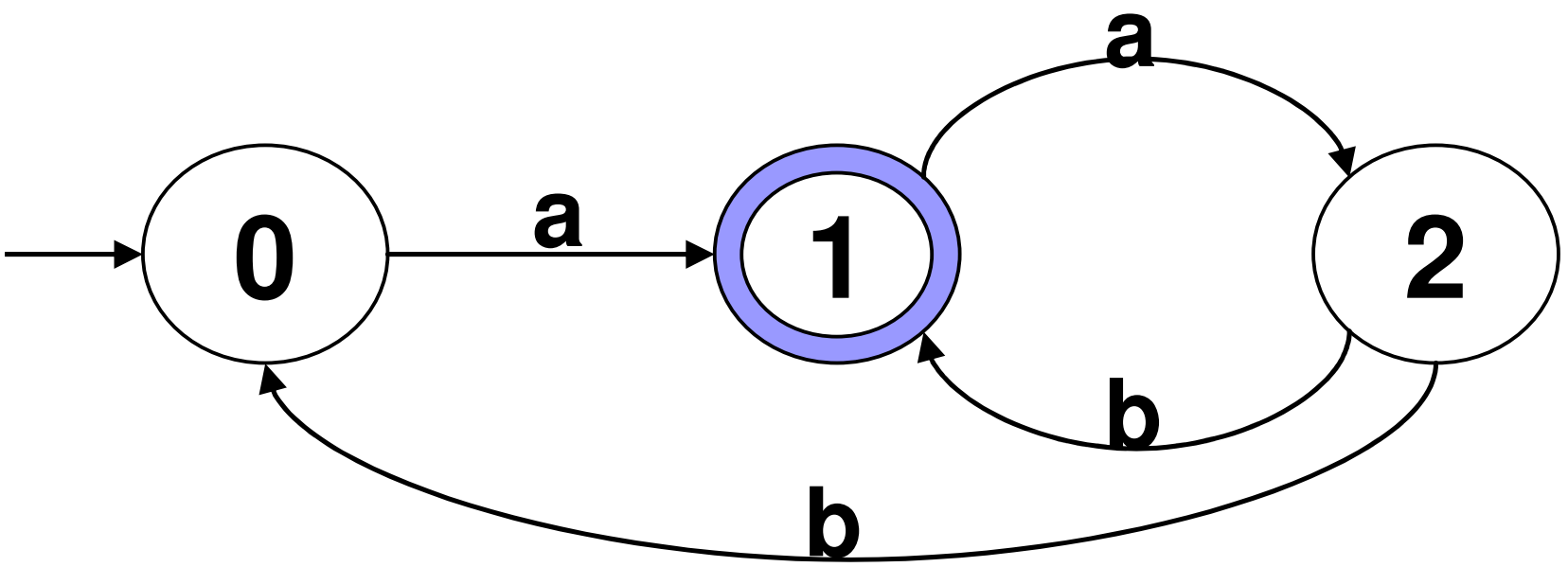
- Le mot *u* appartient-il au langage *L* dans les cas suivants ?
  - $u = a^3cbcbaa,\ L = (a^* cb)^*$
  - $u = a^3cbab,\ L = ((a + b)^* (a^2 + b^3)^* + c)^*$
  - $u = abcab,\ L = (1 + ab + c)^* \cap (a + bc)^*$

Exercice – langage

- L'un des deux langages est inclus dans l'autre ? Si oui lequel
  - L1 = ((1 + ab)* a*)* ; L2 = (ab + ba)*
  - L3 = (a*b)* ; L4 = (a + b)*

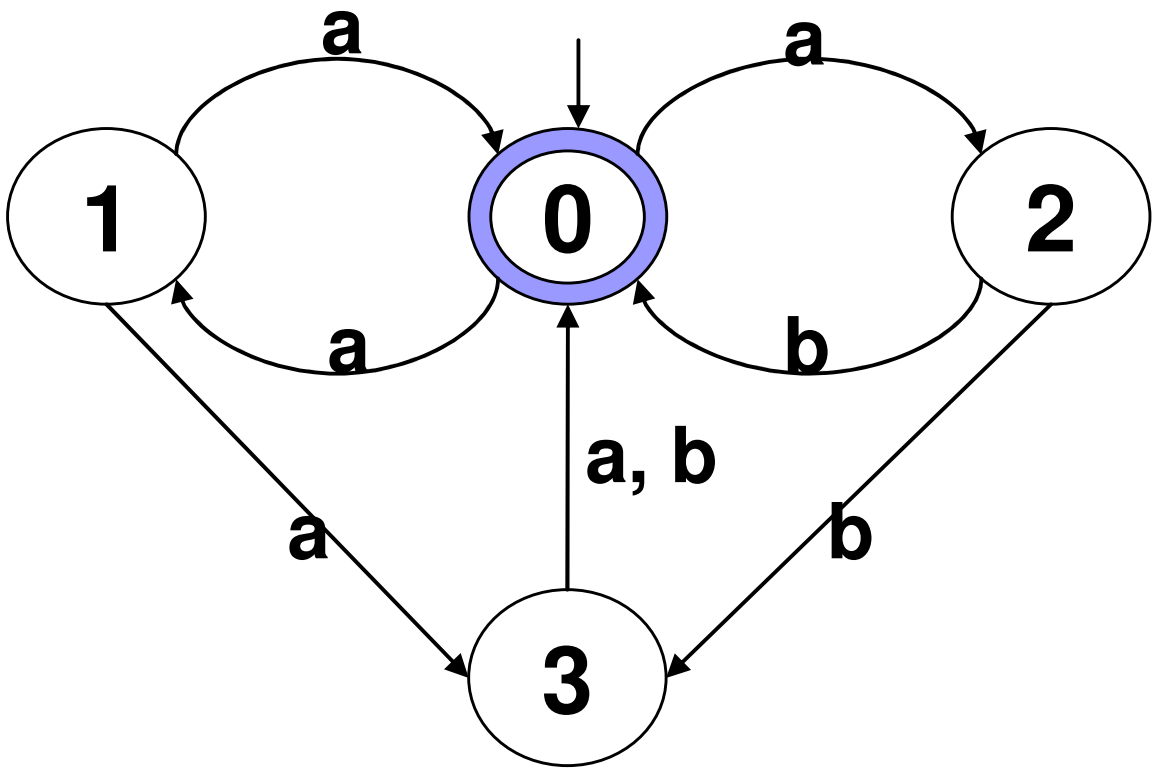
Exercice – automate

- Le mot *u* est-il accepté par l'automate ?
  - *2.a) $u = a^2ba$*



Exercice – automate

- Le mot *u* est-il accepté par l'automate ?
  - *2.b) $u = aba^3b$*



Exercice – langage et automate

- Soit l'alphabet $\Sigma = \{a,b\}$, donner un automate déterministe qui reconnaît chacun des langages suivants (un automate par langage)

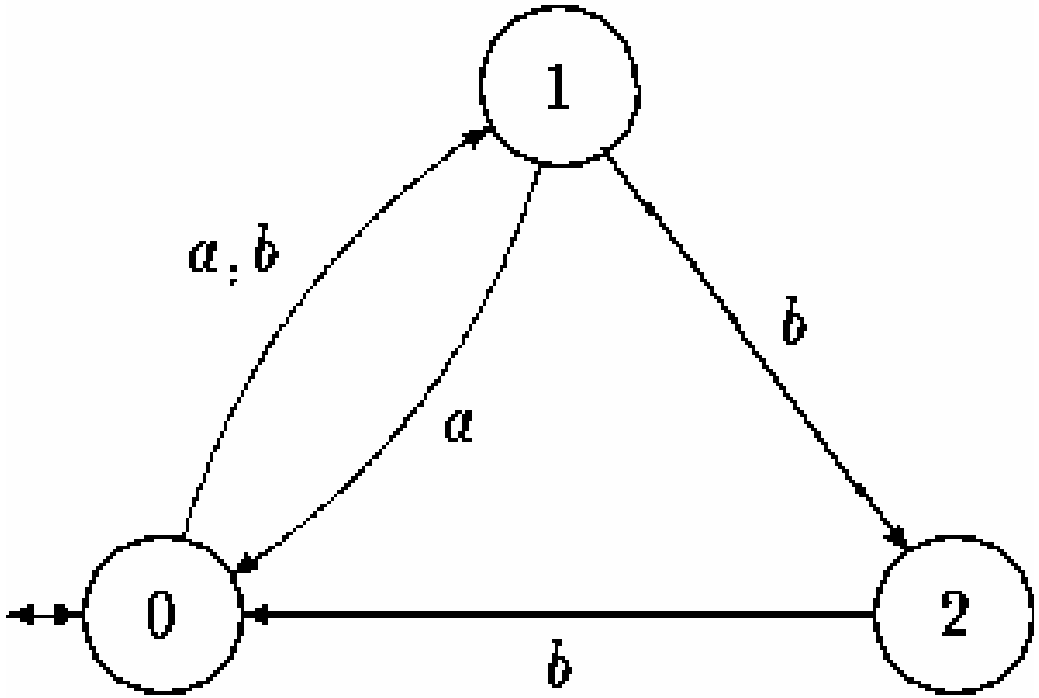
  a. $L0 = \{ w \in \Sigma^* \mid w = a^n b^n \text{ où } n \geq 0\}$
  b. $L1 = \{w \in \Sigma^* \mid w \text{ ne contient pas le facteur } a^2\}$ (i.e. w ne contient pas deux fois de suite a)
  c. $L2 = \{w \in \Sigma^* \mid |w|_b \text{ est divisible par trois}\}$
  d. $L3 = \{w \in \Sigma^* \mid |w|_b = 1\}$

Exercice – Expression régulière

- Simplifiez les expressions suivantes
  - $a^*b^* \cup b^*$
  - $((aba)^*)^*$
  - $(a^*b)^*a^*$
  - $ab^* \cup aa\ (ca)^*(ab^*a)^*$
  - $(a(ab)^*)^* \cup a^*$
- Donner leur automate déterministe équivalent par Glushkov
- Donner leur automate minimal équivalent par calcul des résiduels à gauches

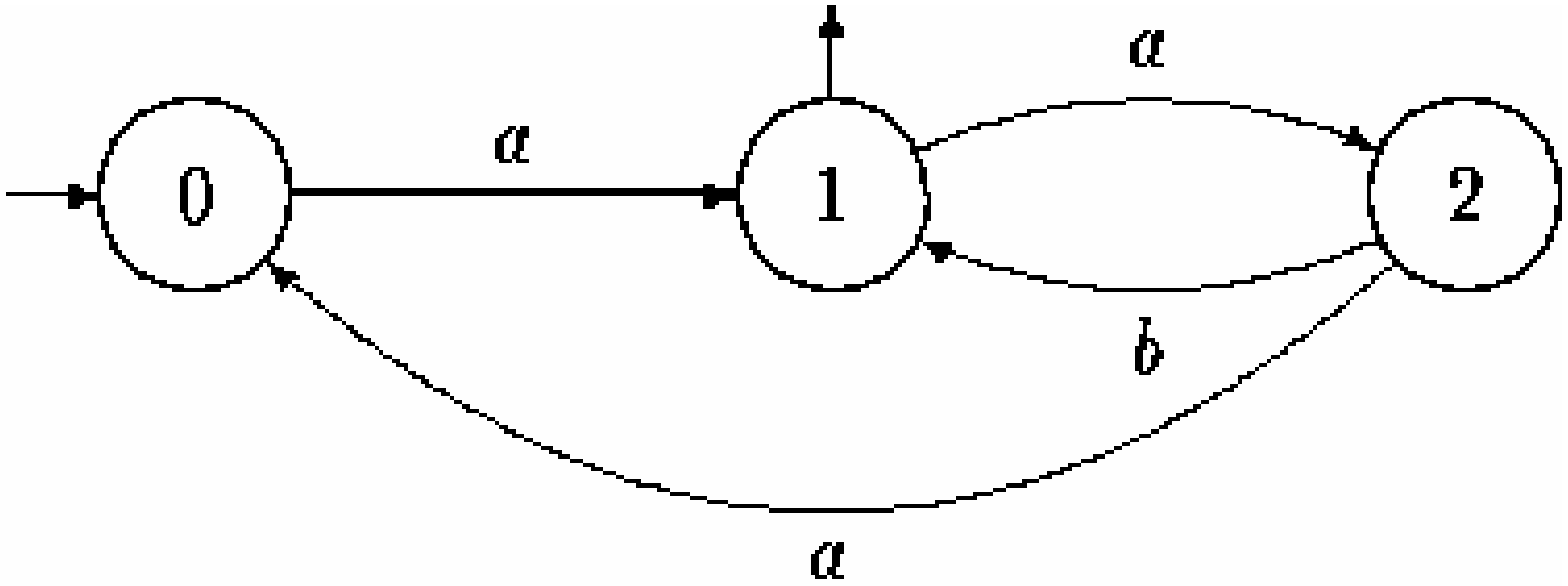
Exercice – Expression régulière et automate

- Donner le langage reconnu par l'automate suivant



Exercice – Expression régulière et automate

- Donner le langage reconnu par l'automate suivant



Exercice – automates NFA / DFA

- Sur l'alphabet $\Sigma = \{a,b\}$,

a. donner un automate non-déterministe qui reconnaît le langage suivant $L = \{w \in \Sigma^* \mid w \text{ se termine par le facteur } ab\}$

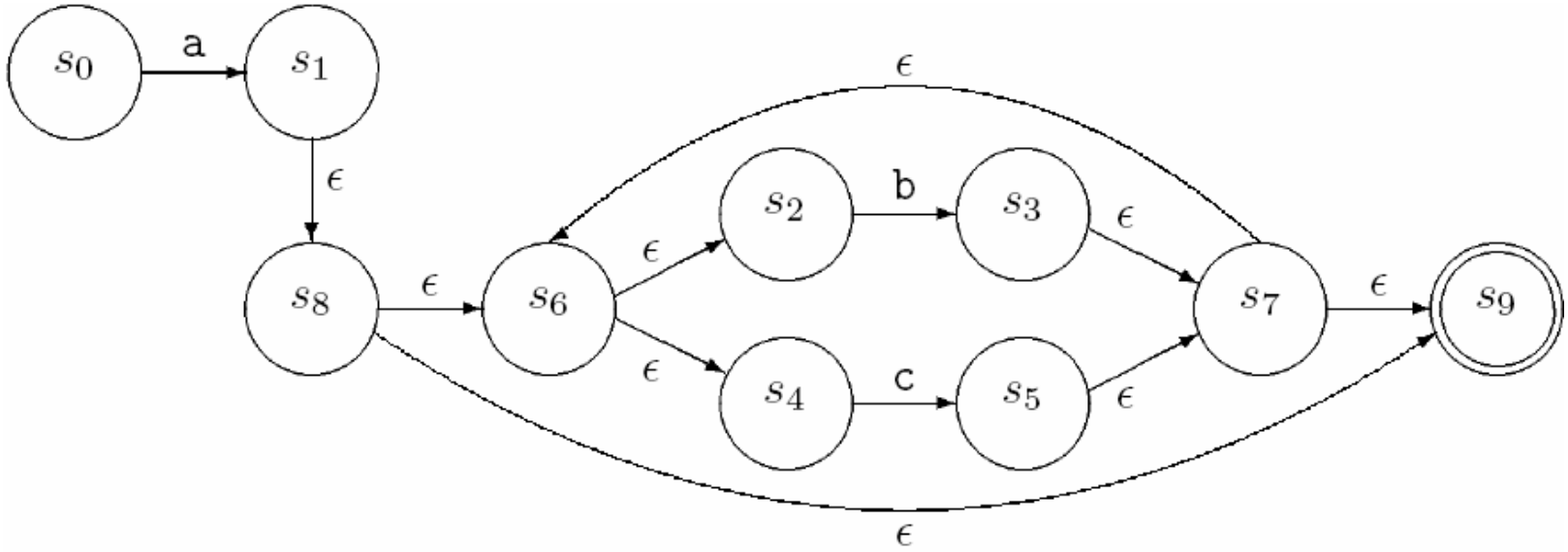
b. donner l'automate déterministe qui lui est équivalent

Exercice – transformation ER en NFA/DFA

- Construire les automates reconnaissant les langages réguliers suivants par (NFA) Thompson et (DFA) par Glushkov
  - *(b|c)*
  - *(b|c)**
  - *a(b|c)**
  - *(010|1)**
  - */'*'(a...z)*'*'/*

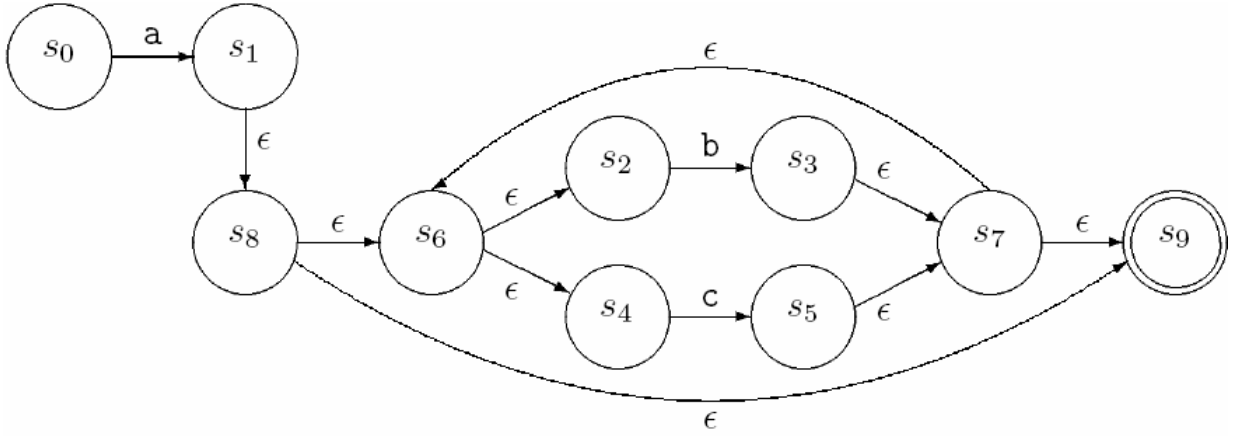
Exercice – determinisation d'automates

1. Construire intuitivement un automate déterministe sans ε-transitions reconnaissant le langage régulier *a(b|c)**
2. Puis construire un automate déterministe sans ε-transitions équivalent à l'automate ε-NFA suivant

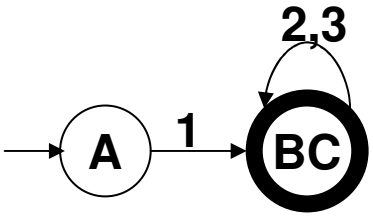


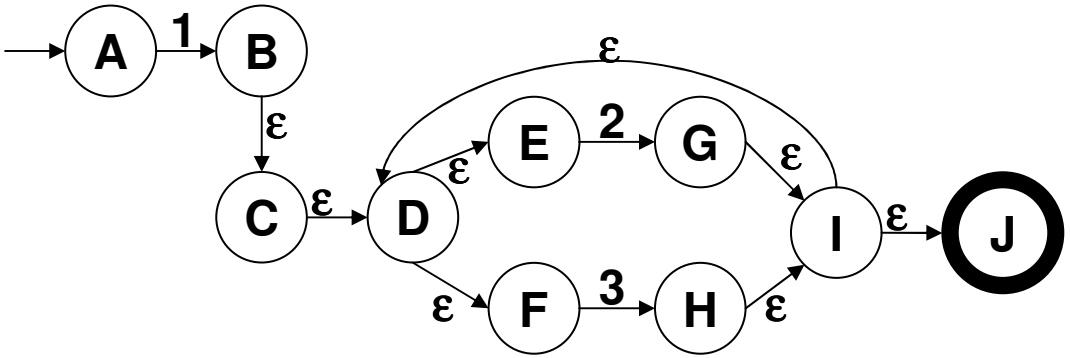
NFA *for* a(b | c)*

Exercice – determinisation d'automates



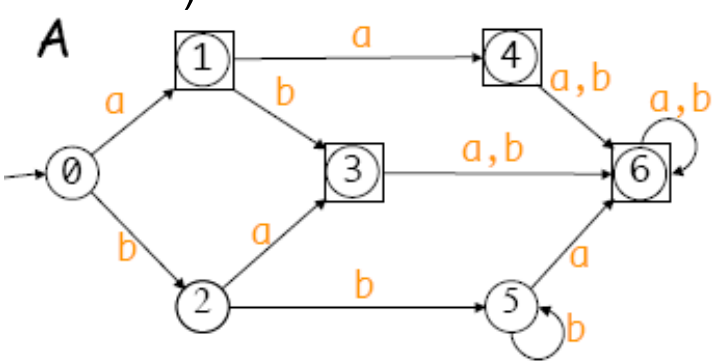
NFA *for* a(b | c)*

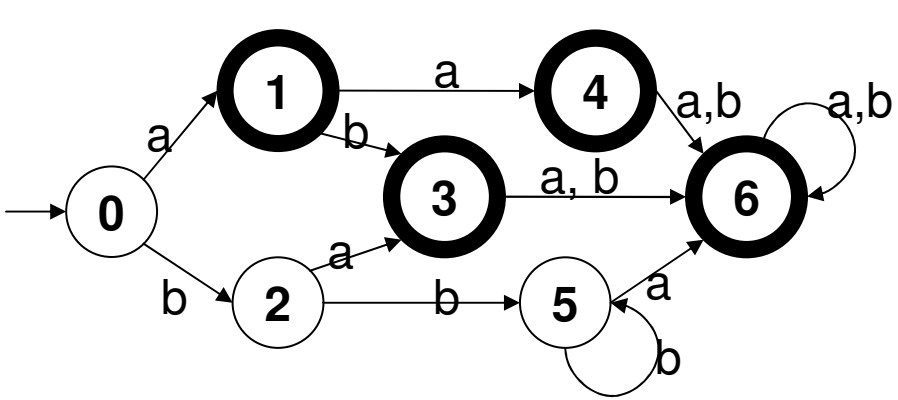


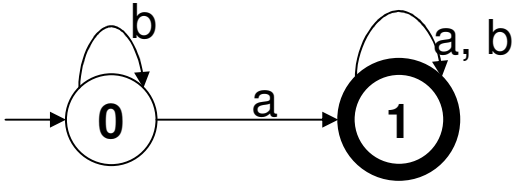


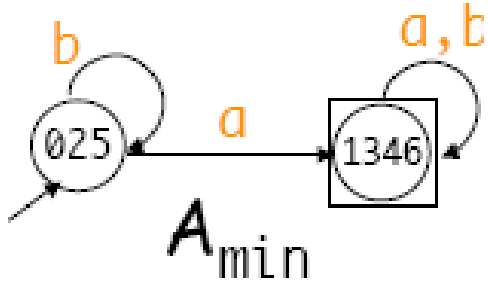
Exercice – minimisation d'automates

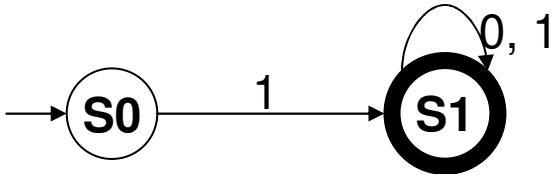
1. Démontrer que A_{min} (resp. B_{min}) est l'automate équivalent à A (resp. à B) par équivalence de leur expression régulière
2. Démontrer que A_{min} (resp. B_{min}) est l'automate minimal de A (resp. de B)





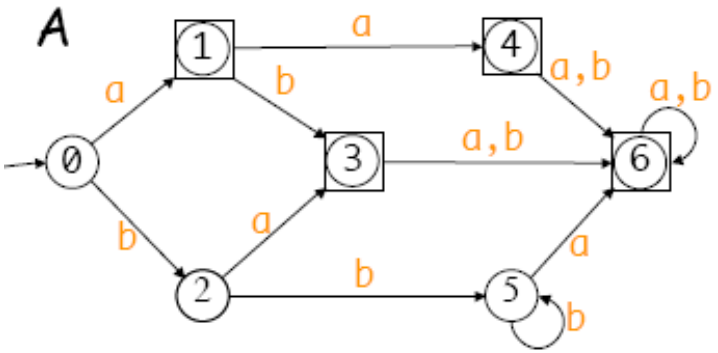


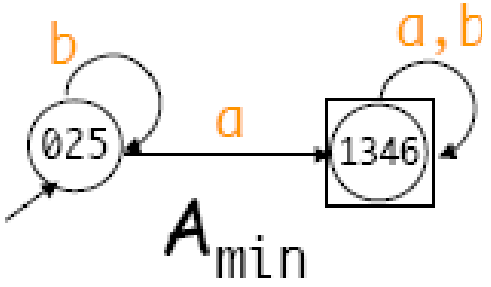


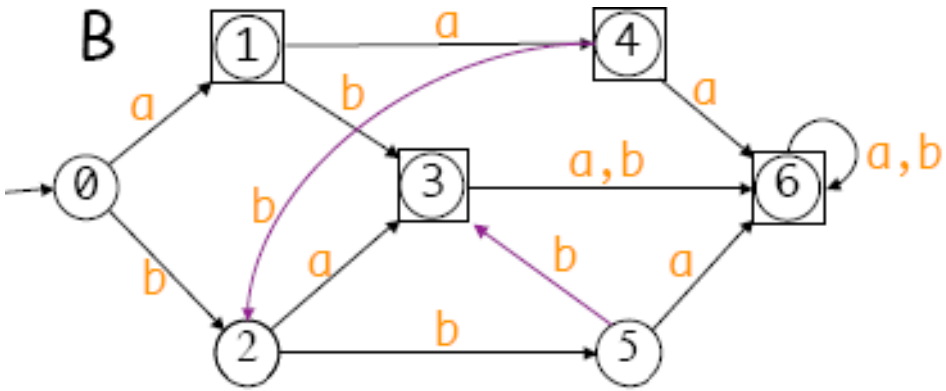


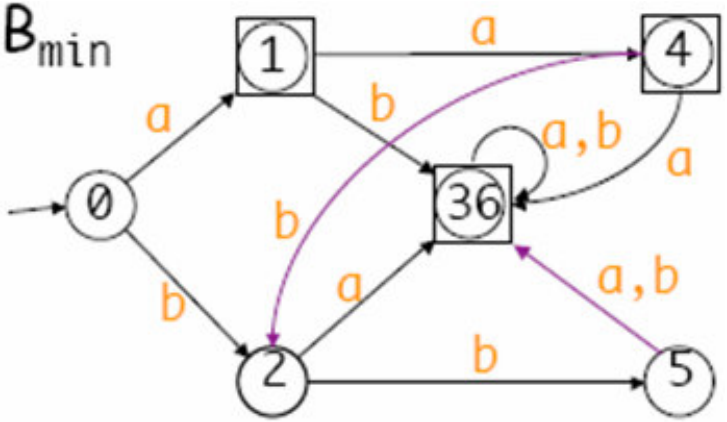
Exercice – minimisation d'automates

1. Démontrer que A_{min} (resp. B_{min}) est l'automate équivalent à A (resp. à B) par équivalence de leur expression régulière
2. Démontrer que A_{min} (resp. B_{min}) est l'automate minimal de A (resp. de B)









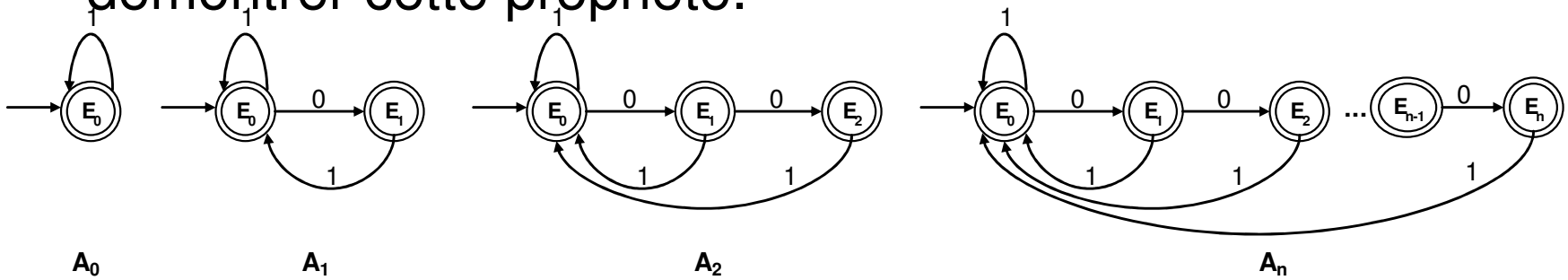
Exercices – divers

- Démontrer que tout langage fini est régulier
- Démontrer que $\{\varepsilon\}$ est un langage régulier
- Démontrer que $\{a^{2^n}, n \in \text{IN}\}$ n'est pas régulier
- Montrez que pour tout langage régulier L1 et L2, le langage $L1 \cap L2$ est régulier
- Est-ce que l'intersection de deux langages réguliers est un langage régulier ? Justifiez !
- Est-ce que le complémentaire d'un langage régulier L $(\Sigma^* \setminus L)$ est un langage régulier ? Justifiez !

Exercices – divers

- Si la complexité en temps de la fonction de transition δ est en $O(1)$ quelle est la complexité en temps de l'algorithme non optimisé de transformation d'un automate non déterministe sans epsilon transition $\mathbf{A_N = <S_N, \Sigma_N, \delta_N, s_{N_0}, F_N >}$ en un automate déterministe

- Soit (Ai)i=0..n une suite d'automates d'états finis sous la forme de la figure 1 (**a**) donner et démontrer la forme régulière générale des langages L(Ai)i=0..n (**b**) quelle est la propriété conservée par la suite (L(Ai))i=0..n (**c**) démontrer cette propriété.



Exercice 6

- Considérez l'expression régulière suivante qui reconnaît des noms de registres :

α = R0 / R00 / R1 / R01 / R2 / R02 / ... / R30 / R31

a. Trouver l'automate non-déterministe N à partir de l'expression régulière α
b. Trouver l'automate déterministe D équivalent à N
c. Trouver l'expression régulière à partir de D
d. Expliquer toute différence existant entre l'expression régulière original α et celle produite par (c)

Fin Rappels

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

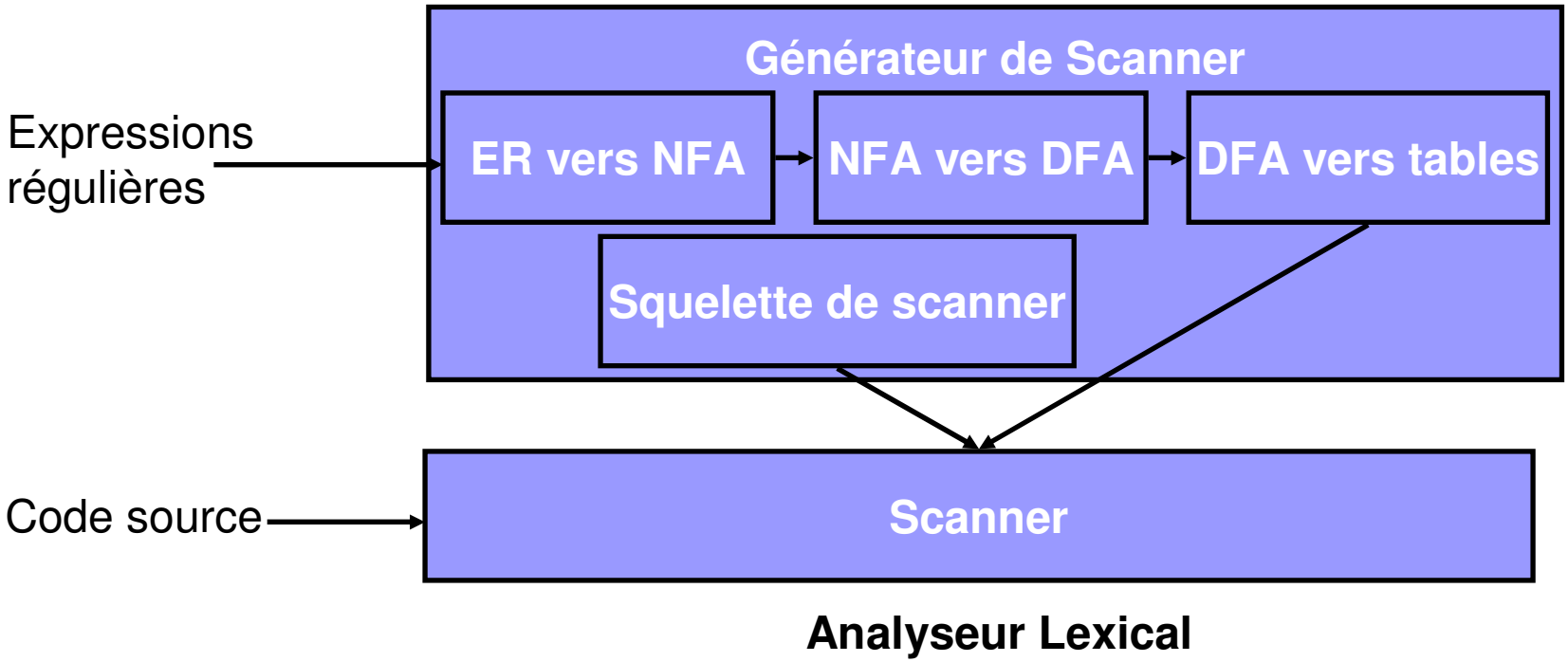
Générateur automatique d'analyseur lexical (FLEX)

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Génération automatique de scanner - Lex



FLEX (Fast Lex)

- Flex est un outil de génération de scanners: analyseurs lexicaux

1. Flex lit un fichier en entrée contenant la spécification du scanner à générer. Chaque élement de la spécification est sous forme d'un couple appelée règle FLEX. Une règle FLEX est formée :
    a. d'une expression régulière (dite étendue car plus riche que les langages réguliers de base)
    b. Et d'une action : code en C associé à l'expression régulière
2. Flex génère comme résultat un fichier C, 'lex.yy.c', qui définit une fonction 'yylex()'

3. Le fichier 'lex.yy.c' est compilé et l'édition de liens est faite avec la librairie '-lfl' pour produire un exécutable

4. Quand l'exécutable est exécuté, il analyse son entrée recherchant des occurrences des expressions régulières. A chaque succès, il exécute le code source qui lui est associée.

Scanner pour une sous partie du langage Pascal - règles

```
%{      /* need this for the call to atof() below */
        #include <math.h>
%}
/* yywrap : fonction  (ici omise) supportant l'analyse de multiples fichiers paramètres comme un fichier unique */
%option noyywrap
/*Les expressions régulières simples et réutilisables */
DIGIT                                   [0-9]
IDF                                     [A-Za-z][A-Za-z0-9]*

%%

/*Les expressions régulières complexes avec leur action */
{DIGIT}+"."{DIGIT}*                     { printf( "FLOAT(%s) ", yytext);}
{DIGIT}+                                { printf( "INT(%s) ", yytext);}
":="                                    { printf( "ASSIGN(%s) ", yytext);}
"("                                     { printf( "<PO> ");}
")"                                     { printf( "</PO> ");}
"program"|"if"|"then"|"begin"|"end"|"procedure"|"function"      { printf( "KEY(%s) ", yytext );}
{IDF}                                   { printf( "IDF(%s) ", yytext );}
"+"|"-"|"*"|"/"                         { printf( "OPR(%s) ", yytext );}
[\n]                                    { printf( "\n");}
"{"[^}\n]*"}"                           /* Consommer one-line comments : aucune action*/
[ \t]+                                  /* Consommer les caractères espaces, et tabulations : aucune action */
.                                       { printf( "ERRORLEX(%s) ", yytext );}
                                        /* joker(.) est toujours placé après toutes les règles */

%%
```

ordre d'interprétation des règles

Scanner pour une sous partie du langage Pascal - main

- main(argc, argv)
  - int argc;
  - char **argv;
- {
  - ++argv, --argc; /* saute le nom du programme */
  - if (argc > 0)
    - yyin = fopen(argv[0], "r");
  - else
    - yyin = stdin;
  - yylex();
- }

Quelques variables FLEX prédéfinies

- FILE* yyin : entrée du scanner
- FILE* yyout : sortie du scanner
- char* yytext : l'image textuelle du token courant
- int yylen : taille du token courant

Description des expression régulières en FLEX (1)

- **x** le caractère ‘x’
- **.** dit “jocker”, n’importe quel caractère (octet) excepté ‘\n’
- **[xyz]** ensemble des caractères ‘x’, ‘y’, ou ‘z’
- **[abj-oZ]** ensemble des caractères ‘a’, ‘b’, ou appartenant à l’intervalle ‘j’-‘o’, ou le cacatère ‘Z’
- **[^A-Z]** complément de l’ensemble des cacartères entre ‘A’ et ‘Z’
- **[^A-Z\n]** tout caractère excepté les majuscules et le cacatère ‘\n’
- **"[xyz]\"foo"** la chaîne de cacatères : ‘[xyz]"foo’

Description des expression régulières en FLEX (2)

- **{r}** l'expansion de la définition de l'expresson régulière r
- **(r)** r ; les parenthèses forcent l'interprétation des ER dans un certain ordre.
- **r*** 0 ou n fois r, où rest une expression régulière
- **r+** 1 ou n fois r
- **r?** 0 ou 1 fois r ("r est optionel ")
- **r{2,5}** 2 jusqu'à 5 fois r
- **r{2,}** 2 fois r ou plus
- **r{4}** exactement 4 fois r

  - Quelques priorités : '*' est plus prioritaire que la concaténation
  - la concaténation est plus prioritaire que le choix ('|')

Description des expression régulières en FLEX (3)

- **rs** l'expression régulière r suivie de l'expression régulière s ("concatenation")
- **r | s** r ou s ("choix ou union")
- **^r** r en début de ligne (début de fichier ou après '\n')
- **r$** r en fin de ligne (avant un '\n'), équivalent à "r/\n"
- **r/s** r seulement si suivi de s. Le texte correspondant à s est retourné à l'entrée avant l'exécution de l'action. L'action verra seulement le texte correspondant à r

- **<s>r** r avec une condition s (start condition) (E**C**A)
- **<s1,s2,s3>r** r avec une des conditions s1, s2, ou s3
- **<*>r** r avec n'importe quelle condition (même exclusive)
- **<<EOF>>** caractère end-of-file

Utilisation de FLEX

- Génération de scanner mini-pascal
  - $ /usr/bin/flex.exe –i pascal.lex
  - produit un fichier lex.yy.c (l'option –i pour ignorer la différence majuscule/minuscule (indifférence à la casse)
- Compilation du code généré
  - $ gcc lex.yy.c -o pascal -lfl

Un exemple en pascal – exercice analysez les erreurs lexicales selon les règles flex précédentes

- $ cat example.pas

```
1.  Program NotALott(input,output);
2.  var
    1.  Count, Items : integer ;
    2.  Size : real ;
    3.  Key : char ;
    4.  Name : string [6] ;
    5.  InputLine : string [80] ;
    6.  IdCode : byte ;
    7.  EndHit : boolean ;
3.  begin
    1.  Count := 0 ;
    2.  Items := Count + 1 ;
    3.  Name := ‘Fred' ;
    4.  Readln (InputLine) ;
    5.  Size := 1.232 ;
    6.  IdCode := 99 ;
    7.  EndHit := (IdCode = 99) ;
4.  end .
```

L’analyseur lexical en action

- ./pascal example.pas

- KEY(program) IDF(NotALott) <PO> IDF(input) **ERRORLEX(,)** IDF(output) </PO> **ERRORLEX(;)**
- IDF(var)
- IDF(Count) **ERRORLEX(,)** IDF(Items) **ERRORLEX(:)** IDF(integer) **ERRORLEX(;)**
- IDF(Size) **ERRORLEX(:)** IDF(real) ERRORLEX(;)
- IDF(Key) **ERRORLEX(:)** IDF(char) ERRORLEX(;)
- IDF(Name) **ERRORLEX(:)** IDF(string) **ERRORLEX([)** INT(6) **ERRORLEX(]) ERRORLEX(;)**
- IDF(InputLine) **ERRORLEX(:)** IDF(string) **ERRORLEX([)** INT(80) **ERRORLEX(]) ERRORLEX(;)**
- IDF(IdCode) **ERRORLEX(:)** IDF(byte) **ERRORLEX(;)**
- IDF(EndHit) **ERRORLEX(:)** IDF(boolean) **ERRORLEX(;)**
- KEY(begin)
- IDF(Count) ASSIGN(:=) INT(0) **ERRORLEX(;)**
- IDF(Items) ASSIGN(:=) IDF(Count) OPR(+) INT(1) **ERRORLEX(;)**
- IDF(Name) ASSIGN(:=) ERRORLEX(') IDF(Fred) **ERRORLEX(') ERRORLEX(;)**
- IDF(Readln) <PO> IDF(InputLine) </PO> **ERRORLEX(;)**
- IDF(Size) ASSIGN(:=) FLOAT(1.232) **ERRORLEX(;)**
- IDF(IdCode) ASSIGN(:=) INT(99) **ERRORLEX(;)**
- IDF(EndHit) ASSIGN(:=) <PO> IDF(IdCode) **ERRORLEX(=)** INT(99) </PO> **ERRORLEX(;)**
- KEY(end) **ERRORLEX(.)**

Exercice 1

- Que fait le programme suivant ?
  - */* yywrap : fonction (ici omise) supportant l'analyse de multiples fichiers paramètres comme un fichier unique */*
  - *%option noyywrap*
  - %%
  - “/*” printf(“<SE>”);
  - “*/” printf(“<ASE>”);
  - “\\n” printf(“<ASN>”);
  - “(”|“[“|“{” printf(“<PO>”);
  - “]”|“}“|“)” printf(“<PF>”);
  - “?”+ printf(“<?;%d>”,yylen);
  - . ;
  - %%

  - int main(){
    - yylex();
    - return 0;
  - }

Exercice 2 - CompteMots

- 1. Générer avec Flex un programme **CompteMots** s'inspirant la commande Unix **wc**: le but est de compter le nombre de caractères, mots et lignes contenus dans un fichier texte.
  - On définira un **mot** comme une séquence maximale de caractères ne contenant ni espace, ni tabulation, ni retour ligne.
- 2. Spécifier cette même solution pour détecter le nombre de formulaire (<form ….>…. </form>) dans un fichier HTML

Solution possible

- /* **Hypothèse 1**:le EOF n'est pas considéré comme caractère* /
- /* **Hypothèse 2**: Toute suite de caractères blancs ou pas suivie d'un EOF est considérée comme une ligne* /

- int wc = 0, cc = 0, lc = 0;

- %%
- [^\t\n]+<<EOF>> /*un mot d'une ligne se terminant par un EOF !! [H1 & H2] */
- {wc++; cc+= yylen-1; lc++;}
- [\t]+<<EOF>> /*des caractères blancs d'une lignes se terminant par un EOF !! [H1 & H2] */
- {cc+= yylen-1; lc++;}
- ^<<EOF>> /*ce n'est ni un caractère ni un mot ni une ligne [H1 & H2] s*/
- [^\t\n]+ /* un mot*/ { wc++; cc+= yylen; }
- [\n]+ /* des fins de ligne */ { cc+=yylen; lc+=yylen; }
- [\t]+ /* des caractères blancs contigus */ {cc+=yylen; }
- %%

- main(){
  - yyin = ….
  - yylex();
  - printf("lc = %d, wc = %d, cc = %d", lc, wc, cc);
- }

Compilation

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Lectures recommandées - livres

1. « Engineering a Compiler » ©, K. D. Cooper & L. Torczon, Morgan Kaufmann 2004
2. « Introduction to Automata Theory Languages and Computation » ©, J. E. Hopcroft, R. Motwani, J. D.Ullman, 2001, Addison Wesley
3. « Compilers and Compiler Generators » ©, P.D. Terry, 2000
4. « Compilateurs avec C++ », Jacques Menu, 1994, Addison Wesley
5. « *Bottom-Up Parsing* », Maggie Johnson & Julie Zelenski, 2008

Plan du cours

1. Module initial Motivation & Définitions
2. Module 1 Analyseur lexical *(langages réguliers et théorie des automates)*
3. **Module 2** Analyseur syntaxique *(langages hors contexte)*
4. Module 3 Analyseur sémantique *(grammaires attribuées)*
5. Module 4 Représentations Intermédiaires
6. Module 5 Génération de code intermédiaire
7. Module 6 Optimisation et Génération de code
8. Conclusion et perspectives

Module 2
Analyseur syntaxique
ou Langages hors contexte

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Analyseur syntaxique – Plan

- Introduction
- Concepts
  - Grammaire Hors-Contexte
  - Grammaire et Dérivation
  - Arbre syntaxique
  - Grammaire Hors-Contexte Linéaires
  - Langages réguliers et HC linéaires
  - Dérivation gauche/droite
- Transformation
  - Éliminer l'Ambiguïté d'une grammaire HC
  - Éliminer la récursivité gauche d'une grammaire HC
  - Rendre une grammaire HC prédictive
- Parsers (analyseurs syntaxiques)
  - Types de parsers : top-bottom LL, bottom-up LR, etc.
- Générateur automatique d'analyseur syntaxique – YACC / Bison

Analyseur syntaxique

- L'analyseur syntaxique vérifie que **l'ordre des tokens** correspond à l'ordre définit pour le langage. On dit que l'on vérifie la **syntaxe** du langage à partir de la définition de sa grammaire

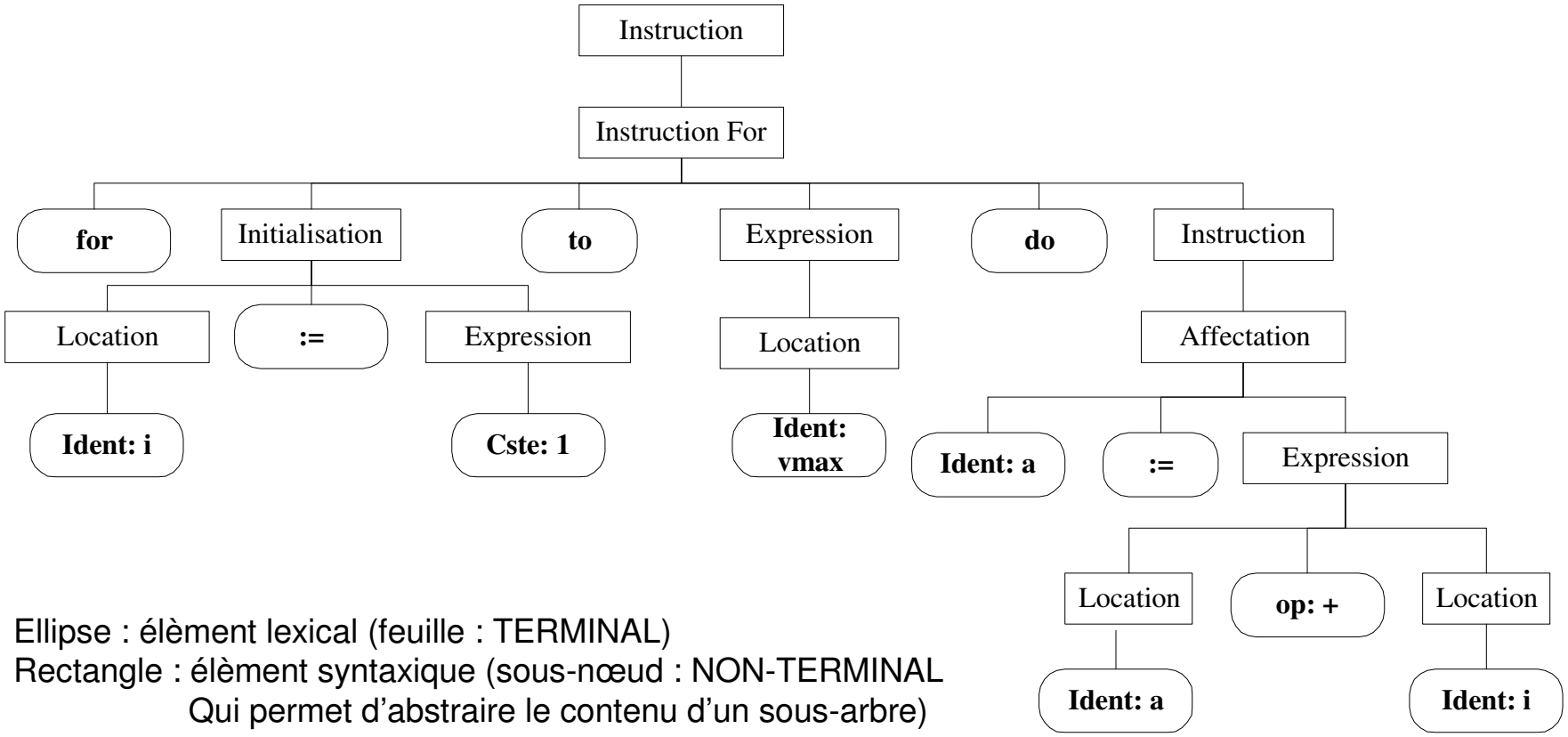
- L'analyse syntaxique produit une représentation sous forme **d'arbre de la suite des tokens** obtenus lors de l'analyse lexicale

Analyseur syntaxique

- Pour effectuer efficacement une analyse syntaxique, le compilateur nécessite :
  1. Une **définition formelle** du langage source,
  2. Une **fonction indicatrice** de l'appartenance d'un programme au langage source,
  3. Un plan de **gestion des entrées illégales**.

Analyseur syntaxique

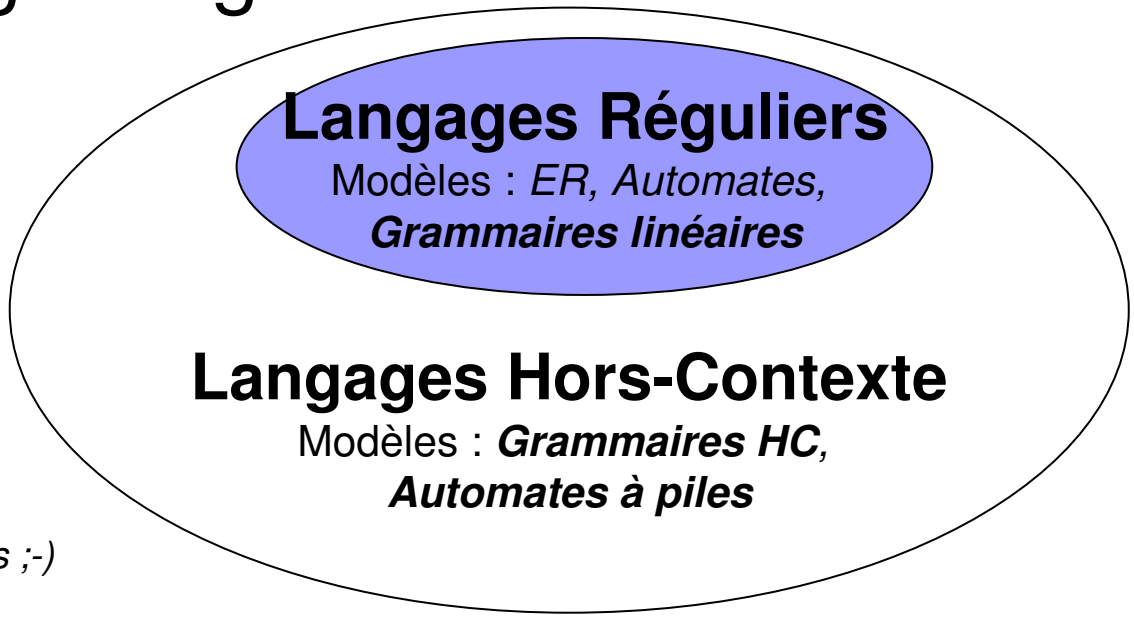
- Arbre syntaxique suivant représente la structure de : for i :=1 to vmax do a :=a+i



Ellipse : élèment lexical (feuille : TERMINAL)
Rectangle : élèment syntaxique (sous-nœud : NON-TERMINAL
Qui permet d'abstraire le contenu d'un sous-arbre)

Langages Hors-Contextes

- Les langages HC Hors-Contextes (a.k.a. *CFL : Context-free Languages*) est une classe de langages plus large que les langages réguliers



*a.k.a. : also known as ;-)

Syntaxe et Grammaire

- La syntaxe est traditionnellement exprimée à l'aide d'une ***grammaire***

- Une **grammaire G** est une **collection de règles de réécriture** qui définissent mathématiquement quand une suite de symbole d'un certain alphabet constitue un mot d'un langage

- *L'ensemble des mots* pouvant être dérivées de ***G*** est appelé le *langage défini par* ***G***, noté ***L*(*G*)**.

Grammaire HC hors-contexte *(a.k.a. CFG Context-free Grammar)*

- Une grammaire hors-contexte (HC) ***G*** *est un 4-uplet* ***G = <T, NT, S, P>*** *où* :
  - ***T*** est l'ensemble des symboles *terminaux* (concrets reconnus lexicalement) ou lettres de l'alphabet
  - ***NT*** est l'ensemble des symboles non-*terminaux* (abstraits, structurant)
  - ***S*** $\in$ ***NT****, appelé symbole initial (Start ou axiome)*
    - Toute dérivation d'un mot de **L(G)** débute par **S**
    - À partir de ***S***, on dérive l'ensemble des mots de ***L(G)***
  - ***P*** est l'ensemble des règles de réécriture ou de production. Formellement, une règle de ***P*** est sous la forme : $NT \rightarrow (T \cup NT)*$

- Notons que la définition de **P** permet uniquement un seul non-terminal dans les parties gauches des règles. Cela assure que la grammaire soit *hors-contexte !!*
  - La grammaire ne tient pas compte du contexte dans lequel se trouve le symbole non-terminal à gauche
  - $? \; NT \; ? \rightarrow (T \cup NT)*$
  - ***La sémantique de l'opérateur de production → : « peut se substituer en » : S → a se lit : S peut se substituer (ou se réécrire) en 'a'***

Langage dérivé par une Grammaire

- Soit ***G = <T, NT, S, P>*** une grammaire. On appelle **langage engendré** par ***G*** l'ensemble ***L(G) = {w ∈ T* / S $\Rightarrow_{r \in P}^{+}$ w}***

  - *où $\Rightarrow_{r \in P}$, est appelée « **dérivation** » et*
    - *Sa sémantique : se lit « application d'une règle de production **r** de **P** »*
  - ***et** $\Rightarrow_{r \in P}^{+}$ dénote la répétition de règles $\Rightarrow_{r \in P}$*
  - *Le chaînage arrière de la dérivation s'appelle « **réduction** »*

Exemple et Simplification de notation

- G1 = <T={0,1}, NT={S}, S, P={r1,..,r5} >
  - **1ère écriture des règles**
    - r1: S → ε
    - r2: S → 0
    - r3: S → 1
    - r4: S → 0 S 0
    - r5: S → 1 S 1
  - **2ème écriture des règles**
    - r1: S → ε
    - r2: | 0
    - r3: | 1
    - r4: | 0 S 0
    - r5: | 1 S 1
  - **3ème écriture des règles (Forme BNF -Backus-Naur Form)**
    - r1: <S> **::=** (ε n'est pas représentable : c'à dire : un vide = ε)
    - r2: / 0
    - r3: | 1
    - r4: | 0 <S> 0
    - r5: | 1 <S> 1
  - **Forme EBNF –Exetnded-Backus-Naur Form**

Exercice

- Quel langage est décrit la grammaire suivante ?
- G1 = <T={0,1}, NT={S}, S, P={r1,..,r5} >
  - r1: S $\rightarrow \varepsilon$
  - r2: | 0
  - r3: | 1
  - r4: | 0 S 0
  - r5: | 1 S 1

Exercice

- Quel est le langage ***L(G2)*** décrit par la grammaire hors-contexte suivante ?

- ***G2 = < T ={a}, NT ={S}, S, P = {r1, r2} >***
  - **r1: S → aS**
  - **r2: | a**

Solution

- **G2:**
  - **r1:** $S \rightarrow aS$
  - **r2:** $\mid a$

- Raisonnons par induction sur la taille des mots w de L(G2)
- $|w| = 1 : S \Rightarrow_{r2} a$
- $|w| = 2 : S \Rightarrow_{r1} aS \Rightarrow_{r2} aa = a^2$
- $|w| = 3 : S \Rightarrow_{r1} aS \Rightarrow_{r1} aaS \Rightarrow_{r2} aaa = a^3$
- $|w| = 4 : S \Rightarrow_{r1} aS \Rightarrow_{r1} aaS \Rightarrow_{r1} aaaS \Rightarrow_{r2} aaaa = a^4$
- $|w| = i : S \Rightarrow_{r1} aS \Rightarrow_{r1} aaS \Rightarrow_{r1} aaS \Rightarrow_{r1} \ldots \Rightarrow_{r1} aaa..aaS \Rightarrow_{r2} \underbrace{aaa..aaa}_{\text{i fois}} = a^i$
- $L(G2) = \cup_{1 \leq i} \{w \in T^* \text{ tq } w = a^i\} = \{ w \in T^* \text{ tq } w = a+\}$
- ***L(G2)*** est le langage de mots formés d'une suite non vide de la lettre 'a'
- ***L(G2)*** *est régulier (c.f. démonstration)*

Exercice

- Soit ***G3 = < T ={a,b}, NT ={S}, S, P = {r1, r2} >*** une grammaire hors-contexte

- **G3:**
  - **r1:** **S → aSb**
  - **r2:** **| ε**

- ***Exercice :***
  - ***Quels sont les mots*** w ***de L(G3) ?***

Solution

- **G3:**
  - **r1:** $S \rightarrow aSb$
  - **r2:** $\mid \varepsilon$

- Raisonnons par induction sur la taille des mots w de L(G1)
- $|w| = 0*2 : S \Rightarrow_{r2} \varepsilon = a^0b^0$
- $|w| = 1*2 : S \Rightarrow_{r1} aSb \Rightarrow_{r2} a\,\varepsilon\,b = ab = a^1b^1$
- $|w| = 2*2 : S \Rightarrow_{r1} aSb \Rightarrow_{r1} aaSbb \Rightarrow_{r2} aa\,\varepsilon\,bb = aabb = a^2b^2$
- $|w| = 3*2 : S \Rightarrow_{r1} aSb \Rightarrow_{r1} aaSbb \Rightarrow_{r1} aaaSbbb \Rightarrow_{r2} aaa\,\varepsilon\,bbb = aaabbb = a^3b^3$
- $|w| = i*2 : S \Rightarrow_{r1} aSb \Rightarrow_{r1} aaSbb \Rightarrow_{r1} aaaSbbb \Rightarrow_{r1} \ldots \Rightarrow_{r1} aa\ldots aSbb\ldots b \Rightarrow_{r2} \underbrace{aa\ldots a}_{\text{i fois}}\,\varepsilon\,\underbrace{bb\ldots b}_{\text{i fois}} = aa\ldots abb\ldots b = a^ib^i$

- ***L(G3)*** $= \cup_{0 \le i} \{w \in T^* \text{ tq } w = a^ib^i\} = \{ w \in T^* \text{ tq } w = a^nb^n \text{ où } n \in IN\}$
- ***L(G3)*** *est* langage de mots formés d'une suite de la lettre 'a' suivie d'une suite de la lettre 'b' de la même taille que la précédente
- ***L(G3)*** *est irrégulier (c.f. démonstration)*
- **Application :** Et si l'on prend a='(' et b=')' ou si l'on prend a='begin' et b='end' : L(G3) = ?

Prouver qu'un Langage n'est pas hors contexte

- **Lemme de l'étoile étendu** *(extended pumping lemma):*
  - **Si** un langage ***L*** sur un alphabet Σ ($\boldsymbol{L} \subseteq \Sigma^*$) est **hors conexte**,
    - $\Rightarrow \exists\, \boldsymbol{n} \in \boldsymbol{IN^*}$ (qui dépendant de *L*) tel que
    - $\forall\, \boldsymbol{w} \in \boldsymbol{L}$, avec $|\boldsymbol{w}| \geq \boldsymbol{n}$, il existe une décomposition de ***w*** en cinq parties ***w = uvxyz*** *telle que*
      - $v \neq \varepsilon$ et $y \neq \varepsilon$
      - $\forall\, k \geq 0,\ uv^kxy^kz \in L$

Exercice

- Donner la grammaire HC (si possible) des langages suivants :
  - L0 = $\{a^n b^m c^m d^n\}$ n>=0, m >=0
  - L1 = $\{a^n b^m c^n d^m\}$
  - L2 = $\{a^n b^m c^l$ où n<m et n <l$\}$
  - L3 = $\{a^n b^n c^n$ où n≥ 1$\}$

Exercice

- Démontrons que le langage L1 = $\{a^n b^m c^n d^m\}$ n'est pas hors contexte
  - L11 = $\{a^n b^m c^n d^{m'/=m}\} = \{a^n b^* c^n d^*\} = \{a^n b^* c^n\} . \{d^*\}$
    - S ::= A D (le lien entre les b et les b est rompu)
    - A ::= a A c | B
    - B ::= bB | ε
    - D ::= dD | ε
  - L12 = $\{a^n b^m c^{n'/=n} d^m\} = \{a^* b^m c^* d^m\} = \{a^*\} \cdot \{b^m c^* d^m\}$
    - S ::= A B (le lien entre les a et les c est rompu)
    - A ::= a A | ε
    - B ::= b B d | C
    - C ::= cC | ε
  - L11 ∪ L12 = $\{a^n b^* c^n d^*\} \cup \{a^* b^m c^* d^m\}$
    - S ::= S1 | S2
    - S1 ::= A1 D1 (le lien entre les b et les b est rompu)
    - A1 ::= a A1 c | B1
    - B1 ::= bB1 | ε
    - D1 ::= dD1 | ε
    - S2 ::= A2 B2 (le lien entre les a et les c est rompu)
    - A2 ::= a A2 | ε
    - B2 ::= b B2 d | C2
    - C2 ::= cC2 | ε

Fermeture des langages hors contextes

- Les langages hors contexte sont fermés par les opérateurs union, concaténation et fermeture de Kleene.

- Si L_1 et L_2 sont des langages hors contexte alors
  - $L_1 \cup L_2$ est un langage hors contexte ($S \rightarrow L_1 \mid L_2$)
  - $L_1 . L_2$ est un langage hors contexte ($S \rightarrow L_1 . L_2$)
  - L_1^* est un langage hors contexte ($S \rightarrow L_1 S \mid \varepsilon$)
  - NB. $L_1 \cap L_2$ n'est pas un langage hors contexte

Grammaires HC linéaires

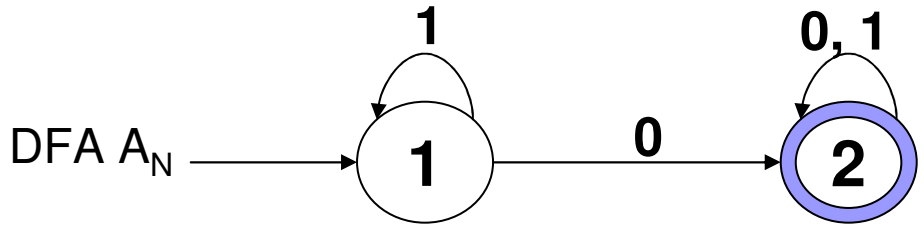
- Une grammaire ***G = <T, NT, S, P>*** *HC est dite* :
  - ***Linéaire Droite :*** si l'ensemble de ses règles de réécriture ***P*** *sont de la* forme : $NT \rightarrow (T \cup T.NT)$
    - La partie droite des règles de récriture contient un symbole terminal OU un symbole terminal suivi d'un symbole non-terminal e.g. **G : S → aS | a**
  - ***Linéaire Gauche :*** si l'ensemble de ses règles de réécriture ***P*** *sont de la* forme : $NT \rightarrow (T \cup NT.T)$
    - La partie droite des règles de récriture contient un symbole terminal OU un symbole non-terminal suivi d'un symbole terminal e.g. **G : S → Sa | a**

Langages réguliers et grammaires

- **Théorèmes** :
  1. Toute **grammaire HC Linéaire Droite G** génère un **langage régulier** (i.e. L(G) est reconnu par un automate d'état fini) e.g. **G : S → aS | a**

  2. Tout **langage régulier L** possède une **grammaire HC Linéaire Droite G** (i.e L(G) = L)

- **Exercice** : Donner une grammaire au langage reconnu par l'automate DFA suivant, puis écrivez ses équations linéaires, que remarquez-vous ?



Algorithme de passage d'un DFA à une CFG

- ***Input :* A = <S, Σ, δ, s_0, F >**
- ***Output : G = <T, NT, S, P> linéaire droite***

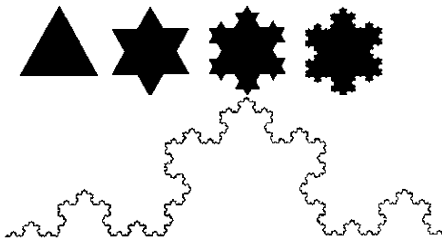
- On fait correspondre
  - À chaque élément s de S, un élément de NT (notons le NT(s))
  - À chaque élément l de Σ, un élément de T (notons le T(l))
  - À chaque élément (s,l,s') de δ, un élément de P (notons le P(s,l,s'))
    - $\delta : S \times \Sigma \rightarrow S$ devient $P : NT \times T \rightarrow NT$ (i.e. $NT \rightarrow T\ .\ NT$)
  - À s_0 le non terminal S
  - À chaque élément s de **F**, une règle $NT(s) \rightarrow \varepsilon$
    - Bien que le concept d'état final reste plus fort du fait qu'il reperésente la convergence de flux d'un graphe (= automate)

- Notons au passage que : $|P| = |\delta| + |F|$

Parenthèse philosophique : les grammaires dans la nature

1. **Sciences naturelles :**
   - Des formes géométriques de la nature sont de dimension fractales (la forme de la partie est une image réduite du tout) : e.g. les arbres, les végétaux, les algues, les flacons de neiges, les régions côtières, l'infiniment grand/petit, etc.
2. **Mathématiques :**
   - Les formes récurrentes des fractales géométriques admettent une syntaxe de langage récurrent
3. **Théorie de langage/Compilation :**
   - Toute forme syntaxique d'un langage est représentable par une Grammaire

- $1 \wedge 2 \wedge 3 \Rightarrow$ Une partie de la nature est définie par des grammaires
- A méditer : l'Univers serait-il entièrement fractal ? Serait-il régit par des règles de réécritures ?



Flocons de koch (de neige)

La Côte bretonne est fractale
De dimension d=4/3

un arbre

Le broccoli

Types de dérivations : gauche/droite

- Soit ***r*** des règles hors-contextes de ***P*** sous la forme $\boldsymbol{NT} \rightarrow (\boldsymbol{T} \cup \boldsymbol{NT})*$
  1. Pour des grammaires avec un seul non terminal dans les parties droites de **r** (e.g. grammaires HC linéaires droites $\boldsymbol{NT} \rightarrow (\boldsymbol{T} \cup \boldsymbol{T.NT})$ ou gauches $\boldsymbol{NT} \rightarrow (\boldsymbol{T} \cup \boldsymbol{NT.T})$)
     - Dériver un mot revient à dériver le seul non-terminal de **r**
  2. Pour les autres grammaires, un problème se pose : Quel non-terminal droit de **r** dériver en premier ?
     - 2 méthodes :
       - Dérivation gauche de $\mathbf{r} \Rightarrow_r$ est celle qui développe toujours le non terminal le plus à gauche de $(\boldsymbol{T} \cup \boldsymbol{NT})*$
       - Dérivation droite de $\mathbf{r} \Rightarrow_r$ est celle qui développe toujours le non terminal le plus à droite de $(\boldsymbol{T} \cup \boldsymbol{NT})*$

Dérivations gauches - exemples

- Soit **G3 *= < T={Number,+,-, ×, ÷ }, NT ={Expr, Op}, S=Expr, P = {r1,..,r6} >*** une grammaire hors-contexte :
  - r1: ***Expr*** → *Expr Op Number*
  - r2: | *Number*
  - r3: ***Op*** → +
  - r4: | –
  - r5: | ×
  - r6: | ÷

- Exemples de *dérivations gauches* de mots dans ***L(G3)***
  - ***Expr*** $\Rightarrow_{r1}$ ***Expr*** *Op Number* $\Rightarrow_{r2}$ *Number* ***Op*** *Number* $\Rightarrow_{r3}$ *Number + Number*
  - ***Expr*** $\Rightarrow_{r1}$ ***Expr*** *Op Number* $\Rightarrow_{r1}$ ***Expr*** *Op Number Op Number* $\Rightarrow_{r2}$ *Number* ***Op*** *Number Op Number* $\Rightarrow_{r3}$ *Number + Number* ***Op*** *Number* $\Rightarrow_{r5}$ *Number + Number × Number*

- Exercice :
  - Trouver une dérivation droite de l'expression : "**Number + Number × Number**"

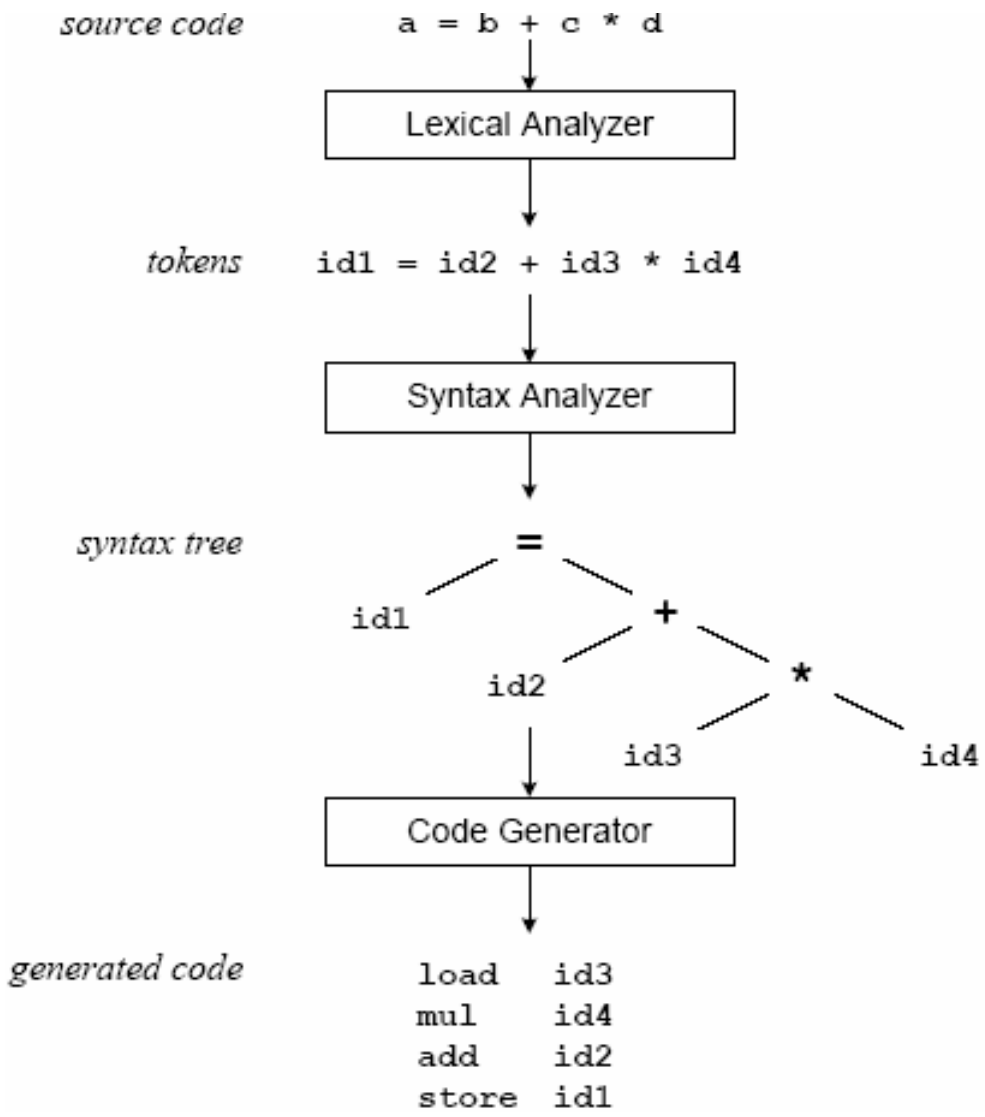
Dérivations gauches vs droites

1. Dérivation droite
   - ***Expr*** $\Rightarrow_{r1}$ *Expr* ***Op*** Number $\Rightarrow_{r5}$ ***Expr*** × Number $\Rightarrow_{r1}$ *Expr* ***Op*** Number × Number $\Rightarrow_{r3}$ ***Expr*** + Number × Number $\Rightarrow_{r2}$ Number + Number × Number
   - règles appliquées dans l'ordre = (r1,r5,r1,r3,r2)

2. Dérivation gauche
   - ***Expr*** $\Rightarrow_{r1}$ ***Expr*** *Op* Number $\Rightarrow_{r1}$ ***Expr*** *Op* Number *Op* Number $\Rightarrow_{r2}$ Number ***Op*** Number *Op* Number $\Rightarrow_{r3}$ Number + Number ***Op*** Number $\Rightarrow_{r5}$ Number + Number × Number
   - règles appliquées dans l'ordre = (r1,r1,r2,r3,r5)

- Remarque :
  - Pour dériver l'expression par les dérivations gauches et droites, on a appliqué les **mêmes règles** {r1,r2,r3,r5} mais **pas dans le même ordre**

Repositionnement de l'analyseur syntaxique

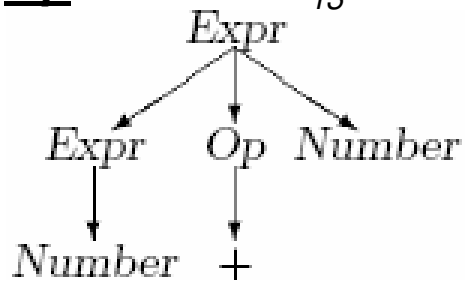


Analyseur syntaxique - fonctionnement

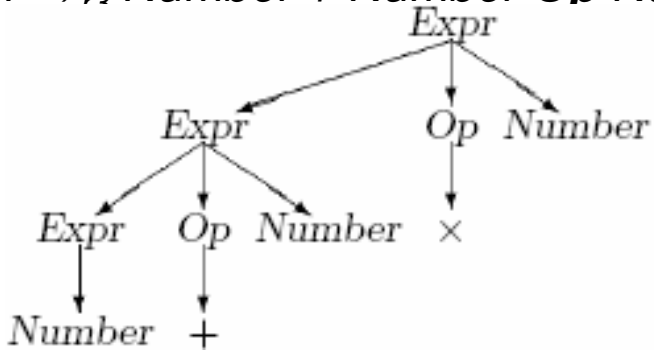
- L'analyseur syntaxique doit découvrir automatiquement pour une expression donnée en langage ***L***, son arbre syntaxique de dérivation par rapport à la grammaire de ce langage ***L***
  1. La **racine de l'arbre** syntaxique est : le symbole non-terminal initial **S**
  2. Les **noeuds de l'arbre** syntaxique sont : le résultat de l'application des règles de production $\in (T \cup NT)*$
  3. Les **feuilles de l'arbre** syntaxique sont : les éléments de l'expression donnée en entrée $\in T*$

Arbres syntaxiques (ou Arbres de dérivation)

- ***Expr*** $\Rightarrow_{r1}$ ***Expr*** *Op Number* $\Rightarrow_{r2}$ *Number* ***Op*** *Number* $\Rightarrow_{r3}$ *Number + Number*



- ***Expr*** $\Rightarrow_{r1}$ ***Expr*** *Op Number* $\Rightarrow_{r1}$ ***Expr*** *Op Number Op Number* $\Rightarrow_{r2}$ *Number* ***Op*** *Number Op Number* $\Rightarrow_{r3}$ *Number + Number* ***Op*** *Number* $\Rightarrow_{r5}$ *Number + Number × Number*



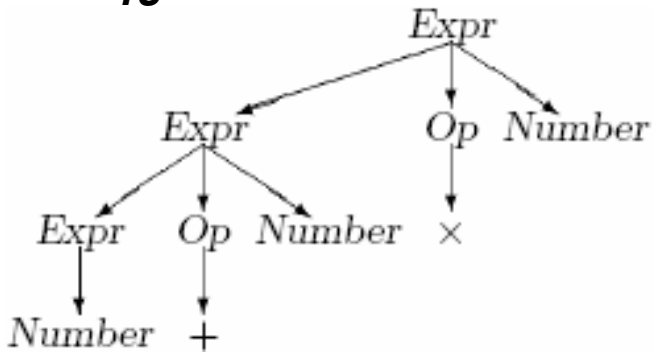
- ***L(G3)*** regroupe toute expression arithmétique {×, ÷, −, +} sur les nombres
- **Exercice :** Donner les arbres des dérivations G & D de “Number + Number × Number”

Arbre syntaxique de dérivation gauche/droite *(leftmost/rightmost derivation)*

1. Dérivation droite
   - ***Expr*** $\Rightarrow_{r1}$ *Expr* ***Op*** Number $\Rightarrow_{r5}$ ***Expr*** × Number $\Rightarrow_{r1}$ *Expr* ***Op*** Number × Number $\Rightarrow_{r3}$ ***Expr*** + Number × Number $\Rightarrow_{r2}$ Number + Number × Number

2. Dérivation gauche
   - ***Expr*** $\Rightarrow_{r1}$ ***Expr*** *Op* Number $\Rightarrow_{r1}$ ***Expr*** *Op* Number *Op* Number $\Rightarrow_{r2}$ Number ***Op*** Number *Op* Number $\Rightarrow_{r3}$ Number + Number ***Op*** Number $\Rightarrow_{r5}$ Number + Number × Number

- Remarque :
  - Les dérivations gauches et droites, aboutissent au **même arbre syntaxique**

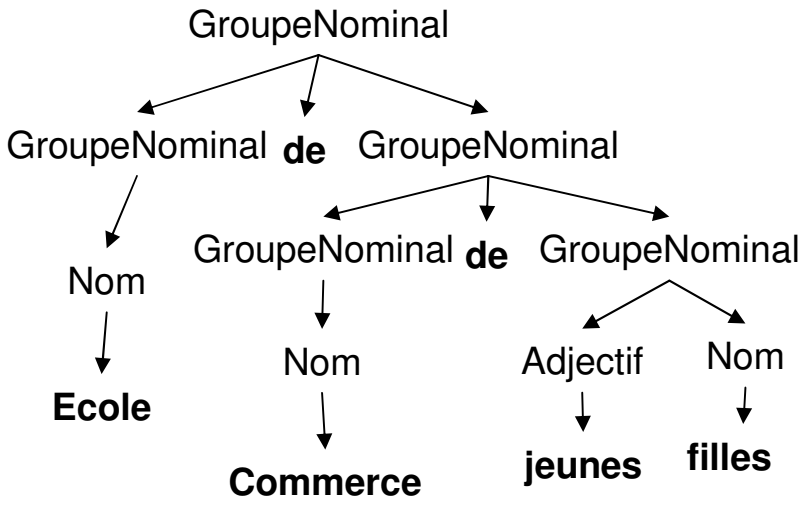


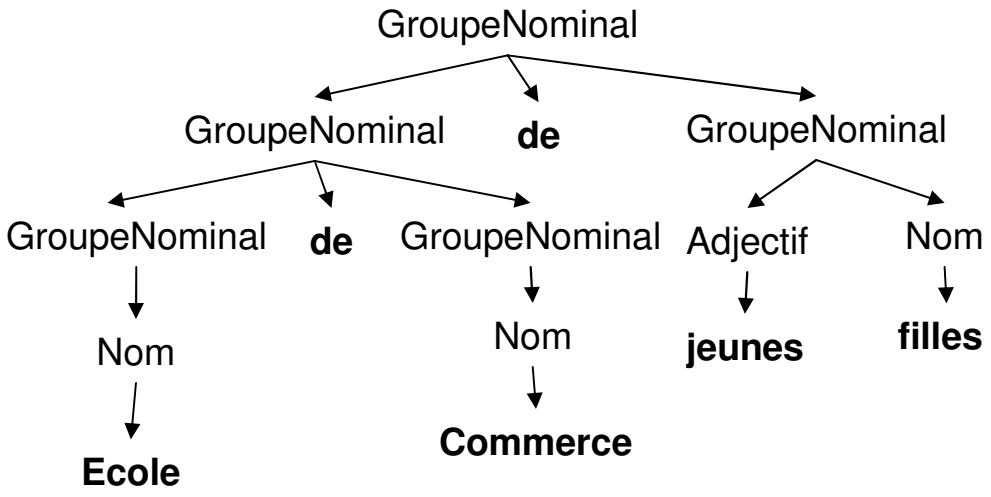
Grammaires ambigües

Exercice

- Soit les règles de la grammaire suivante :
  - GrammaireSimpliste :
    - r1: **GroupeNominal** → Nom
    - r2 : | **Adjectif** Nom
    - r3 : | **GroupeNominal** de **GroupeNominal**
  - Trouver un arbre de dérivation du mot w = « **Ecole de Commerce de jeunes filles** »

Grammaire Ambiguë





- 2 interprétations différentes :
  1. School of Young Girls Business
  2. Business School of Young Girls
- GrammaireSimpliste est donc ambiguë !!

Exercice

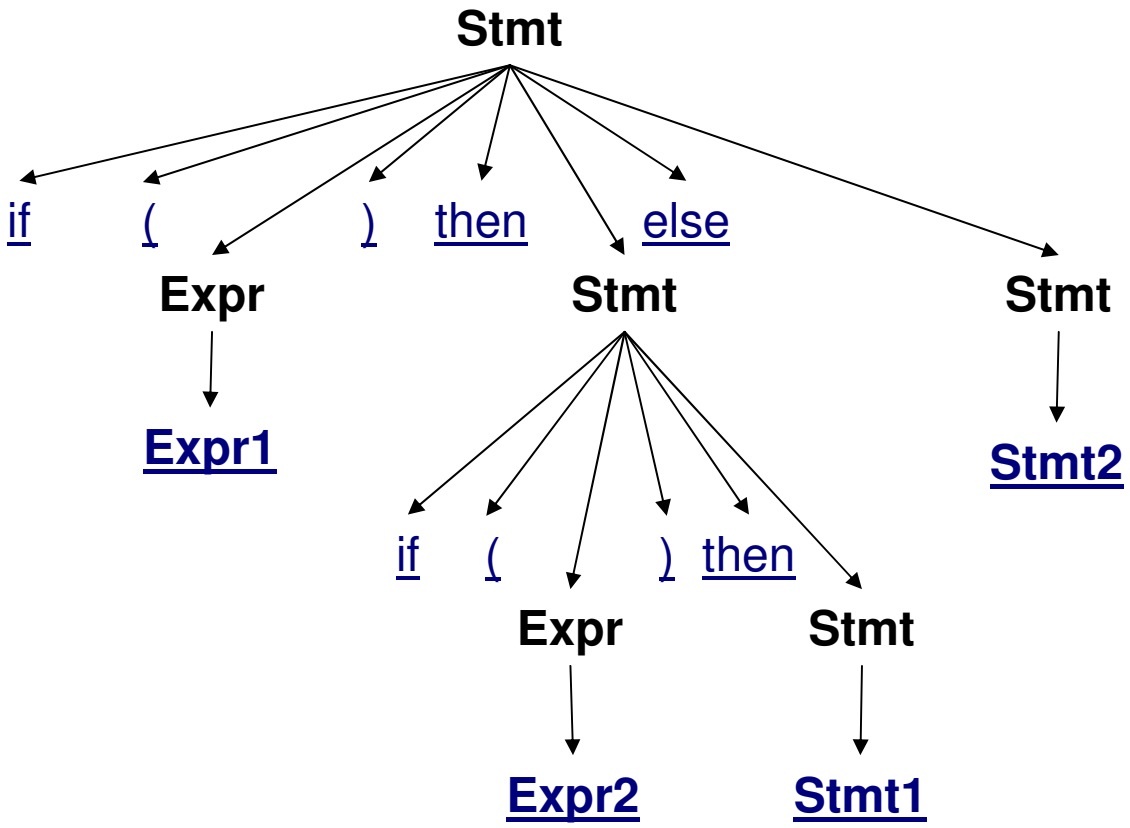
- *G4:*
  - r1: **Stmt** → if (**Expr**) then **Stmt** else **Stmt**
  - r2: | if (**Expr**) then **Stmt**
  - …
  - rn | . . .

  - Exercice :
    - Trouver un arbre de dérivation du mot

    w = if (*Expr*1) then if (*Expr2*) then *Stmt*1 else *Stmt2*

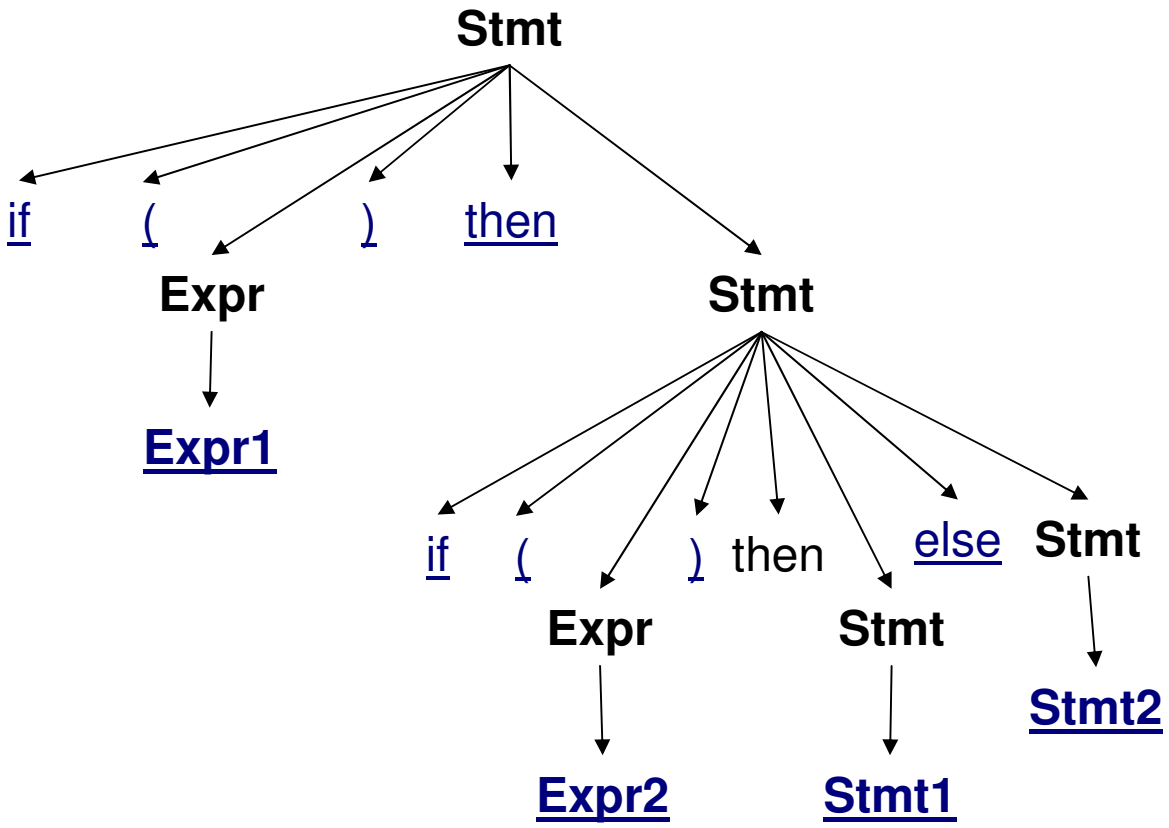
Solution 1 (r1,r2)

- Stmt $\Rightarrow^{r1}$
- if (**Expr**) then
  - **Stmt**
- else **Stmt**
- $\Rightarrow^{r2}$
- if (**Expr**) then
  - if (**Expr**) then
    - **Stmt**
- else **Stmt**
- **…**
- $\Rightarrow$
- if (*Expr*1) then
  - if (*Expr2*) then
    - *Stmt*1
- else
  - *Stmt2*



Solution 2 (r2,r1)

- Stmt $\Rightarrow^{r2}$
- if (**Expr**) then
  - **Stmt**
- $\Rightarrow^{r1}$
- if (**Expr**) then
  - if (**Expr**) then
    - **Stmt**
  - else **Stmt**
- **…**
- $\Rightarrow$
- if (*Expr*1) then
  - if (*Expr*2) then
    - *Stmt*1
  - else
    - *Stmt*2



Constat

- *A priori* (r1,r2) doit produire le même arbre syntaxique de (r2,r1)
- C'est le contraire qui se produit
  - Donc, la grammaire est anormale !

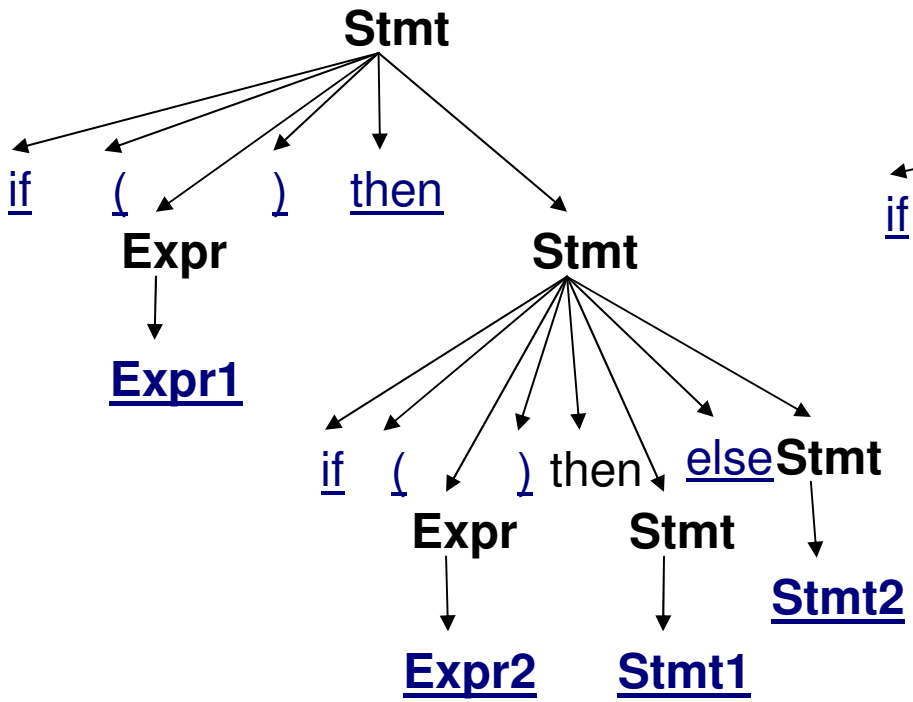
Grammaire Ambiguë

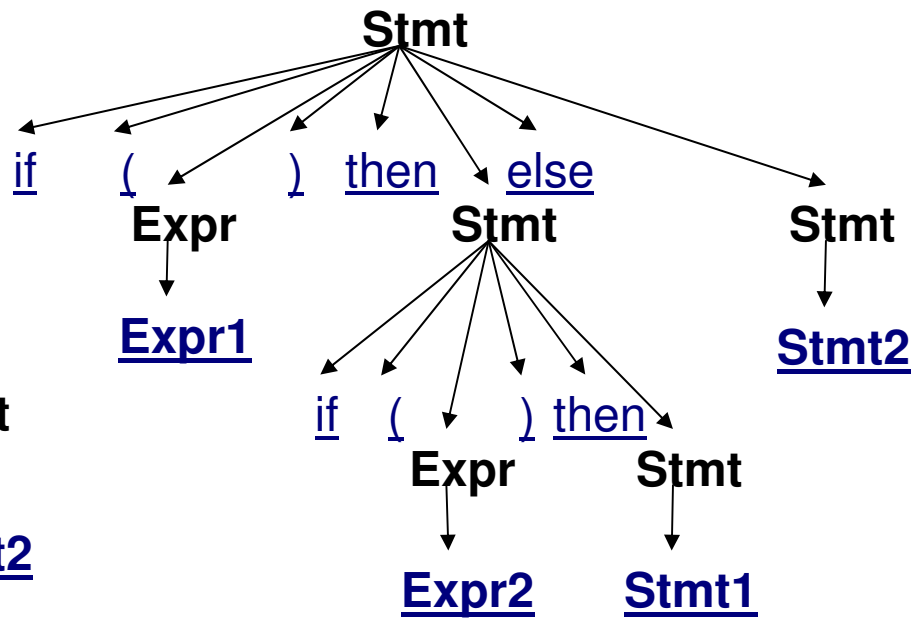
- Une grammaire ***G*** *est* dite *ambiguë ssi* Il existe un mot **w** ∈ ***L*(*G*)** qui possède plusieurs (par ex. 2) arbres de dérivations.

- G4 est une grammaire ambiguë du fait qu'on a trouvé deux dérivations du mot w = « if (*Expr*1) then if (*Expr*2) then *Stmt*1 else *Stmt*2 »

Exemple d'ambiguïté

- w = « **if (*Expr*1) then if (*Expr*2) then *Stmt*1 else *Stmt*2** »





- **2** arbres syntaxiques différents d'une expression (i.e. **ambigüité de la grammaire**) ⇒ **2** interprétations différentes ⇒ **2** sémantiques différentes ⇒ **2** codes possibles à générer pour cette expression ⇒ **Problème de non-déterminisme pour le compilateur !!**

Grammaire Ambiguë

- L'ambiguïté implique que l'analyseur syntaxique ne pourra pas découvrir d'une manière unique et définitive l'arbre syntaxique de cette expression.

- Si l'analyseur syntaxique ne peut pas décider la structure syntaxique d'une expression, décider du sens (i.e. la sémantique) et donc du code exécutable équivalent à cette expression ne sera pas possible !!

- L'Ambiguïté est donc une propriété indésirable dans une grammaire.

Élimination de l'ambiguïté d'une grammaire

1. Dans l'absolu, il n'existe pas d'algorithme qui répond systématiquement si une grammaire est ambiguë *Ooops* :-(
   - Prouver qu'une grammaire G est non ambiguë revient à étudier l'unicité des arbres de dérivations de tous les mots de L(G) qui peut être infini
   - *Prouver l'ambiguïté est un problème semi-décidable* !

Élimination de l'ambiguïté d'une grammaire

2. Et si on peut décider qu'une grammaire est ambiguë, on ne pourra pas toujours la désambiguïser. Car toutes les grammaires ambiguës ne peuvent pas être désambiguïsées. *Ooops* :-(
   - Il existe des langages héréditairement ambigus (*inherently ambiguous*) (c.f. Hopcroft'2001 pour la preuve)
     - e.g. $L = \{a^n b^n c^m d^m, n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n, n \geq 1, m \geq 1\}$

Exercice

- *Donner une grammaire reconnaissant le langage L = $\{a^n b^n c^m d^m, n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n, n \geq 1, m \geq 1\}$*

Solution

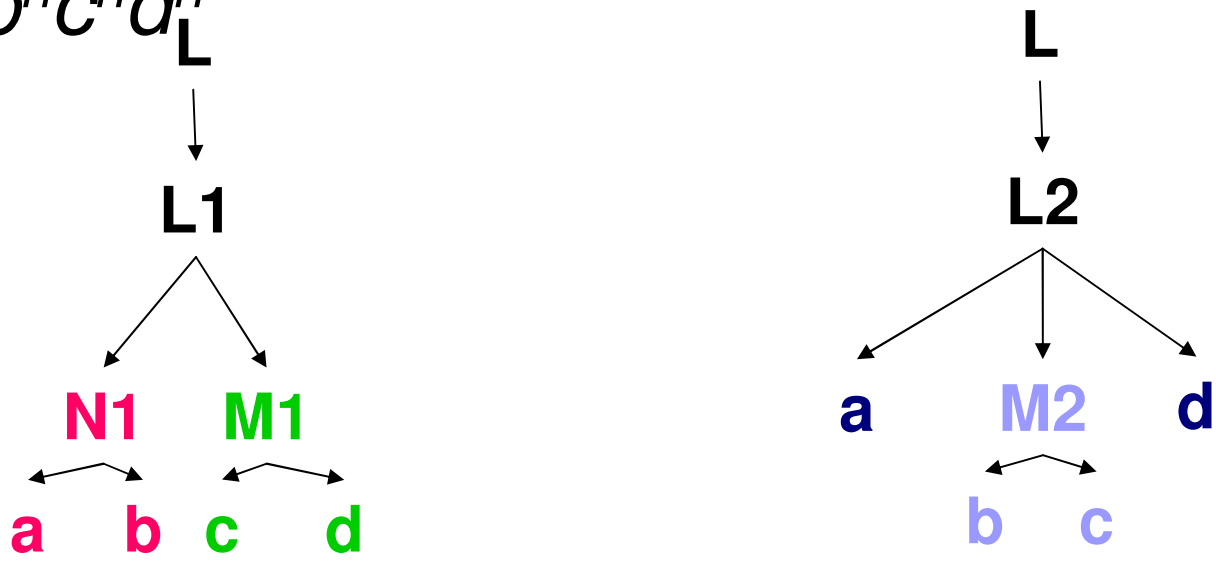
- $L=\{a^n b^n c^m d^m, n\geq 1,\ m\geq 1\} \cup \{a^n b^m c^m d^n, n\geq 1,\ m\geq 1\}$
  - **G:**
    - $R_1 : L \rightarrow L_1$
    - $R_2 : L \rightarrow L_2$
    - $R_3 : L_1 \rightarrow N_1\ M_1$
    - $R_4 : N_1 \rightarrow a\ N_1\ b$
    - $R_5 : N_1 \rightarrow a\ b$
    - $R_6 : M_1 \rightarrow c\ M_1\ d$
    - $R_7 : M_1 \rightarrow c\ d$
    - $R_8 : L_2 \rightarrow a\ L_2\ d$
    - $R_9 : L_2 \rightarrow a\ M_2\ d$
    - $R_{10} : M_2 \rightarrow b\ M_2\ c$
    - $R_{11} : M_2 \rightarrow b\ c$

Exercice

- *Démontrer que la grammaire G est ambiguë*
  - ***G:***
    - $R_1 : L \rightarrow L_1$
    - $R_2 : L \rightarrow L_2$
    - $R_3 : L_1 \rightarrow N_1 M_1$
    - $R_4 : N_1 \rightarrow a N_1 b$
    - $R_5 : N_1 \rightarrow a b$
    - $R_6 : M_1 \rightarrow c M_1 d$
    - $R_7 : M_1 \rightarrow c d$
    - $R_8 : L_2 \rightarrow a L_2 d$
    - $R_9 : L_2 \rightarrow a M_2 d$
    - $R_{10} : M_2 \rightarrow b M_2 c$
    - $R_{11} : M_2 \rightarrow b c$

Solution

- *La grammaire G est ambiguë car le mot « **abcd** » possède deux arbres de dérivation. D'ailleurs c'est vrai, pour tout mot appartenant à L1 et L2, soit n>= 1, $a^n b^n c^n d^n$*



Solution

- *Donner une idée sur le fait que la grammaire G ne peut pas être désambiguïsée? (Par ailleurs L1 ∩ L2 n'admet de CFG)*
  - ***G:***
    - $R_1 : L \rightarrow L_1$
    - $R_2 : L \rightarrow L_2$
    - $R_3 : L_1 \rightarrow N_1 M_1$
    - $R_4 : N_1 \rightarrow a N_1 b$
    - $R_5 : N_1 \rightarrow a b$
    - $R_6 : M_1 \rightarrow c M_1 d$
    - $R_7 : M_1 \rightarrow c d$
    - $R_8 : L_2 \rightarrow a L_2 d$
    - $R_9 : L_2 \rightarrow a M_2 d$
    - $R_{10} : M_2 \rightarrow b M_2 c$
    - $R_{11} : M_2 \rightarrow b c$

Élimination de l'ambiguïté d'une grammaire

3. Heureusement, en pratique sur les grammaires liées aux langages de programmation, il existe des techniques connues d'élimination d'ambiguïté, *Ahhhhh* :)

Exemple de grammaire ambiguë et de son équivalente non-ambiguïté

- *G4:*
  - r1: **Stmt** → if (**Expr**) then **Stmt** else **Stmt**
  - r2: | if (**Expr**) then **Stmt**
  - …
  - rn | . . .
- Cause de l'ambiguïté de G4 : l'on ne sait jamais à quel if, le dernier else est attaché ?
- Solution non-ambigüe :

$$
\begin{array}{lcl}
\textit{Stmt} & \rightarrow & \textit{WithElse} \\
 & | & \textit{LastElse} \\
\textit{WithElse} & \rightarrow & \texttt{if (}\ \textit{Expr}\ \texttt{) then}\ \textit{WithElse}\ \texttt{else}\ \textit{WithElse} \\
 & | & \textit{Assignment} \\
 & | & \textit{other statements} \ldots \\
\textit{LastElse} & \rightarrow & \texttt{if (}\ \textit{Expr}\ \texttt{) then}\ \textit{Stmt} \\
 & | & \texttt{if (}\ \textit{Expr}\ \texttt{) then}\ \textit{WithElse}\ \texttt{else}\ \textit{LastElse}
\end{array}
$$

Solutions

- STMT ::=
  - if (Expr) then STMT else STMT endif
  - if (Expr) then STMT endif

- if (*Expr*1) then
  - if (*Expr*2) then *Stmt*1
  - else *Stmt*2
  - endif
- endif

- if (*Expr*1) then
  - if (*Expr*2) then
    - *Stmt*1
  - endif
- else
  - *Stmtt2*
- endif

Convention (1) le else répond au premier if

- Ou
- Start = WE
- WE ::=
  - if (Expr) then WE else WOE
  - WOE
- WOE ::=
  - if (Expr) then WOE
  - OS …

Convention (2) le else répond au dernier if

- Ou
- Start = WOE
- WE ::=
  - if (Expr) then OS else WE
  - OS …
- WOE ::=
  - if (Expr) then WOE
  - WE

Exercice

a) La grammaire suivante est-elle ambiguë ?

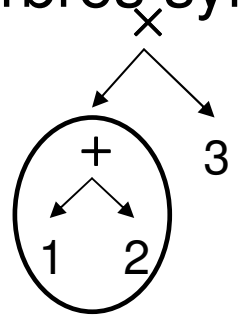
- G_{arith}:
  - r1 : $E \rightarrow E \times E$
  - r2 : $E \rightarrow E + E$
  - r3 : $E \rightarrow$ Nombre

b) Si Oui Pourquoi ?

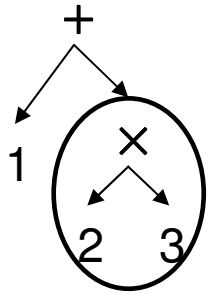
Exemple d'ambiguïté et élimination de cette ambiguïté

- Le mot "$1 + 2 \times 3$" possède deux dérivations droites
  1. $E \Rightarrow_{r1} E \times E \Rightarrow_{r2} E + E \times E \Rightarrow^*_{r3} 1 + 2 \times 3$
  2. $E \Rightarrow_{r2} E + E \Rightarrow_{r1} E + E \times E \Rightarrow^*_{r3} 1 + 2 \times 3$
- Donc 2 arbres syntaxiques

Arbre 1



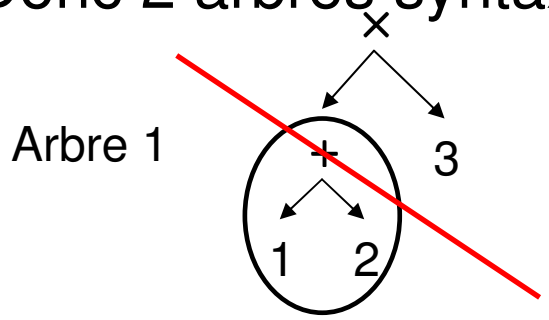
Arbre 2

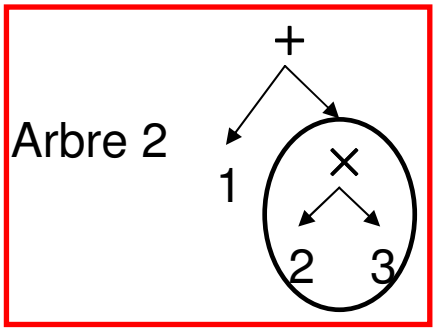


- Donc 2 interprétations possibles
  1. **$(1 + 2) \times 3 \equiv 9$**
  2. **$1 + (2 \times 3) \equiv 7$**

Exemple d'ambiguïté et élimination de cette ambiguïté

- Le mot "$1 + 2 \times 3$" possède deux dérivations droites
  1. ~~$E \Rightarrow_{r1} E \times E \Rightarrow_{r2} E + E \times E \Rightarrow^*_{r3} 1 + 2 \times 3$~~
  2. $E \Rightarrow_{r2} E + E \Rightarrow_{r1} E + E \times E \Rightarrow^*_{r3} 1 + 2 \times 3$
- Donc 2 arbres syntaxiques





- Donc 2 interprétations possibles
  1. ~~**$(1 + 2) \times 3 \equiv 9$**~~
  2. **$1 + (2 \times 3) \equiv 7$**
- C'est la 2ème interprétation qui correspond aux règles de priorité usuelles des opérateurs arithmétiques !!

Causes de l'ambiguïté et idées de solutions

1. Ambiguïté pour les expressions de G_{arith} sur l'opérateur $\times$
   - Cause : G_{arith} accepte aussi bien l'associativité gauche que droite de $\times$
   - Solution : On choisit d'accepter *l'associativité à gauche de* $\times$ (e.g. l'interprétation $(1 \times 2) \times 3$ mais pas $1 \times (2 \times 3)$.)

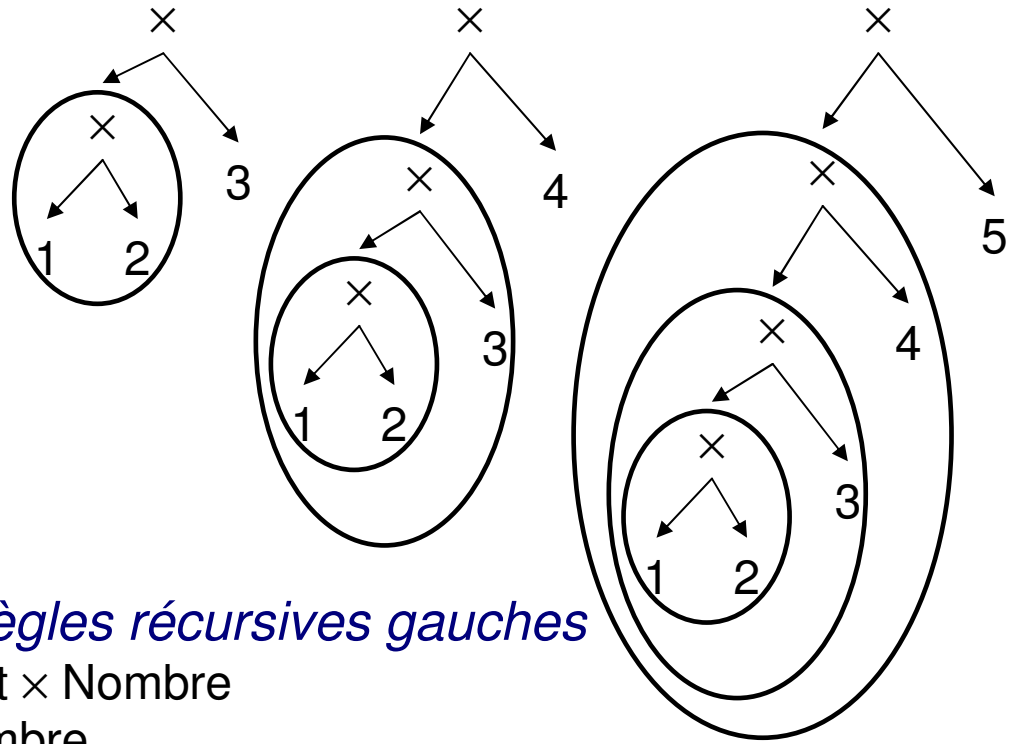
2. Ambiguïté pour les expressions de G_{arith} sur l'opérateur $+$
   - Cause : G_{arith} accepte aussi bien l'associativité gauche que droite de $+$
   - Solution : On choisit d'accepter *l'associativité à gauche de* $+$ (e.g. l'interprétation $(1 + 2) + 3$ mais pas $1 + (2 + 3)$.)

3. Ambiguïté pour les expressions de G_{arith} sur les 2 opérateurs $\times$ *et* $+$
   - Cause : G_{arith} ne différencie pas entre les priorités de $\times$ et $+$
   - Solution : On choisit *priorité*$(\times) >$ *priorité*$(+)$ (e.g. l'interprétation $1 + (2 \times 3)$ mais pas $(1 + 2) \times 3$.)

- NB. Rien ne nous empêche d'adopter l'associativité droite !!
- Suite de l'exercice :
  - c) Essayer de réfléchir sur les transformations de G_{arith} pour supporter ces contraintes nécessaires à sa désambiguïsation

Exemple d'ambiguïté et élimination de cette ambiguïté

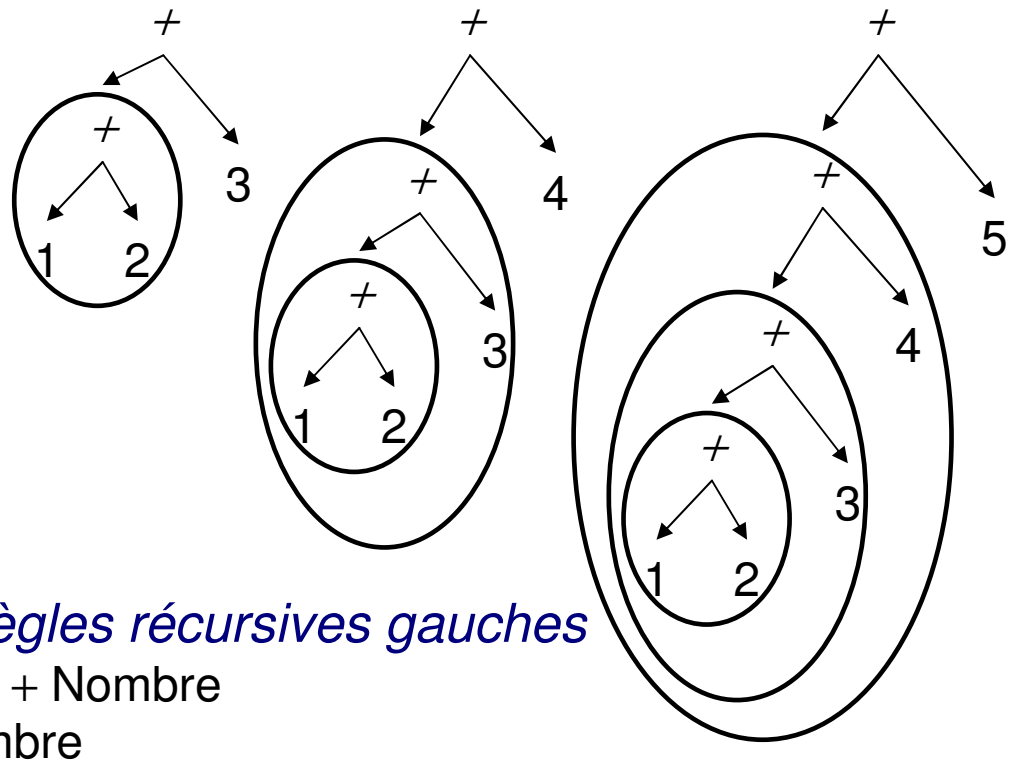
- Accepter *l'associativité à gauche de* $\times$



- $\Rightarrow$ Introduire des *règles récursives gauches*
  - R_{11} : Mult $\rightarrow$ Mult $\times$ Nombre
  - R_{12} : Mult $\rightarrow$ Nombre

Exemple d'ambiguïté et élimination de cette ambiguïté

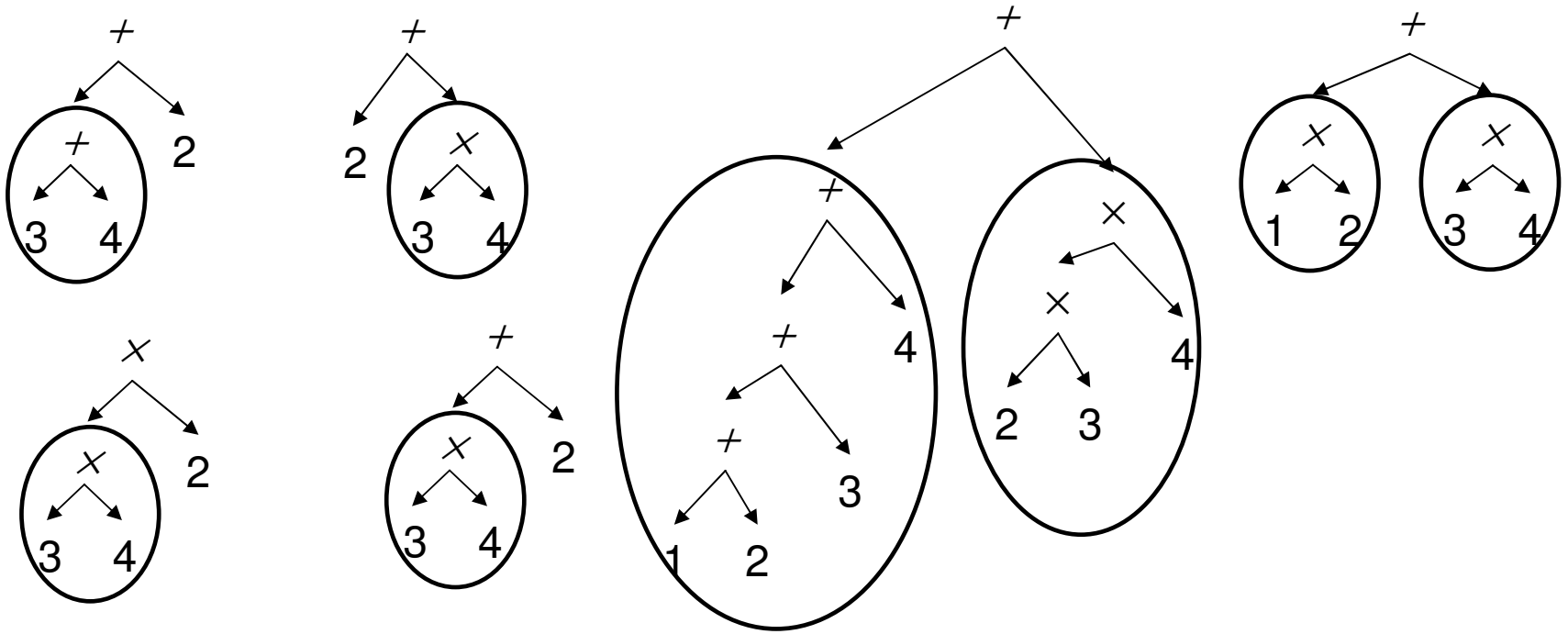
- Accepter *l'associativité à gauche de +*



- ⇒ Introduire des *règles récursives gauches*
  - R_{21} : Add → Add + Nombre
  - R_{22} : Add → Nombre

Exemple d'ambiguïté et élimination de cette ambiguïté

- Choisir *priorité(×) > priorité(+)*



- ⇒ Introduire des *règles récursives gauches*
  - R_{31} : Add → Add + Mult (*plus générale que* R_{21}, *car* R_{12} : Mult → Nombre)
  - R_{32} : Add → Mult (*plus générale que* R_{22}, *car* R_{12} : Mult → Nombre)

Exemple d'ambiguïté et élimination de cette ambiguïté : Synthèse 1

- G'_{arith} < T={Nombre,+, ×}, NT ={Mult, Add}, S=Add, P = {R_{11}, R_{12}, R_{31}, R_{32}} >
  - R_{11} : Mult → Mult × Nombre
  - R_{12} : Mult → Nombre
  - ~~R_{21} : Add → Add ≠ Nombre~~ (car *R_{31} est plus générale que R_{21}*)
  - ~~R_{22} : Add → Nombre~~ (car *R_{32} est plus générale que R_{22}*)
  - R_{31} : Add → Add + Mult
  - R_{32} : Add → Mult

  - On a pris *S=Add* car *Add* est plus général de *Mult* (c.f. R_{32}, de *Add* on peut aller vers *Mult* mais l'inverse est faux)

  - G'_{arith} n'est pas ambiguë par construction !!

Contrainte complémentaire
(priorité forcée et associativité droite forcée)

- On souhaite que G'_{arith} puisse supporter
  1. $1 + 2 \times 3$ à être interprété $(1 + 2) \times 3$
  2. et $3 \times 1 + 2$ à être interprété $3 \times (1 + 2)$
  3. et $1+ 2 \times 3 + 4$ à être interprété $(1 + 2) \times (3 + 4)$
  4. Et Nouveau langage avec nouveaux terminaux :
     - $T := T \cup \{‘(‘, ‘)’\}$

- Suite de l’exercice :
  - d) Penser à une solution à ces nouveaux problèmes

Exemple d'ambiguïté et élimination de cette ambiguïté : Problème résolu

- Solution possible : introduire les parenthèses
  - NT devient {Mult, Add, Aux}
  - P devient {R'_{11}, R'_{12}, R'_{13}, R'_{14}, R_{31}, R_{32}}
    - R'_{11} : Mult $\rightarrow$ Mult $\times$ Aux *(plus générale que R_{11}, car* R'_{13} : Aux$\rightarrow$ Nombre*)*
    - R'_{12} : Mult $\rightarrow$ Aux *(plus générale que R_{12}, car* R'_{13} : Aux$\rightarrow$ Nombre*)*
    - R'_{13} : Aux $\rightarrow$ Nombre
    - R'_{14} : Aux $\rightarrow$ (Add)

    - Aulieu de
    - ~~R_{11} : Mult $\rightarrow$ Mult $\times$ Nombre~~
    - ~~R_{12} : Mult $\rightarrow$ Nombre~~

Features de notre nouvelle grammaire

1. Le (Add) peut être fils à gauche et à droite du Add
   - (1+2) + (3+4)
2. Le Add peut être asociatif droit (forcé avec parenthèse)
   - (1 + (2 + (3 + 4)))
3. Le (Mult) peut être fils à gauche et à droite du Mult
   - (1*2) * (3*4)
4. Le Mult peut être asocaitif droit (forcé avec parenthèse)
   - (1 * (2 * (3 * 4)))

Vérification du résultat trouvé

- 4 cas avec deux opérations dans une expression parenthésées
  - + à gauche du × : x + y + z
    - Cas particuliers : Nombre + Add, Add + Nombre
  - × à gauche du + : x + y × z
    - Exemples : (1 + 2) × 3
    - Cas particuliers : Nombre + Mult, Add × Nombre
  - + à gauche du + : x × y + z
    - Exemples : 3 × (1 + 2)
    - Cas particuliers : Nombre × Add, Mult + Nombre
  - × à gauche du × : x × y × z
    - Cas particuliers : Mult × Nombre, Nombre × Mult
- Seuls l'associativé gauche sera bien sûr prise en compte
- Si les fils gauche et droits sont des multiplications associatives gauches, leur combinaison avec un × ou + en racine est – elle acceptable est accptée (c.a.d, donne-t-elle un arrbre associatif gauche ?

Exemple d'ambiguïté et élimination de cette ambiguïté : Synthèse 2

- G'_{arith} < T={Nombre,(,), +, ×}, NT={Mult, Add, Aux}, S=Add, P = {R_1, R_2, R_3, R_4, R_5, R_6}>
  - R1 (R'_{11}) : Mult → Mult × Aux
  - R2 (R'_{12}) : Mult → Aux
  - R3 (R'_{13}) : Aux → Nombre
  - R4 (R'_{14}) : Aux → (Add)
  - R5 (R_{31}) : Add → Add + Mult
  - R6 (R_{32}) : Add → Mult

    - 1 + 2 × 3 pourra être interprété sans ambiguïté (1 + 2) × 3 :
      - Add $\Rightarrow_{R6}$ Mult $\Rightarrow_{R1}$ Mult × Aux $\Rightarrow_{R2}$ Aux × Aux $\Rightarrow_{R4}$ (Add) × Aux $\Rightarrow_{R5}$ (Add + Mult) × Aux $\Rightarrow_{R6}$ (Mult + Mult) × Aux $\Rightarrow_{R2}$ (Aux + Mult) × Aux $\Rightarrow_{R2}$ (Nombre + Mult) × Aux $\Rightarrow^*$ (Nombre + Nombre) × Nombre

    - 3 × 1 + 2 peut être interprété sans ambiguïté 3 × (1 + 2)
      - Add $\Rightarrow_{R6}$ Mult $\Rightarrow_{R1}$ Mult × Aux $\Rightarrow_{R2}$ Aux × Aux $\Rightarrow_{R3}$ Nombre × Aux $\Rightarrow_{R4}$ Nombre × (Add) $\Rightarrow^*$ Nombre × (Nombre + Nombre)

  - G'_{arith} reste non ambiguë par construction (vous pourrez toujours -mais en vain- essayer de trouver des contre-exemples) !!

Exercice

- Et si l'on ajoutait l'opérateur '-' au langage cible
- Avec priorité ('-') = priorité ('+')
- Quelle serait la nouvelle grammaire

Solution

- priorité (‘-’) = priorité (‘+’)
  - Add = Min
  - D’un point de vue syntaxique, on ne s’intéresse pas à la différence sémantique entre + et –, mais uniquement à leur équivalence au vu de l’ambiguïté.
  - Add = Min = AddMin (même classe d’équivalence en termes de priorités et d’associativité et de réaction avec le *)

Et si l'on ajoutait l'opérateur '-'

- La priorité du moins = la priorité du plus
- Add = même classe d'équivalence du Minus (AddMin)

- $G'_{arith} < T=\{Nombre,(,), +, \times, -\}, NT=\{Mult, AddMin, Aux\}, S=AddMin, P = \{R_1, R_2, R_3, R_4, R_5, R_6, R_7\}>$
  - R1 (R'_{11}) : Mult $\rightarrow$ Mult $\times$ Aux
  - R2 (R'_{12}) : Mult $\rightarrow$ Aux
  - R3 (R'_{13}) : Aux $\rightarrow$ Nombre
  - R4 (R'_{14}) : Aux $\rightarrow$ (AddMin)
  - R5 (R_{31}) : AddMin $\rightarrow$ AddMin + Mult
  - R6 ($R_{31'}$) : AddMin $\rightarrow$ AddMin - Mult
  - R7 (R_{32}) : AddMin $\rightarrow$ Mult

Analyseur syntaxique - variantes

- Il existe 2 types d'analyseurs syntaxiques (parseurs)
  - Analyseur Descendant (top-down)
    - Se programme d'une manière très systématique
    - Algorithme :
      - Commence par la racine et procède en descendant l'arbre syntaxique jusqu'aux feuilles.
      - A chaque étape, l'analyseur choisit un nœud parmi les symboles non-terminaux et développe l'arbre à partir de ce noeud.
    - Exemples : Analyseur récursif descendant, Analyseur LL(1)
    - Il existe des solutions manuelles de programmation de ces parsers.
    - Chaîiange avant, dérivation de gauche à droite
  - Analyseur Ascendant (bottom-up)
    - Génère directement un automate (à pile = Machine de Turing) d'analyse et son implantation à partir de spécification syntaxique du langage
    - Algorithme :
      - Commence par les feuilles et procède en remontant l'arbre syntaxique jusqu'à la racine.
      - A chaque étape, l'analyseur ajoute des nœuds qui développent l'arbre partiellement construit.
      - Chaîiange arrière, dérivation de droite à gauche
    - Exemples : Analyseur LR(1), Analyseur LALR(1), Analyseur SLR(1)
      - LR(1) $\supset$ LALR(1) $\supset$ SLR(1)
    - Il existe des solutions automatiques de génération de ces parsers : **BISON**, **YACC**

Analyseur syntaxique descendant (top-down)

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Analyseur syntaxique descendant (top-down) récursif - à dérivation gauche

- *token ← next_token (lire le token suivant)*
- *root ← S (symbole non-terminal Start)*
- *node ← root*
- *Répéter jusqu'à la fin du programme à analyser*
  - *if (node ∈ T)*
    - *if (node ≅ token) then // **(a)** ce qui a été lu est ce qui est prévu*
      - *node ← next_node*
      - *token ← next_token*
    - *else // **(b)** ce qui a été lu est différent de ce qui est prévu*
      - *Effectuer un Backtracking !!*
  - *else if (node ∈ NT) then // **(c)** on rencontre un non-terminal*
    - *Choisir une règle de réécriture "node→β"*
    - *Développer l'arbre syntaxique sous node en ajoutant β*
    - *node ← symbole le plus à gauche de β*
  - *end if*
  - *if (node is empty)*
    - *if (token == eof) then*
      *// **(d)** tous les tokens ont été couverts par symboles de la règle courante*
      - *Accepter*
    - *else // **(e)** tous les tokens n'ont pas été couverts par symboles de la règle courante*
      - *Effectuer un Backtracking !!*
  - *end if*
- *Fin Répéter*

Exemples de cas d'analyse descendante

nodes	***tokens***
(a) Identifier – *Term*	**x - 2 × y**
(b) Identifier + *Term*	**x - 2 × y**
(c) Expr	**x - 2 × y**
(d) Identifier - Number × Number	**x - 2 × 3**
(e) Identifier – Number	**x - 2 × y**

Analyseur syntaxique descendant (top-down) récursif

- Exercice
  - Réfléchissez sur
    1. la **correction** ;
    2. Puis sur l'**efficacité** de cet algorithme d'analyse syntaxique descendante récursive !

Analyseur syntaxique descendant (top-down) récursif - correction

1. Correction

- Pour une grammaire du type
  - R1 : S → Mult
  - R2 : Mult → Mult × Aux
  - R3 : Mult → Aux
  - R4 : Aux → Nombre

- Problème :
  - L'algorithme risque de **développer** Mult en Mult × Aux puis en Mult × Aux × Aux puis en Mult × Aux × Aux × Aux × Aux × Aux × Aux × Aux **indéfiniment** (cas **(c)**) : **bouclage de l'analyseur** !!

- Solution :
  - **Éliminer la récursivité gauche** de la grammaire pour une analyse top-down

Analyseur syntaxique descendant (top-down) récursif - efficacité

2. Efficacité

1. Pour une grammaire du type G < T={identifier,[,],(,)}, NT={*S*, *Factor*, *Exprlist*}, S=S, P = {R_1, R_2, R_3, R_4}>
   - R1 : S → *Factor*
   - R2 : *Factor* → identifier
   - R3 : *Factor* → identifier [*Exprlist*]
   - R4 : *Factor* → identifier (*Exprlist*)

- Problème :
  - L'efficacité de l'analyseur descendant récursif dépend énormément du nombre de *backtracking* qu'il effectue après mauvais choix de règle à développer !!

- Solution :
  - **Détecter la règle de réécriture la plus adéquate** avant de même de la développer
  - Factoriser à gauche (les préfixes des parties droites des règles) (comme pour les dictionnaires de données). On parle *d'analyse syntaxique prédictive*.

Analyseur syntaxique descendant (top-down) récursif - efficacité

2. Efficacité

- R1 : S → *Factor*
- R2 : *Factor* → identifier
  - Select(R2) = First(identifier) = identifier
- R3 : *Factor* → identifier [*Exprlist*]
  - Select(R3) = First(identifier[Exprilist]) = identifier
- R4 : *Factor* → identifier (*Exprlist*)
  - Select(R4) = First(identifier(Exprilist)) = identifier
  - G n'est pas prédictive LL(1)
    - Select(R2) intesect Select(R3) <> vide = identifier
    - Select(R3) intesect Select(R4) <> vide = identifier
    - Select(R2) intesect Select(R4) <> vide = identifier

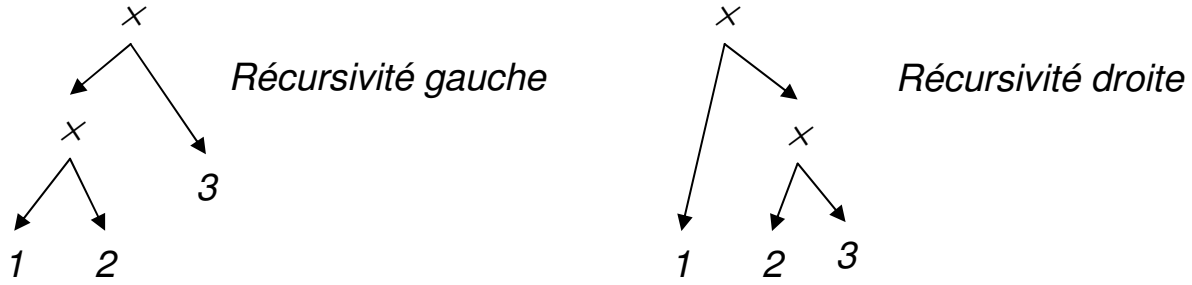
- On factorisera les partie droites de R2, R3 et R4 par leur intersection = identifier
- La factorisation est un algorithme itératif (factorisation (R4, factorisation (R2,R3)))
- R1 : S → *Factor*
- R2 : *Factor* → identifier *FactorAux*
- R3 : *FactorAux* → ε
- R4 : *FactorAux* → [*Exprlist*]
- R5 : *FactorAux* → (*Exprlist*)

Récursivité gauche

- Une règle de réécriture est dite récursive gauche si
  - le premier symbole sur la partie droite de la règle est le même que celui sur sa partie gauche
    - Exemple :
      - $S \rightarrow Sa$
  - ou si le symbole de la partie gauche de la règle apparaît sur sa partie droite et tous les symboles qui le précèdent peuvent dériver le mot vide
    - Exemple :
      - $S \rightarrow TSa$
      - $T \rightarrow \varepsilon \mid \ldots$
      - $S \rightarrow Ta$ (récursivité gauche par transitivité)
      - $T \rightarrow S \mid \ldots$

Récursivité droite *versus* gauche

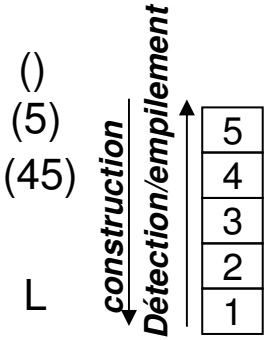
a) Associativité



b) Taille de la pile

a) Soit L='(1 2 3 4 5) une liste à reconnaître et construire

a) G1 : List → List Elt | Elt
- 5 4 3 2 1 est le sens de détection (*Elt* est en fin de *List → List Elt*)
- 5 4 3 2 1 est le sens de construction *(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 '()))))))*
- Utilisation **minimale** (cste + 1) de la pile



b) G2 : List → Elt List | Elt
- 1 2 3 4 5 est le sens de détection (*Elt* est au début de *List → Elt List*)
- 5 4 3 2 1 est le sens de construction *(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 '()))))))*
- On doit empiler tous les éléments afin de construire la liste
- Utilisation **maximale** (cste + 5) de la pile

Récursivité droite *versus* gauche

a) Évaluation arithmétique en VFN

- En Calcul virgule flottante (codage en Mantisse et Exposant), si N est assez grand et ε_i très petits (e.g. $N - \varepsilon_i > 10^{15}$) $\Rightarrow N + \varepsilon_i \equiv N$
  - Pour l'expression $N + \varepsilon_1 + \varepsilon_2 + ..+\varepsilon_n$:
    - Évaluation de gauche à droite donne $(((\ \ .. (N + \varepsilon_1)+ \varepsilon_2)+ ..+ \varepsilon_n) = N$:
  - Pour l'expression $\varepsilon_1 + \varepsilon_2 + ..+\varepsilon_n + N$:
    - Évaluation de gauche à droite donne réellement $\Sigma_i^n \varepsilon_i + N$, car par exemple $10^{15} >> N - \Sigma_i^n \varepsilon_i$
  - Cas symétriques pour l'évaluation de droite à gauche

b) Sens d'analyse syntaxique

- La récursivité gauche peut produire le bouclage de l'analyseur top-down, car elle peut permettre le développement de l'arbre syntaxique indéfiniment sans générer de symbole terminal

Récursivité droite *versus* gauche

Type	(b) Associativité	(b) Taille de la pile	(c) Évaluation arithmétique en VFN	(d) Sens d'analyse syntaxique
Récursivité gauche	***Généralement**** **Gauche**	**Économe** **+ +**	**Évaluation de gauche-à-droite**	**Peut produire le bouclage du parser Top-down** **- -**
Récursivité droite	***Généralement**** **Droite**	**Gourmande** **- -**	**de droite-à-gauche**	**Adaptée au parser Top-down** **+ +**

** Il existe des grammaires récursives droites équivalentes à des grammaires récursives gauches ! Donc conservant la même associativité !!*

Élimination de la récursivité gauche

– Transformation de la récursivité directe

- *α et β sont une suite de NT et de T*
- ***G : Grammaire récursive gauche (NT = {S})***
  - ***S*** → ***S*** *α* *(i.e. tout mot de L(G) se termine pas une suite de α)*
  - | *β* *(i.e. tout mot de L(G) commence par β ou contient seulement β)*

  - $S \Rightarrow S\,\alpha \Rightarrow S\,\alpha\,\alpha \Rightarrow \ldots \Rightarrow S\,\alpha^* \Rightarrow$ ***β α****
  - ***L(G') = β α****

- ***G' : Grammaire non-récursive gauche équivalante à G (NT' = {S,R})***
  - ***S*** → *β* ***R*** *(i.e. tout mot de L(G') commence par β)*
  - ***R*** → *α* ***R*** *(i.e. tout mot de L(G') se termine éventuellement pas une suite de α)*
  - | ε

  - $S \Rightarrow \beta\,R \Rightarrow \beta\,\alpha\,R \Rightarrow \alpha\,\alpha\,R \Rightarrow \ldots \Rightarrow \alpha^*\,R \Rightarrow$ ***β α****
  - ***L(G') = β α****

- L(G) = L(G') donc la transformation de G en G' a conservé le langage reconnu

Élimination de la récursivité gauche – Idée de l'a*lgorithme général*

- *Récursivité gauche directe :*
  - *L'algorithme élimine systématiquement la récursivité gauche par la transformation directe*

- *Récursivité gauche cachée :*
  - *S'il existe des règles $\alpha \rightarrow \beta$, $\beta \rightarrow \gamma$ et $\gamma \rightarrow \alpha\delta$, l'algorithme les combine et détecte la récursivité gauche indirecte $\alpha \rightarrow \alpha\delta$, puis applique la transformation de la récursivité directe*

Exercice

- Éliminer la récursivité gauche de la grammaire suivante :
  - G'_{arith} < T={Nombre,+, ×}, NT={Mult, Add, Aux}, S=Add, P = {R_1, R_2, R_3, R_4, R_5, R_6}>
    - R1 : Mult $\rightarrow$ Mult $\times$ Aux
    - R2 : Mult $\rightarrow$ Aux
    - R3 : Aux $\rightarrow$ Nombre
    - R4 : Aux $\rightarrow$ (Add)
    - R5 : Add $\rightarrow$ Add + Mult
    - R6 : Add $\rightarrow$ Mult

Solution

- G'$_{arith}$ contient de la récursivité gauche directe

Règles récursives gauche NT={Mult, Aux, Add}	**Grammaire récursive droite équivalente (i.e. conserve l'associativité droite et les priorités de G'$_{arith}$) NT={Mult, Mult', Aux, Add, Add'}**
R_1 : Mult → Mult × Aux R_2 : Mult → Aux R_3 : Aux → Nombre R_4 : Aux → (Add) R_5 : Add → Add + Mult R_6 : Add → Mult	// **Transformation directe de Mult** Mult → Aux Mult' Mult' → × Aux Mult' |ε // **Aux n'est pas récursive gauche** Aux → Nombre Aux → (Add) // **Transformation directe de Add** Add → Mult Add' Add' → + Mult Add' |ε

Question

- Est-ce qu'on éliminant la récursivité gauche (qui nous a servi pour coder l'asociativité gauche nécessaire à la désambiguisation) on élimine l'associativité gauche ??
  - Non
  - Car, L'élimination de la récursivité gauche, transforme l'écriture de la grammaire sans altérer ni le langage ni ses propriétés inhérentes à cette grammaire (ex. l'associativité gauche)

Élimination de la récursivité gauche - *Algorithme général*

1. *Arranger les symboles non-terminaux dans un certain ordre*
   - $A_1, A_2, \ldots, A_n$

2. *Pour i ← 1 to n {*
   1. *Pour j ← 1 to i-1 {*
      - *Remplacer Input par Output*
        - ***Input** : Règle de réécriture de la forme $\boldsymbol{A_i \rightarrow A_j \gamma}$*
        - *où $\boldsymbol{A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k}$ sont toutes les règles actuelles de A_j*
        - ***Output** : Règles $\boldsymbol{A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma}$,*
        - *(remplacement de $\boldsymbol{A_j}$ par sa partie droite)*
      - *} // fin Pour*
   2. *Éliminer toute récursivité gauche immédiate sur A_i en utilisant la transformation directe (du transparent précédent)*
   - *} // fin Pour*

Grammaire prédictive (dite LL(1))
Left-to-right with **L**eftmost parse

- Soit ***R: A*** → ***α*** une règle de production, on définit **Select(R)** = **First(*α*)** que l'on définit comme le critère de sélection de la règle R
  - {ensemble des symboles terminaux qui peuvent apparaître comme premiers symboles dans les mots dérivables à partir de *α}*

- Soit ***G = <T, NT, S, P>*** une grammaire, G est dite prédictive (dite LL(1)) ssi
- ∀ les règles A→ $\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \cdots \alpha_n$ de **P (un NT qui dérive plusieurs règles)**
  - ∀ i<>j $First(\alpha_i) \cap First(\alpha_j) = \varnothing$

Grammaire prédictive (dite LL(1))

- R1 : S → T
- R2 : T → x | B
- R3 : B → y | Z
- R4 : Z → t

- **R1 : S → T**
- **T est un NT qui dérive plusieurs règles**
- **R21 : T → x Select(R21) = First(x)={x}**
- **R22 : T → B Select(R22) = First(B)=First(y|Z) = First(y) union First(Z) = {y, t} intersection vide avec {x} !!**
- **B est un NT qui dérive plusieurs règles**
- **R31 : B → y Select(R31) = First(y) = {y}**
- **R32 : B → Z Select(R32) = First(Z) = {t} intersection vide avec {y} !**
- **R4 : Z → t**

- Linéarité de la fonction First

Select(R1) = First(T) = {x,y,t}
First(T) = First(x | B) = {x} union First(B)
First(B) = First(y | Z) = {y} union First(Z)
First(Z) = {t}

Algorithme de factorisation à gauche

- *for each* $A \in NT$
  - *Trouver le plus long préfixe « α » commun à deux ou plusieurs parties droites de règles relatives à A*
  - *if* $(\alpha \neq \varepsilon)$ *then*
    - *remplacer les règles de A de la forme*
      - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$
    - *par :*
      - $A \rightarrow \alpha\ B \mid \gamma$
      - $B \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$
      - $NT \leftarrow NT \cup \{B\}$ *// ajouter B à l'ensemble des symboles non-terminaux*
- *Répéter jusqu'à ce qu'il n'y ait plus de préfixe commun*

Exercice

- *Soit la grammaire suivante*
  - *G5 :*
    - ***R1: S*** → ***Factor*** *% une seule règle*
    - ***R2: Factor*** → **identifier** % Select(R2) = First(Identifier)={identifier}
    - ***R3:*** / **identifier [** ***Exprlist*** **]** % Select(R3) = First(Identifier[*Exprlist*])={identifier}
    - **R4** / **identifier (** ***Exprlist*** **)** % Select(R4) = First(Identifier(*Exprlist*))={identifier}
- G5 n'est pas une grammaire prédictive LL(1)
  - First(identifier) ∩ First(identifier [*Exprlist*]) ≠∅
- *Rendez la grammaire G5 prédictive LL(1)*

Grammaire prédictive LL(1)

- *G5':*
  - *Factor* → Identifier *Arguments*
  - *Arguments* → [*Exprlist*]
  - | (*Exprlist*)
  - | ε
- *G5' est la grammaire prédictive équivalente à G5:*
  - ∀ i,j First(α_i) ∩ First(α_j) = ∅
    - First([Exprlist]) ∩ First((Exprlist)) = ∅
    - First([Exprlist]) ∩ First(ε) = ∅
    - First((Exprlist)) ∩ First(ε) = ∅

Etape de réalisation d'un parser descendant LL(1)

1. Désambiguïsation de la grammaire
2. Élimination de la récursivité gauche la grammaire résultante de (1)
3. Rendre la grammaire résultante LL(1) de (2)
4. Transformer l'ensemble des non terminaux en un ensemble de fonctions (prédicats) (en langage de programmation de votre choix)

Analyseur syntaxique descendant (top-down) LL(1) prédictif

- Pour chaque non-terminal $A \in NT$ de la grammaire LL(1) tel que :
  - $A \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$
- On programme une routine *Parse_A()* qui reconnaît ce non-terminal *A*. Chaque routine dépend des autres routines pour reconnaître les non-terminaux et teste directement les symboles terminaux qui apparaissent dans sa propre partie droite

- Boolean Parse_*A()*{
  - *if* (*current_token* $\in$ First(β_1))
    - *Reconnaître* β_1 *; return true*
  - *else if* (*current_token* $\in$ First(β_2))
    - *Reconnaître* β_2 *; return true*
  - *…*
  - *else if* (*current_token* $\in$ First(β_n))
    - *Reconnaître* β_3 *; return true*
  - *else {*
    - *reporter une erreur basée sur A et le token courant (current_token)*
    - *return false*
  - *}*
- *} //* On a $\forall$ i (*current_token* $\in$ First(β_i)) sont des conditions disjointes du fait que la grammaire est LL(1) $\forall$ i,j First(β_i) $\cap$ First(β_j) = $\varnothing$,

Analyseur syntaxique descendant (top-down) LL(1) prédictif

- $A \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$
- *Avec* $\beta_1 = \gamma\delta\rho$, with $\gamma, \rho \in NT$ and $\delta \in T$,

- *Reconnaître* β_1 :
  - *if (Parse_*γ*() = false)*
    - *return false*
  - *else if (current_token /= δ) {*
    - *reporter une erreur à reconnaître δ dans la règle* $A \rightarrow \gamma\delta\rho$
    - *return false*
  - *}*
  - *current token ← next_token()*
  - *if (Parse_*ρ*() = false)*
    - *return false*
  - *else*
    - *return true %% j'ai reconnu* $\beta_1 = \gamma\delta\rho$

Analyseurs LL - Généralisation

- Analyseur LL(k)
  - Un analyseur prédictif LL est appelé LL(*k*) s'il a besoin d'avoir une visibilité de *k tokens* pour analyser une expression sans backtracking.
  - Analogie avec les DFA :
    - LL(1) est plus déterministe (au sens des automates) que LL(k > 1) bien évidemment !!
    - Pouvez-vous illustrer cette analogie ?

Analyseur LL – synthèse

- Force
  - Il est simple de créer des parsers (analyseurs syntaxiques) manuellement à partir des grammaires LL
  - En plus un parser LL possède les avantages suivants
    - **Rapidité**
    - **Localité spatiale du code** (l'analyseur accède relativement à une petite portion d'espace d'adressage durant toute son exécution)
    - **Bonne stratégie de détection d'erreur**
    - **Code orienté par la la grammaire** (représentation concise, haut niveau)
    - **Effort d'implantation réduit.**
- Limite
  - Malheureusement, toutes les grammaires ne peuvent pas être prédictives. Exemple, le language
  - $L = \{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$ *n'admet pas de grammaire HC prédictive LL(1)*
- Idées de solution
  - D'autres analyseurs (ascendants) permettent de reconnaître les grammaires qui ne sont pas prédictives (non LL) (i.e. LR, LALR, SLR).

Non-prédictivité d'un langage

- *Pourquoi $L = \{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$ n'admet pas de grammaire HC prédictive LL(1) ?*
- *Soit G la grammaire reconnaissant L*
  - G : < T={0, 1, a, b}, NT={S1, S2}, S=S1, P = {R_1, R_2, R_3 , R_4, R_5}>
    - R1 : S → S1 | S2
    - R2 : S1 → a S1 b
    - R3 : S1 → 0
    - R4 : S2 → a S2 bb
    - R5 : S2 → 1
    - First(S) = First(S1) ∩ First(S2) = {a, 0} ∩ {a, 1} = {a} ≠ ∅
    - G n'est donc pas prédictive et ne peut pas le devenir car si l'on écrit S → aS', on ne pourra pas éliminer a du début de S1 et de S2, c'est un élément qui doit se répéter à chaque production !

Transformations des grammaires CFG – pour l'analyseur syntaxique

1. Éliminer l'ambiguïté ;
2. Éliminer la récursivité gauche ;
3. Rendre la grammaire prédictive LL(1).

- D'autres transformations n'en sont pas moins importantes

Autres transformations intuitives des grammaires CFG - Exercice

- Trouver intuitivement les problèmes qui paraissent clairement dans les grammaires CFG suivantes :
  - G1 : < T={Nombre, ×}, NT={Mult, Aux1, Aux2}, S=Mult, P = {R_1, R_2, R_3 , R_4}>
    - R1 : Mult → Mult × Aux1
    - R2 : Mult → Aux1
    - R3 : Aux1 → Aux2
    - R4 : Aux2 → Nombre
  - G2 : < T={Nombre, +, (}, NT={Add, Aux1, Aux2}, S=Aux1, P = {R_1, R_2, R_3 , R_4}>
    - R1 : Aux1 → (Add)
    - R2 : Add → Add + Nombre
    - R3 : Add → Nombre
    - R4 : Aux2 → Aux1

Aux2 : code mort, sommet inaccesible

Autres transformations intuitives des grammaires CFG - Suite

- Trouver intuitivement les problèmes posés par les grammaires CFG suivantes :
  - G3 : < T={+, (}, NT={Add, Aux}, S=Aux, P = {R_1, R_2, R_3}>
    - R1 : Aux → (Add)
    - R2 : Add → Add + Aux
    - R3 : Add → Aux
  - G4 : < T={Nombre, +}, NT={Add, Aux, Aux1}, S=Add, P = {R_1, R_2, R_3 , R_4 , R_5}>
    - R1 : Aux1 → Add
    - R2 : Add → Add + Aux
    - R3 : Add → Aux
    - R4 : Aux → Nombre
    - R5 : Aux → Aux1

Autres transformations intuitives des grammaires CFG

1. Transformation optionnelle :
    1. **Minimiser la grammaire** : (1) pour la simplifier et (2) pour optimiser le temps de traitement de l'analyseur syntaxique

2. Transformations critiques partant du plus simple à détecter au plus dur :
    1. **Corriger la non accessibilité** d'un symbole non-terminal de la grammaire
    2. **Corriger la cyclicité** de la grammaire
    3. **Corriger la non-convergence** de la grammaire à des symboles terminaux

Exercice

- Soit la grammaire suivante, réaliser un analyseur syntaxique qui lui correspond :
- G :< T= {‘a’,’.’}, NT={S} ,start=S, P={R1,R2}>
  - R1: S $\rightarrow$ aS
  - R2: S $\rightarrow$ a.

  - G n’est pas LL(1)
    - Select(R1) = First(aS) = {a}
    - Select(R2) = First(a.) = {a}
    - First(a.) $\cap$ First(aS) = {a} $\neq \varnothing$

Solution 1 - intuitive : sans transformation LL(1) de G

- boolean a=false ;
-
- boolean S() {
  - char c = getchar() ;
  - if (c == 'a'){
    - a = true ;
    - return S() ;
  - }else{
    - if ((c == '.') && (a == true)) { // on se souvient qu'on a déjà lu 'a'
      - return true;
    - }else{
      - return false // on a lu un caractère qui n'appartient par à l'ensemble des terminaux T
    - }
  - }
- }

Solution 2 - réfléchie avec transformation LL(1) de G – tactique (1) : le NT lit ses tokens

- G :< T= {‘a’,’.’}, NT={S} ,start=S, P={R1,R2}>
  - R1: S → aS
  - R2: S → a.

- Soit G’ :< T= {‘a’,’.’}, NT={S,T} ,start=S, P={R1’,R2’}>
  - R1’: S → aT
  - R2’: T → S
  - R2’: T → .

- boolean S() ; // la routine correspondant au non-terminal S
- boolean T() ; // la routine correspondant au non-terminal T

- char c;
- int main(char* argv[], int argc){
  - S(); // Start = S
- }
- boolean S() {
  - c = getchar() ;
  - if (c == ’a’){
    - return T() ; // S → aT
  - }else{
    - return false
  - }
- }

- boolean T() {
  - c = getchar() ;
  - if (c == ’.’){
    - return true ; // T → .
  - }else{
    - ungetc(c, stdin);
    - return S(); // T → S
  - }
- }

Solution 3 - réfléchie avec transformation LL(1) de G – tactique (2) : le NT ne lit pas ses tokens

- G :< T= {‘a’,’.’}, NT={S} ,start=S, P={R1,R2}>
  - R1: S → aS
  - R2: S → a.

- Soit G’ :< T= {‘a’,’.’}, NT={S,T} ,start=S, P={R1’,R2’}>
  - R1’: S → aT
  - R2’: T → S
  - R2’: T → .

- boolean S() ; // la routine correspondant au non-terminal S
- boolean T() ; // la routine correspondant au non-terminal T

- char c;
- int main(char* argv[], int argc){
  - char cglobal = getchar() ;
  - S(cglobal); // Start = S
- }
- boolean S(char c) {
  - if (c == ’a’){
    - cglobal = getchar() ;
    - return T(cglobal) ; // S → aT
  - }else{
    - return false
  - }
- }

- boolean T(char c) {
  - if (c == ’.’){
    - return true ; // T → .
  - }else{
    - return S(c); // T → S
  - }
- }

Exercices en Analyse syntaxique

Extraits d'examens de l'ENSIMAG 2001, 2000 et 1996

Prof. K. Baïna 2010 ©, 7ème édition

Exercices

- Quelle est la complexité en temps de reconnaissance d'un mot par une grammaire linéaire ?

- Quel gain en complexité si l'on réduit (optimise) le nombre de non terminaux dans une grammaire ?

Ex1 : Analyse syntaxique (7 points) – ENSIMAG Sep. 2001

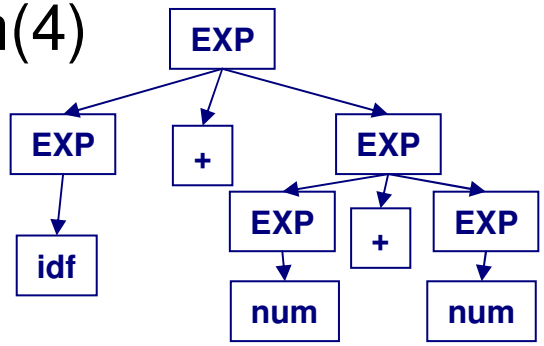
- Soit la grammaire suivante (sous-ensemble des expressions entières du langage C)
- G1 ::< T= {idf,num,=,(,),*,-,+,',',}, NT={Exp},start=Exp, P={R_1,R_2,R_3,R_4,R_5,R_6,R_7,R_8}>

  - R_1 : Exp → idf = Exp
  - R_2 : Exp → Exp , Exp
  - R_3 : Exp → (Exp)
  - R_4 : Exp → Exp + Exp
  - R_5 : Exp → Exp - Exp
  - R_6 : Exp → Exp * Exp
  - R_7 : Exp → idf
  - R_8 : Exp → num

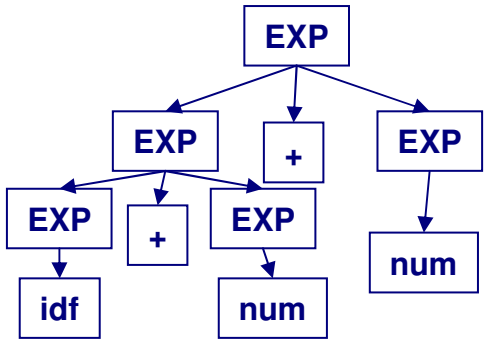
- Q1 - Montrer que G1 est ambiguë

Solution Q1 – G1 est ambiguë

- Prenons x + 3 + 4
  - Arbre 1 : Exp ⇒ Exp + Exp ⇒ idf + Exp ⇒ idf + Exp + Exp ⇒ idf + num + Exp ⇒ idf + num + num : idf(x) + num(3) + num(4)



Arbre 1



Arbre 2

  - Arbre 2 : Exp ⇒ Exp + Exp ⇒ Exp + Exp + Exp ⇒ idf + Exp + Exp ⇒ idf + num + Exp ⇒ idf + num + num : idf(x) + num(3) + num(4)

Ex1 : Analyse syntaxique – ENSIMAG Sep. 2001 (suite)

- Q2 - Construire une grammaire G2, non, ambiguë, en tenant compte des priorités et associativités suivantes
  - R_1 : Exp → idf = Exp
  - R_2 : Exp → Exp , Exp
  - R_3 : Exp → (Exp)
  - R_4 : Exp → Exp + Exp
  - R_5 : Exp → Exp - Exp
  - R_6 : Exp → Exp * Exp
  - R_7 : Exp → idf
  - R_8 : Exp → num

Op	Priorité	Associativité
*	⊕ 4	Gauche
+ -	3	Gauche
=	2	Droite
,	1	Gauche

Solution Q2 – Désambiguïsation - Méthode

1. Commencer par traiter l'associativité d'un seul opérateur (par exemple *)
   - MULT → MULT * E
   - MULT → E
   - E → idf
   - E → num
2. Puis traiter l'associativité d'un autre opérateur (par exemple +)
   - ADD → ADD + E
   - ADD → E
3. Ensuite merger + et * en respectant la priorité de l'* sur le +
   - ADD → ADD + MULT
   - ADD → MULT
4. Ensuite généraliser pour les autres opérateurs dans l'ordre de la priorité (+ et – auront les mêmes priorités donc les confendre en une même classe d'équivalence ADDSUB par exemple)
5. Ensuite Introduire les expressions parenthésées

- On partage Exp en 4 classes disjointes(Vir, Eq, AddSubb, Mult)
- On fixe l'associativité de ces classes et donc leur récursivité
- On détermine l'ordre de dépendance entre les classes : Vir --> Eq --> AddSub --> Mult --> (Vir)
- On commence à traiter la classe qui dépend le moins des autres Mult (la plus prioritaire), petit à petir jusqu'à converger à la moins prioritaire

Solution Q2 – Grammaire désambiguïsée (preuve : récurrence sur les 5 étapes de construction)

- G2 ::< T= {idf,num,=,(,),*,-,+,',',}, NT={VIR,EQ,ADDSUB,MULT,EXP}, start=VIR, P={R_1,R_2,R_3,R_4,R_5,R_6,R_7,R_8,R_9,R_{10},R_{11},R_{12}}>
  - R1: VIR → EQ
  - R2: VIR → VIR , EQ
  - R3: EQ → idf = EQ %% cas particulier la grammaire d'origine était simplement récursive
  - R4: EQ → ADDSUB %% à droite pas doublement récursive !!
  - R5: ADDSUB → ADDSUB + MULT
  - R6: ADDSUB → ADDSUB – MULT
  - R7: ADDSUB → MULT
  - R8: MULT → MULT * EXP
  - R9: MULT → EXP
  - R10: EXP → idf
  - R11: EXP → num
  - R12: EXP → (VIR)

Best Practice (Astuce)

- La démarche de désambigüisation part du plus prioritaire (la classe qui ne dépend de rien) au moins prioritaire (la classe qui dépend de toutes les autres)

- Mais le moins prioritaire constitue toujours le Start de la grammaire

Ex1 : Analyse syntaxique – ENSIMAG Sep. 2001 (suite)

- Q3 – Construire une grammaire G3, équivalente à G1, qui soit LL(1).
  a. Préciser les étapes de la construction
  b. Démontrer le caractère LL(1) de G3

Solution Q3 – Elimination de la récursivité gauche (avant la transformation LL(1))

- G3 ::< T= {idf,num,=,(,),*,-,+,',',}, NT={VIR,VIRAUX,EQ,EQ,ADDSUB,ADDSUBAUX,MULT,MULTAUX,EXP}, start=VIR, P={R_1',R_2',R_3',R_4',R_5',R_6',R_7',R_8',R_9',R_{10}',R_{11}',R_{12}',R_{13}',R_{14}',R_{15}',R_{16}'}>
  - R1': VIR → EQ VIRAUX %% **(8)** Start=VIR n'est pas récursive gauche car EQ ne peut pas dériver VIR
  - de **(1)..(8)**, la grammaire G, n'est pas récursive gauche !!
  - R2': VIRAUX → , EQ VIRAUX %% **(7)** VIRAUX n'est pas récursive gauche !!
  - R3': VIRAUX → ε
  - R4': EQ → idf = EQ %% **(6)** EQ est déjà récursive droite
  - R5': EQ → ADDSUB %% et ne peut pas être récursive gauche car ADDSUB ne peut pas dériver EQ !
  - R6': ADDSUB → MULT ADDSUBAUX %% **(5)** ADDSUB n'est pas réc. gche. car MULT ne peut pas dériver ADDSUB
  - R7': ADDSUBAUX → – MULT ADDSUBAUX %% **(4)** ADDSUBAUX n'est pas récursive gauche !!
  - R8': ADDSUBAUX → + MULT ADDSUBAUX
  - R9': ADDSUBAUX → ε
  - R10': MULT → EXP MULTAUX %% **(3)** MULT n'est pas récursive gauche car EXP ne peut pas dériver MULT
  - R11': MULTAUX → * EXP MULTAUX %% **(2)** MULTAUX n'est pas récursive gauche !!
  - R12': MULTAUX → ε
  - R13': EXP → idf %% **(1)** EXP n'est pas récursive gauche !!
  - R14': EXP → num
  - R15': EXP → (VIR)

Solution Q3 – Calcul des directeurs (ou Select) des règles

- **R1': VIR → EQ VIRAUX** %% une seule règle

- **R2': VIRAUX → , EQ VIRAUX** %% Select(R2') = First(, EQ VIRAUX) = {,}
- **R3': VIRAUX → ε** %% Select(R3') = First(ε) = ∅

- **R4': EQ → idf = EQ** %% Select(R4') = First(idf = EQ) = {**idf**}
- **R5': EQ → ADDSUB** %% Select(R5') = First(ADDSUB) = First(MULT) = First(EXP)={**idf**,num,(}
- %% Select(R4') ∩ Select(R5') = {**idf**} ≠ ∅ ⇒ $\neg LL_{(1)}$(EQ) ⇒ $\neg LL_{(1)}$(VIR)
- **R6': ADDSUB → MULT ADDSUBAUX** %% une seule règle

- **R7': ADDSUBAUX → – MULT ADDSUBAUX** %% Select(R7') = First(– MULT ADDSUBAUX) ={-}
- **R8': ADDSUBAUX → + MULT ADDSUBAUX** %% Select(R8') = First(+ MULT ADDSUBAUX) ={+}
- **R9': ADDSUBAUX → ε** %% Select(R9') = First(ε) = ∅

- **R10': MULT → EXP MULTAUX** %% une seule règle

- **R11': MULTAUX → * EXP MULTAUX** %% Select(R11') = First(* EXP MULTAUX) = {*}
- **R12': MULTAUX → ε** %% Select(R12') = First(ε) = ∅

- **R13': EXP → idf** %% Select(R13') = First(idf) = {idf}
- **R14': EXP → num** %% Select(R14') = First(num) = {num}
- **R15': EXP → (VIR)** %% Select(R15') = First((VIR)) = { (}

Solution Q3 – factorisation à gauche de EQ

- Partie ¬LL(1) de la garammaire G3
  - R4': EQ → idf = EQ
  - R5': EQ → ADDSUB
  - R6': ADDSUB → MULT ADDSUBAUX
  - R10': MULT → EXP MULTAUX
  - R13': EXP → idf
  - R14': EXP → num
  - R15': EXP → (VIR)

- Par transitivité, on développe EXP dans MULT, puis MULT dans ADDSUB, puis ADDSUB puis EQ
  - R40': EQ → idf = EQ
  - R41': EQ → idf MULTAUX ADDSUBAUX
  - R42': EQ → num MULTAUX ADDSUBAUX
  - R43': EQ → (VIR) MULTAUX ADDSUBAUX

- Puis on factorise à gauche par Select(R4') ∩ Select(R5') = {idf}
  - R40'': EQ → idf EQAUX %% Select(R40'') = First(idf) = {idf}
  - R41'': EQ → num MULTAUX ADDSUBAUX %% Select(R41'') = First(num MULTAUX ADDSUBAUX) = {num}
  - R42'': EQ → (VIR) MULTAUX ADDSUBAUX %% Select(R42'') = First((VIR) MULTAUX ADDSUBAUX) = {'('}

  - R50'': EQAUX → = EQ %% Select(R50'') = First(= EQ) = { '=' }
  - R51'': EQAUX → MULTAUX ADDSUBAUX %% Select(R51'') = First(MULTAUX) = { '*' }

Solution Q3 – factorisation à gauche de EQ

- R1': VIR → EQ VIRAUX %% une seule règle

- R2': VIRAUX → , EQ VIRAUX %% Select(R2') = First(, EQ VIRAUX) = {,}
- R3': VIRAUX → ε %% Select(R3') = First(ε) = ∅

- R40'': EQ → idf EQAUX %% Select(R40'') = First(idf) = {idf}
- R41'': EQ → num MULTAUX ADDSUBAUX %% Select(R41'') = First(num MULTAUX ADDSUBAUX) = {num}
- R42'': EQ → (VIR) MULTAUX ADDSUBAUX %% Select(R42'') = First((VIR) MULTAUX ADDSUBAUX) = {'('}

- R50'': EQAUX → = EQ %% Select(R50'') = First(= EQ) = { '=' }
- R51'': EQAUX → MULTAUX ADDSUBAUX %% Select(R51'') = First(MULTAUX) = { '*' }

- R6': ADDSUB → MULT ADDSUBAUX %% une seule règle

- R7': ADDSUBAUX → – MULT ADDSUBAUX %% Select(R7') = First(– MULT ADDSUBAUX) ={-}
- R8': ADDSUBAUX → + MULT ADDSUBAUX %% Select(R8') = First(+ MULT ADDSUBAUX) ={+}
- R9': ADDSUBAUX → ε %% Select(R9') = First(ε) = ∅

- R10': MULT → EXP MULTAUX %% une seule règle

- R11': MULTAUX → * EXP MULTAUX %% Select(R11') = First(* EXP MULTAUX) = {*}
- R12': MULTAUX → ε %% Select(R12') = First(ε) = ∅

- R13': EXP → idf %% Select(R13') = First(idf) = {idf}
- R14': EXP → num %% Select(R14') = First(num) = {num}
- R15': EXP → (VIR) %% Select(R15') = First((VIR)) = { (}

Ex2 : Analyse syntaxique (9 points) – ENSIMAG Déc. 2000

- On étudie un petit langage, que l'on nommera ZZ, dont les programmes ont la forme suivante

- var %% section de déclaration de variables

  - b : bool := true ; %% une déclaration possible avec initialisation
  - c : bool := false ; %% deux types possibles int et bool
  - x : int := 10 ;
  - y, z : int ; %% plusieurs déclarations possibles sans initialisation

- begin %% section de déclaration de variables

  - z == y + 1; %% deux opérateurs possibles d'affectations == et :=
  - b := not b or c and true ; %% expression booléennes
  - c := not c ;
  - x := x + 11 ; %% expression arithmétique entière
  - y == 42 * x ;

- end

Ex2 : Analyse syntaxique (9 points) – ENSIMAG Déc. 2000 (suite)

- Soit l'ensemble des terminaux T = {BEGIN, VAR, END, BOOL, INT, TRUE, FALSE, OR, NOT, AND, "+", "*", ":=", "==", ";", ",", ":", "(", ")", idf, const}
  - **idf** représente tous les identificateurs comme b
  - **const** représente toues les notations décimales de constantes comme 42

- Q1 (2 pts) –
  a. Donner une grammaire G qui soit LL(1) pour le sous-langage des expressions entières du langage « **EXP_INT** ».. On veillera à tenir compte des priorités usuelles des opérateurs entiers (* plus prioritaire que +)
  b. Justifier le caractère LL(1) de G

La grammaire EXP_INT (LL(1))

- R1 : EXP_INT → ADD
- R2: ADD → MULT ADDAUX
- R3 : ADDAUX → **+** MULT ADDAUX — Select(R3) = {'+'}
- R4: | ε — Select(R4) = ∅
- R5: MULT → EXPI MULTAUX
- R6: MULTAUX → * EXPI MULTAUX — Select(R6) = {'*'}
- R7: | ε — Select(R7) = ∅
- R8: EXPI → IDF — Select(R8) = {IDF}
- R9: | CONST — Select(R9) = {CONST}
- R10: | (EXP_INT) — Select(R10) = {'('}

Ex2 : Analyse syntaxique (9 points) – ENSIMAG Déc. 2000 (suite)

- Q2 (2 pts) –
  a. Donner une grammaire G qui soit LL(1) pour le sous-langage des expressions booléennes du langage « **EXP_BOOL** ». On veillera à tenir compte des priorités usuelles des opérateurs booléens (**and** plus prioritaire que le **not** plus prioritaire que **or)**
  b. Justifier le caractère LL(1) de G

La grammaire EXP_BOOL (LL(1))

- R1': EXP_BOOL → OR
- R2': OR → NOT ORAUX
- R3': ORAUX → **or** NOT ORAUX — Select(R3') = {or}
- R4': | ε — Select(R4') = ∅
- R5': NOT → AND NOTAUX
- R6': NOTAUX → **not** AND NOTAUX — Select(R6') = {not}
- R7': | ε — Select(R7') = ∅
- R8': AND → EXPB ANDAUX
- R9': ANDAUX → **and** EXPB ANDAUX — Select(R9') = {and}
- R10': | ε — Select(R10') = ∅
- R11': EXPB → IDF — Select(R11') = {IDF}
- R12': | TRUE — Select(R12') = {TRUE}
- R13': | FALSE — Select(R13') = {FALSE}
- R14': | (EXP_BOOL) — Select(R14') = {'('}

Ex2 : Analyse syntaxique (9 points) – ENSIMAG Déc. 2000 (suite)

- Q3 (2 pts) –
  - On cherche à décrire le langage de toutes les expressions du langage ZZ, en écrivant
    - R: EXP → EXP_INT | EXP_BOOL
  a. Dire pourquoi une grammaire de toutes les expressions qui contient la règle R ne peut pas être LL(1)
  b. Indiquer une idée pour obtenir une grammaire LL(1) de toutes les expressions du langage ZZ

La grammaire EXP (non LL(1))

- R: EXP → EXP_INT
  - Select(R) = First(EXP_INT) = First(ADD) = First(MULT) = First(EXP) = {IDF, CONST,'('}
- R': EXP → EXP_BOOL
  - Select(R') = First(EXP_BOOL) = First(OR) = First(NOT) = First(AND) = First(EXPB) = {IDF, TRUE, FALSE,'('}
- Select(R') ∩ Select(R') = {IDF}
- Idées pour la rendre LL(1)
  - Objectif :
    - Faire remonter IDF dans EXP_INT et dans EXP_BOOL fin de le factoriser à gauche EX, de la sorte à avoir : EXP→ IDF EXP' et EXP' → EXP_INT' | EXP_BOOL'
  - Comment :
    - On développe EXPB dans AND, et AND dans NOT, et NOT dans OR, et OR dans EXP_INT
    - On développe EXPI dans MULT, et MULT dans ADD, et ADD dans EXP_INT

Ex2 : Analyse syntaxique (9 points) – ENSIMAG Déc. 2000 (suite)

- Q4 (3 pts) –
  a. Donner une grammaire LL(1) du langage ZZ différenciant clairement les déclarations des instructions. On supposera l'existence d'un non-terminal EXPR qui décrit toutes les expressions du langage ZZ.
  b. Justifier le caractère LL(1) de la grammaire trouvée.

    - Solution « similaire » à l'énoncé de l'exercice suivant !

Ex3 : Analyse syntaxique (7 points) – ENSIMAG Déc. 1996

- On s'intéresse à un langage permettant de manipuler des expressions ensemblistes, les ensembles étant des ensembles d'entiers. Un programme est une suite de déclarations suivie d'une suite d'instructions. Les non-terminaux *program*, *liste-inst*, *inst*, *decl* et *type* sont décrits par les règles suivantes :
  - Sur l'ensemble T = {BEGIN, END, SET, INTERVAL, IDF, ":=", ";", ":"}, on définit la grammaire suivante.
  - G:
    - R1: Program → Liste_Decl **begin** Liste_Inst **end**
    - R2: Liste_Decl → Liste_Decl Decl | ε
    - R3: Decl → **idf** : Type ;
    - R4: Type → **set** | **interval**
    - R5: Liste_Inst → Liste_Inst Inst | ε
    - R6: Inst → **idf** := Exp ;
  - Le type set désigne le type des ensembles d'entiers et le type interval désigne le type des intervalles d'entiers

Ex3 : Analyse syntaxique (7 points) – ENSIMAG Déc. 1996 (suite)

- Soit l'ensemble des terminaux suivant
  - T = {idf, num, "." , "∪", "∩", "–" , "(", ")", "{", "}"}
- Soit LSET le langage des expressions ensemblistes construites à partir :
  - d'identificateurs
  - de constantes d'intervalles, noté **num .. num**)
  - de constantes d'ensembles, définis en extension, noté **{num, …, num}** ou **{}** pour l'ensemble vide
  - des opérateurs binaires union noté ∪, intersection noté ∩ et différence noté –
  - des expressions parenthésées

Ex3 : Analyse syntaxique (7 points) – ENSIMAG Déc. 1996 (suite)

- Q1 (2 pts) –
  - Donner une grammaire non ambiguë des expressions du langage LSET prenant en compte les propriétés des opérateurs en tenant compte des priorités et associativités suivantes
  - Exemples de mots : 3..5 ∩ {}, (1..3 ∪ x) ∩ {2,5,3}, x ∩ y – z

Opérateur	Priorité ⊕	Associativité
∩	4	Gauche
–	3	Gauche
∪	2 ↓	Gauche

Ex3 : Analyse syntaxique (7 points) – ENSIMAG Déc. 1996 (suite)

- Q2 (2 pts) –
  a. Donner une grammaire LL(1) pour le langage LSET
  b. Justifier le caractère LL(1) de la grammaire trouvée
- Q3 (3 pts) –
  a. Ecrire les procédure d'analyse LL(1) pour le langage des expressions LSET. On utilisera la fonction next_token : $\varnothing \rightarrow$ Token est la fonction qui lit le prochain mot et retourne un token de l'ensemble {':',';',',',affect,idf,set,interval,begin,end,deux_points,'(',')','∪','∩', '–'} où deux_points est l'unité lexicale associée aux .., et affect est l'unité lexicale associée à ":="

Analyseur syntaxique ascendant (bottom-up) utile pour comprendre bison

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Analyseurs *descendant* vs *ascendant*

	dérivation	**chaînage**	**algorithme**	**grammaires couvertes**
Analyseur descendant	gauche	Avant « *dérivation* »	Récursif descendant, LL(k>1)	Non ambiguës, Non récursive gauche
Analyseur ascendant	droite	Arrière « *réduction* »	Shift/Reduce, SLR, LALR, LR	Non ambiguës, Non récursive droite ?

Analyseur syntaxique ascendant (bottom-up) – Principe

- ***Principe***
  - *S'il existe un arbre de dérivation d'une expression à partir du* start *S*
    - *(i.e* $S=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n$*=expression)*
  - Alors, l'analyseur ascendant (bottom-up)
    - découvrira la dérivation $\gamma_{n-1} \Rightarrow \gamma_n$
    - ajoutera les noeuds impliqués par la dérivation $\gamma_{n-1} \Rightarrow \gamma_n$
    - Puis, découvrira $\gamma_{n-2} \Rightarrow \gamma_{n-1}$.
    - Jusqu'à atteindre le start *S : **Chaînage arrière (Backward) avec dérivation à droite (right-most derivation)***

- **Comment**
  - L'analyseur ascendant remonte les règles aulieu de les dériver comme l'analyseur descendant. La dérivation est à droite au lieu d'être à gauche pour les analyseurs descendants.

- **Avantages**
  - En général, les analyseurs syntaxiques ascendants sont plus puissants que les analyseurs descendants : au moins par ce que les premiers prennent en compte un ensemble de grammaires HC plus large que l'ensemble des grammaires LL.
  - Cependant, et cela n'est pas surprenant, la construction logicielle requise est proportionnellement plus complexe.

L'analyse ascendante ***shift-reduce : récursif ascendant***

L'algorithme *Shift-Reduce (récursif ascendant) :* Concepts et opérations

- Concepts :
  - La **frontière haute** (ou *upper frontier*) : L'analyse commence par construire des feuilles pour chacun des tokens et étend l'arbre par les feuilles jusqu'à arriver à la racine. la **frontière haute** désigne le plus haut sommet de l'arbre partiel (construit pendant l'analyse).
  - **Handle** : une règle $\alpha \rightarrow \beta$ telle que β est en tête de pile
  - **Partie semi-digérée** (semi-digested) : partie du flux des tokens empilés dans la pile
  - **Partie non digérée** (undigested) : partie du flux des tokens à analyser dans le futur

- Opération
  - **Réduction** (**REDUCE**) : remonter la dérivation du handle en chaînage arrière
  - **Décalage** (**SHIFT** ou extends) : Quand la frontière haute ne contient aucun « handle », l'analyseur repousse (décale) la frontière en ajoutant un token à droite de la frontière

L'algorithme *Shift-Reduce (récursif ascendant)* (décalage-réduction)

- ***push*** INVALID
- *token ← next_token()*
- *repeat until (top of stack = Start && Token = EOF)*
  - *if (we have a handle α → β on top of the stack) then*
    - ***REDUCE*** *by α→β // remonter la dérivation en chaînage arrière*
    - ***pop*** *β symbols off the stack ;* ***push*** *α onto the stack*
  - *else if (Token ≠* EOF*) then*
    - ***SHIFT*** *// décaler la tête de lecture de gauche à droite*
    - ***push*** *token ; token ← next_token()*
  - *else report syntax error & halt*
- *end repeat*
  - *Il y a autant de shifts que de nombre de tokens dans le flux d'entrée (évident) (notant ce nombre s)*
  - *Il y a autant de réductions que de règles appliquées en chaînage arrière (évident) (notant ce nombre r)*
  - ***Problème de non déterminisme et inneficacité*** *: L'algorithme effectue |reduce|+|shift| = r+s recherche de règle (handle).*
    - *i.e. A chaque shift et à chaque réduction, on cherche une règle α → β correspondante (handle)*
    - *Ceci est similaire au comportement de l'analyseur descendant récursif*

Exercice préliminaire

- Soit la grammaire

1.	*S*	$\rightarrow$	*Expr*
2.	*Expr*	$\rightarrow$	*Expr + Term*
3.	*Expr*	$\rightarrow$	*Expr – Term*
4.	*Expr*	$\rightarrow$	*Term*
5.	*Term*	$\rightarrow$	*Term* $\times$ *Factor*
6.	*Term*	$\rightarrow$	*Term* $\div$ *Factor*
7.	*Term*	$\rightarrow$	*Factor*
8.	*Factor*	$\rightarrow$	*(Expr)*
9.	*Factor*	$\rightarrow$	*Num*
10.	*Factor*	$\rightarrow$	*Id*

1. Donner le flux des tokens de l'expression x – 2 * y
2. Donner sa dérivation droite

Solution

- Le flux des tokens de x – 2 * y est Id – Num * Id

- **La dérivation droite de l'expression est :**

S $\Rightarrow$ **Expr**
$\Rightarrow$ Expr – **Term**
$\Rightarrow$ Expr – Term $\times$ **Factor**
$\Rightarrow$ Expr – **Term** $\times$ Id
$\Rightarrow$ Expr – **Factor** $\times$ Id
$\Rightarrow$ **Expr** – Num $\times$ Id
$\Rightarrow$ **Term** – Num $\times$ Id
$\Rightarrow$ **Factor** – Num $\times$ Id
$\Rightarrow$ Id – Num $\times$ Id

Suite (chaînage arrière - backward)

- Remontons cette dérivation résultante en **chaînage arrière**
  - Id – Num × Id
    - ⇐ **Factor** – Num × Id
    - ⇐ **Term** – Num × Id
    - ⇐ **Expr** – Num × Id
    - ⇐ Expr – **Factor** × Id
    - ⇐ Expr – **Term** × Id
    - ⇐ Expr – Term × **Factor**
    - ⇐ Expr – **Term**
    - ⇐ **Expr**
    - ⇐ **S**

L'algorithme *Shift-Reduce : illustration*

l'analyseur lexical renvoie les tokens **x – 2 * y**

Id – Num * Id

ordre des shifts (1) (2) (3) (4) (5)

Id – Num × Id
⇐ Id – Num × Id
⇐ Factor – Num × Id
⇐ Term – Num × Id
⇐ Expr – Num × Id
⇐ Expr – Num × Id
⇐ Expr – Num × Id
⇐ Expr – Factor × Id
⇐ Expr – Term × Id
⇐ Expr – Term × Id
⇐ Expr – Term × Id
⇐ Expr – Term × Factor
⇐ Expr – Term
⇐ Expr
⇐ S

reduce

x shift Partie non digérée des tokens

	Token	Frontier **(tête de pile)**	Handle **(règle)**	Action
1. *push*	Id		— none —	***shift*** (1)
2.	–	Id	⟨Factor→Id,1⟩ (*push* *pop*)	*reduce*
3.	–	Factor	⟨Term→Factor,1⟩ (*push* *pop*)	*reduce*
4.	–	Term	⟨Expr→Term⟩ (*push* *pop*)	*reduce*
5 *push*	–	Expr	— none —	***shift*** (2)
6 *push*	Num	Expr –	— none —	***shift*** (3)
7.	×	Expr – Num	⟨Factor→Num,3⟩ (*push* *pop*)	*reduce*
8.	×	Expr – Factor	⟨Term→Factor⟩ (*push* *pop*)	*reduce*
9 *push*	×	Expr – Term	— none —	***shift*** (4)
10 *push*	Id	Expr – Term ×	— none —	***shift*** (5)
11.	EOF	Expr – Term × Id	⟨Factor→Id⟩ (*push* *pop*)	*reduce*
12.	EOF	Expr – Term × Factor	⟨Term→Term × Factor⟩ (*push* *pop*)	*reduce*
13.	EOF	Expr – Term	⟨Expr→Expr – Term⟩ (*push* *pop*)	*reduce*
14.	EOF	Expr	⟨ S →Expr⟩ (*push* *pop*)	*reduce*
15.	EOF	S	— none —	*accept*

Table-Driven parsing : analyse dirigée par une table

2. Id

3. *Factor* → Id

4. *Term* → *Factor* → Id

5. *Expr* → *Term* → *Factor* → Id

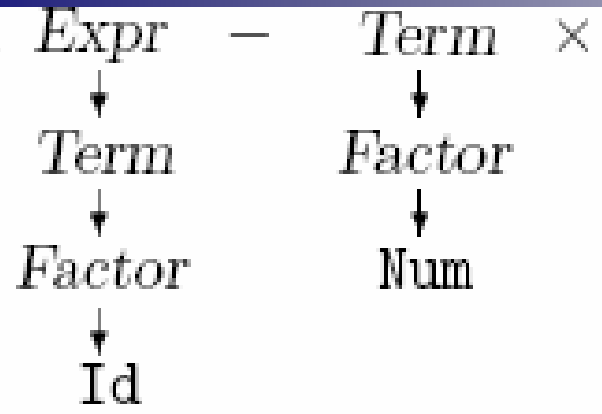
6. *Expr* → *Term* → *Factor* → Id −

7. *Expr* → *Term* → *Factor* → Id − Num

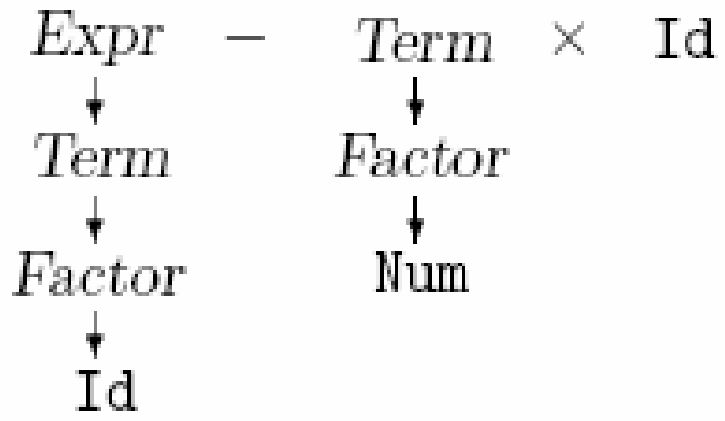
8. *Expr* → *Term* → *Factor* → Id − *Factor* → Num

9. *Expr* → *Term* → *Factor* → Id − *Term* → *Factor* → Num

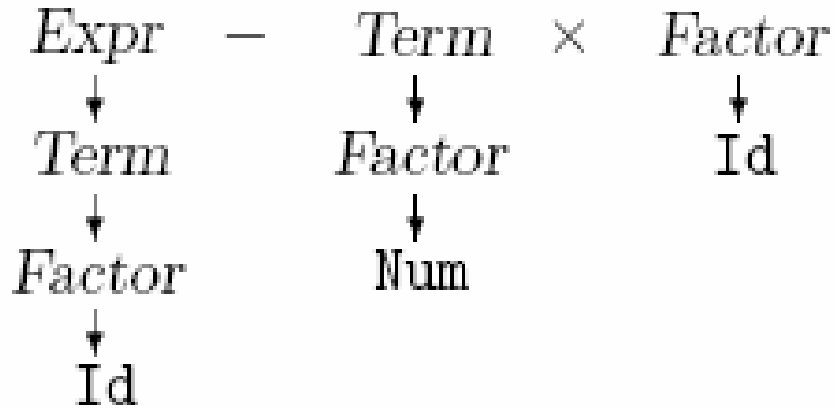
10.



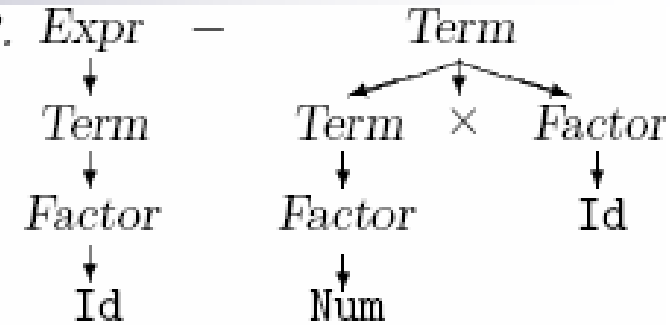
11.



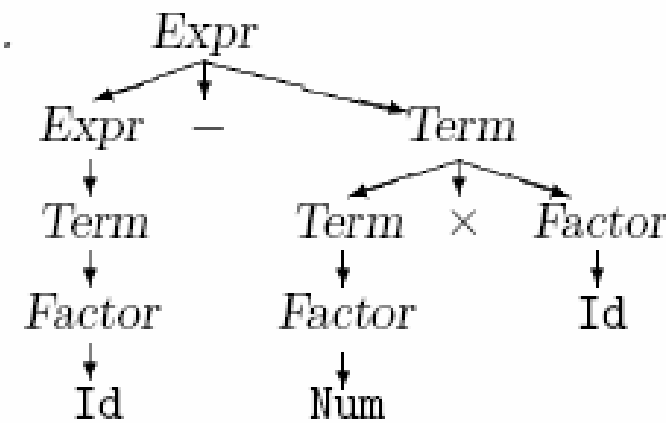
12.



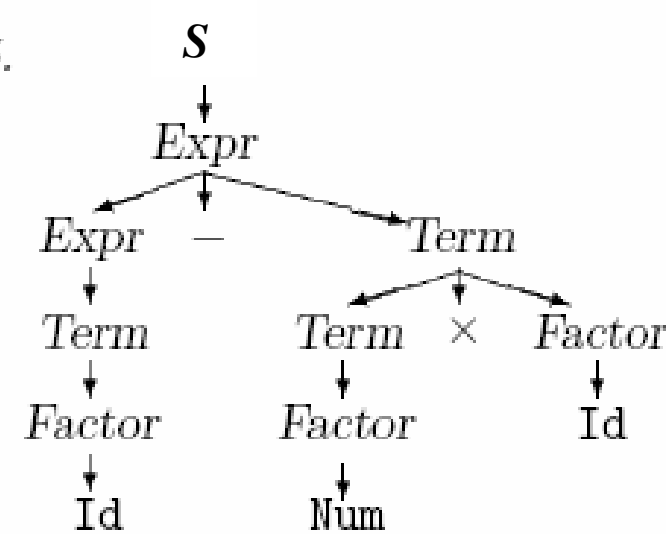
13.



14.



15.



Problème de non déterminisme et inefficacité de Shift-Reduce (récursif ascendant)

- *L'algorithme Shift-Reduce effectue |reduce|+|shift| = r+s recherche de règle (handle).*
  - *i.e. A chaque shift et à chaque réduction, on cherche une règle α → β correspondante (handle)*

- *De la même manière que la famille LL était la solution de l'inefficacité de l'analyseur descendant récursif. La famille LR est la solution à l'inefficacité de l'analyseur ascendant shift-reduce.*

L'analyse ascendante : Famille ***LR***

Prof. K. Baïna 2010 ©, 7ème édition

Variantes Analyseur Ascendant (bottom-up)

- Les analyseurs LR représentent des familles d'algorithmes de parsers largement utilisés. Leur traitement sont mieux gérés par la machine que par l'humain. Mais l'étude de ces algorithmes permet de
  - comprendre les erreurs rencontrées par les générateurs de parsers
  - Comment ces erreurs se produisent
  - Comment ces erreurs peuvent être corrigées

La position d'analyse •

- • placé au coeur d'une règle de production
  - Signifie que l'on a accepté ce qui précède le point dans le corps et que l'on est prêt à accepter ce qui suit le point
  - C'est une tête de lecture qui va classifier l'ensemble P en un ensemble de classe d'équivalence selon un point de vue temporel (analogie *logique temporelle*)
  - Exemple :
    - Expr ⇒ Expr • + terme
    - signifie que l'on est en train d'accepter une expression : on a déjà accepté une sous expression et on est prêt à accepter le terminal +, suivi d'un terme

Analyse SLR

- Une grammaire est dite SLR **si on peut construire une table** d'analyse par l'algorithme spécifié dans ce qui suit

- L'idée centrale de la méthode SLR est, étant donnée une **position** d'analyse •, est d'obtenir par **fermeture transitive** toutes les **possibilités de continuer** l'analyse du texte source, en tenant compte de **toutes** les productions de la grammaire

Définition : La fonction Follow (*look-ahead*)

- Soit N $\in$ NT, **Follow**(N) = {ensemble des symboles terminaux qui peuvent apparaître après N}
- Remarque : si M $\rightarrow N \in$ P, alors Follow(N) $\supset$ Follow(M)
- Les tokens successeurs de M sont également successeurs de N. L'inverse n'est pas vrai.

  - Exemple :
    - ***S*** $\rightarrow E\ \$$
    - ***E*** $\rightarrow E + T$
    - ***E*** $\rightarrow T$
    - ***T*** $\rightarrow T * F$
    - ***T*** $\rightarrow F$
    - ***F*** $\rightarrow i$

Follow(E) = { + $ }
Follow(T) = { * + $ }
Follow(F) = { * + $ }

Follow(F) $\supset$ Follow(T) $\supset$ Follow(E)

Analyse SLR

- *Soit la grammaire :*
  1. *Expr* → *Expr + Term*
  2. *Expr* → *Term*
  3. *Term* → *Term* × *Factor*
  4. *Term* → *Factor*
  5. *Factor* → *(Expr)*
  6. *Factor* → *Id*

- On ajoute un non terminal
  1. *S* → *Expr*

Follow(Expr) = { +) }
Follow(Term) = { × +) }
Follow(Factor) = { × +) }

Analyse SLR

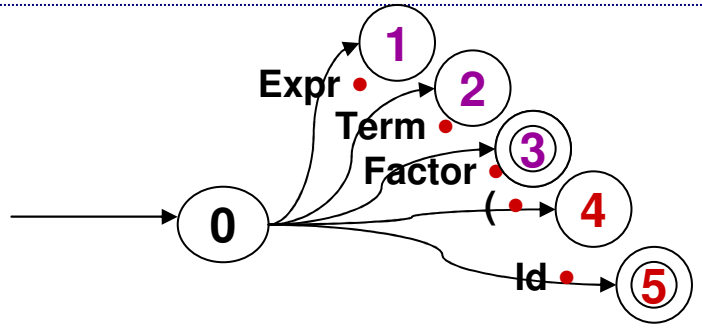
- Au début de l'analyse
  1. *S* → • *Expr*
  - On n'a rien consommé et on est prêt à consommer une expression

- État 0
  1. *R0 : S* → • *Expr*
  2. *R1: Expr* → • *Expr + Term*
  3. *R2 : Expr* → • *Term*
  4. *R3 : Term* → • *Term × Factor*
  5. *R4 : Term* → • *Factor*
  6. *R5 : Factor* → • *(Expr)*
  7. *R6 : Factor* → • *Id*

Analyse SLR

- Au début de l'analyse
  1. *S* → • *Expr*
- Donc, depuis l'état 0 on peut consommer (1) une expression et par fermeture transitive : (2) un terme, ou (3) un facteur, ou (4) une parenthèse, ou (5) un identificateur

1.	*S*	→	*Expr* •	**État 1 (Expr)**
2.	*Expr*	→	*Expr* • + *Term*	
3.	*Expr*	→	*Term* •	**État 2 (Term)**
4.	*Term*	→	*Term* • × *Factor*	
5.	*Term*	→	*Factor* •	**État 3 (Factor)**
6.	*Factor*	→	(• *Expr*)	**État 4 ('(')**
7.	*Factor*	→	*Id* •	**État 5 (Id)**

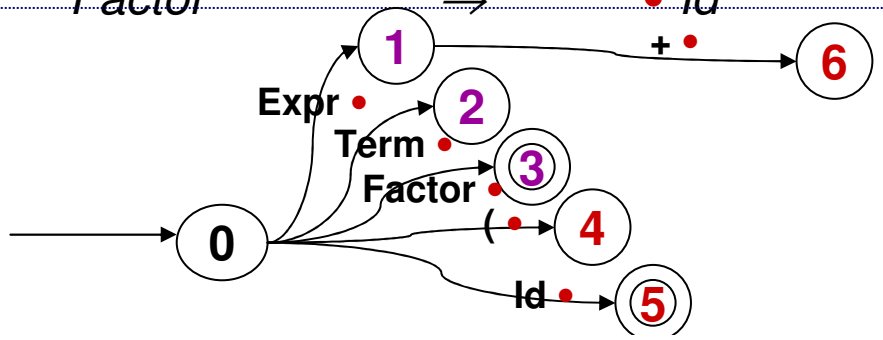


Analyse SLR

- A l'état 1
  1. *S* → *Expr* •
  2. *Expr* → *Expr* • + *Term*
- Donc, depuis l'état 1 on peut consommer (1) un "+" et on se retrouvera au début d'un terme (• *Term)*, donc, par fermeture transitive : devant un facteur (• *Factor)*, devant une parenthèse (• "*(*"), ou devant un identificateur (• *Id*)

1.	*Expr*	→	*Expr* + • *Term*
2.	*Term*	→	• *Term* × *Factor*
3.	*Term*	→	• *Factor*
4.	*Factor*	→	• (*Expr*)
5.	*Factor*	→	• *Id*

État 6



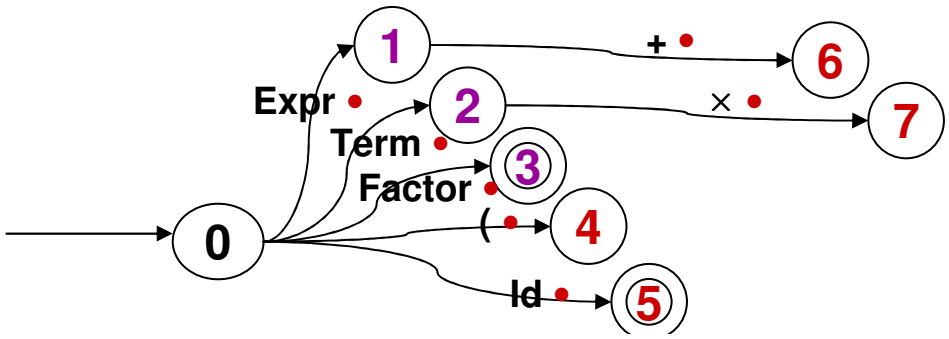
Analyse SLR

- A l'état 2

1.	*Expr*	→	*Term* •
2.	*Term*	→	*Term* • × *Factor*

- Donc, depuis l'état 2 on peut consommer (1) un "×" et on se retrouvera au début d'un facteur (• *Factor*), donc, par fermeture transitive : devant une parenthèse (• "*(*"), ou devant un identificateur *(• Id)*

1.	*Term*	→	*Term* × • *Factor*	
2.	*Factor*	→	• *(Expr)*	**État 7**
3.	*Factor*	→	• *Id*	

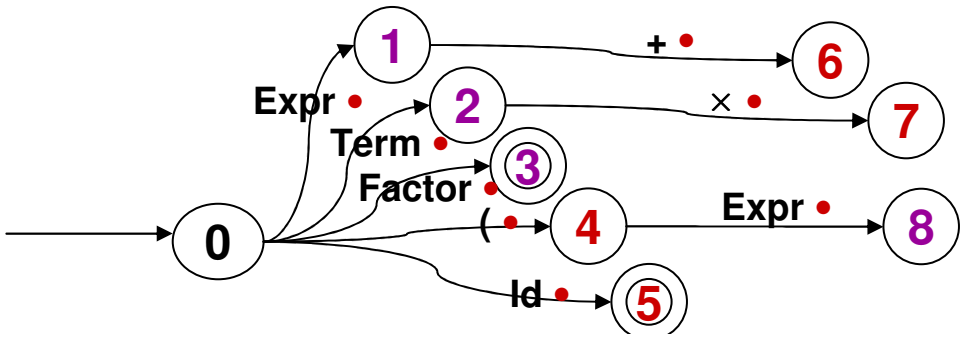


Analyse SLR

- A l'état 4
  1. *Factor* → *(• Expr)*
- Donc, depuis l'état 4 on peut consommer une expression, donc, on se retrouvera devant une parenthèse (• "*)*"), ou par fermeture transitive devant un "+" *(• +)*

1.	*Factor*	→	*(Expr •)*	**État 8**
2.	*Expr*	→	*Expr • + Term*	

Follow(Expr) ={), +}
On prend les règles qui correspondent aux follows directs



Analyse SLR

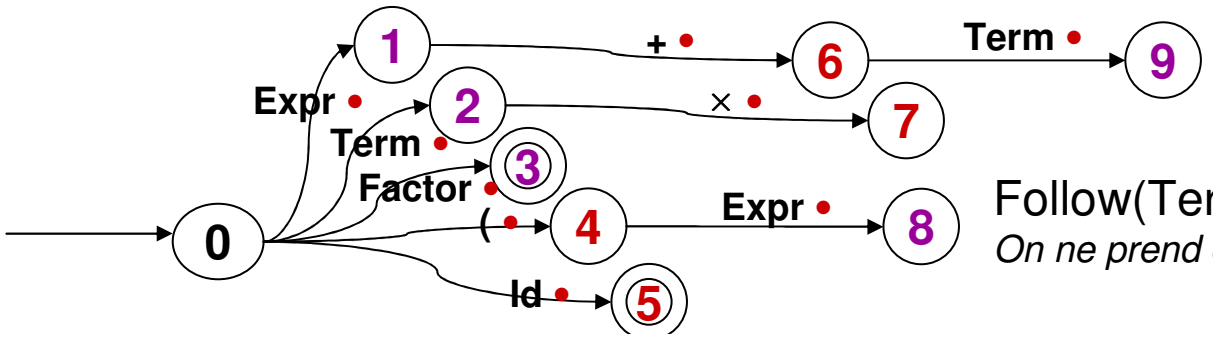
- A l'état 6

1.	*Expr*	→	*Expr* + • *Term*
2.	*Term*	→	• *Term* × *Factor*
3.	*Term*	→	• *Factor*
4.	*Factor*	→	• (*Expr*)
5.	*Factor*	→	• *Id*

- Donc, depuis l'état 6 on peut consommer un terme, donc, on se retrouvera devant une parenthèse (• "*)*"), ou par fermeture transitive devant un "+" *(• +)*

1.	*Expr*	→	*Expr* + *Term* •
2.	*Term*	→	*Term* • × *Factor*

État 9



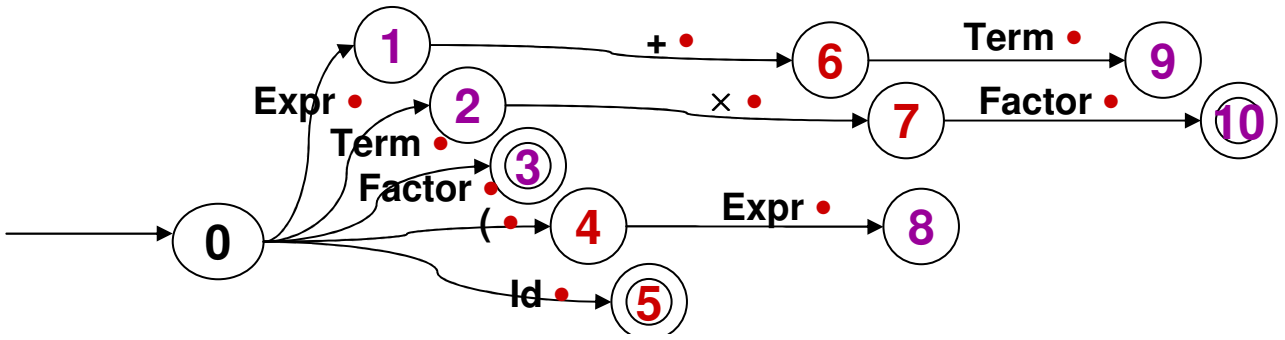
Follow(Term) ={), +, ×}

On ne prend que les follows directs

Analyse SLR

- A l'état 7
  1. *Term* → *Term* × • *Factor*
  2. *Factor* → • (*Expr*)
  3. *Factor* → • *Id*
- Donc, depuis l'état 7 on peut consommer un facteur, qui ne possède pas de follow directs, donc pas de nouvelles règles.

1. *Term* → *Term* × *Factor* • **État 10**



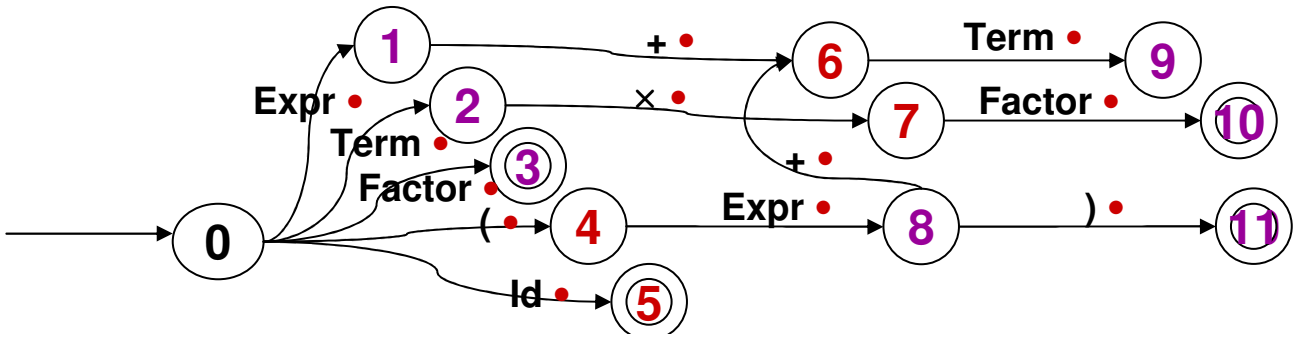
Analyse SLR

- A l'état 8

1.	*Factor*	→	*(Expr •)*
2.	*Expr*	→	*Expr • + Term*

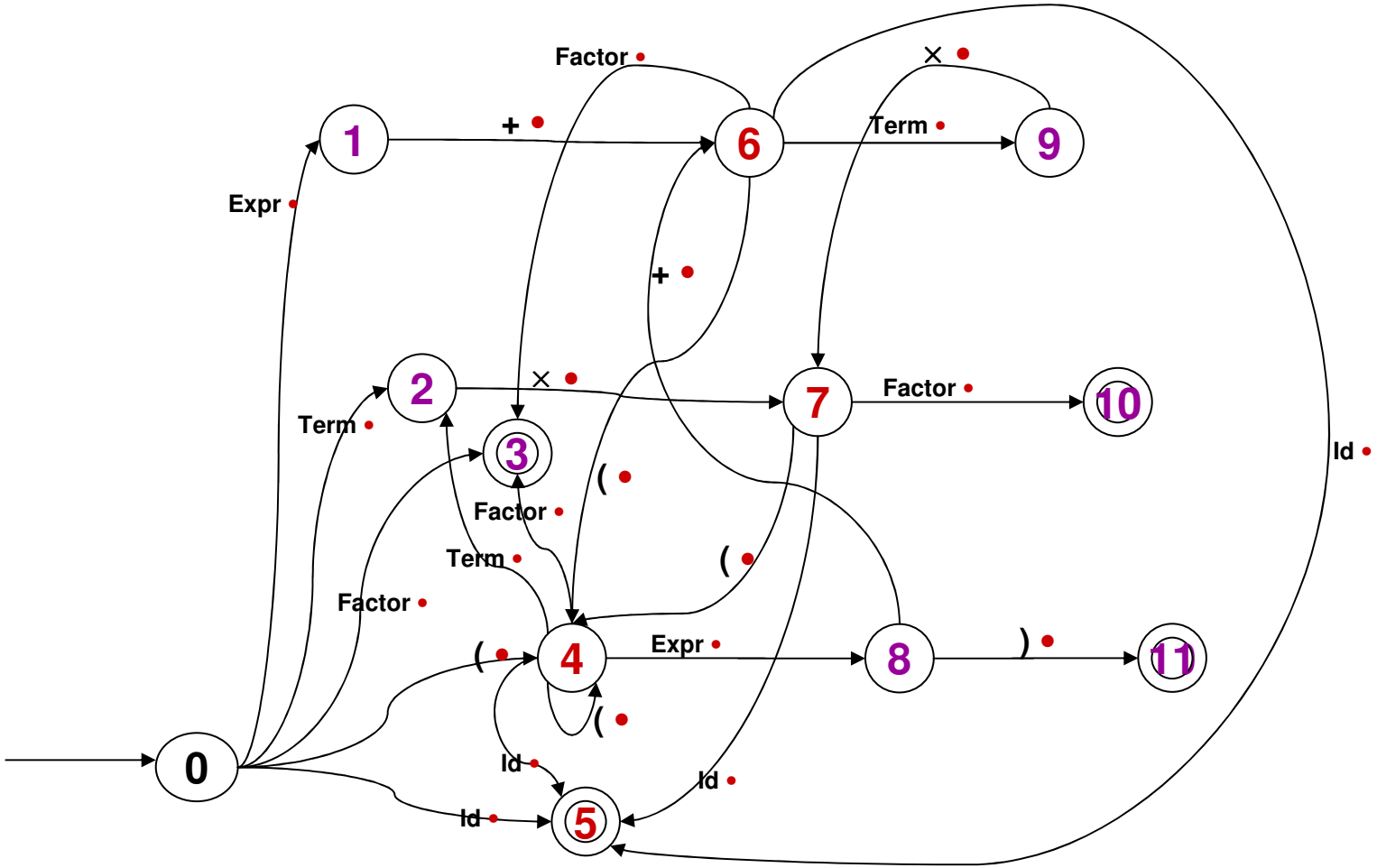
- Donc, depuis l'état 7 on peut consommer une "*)*" ( ) •)
- Ou consommer "+" (+ •) et se trouver devant un terme

1.	*Factor*	→	*(Expr) •*	**État 11**
2.	*Expr*	→	*Expr + • Term*	
3.	*Term*	→	*• Term × Factor*	
4.	*Term*	→	*• Factor*	**État 6**
5.	*Factor*	→	*• (Expr)*	
6.	*Factor*	→	*• Id*	



Analyse SLR

- Après plusieurs itérations



Parser SLR : 4 opérations (SHIFT, REDUCE, ACCEPT, ERROR) et 2 tables ACTION et GOTO

- Si l'état **état_i** contient une position d'analyse de la forme:
  notion ⇒ préfixe ● **terminal** suffixe
- et que: goto (état_i, terminal) = état_destination
  - alors on choisit:
  - **ACTION** (état_i, terminal) = **SHIFT** état_destination

- Si l'état état_i contient une position d'analyse:
  notion ⇒ **corps** ● où notion n'est pas S,
  - alors pour tout terminal de Follow(notion) on fixe:
  - **ACTION**(état_i, terminal) = **REDUCE** 'notion ⇒ corps'

- Si l'état **état_i** contient la position d'analyse :
  S ⇒ **axiome** ●
  - alors on choisit :
  - **ACTION**(état_i, **FIN**) = **ACCEPT (dernier REDUCE)**

- Toutes les entrées de la table **action** qui n'ont **pas été garnies** par les quatre considérations ci-dessus sont marquées par:
  - **ACTION**(état_i, terminal_i) = **ERROR**

- on **garde** le contenu de la table **GOTO** pour toutes les entrées dont le second argument est une notion **non terminale**

La table ACTION (états × ∑=T → états)

Etat	IDENT	+	*	(	)	FIN
0	S 5			S 4		
1		S 6				ACC
2		R 2	S 7		R 2	R 2
3		R 4	R 4		R 4	R 4
4	S 5			S 4		
5		R 6	R 6		R 6	R 6
6	S 5			S 4		
7	S 5			S 4		
8		S 6			S 11	
9		R 1	S 7		R 1	R 1
10		R 3	R 3		R 3	R 3
11		R 5	R 5		R 5	R 5

La table GOTO (états $\times \sum =$ NT$\rightarrow$ états)

Etat	EXPRESSION	TERME	FACTEUR
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			

Réecriture de l'algorithme shift-reduce avec les table ACTION et GOTO

- push(0);
- read_next_token();
- **while** (true) {
  - s = top(); /* current state is taken from top of stack */
  - if (ACTION[s,current_token] == 'si') /* **shift** and go to state i */ {
    - push(i);
    - *read_next_token*();
  - }else if (ACTION[s,current_token] == 'ri')
    /* **reduce** by rule i: X ::= A1...An */ {
    - perform pop() n times;
    - s = top(); /* restore state before reduction from top of stack */
      push(GOTO[s,X]); /* state after reduction */
  - }else if (ACTION[s,current_token] == 'a')
    - **success** !! //**accept**
    - exit ;
  - else
    - **error**();
    - exit ;
- }

Limites de l'analyse SLR

- Une grammaire est dite non SLR **si l'on ne peut pas construire les tables** d'analyse **ACTION et GOTO** par l'algorithme spécifié précédemment.
  - Du fait que la construction de la table **ACTION** peut conduire à des conflits parce qu'on devrait mettre plus d'une valeur dans certaines de ses entrées. Deux problèmes existent
    - **shift/reduce**
    - **reduce/reduce**

Types de conflits LR

i est ici un terminal, mais il aurait pu être un NT

- **shift/reduce**
  - Le parser n'arrive pas à décider entre un ***shift*** et un ***reduce***
    - ***S*** → *E $*
    - ***E*** → *E + T*
    - ***E*** → *T*
    - ***T*** → *(E)*
    - ***T*** → *i*
    - ***T*** → *i [E]*

T→ i • *reduce*
T→ i • [E] *shift*

Follow(E) = { $ +)] }
Follow(T) = { $ +)] }

  - Dans un état donné on peut aussi bien faire un *shift* (et continuer à consommer le token suivant) qu'un *reduce (*et remonter la règle courante et la réduire complètement à son NT à gauche*)*.
  - Le générateur de parsers privilégie de sélectionner la réduction aulieu d'effectuer un shift, mais cela peut engendrer un comportement illicite
  - L'origine du conflit peut être une ambiguité (mais ce n'est pas nécessaire)
    - ex. la grammaire if-the-else
    - r1: **Stmt** → if (**Expr**) then **Stmt** • else **Stmt** ***shift***
    - r2: | if (**Expr**) then **Stmt** • ***reduce***

Types de conflits LR

- **reduce/reduce**
  - La grammaire contient deux règles de production qui possèdent la même partie droite, deux parties gauches différentes, et les deux sont applicables (deux réductions possibles)
  - Dans un état donné, deux réductions sont possibles. Une position d'analyse • est à la fin de deux règles du même état.
  - Ce conflit est plus sévère que le conflit shift/reduce
    - ***S*** $\rightarrow$ *A a A b*
    - ***S*** $\rightarrow$ *B b B a*
    - ***A*** $\rightarrow \varepsilon$
    - ***B*** $\rightarrow \varepsilon$

A $\rightarrow \varepsilon$ • *reduce*

B $\rightarrow \varepsilon$ • *reduce*

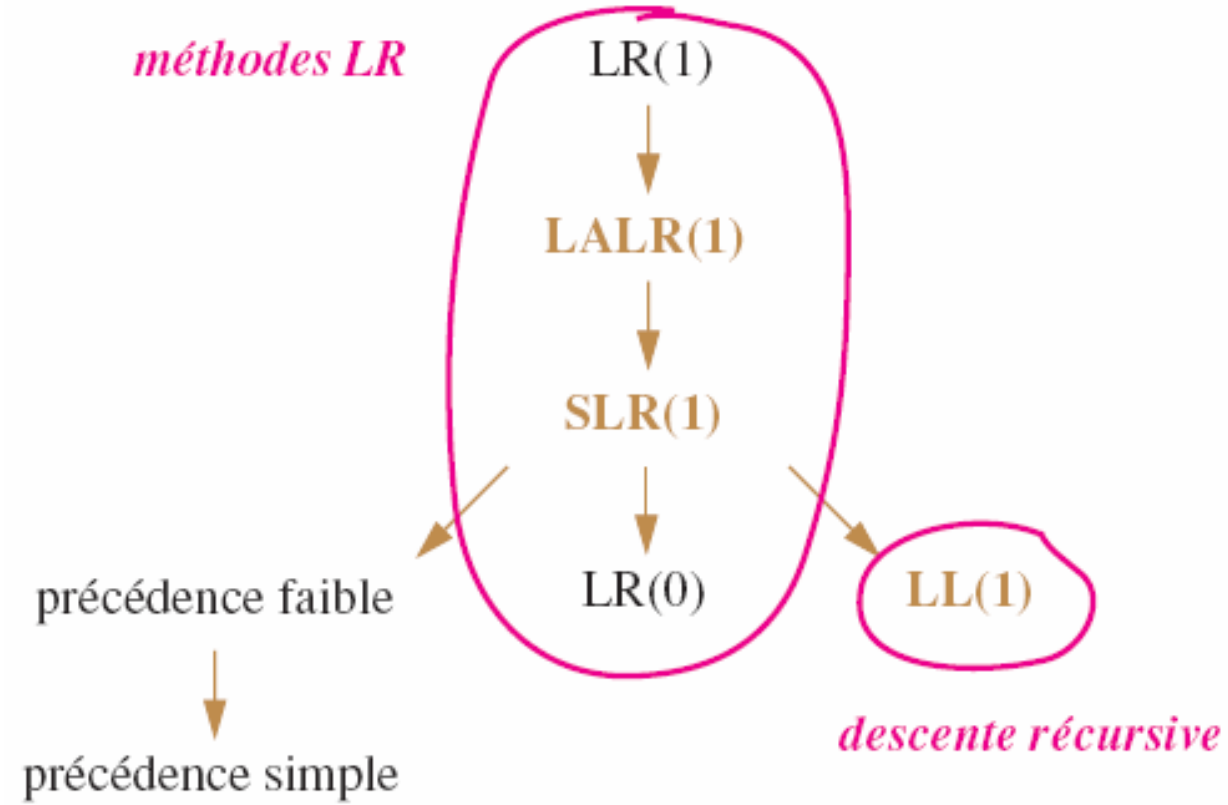
Familles des grammaires LR

	Nom	Propriétés	Caractéristique
LR(0)		***Pas besoin de look-ahead*** (un token en plus du token courant pour décider les conflits réduire ou shift)	Le plus faible des parsers vu la taille de la classe de grammaires pouvant traiter
SLR(1)	Simple LR	***Avec besoin d'un seul token comme look-ahead*** Couvre moins de langages, que LR mais, avec une table d'analyse moins volumineuse	La classe des grammaires LR ayant besoin d'un look-ahead d'un token pour être analysées
LALR(1)	Look-Ahead LR	***Version simplifiée de LR(1).*** Couvre beaucoup de langages, avec la même taille de table d'analyse SLR(1)	La classe des grammaires LR ayant besoin de plus d'un look-ahead. Classe Populaire. Bon compromis entre taille de lacasse des grammaires traitées et taille de la table d'analyse. Presque aussi puissante que LR(1). A la base de Bison.
LR(1)	**L**eft-to-right with **R**ightmost parse	***Avec prédiction look-ahead de plus d'un token.*** Couvre beaucoup de langages, mais, la table d'analyse très volumineuse	Canonique. Non utilisée en pratique à cause de la taille de sa table.

+ faible

+ fort

- SLR est la plus petite classe des grammaires LR. Les autres grammaires non SLR nécessitent d'autres types d'analyses. LR(0) $\subset$ SLR(1) $\subset$ LALR(1) $\subset$ LR(1)



- SLR est la plus petite classe des grammaires LR. Les autres grammaires non SLR nécessitent d'autres types d'analyses. LR(0) ⊂ SLR(1) ⊂ LALR(1) ⊂ LR(1)

contient l'ensemble des langages du type

Décision LR(0), SLR(1) ou LALR(1)

- S'il n'y a ni un conflit reduce/reduce ni un conflit shift/reduce, alors
  - la grammaire est LR(0)
- Sinon, si on a besoin d'un seul token look-ahead (Follow) pour éviter les conflits, alors
  - la grammaire est SLR(1)
- Sinon, si on a besoin de plus d'un token de look-ahead (Follow), alors
  - la grammaire est LALR(1)

Meilleurs pratiques

- Pour réduire la taille des tables ACTION et donc améliorer l'efficacité du parser ascendant
  - Réduire le nombre de terminaux sans ambiguité

Ancienne grammaire

S	→	*Expr*
Expr	→	*Expr + Term*
Expr	→	*Expr – Term*
Expr	→	*Term*
Term	→	*Term × Factor*
Term	→	*Term ÷ Factor*
Term	→	*Factor*
Factor	→	*(Expr)*
Factor	→	*Num*
Factor	→	*Id*

Nouvelle grammaire

S	→	*Expr*
Expr	→	*Expr* ***AddSub*** *Term*
Expr	→	*Term*
Term	→	*Term* ***MultDiv*** *Factor*
Term	→	*Factor*
Factor	→	*(Expr)*
Factor	→	***Val***

Meilleurs pratiques

- Pour réduire les intersections entre les follows pour éviter les accidents (s/r, r/r) entre les non terminaux
  - Exemple : éliminer les héritages des follows en réduisant les niveaux d'indirection Expr → Term → Factor …..
- Éliminer les ambiguïté pour réduire les conflits shift/reduce

Exercices

Exercice 1

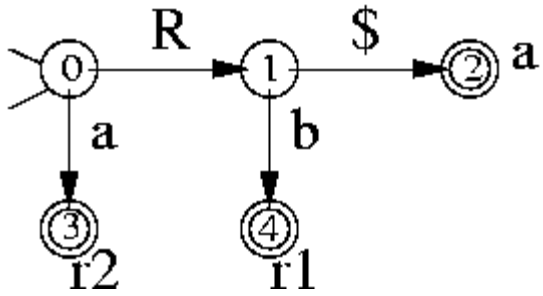
- **r0:** S $\rightarrow$ R $
- **r1:** R $\rightarrow$ R b
- **r2 :** R $\rightarrow$ a

- Construisez l'automate d'analyse de cette grammaire.
- Construisez ses tables ACTIONS et GOTO.
- De quel type est cette grammaire ?

Solution

- **r0:** S → R $
- **r1:** R → R b
- **r2 :** R → a

1. État 0
   1. S → • R $
   2. R → • R b
   3. R → • a
2. GOTO (0,R) = Etat 1
   1. S → R • $
   2. R → R • b
3. GOTO (1,$) = Etat 2
   1. S → R $ • **ACCEPT (a)**
4. GOTO (0,a) = Etat 3
   1. R → a • **REDUCE r2 (r2)**
5. GOTO (1,b) = Etat 4
   1. R → R b • **REDUCE r1 (r1)**



La grammaire est LR(0)

Le SLR(1) optimise les cases car on réduit que pour les tokens de Follow(notion)

state	action			goto	
	a	b	$	S	R
0	s3				1
1		s4	s2		
2	a	a	a		
3	r2	r2	r2		
4	r1	r1	r1		

Exercice 2

S → E $

E → E + T

E → T

T → (E)

T → i

T → i [E]

Shift/Reduce Conflict

T → i •	reduce
T → i • [E]	shift

- Ambiguë ?
- LL(1) ?
- LR(0) ?
- SLR(1) ?

Exercice 2

S → E $

E → E + T

E → T

T → (E)

T → i

T → i [E]

Shift/Reduce Conflict	
T → i •	reduce
T → i • [E]	shift

Follow(E) = { +)] $ }

Follow(T) = { +)] $ }

	i	+	(	)	$	[	]
…							
5							
…	…….						

Exercice 2

S → E $

E → E + T

E → T

T → (E)

T → i

T → i [E]

Shift/Reduce Conflict	
T → i •	reduce
T → i • [E]	shift

Follow(E) = { +)] $ }

Follow(T) = { +)] $ }

	i	+	(	)	$	[	]
…							
5							reduce T → i
…	…….						

Exercice 2

S → E $

E → E + T

E → T

T → (E)

T → i

T → i [E]

Shift/Reduce Conflict

T → i •	reduce
T → i • [E]	shift

Follow(E) = { +)] $ }

Follow(T) = { +)] $ }

	i	+	(	)	$	[	]
…							
5		r		r	r		reduce T → i
…	…….						

Exercice 2

S → E $

E → E + T

E → T

T → (E)

T → i

T → i [E]

Shift/Reduce Conflict	
T → i •	reduce
T → i • [E]	shift

Follow(E) = { +)] $ }

Follow(T) = { +)] $ }

	i	+	(	)	$	[	]
…							
5		r		r	r	s10	reduce T → i
…	…….						

Exercice 2

S → E $

E → E + T

E → T

T → (E)

T → i

T → i [E]

Follow(E) = { +)] $ }

Follow(T) = { +)] $ }

Shift/Reduce Conflict

T → i •	reduce
T → i • [E]	shift

- Ambiguë ?
- LL(1) ?
- LR(0) ?
- SLR(1) ?

non-ambiguë, non-LL(1), non-LR(0), mais SLR(1)

Exercice 3

S → E $

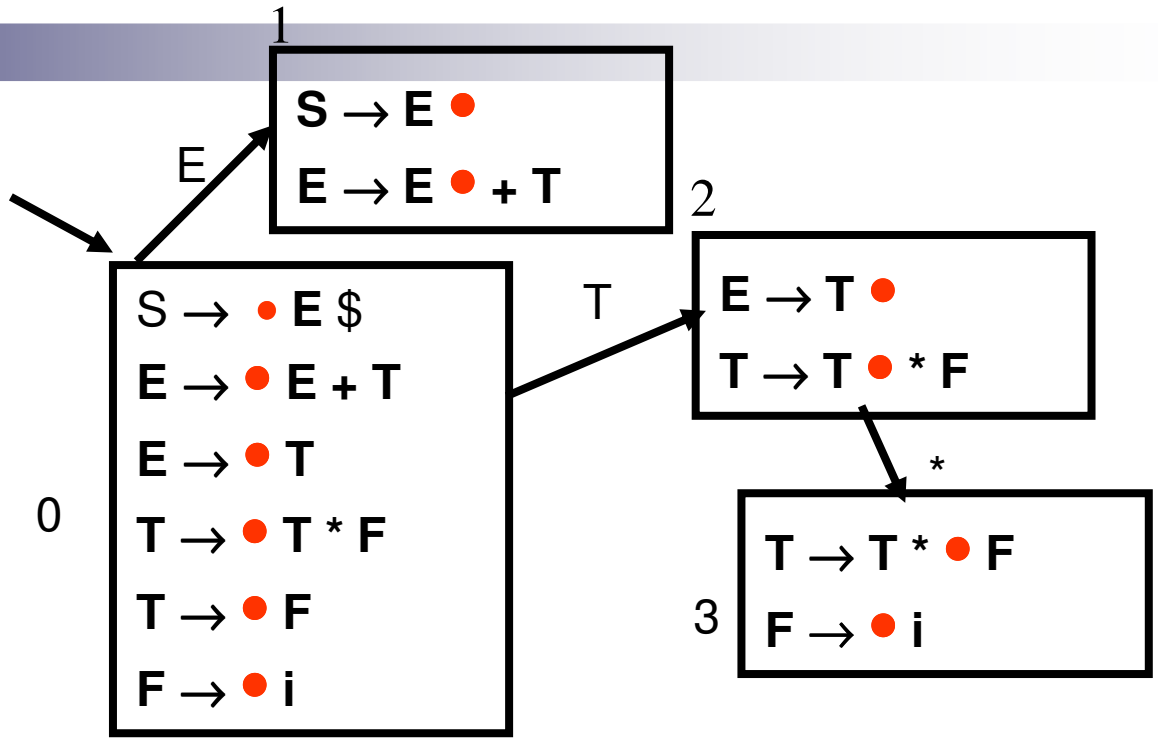
E → E + T

E → T

T → T * F

T → F

F → i



Follow(E) = { + $ }

Follow(T) = { * + $ }

Follow(F) = { + $ }

- Ambiguë ?
- LL(1) ?
- LR(0) ?
- SLR(1) ?

Exercice 3

S → E $

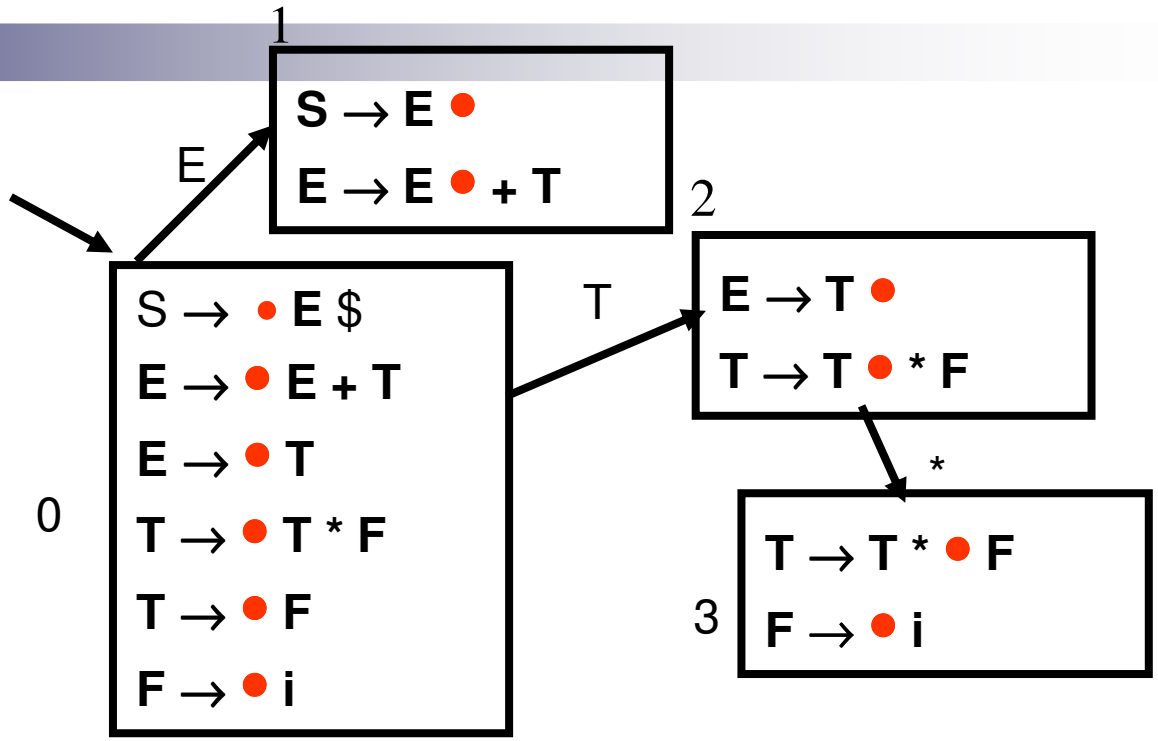
E → E + T

E → T

T → T * F

T → F

F → i



Follow(E) = { + $ }

Follow(T) = { * + $ }

Follow(F) = { * + $ }

	i	+	*	
0	…			
1				
2	…	reduce E→T	s3	
	…			

Exercice 3

S → E $

E → E + T

E → T

T → T * F

T → F

F → i

Follow(E) = { + $ }

Follow(T) = { * + $ }

Follow(F) = { * + $ }

Shift/Reduce Conflict

E → T •	reduce
T → T • * F	shift

- Ambiguë ?
- LL(1) ?
- LR(0) ?
- SLR(1) ?

non-ambiguë, non-LL(1), non-LR(0), mais SLR(1)

Exercice 4

S → A a A b

S → B b B a

A → ε

B → ε

Follow(A) = { a, b }

Follow(B) = { a, b }

Reduce/Reduce Conflict

(1) A → ε • reduce

(2) B → ε • reduce

- Ambiguë ?
- LL(1) ?
- LR(0) ?
- SLR(1) ?

LL(1), mais non SLR(1)

non SLR(1) : car pour décider de réduire par (1) ou (2),
besoin de deux éléments Follow du fait que A et B peuvent être suivis des mêmes éléments

Exercice 5

S → A | x b

A → a A b | B

B → x

Shift/Reduce Conflict

S → x • b shift

B → x • reduce

- Ambiguë ?
- LL(1) ?
- LR(0) ?
- SLR(1) ?

Follow(A) = { b }

Follow(B) = { b }

non LL(1), non SLR(1), mais LALR(1) donc LR(1)

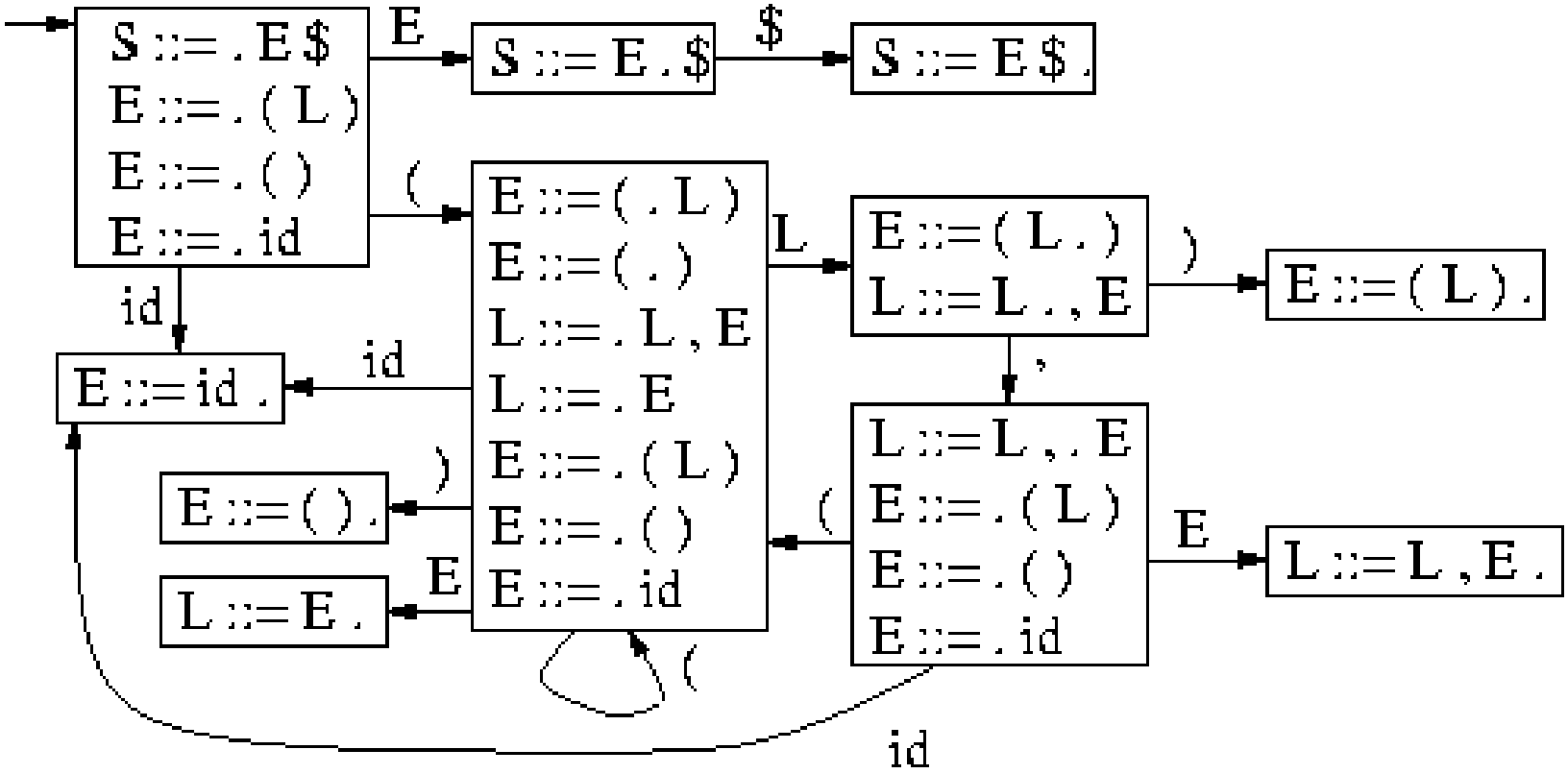
non SLR(1) : car besoin de 2 éléments

Exercice 6

- $S \rightarrow E\ \$$
- $E \rightarrow (\ L\ )$
- $E \rightarrow (\ )$
- $E \rightarrow id$
- $L \rightarrow L\ ,\ E$
- $L \rightarrow E$

- Construisez l'automate d'analyse de cette grammaire.
- Construisez ses tables ACTIONS et GOTO.
- De quel type est cette grammaire ?

Solution


S ::= . E $
E ::= . (L)
E ::= . ( )
E ::= . id
E
S ::= E . $
$
S ::= E $.
(
E ::= (. L)
E ::= (.)
L ::= . L , E
L ::= . E
E ::= . (L)
E ::= . ( )
E ::= . id
L
E ::= (L .)
L ::= L . , E
)
E ::= (L) .
id
E ::= id .
id
,
L ::= L , . E
E ::= . (L)
E ::= . ( )
E ::= . id
E
L ::= L , E .
)
E ::= ( ) .
(
E
L ::= E .
(
id

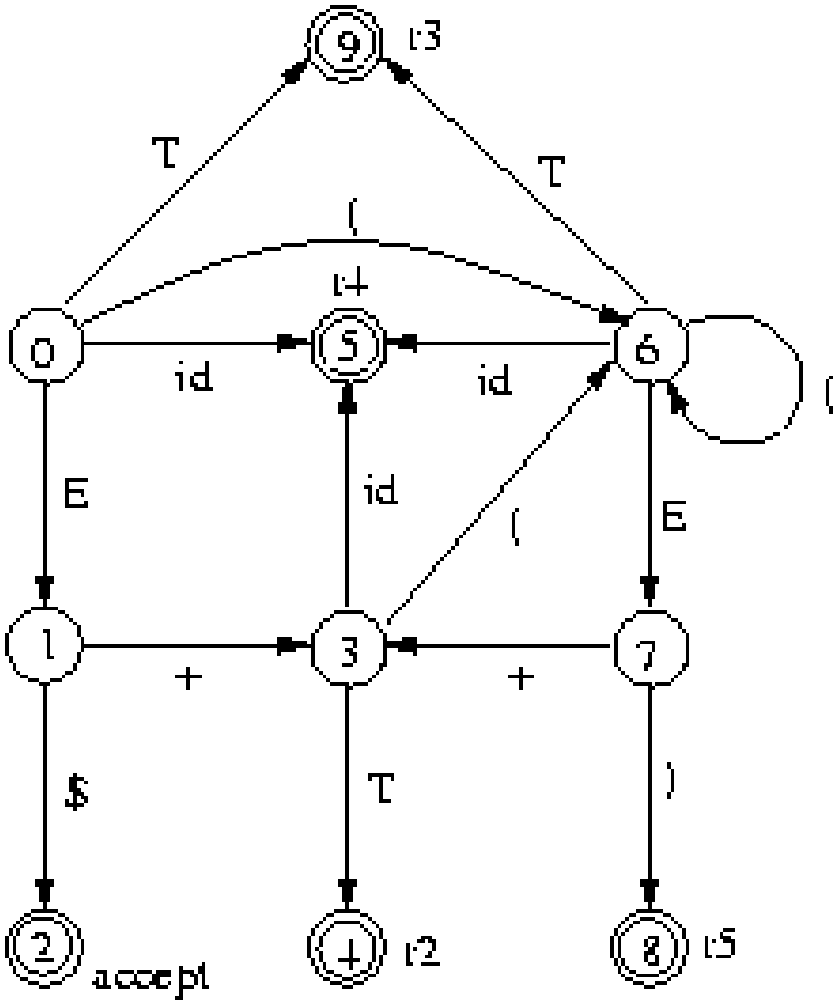
Exercice 7

- Même exercice pour
  - S $\rightarrow$ E $
  - E $\rightarrow$ E + T
  - E $\rightarrow$ T
  - T $\rightarrow$ id
  - T $\rightarrow$ (E)

Solution

- I0:
  - S ::= • E $
  - E ::= • E + T
  - E ::= • T
  - T ::= • Id
  - T ::= • (E)
- I1:
  - S ::= E • $
  - E ::= E • + T
- I2:
  - S ::= E $ •
- I3:
  - E ::= E + • T
  - T ::= • Id
  - T ::= • (E)
- I4:
  - E ::= E + T •
- I5:
  - T ::= id •
- I6:
  - T ::= (• E)
  - E ::= • E + T
  - E ::= • T
  - T ::= • Id
  - T ::= • (E)
- I7:
  - T ::= (E •)
  - E ::= E • + T
- I8:
  - T ::= (E) •
- I9:
  - E ::= T •

Solution



Des concepts à la pratique - Introduction au générateur d'analyseur syntaxique ascendant BISON

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

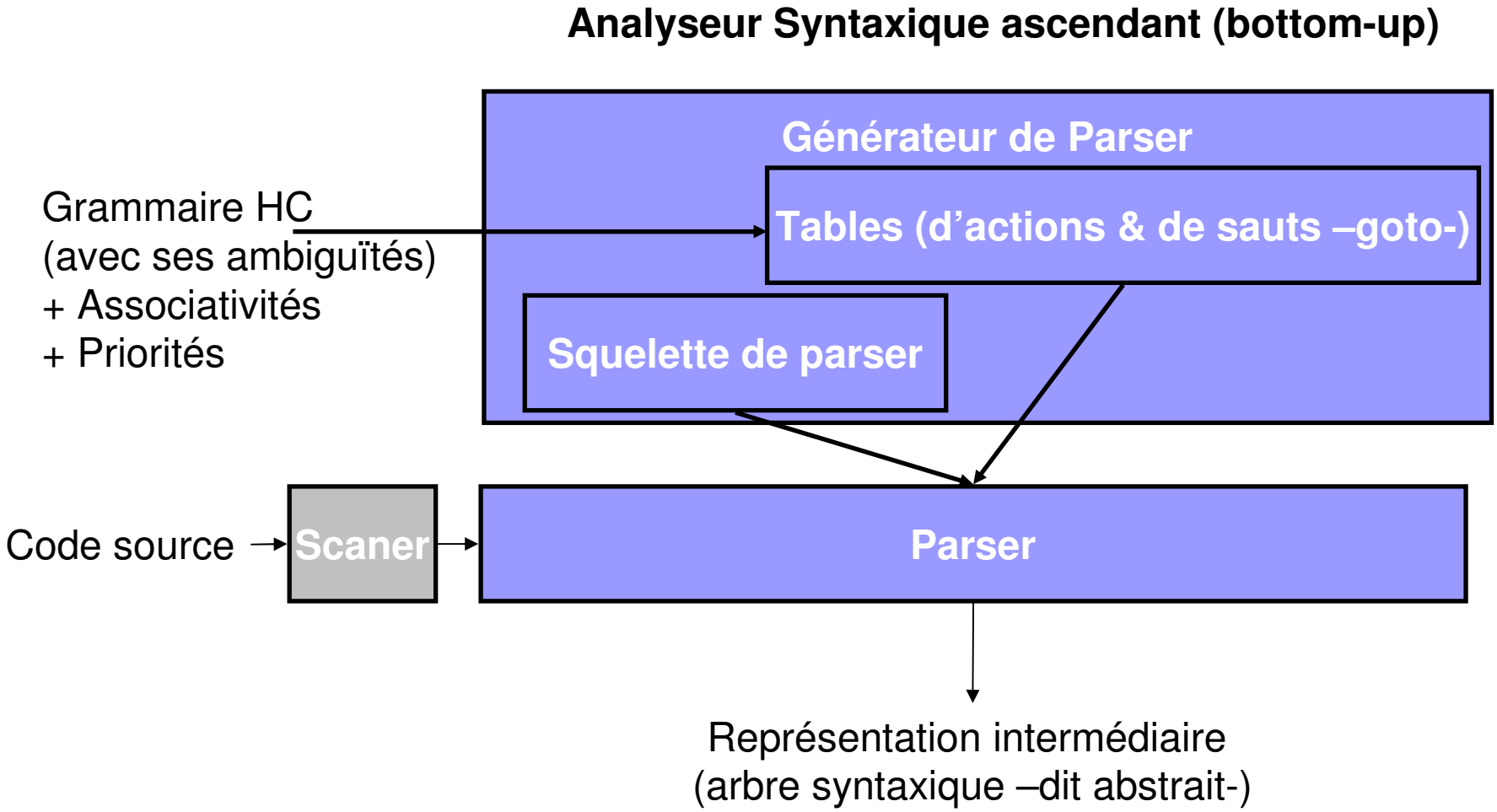
***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Génération automatique de parser – Yacc/BISON

- Le générateur d'analyseur syntaxique (de parser)
  1. *Prend en entrée un ensemble de règles grammaticales hors contexte, les règles d'associativités et de priorités,*
  2. *Construit un squelette de parser sous forme d'automate à pile (i.e. à mémoire) (plus proche des machines de Turing que des automates d'états finis) et transforme les règles en table de transitions de cet automate,*
  3. *Convertit l'automate à pile en un programme exécutable en encodant les transitions en table indexée par les éléments de T (terminaux) et de NT (non-terminaux) et passe cela en paramètre au squelette de parser construit auparavant, le tout dans un langage cible (généralement C).*

Génération automatique de parser – Yacc/BISON



BISON

- Entrée (e.g. grammar.y)
  - BISON prend une description de grammaire HC (plus particulièrement une grammaire LALR(1))
- Sortie (e.g. grammar.tab.c)
  - Un analyseur syntaxique bottom-up en C capable d'analyser la grammaire en entrée

Structure d'une spécification BISON (1)

- %{
- *déclarations C*
- %}

- *Déclarations Bison*
- %%

- *Règles de grammaire*
- %%
- *Code C additionnel*

- Définitions de Macros

- Déclare fonctions et variables qui sont utilisées dans les actions des règles de grammaires

- Les "#include"
- %{, %} peut être omise si aucune déclaration C n'est nécessaire

- Exemple :
  - %{
  - #include <math.h>
  - #include "calc.h" /*Contains some types definitions*/
  - int yylex (void);
  - void yyerror (char const *);
  - %}

Structure d'une spécification BISON (2)

- %{
- *déclarations C*
- %}

- *Déclarations Bison*
- %%

- *Règles de grammaire*
- %%
- *Code C additionnel*

La déclaration des NT (%type) est optionnelle

- Déclarations qui définissent les symboles terminaux et non-terminaux
- Spécifient les règles de précédence et les règles de priorité
- Les symboles terminaux à un, seul caractère n'ont pas besoin d'être déclarés
- Un simple symbole terminal :
  - %token *name*
- Un terminal sous forme de chaîne de caractère
  - %token *arrow "=>"*
- Exemple:
  - %**start S**
  - %**token NOMBRE**
  - %**type instruction**
  - %**left '-' '+'**
  - %**left '*'**

Structure d'une spécification BISON (3)

- %{
  déclarations C
  %}

- *Déclarations Bison*

%%

- *Règles de grammaire*

%%

- *Code C additionnel*

- Une règle de grammaire dans Bison est sous forme :
  - *result : components…*
    ;

  Example:
  - *exp : exp '+' exp*
    ;

- Sémantique : Usuellement, Il y a une action qui suit les composants de la règle (sa partie droite)

  - *expr : expr '+' expr*
  - *{ $$ = $1 + $3; }*
  - *;*

Exemple d'utilisation de BISON – Spécification de la grammaire (1)

- $ cat arith.y
  - %{
  - #include <ctype.h>
  - #include <stdio.h>
  - int yylex(void);
  - void yyerror (char const *s);
  - int result ;
  - %}
  - /* *1. Le start est le non-terminal S (NT explicite)* */
  - %**start S**
  - /* *2. Définitions des tokens (terminaux). NB. Ce n'est pas nécessaire de déclarer token les terminaux qui tiennent sur une seule lettre (ex. '+', '-', '*', etc.* */
  - %**token NOMBRE** /* *(T explicite)* */
  - /* *3. Désambiguisation* */ /*(+, - T implicite) */
  - %**left '-' '+'** /* Désambiguisation : '+' et '-' sont associatifs gauches de même priorité */
  - %**left '*'** /* '*' est associatif gauche et est plus prioritaire que le '+' et le '-' */
  - %%

1.a- Spécification du start, des terminaux, des priorités, des associativités

Exemple d'utilisation de Bison – Spécification de la grammaire (2)

- /* 4. Implantation des règles et des actions simplistes associées */
- S :
- Exp { $$ = result = $1;}
- ;

1.b- Spécification des Règles

- Exp : /* *(NT implicite) */
- Exp '+' Exp { $$ = $1 + $3 ; }
- Exp '-' Exp { $$ = $1 - $3 ;}
- Exp '*' Exp { $$ = $1 * $3 ; }
- '(' Exp ')' { $$ = $2 ; }
- NOMBRE { $$ = $1 ; }
- ;
- %%

Actions sémantiques de la grammaire attribuées

- Dans l'analyse bottom-up,
  - ++ Plus de besoin de transformer la grammaire !
  - -- Mais le résultat dépendra du générateur d'analyseur utilisé !!

Exemple d'utilisation de BISON – Spécification de la grammaire (3)

- int **main**(int argc, char** argv){
- */* yyparse : fonction d'analyse syntaxique bottom-up */*
- int retour = **yyparse**(); // 0 ou 1
- printf("%d", result);
- return retour ;
- }

- */* yyerror : fonction de gestion d'erreur appelée par yyparse en cas d'erreur */*
- void **yyerror** (char const *s)
- {
- printf ("%s\n", s) ;
- }

- /* yylex : fonction lire_token (ici elle est simple sans utilisation de FLEX) */

1.c- Spécification du main + quelques fonctions en C

- int **yylex**(){
  - char c;
  - c = getchar();
  - if (c== '\n') return 0;
  - if ((c==' ') || (c=='\t')) return yylex();
  - if (c == '+') return '+';
  - if (c == '-') return '-';
  - if (c == '*') return '*';
  - if (c == '(') return '(';
  - if (c == ')') return ')';
  - if (isdigit(c)) {
  - ungetc (c, stdin);
  - scanf ("%d", &yylval);
  - return NOMBRE ;
  - }
- }

Token explicite/implicites

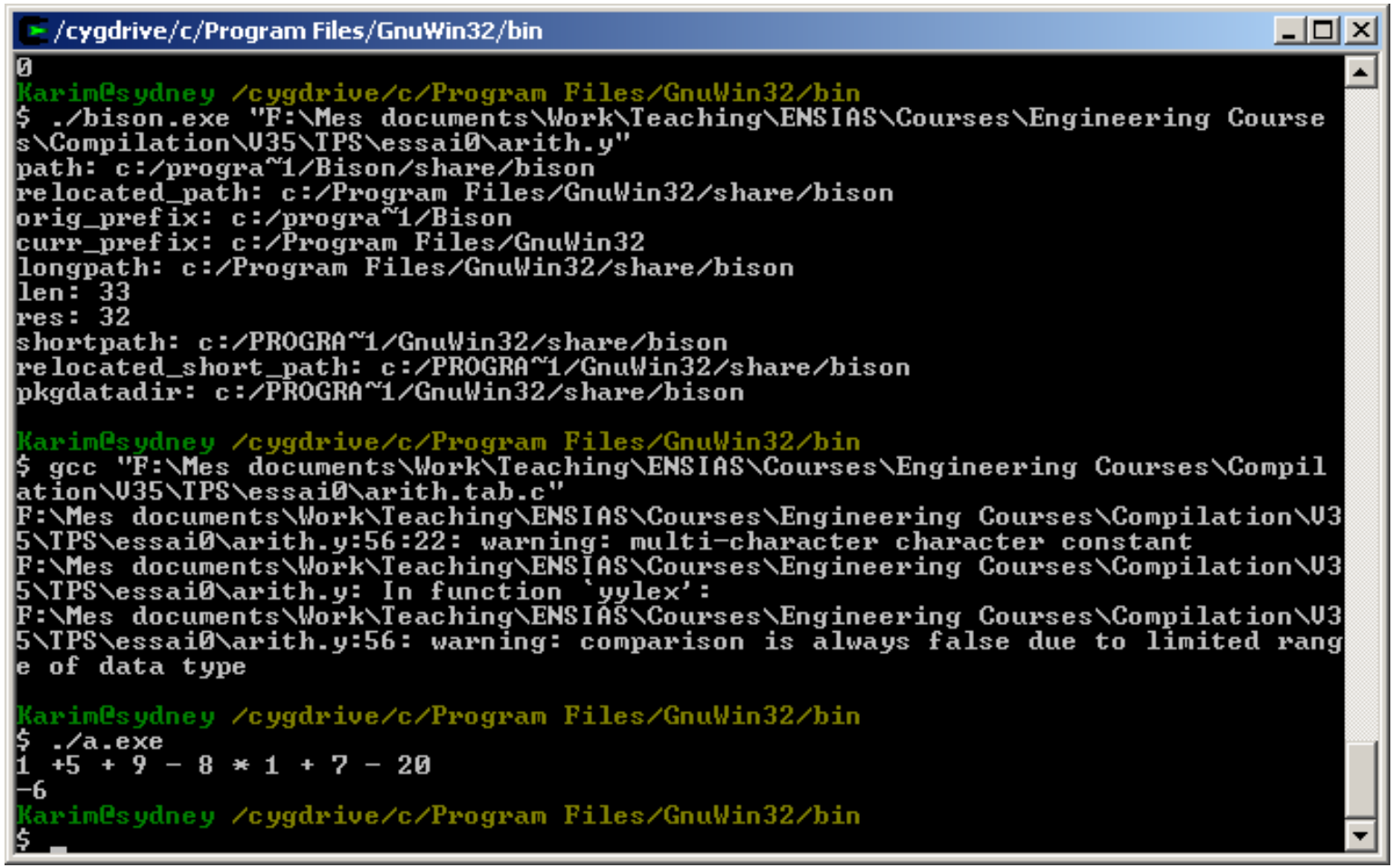
- Un token X est un code qui identifie d'une manière unique l'ER qui reconnaît les mots de type X
- La fonction indicatrice yylex renvoie ce token pour montrer qu'elle a reconnu un mot de type X
- Les tokens explicites sont des constantes
  - token NOMBRE
  - equiv #define NOMBRE uneconstante
- Les tokens implicites '+', '-', etc. sont représentés par le propores code ascii
  - equiv #define PLUS '+'
  - #define MOINS '-'
  - #define MULT '*'
  - #define PARO '('
  - #define PARO ')'
  - On pourrait définir %token PLUS MOINS ….
  - Ou %token PLUS '+'
  - %token MOINS '-'
  - Etc.

```
/* yylex : fonction lire_token (ici elle est simple sans utilisation de FLEX) */

int yylex(){
    char c;
    c = getchar();
    if (c== '\n') return 0;
    if ((c==' ') || (c=='\t')) return yylex();
    if (c == '+') return '+'; // equiv return PLUS;
    if (c == '-') return '-'; // equiv return MOINS;
    if (c == '*') return '*'; // equiv return PARF;
    if (c == '(') return '('; // equiv return PARO;
    if (c == ')') return ')'; // equiv return PARF;
    if (isdigit(c)) {
      ungetc (c, stdin);
      scanf ("%d", &yylval);
      return NOMBRE ;
    }
}
```

Quelques fonctions BISON

- int yyparse ()
  - Fonction générée par BISON pour l'analyse syntaxique/contextuelle, doit être appelée par le main
- void yyerror(char const *s)
  - Fonction appelée par yyparse pour la gestion des erreurs, doit être écrite par le programmeur
- int yylex ()
  - appelé par le yyparse pour l'analyse léxicale des tokens (peut être générée par FLEX ou écrite par le programmeur)

Exemple d'utilisation de Yacc/Bison – l'analyseur généré en action *wow ;-)*



(2) Génération automatique de l'analyseur bottom-up

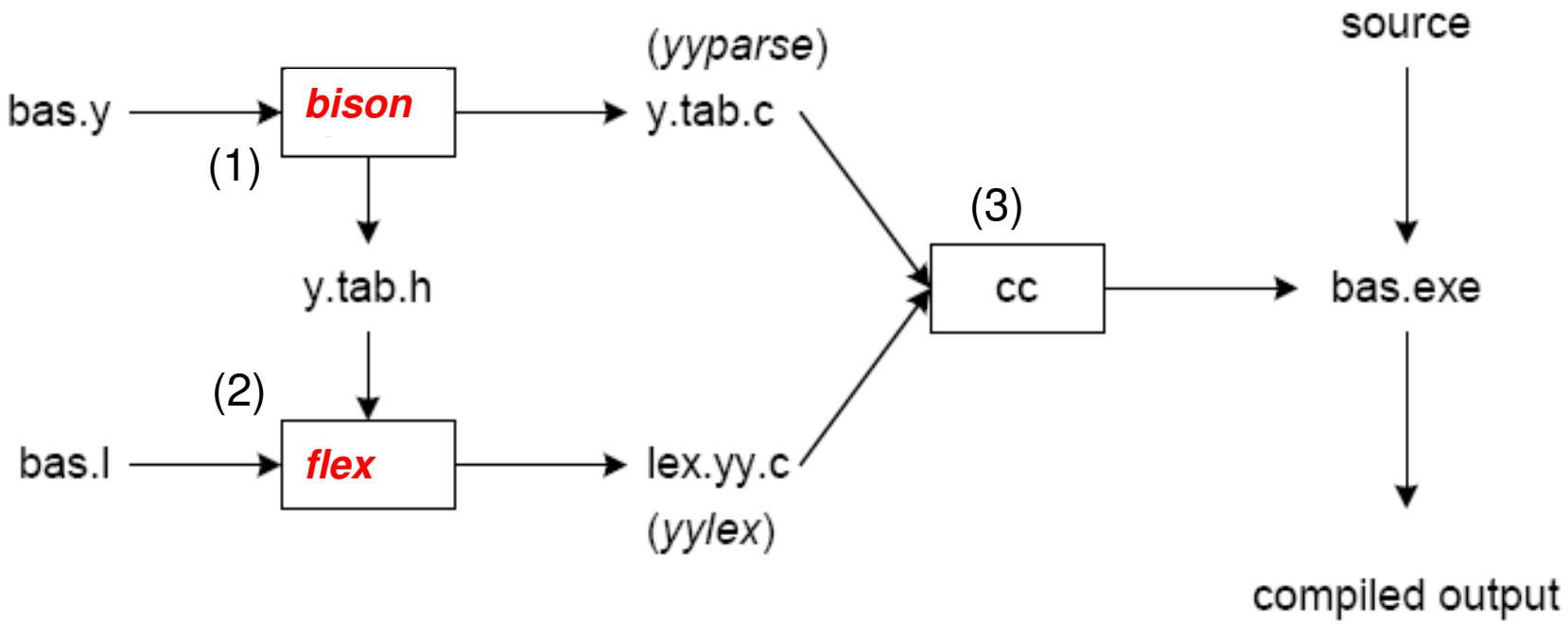
(4) Compilation de l'analyseur généré

(3) Exécution de l'analyseur généré (interpréteur)

Etapes d'exploitation de BISON

1. Génération de l'analyseur BISON
   - bison parserzz.y
2. Compilation de l'analyseur
   - gcc -o parserzz parserzz.tab.c
3. Exécution de l'analyseur
   - ./parserzz < monpremierpgm.zz

FLEX & BISON ensemble



FLEX & BISON ensemble

Bison	Flex
EXP : EXP ‘+’ EXP	‘’+’’ { return ‘+’ ;} // ‘+’ est un token implicite
%token plus EXP : EXP plus EXP	‘’+’’ { return plus ;} // ‘+’ est un token explicite
%token plus ‘+’ EXP : EXP plus EXP	‘’+’’ { return plus ;} // ‘+’ est un token explicite

FLEX & BISON ensemble

- Dans les spécifications BISON : définir des tokens en partie déclaration BISON
  - Exemple :
    - %{
    - %}
    - %token IDF_TOK PROGRAM_TOK IF_TOK THEN_TOK BEGIN_TOK END_TOK PROCEDURE_TOK FUNCTION_TOK IDF_TOK INT_TOK
    - %%
    - Au fait, un TOKEN est une constante qui matérialise al présence d'un terminal et qui gère la communication entre l'analyseur lexical et syntaxique
    - Nous n'avons plus besoin d'écrire la fonction yylex dans la partié code C (3ème section) de la spécification BISON
- Dans les spécifications des ER dans FLEX retourner ces tokens dès détection
  - Exemple
    - {%
      - #include "parserzz.tab.h " // fichier généré par bison -d (continet les définitions des tokens)
      - #include "types_parserzz.h" // fichier écrit par le programmeur et qui va contenir la définition de YYSTYPE
      - // (l'union des valueTypes). Doit aussi être inclus dans parserzz.y
      - // (utiliser la macro #ifndef TYPES #define TYPES #include "types_parserzz.h" #endif
      - // pour éviter les inclusions multiples)
    - %}
    - %%
    - …
    - "procedure" {return PROCEDURE_TOK;}
    - "function" {return FUNCTION_TOK;}
    - {IDF} { yylval.name = (char *) malloc(sizeof(yytext)) ; strcpy(yylval.name, yytext); return IDF_TOK;}
    - . { return yytext[0]; }

Etapes d'exploitation de BISON & FLEX ensemble

- Génération de l'analyseur syntaxique BISON (avec l'option –d pour générer un fichier parserzz.tab.h en plus de parserzz.tab.c)
  - bison –d parserzz.y
- S'assurer que lex.yy.c et yy.tab.c font référence aux mêmes tokens (i.e. incluent le même parserzz.tab.h file)
- Génération de l'analyseur lexical FLEX (produit lex.yy.c)
  - flex –i scanerzz.l
- Compilation
  - gcc -o parserzz parserzz.tab.c lex.yy.c
- Exécution de l'analyseur
  - ./parserzz < monpremierpgm.zz

Compilation

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Lectures recommandées - livres

1. « Engineering a Compiler » ©, K. D. Cooper & L. Torczon, Morgan Kaufmann 2004
2. « Introduction to Automata Theory Languages and Computation » ©, J. E. Hopcroft, R. Motwani, J. D.Ullman, 2001, Addison Wesley
3. « Compilers and Compiler Generators » ©, P.D. Terry, 2000

Plan du cours

1. Module initial Motivation & Définitions
2. Module 1 Analyseur lexical *(langages réguliers et théorie des automates)*
3. Module 2 Analyseur syntaxique *(langages hors contexte)*
4. **Module 3** Analyseur sémantique *(grammaires attribuées)*
5. Module 4 Représentations Intermédiaires
6. Module 5 Génération de code intermédiaire
7. Module 6 Optimisation et Génération de code
8. Conclusion et perspectives

Questions ouvertes

1. Quels peuvent être les limites du formalisme de grammaire HC ?
   - Le formalisme de grammaire HC ne peut pas décrire les aspects suivants
     - Types des variables
     - Compatibilité du type et des opérations
     - Valeurs courantes des variables
     - Liens des identificateurs qui portent le même nom avec leur variable
     - Vérifier si les variables ont été déclarées avant leur utilisation
2. Comment peut – on palier ce problème ?

- On peut déclarer :
- float x; int x; double x;
- Car syntaxiquement c'est correct
- P ::= LD LI
- LD ::= LD D | D
- LI ::= LI I | I
- I ::= ….
- D ::= type idf
- Les déclarations et les instructions ne partagenet pas une connaissance commune

Module 3
Analyseur sémantique (contextuel)

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

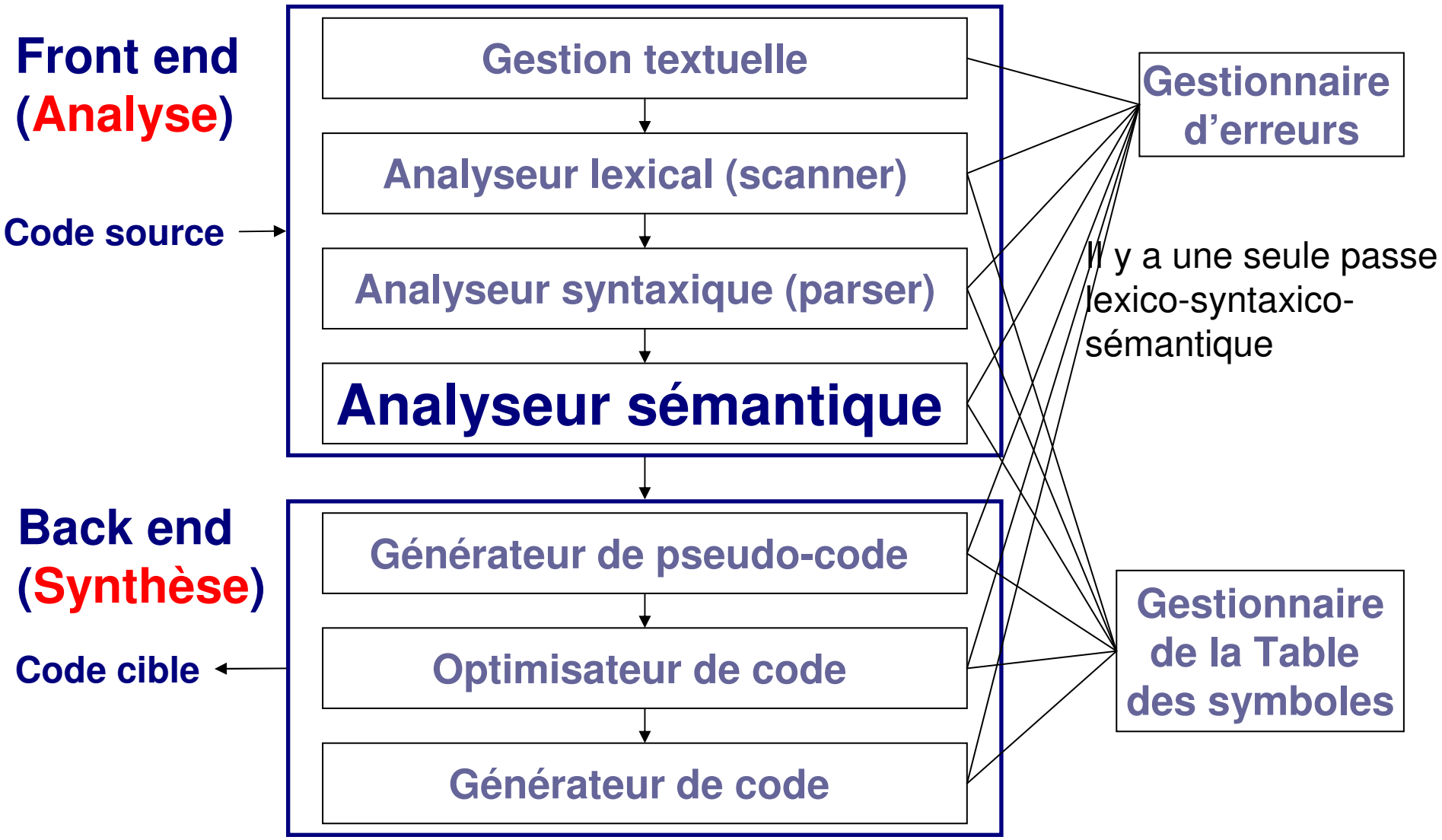
Analyseur sémantique – Plan

- Introduction
- Concepts
  - Arbre abstrait
  - Représentation intermédiaire
  - Grammaires attribuées
  - Synthèse, héritage d'attributs
  - Environnement des noms et typage
- Transformations
- Aspects opérationnels : actions sémantique en BISON

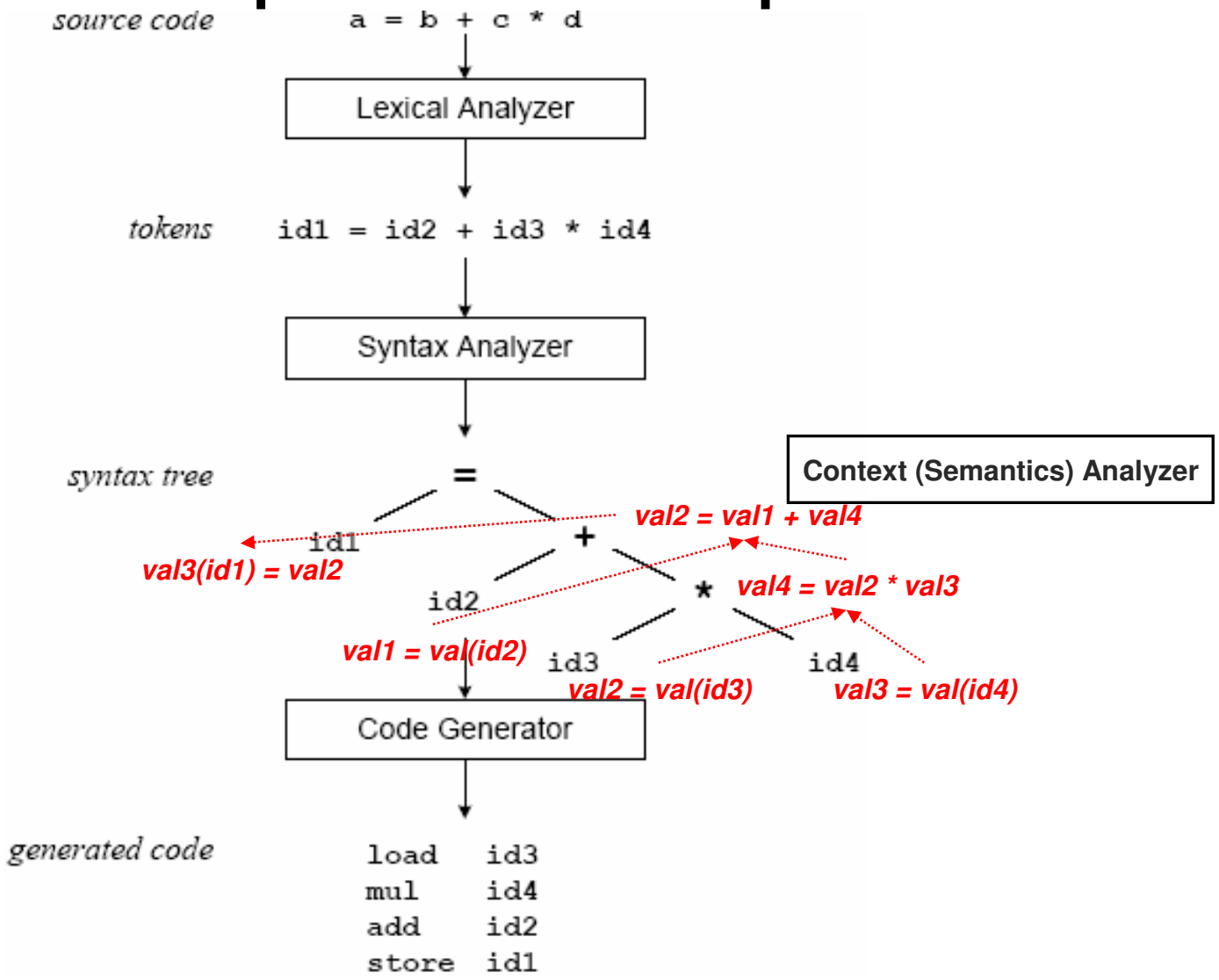
Analyseur sémantique

- Pour enrichir (décorer) l'arbre syntaxique (produit par l'analyse syntaxique), l'analyseur sémantique
  1. parcourt l'arbre syntaxique
  2. sauvegarde des informations à propos
     - des types des variables (selon leur déclaration)
     - des dimensions des variables (selon leur déclaration)
     - taille de stockage (selon le type et la dimension),
  3. utilise ces informations pour vérifier la correction des types de différentes expressions et instructions du code
  4. détermine où il doit insérer une conversion entre des types de données (e.g. transformer un flottant en un entier)

Positionnement de l'analyseur sémantique



Exemple simple de compilation



Grammaires attribuées

- Une **grammaire attribuée** (attributed grammar) est constituée
  - d'une grammaire hors-contexte
  - Augmentée
    1. d'un ensemble d'attributs
    2. d'un ensemble de règles qui spécifient certains calculs (actions sémantiques)
       - Chaque règle définit la valeur d'un attribut (variable), fonction des valeurs des autres attributs
       - Chaque règle associe l'attribut (qu'elle décrit) à un symbole de la grammaire
       - Chaque instance des symboles de la grammaire apparaissant dans l'arbre syntaxique correspondent à une instance de l'attribut
    - d'un ensemble de conditions (prédicats)
      - Exemples : déclaration des variables, déclaration unique des variables, compatibilité des types

Intérêts des grammaires attribuées

1. Vérification de type (analyse contextuelle : *context-sensitive analysis*)
2. Construction de représentation intermédiaires (*AST*)
3. Calcul dirigé par la syntaxe (*interpréteur*)
4. Génération de code (*compilateur*)

Grammaires attribuées – exemple 1

(Attributs : S.(val), Exp.val, NOMBRE.val *)*

- S :
- Exp1 {S.val = Exp1.val;}
- ;

- Exp0 : /* *(NT implicite) */*
- Exp1 '+' Exp2 { Exp0.val = Exp1.val + Exp2.val; }
- Exp1 '-' Exp2 {Exp0.val = Exp1.val - Exp2.val;}
- Exp1 '*' Exp2 {Exp0.val = Exp1.val * Exp2.val; }
- '(' Exp1 ')' {Exp0.val = Exp1.val; }
- NOMBRE {Exp0.val = NOMBRE.val }

Grammaires attribuées – exemple 1 - Implémentation sous Bison

- S :
- Exp { $$ = result = $1;}
- ;

- Exp : /* *(NT implicite)* */
- Exp '+' Exp { $$ = $1 + $3 ; }
- Exp '-' Exp { $$ = $1 - $3 ;}
- Exp '*' Exp { $$ = $1 * $3 ; }
- '(' Exp ')' { $$ = $2 ; }
- NOMBRE { $$ = $1 ; }

Grammaires attribuées – exemple 2

- Soit la Grammaire HC du langages des listes binaires sur $\Sigma=\{0,1\} \cup \{-,+\}$
- G:
  - R1: Number $\rightarrow$ Sign List
  - R2: Sign $\rightarrow$ +
  - R3: Sign $\rightarrow$ -
  - R4: List $\rightarrow$ Bit
  -
  - R5: List $\rightarrow$ List Bit
  - R6: Bit $\rightarrow$ 0
  - R7: Bit $\rightarrow$ 1

Grammaires attribuées – exemple 2

- Soit la Grammaire HC du langages des listes binaires sur $\Sigma=\{0,1\} \cup \{-,+\}$
- G: ***(* Attributs : Number.(val), List.(pos,val), Sign.(neg), Bit.(pos,val) *)***
  - R1: Number → Sign List
    List.pos ← 0
    if (Sign.neg = true) then {
    Number.val ← - List.val
    } else { Number.val ← List.val }
  - R2: Sign → +
    Sign.neg ← false
  - R3: Sign → -
    Sign.neg ← true
  - R4: List → Bit
    Bit.pos ← List.pos
    List.val ← Bit.val
  - R5: $List_0 \rightarrow List_1$ Bit
    $Bit.pos \leftarrow List_0.pos$ *(* les indices pour clarifier *)*
    $List_1.pos \leftarrow List_0.pos + 1$
    $List_0.val \leftarrow List_1.val + Bit.val$
  - R6: Bit → 0
    Bit.val ← 0
  - R7: Bit → 1
    $Bit.val \leftarrow 2^{Bit.pos}$

Grammaires attribuées – exemple 2

- Attributs de G :
  - Number.val : entier
  - List.pos : entier
  - List.val : entier
  - Sign.neg : booléen
  - Bit.pos : entier
  - Bit.val : entier

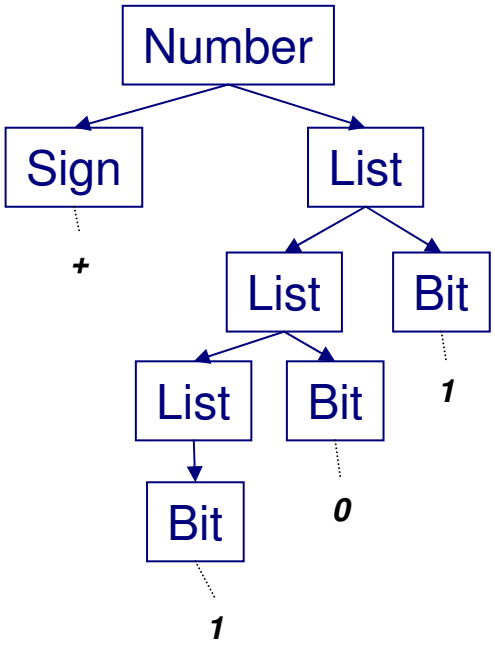
- Dans une règle,
  - un **attribut** est dit **hérité** si sa valeur est calculée à partir d'un attribut de ses parents
    - Soit X0 → X1 ... Xn une règle, A(Xj) = f(A(X0), ..., A(Xi), …)
      - Exemple : R4: List → Bit **Bit.pos ← List.pos**
  - un **attribut** est dit **synthétisé** si sa valeur est calculée à partir d'un attribut de ses fils
    - Soit X0 → X1 ... Xn une règle, A(X0) = f(A(X1), ... , A(Xn))
      - Exemple : R4: List → Bit **List.val ← Bit.val**
  - un **attribut intrinsèque** associé à un terminal
    - Valeurs Lexicales fournies par l'analyseur lexical

Où sont stockés les attributs ?

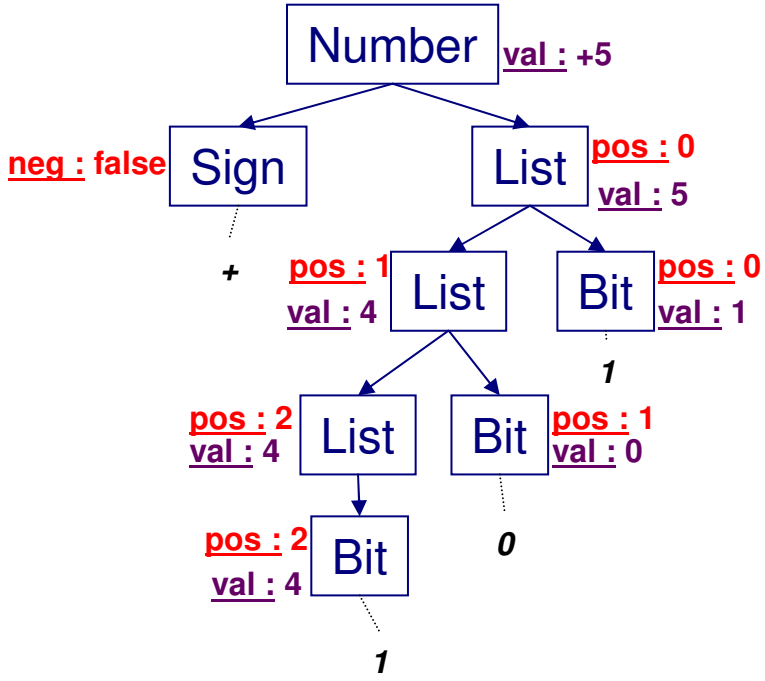
- Pour chaque symbole de la grammaire, le fait de l'empiler (dans la pile lors de l'opération shift) empile également la valeur de son attribut
- Les attributs synthétisés peuvent être évalués durant l'étape de réduction
- Au moment de l'évaluation, tous les attributs a_i dont dépend un attribut X (=f(ai)) sont sur la pile en position connue vu le caractère deterministe de l'algorithme shift-reduce.
  - Exemple :
  - Soit la règle $A \rightarrow XY$ { A.val = X.val + Y.val }
  - Reduce :
    - À la réduction par $A \rightarrow XY$, *la tête de pile* contient X.val & Y.val
      - pop X.val & Y.val
      - Calculer A.val
      - push A.val

      } // similaire au comportement de shift-reduce
  - Shift : *action[s,a] = shift s* devient *action[s,a] = shift a*.$$, *s*

Grammaires attribuées – arbre syntaxique attribué (décoré)

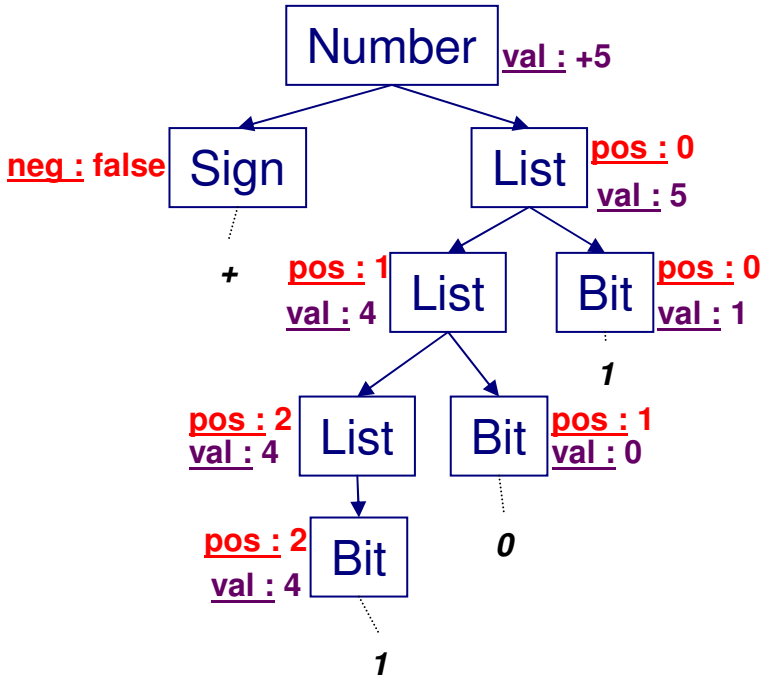


Arbre syntaxique

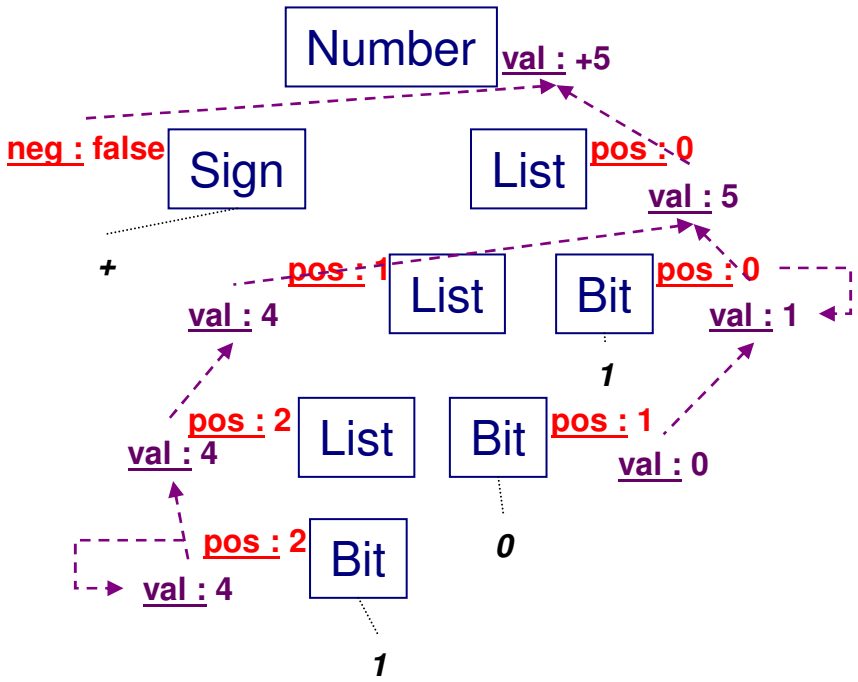


Arbre attribué (décoré)

Grammaires attribuées – graphe de dépendances

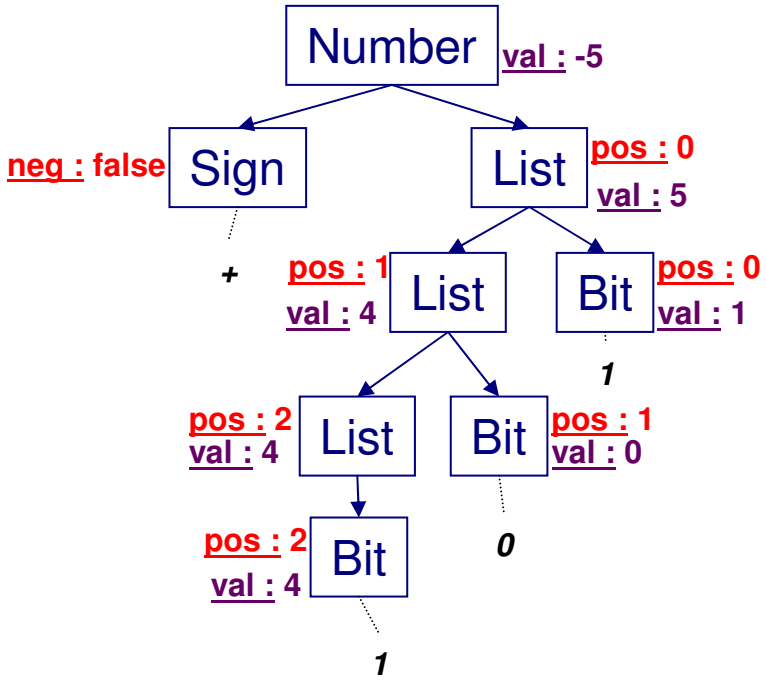


Arbre attribué (décoré)

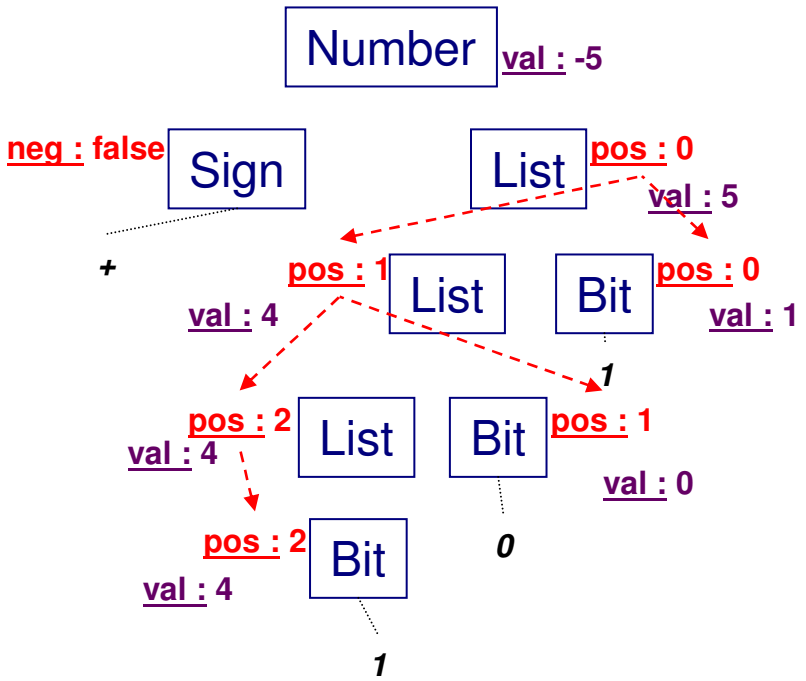


Graphe de dépendance
pour l'évaluation
de attribut **synthétisé** val

Grammaires attribuées – graphe de dépendances (suite)



Arbre attribué (décoré)



Graphe de dépendance pour l'évaluation de attribut **hérité** pos

Grammaires attribuées – graphe de dépendances (suite)

- Le graphe de dépendances d'une grammaire attribuée
  - Décrit les **dépendances entre attributs**, engendrées par les règles de calcul des attributs
  - Est toujours **sans-circuit** (i.e. les règles de calcul des attributs possèdent une relation d'ordre qui régit leur exécution). Le même attribut nécessaire pour l'héritage et pour la synthèse (mode in out : interdit par bison)
    - Exemple : soit la règle A → B
      - Et soient les actions sémantiques suivantes
        - A.s = B.i
        - B.i = A.s + 1
  - Définit le **dataflow** (flux des valeurs des attributs)
    - Dataflow top-down si héritage d'attributs, (in)
    - Dataflow bottom-up si synthétisation d'attributs. (out)

Classe de grammaires attribuées

- Grammaires **s-attribuées** (s-attributed grammar) : **possèdent seulement des attributs synthétisés**. Elles sont donc compatibles avec une évaluation effectuée en parallèle avec une analyse syntaxique ascendante
  - *Les générateurs d'analyseurs ascendants (LR) ne permettent d'utiliser que des grammaires s-attribuées (en l'occurence Bison).*

- Grammaires **l-attribuées** (l-attributed grammar) : **possèdent seulement des attributs hérités**. Les attributs d'un noeud ne dépendent que du père ou des frères a sa gauche, d'où leur compatibilité avec une évaluation effectuée en parallèle avec une analyse syntaxique descendante
  - *S-attributed $\subset$ L-attributed*

- Grammaires **ordonnées** : l'ordre d'évaluation est fixé, qui a la différence des précédentes peut demander plusieurs visites d'un noeud

- Grammaires **fortement non circulaires** : garantissent que le graphe de dépendance des règles d'évaluation est sans-circuit

Méthodes d'évaluation des attributs

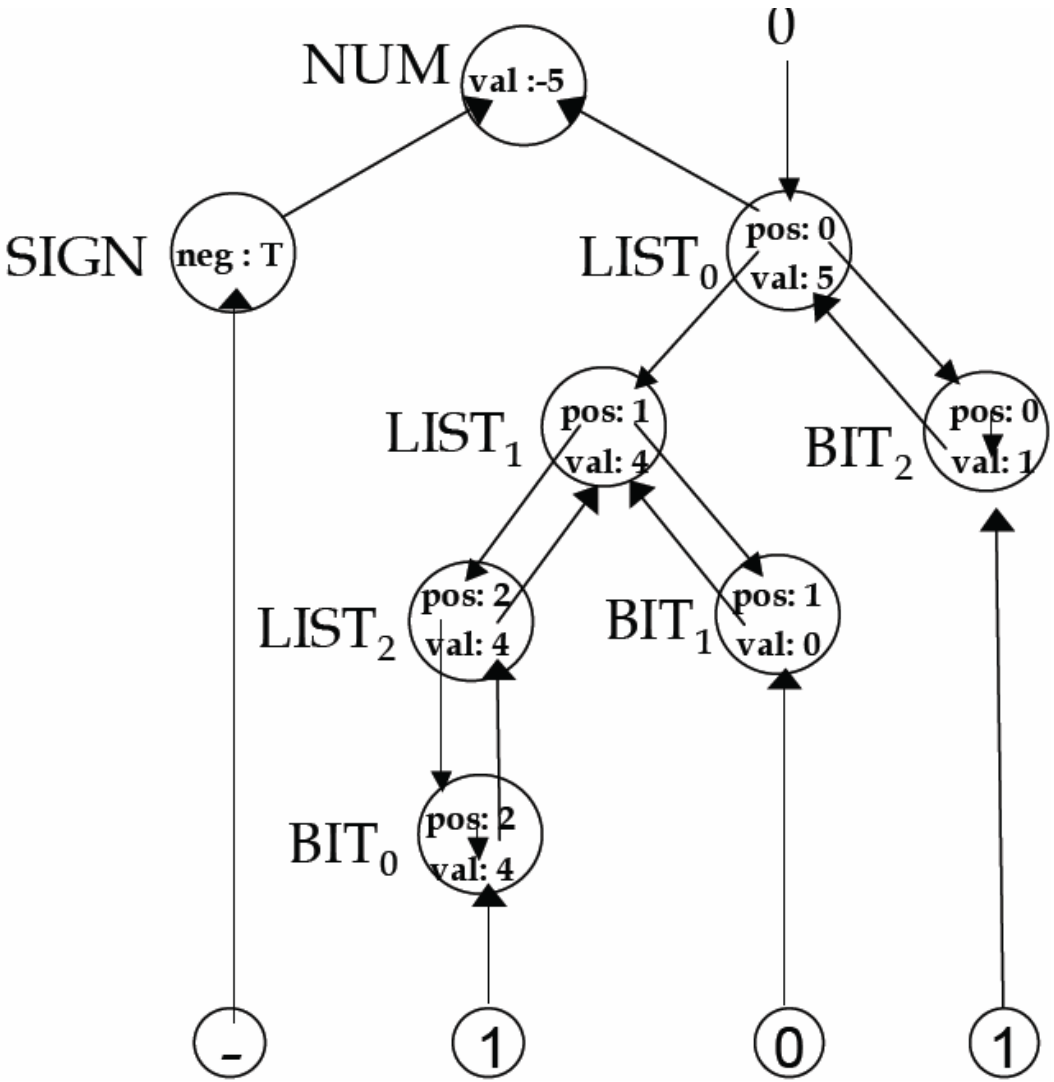
- La grammaire attribuée étant une spécification de haut niveau indépendant du schéma d'évaluation, il y a besoin d'une méthode d'évaluation des attributs
  - ***Méthodes à base de la grammaire (Rule-based methods - treewalk)***
    1. Analyser les règles au moment de génération du compilateur (design time)
    2. Déterminer un ordre statique à ce stade
    3. Effectuer l'évaluation des noeuds dans cet ordre au moment de la compilation (run time)

Plus courante ◎
  - ***Méthodes à base d'arbre (Parse-tree methods - dynamic)***
    1. Construire l'arbre syntaxique
    2. Construire le graphe de dépendance
    3. Trouver un ordre topologique (*ordre légal d'évaluation*) dans le graphe
    4. Effectuer l'évaluation à la base de cet ordre (*graphe avec circuit refusé*)

  - ***Méthodes ad-hoc (Ad-hoc methods - passes, dataflow)***
    1. ignorer l'arbre syntaxique et la grammaire
    2. choisir un ordre convenable et l'utiliser (parcours à chaînage avant ou arrière forward-backward, parcours alternés)

  - ***Problèmes :*** *Circularité du graphe + La meilleure stratégie d'évaluation dépend de la structure de la grammaire*

Méthode à base d'arbre



1. SIGN.neg
2. LIST0.pos
3. LIST1.pos
4. LIST2.pos
5. BIT0.pos
6. BIT1.pos
7. BIT2.pos
8. BIT0.val
9. LIST2.val
10. BIT1.val
11. LIST1.val
12. BIT2.val
13. LIST0.val
14. NUM.val

Évaluation selon le type de la grammaire attribuée et le type de l'analyseur

	Analyseur LR	**Analyseur LL**
Grammaire L-attribuée	Parcours en profondeur d'abord DFS (depth first search)	Une seule passe bottom-up
Grammaire S-attribuée	Une seule passe top-down	Transformation S2L
hybride		

S-attributed $\subset$ L-attributed
Donc une grammaire S-attribuée peut être transformée systématiquement en L-attribuée. L'inverse est également possible, mais pas toujours.

Évaluation un nœud avec attributs hérités et synthétisés : Parcours en profondeur d'abord DFS (*depth first search*)

- **procedure** DFS(n: node) // n est **begin**
  - **for** each child m of n, from left to right **do**
    - **begin**
      - evaluate inherited attributes of m according to their parent values
      - DFS(m)
    - **end**
  - evaluate synthesized attributes of n
- **end**

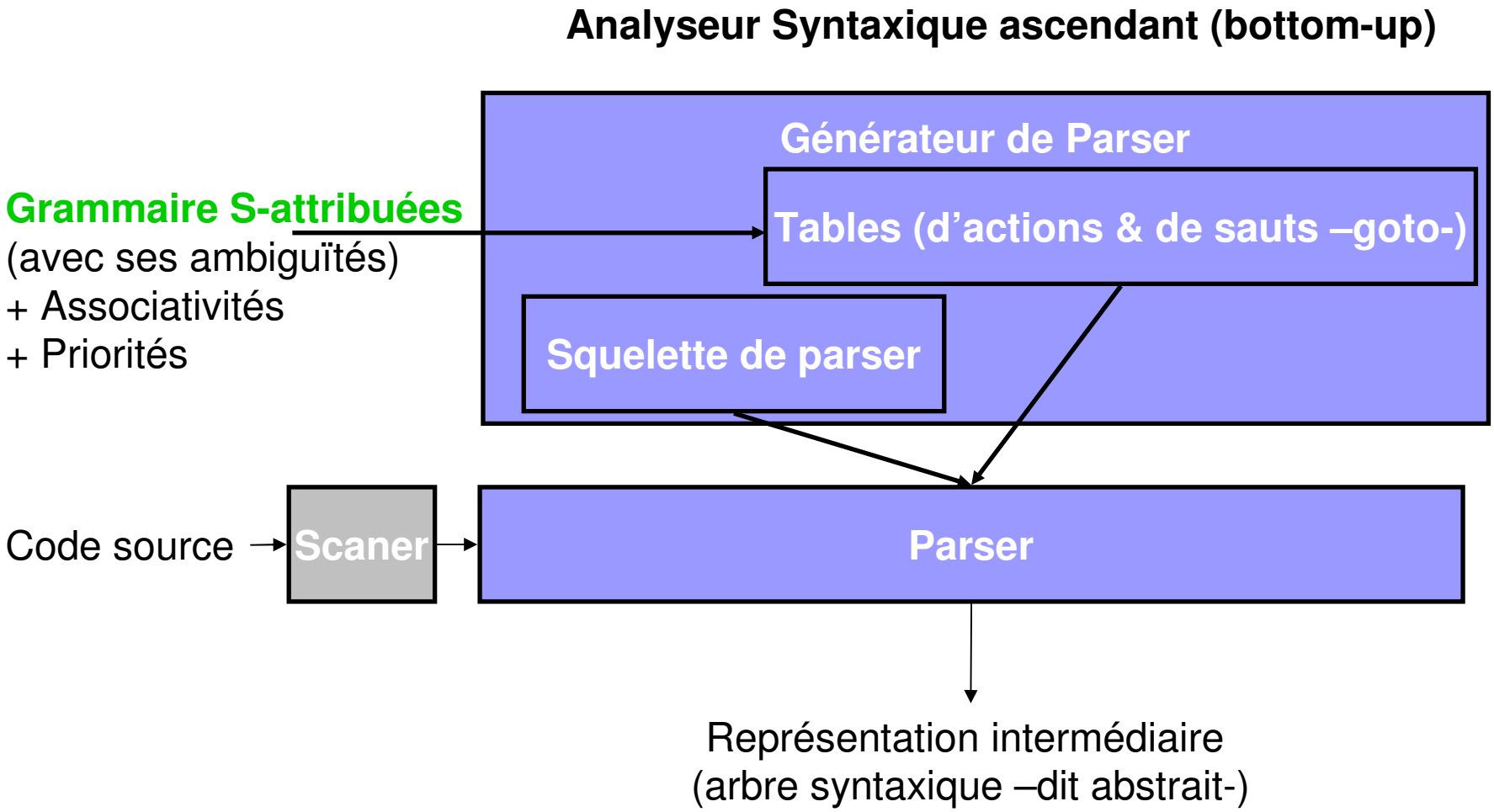
Des concepts à la pratique - Introduction au générateur d'analyseur sémantique ascendant avec BISON

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Génération automatique de parser – BISON



BISON & l'analyse sémantique (1)

- Attributs par défaut de BISON
  - À chaque symbole (terminal ou pas), bison associe une valeur
    - $$ désigne la valeur de l'attribut du non-terminal gauche d'une règle (valeur escomptée)
    - $1 désigne la valeur du 1er symbole de la partie droite,
    - $2 désigne du 2ème symbole de la partie droite (terminal ou pas), etc.
    - Yylval est la valeur associé au $I courant et est de type YYSTYPE (#define YYSTYPE int)
    - Ex. Si strcmp(yytext,"2009") == 0
      - yylen = strlen(yytext) se fait automatiquement
      - yylval = ascii2int(yytext) doit être codé par le programmeur lors de l'analyse lexicale !!!

- Cas 1 - un attribut de même type simple pour les non-terminaux de la grammaire
  - Exemple : exp.**epsi**, terme.**epsi**, facteur.**epsi**, elem.**epsi** : boolean (int)
  - Bison utilise par défaut le type int pour les attributs (utilisés dans les actions semantiques).
    - Exemple : Exp: Exp '+' Term {$$ = $1 || $3;}
  - Pour spécifier d'autres types, il faut définir YYSTYPE en macro (section 1 de la spécification BISON) le type de yylval (YYSTYPE yylval;)
    - Exemple : #define YYSTYPE double

BISON & l'analyse sémantique (2)

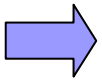
- Cas 2 - plusieurs attributs de types complexes pour les non-terminaux de la grammaire
  - Exemple : ***Number.(val), List.(pos,val), Sign.(neg), Bit.(pos,val)***
    - Dans la partie Déclarations C
      - typedef struct {
        - double **val**; /* List.val, Bit.val */
        - int **pos**; /* List.pos, Bit.pos */
      - } attribute ;
    - Dans la partie Déclarations Bison
      - %union valueType{
        - int neg ; /* sign.neg */
        - double val ; /* number.val */
        - attribute attr; /* List & Bit attribute */
      - } /* %union définit YYSTYPE (i.e les variables des attributs qui peuvent être en < >) */

équivaut à

```
typedef union {
        int neg ;  /* sign.neg */
        double val ;/* number.val */
        attribute  attr; /* List & Bit attribute */
} valueType;
#define YYSTYPE valueType
```

    - Dans la partie Règles de grammaire Bison

R1: **Number → Sign List**
List.pos ← 0
if (Sign.neg = true) then {
Number.val ← - List.val
} else { Number.val ← List.val }



```
Number : Sign List    {
        $<attr>2.pos  = 0;
            if ($<neg>1 == 1) {
               $<val>$ = result = - $<attr>2.val;
            } else { $<val>$ = result = $<attr>2.val;}
               }
;
```

BISON & l'analyse sémantique (2)

- Implicitement :
  - **attr(pos,val**) est associé à **List** : $2 de cette règle
    - $<attr>2.pos, $<attr>2.val
  - **neg** est associé à **Sign** : $1 de cette règle
    - $<neg>1
  - **val** est associé à **Number** : $$ de cette règle
    - $<val>$

Association explicite (optionnelle) entre T/NT et attributs

- %union{
  - ... /* attributs pour les terminaux : entiers, reels ...
  - struct Instruction *instr;
  - struct Expression *expr;
  - struct Denotation *denot;
  - struct { const char* id; struct Expression* ex; } idex;
  - struct { Type* typ; struct ListDeclaration* lde; } tyde;
- }

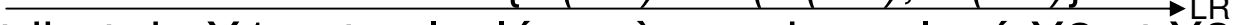
- %type<instr> instruction
- %type<expr> expression
- %type<denot> denotation prefixe
- %type<tyde> declaration
- %type<idex> decl

Association explicite (optionnelle) entre T/NT et attributs

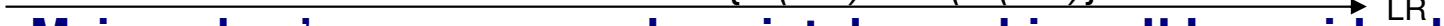
- %union{
  - double r;
- }

- %type<r> E
- %token<r> N

S-attribut & L-attribut sous Bison

- **S-attribut ↑**
  - **X1 : X2 X3** $\{A(X1) = f\,(A(X2), A(X3)\}$
    

  - L'attribut de X1 est calculé après avoir analysé X2 et X3 et ainsi il sera possible de le synthétiser

- **L-attribut ↓**
  - **On aurait bien aimé faire**
    - **X1 : X2** $\{A(X3) = f\,(A(X1)\}$ **X3**
      

    - **Mais, cela n'a aucun sens du point de vue bison !! Les mid-rules ont une tout autre comportement**
  - **Solutions :**
    1. Soit transformer la grammaire L-attribuée en S-attribuée
    2. Soit utiliser un contexte global que l'on accède/alimente au besoin (pile/table/listes autre que celles manipulée par le shift-reduce)

Exemple de Transformation d'une L-attribuée en S-attribuée

```
%start Number                    /* Axiome : le start est le non-terminal Number*/
%type <value> Number Sign List Bit

%union value{
int neg ; /* sign.neg */
double val ; /* number.val, List.val, Bit.val*/
}

%%
/* 4. GRAMMAR RULES DEFINITION AND RELATED ACTIONS */
Number : Sign List { if ($<neg>1 == 1) {$<val>$ = result = - $<val>2; }
                          else{ $<val>$ = result = $<val>2;} }
       ;

Sign :          '+'    {$<neg>$ = 0;}        |  '-'            {$<neg>$ = 1;} |  {$<neg>$ = 0;} ;

List :   Bit {  $<val>$ = $<val>1; }

| List Bit {    $<val>$ = (2*$<val>1) + ($<val>2);  }
;

Bit :  '0'                          {$<val>$ = 0;} | '1' { $<val>$ = 1; }  ;
```

Exercices en Analyse Sémantique

Extraits d'examens de l'ENSIMAG 2001, 2000 et 1996

Prof. K. Baïna 2010 ©, 7ème édition

Ex1 : Analyse sémantique (2 points) – ENSIMAG Mai. 1997

- On rappelle que le langage des expressions régulières peut être décrit par la grammaire G suivante :
  - exp → exp + terme | terme
  - terme → terme facteur | facteur
  - facteur → elem | (exp) | facteur * | epsilon | vide
  - elem → a | b | c
- T = {+,(,),*,epsilon,vide,a,b,c}. Le terminal epsilon représente la chaîne vide ε et vide représente le langage vide ∅. On rappelle que L(e) désigne le langage dénoté par l'expression régulière e.

Ex1 : Analyse sémantique (2 points) – ENSIMAG Mai. 1997 (suite)

- Question - On veut calculer, pour toute expression régulière e, si $\varepsilon \in L(e)$. Donner sur la grammaire précédente un calcul d'attributs permettant d'évaluer un attribut booléen pour chaque forme possible d'expression (**true** si $\varepsilon \in L(e)$, **false** sinon)

Ex1 : Solution

1.	exp →	exp + terme	**exp0.epsi ← exp1.epsi OR terme.epsi**
2.	exp →	terme	**exp.epsi ← terme.epsi**
3.	terme→	terme facteur	**terme0.epsi ← terme1.epsi AND facteur.epsi**
4.	terme →	facteur	**terme.epsi ← facteur.epsi**
5.	facteur →	elem	**facteur.epsi ← elem.epsi**
6.	facteur →	(exp)	**facteur.epsi ← exp.epsi**
7.	facteur →	facteur *	**facteur.epsi ← TRUE %%car** $\forall\ \alpha\ \varepsilon \in \alpha^*$
8.	facteur →	epsilon	**facteur.epsi ← TRUE**
9.	facteur →	vide	**facteur.epsi ← FALSE**
10.	elem →	a	**elem.epsi ← FALSE %%car** $\varepsilon \notin$ **{a}**
11.	elem →	b	**elem.epsi ← FALSE %%car** $\varepsilon \notin$ **{b}**
12.	elem →	c	**elem.epsi ← FALSE %%car** $\varepsilon \notin$ **{c}**

Ex2 : Analyse sémantique (7.5 points) – ENSIMAG Déc. 2000

- Soit la grammaire abstraite suivante du langage ZZ

1. PROG → LISTE_DECL LISTE_INST

2. LISTE_DECL → DECL
3. LISTE_DECL → LISTE_DECL DECL
4. DECL → **idf** TYPE CONST_IB
5. DECL → LISTE_IDF TYPE
6. TYPE → **INT**
7. TYPE → **BOOL**
8. CONST_IB → **const**
9. CONST_IB → **TRUE**
10. CONST_IB → **FALSE**
11. LISTE_IDF → **idf**
12. LISTE_IDF → LISTE_IDF **idf**

13. LISTE_INST → INST | LISTE_INST INST
14. INST → **idf** DEF EXPR
15. DEF → == | :=
16. EXPR → **idf** | CONST_IB | EXPR OPBIN EXPR | OPUN EXPR
17. OPBIN → **+** | * | **OR** | **AND**
18. OPUN → **NOT**

Ex2 : Analyse sémantique (7.5 points) – ENSIMAG Déc. 2000 (suite)

- **Q1- (2.5 points)** Écrire un système d'attributs pour construire l'environnement de typage, sur la grammaire de la partie déclarations du langage. Pendant cette construction, on vérifiera :
  a) Qu'il n'y a pas de double (ou triple, …) définition des variables
  b) Que les variables initialisées le sont avec des valeurs du bon type : par exemple « x INT TRUE » est incorrecte

- **Q2- (2.5 points)** Écrire un système d'attributs pour la partie instructions du langage pour vérifier que :
  - tous les identificateurs utilisés sont bien déclarés
  - Et que tout est correctement typé (les contraintes de typage sur les instructions de la forme == et := sont les mêmes : les types doivent être identiques à droite et à gauche. Les instructions héritent de l'environnement de déclaration construit dans la question précédente)

- **Q3- (2.5 points)** Écrire un système d'attributs pour la partie instructions tout en héritant de l'environnement construit sur la partie déclarations, pour vérifier que :
  - tout identificateur déclaré est partie gauche d'une instruction et une seule,
  - et que si cette instruction est de la forme :=, l'identificateur a été déclaré avec valeur initiale.

Ex2 – Q1: solution 1

1. **PROG → LISTE_DECL envin↓ envout↑ correct↑ LISTE_INST envin↓ envout↑ correct↑**
   - { LISTE_DECL.envin = ∅
   - LISTE_INST.envin := LISTE_DECL.envout
   - LIST_INST.correct := LISTE_DECL.correct
   - PROG.correct := LISTE_DECL.correct }

2. **LISTE_DECL envin↓ envout↑ correct↑ → DECL envin↓ envout↑ correct↑**
   - { DECL.envin := LISTE_DECL.envin,
   - LISTE_DECL.envout := DECL.envout
   - LISTE_DECL.correct := DECL.correct }

3. **LISTE_DECL_1 envin↓ envout↑ correct↑ → LISTE_DECL_2 envin↓ envout↑ correct↑ DECL envin↓ envout↑ correct↑**
   1. { DECL.envin := LIST_DECL_1.envin
   2. DECL..envin := LIST_DECL_2.envout
   3. LISTE_DECL_1..envout := DECL.envout
   4. LISTE_DECL_1.correct := LISTE_DECL_2.correct AND DECL.correct }

4. **DECL envin↓ envout↑ correct↑ → idf name↑ TYPE name↑ CONST_IB category↑ value↑**
   - { if (idf.name ∉ DECL.envin) then
     - if (TYPE.name == INT) AND (CONST_IB.category == INT) OR (TYPE.name == BOOL) AND (CONST_IB.category == BOOL) then
       - DECL.evout := DECL.envin ∪ {(idf.name, TYPE.name, CONST_IB.value) then
       - DECL.correct:=true
     - else
       - DECL.correct:=false
       - DECL.envout := DECL.envin
   - else
     - DECL.correct:=false
     - DECL.envout := DECL.envin }

Ex2 – Q1: solution 1 (suite)

5. **DECL envin↓ envout↑ correct↑ → LISTE_IDF envin↓ idflist↑ correct↑ TYPE name↑**
   - *{* LISTE_IDF.envin := DECL.envin
   - DECL.correct := LISTE_IDF.correct
   - DECL.envout := DECL.envin ∪ {(idf.name, TYPE.name), idf ∈ LISTE_IDF.idflist } *}*

6. **TYPE name↑ → INT**
   - {TYPE.name = INT}

7. **TYPE name↑ → BOOL**
   - {TYPE.name = BOOL}

8. **CONST_IB category↑ value↑ → const**
   - { CONST_IB.category=INT; CONST_IB.value = const.value }

9. **CONST_IB category↑ value↑ → TRUE**
   - { CONST_IB.category=BOOL ;}

10. **CONST_IB category↑ value↑ → FALSE**
    - { CONST_IB.category=BOOL ;}

11. **LISTE_IDF envin↓ idflist↑ correct↑ → idf name↑**
    - { if (idf.name ∉ LISTE_IDF.envin) then
      - LISTE_IDF.correct := true
      - LISTE_IDF.idflist := {idf.name}
    - else
      - LISTE_IDF.correct := false }

12. **LISTE_IDF_1 envin↓ idflist↑ correct↑ → LISTE_IDF_2 envin↓ idflist↑ correct↑ idf name↑**
    - { LISTE_IDF_2.envin := LISTE_IDF_1.envin
    - if (($\text{idf.name} \notin \text{LISTE\_IDF}_1\text{.envin}$) AND ($\text{idf.name} \notin \text{LISTE\_IDF}_2\text{.idfllist}$)) then
      - LISTE_IDF_1.idflist := LISTE_IDF_2.idflist ∪ {idf.name}
      - LISTE_IDF_1.correct := LISTE_IDF_2.correct
    - else
      - LISTE_IDF.correct := false }
      - LISTE_IDF_1.idflist := LISTE_IDF_2.idflist

Ex2 – Q1-a: solution 2

1. PROG → LISTE_DECL LISTE_INST
    1. LD.envin := ∅; LI.envin := LD.envout
    2. P.corect := LD.correct
2. LISTE_DECL → DECL
    1. D.envin := LD.envin; LD.envout:= D.envout
    2. LD.corect := D.correct
3. LISTE_DECL → LISTE_DECL DECL
    1. LD_1.envin := LD_0.envin; D.envin := LD_1.envout;
    2. LD_0.envout := D.envout
    3. LD_0.correct := LD_1.correct AND D.correct
4. DECL → **idf** TYPE CONST_IB
    1. Si (idf not in D.envin)
        1. D.envout := D.envin union idf; D.correct := true
    2. Sinon D.envout := D.envin ; D.correct := false
5. DECL → LISTE_IDF TYPE
        1. LI.envin := D.envin; D.envout := LI.envout; D.correct := LI.correct
6. LISTE_IDF → **idf**
    1. Si (idf not in LI.envin)
        1. LI.envout := LI.envin union idf; D.correct := true
    2. Sinon LI.envout := LI.envin ; LI.correct := false
7. $LISTE\_IDF_0$ → $LISTE\_IDF_1$ **idf**
    1. **LI1.envin := LI0.envin**
    2. Si (idf not in LI1.envout)
        1. LI1.envout := LI1.envout union idf; D.correct := true
    3. Sinon LI.correct := false
    4. LI0.envout := LI1.envout

Ex2 – Q1-b: solution 2 (suite)

1. DECL → **idf** TYPE CONST_IB
    1. if (TYPE.name == CONST_IB.category)
        1. correct := true
    2. else correct := false
2. TYPE → **INT**
    1. TYPE.name = INT
3. TYPE → **BOOL**
    1. TYPE.name = BOOL
4. CONST_IB → **const**
    1. CONST_IB.category = INT
5. CONST_IB → **TRUE**
    1. CONST_IB.category = BOOL
6. CONST_IB → **FALSE**
    1. CONST_IB.category = BOOL

Il faut choisir quelle est la priorité de l'erreur ! Si idf n'est pas unique, on regarde ou pas sa déclaration ?

Ex3 : Analyse sémantique (2 points) – ENSIMAG Sep. 2001

- Une expression de la forme «$Exp_1\ op\ Exp_2$», où *op* est l'un des opérateurs +, , ou *, peut être évaluée soit de gauche à droite, soit de droite à gauche.
- Q1- Donner un exemple d'expression qui donne un résultat différent suivant l'ordre d'évaluation choisi

- Q2- Proposer un calcul d'attributs sur les règles de la grammaire G permettant de savoir si une expression réalise ou non un effet de bord.

Ex3 : Analyse sémantique (4 points) – ENSIMAG Sep. 2001 (suite)

- Q3- Proposer un calcul d'attributs sur les règles de la grammaire G permettant de savoir si une expression réalise ou non un effet de bord :
  - G :< T= {idf,num,=,(,),*,-,+,',',}, NT={Exp},start=Exp, P={R_1,R_2,R_3,R_4,R_5,R_6,R_7,R_8}>
    - R_1 : Exp $\rightarrow$ idf = Exp
    - R_2 : Exp $\rightarrow$ Exp , Exp
    - R_3 : Exp $\rightarrow$ (Exp)
    - R_4 : Exp $\rightarrow$ Exp + Exp
    - R_5 : Exp $\rightarrow$ Exp - Exp
    - R_6 : Exp $\rightarrow$ Exp * Exp
    - R_7 : Exp $\rightarrow$ idf
    - R_8 : Exp $\rightarrow$ num

Compilation

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Lectures recommandées - livres

1. « Engineering a Compiler » ©, K. D. Cooper & L. Torczon, Morgan Kaufmann 2004
2. « Introduction to Automata Theory Languages and Computation » ©, J. E. Hopcroft, R. Motwani, J. D.Ullman, 2001, Addison Wesley
3. « Compilers and Compiler Generators » ©, P.D. Terry, 2000

Plan du cours

1. Module initial Motivation & Définitions
2. Module 1 Analyseur lexical *(langages réguliers et théorie des automates)*
3. Module 2 Analyseur syntaxique *(langages hors contexte)*
4. Module 3 Analyseur sémantique *(grammaires attribuées)*
5. **Module 4** Représentations Intermédiaires
6. Module 5 Génération de code intermédiaire
7. Module 6 Optimisation et Génération de code
8. Conclusion et perspectives

Module 4
Représentations Intermédiaires

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

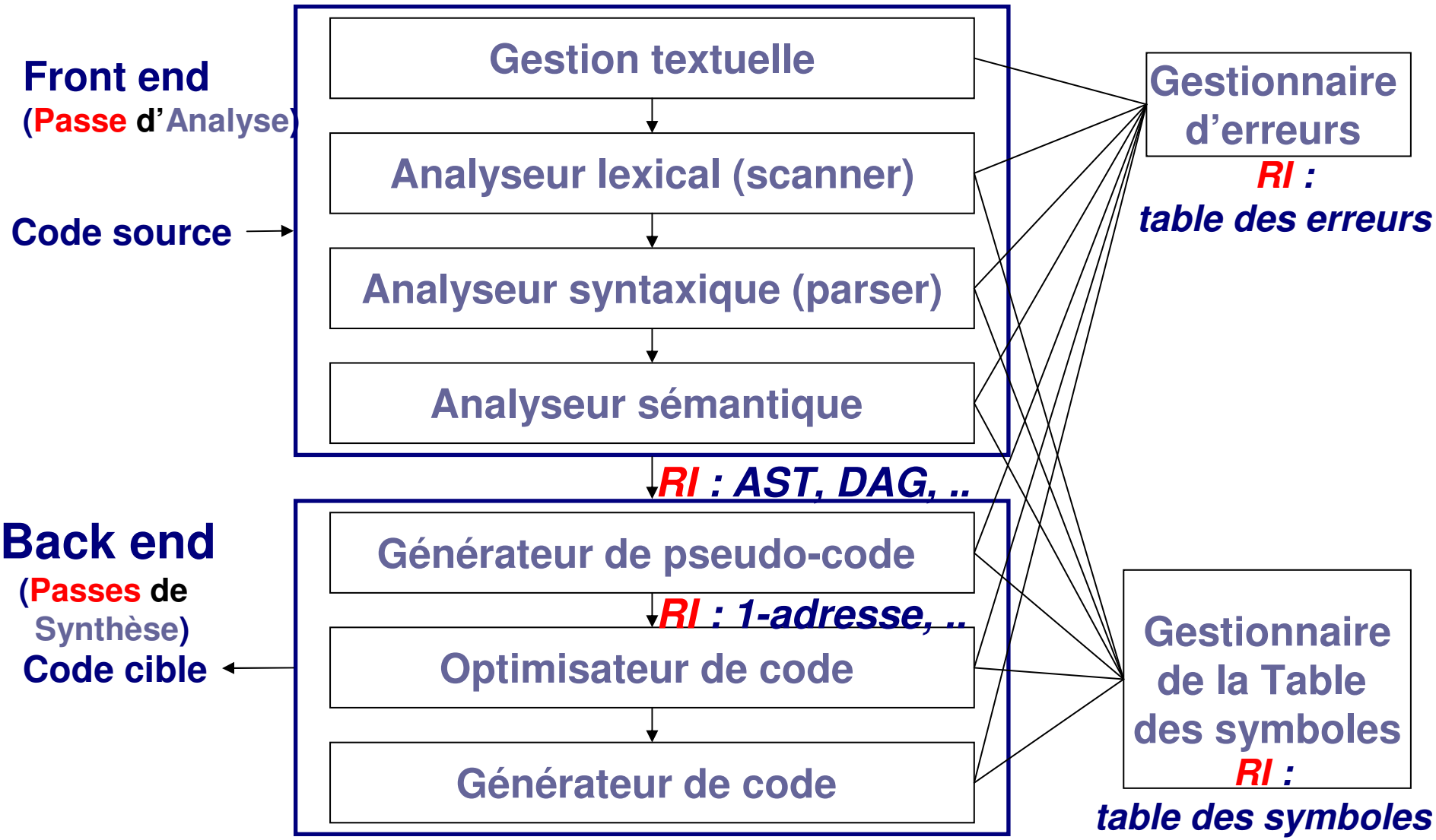
Représentations intermédiaires – Plan

- Introduction
- Concepts
  - Types de RI (graphiques, linéaires, hybrides)
- RI Graphiques
  - ST, AST, DAG, CFG
- RI Remarquables (Applications)
  - Tables des symbôles, CFG des instructions, Gestionnaire des symbole
- Construction de RI et C et utilisations sous bison
- Application complètes des RI :
  - Réalisation de l'interpréteur du langage ZZ

Représentations intermédiaires

- Lors de chacune de ses passes (parcours du code ou d'une de ses transformations), le compilateur consomme une structure qui représente le code d'entrée et en produit une nouvelle représentation dans un nouveau format interne
- On parle de représentation intermédiaire (ou *RI*)

Passes d'un compilateur



Représentations intermédiaires - Caractéristiques

1. **Modèle** de la structure de la RI
   - la clarté, la lisibilité, l'expressivité de la RI dépend de son modèle

2. **Niveau d'abstraction** de la RI
   - des propriétés/transformations de la RI peuvent être mesurées/effectuées si le niveau d'abstraction s'y prête

3. **Coût** des opérations et des transformations sur la RI
   - La faisabilité d'une transformation sur la RI est pondérée par sa rapidité et son coût mémoire

Représentations intermédiaires - Modèles

- RI Graphiques
  - **RI encodant la connaissance (d'un niveau haut : high level) du compilateur sous forme de graphe. Les algorithmes pour ce type de RI sont exprimés en termes des nodes et des arcs, en termes de listes et d'arbres**
    - Exemples : arbres syntaxiques, arbres abstraits, DAG

- RI Linéaires (pseudo-code) *(voir chapitre génération de pc)*
  - **RI assemblant la connaissance (d'un niveau bas : low level) du compilateur sous forme le pseudo-code pour une machine virtuelle/abstraite particulière. Code simple, compacte et facile à réarranger. Les algorithmes pour ce type de RI opèrent sur de simples séquences linéaires d'opérations**
    - Exemples : bytecode, three-address code

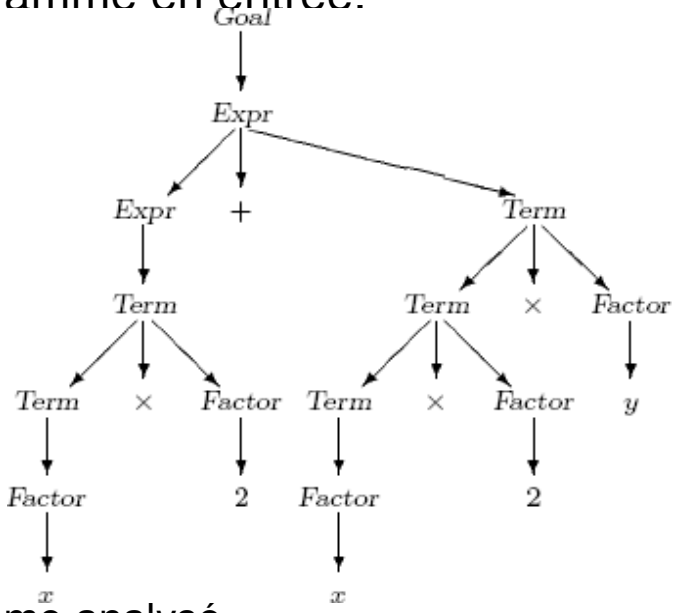
- RI Hybrides
  - **RI combinant des éléments des modèles graphiques et linéaires des structures des IR, dans l'objectif de profiter de leur forces mutuelles. Les RI hybrides utilisent un code linéaire bas-niveau pour représenter les blocks de code et un graphe pour représenter le flot de contrôle entre ces blocks**
    - Exemples : Control Flow Graph (flot de contrôle)

RI Graphiques - *Arbre syntaxique*
(Parse Tree)

- **L'arbre syntaxique** est une représentation graphique de la dérivation, (i.e. l'analyse syntaxique), qui correspond au programme en entrée.

$$
\begin{array}{lcl}
Goal & \rightarrow & Expr \\
Expr & \rightarrow & Expr + Term \\
 & | & Expr - Term \\
 & | & Term \\
Term & \rightarrow & Term \times Factor \\
 & | & Term \div Factor \\
 & | & Factor \\
Factor & \rightarrow & \texttt{Number} \\
 & | & \texttt{Id}
\end{array}
$$

produit



- **avantages**
  - Représentation fidèle de la syntaxe du programme analysé

- **inconvénients**
  - Assez grand (en nombre de noeud), assez profond relativement aux informations qu'il contient ⇒ coût en mémoire de stockage et coût en temps de traitement élevés
  - La séquentialité, L'alternative, Le parallélisme ne sont pas capturées par cette RI

RI Graphiques - ***Arbre syntaxique*** **(Parse Tree)** – Exercice b2-4ac

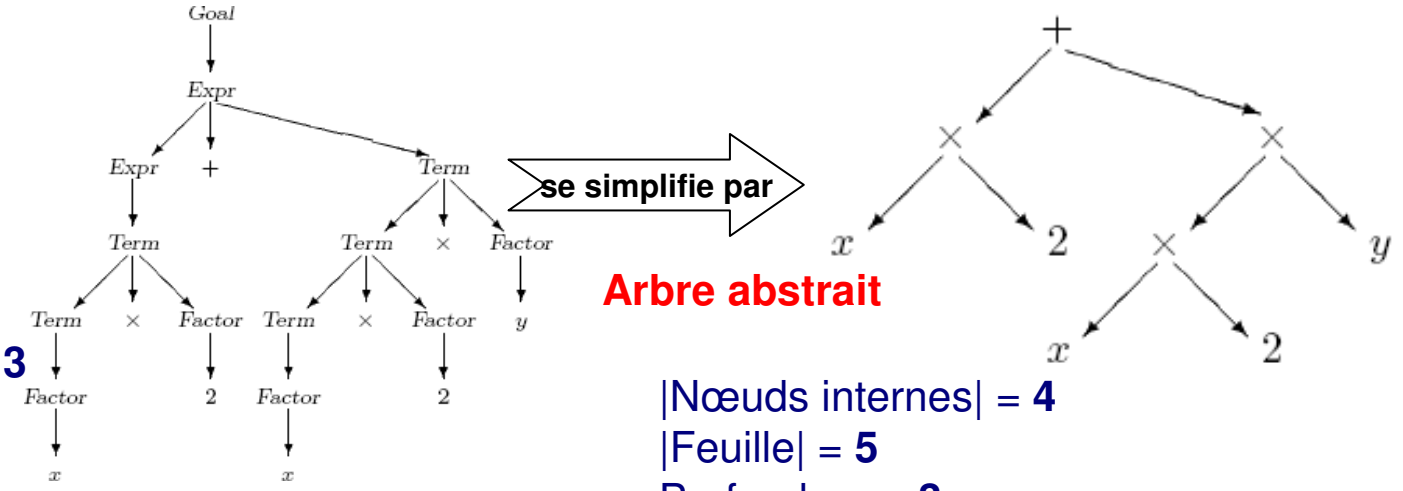
- Soit l'expression arithmétique suivante « b*b – 4*a*c »
  a. Quelle est l'arbre syntaxique généré par la grammaire suivante ?

Goal	→	*Expr*
Expr	→	*Expr* + *Term*
	|	*Expr* − *Term*
	|	*Term*
Term	→	*Term* × *Factor*
	|	*Term* ÷ *Factor*
	|	*Factor*
Factor	→	`Number`
	|	`Id`

RI Graphiques - *Arbre abstrait*

(AST – Abstract Syntax Tree)

- Un **arbre abstrait** (AST) représente les **éléments essentiels de l'arbre syntaxique** et se débarrasse des noeuds internes sans aucune information complémentaire



Arbre syntaxique

|Nœuds internes| = **13**
|Feuille| = **9**
Profondeur = **6**

Arbre abstrait

|Nœuds internes| = **4**
|Feuille| = **5**
Profondeur = **3**

- **Avantages**
  - Meilleurs lisibilité, coût de stockage et coût de traitement !
  - Il est préférable d'opter pour les arbres abstraits !
- **Inconvénients**
  - La séquentialité, L'alternative, Le parallélisme ne sont pas capturées par cette RI
  - Représentation aveugle : ne détecte pas les représentations redondantes

RI Graphiques - ***Arbre abstrait*** (AST – Abstract Syntax Tree) – Exercice b^2-4ac (suite)

b. Quelle est l'arbre abstrait correspondant à la même expression « b*b – 4*a*c » ?

Goal	→	*Expr*
Expr	→	*Expr* + *Term*
	|	*Expr* − *Term*
	|	*Term*
Term	→	*Term* × *Factor*
	|	*Term* ÷ *Factor*
	|	*Factor*
Factor	→	`Number`
	|	`Id`

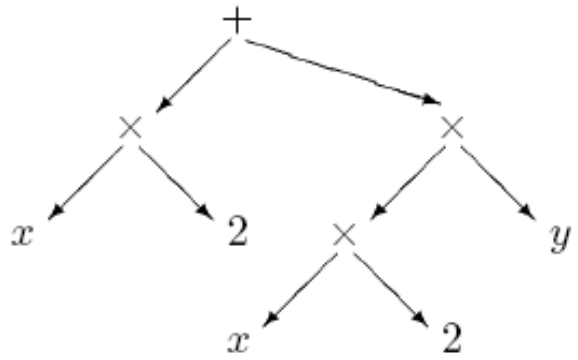
Le type abstrait de stockage et arbre abstrait - Exercice

- Soit la grammaire suivante
  - S → Exp
  - Exp → Exp '+' Exp |
  - Exp '-' Exp |
  - Exp '*' Exp |
  - '(' Exp ')' |
  - NOMBRE

- Construire les cfg nécessaires pour stocker les arbres syntaxiques abstraits dérivés par G

Le type abstrait de stockage et arbre abstrait – Solution (AST.h)

```
typedef enum {NB=0, OP=1, IDF = 2} Type_Exp ;

typedef enum {plus, moins, mult} Type_Op;

typedef enum {false, true} boolean;

struct Exp ; /* pré déclaration de la structure de stockage d'une expression */

typedef struct Exp * AST;

typedef union {
  double nombre ;
  char *idf;
  struct {
    Type_Op top;
    AST expression_gauche ;
    AST expression_droite ;
  } op;
} valueType;

struct Exp {
  Type_Exp type ;
  valueType noeud ;
};
```



L'arbre va stocker tous les attributs calculés sur le mot de la grammaire !!
Afin de pouvoir effectuer d'autres passes (re-parcours de l'arbre et des ses attributs)

(AST.h) suite

- // précondition : a<> NULL and est_feuille(a) == false
- AST **arbre_gauche**(AST a);

- // précondition : a<> NULL and est_feuille(a) == false
- AST **arbre_droit**(AST a);

- // précondition : a<> NULL and est_feuille(a) == false
- Type_Op **top**(AST a);

- // précondition : a<> NULL
- boolean **est_feuille**(AST a);

- AST **creer_feuille_nombre**(double n);

- // op in {'+', '-, '*'} and arbre_g <> NULL and arbre_d <> NULL
- AST **creer_noeud_operation**(char op, AST arbre_g, AST arbre_d);

- // affichage par parcours infixé de l'arbre abstrait
- // précondition : a<> NULL
- void **afficher_infixe_arbre(**AST ast);

- // affichage par parcours postfixé de l'arbre abstrait
- // précondition : a<> NULL
- void **afficher_postfixe_arbre(**AST ast);

- // évaluation par parcours postfixé de l'arbre abstrait
- // précondition : a<> NULL
- double **evaluer**(AST ast);

(AST.c)

```
#include "AST.h"

// précondition : a<> NULL and est_feuille(a) == false
AST arbre_gauche(AST a){
  return a->noeud.op.expression_gauche;
}

// précondition : a<> NULL and est_feuille(a) == false
AST arbre_droit(AST a){
  return a->noeud.op.expression_droite;
}

// précondition : a<> NULL and est_feuille(a) == false
Type_Op top(AST a){
  return a->noeud.op.top;
}

// précondition : a<> NULL
boolean est_feuille(AST a){
  return (a->type != OP) ;
}

AST creer_feuille_nombre(double n){
  AST result = (AST) malloc (sizeof(struct Exp));
  result->type=NB;
  result->noeud.nombre = n;
  return result;
}
```

(AST.c) suite

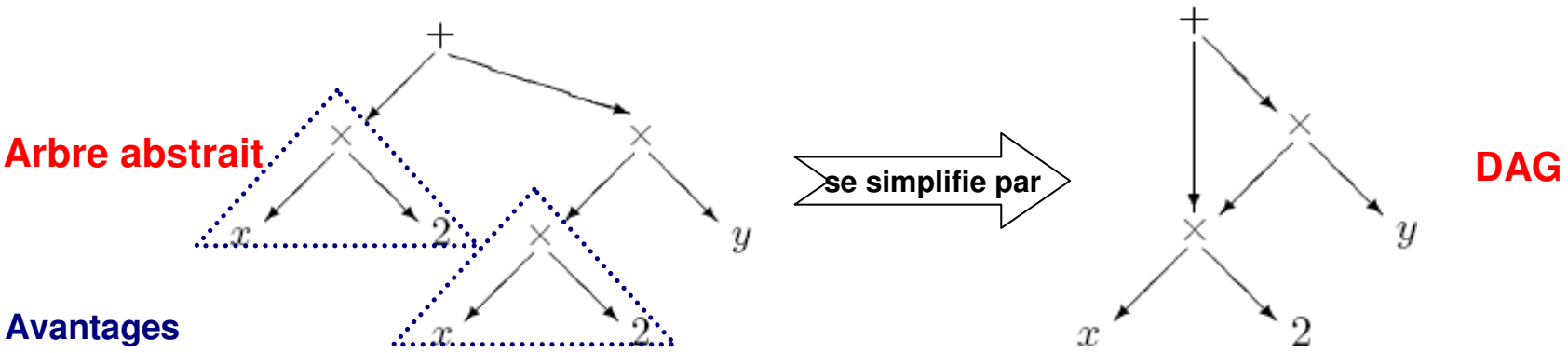
```
AST creer_noeud_operation(char op, AST arbre_g, AST arbre_d){
  AST result= (AST) malloc (sizeof(struct Exp));
  result->type=OP;
  result->noeud.op.top = ((op=='+')?plus:((op=='-')?moins:mult));
  result->noeud.op.expression_gauche = arbre_g;
  result->noeud.op.expression_droite = arbre_d;
  return result;
}

// affichage par parcours infixé de l'arbre abstrait
// précondition : a<> NULL
void afficher_infixe_arbre (AST ast){
  // if (est_feuille(ast)){
  switch(ast->type) {
  case NB : printf(" %lf",ast->noeud.nombre); break;
  case IDF :  printf(" %lf",value(ast->noeud.idf)); break;
  case OP :
    afficher_infixe_arbre(arbre_gauche(ast));
    switch(ast->noeud.op.top){
    case plus : printf(" + "); break;
    case moins : printf(" - "); break;
    case mult : printf(" * "); break;
    }
    afficher_infixe_arbre(arbre_droit(ast));
    break;
  }
}
```

```
// évalation par parcours postfixé de l'arbre abstrait
// précondition : a<> NULL
double evaluer(AST ast){
  double valg, vald;
  //if (est_feuille(ast)){
  switch(ast->type) {
  case NB : return ast->noeud.nombre; break;
  case IDF : return value(ast->noeud.idf); break;
  case OP :
    valg = evaluer(arbre_gauche(ast));
    vald = evaluer(arbre_droit(ast));
    switch(ast->noeud.op.top){
    case plus : return valg + vald; break;
    case moins : return valg - vald; break;
    case mult : return valg * vald; break;
    }
    break;
  }
}
```

```
// affichage par parcours postfixé de l'arbre abstrait
// précondition : a<> NULL
void afficher_postfixe_arbre (AST ast){
  // if (est_feuille(ast)){
  switch(ast->type) {
  case NB : printf(" %lf",ast->noeud.nombre); break;
  case IDF :  printf(" %lf",value(ast->noeud.idf)); break;
  case OP :
    afficher_postfixe_arbre(arbre_gauche(ast));
    afficher_postfixe_arbre(arbre_droit(ast));
    switch(ast->noeud.op.top){
    case plus : printf(" + "); break;
    case moins : printf(" - "); break;
    case mult : printf(" * "); break;
    }
    break;
  }
}
```

RI Graphiques – *Graphes Orientés sans circuit*(*) (DAG – Directed Acyclic Graph)

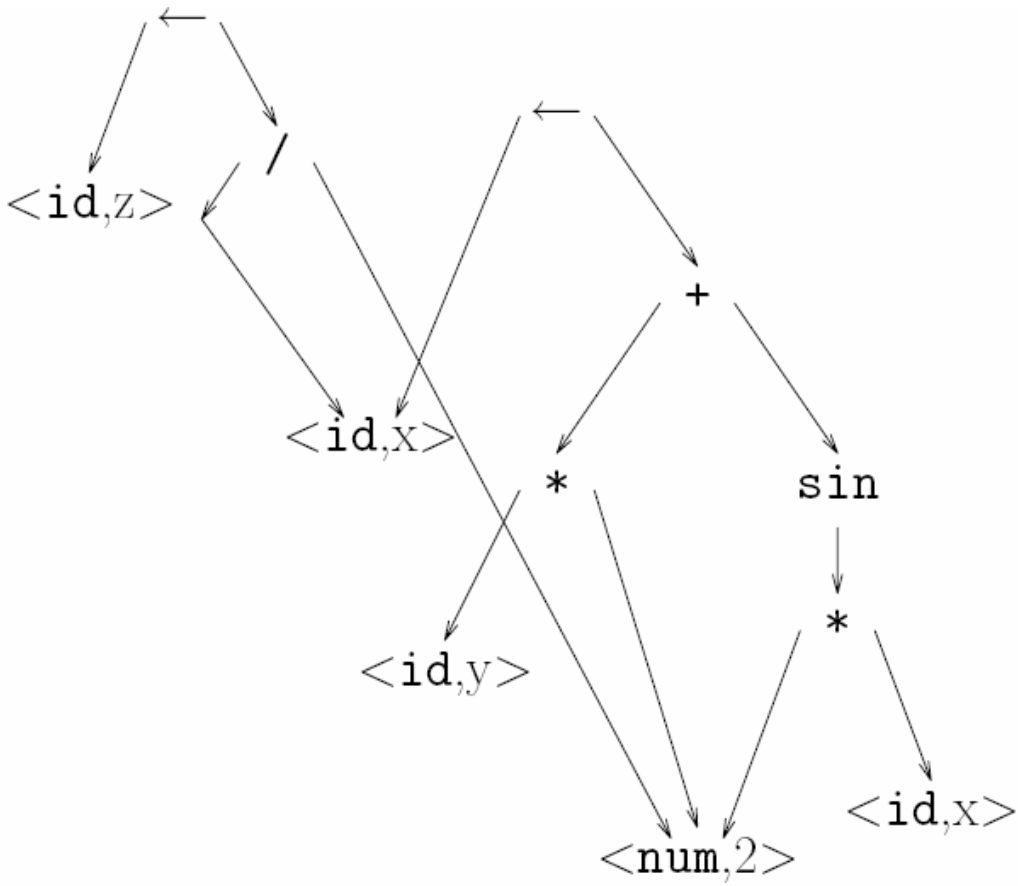
- Un **graphe orienté sans circuit** (DAG) est une contraction de l'arbre abstrait. Il élimine les redondances non nécessaires : en représentant une et une seule fois les noeuds et arcs identiques
  - Dans un graphe orienté sans circuit, les noeuds peuvent avoir des parents multiples
- Éliminer la redondance ⇒ Éliminer la génération multiple de code (code plus compacte et plus rapide)



- **Avantages**
  - Représentation plus compacte que les arbres abstraits
- **Inconvénients**
  - La difficulté de calcul (ie de la construction) du DAG

(*) **sans circuit** et pas **acyclique**

RI Graphiques – *Graphes Orientés sans circuit*(*) (DAG – Directed Acyclic Graph) - Exemple




```
x ← 2 * y + sin(2*x)
z ← x / 2
```

Les identités d'arbres

- Identité de référence
  - A1(op1, AG1,AD2) ≡ A1(op1, AG1,AD2)
    - Ssi @ l'emplacement de A1 = @ l'emplacement de A2
- Identité structurelle
  - A1(op1, AG1,AD2) ≡ A2(op2, AG2,AD2)
    - Ssi op1 ≡ op2 and AG1 ≡ AG2 and AD1 ≡ AD2
  - A1(N1) ≡ A2(N2)
    - Ssi N1 ≡ N2
- Identité de valeur
  - A1(op1, AG1,AD2) ≡ A1(op1, AG1,AD2)
    - Ssi evaluer(A1) = evaluer(A2)

Exercice

- Donner des idées algorithmiques et de structures de données efficaces pour construire un DAG à partir d'un programme analysé

Solution

- Prédicat d'identité structurelle :
  - est-identique : Arbre $\times$ Arbre $\rightarrow$ Booléen
- Fonction de hachage :
  - H : Arbre $\rightarrow$ {1..K} (md5)
- Structure de stockage de l'ensemble des arbres déjà crées :
  - Table de hachage (on suppose une distribution uniforme)
- Calcul de hachage en O(1), Recherche parmi les arbres de même clef en O(N/K), N étant le nombre d'arbres et K le nombre de clés (pour md5 K = N)
- Si H(A) appartient à la table de hachage
  - On ne recrée pas l'arbre A, mais on pointe dessus
- Sinon
  - On le crée et on stocke sa référence dans la table de hachage

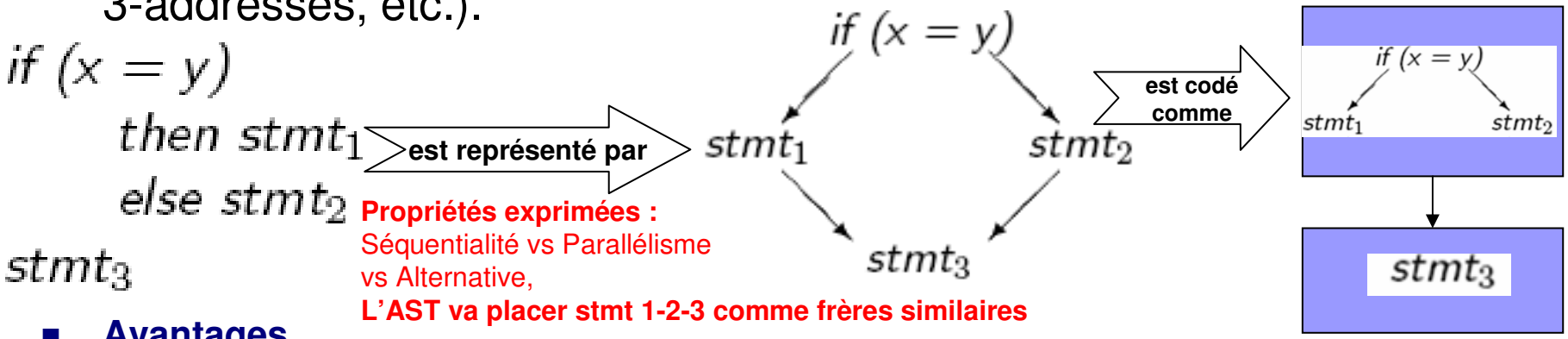
RI Graphiques – *Graphes Orientés sans circuit*(*) (DAG – Directed Acyclic Graph) – Exercice b^2-4ac (suite)

c. Quelle est le graphe orienté sans circuit correspondant à la même expression « b*b – 4*a*c » ?

$$
\begin{array}{lcl}
\mathit{Goal} & \rightarrow & \mathit{Expr} \\
\mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
 & | & \mathit{Expr} - \mathit{Term} \\
 & | & \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term} \times \mathit{Factor} \\
 & | & \mathit{Term} \div \mathit{Factor} \\
 & | & \mathit{Factor} \\
\mathit{Factor} & \rightarrow & \texttt{Number} \\
 & | & \texttt{Id}
\end{array}
$$

RI Hybrides – *Graphes de flot de contrôles* (Control Flow Graph)

- Un **graphe de flot de contrôle** modélise la manière dont les contrôles sont passés entre les blocks de codes. Les noeuds correspondent à des blocks basic et les arcs correspondent aux passages de contrôles entre les blocks.
- Les graphes de flot de contrôle représentent les relations entre blocks tandis que les opérations de chaque block sont représentées selon une autre RI (e.g. graphique : AST, DAG, ou linéaire : code à 3-addresses, etc.).



Propriétés exprimées :
Séquentialité vs Parallélisme vs Alternative,
L'AST va placer stmt 1-2-3 comme frères similaires

- **Avantages**
  - Représentation graphique propre des possibilités des flot de contrôle pendant l'exécution (run-time)

- *D'autres représentations intermédiaires graphiques riches existent !*

RI Hybrides – *Graphes de flot de contrôles*

(Control Flow Graph) –

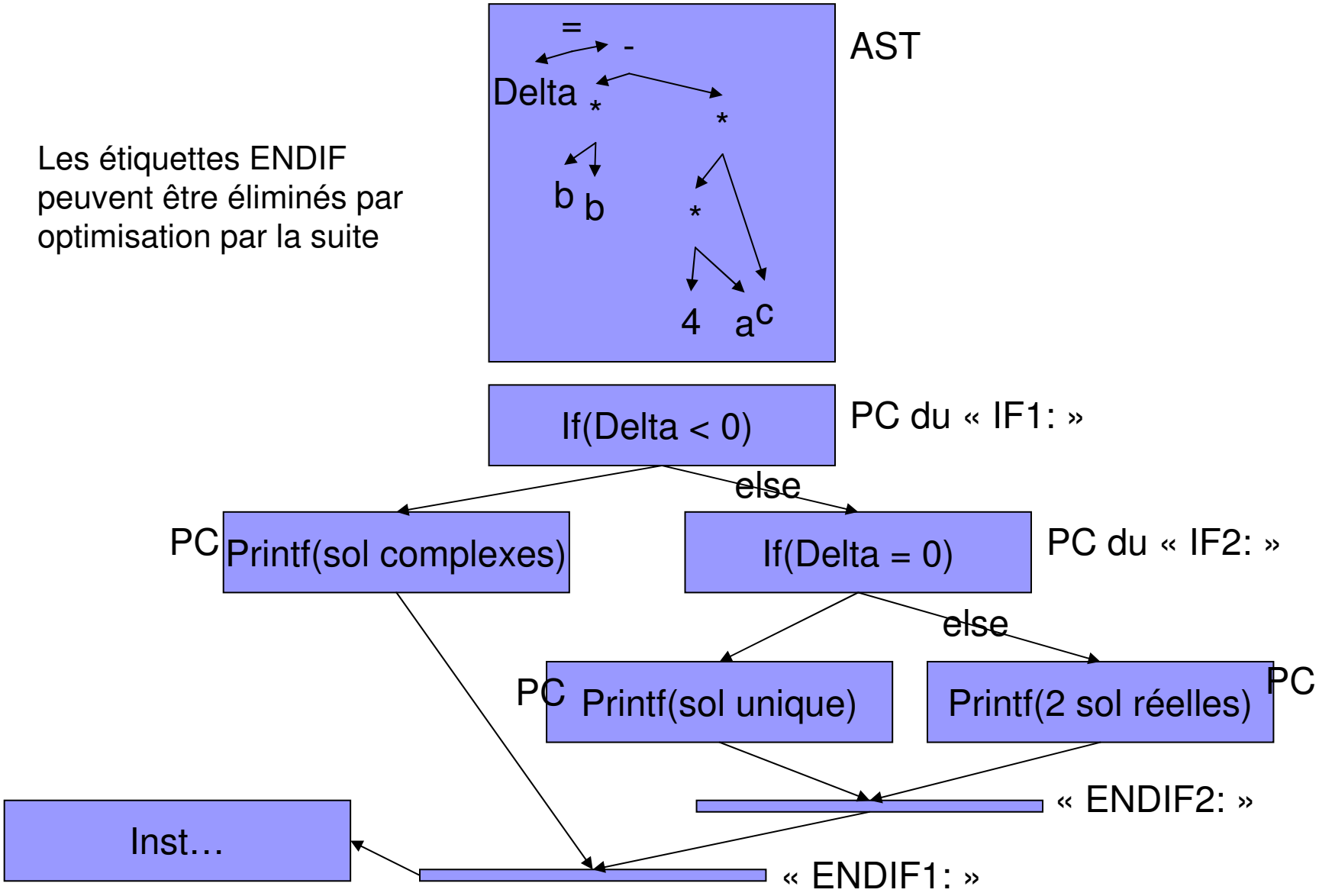
Exercice b^2-4ac (suite)

d. Quelle est le graphe de flot de contrôles correspondant aux instructions suivantes :

- Delta = b*b – 4*a*c
- if (Delta < 0) then
  - printf(solutions complexes)
- else
  - if (Delta == 0) then
    - printf(une seule solution)
  - else
    - printf(deux solutions réelles)
- Inst;

Solution (CFG)

- Les étiquettes ENDIF peuvent être éliminés par optimisation par la suite



Des concepts à la pratique - Introduction à la construction de représentations intermédiaires avec BISON

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

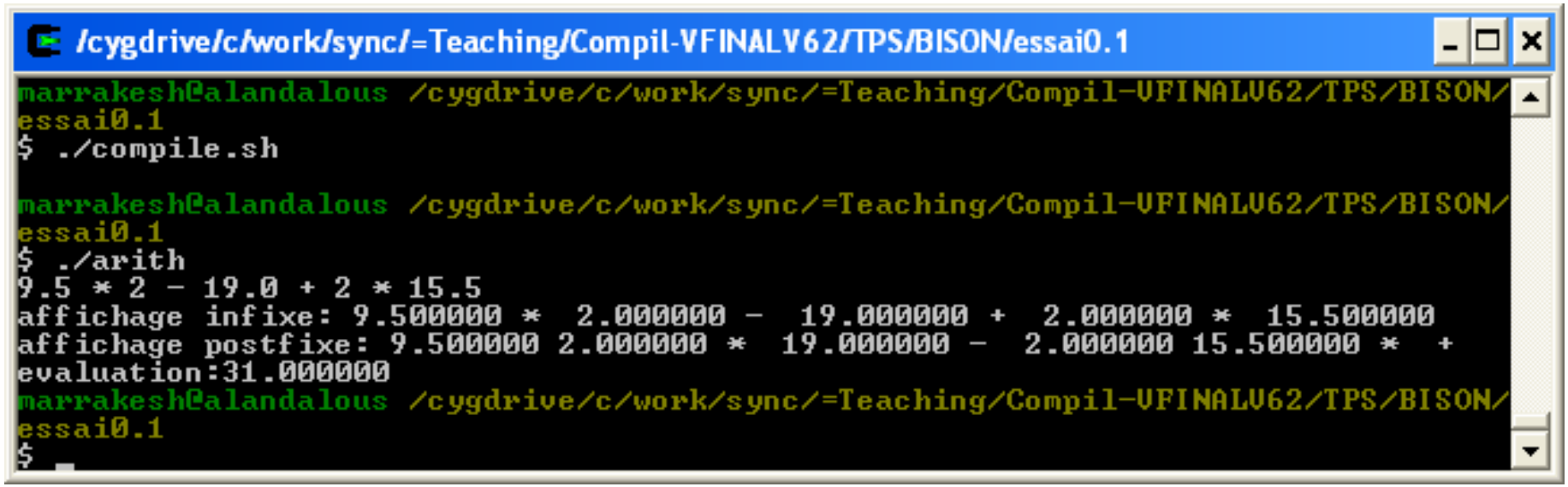
(arith-with-AST.y)

```
%{
#include <stdio.h>
#include "AST.h"

// les $$ et $i contiendront des arbres syntaxiques abstraits
#define YYSTYPE AST

  int yylex(void);
  void yyerror (char const *s);
  double result ;
  AST ast;

  %}

%start S                        /* Axiome : le start est le non-terminal S */

%token NOMBRE      /* Définitions des tokens (terminaux) */

%left '-' '+'                   /* Désambiguisation : '+' et '-' sont associatifs gauches de même priorité */
%left '*'                 /*          '*' est associatif gauche et est plus prioritaire que le '+' et le '-' */

%%
S :
Exp                             { $$ = ast = $1;}
;

Exp :
Exp '+' Exp                     { $$ = creer_noeud_operation('+', $1, $3); }          |
Exp '-' Exp                     { $$ = creer_noeud_operation('-', $1, $3); }          |
Exp '*' Exp                     { $$ = creer_noeud_operation('*', $1, $3); }          |
'(' Exp ')'                      { $$ = $2 ; }                                        |
NOMBRE                          {$$ = creer_feuille_nombre($1->noeud.nombre) ; }
;
%%
```

```
int main(int argc, char** argv){
  int retour = yyparse();
 printf("affichage infixe:");  afficher_infixe_arbre(ast);
  printf("\naffichage postfixe:");  afficher_postfixe_arbre(ast);
  printf("\nevaluation:");  result = evaluer(ast);
  printf("%lf", result);
  return retour ;
}

void yyerror (char const *s) {     printf ("%s\n", s) ;     }

int yylex(){
char c = getchar();

 if (c== '\n') return 0;
 if ((c==' ') || (c=='\t')) return yylex();
 if (c == '+') return '+';
 if (c == '-') return '-';
 if (c == '*') return '*';
 if (isdigit(c)) {
   ungetc (c, stdin);
   yylval = (AST) malloc(sizeof(struct Exp));
   scanf ("%lf", &(yylval->noeud.nombre));
   return NOMBRE ;
 }
}
```

Compile.sh

- #!/bin/sh
- bison arith-with-AST.y
- gcc AST.h AST.c arith-with-AST.tab.c -o arith


/cygdrive/c/work/sync/=Teaching/Compil-VFINALV62/TPS/BISON/essai0.1
marrakesh@alandalous /cygdrive/c/work/sync/=Teaching/Compil-VFINALV62/TPS/BISON/essai0.1
$ ./compile.sh
marrakesh@alandalous /cygdrive/c/work/sync/=Teaching/Compil-VFINALV62/TPS/BISON/essai0.1
$ ./arith
9.5 * 2 - 19.0 + 2 * 15.5
affichage infixe: 9.500000 * 2.000000 - 19.000000 + 2.000000 * 15.500000
affichage postfixe: 9.500000 2.000000 * 19.000000 - 2.000000 15.500000 * +
evaluation:31.000000
marrakesh@alandalous /cygdrive/c/work/sync/=Teaching/Compil-VFINALV62/TPS/BISON/essai0.1
$

Représentations intermédiaires remarquables

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

RI remarquables : Table des symboles

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Rôle de la table des symboles

1. Stocker et Réserver les mots clefs du langages (mots réservés)
    1. Interdire leur redéfinition en variables ou fonctions..
2. Stocker les IDF
    1. Interdire la duplication des déclarations
3. Stocker les cfg des variables/fonctions
    1. Vérifier la compatibilité opération / type des variables
    2. Vérifier l'arité des tableaux et des focntions
    3. Vérifier les retours des fonctions
4. Stocker les valeurs initiales des variables
    1. Vérifier la possibilité d'effectuer des calculs
5. Imbriquer les environnements (notion de Portée : scope)
    1. Permettre la gestion de la visibilité des variables (les variables locales masquent les variables globales)

- **Plusieurs applications informatiques**
    - En base de données : Dictionnaire des données (tables, champs, clefs, contraintes, etc.)
    - En compilation : Le compilateur maintient une table des symboles, qu'il consulte pour vérifier les noms et les cfg des variables
- **Alias :** dictionnaire des données, environnement des variables, contexte de typage…

Table de symboles à portée simple

- **Flux de tokens lexicaux :**
  - WHILE A > 3 * B DO A := A - 1 END

- **Table de symboles résultante**

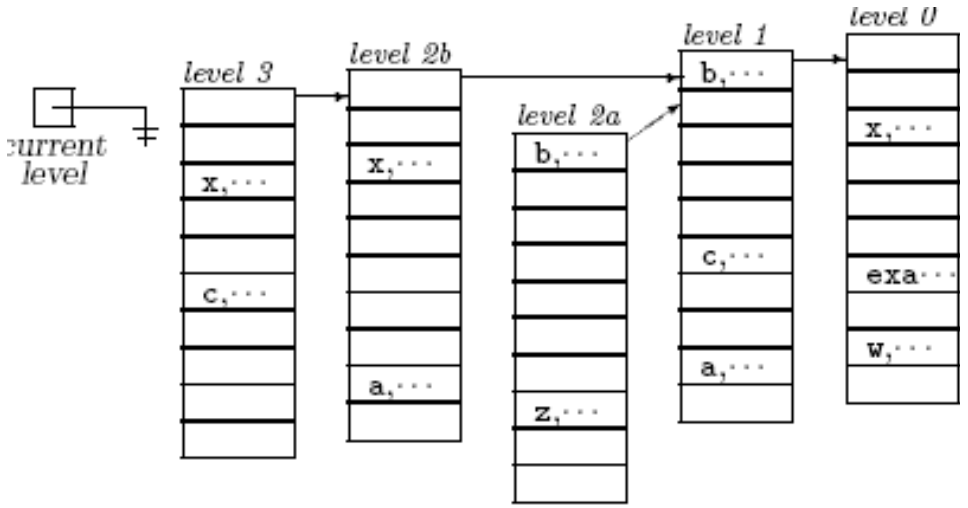
Choix de Stockage dans TS

Numéro de symbole	*Token*	*Type du Token*	*Propriété*
10	**A**	*Identifier*	*name A*
11	**B**	*identifier*	*name B*
…			
100	**>**	*Operator*	*Comparison*
101	**:=**	*Operator*	*Assignment*
102	**-**	*Operator*	*Substraction*
103	*	*operator*	*multiplication*
…			
1000	**WHILE**	*Keyword*	
1001	**DO**	*Keyword*	
1002	**END**	*Keyword*	
…			
5000	**1**	*constant literal*	*value 1*
5002	**3**	*constant literal*	*value 3*

- **L'instruction peut être recodées :** 1000, 10, 100, 5002, 1001, 10, 101, 10, 102, 5000, 1002

Table de symboles à portées imbriquées (nested scopes)

```
static int w;       /* level 0 */
int x;

void example(a,b);
  int a, b;         /* level 1 */
{
  int c;
  {
    int b, z;       /* level 2a */
      ...
  }
  {
    int a, x;       /* level 2b */
      ...
      {
        int c, x; /* level 3 */
        b = a + b + c + x;
      }
  }
}
```



Level	*Names*
0	w, x, example
1	a, b, c
2a	b, z
2b	a, x
3	c, x

Implémentation 1 - Tableaux d'accès direct

- Le symbole (l'enregistrement) x est stocké à l'emplacement T[key(x)]
- Il y a un emplacement par symbole : key(x) : définit la clef unique de x, l'ensemble U des clés uniques
  - exemple 1 : key (abonne(id,nom, prenom))=id
  - exemple 2 : key(var) = nbvar++; // avec nbvar le nombre de variables courant
- Soit card (U)=m, on utilise un tableau T[0..m-1] où chaque position correspond à une clé.
- En plus de la table des symboles, il faut gérer une table de correspondance x / key(x) : symbole(key(x))=x, key(symbole(k)))=k
- **Condition d'application :** l'ensemble U est *petit*

Tableaux d'accès direct

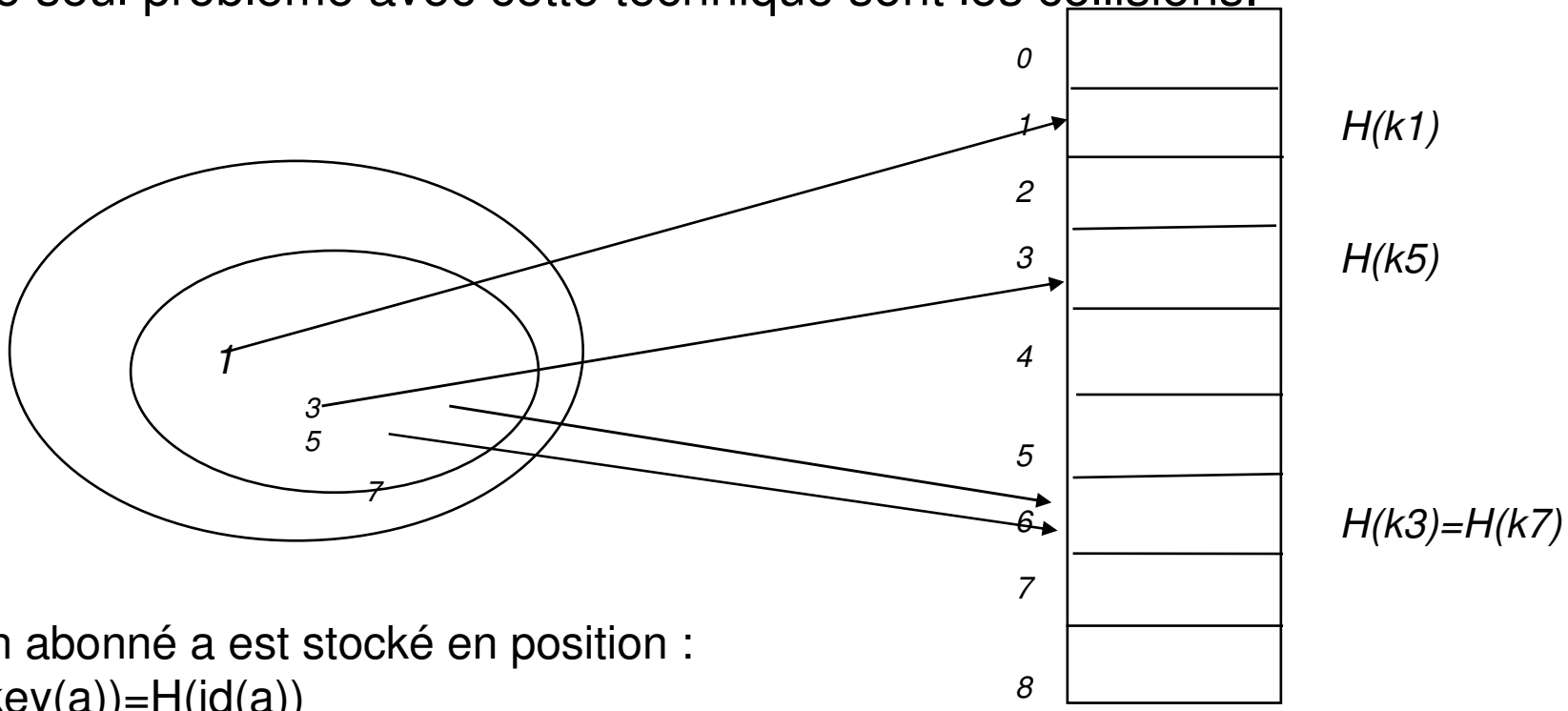
- Les opérations sont très simples et exécutées en temps constant O(1).
  - inTS(T,x) // encore faut – il résoudre key(x) =? k
    - return T[k=key(x)]
  - ajouter_symbole_a_TS(T,x)
    - T[k=key(x)] := x
  - Supprimer_symbole_de_TS(T,x)
    - T[k=key(x)] := NIL

Tableaux d'accès direct

- Désavantage de l'adresse directe
  - Si l'ensemble des clés possibles U est grand, le tableaux d'ccaès direct est grand : Problème de mémoire
  - Stocker/Calculer la correspondance entre x et key(x) peut être coûteux en mémoire/temps
  - Si l'ensemble des clés réellement utilisées est beaucoup plus petit que U, une grande partie de la mémoire est gaspillée.

Implémentation 1 - Tableaux d'hashage

- Avec l'adresse directe, un élément à clé *k* est placé en position *k*.
- Les hash tables (table de hashage) sont des structures des données très appropriées pour mettre en place un dictionnaire et le consulter. Avec hashage, il est stocké en position H(k).
- Le seul problème avec cette technique sont les collisions.



- Un abonné a est stocké en position : H(key(a))=H(id(a))

un emplacement par classe d'éléments (symboles)

Une table par niveau

Tableaux d'hashage

- **Collision :** lorsque deux éléments ayant clés différentes ont la même valeur de hashage.
  - key(enregitrement 1) = key(enregitrement 2)
- **Solution : Adressage chaîné (chaîning):**
  - les éléments sont associés à la même position dans une liste chaînée.
    - la complexité de recherche, insertion et effacement est proportionnelle à la longueur de cette liste.

Fonction de hashage

- Une bonne fonction d'hashage satisfait (approximativement) la présupposition de hashage uniforme (*loi de distribution uniforme*)
  - pour toutes positions, une clé a la même probabilité d'être associée à cette position
  - En général, une bonne technique consiste à développer une fonction indépendante de toutes régularités des distributions dans les données.

Exemples de Code de hash

- Hash : string $\rightarrow$ IN
- Hash(S) =
  - H1(S) = somme (Si)
    - Diverge
    - H1(S) = H1(permutation(S))
  - H2(S) = somme (Si * i)
    - Diverge
    - Mieux que H1 car H2(S) <> H2(permutation(S))
  - somme (Si * i) % N $\in$ [0 .. N-1]

- Techniques connues de Hashage :
  - Hashage par divisions, Hashage par multiplication, hashage aléatoire (universel), hashage pseudo-aléatoire ,double hashage

Fonction d'hashage

- Hashage par divisions: $H(k) = k \bmod m$
  - m est choisi tel que le résultat de la division dépend de tous les chiffres de k
  - m est premier, pas trop proche d'une puissance de 2 ou de 10
- Hashage par multiplication:
  - $H(k) = \lfloor m * (k * A - \lfloor k * A \rfloor) \rfloor$
  - en général, A dans]0,1[et $m = 2^p$
- Hashage universel = hashage aléatoire

Adressage ouvert (Open addressing)

- L'adressage d'une table de hashage est dit ouvert si les éléments du tableau sont les clés elles mêmes.
- Autre technique que *l'adressage chaîné* pour résoudre les collisions
- On n'utilise aucune structure externe, mais on trouve les cases disponibles par double hashage (H une fonction de sondage)
  - *H(k,0)=H1(k)*
  - *H(k,i) = (H1(k) + i*H2(k)) mod m*
- La méthode d'insertion consiste en cas de collision pour une clé k à sonder les places disponibles H(k,i) à partir de i=0 jusqu'à m-1. Si au bout de m sondages, aucune place disponible n'a été détectée, la table est dite **saturée**.

D’autres techniques d’implémentation de la table des symboles existent

- Des alternatives plus intéressantes que les hashtables existent en termes de la taille de la table et du coût en temps (dégradé) au pire des cas
  - Multiset discrimination

Le type abstrait de stockage et liste des déclarations - Exercice

- Soit la grammaire suivante
  - **LISTE_DECL** → DECL | LISTE_DECL DECL ;
  - **DECL** → idf TYPE CONST ;
  - **TYPE** → int | bool ;
  - **CONST** → nombre | true | false ;

- Construire les cfg nécessaires pour stocker les listes des déclarations dérivées par G

Le type abstrait de stockage et liste des déclarations – Solution (tableSymb.h)

- typedef enum {Int, Bool} **type**;

- typedef struct {
- char ***name**; // IDF.varattribute.name
- int **nbdecl**; // IDF.varattribute.nbdecl
- type **typevar**; // IDF.varattribute.typevar
- boolean **correct**; // IDF.correct (initialisation corecte)
- int **valinit**; // IDF.valinit
- int **linenumdecl**; // IDF.linenumdecl
- } **varvalueType**;

- typedef struct {
- type **typename**; // CONST.typename
- int **valinit**; // CONST.valinit
- } **constvalueType**;

- typedef union {
- varvalueType **varattribute**; // IDF.varattribute
- constvalueType **constattribute**; // CONST.constattribute
- type **typename**; // TYPE.typename
- } **valueType**;

- #define **YYSTYPE valueType**

Le type abstrait de stockage et liste des déclarations – Solution (tableSymb.h) suite

- // Affichage de la table des symbôles
- void **afficherTS**();

- // Recherche d'une variable et retour de son éventuel indice (clef) dans rangvar
- boolean **inTS**(char * varname, int* rangvar);

- // Ajouter une nouvelle variable dans la table des symbôles
- Précondiction : inTS(newvar.name, &i) == false
- void **ajouter_nouvelle_variable_a_TS**(varvalueType newvar);

Le type abstrait de stockage et liste des déclarations – Solution (tableSymb.c) suite

TS

name	nbdecl	typevar	correct	valinit	linenumdecl

```
#include "tableSymb.h"

#define NBS 100
static varvalueType TS[NBS];
static int NBVAR = 0;

// Affichage de la table des symbôles
void afficherTS(){
 int i=0;
  for (i=0; i<NBVAR; i++) {
    printf("variable %d = %s, de type %s,
    initialisee à %s, declaree %d fois\n",
    i,
    TS[i].name
    TS[i].typevar==Int?"int":"bool",
    (TS[i].typevar==Int?itoa(TS[i].valinit):
      (TS[i].valinit==true?"true":"false")),TS[i].nbdecl);
  }
}
```

```
// Recherche d'une variable
// et retour de son éventuel indice (clef) dans rangvar
boolean inTS(char * varname, int* rangvar){
 int i =0;
  while ((i < NBVAR) && (strcmp(TS[i].name,varname) != 0)) i++;
  if (i == NBVAR) return false;
  else { *rangvar = i; return true;}
}
```

```
// Ajouter une nouvelle variable dans la table des symbôles
void ajouter_nouvelle_variable_a_TS(varvalueType newvar){
   TS[NBVAR].nbdecl = newvar.nbdecl;
   TS[NBVAR].name = (char *)malloc(strlen(newvar.name));
   strcpy(TS[NBVAR].name,newvar.name);
   TS[NBVAR].linenumdecl = newvar.linenumdecl;
   TS[NBVAR].correct = newvar.correct;
   TS[NBVAR].typevar = newvar.typevar;
   TS[NBVAR].valinit = newvar.valinit;

   NBVAR++;
}
```

RI remarquables :
CFG des instructions

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Le type abstrait de stockage et liste des instructions - Exercice

- Soit la grammaire suivante
  - **LISTE_INST** → INST | LISTE_INST INST
  - **INST** →
    - idf ‘=‘ CONST |
    - PRINT idf |
    - if ‘(‘ idf “==“ CONST ‘)’ then LISTE_INST endif
  - **CONST** → nombre | true | false ;

- Construire les cfg nécessaires pour stocker les CFG des instructions dérivés par G

Le type abstrait de stockage et CFG – Solution (cfg.h)

```
typedef enum {Print, Assign, If} Type_INST ;

struct INST; // pré déclaration

struct LIST_INST; // pré déclaration

typedef struct INST {
  Type_INST typeinst;

  union  {
    struct  { // PRINT IDF
      int rangvar; // indice (clef) de l'idf, à afficher
    } printnode;

    struct  { // IDF = CONST
      int rangvar; // indice (clef) de l'idf lexp
      int right; // la valeur rexp
    } assignnode;

    struct  {  // IF '(' IDF "==" CONST ')' then LISTE_INST endif
      int rangvar;                    // indice (clef) de l'idf à comparer
      int right;                      // la valeur de comparaison
      struct LIST_INST * thenlinst; // then list of instructions
      struct LIST_INST * elselinst; // else list of instructions (non utilisé !!)
    } ifnode;

  } node;
} instvalueType;
```

```
typedef struct LIST_INST {
  struct INST first;
  struct LIST_INST * next;
} listinstvalueType;

typedef struct {
  type typename;        // CONST.typename
  int valinit;          // CONST.valinit
} constvalueType;

typedef union {
varvalueType varattribute;          // IDF.varattribute
  constvalueType constattribute;    // CONST.constattribute
instvalueType instattribute;        // INST.instattribute
  listinstvalueType listinstattribute;  // LIST_INST.listinstattribute
} valueType;

#define YYSTYPE valueType
```

Le type abstrait de stockage et CFG – Solution (cfg.h) suite

- // insère une instruction en queue de la liste des instructions
- void **inserer_inst_en_queue**(listinstvalueType * listinstattribute, instvalueType instattribute);

- // interprète une instruction
- void **interpreter_inst**(instvalueType instattribute);

- // interprète une liste instruction
- void **interpreter_list_inst**(listinstvalueType * listinstattribute);

Le type abstrait de stockage et CFG – Solution (cfg.c)

```
#include  "cfg.h"
// insère une instruction en queue de la liste des instructions
void inserer_inst_en_queue(listinstvalueType * plistinstattribute, instvalueType instattribute){
  listinstvalueType * liste = (listinstvalueType *) malloc(sizeof(listinstvalueType));
  liste->first = instattribute;
  liste->next = NULL;

  if (plistinstattribute->next == NULL)
          plistinstattribute->next = liste;
  else    listinstvalueType * pliste = plistinstattribute;

  while(pliste->next != NULL)
          pliste = pliste->next;

  pliste->next = liste;
}
```

```
// interprète une liste instruction
void interpreter_list_inst(listinstvalueType * listinstattribute){
if  (listinstattribute != NULL)
        interpreter_inst(listinstattribute->first);
        interpreter_list_inst(listinstattribute->next);
}
```

```
// interprète une instruction
void interpreter_inst(instvalueType instattribute){
 switch(instattribute.typeinst){
  case Print :
   if (typevar(instattribute.node.printnode.rangvar) == Bool){
     printf("%s\n", ((valinit(instattribute.node.printnode.rangvar)==false)?"false":"true"));
    }else{
     printf("%d\n", valinit(instattribute.node.printnode.rangvar));
    } break;
  case Assign :
        set_valinit(instattribute.node.assignnode.rangvar, instattribute.node.assignnode.right);
   break;
  case If :
     if ( valinit(instattribute.node.ifnode.rangvar) == instattribute.node.ifnode.right )
          interpreter_list_inst(instattribute.node.ifnode.thenlinst);
     break;
  }
}
    // valinit et set_valinit : accesseur et setteur de la valeur initial d'une variable dans la TS
    // typevar : accesseur du type d'une variable dans la TS
```

RI remarquables : Gestionnaires des erreurs

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Le type abstrait de stockage et gestionnaire des erreurs - Exercice

- Soit la grammaire suivante
  - **PROG** → LISTE_DECL LISTE_INST
  - **LISTE_DECL** → DECL | LISTE_DECL DECL ;
  - **DECL** → idf TYPE CONST ;
  - **LISTE_INST** → INST | LISTE_INST INST
  - **INST** →
    - idf ‘=‘ CONST |
    - PRINT idf |
    - if ‘(‘ idf “==“ CONST ‘)’ then LISTE_INST endif
  - **TYPE** → int | bool ;
  - **CONST** → nombre | true | false ;

- Construire les cfg nécessaires pour stocker les erreurs dérivées par G

Le type abstrait de stockage et gestionnaire des erreurs – Solution (error.h)

- typedef enum { NonDeclaredVar,
- BadlyInitialised,
- AlreadyDeclared,
- IncompatibleAssignType,
- IncompatibleCompType} **errorType**;

- typedef struct {
- char ***name**; // nom de l'identificateur qui pose problème
- int **linenumdecl**;
- errorType **errort**;
- } **error**;

- error * **creer_erreur**(errorType et, int line, char * name);

- void **creer_erreur_instruction**(errorType et, int line, char* name);

- void **creer_erreur_declaration**(errorType et, int line, char* name);

- void **afficher_erreur**(errorType et, int line, char* name);

- void **afficher_erreurs**();

- int **nombre_erreurs**();

(error.c)

Le type abstrait de stockage et gestionnaire des erreurs – Solution

```
#include "error.h"

#define NBERRMAX 100
static int NBERRDECL = 0;
static int NBERRINST = 0;

static error * ERDECL[NBERRMAX];
static error * ERINST[NBERRMAX];

void afficher_erreur(errorType et, int line, char* name){
  printf("ligne %d : %s ", line,  name);
  switch (et){
  case NonDeclaredVar:
        printf("variable non declaree\n");
        break;
  case IncompatibleAssignType:
        printf("incompatible avec la valeur d'affectation\n");
        break;
case BadlyInitialised:
        printf("variable mal initialisee\n");
        break;
  case AlreadyDeclared:
        printf("variable deja declaree\n");
        break;
case IncompatibleCompType:
        printf("incompatible avec la valeur de comparaison\n");
        break;
  }
}

int nombre_erreurs(){  return  NBRERRDECL + NBERRINST; }

void creer_erreur_instruction(errorType et, int line, char* name){
    ERINST[NBERRINST++]= creer_erreur(et, line, name);
}

void creer_erreur_declaration(errorType et, int line, char* name){
   ERDECL[NBERRDECL++]= creer_erreur(et, line, name);
}
error * creer_erreur(errorType et, int line, char* name){
  error * e = (error*) malloc (sizeof (error) );
  e->name = (char *) malloc (strlen(name));
  strcpy(e->name, name);
  e->linenumdecl = line;
  e->errort = et;


  return e;
}
void afficher_erreurs(){
  int idecl = 0;
  int iinst = 0;
  while (idecl < NBERRDECL) {
    afficher_erreur(ERDECL[idecl]->errort,
                    ERDECL[idecl]->linenumdecl,
                    ERDECL[idecl]->name);
    idecl++;
  }
 while (iinst < NBERRINST) {
    afficher_erreur(ERINST[iinst]->errort,
                    ERINST[iinst]->linenumdecl,
                    ERINST[iinst]->name);
    iinst++;
  }}
```

Utilisations des représentations remarquable pour développer un interpréteur du langage ZZ sous BISON

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

```
%{
#include <ctype.h>
#include <stdio.h>
#include <math.h>

#include "cfg.h"
#include "tableSymb.h"
#include "error.h"

int yylex(void);
void yyerror (char const *s);

int rangvar;
listinstvalueType liste_instructions_programme;
%}
```

- /* 1. START AXIOM */
- %start PROG

- /* 2. TOKENS */
- %token IDF TRUE FALSE ASSIGN OR AND NOT NOMBRE INT BOOL IF THEN EGAL ENDIF PARO PARF PRINT

- %%

- **PROG : LISTE_DECL LISTE_INST** {
- liste_instructions_programme = $<listinstattribute>2 ;
- };

- **LISTE_DECL : DECL | LISTE_DECL DECL;**

- **LISTE_INST : INST {**
- $<listinstattribute>$.first = $<instattribute>1;
- $<listinstattribute>$.next = NULL;
- }

- | **LISTE_INST INST** {
- **inserer_inst_en_queue**(&$<listinstattribute>1, $<instattribute>2);
- $<listinstattribute>$.first = $<listinstattribute>1.first;
- $<listinstattribute>$.next = $<listinstattribute>1.next;
- }
- ;

```
DECL : IDF TYPE CONST {
  // test de la duplication de la déclaration de la variable
  varvalueType newvar;
  if (inTS($<varattribute>1.name, &rangvar) == false){
    newvar.nbdecl = 1;
    newvar.name=(char *)malloc(strlen($<varattribute>1.name));
    strcpy(newvar.name,$<varattribute>1.name);
    newvar.linenumdecl=$<varattribute>1.linenumdecl;

    // test de la compatibilité des cfg
    if ($<typename>2 == $<constattribute>3.typename) {
      newvar.correct = true;
      newvar.typevar = $<typename>2;
      newvar.valinit = $<constattribute>3.valinit;
    }else{
      newvar.correct = false;
      creer_erreur_declaration(BadlyInitialised,
                               $<varattribute>3.linenumdecl,  newvar.name);
    }
    ajouter_nouvelle_variable_a_TS(newvar);

  }else{
    incrementer_nombre_declarations(rangvar);  // equiv à TS[rangvar].nbdecl++
    creer_erreur_declaration(AlreadyDeclared,
                             $<varattribute>1.linenumdecl,  $<varattribute>1.name);
  } };
```

```
INST : IDF ASSIGN CONST
{
  if (inTS($<varattribute>1.name, &rangvar) == false){
    creer_erreur_instruction(NonDeclaredVar,
                             $<varattribute>1.linenumdecl, $<varattribute>1.name);

  }else{
    // variable déjà déclarée
    if ( typevar(rangvar) != $<constattribute>3.typename) {
      creer_erreur_instruction(IncompatibleAssignType,
                               $<varattribute>1.linenumdecl, $<varattribute>1.name);
    }else{
      // variable bien initialisée
      $<instattribute>$.typeinst              = Assign;
      $<instattribute>$.node.assignnode.rangvar = rangvar;
      $<instattribute>$.node.assignnode.right   = $<constattribute>3.valinit;
    }
  }
} |
```

```
IF PARO IDF EGAL CONST PARF THEN LISTE_INST ENDIF
{
  if (inTS($<varattribute>3.name, &rangvar) == false){
     creer_erreur_instruction(NonDeclaredVar,
                                  $<varattribute>3.linenumdecl, $<varattribute>3.name);

  }else{
    // variable déjà déclarée

    if ( typevar(rangvar) != $<constattribute>5.typename) {
     creer_erreur_instruction(IncompatibleCompType,
                                   $<varattribute>3.linenumdecl, $<varattribute>3.name);

    }else{
      // variable bien comparée
      $<instattribute>$.typeinst = If;
      $<instattribute>$.node.ifnode.rangvar = rangvar;
      $<instattribute>$.node.ifnode.right = $<constattribute>5.valinit;
      $<instattribute>$.node.ifnode.thenlinst=  & $<listinstattribute>8;
    }
  }
} |
```

```
PRINT IDF
{
  if (inTS($<varattribute>2.name, &rangvar) == false){
    creer_erreur_instruction(NonDeclaredVar,
                                $<varattribute>2.linenumdecl, $<varattribute>2.name);
  }else{
    // variable déjà déclarée
    $<instattribute>$.typeinst = Print;
    $<instattribute>$.node.printnode.rangvar  = rangvar;
  }
};

TYPE : INT   {$<typename>$ = Int;} |
       BOOL    {$<typename>$ = Bool;};

CONST :
        NOMBRE {$<constattribute>$.typename = Int;
                  $<constattribute>$.valinit = $<constattribute>1.valinit;} |
        TRUE   {$<constattribute>$.typename = Bool;
                  $<constattribute>$.valinit = true;}                    |
        FALSE  {$<constattribute>$.typename = Bool;
                  $<constattribute>$.valinit = false;};
%%
```

```
#define debug 0

int main(int argc, char** argv){
  int retour = yyparse();

 if (nombre_erreurs() > 0){
    afficher_erreurs();
  }else{
    interpreter_list_inst(&liste_instructions_programme);
  }

if (debug){
    printf("\nSynthèse de l'Etat de la table des symboles :\n");
    afficherTS();
  }
  return retour ;
}
```

Makefile

- **bis** : ZZ.y
- 	bison -d ZZ.y

- **lex** : ZZ.l bis
- 	flex -i --yylineno ZZ.l
- 	mv lex.yy.c ZZ.yy.c
- 	gcc -c ZZ.yy.c

- **cfg** : cfg.h tableSymb.h cfg.c
- 	gcc -c cfg.c

- **error** : error.h error.c
- 	gcc -c error.c

- **tableSymb** : tableSymb.h cfg.h tableSymb.c
- 	gcc -c tableSymb.c

- **tab** : ZZ.tab.h ZZ.tab.c
- 	gcc -c ZZ.tab.c

- **out** : cfg error tableSymb lex tab
- 	gcc -o ZZ cfg.o error.o tableSymb.o ZZ.yy.o ZZ.tab.o

- **clean** :
- 	rm *.o
- 	 rm *.tab.h
- 	rm *.tab.c
- 	rm *.yy.c
- 	rm *~
- 	rm *.exe
- 	rm *dump

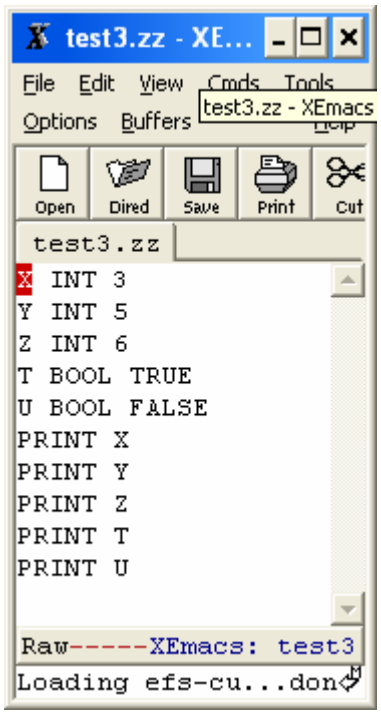
Exemples d'exécution de l'interpréteur ZZ

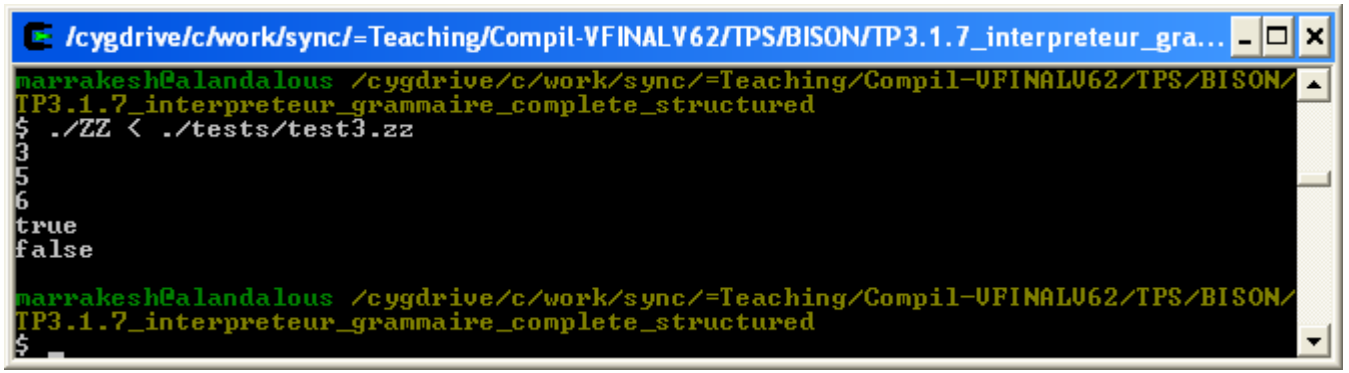
Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

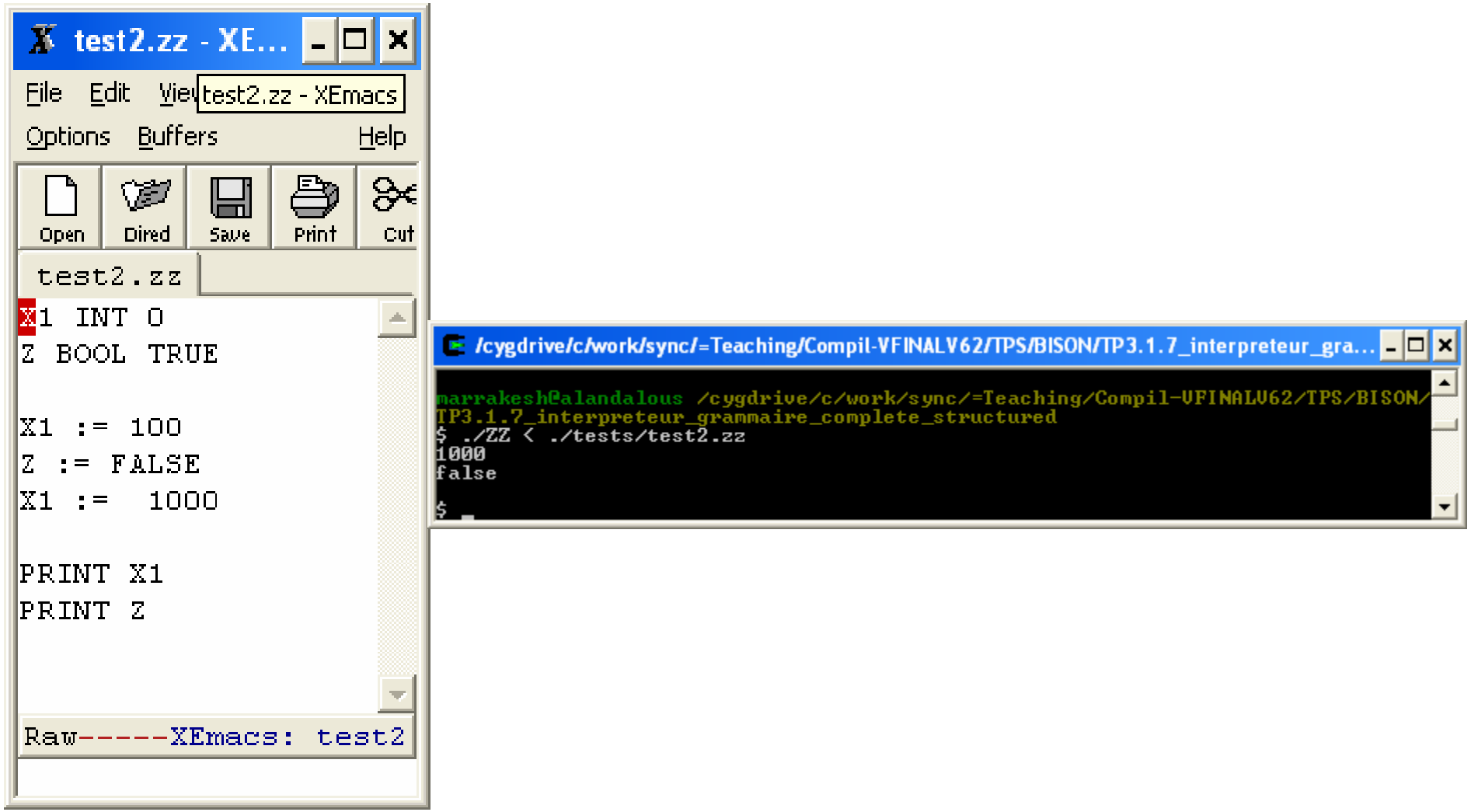
Prof. K. Baïna 2010 ©, 7ème édition

Tester dés déclarations, et des affichages

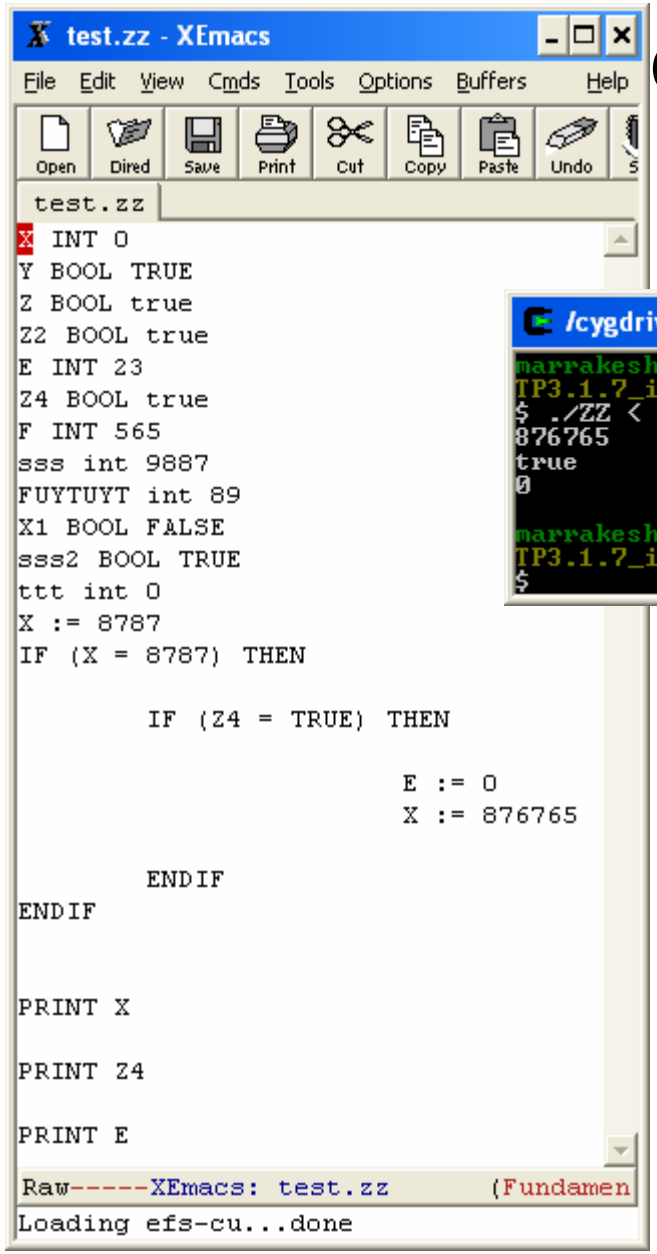


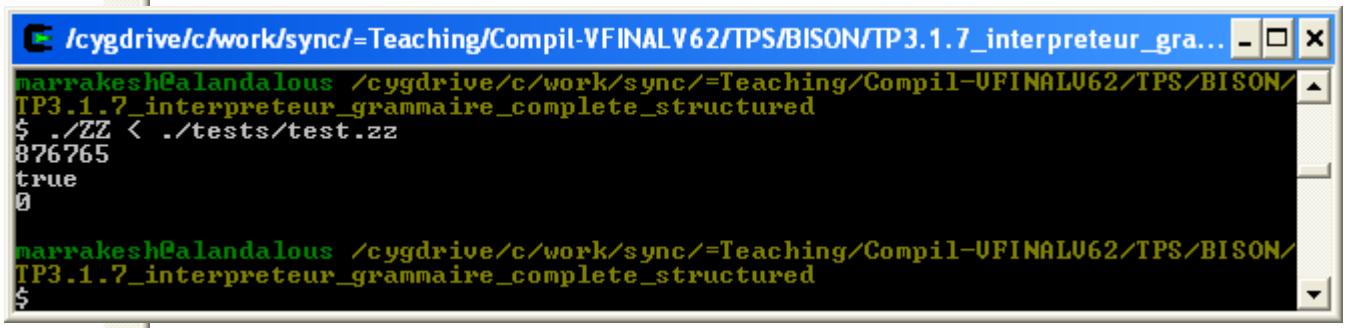


Tester dés déclarations, des affectations et des affichages

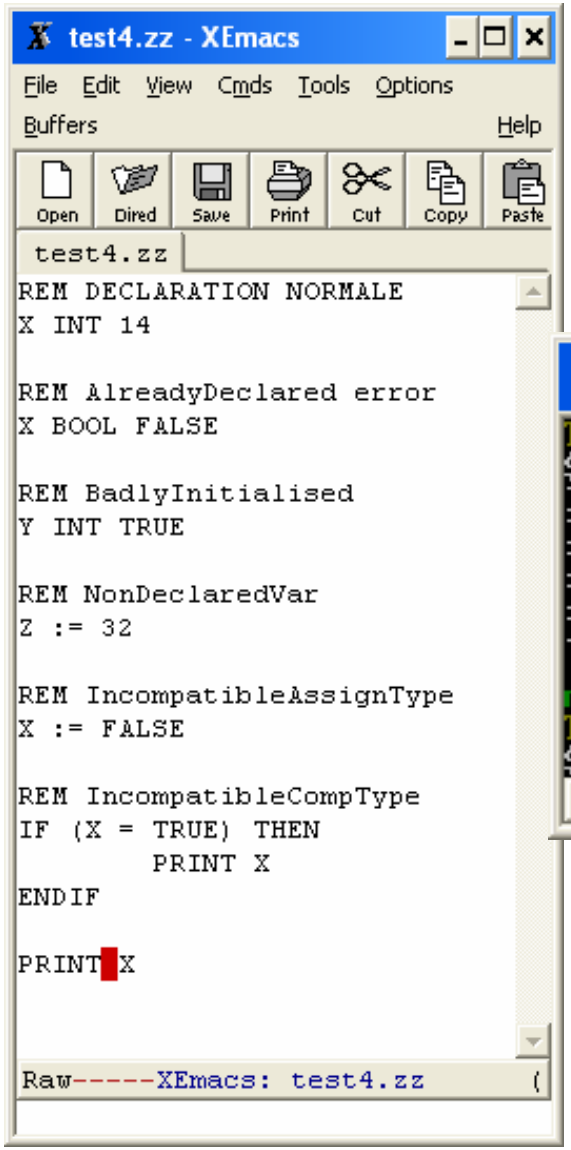


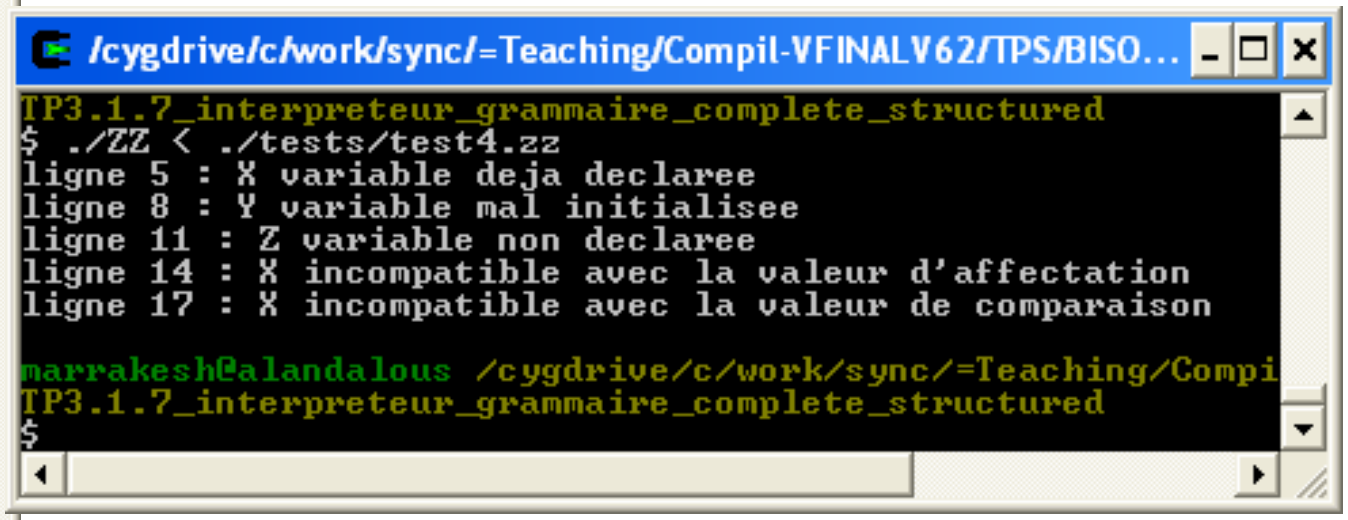
Tester dés déclarations, des affectations, des conditionnelles et des affichages



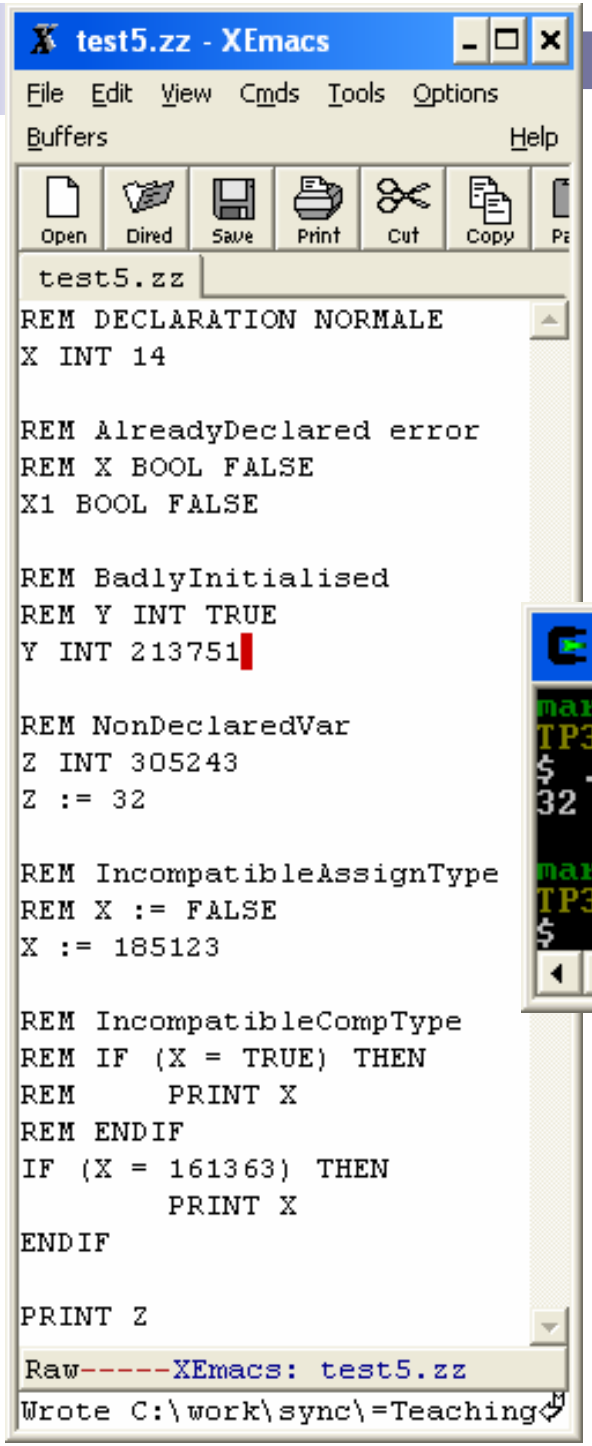


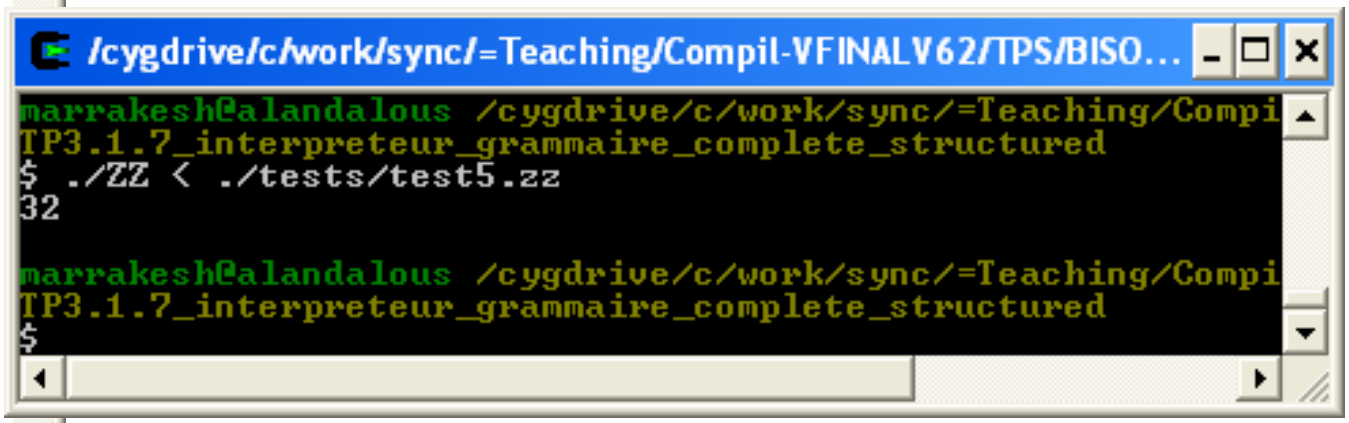
Et le tester des erreurs





Et la correction des erreurs





Compilation

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Lectures recommandées - livres

1. « Engineering a Compiler » ©, K. D. Cooper & L. Torczon, Morgan Kaufmann 2004
2. « Introduction to Automata Theory Languages and Computation » ©, J. E. Hopcroft, R. Motwani, J. D.Ullman, 2001, Addison Wesley
3. « Compilers and Compiler Generators » ©, P.D. Terry, 2000

- Cours « Machines virtuelles », Samuel Tardieu, ENST
- Cours « Compilation et interprétation », Danny Dubé 2006

Plan du cours

1. Module initial Motivation & Définitions
2. Module 1 Analyseur lexical *(langages réguliers et théorie des automates)*
3. Module 2 Analyseur syntaxique *(langages hors contexte)*
4. Module 3 Analyseur sémantique *(grammaires attribuées)*
5. Module 4 Représentations Intermédiaires
6. **Module 5** Génération de code intermédiaire
7. Module 6 Optimisation et Génération de code
8. Conclusion et perspectives

Module 5 Générateur de pseudo-code (code intermédiaire)

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

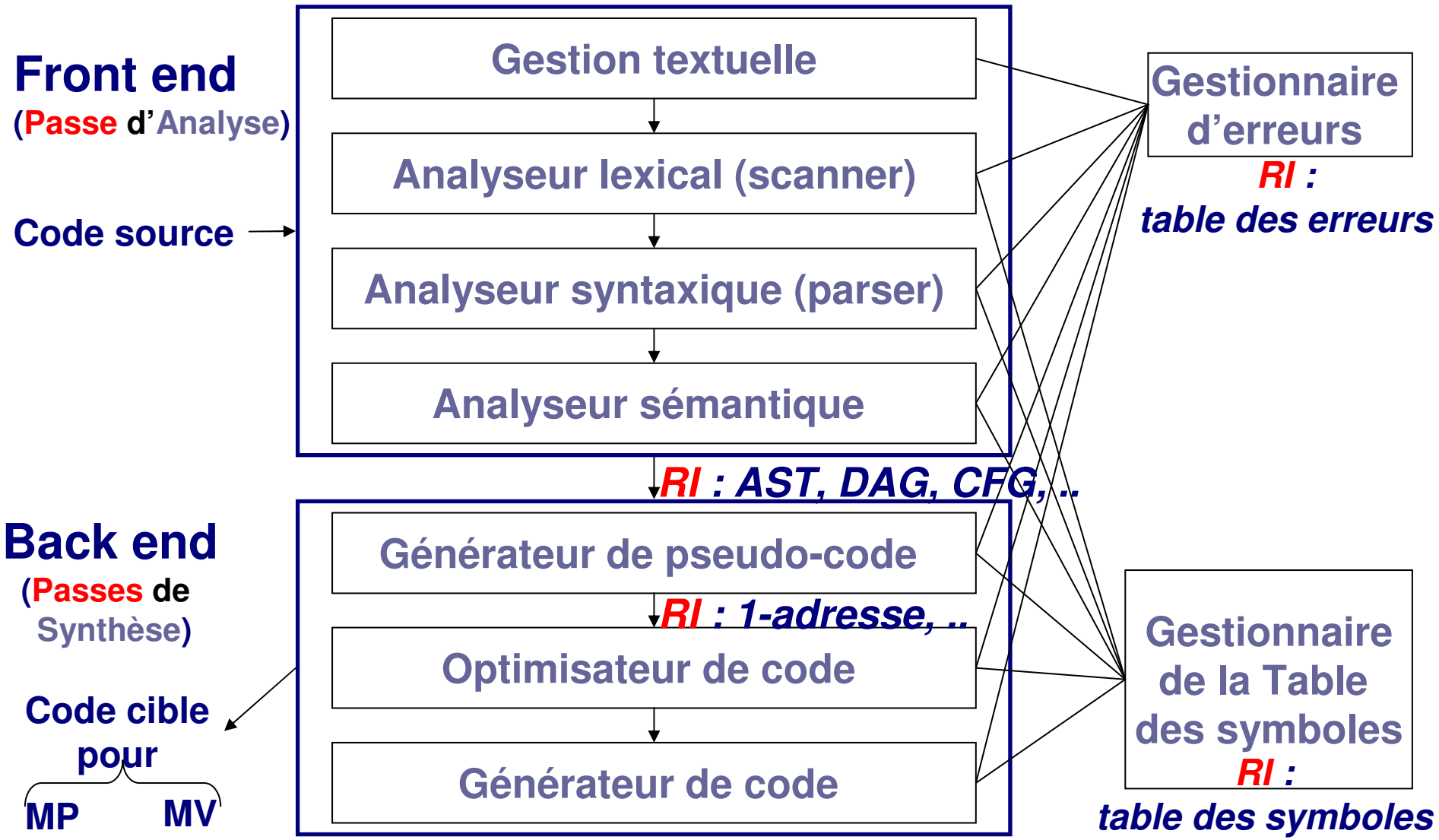
***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Représentations intermédiaires linéaire : pseudo-code – Plan

- Introduction et Motivation
- Concepts
  - RI Liénaires : code à 1-adresse, 2-adresses, ..
  - Machine virtuelle
- Transformation
  - Calcul de nombre minimum de registres
  - Génération de pseudo-code
- Générateurs de pseudo-code
  - générateurs de pseudo-code : 1-adresse, 2-adresses, 3-adresses (quadruples), etc.

Passes d'un compilateur



Concepts

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Pseudo-code et Machine virtuelle

- Définitions
- Rôle :
  - Pourquoi le besoin de pseudo-code ?
  - Différence entre code et pseudo-code ?
- Conteneur : MV
  - Définition
  - Rôle
  - Comportement

Motivation : Pourquoi la génération de code intermédiaire ?

- La compilation en code natif de la RI graphique peut avoir des désavantages :
  - Le code compilé n'est en général pas portable d'une architecture à une autre
  - Le code compilé peut générer du code redondant (non optimal) du fait de la compilation instruction par instruction sans vision globale
  - Le code compilé ne suit pas les évolutions des architectures d'un même contructeur :
    - Un programme compilé pour un Pentium 4 ne tournera pas sur un 386 (instructions manquantes)
    - Un programme compilé pour un 386 ne sera pas optimisé pour Pentium 4 (pas d'instructions MMX, ordonnancement)

Motivation : Pourquoi la génération de code intermédiaire ?

- Solution :
  - Concevoir une machine virtuelle qui possède un code intermédiaire
  - Compiler le code source vers ce code intermédiaire (*as* pseudo-code ou bytecode)
  - Avec les possibilités de :
    - L'interpréter : à l'aide de l'interpréteur de la machine virtuelle
    - Le Compiler (transcrire=synthétiser) vers le code natif spécifique au besoin
    - Le Compiler (transcrire la première fois et l'exécuter les fois suivantes) à la volée vers le code natif (JIT : Just in Time) : transformer le bytecode en code natif au chargement ou à la première fois

Pseudo-code - Définition

- **Pseudo-code**
  - RI assemblant la **connaissance** (d'un niveau bas : low level) du compilateur
  - Code qui tourne sur une machine virtuelle (VM)
  - Code simple, compacte et facile à réarranger
  - Facile à débuguer
  - Permet une optimisation indépendante de la machine cible
  - Indépendant de la machine cible (Portable)
  - Les algorithmes pour ce type de RI opèrent sur de simples séquences linéaires d'opérations
  - Alias : Byte code, code intermédiaire

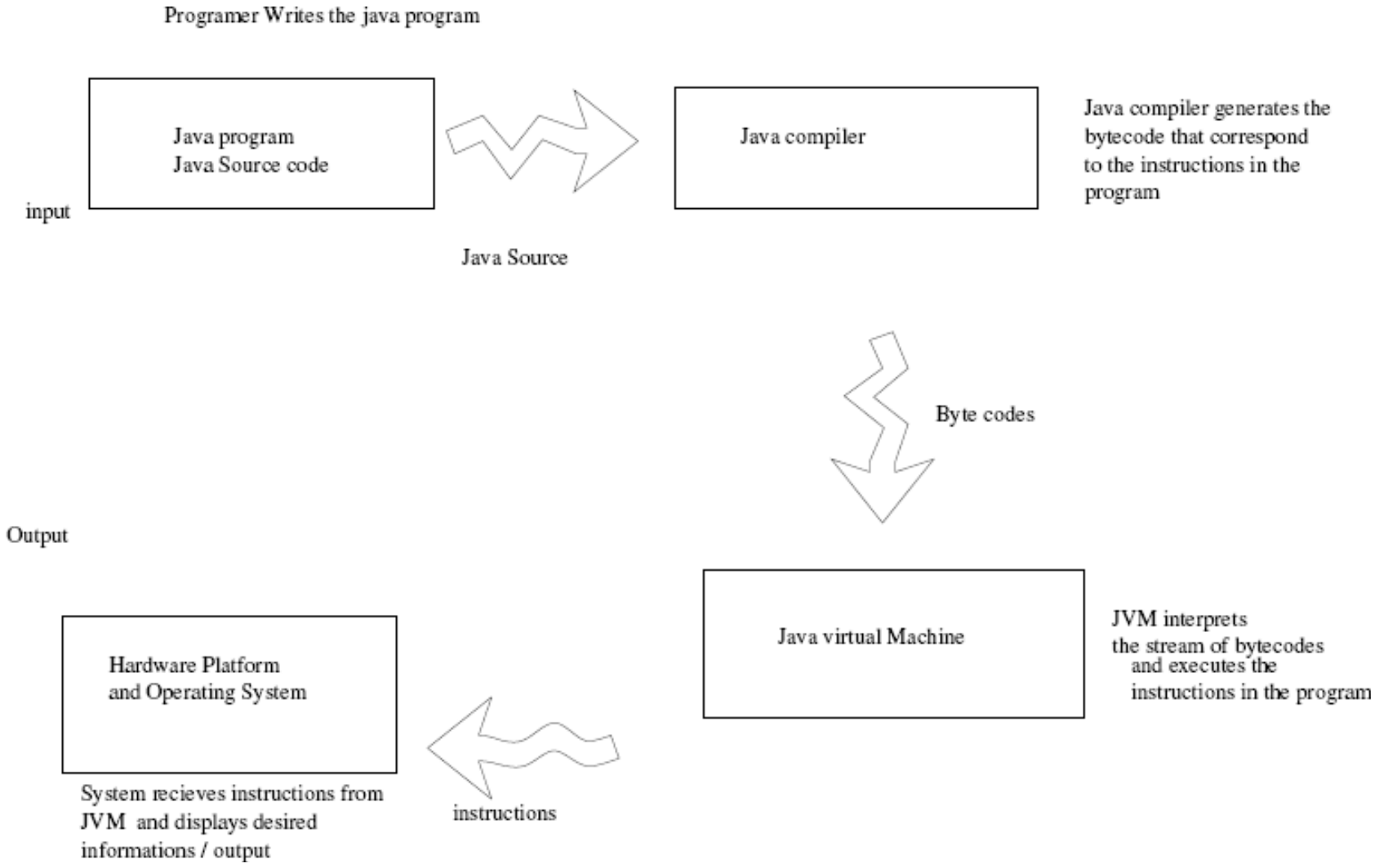
- **Code**
  - Alias : Code natif
  - Tourne sur la machine physique (MP)
  - Plus rapide

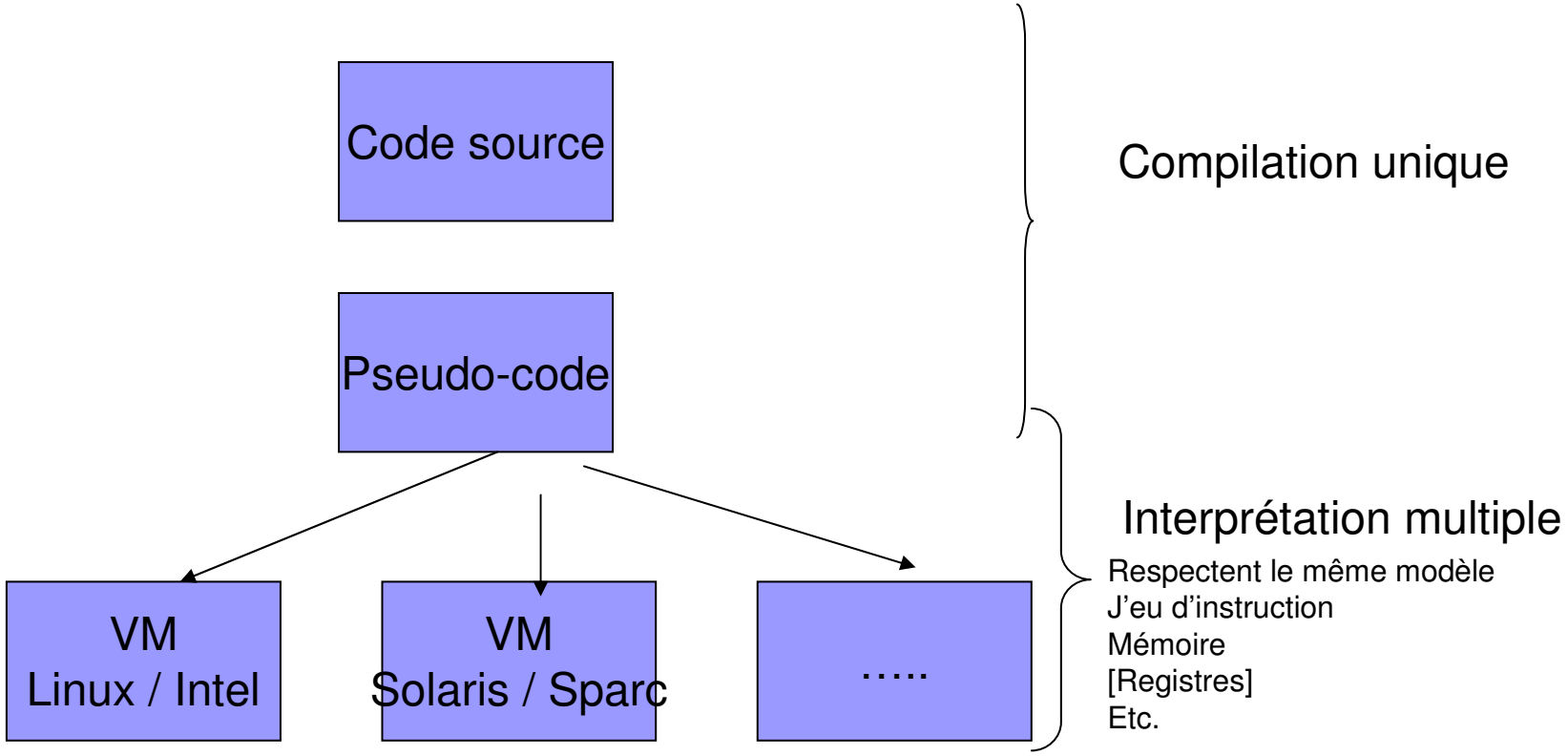
Machine virtuelle (as abstraite) - Définition

- Une Machine virtuelle définit une architecture d'ordinateur abstrait
  - Spécifie le **jeu d'instructions virtuelles** que cette architecture peut exécuter. Ce jeu d'instructions peut être prédéterminé ou dynamique, simples ou complexes
  - Contient une **mémoire virtuelle**
  - Contient *éventuellement* un ensemble de **registres virtuels**

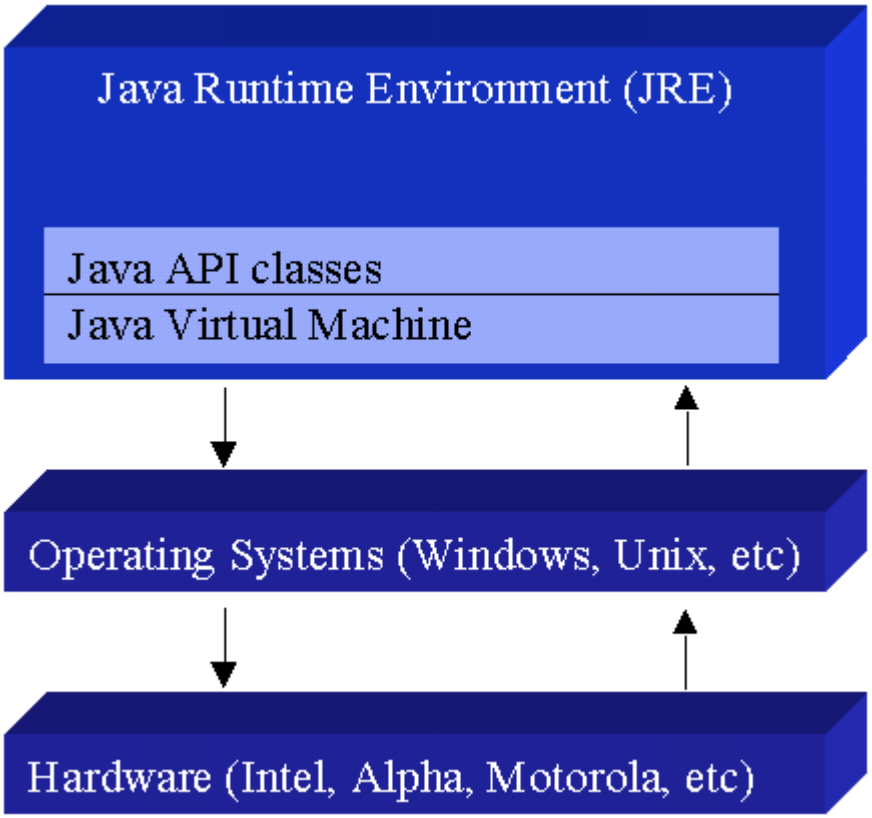
- Certaines machines virtuelles émulent de vraies architectures :
  - émulateur 68LC040 d'Apple pour ses systèmes à base de PowerPC, émulateur DragonBall de Palm Inc pour ses assistants personnels à base d'ARM, émulateur PlayStation pour Unix

Machine virtuelle - Exemple (Java Virtual Machine)



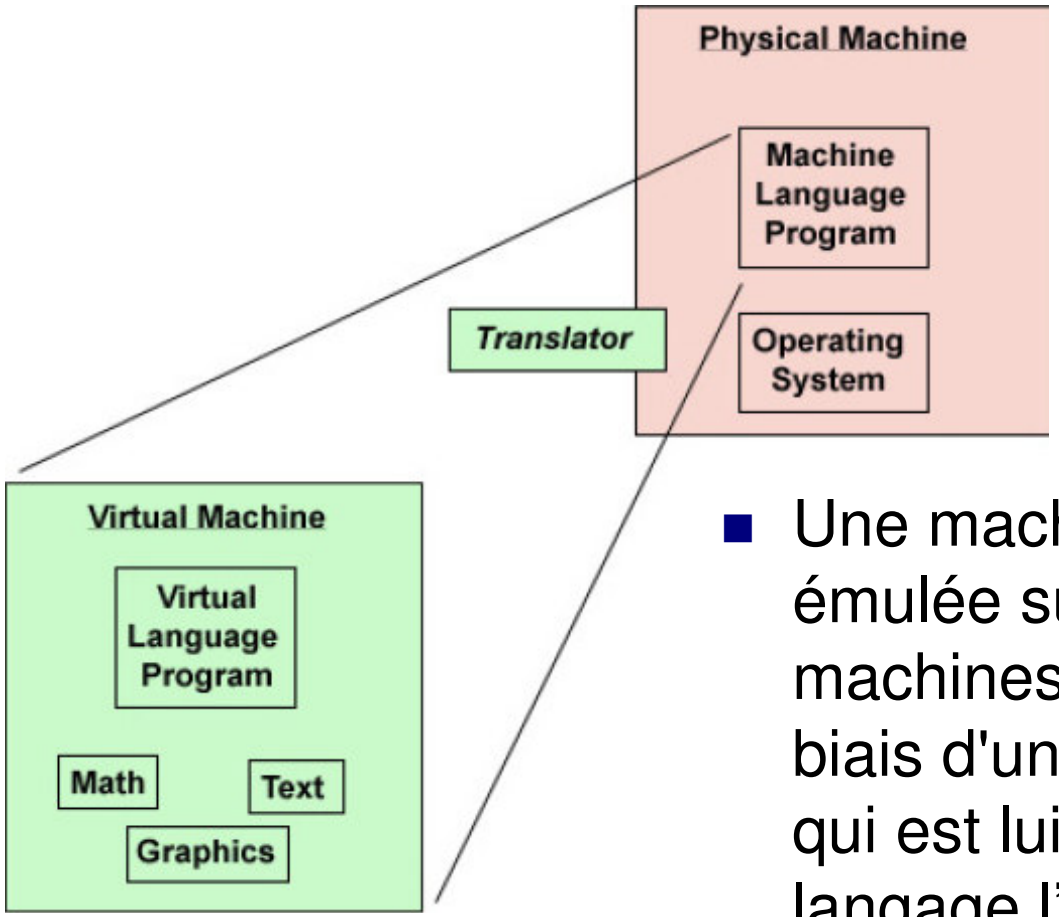

Code source
Pseudo-code
VM
Linux / Intel
VM
Solaris / Sparc
.....
Compilation unique
Interprétation multiple
Respectent le même modèle
J'eu d'instruction
Mémoire
[Registres]
Etc.

Machine virtuelle – Exemple (JVM)



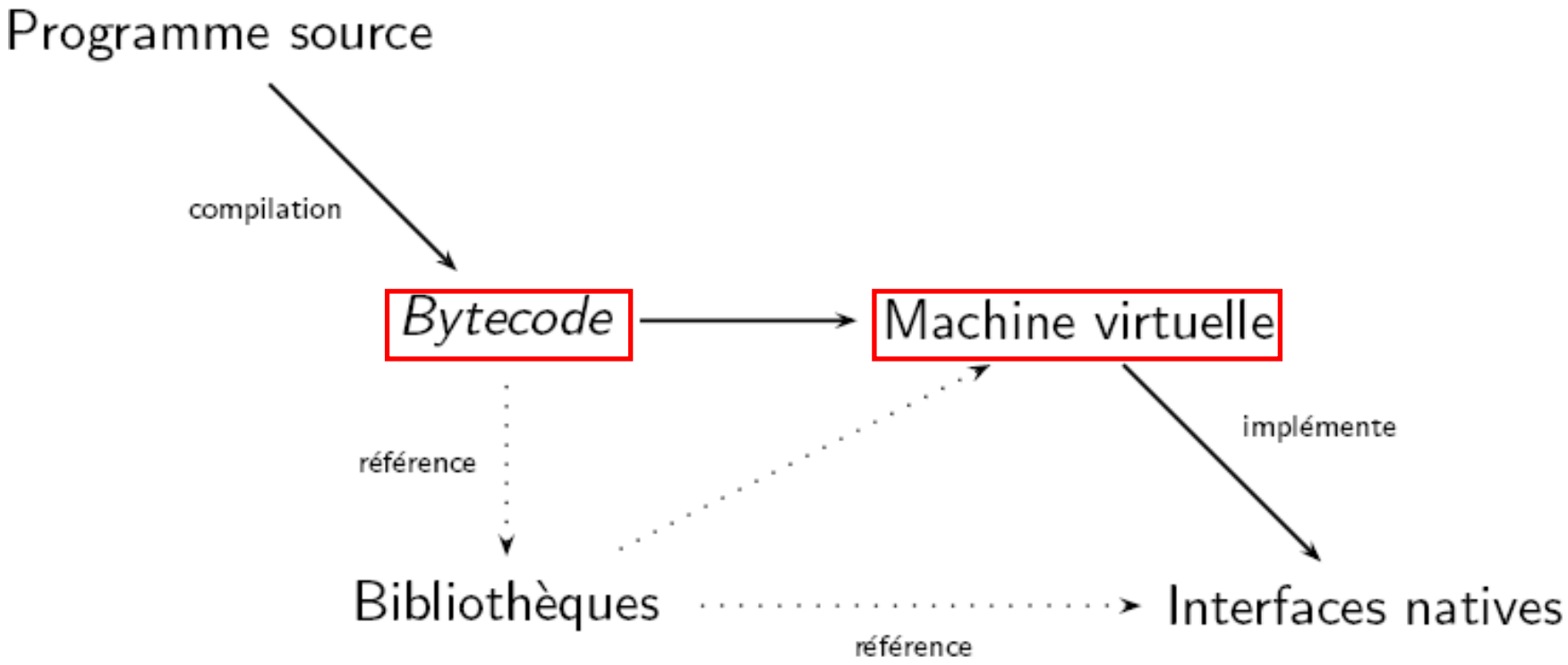
- La Machine virtuelle est responsable d'interpréter le Bytecode et le traduire en actions ou en appels OS

Machine virtuelle *versus* Machine Physique



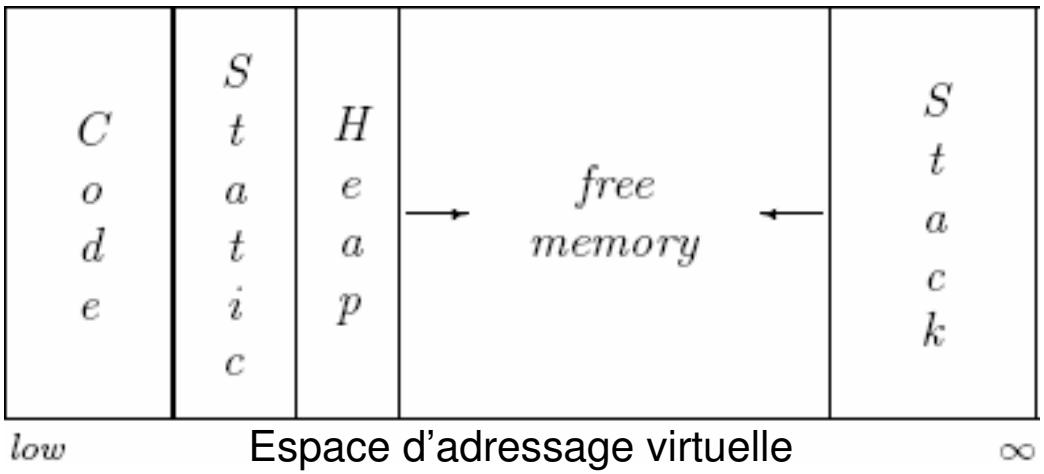
- Une machine virtuelle est émulée sur une ou plusieurs machines physiques (par le biais d'un moteur d'exécution) qui est lui compilé dans le langage l'assembleur de cette machine physique

Machine virtuelle - Processus de compilation

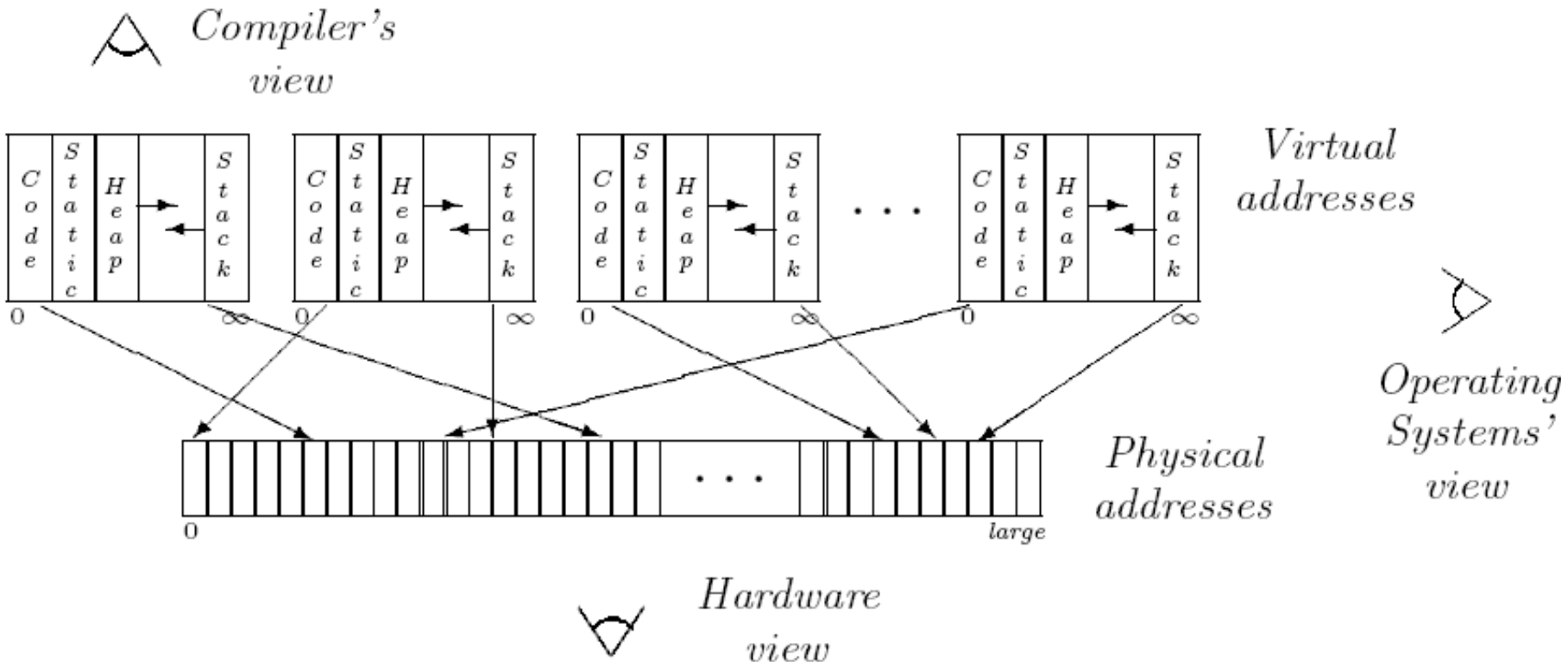


Machine virtuelle - mémoire

- Mémoire Code
  - Opérations : chargement du code
- Mémoire statique
  - Allocations statiques
    - Opérations : déclarations de variables globales
- Tas (heap)
  - Allocations dynamiques
    - Opérations : réservation de zone pour pointeurs
- Pile (stack)
  - Adresses de retour
  - Valeurs des paramètres
  - Variables locales
  - Valeur de retour
    - Opérations : PUSH/POP…



La mémoire : selon plusieurs points de vues



Machine virtuelle - Jeu d'instructions

- Le choix du jeu d'instructions de la machine virtuelle doit être fait judicieusement.
- Il doit être assez riche pour pouvoir exprimer tous les calculs possibles dans le langage source
- S'il est trop étendu :
  - Côté programmeur du compilateur : le générateur de code devient alors lourd. Tout changement de langage cible est coûteux.
  - Côté taille de code produit : le code devient énorme vue la tailles des instructions (Code opération énorme)
- S'il est trop restreint, les séquences de code intermédiaire à générer deviennent plus longues
  - la réalisation des optimisations subséquentes est plus difficile.

Machine virtuelle - Types de jeu d'instructions

- Les représentations communes du pseudo-code dépend de la machine cible:
  - **code à 1-adresse** (*stack code*) : pour les machines à *Pile.* Les opérations supposent un opérande implicite « *implicit name space* » (la pile). Simple à générer et à exécuter.
  - **code à 2-adresse** : pour les machines à opération mémoire-registre architectures CISC (Complex Instruction Set Computers)
    - exemples : Intel 80x86, Motorola 680x0
  - **code à 3-addresses** : pour les architectures RISC (Reduced Instruction Set Computers)
    - exemples : **A**corn **R**ISC **M**achine : ARM, Alpha, Sprac, MIPS

RI Linéaires – *Exemple* x – 2 × y

Invariants :
Pré-condition : Les paramètres de tte op sont dans la pile
Post-condition : Le résultat de tte op est dans la pile

push	2		loadi	2	⇒	t_1		loadi	2	⇒	t_1
load	y		load	y	⇒	t_2		load	y	⇒	t_2
multiply			mult	t_2	⇒	t_1		mult	t_1, t_2	⇒	t_3
load	x		load	x	⇒	t_3		load	x	⇒	t_4
subtract			sub	t_1	⇒	t_3		sub	t_4, t_3	⇒	t_5
one-address code			*two-address code*					*three-address code*			

Écriture humaine
Non automatisé

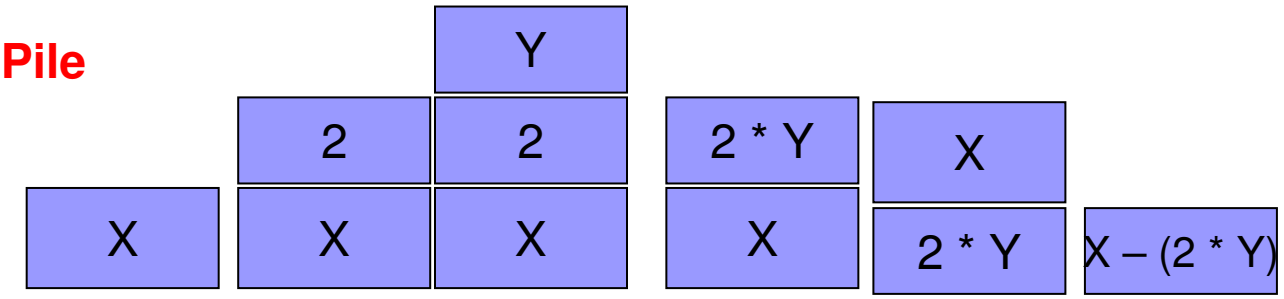
* Un pile
* Des registres
- Un registre est accédé en R/W
+ des instructions de tailles moyennes
+ taille du programme (nb instruction) plus compacte que le 1-@ code

* Un pile
* Des registres
(des registres accédés en R, un registre est accédé en W)
- Insctructions longues (taille binaire)
+ parallélisme (pipeline) efficace
+ taille du programme (nb instruction) plus compacte que le 2-@ code

* Une pile
- Pas de registres
+ code compact
+ instructions courtes (taille binaire)
+ moins d'hypothèse (conséquence portabilité bytecode java est un one-address code)

Exemple x – 2 × y (G, D, Op)

- Load x / tête de pile = x
- Push 2 // tête de pile = 2
- Load y // tête de pile = y
- Mult // tête de pile = 2 * y
- Swap //
- Sub



RI Linéaires – *code à 1-adresse*

(One-address code)

- Plusieurs architectures à base de Pile ont été construites, les codes à 1-adresse ont apparu pour répondre à la demande de compilation pour ce type d'architecture

- Un **code à 1-adresse** (*a.k.a* code pour machine à *Pile*-stack machine code), est un code où la plupart des opérations manipule la *Pile*
  - suppose l'existence d'un opérande implicite *Pile*
  - *Possède un jeu d'instruction : PUSH, POP, SUB, DUPL, SWAP, etc.*
  - a permis de construire les interpréteurs de bytecode pour des langages tels Smalltalk-80 et Java

- **Avantage**
  - est compacte en taille des programmes : La *Pile* crée un espace de nommage (namespace) implicite qui n'a pas besoin d'être représenté dans la RI

RI Linéaires – *code à 1-adresse*

sémantique des opérations

- *Jeu d'instruction : PUSH, POP, SUB, DUPL, SWAP, etc.*
  - Sémantique des opérations binaires op ∈ (**ADD**, **SUB**, **MULT**, etc.) : Elles dépilent les deux éléments de tête de *Pile* et empile leur résultat dans la *Pile* (***Push(Pop() <op> Pop()***)
  - Sémantique de l'opération **SWAP** inter-change les deux éléments de tête de *Pile.* ***SWAP*** *est utile pour les opérations non commutatives*
  - Sémantique de l'opération **DUPL** duplique l'élément en tête de Pile. ***DUPL*** *est utile pour l'exposant (**2) et pour la multiplication (*2)*
  - Sémantique de l'opération **STORE** (i.e. nécessaire à **l'affectation** vers une variable) : dépile la pile et la stocke à l'adresse mémoire de la variable en paramètre
  - **LOAD** : charger sa valeur depuis la mémoire de la variable, puis empiler sa valeur
  - **PUSH** : empiler une valeur constante

RI Linéaires – *code 2-adresses*

- Un **code à 2-adresses** est une autre RI qui permet des instructions de la forme $X \leftarrow X$ op Y, avec un unique opérateur et au plus deux noms
- C'est un 1-address par définition
- Il est adapté à des architectures avec des opérations mémoire-registre
- En utilisant deux noms (au lieu de 3), on économise de l'espace programme au détriment de modifier un des opérandes

RI Linéaires – *code 3-adresses*

- Un **code à 3-adresses** est une autre RI qui permet des instructions de la forme $Z \leftarrow X$ op Y, avec un unique opérateur et au plus trois noms
- C'est par définition un 1-adresse, cependant on écrase pas un registre utilisé pour le stockage de la valeur d'un sous-arbre
- Il est adapté à des architectures RISC avec des opérations mémoire-registre
- En utilisant trois noms, on ne modifie pas un des 2 autres opérandes
- Les quadruples des opérations indépendemment de la machine (ni convention MOTOROLA ni INTEL)
  - (code opération, opérande1, opérande 2, opérande 3)

Exemple de comparaison 2 & 3 @ code

- Si la taille de l'instruction 3-@ code est plus importante que le 2-@ code
- La taille du programme peut être bcp plus compacte

- Exemple :
  - 2 instructions
    - ADD R1, R2 // R1 := R1 +R2 (convention intel)
    - MOV R1, R3 // R3 := R1
    - Taille 1 = 2 codop + 4 emplacement registres
  - 1 instruction
    - ADD R1, R2, R3 // R3 := R1 +R2 (par convention c'est R3 qui est modifié)
    - Taille 2 = 1 codop + 3 emplacement registres

Génération de pseudo-code

- A partir de la structure arborescente de la RI graphique d'un programme, la **génération de pseudo-code** en crée une bonne **représentation intermédiaire linéaire**.

- La génération nécessite un **nouveau parcours** (**passe**) de la représentation intermédiaire produite par la passe d'analyse

Génération du Pseudo-code – **1-address**

Algorithme de Génération de pseudo-code d'expressions arithmétiques binaires – ***selon code à 1-adresse***

- void générer_code_constante (CONST)
  - Générer l'instruction **PUSH** pour empiler sa valeur littérale
    - Sémantique de **PUSH** : empiler la valeur de la constante

- void générer_code_variable(IDF)
  - Générer l'instruction **LOAD** pour charger sa valeur
    - Sémantique de **LOAD** : lire sa valeur de l'adresse mémoire de la variable, puis l'empiler

- void générer_code_affectation(leftEXP, rigthEXP)
  - générer_code(rightEXP)
  - Générer l'instruction **STORE leftEXP** pour stocker la valeur dans leftEXP

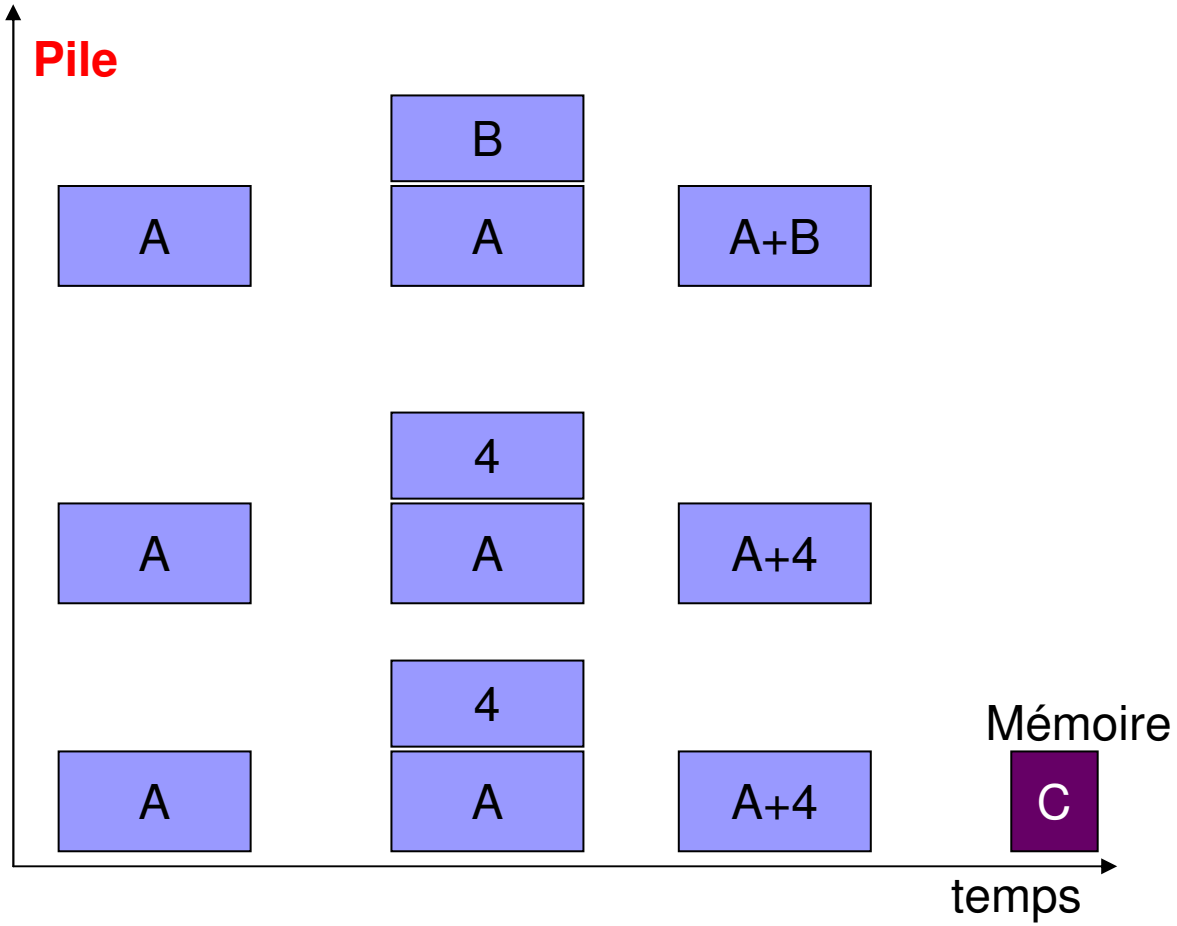
- void générer_code_expression(EXP) « algorithme de parcours post-fixé »
  - générer_code(arbre_gauche(EXP))
  - générer_code(arbre_droit(EXP))
  - Générer l'instruction liée à l'opérateur de EXP : ADD, SUB, MULT, etc.
    - Si l'opération n'es pas commutative : prévoir un SWAP

Exemples d'application de l'algorithme et interprétation 1-address code

Pseudo-code

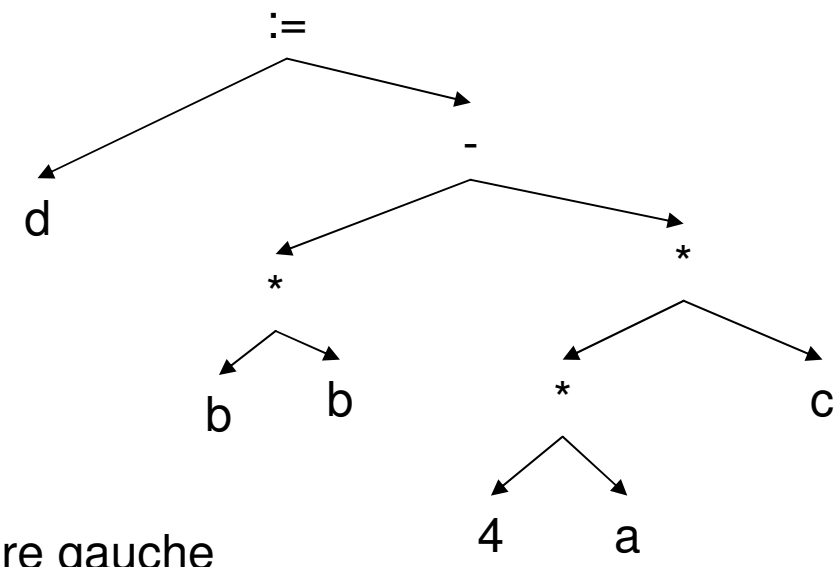
1. A + B
   - LOAD A
   - LOAD B
   - ADD
2. A + 4
   - LOAD A
   - PUSH 4
   - ADD
3. C := A + 4
   - LOAD A
   - PUSH 4
   - ADD
   - STORE C

Interprétation par la VM



Exercice 1

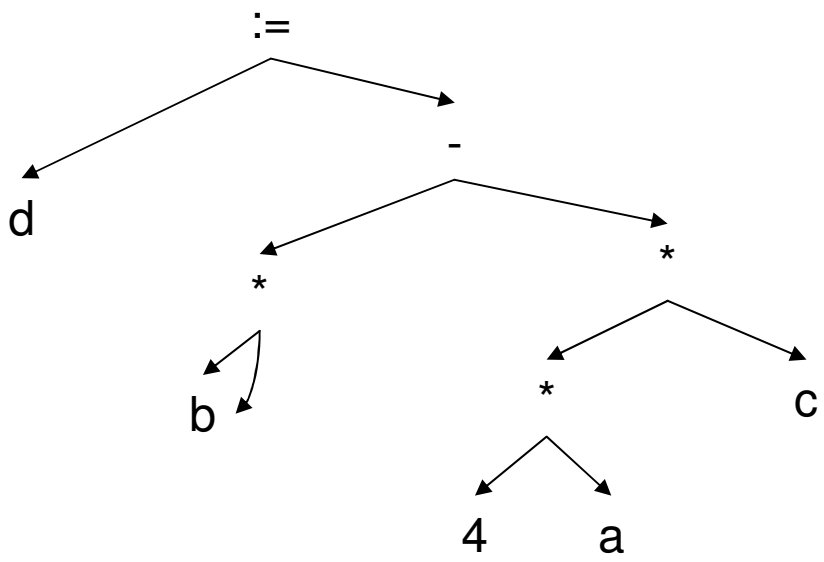
- Générer le pseudo-code 1-address code pour “d := b^2 – 4 a c”



Produite par une grammaire gauche

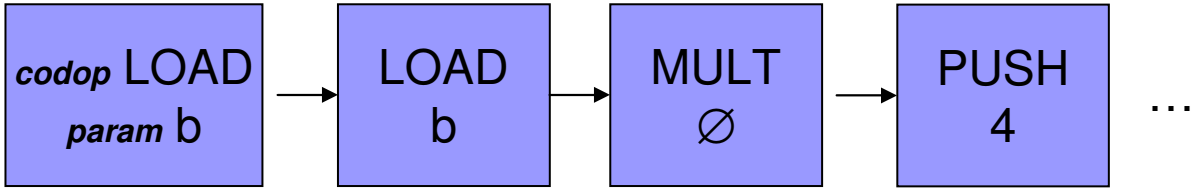
Remarques

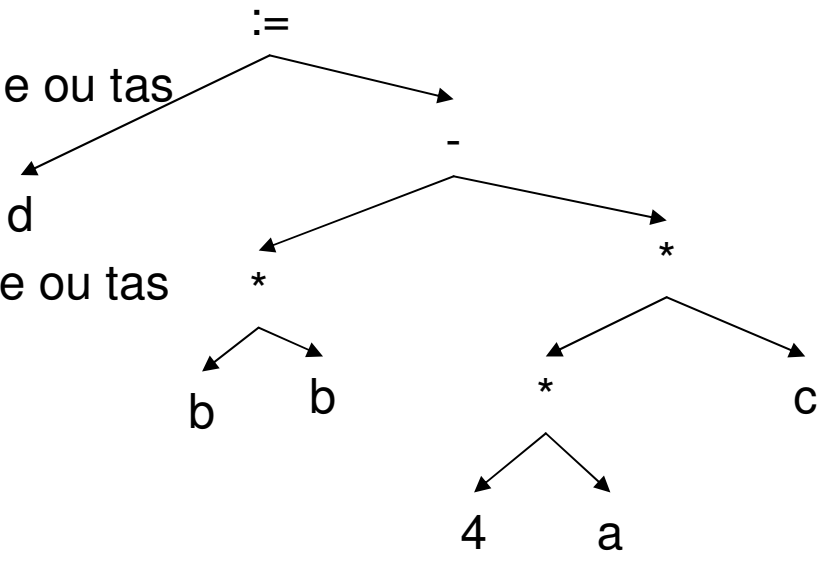
- On peut considérer le DAG de l'instruction "d := b^2 – 4 a c"



Solution 1

1. Load b // depuis la mémoire statique ou tas
2. Load b
3. Mult
4. Push 4
5. Load a // depuis la mémoire statique ou tas
6. Mult
7. Load c // depuis la mémoire statique ou tas
8. Mult
9. Swap
10. Sub
11. Store d // vers la mémoire statique ou tas

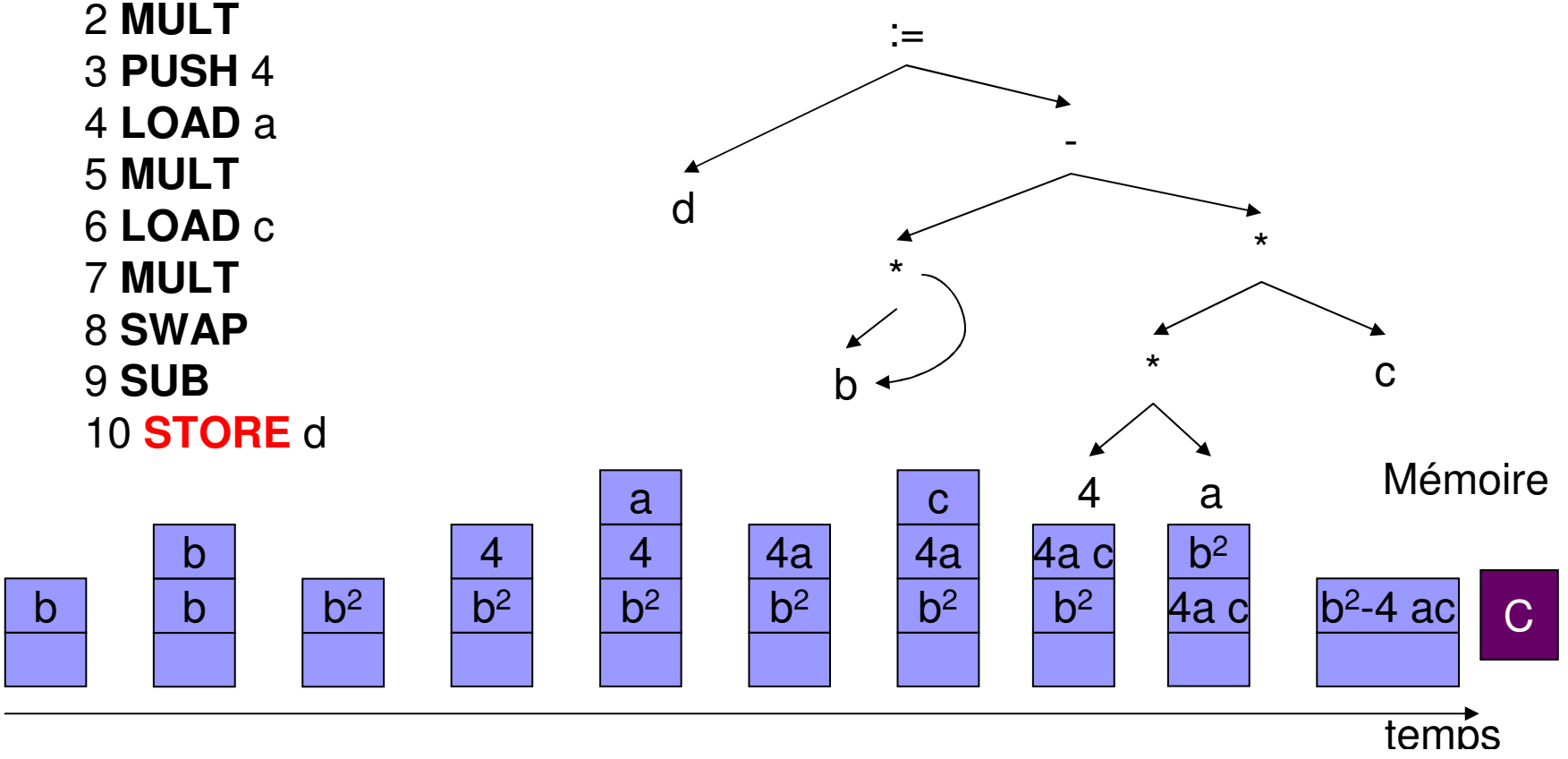




RI Linéaires – *code à 1-adresse*
(One-address code) en considérant le DAG

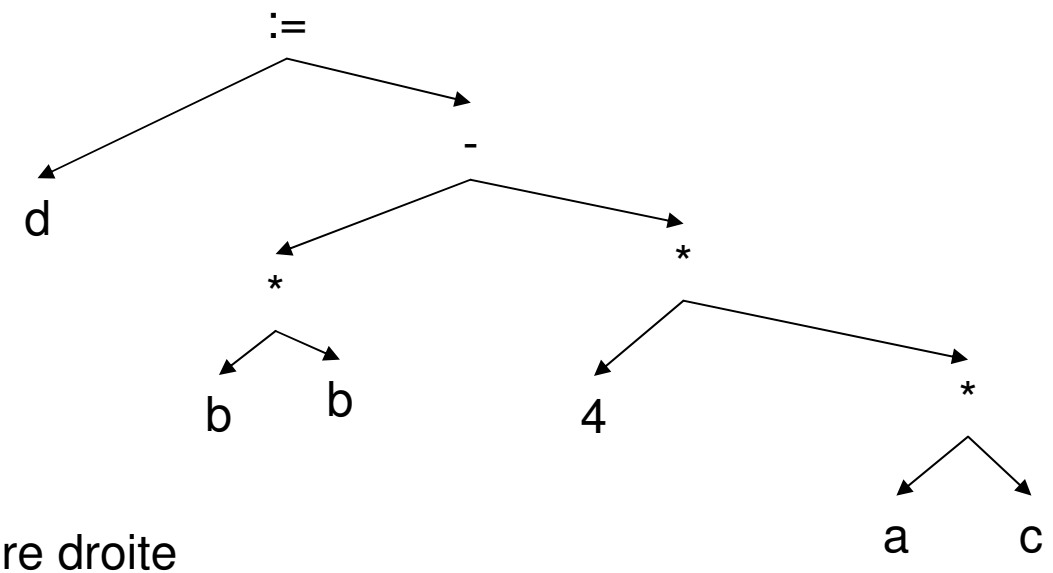
- Le pseudo-code à générer pour "d = b^2 – 4 a c"

  0 **LOAD** b - - charge b depuis son emplacement et l'empile *(b=tête_pile())*
  1 **DUPL** - - duplique la tête de pile
  2 **MULT**
  3 **PUSH** 4
  4 **LOAD** a
  5 **MULT**
  6 **LOAD** c
  7 **MULT**
  8 **SWAP**
  9 **SUB**
  10 **STORE** d



Exercice 2

- Générer le pseudo-code 1-address code pour “d := b^2 – 4 a c”



Produite par une grammaire droite

Code pour évaluation de droite à gauche de 4 * a * c

1. Load b // depuis la mémoire statique ou tas
2. Load b
3. Mult
4. Push 4
5. Load a // depuis la mémoire statique ou tas
6. Load c // depuis la mémoire statique ou tas
7. Mult
8. Mult
9. Swap
10. Sub
11. Store d // vers la mémoire statique ou tas

Le type abstrait de stockage et pseudo-code – Solution (pseudocode.h)

```
typedef enum
        {ADD, DUPL, LOAD, MULT, POP, PUSH, SUB, STORE, SWAP} CODOP;

typedef struct pseudocodenode * pseudocode;

typedef union {
  char * var;
  double _const;
} Param;

struct pseudoinstruction{
  CODOP codop;
  Param param; // une opération possède un paramètre au maximum
};

struct pseudocodenode{
  struct pseudoinstruction first;
  struct pseudocodenode * next;
};
```

Le type abstrait de stockage et pseudo-code – Solution (pseudocode.h)

- // precondition pc1 <> NULL et pc2 <> NULL
- // insere pc2 en queue de pc1
- void **inserer_code_en_queue**(pseudocode pc1, pseudocode pc2);

- // affiche une pseudo insctruction
- void **afficher_pseudo_instruction**(struct pseudoinstruction pi);

- // precondition pc <> NULL
- // afiche un pseudocode
- void **afficher_pseudo_code**(pseudocode pc);

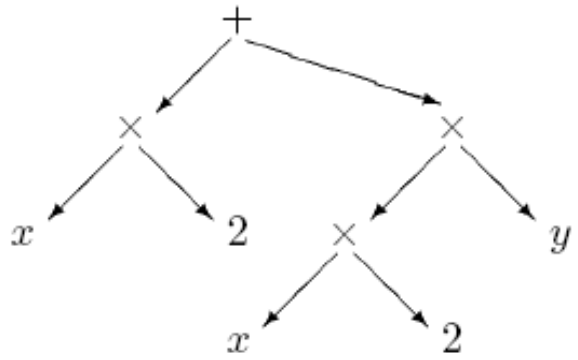
Le type abstrait de stockage et pseudo-code – Solution (pseudocode.c)

```
#include "pseudocode.h"

#define debug 0

#define NULL ((void *) 0)

void afficher_pseudo_instruction(struct pseudoinstruction pi){
  switch(pi.codop){
  case ADD:  printf("ADD\n"); break;
  case DUPL:  printf("DUPL\n"); break;
  case LOAD:  printf("LOAD "); printf("%s\n",pi.param.var);
              break;
  case MULT:  printf("MULT\n"); break;
  case POP:  printf("POP\n"); break;
  case PUSH:  printf("PUSH "); printf("%lf\n",pi.param._const);
               break;
  case SUB:  printf("SUB\n"); break;
  case STORE:  printf("STORE\n"); printf("%s\n",pi.param.var); break;
  case SWAP:  printf("SWAP\n"); break;
  }
}

// precondition pc <> NULL
void  afficher_pseudo_code(pseudocode  pc){
   if (pc != NULL){
    afficher_pseudo_instruction(pc->first);
    afficher_pseudo_code(pc->next);
  }
}
```

```
// precondition pc1 <> NULL et pc2 <> NULL
void inserer_code_en_queue(pseudocode  pc1,
                           pseudocode pc2){
  if (debug) {
    afficher_pseudo_code(pc1);
    afficher_pseudo_code(pc2);
  }

  if (pc1->next == NULL) {
    pc1->next = pc2;
  }else{
    pseudocode pc = pc1;
    while(pc->next != NULL) {
      pc = pc->next;
    }

    pc->next = pc2;
  }
  if (debug) {
    afficher_pseudo_code(pc1);
    printf("\n");
  }
}
```

Rappel du type abstrait de stockage et arbre abstrait – Solution (AST.h)

- typedef enum {NB=0, OP=1, IDF = 2} Type_Exp ;
- typedef enum {plus, moins, mult} Type_Op;
- typedef enum {false, true} boolean;
- struct Exp ; /* pré déclaration de la structure de stockage d'une expression */
- typedef struct Exp * **AST**;
- typedef union {
- double nombre ;
- char *idf;
- struct {
- Type_Op top;
- AST expression_gauche ;
- AST expression_droite ;
- } op;
- } **valueType**;
- struct **Exp** {
- Type_Exp type ;
- valueType noeud ;
- };



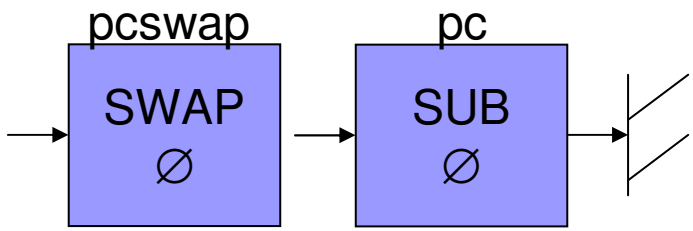
L'arbre va stocker tous les attributs calculés sur le mot de la grammaire !!
Afin de pouvoir effectuer d'autres passes (re-parcours de l'arbre et des ses attributs)

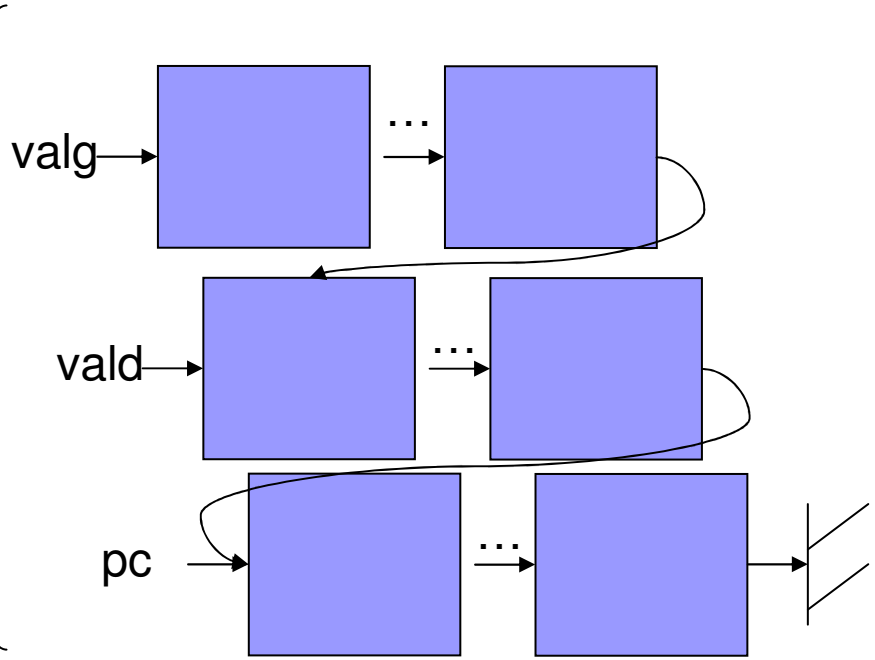
génération de pseudo-code

(AST.c)

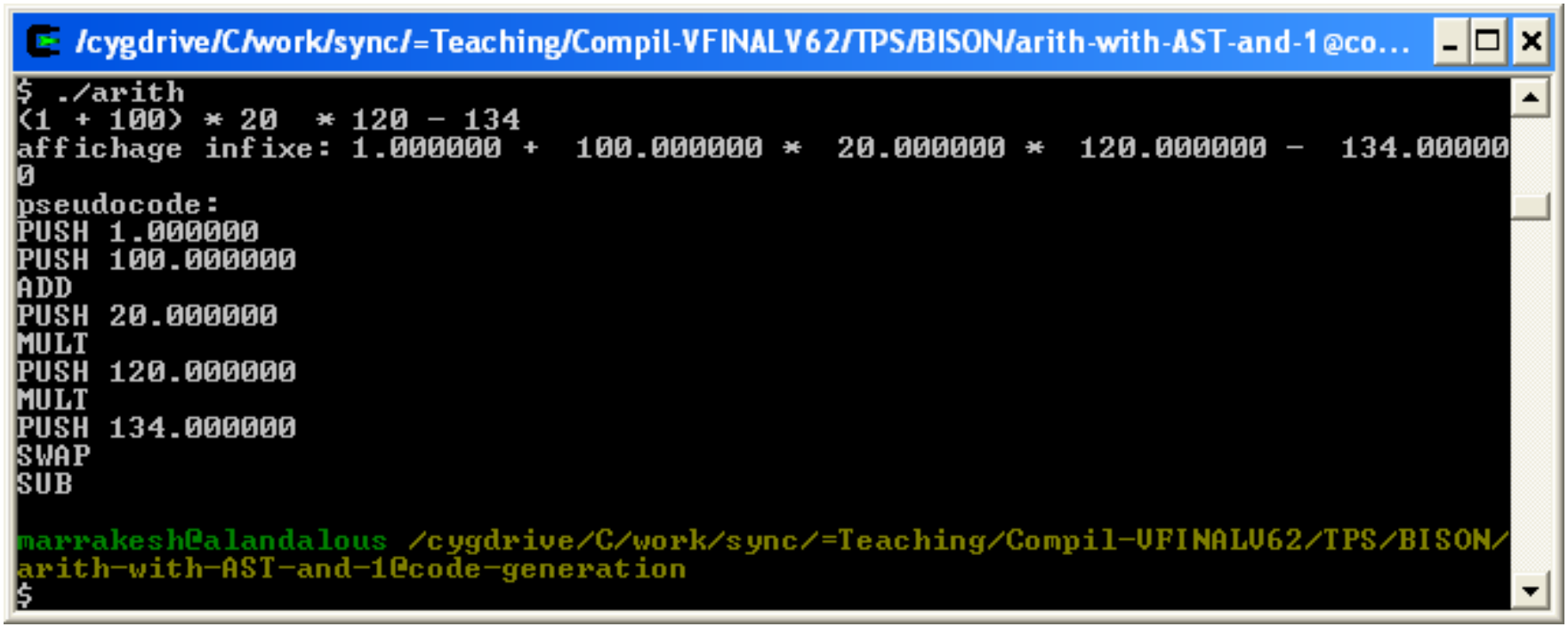
```
// génère le pseudo-code relatif à l'AST
// précondition ast <> NULL
pseudocode generer_pseudo_code(AST ast){
  pseudocode  pc = (pseudocode)malloc(sizeof (struct pseudocodenode));
  pseudocode valg, vald;

  switch(ast->type) {
  case NB :
    pc->first.codop = PUSH;
    pc->first.param._const = ast->noeud.nombre;
    break;

  case IDF :
    pc->first.codop = LOAD;
    pc->first.param.var = ast->noeud.idf;
    break;

  case OP :
    valg = generer_pseudo_code(arbre_gauche(ast));
    vald = generer_pseudo_code(arbre_droit(ast));
```

```
switch(ast->noeud.op.top){
    case plus :
      pc->first.codop = ADD; break;
      pc->next = NULL;
case moins :
      pc->first.codop = SUB; // opération non commutative
      pc->next = NULL;
      pseudocode  pcswap = (pseudocode)malloc(sizeof (struct pseudocodenode));
      pcswap->first.codop = SWAP;
      pcswap->next = pc;
      pc = pcswap;
      break;

    case mult :
      pc->first.codop = MULT; break;
      pc->next = NULL;
    }
    inserer_code_en_queue(vald, pc);
    inserer_code_en_queue(valg, vald);
    pc = valg;
break;  }

return pc;}
```





Exemple d'exécutions



Grammaire du langage ZZ étendue

- **PROG** : LISTE_DECL LISTE_INST
- **LISTE_DECL** : DECL | LISTE_DECL DECL;
- **DECL** : IDF TYPE CONST_IB
- **TYPE** : INT | DOUBLE | BOOL;
- **LISTE_INST** : INST | LISTE_INST INST
- **INST** : IDF ASSIGN TRUEFALSE
- | IDF ASSIGN EXP
- | IF PARO IDF EGAL EXP PARF THEN LISTE_INST ENDIF
- | IF PARO IDF EGAL EXP PARF THEN LISTE_INST ELSE LISTE_INST ENDIF
- | PRINT IDF
- **EXP** : EXP PLUS EXP | EXP MOINS EXP | EXP MULTIP EXP | EXP DIVIS EXP | PARO EXP PARF | ICONST | DCONST | IDF
- **CONST_IB** : ICONST | DCONST | TRUEFALSE
- **TRUEFALSE** : TRUE | FALSE

Algorithme de génération de pseudo-code pour des contrôles complexes – if-then-else

```
if cond then                    t1 = cond
    then_statements             if not t1 goto else_label_1
else                            pseudo-code de then_statements
    else_statements;            goto endif_label_1
                                else_label_1:
                                pseudo-code de else_statements;
end if;                         endif_label_1:
```

- Ajouter des instruction de type goto : JMP, JE, JNE, JGE…
- Prendre en compte les labels (PK)
- Générer les labels
- Générer les pseudo-code de pour les noeuds fils (instructions simples)
- Placer les labels dans le code
- Peut être besoin de label pour les then_statements aussi

Génération de pseudo-code pour des contrôles complexes – if-then-else : partie elsif

- Pour chaque alternative, placer dans le pseudo-code le label else_label courant et en générer un nouveau. Toutes les alternatives héritent le label de la fin de leur parent

```
if cond then s1                    t1 = cond1
                                   if not  t1 goto else_label1
                                   pseudo-code de s1
                                   goto endif_label
elsif cond2 then s2                else_label1:
                                   t2 = cond2
                                   if not t2 goto else_label2
                                   pseudo-code de s2
                                   goto endif_label
else s4                            else_label2:
                                   pseudo-code de s4
end if;                            endif_label:
```

Génération de pseudo-code pour des contrôles complexes – boucles while

- Générer deux labels: start_loop, end_loop

```
while (cond) {
    s1;
    if (cond2) break;
    s2;
    if (cond3) continue;
    s3;
};
```

```
start_loop:
 if (!cond) goto end_loop
 pseudo-code de s1
 if (cond2) goto end_loop
 pseudo-code de s2
 if (cond3) goto start_loop:
pseudo-code de s3
 goto start_loop
end_loop:
```

Génération de pseudo-code pour des contrôles complexes – boucles for

- Sémantique: la boucle ne s'exécute pas pour un intervalle vide, donc, on teste avant la première itération

```
for J in expr1..expr2 loop        J = expr1
                                  start_label:
                                  if J > expr2 goto
                                      end_label
S1                                pseudo-code de S1
end loop;                         J = J +1
                                  goto start_label
                                  end_label:
```

Génération de pseudo-code pour des contrôles complexes – boucles for (tests)

```
for K in expr1 .. expr2 loop
    S1;
end loop;
```

```
t1 = expr1
t2 = expr2
K = t1 - 1
goto test_label
start_label:
pseudo-code de S1
test_label:
K = K + 1
if K > t2 goto
end_label
goto start_label:
end_label:
```

Génération de pseudo-code pour des contrôles complexes – raccourcis

- Les expressions raccourcis sont traitées au même ordre que leur expression normalisée

- if B1 or else B2 then S1… -- if (B1 || B2) { S1..
  - if B1 goto then_label
  - if not B2 goto else_label
  - then_label:
  - *pseudo-code de* S1
  - else_label:

- Hériter des labels des expressions normalisées
- Générer les labels additionnels pour les raccourcis composés

Génération de pseudo-code pour des contrôles complexes – case (switch)

- Si l'intervalle du paramètre x est petit et la plupart des cas sont traités, on crée une table de label "jump table" comme un vecteur d'adresses, et on génère des jump indirects.

```
                                table label1, label2
                                …
case x is                       jumpi x table
when up: y := 0;                label1:
                                y = 0
                                goto end_case
when down : y := 1;             label2:
                                y = 1
                                goto end_case
end case;                       end_case:
```

Gestion de la table des labels (if then else if …) est moins optimales qu'un switch/case..

- La fonction
  - static int t (int x, int y, int z) {
    - return x+y*z;
  - }

- se compile en
  - 0:iload_0 // charger x et l'empiler
  - 1:iload_1 // charger y et l'empiler
  - 2:iload_2 // charger z et l'empiler
  - 3:imul // dépiler z et y, réaliser y * z et empiler le produit
  - 4:iadd // dépiler le produit et x, réaliser x + y * z et empiler l'addition
  - 5:ireturn // return result (soit empiler dans la pile, oubien affeter un registre réservé ex : R0)

Pseudo-code

High-level code (C style)

```
int mult(int x, int y) {
  int sum;
  sum = 0;
  for(int j = y; j != 0; j--)
      sum += x;  // Repetitive addition
  return sum;
}
```

First approximation

```
function mult
  args x, y
  vars sum, j
  sum = 0
  j = y
loop:
  if j = 0 goto end
  sum = sum + x
  j = j - 1
  goto loop
end:
  return sum
```

Pseudo VM code

```
function mult(x,y)
   push 0
   pop sum
   push y
   pop j
label loop
   push 0
   push j
   eq
   if-goto end
   push sum
   push x
   add
   pop sum
   push j
   push 1
   sub
   pop j
   goto loop
label end
   push sum
   return
```

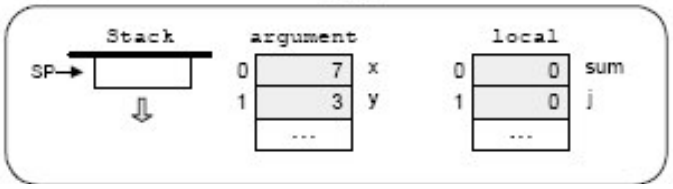
Final VM code

```
function mult 2      // Two local vars
   push constant 0
   pop local 0       // sum = 0
   push argument 1
   pop local 1       // j = y
label loop
   push constant 0
   push local 1
   eq
   if-goto end       // If j = 0 goto end
   push local 0
   push argument 0
   add
   pop local 0       // sum = sum + x
   push local 1
   push constant 1
   sub
   pop local 1       // j = j - 1
   goto loop
label end
   push local 0
   return            // Return sum
```

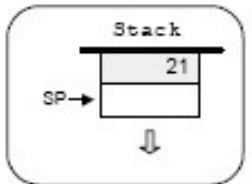
Généralement : Code final = T(pseudo-code)
Code final = Tn(Tn-1(Tn-2(Tn-3(pseudo-code))))

Exemple : Ti : optimisation ou raffinement

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



Grammaire du langage ZZ étendue

- **PROG** : LISTE_DECL LISTE_INST
- **LISTE_DECL** : DECL | LISTE_DECL DECL;
- **DECL** : IDF TYPE CONST_IB
- **TYPE** : INT | DOUBLE | BOOL;
- **LISTE_INST** : INST | LISTE_INST INST
- **INST** : IDF ASSIGN TRUEFALSE
- | IDF ASSIGN EXP
- | IF PARO IDF EGAL EXP PARF THEN LISTE_INST ENDIF
- | IF PARO IDF EGAL EXP PARF THEN LISTE_INST ELSE LISTE_INST ENDIF
- | PRINT IDF
- **EXP** : EXP PLUS EXP | EXP MOINS EXP | EXP MULTIP EXP | EXP DIVIS EXP | PARO EXP PARF | ICONST | DCONST | IDF
- **CONST_IB** : ICONST | DCONST | TRUEFALSE
- **TRUEFALSE** : TRUE | FALSE

Le type abstrait de stockage et pseudo-code étendue – Solution (pseudocode.h)

- typedef enum {ADD, *DIV*, DUPL, *JMP*, *JNE*, *LABEL*, LOAD, MULT, POP, *PRNT*, PUSH, SUB, STORE, SWAP} CODOP;

- typedef union {
- char * var;
- double _const;
- char * label_name;
- } Param;

- struct pseudoinstruction{
- CODOP codop;
- Param param;
- };

Exercice : donner le code à 1 adresse équivalent à ce programme

calculs.zz

```
Rayon INT 4

REM CASTING IMPLICITE de 0 vers 0.0
Perimetre DOUBLE 0

Surface DOUBLE 0.0
Pi DOUBLE 3.14
surfacecorrecte BOOL FALSE
perimetrecorrect BOOL FALSE

REM CASTING IMPLICITE de Rayon vers (DOUBLE) Rayon
Perimetre := 2.0 * Pi * Rayon
Surface := Pi * Rayon * Rayon

PRINT Rayon
PRINT Perimetre
PRINT Surface

IF (Surface = ((3.14 * 4.0) * 4.0) ) THEN
        surfacecorrecte := TRUE

        REM CASTING IMPLICIT DE 4
        IF (Perimetre = (2.0 * (3.14 * 4)) ) THEN
                perimetrecorrect := TRUE
        ELSE
                perimetrecorrect := FALSE
        ENDIF
ELSE
        surfacecorrecte := FALSE
ENDIF

PRINT surfacecorrecte
PRINT perimetrecorrect
```

Raw-----XEmacs: calculs.zz (Fundamental)----All----------

Wrote C:\work\sync\=Teaching\Compil-VFINALV62\TPS\BISON\TP3.1.11_interpreteur_gra

```
pseudocode generer_pseudo_code_inst(instvalueType instattribute){
  static label_index = 0;
  pseudocode  pc = (pseudocode)malloc(sizeof (struct pseudocodenode)), pc1,pc2,pc3,pc4,pc5;
  pseudocode  rexpcode;
  char * label_name;
  char *label_num;

  switch(instattribute.typeinst){

// PRINT IDF
  case PrintIdf :
    pc->first.codop = LOAD;
    pc->first.param.var = name(instattribute.node.printnode.rangvar);
    rexpcode = (pseudocode)malloc(sizeof (struct pseudocodenode));
    rexpcode->first.codop = PRNT;
    rexpcode->next = NULL;
    pc->next = rexpcode;
    break;

// IDF ASSIGN EXP
  case AssignArith :
    rexpcode = generer_pseudo_code_ast(instattribute.node.assignnode.right);

    pc->first.codop = STORE;
    pc->first.param.var = name(instattribute.node.assignnode.rangvar);
    pc->next = NULL;
    inserer_code_en_queue(rexpcode, pc);
    pc = rexpcode;
    break;
```

```
// IDF ASSIGN TRUEFALSE
case AssignBool :
    pc->first.codop = PUSH;
    pc->first.param._const = instattribute.node.assignnode.right0;

    pc1 = (pseudocode)malloc(sizeof (struct pseudocodenode));
    pc1->first.codop = STORE;
    pc1->first.param.var = name(instattribute.node.assignnode.rangvar);
    pc1->next = NULL;

    pc->next = pc1;

    break;
```

- // IF PARO IDF EGAL EXP PARF THEN LISTE_INST ENDIF

- case **IfThenArith** :
- pc = generer_pseudo_code_ast(instattribute.node.ifnode.right);

- pc1 = (pseudocode)malloc(sizeof (struct pseudocodenode));
- pc1->first.codop = **LOAD**;
- pc1->first.param.var = name(instattribute.node.ifnode.rangvar);
- pc1->next = NULL;

- pc2 = (pseudocode)malloc(sizeof (struct pseudocodenode));
- pc2->first.codop = **JNE**;
- label_num=itoa(label_index++);
- pc2->first.param.label_name = (char*) malloc(6+strlen(label_num));
- strcpy(pc2->first.param.label_name, "endif");
- strcat(pc2->first.param.label_name, label_num);
- pc2->next = NULL;

- pc3 = generer_pseudo_code_list_inst(instattribute.node.ifnode.thenlinst);

- pc4 = (pseudocode)malloc(sizeof (struct pseudocodenode));
- pc4->first.codop = **LABEL**;
- pc4->first.param.label_name = pc2->first.param.label_name;
- pc4->next = NULL;

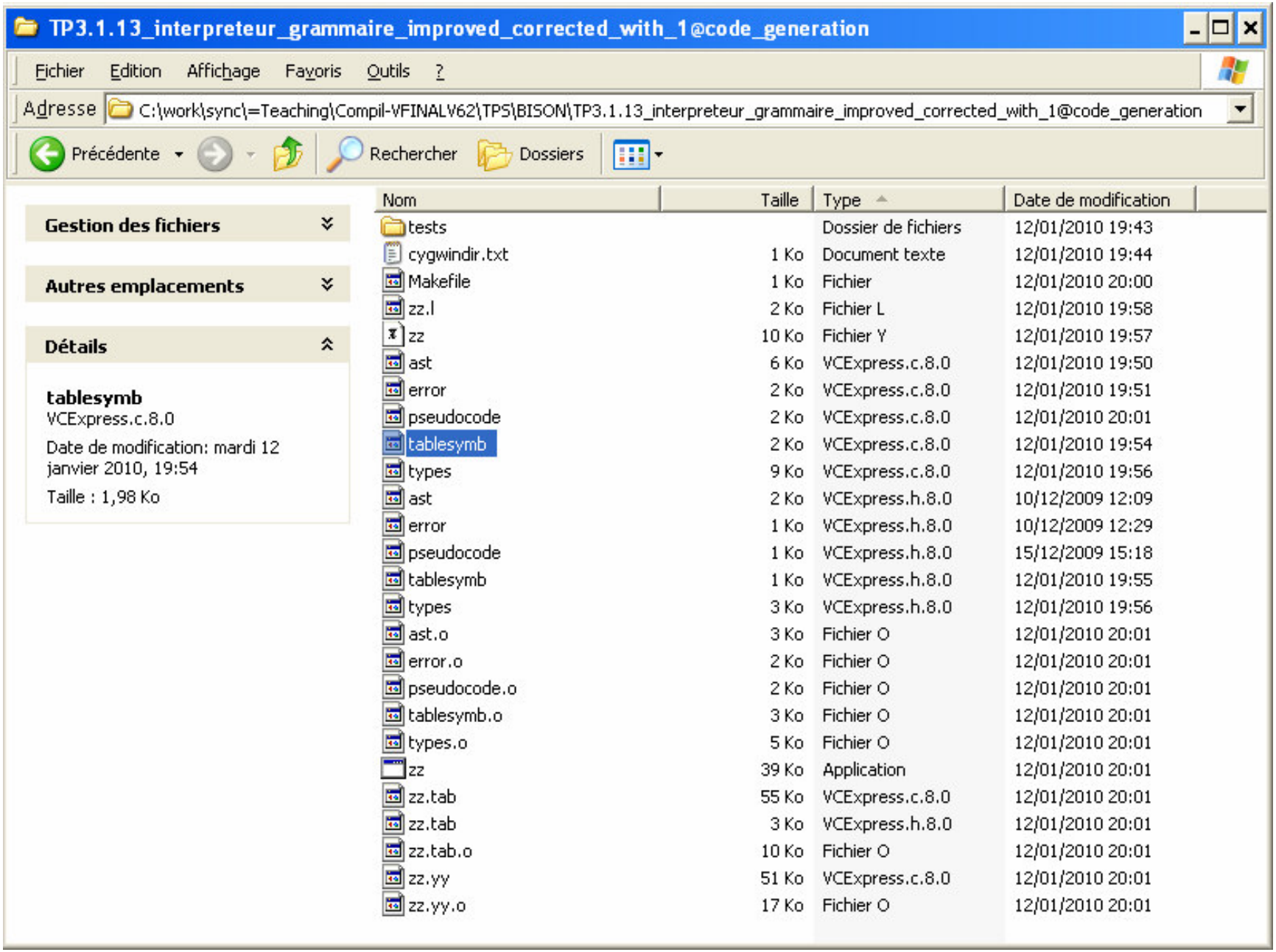
- inserer_code_en_queue(pc3, pc4); pc2->next = pc3; pc1->next = pc2; inserer_code_en_queue(pc, pc1);
- break;

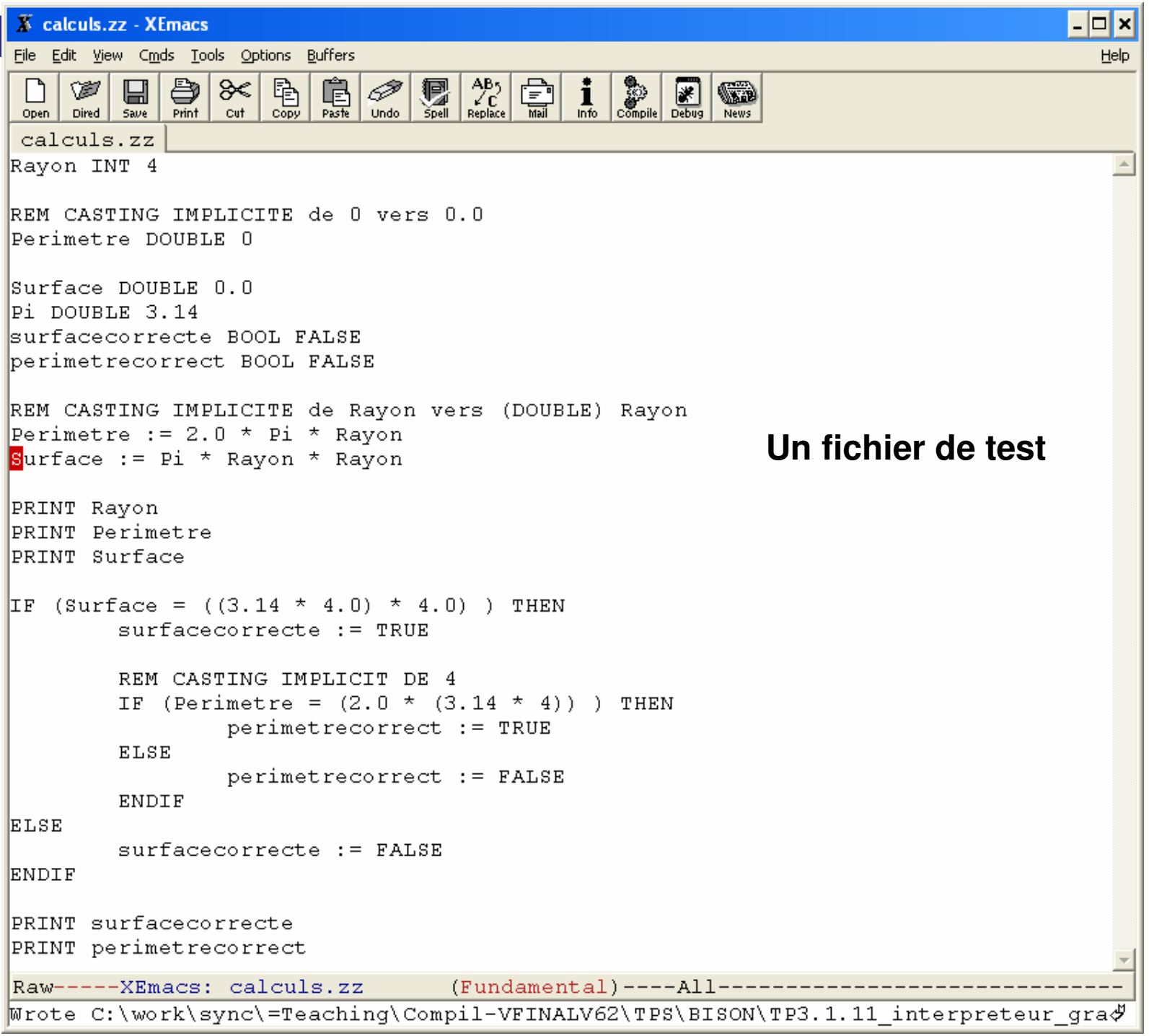
```
// IF PARO IDF EGAL EXP PARF THEN LISTE_INST ELSE LISTE_INST ENDIF
case IfThenElseArith :
  pc = generer_pseudo_code_ast(instattribute.node.ifnode.right);

    pc1 = (pseudocode)malloc(sizeof (struct pseudocodenode));
    pc1->first.codop = LOAD;
    pc1->first.param.var = name(instattribute.node.ifnode.rangvar);
    pc1->next = NULL;

    pc2 = (pseudocode)malloc(sizeof (struct pseudocodenode));
    pc2->first.codop = JNE;
    label_num=itoa(label_index++);
    pc2->first.param.label_name = (char*) malloc(6+strlen(label_num));
    strcpy( pc2->first.param.label_name, "else");
    strcat( pc2->first.param.label_name, label_num);
    pc2->next = NULL;

    pc3 = generer_pseudo_code_list_inst(instattribute.node.ifnode.thenlinst );

    pc31 = (pseudocode)malloc(sizeof (struct pseudocodenode));
    pc31->first.codop = JMP;
    pc31->first.param.label_name = (char*) malloc(6+strlen(label_num));
    strcpy( pc31->first.param.label_name, "endif");
    strcat( pc31->first.param.label_name, label_num);

    pc4 = (pseudocode)malloc(sizeof (struct pseudocodenode));
    pc4->first.codop = LABEL;
    pc4->first.param.label_name = pc2->first.param.label_name;
    pc4->next = NULL;

    pc31->next = pc4;
```

```
    pc5 = generer_pseudo_code_list_inst(instattribute.node.ifnode.elselinst );
    pc4->next = pc5;

    pc6 = (pseudocode)malloc(sizeof (struct pseudocodenode));
    pc6->first.codop = LABEL;
    pc6->first.param.label_name = (char*) malloc(strlen( pc31->first.param.label_name)+1);
    strcpy( pc6->first.param.label_name,  pc31->first.param.label_name);
    pc6->next = NULL;

    inserer_code_en_queue(pc5, pc6);
    inserer_code_en_queue(pc3, pc31);
    pc2->next = pc3;
    pc1->next = pc2;
    inserer_code_en_queue(pc, pc1);

    break;
   }

   return pc;
```

```
void afficher_pseudo_instruction(struct pseudoinstruction pi){
  switch(pi.codop){
  case ADD:   printf("ADD\n");  break;
  case DIV:   printf("DIV\n"); break;
  case DUPL:  printf("DUPL\n"); break;
  case LABEL: printf("%s:\n",pi.param.label_name); break;
  case LOAD:  printf("LOAD "); printf("%s\n",pi.param.var); break;
  case MULT:   printf("MULT\n"); break;
  case POP:   printf("POP\n");  break;
  case PUSH:  printf("PUSH "); printf("%lf\n",pi.param._const); break;
  case SUB:   printf("SUB\n");  break;
  case STORE: printf("STORE "); printf("%s\n",pi.param.var); break;
  case SWAP:  printf("SWAP\n"); break;
  case PRNT:  printf("PRINT\n");break;
  case JNE:   printf("JNE "); printf("%s\n",pi.param.label_name);break;
  case JMP:   printf("JMP "); printf("%s\n",pi.param.label_name);break;
  }
}
```

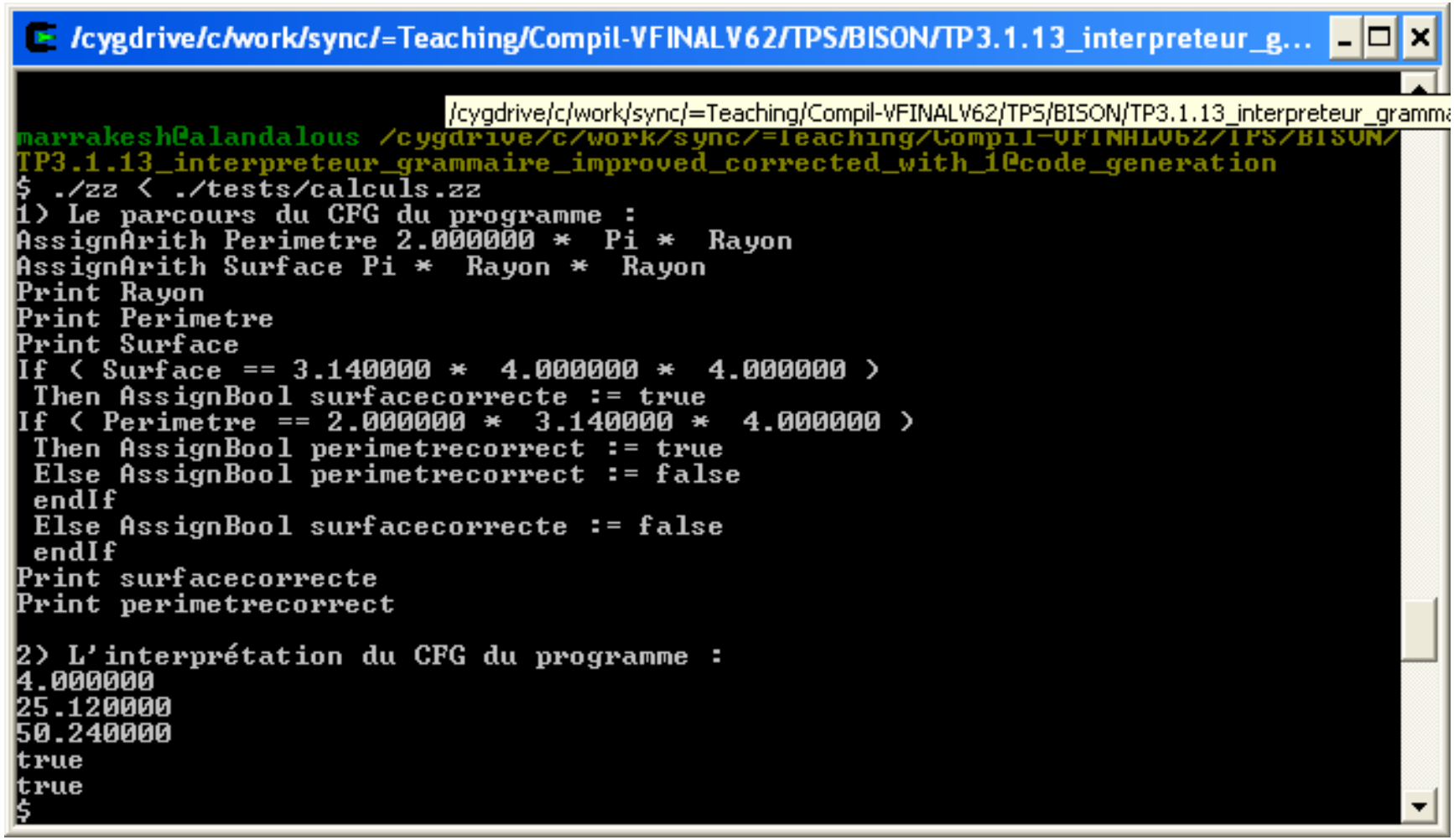
Exploration des fichiers de code


TP3.1.13_interpreteur_grammaire_improved_corrected_with_1@code_generation
Fichier Edition Affichage Favoris Outils ?
Adresse C:\work\sync\=Teaching\Compil-VFINALV62\TPS\BISON\TP3.1.13_interpreteur_grammaire_improved_corrected_with_1@code_generation
Précédente Rechercher Dossiers
Gestion des fichiers
Autres emplacements
Détails
tablesymb
VCExpress.c.8.0
Date de modification: mardi 12 janvier 2010, 19:54
Taille : 1,98 Ko
Nom Taille Type Date de modification
tests Dossier de fichiers 12/01/2010 19:43
cygwindir.txt 1 Ko Document texte 12/01/2010 19:44
Makefile 1 Ko Fichier 12/01/2010 20:00
zz.l 2 Ko Fichier L 12/01/2010 19:58
zz 10 Ko Fichier Y 12/01/2010 19:57
ast 6 Ko VCExpress.c.8.0 12/01/2010 19:50
error 2 Ko VCExpress.c.8.0 12/01/2010 19:51
pseudocode 2 Ko VCExpress.c.8.0 12/01/2010 20:01
tablesymb 2 Ko VCExpress.c.8.0 12/01/2010 19:54
types 9 Ko VCExpress.c.8.0 12/01/2010 19:56
ast 2 Ko VCExpress.h.8.0 10/12/2009 12:09
error 1 Ko VCExpress.h.8.0 10/12/2009 12:29
pseudocode 1 Ko VCExpress.h.8.0 15/12/2009 15:18
tablesymb 1 Ko VCExpress.h.8.0 12/01/2010 19:55
types 3 Ko VCExpress.h.8.0 12/01/2010 19:56
ast.o 3 Ko Fichier O 12/01/2010 20:01
error.o 2 Ko Fichier O 12/01/2010 20:01
pseudocode.o 2 Ko Fichier O 12/01/2010 20:01
tablesymb.o 3 Ko Fichier O 12/01/2010 20:01
types.o 5 Ko Fichier O 12/01/2010 20:01
zz 39 Ko Application 12/01/2010 20:01
zz.tab 55 Ko VCExpress.c.8.0 12/01/2010 20:01
zz.tab 3 Ko VCExpress.h.8.0 12/01/2010 20:01
zz.tab.o 10 Ko Fichier O 12/01/2010 20:01
zz.yy 51 Ko VCExpress.c.8.0 12/01/2010 20:01
zz.yy.o 17 Ko Fichier O 12/01/2010 20:01

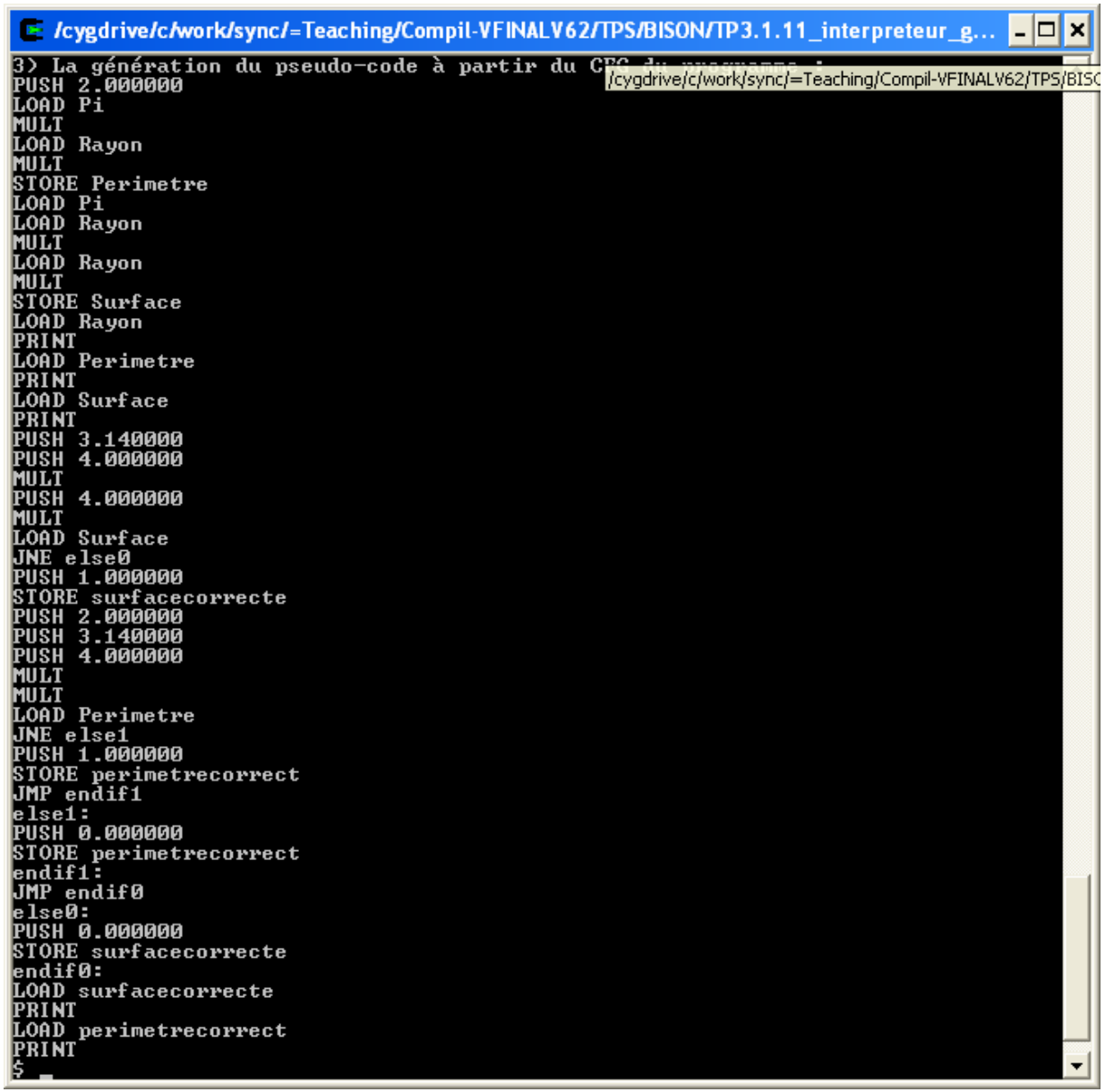


Un fichier de test

1-2) CFG : Affichage & interprétation



3) CFG : génération de pseudo-code



Interprétation du Pseudo-code

Boucle d'interprétation du pseudo-code

- typedef enum {ADD, *DIV*, DUPL, JMP, *JNE*, *LABEL*, LOAD, MULT, POP, *PRNT*, PUSH, SUB, STORE, SWAP} CODOP;

- void interpreter (pseudocode ip) {
  - while (ip->next != NULL) {
    - switch (ip->first.codop) {
    - case **LOAD**:
      - empiler(*(@ARG1))
    - case **PUSH**:
      - empiler(ARG1);
    - case **STORE**:
      - (@ARG1) := dépiler()
    - case **SWAP**:
      - (@TMP1) := dépiler() ; (@TMP2) := dépiler()
      - empiler(*(@TMP2)) ; empiler(*(@TMP1)); break;
    - case **ADD**:
      - empiler(dépiler() + dépiler()); break;
    - case **SUB**:
      - empiler(dépiler() - dépiler()); break;
    - case **MULT**:
      - empiler(dépiler() * dépiler()); break;
    - case **DIV**:
      - empiler(dépiler() / dépiler()); break;
    - …
    - } }

Il faut un TAD Pile pour simuler la pile Système

Génération du Pseudo-code – **2-address**

Allocation des registres – *selon code à 2-adresses*

1. Pour stocker les résultats intermédiaires (temporaires), les plus utilisés, dans des registres deux tactiques existent :
   1. Tactique (1) : Supposer le nombre de registres de la machine infini
      - Créer de nouveaux emplacements temporaires pour chaque résultat intermédiaire
   2. Tactique (2) « Meilleur modèle » : poser un nombre fini de registres
      - Soit une pile des **registres disponibles** (ordre LIFO d'allocation/libération de registres)
      - Sélectionner un registre R1 pour contenir le résultat
      - Calculer le premier opérande dans R1
      - Calculer le second opérande en utilisant des registres restant
      - Calculer le résultat dans R1

- Problème : Comment minimiser le nombre des registres nécessaires ?
  - Ce problème est NP-dur. Cependant des heuristiques existent.

Algorithme de calcul du nombre de registres minimum (algorithme de Aho-Sethi)

- Le nombre de registres minimum est un attribut que l'on calcule puis on le stocke dans la RI graphique. L'algorithme suivant (dit de Aho-Sethi) détaille son système de calcul
  - **NBREG_MIN**(AST)
    - Pour une constante (AST = CONST) : return **1** ; // **LOAD CONST, R**
    - Pour une variable (AST = IDF) : return **1** ; // **LOAD IDF, R**
    - Pour un arbre (AST = EXP):
      - min_g := **NBREG_MIN**(arbre_gauche(AST)) ;
      - min_d := **NBREG_MIN**(arbre_droit(AST)) ;
      - if (min_g /= min_d) then
        - return max (min_g, min_d) ;
      - else
        - return min_g + 1 ;

NBREG_MIN peut être calculé en même temps que la construction de l'AST (pendant la 1ère passe)

Génération de pseudo-code – *selon code à 2-adresses*

- Traverser la RI graphique (ex. AST) et à chaque niveau, **générer le pseudo-code du sous arbre dont l'attribut est plus grand.**
  - Au fait, Calculer **le plus grand arbre** (prioritairement par rapport au nombre de registres nécessaires) en premier, vient du fait qu'il va libérer **nombre de registres – 1** >= **min(gauche,droit)** pour calculer l'expression moins lourde qui sera amplement satisfaite avec ces registres libérés

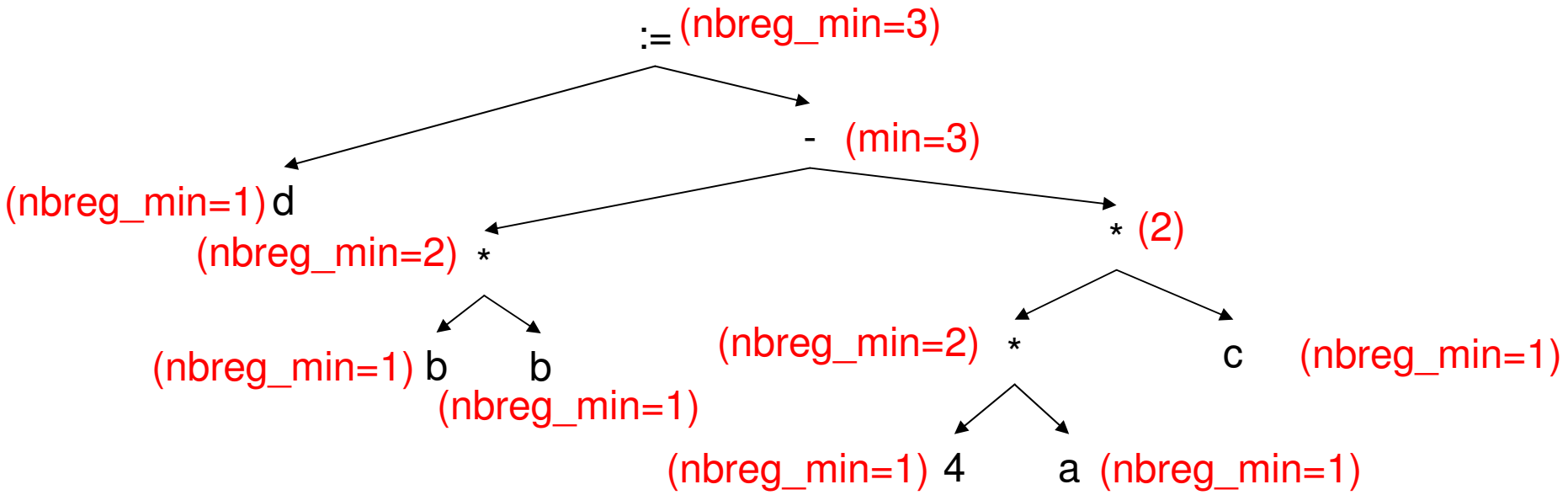
- Si NBREG_MIN(arbre_gauche(AST)) == NBREG_MIN(arbre_droit(AST)) consomment le même nombre de registres, on choisit de générer le code de l'arbre gauche, puis celui de l'arbre droit

Générer le pseudo-code d'expressions – ***selon code à 2-adresses :*** Exercice

- Soit l'expression arithmétque suivante « d := b*b – 4*a*c »
  a. De combien de registres aura besoin l'évaluation de cette expression ?
  b. Proposer une solution de pseudo-code avec le jeu d'instructions à 2-adresses LOAD, MULT, SUB

Générer le pseudo-code d'expressions – *selon code à 2-adresses : Solution*

- b*b – 4*a*c : a besoin de 3 registres

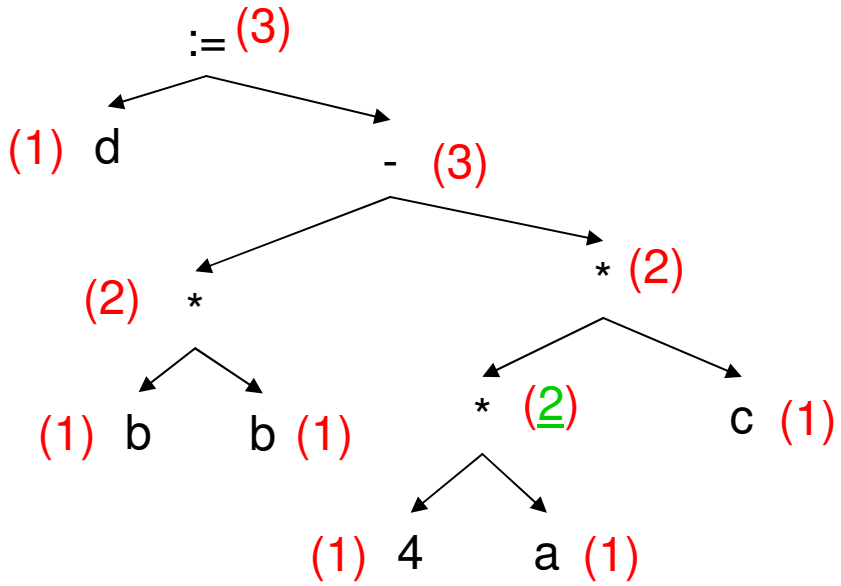


- Conventions :
  - Je génère le code de l'Expression la plus consommatrice de registres, si arbre_gauche et arbre_droit ont le même nombre de registres, je peux choisir l'arbre le plus à gauche.
  - Convention MOTOROLA

Générer le pseudo-code d'expressions – ***selon code à 2-adresses :*** *Solution 1 (associativité droite)*

1. Conventions :
   1. La machine abstraite contient N registres de données
   2. Les registres sont consommés selon un ordre LIFO croissant R_1-R_N
   3. Le registre R1 contiendra le résultat du calcul final
   4. Quand les deux sous arbres gauche et droit sont de la même taille (en termes de nombre de registres, on commence par évaluer le sous-arbre gauche (ce qui est le cas de Spécificité de $B^2 – 4AC$)
   5. SENS MOTOROLA

2. LOAD B,R1
3. LOAD B,R2
4. MULT R2,**R1** // **R1** = b^2, R2 est libre
5. LOAD 4,R2
6. LOAD A,R3
7. MULT R3,**R2** //**R2** = 4A, R3 est libre
8. LOAD C,R3
9. MULT R2,**R3** //**R3** = 4AC, R2 est libre
10. SUB **R3**,**R1** // **R1** = R1 – R3
11. **STORE R1, D**



Générer le pseudo-code d'expressions – ***selon code à 2-adresses :*** *Solution 2 (associativité gauche)*

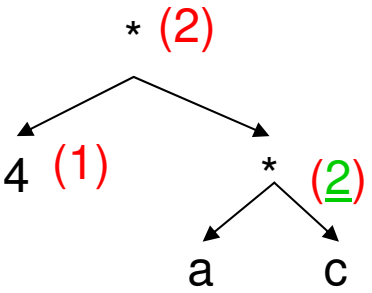
- b*b – 4*a*c : a besoin de 3 registres
- Conventions : Registre R_1-R_N et SENS INTEL

Naïve gauche-à-droite (R1-R4)

```
LOAD R1, b
LOAD R2, b
MULT R1, R2, R1

LOAD R2, 4
LOAD R3, a
LOAD R4, c
MULT R3, R4, R3
MULT R2, R3, R2
SUB R1, R2
STORE R, D
```

optimale (R1-R3)

```
LOAD R1, b
LOAD R2, b
MULT R1, R2, R1

LOAD R2, a
LOAD R3, c
MULT R2, R3, R2↑
LOAD R3, 4
MULT R2, R3, R2
SUB R1, R2
STORE R1, D
```



Génération du Pseudo-code – **3-address**

Générer le pseudo-code d'expressions – ***selon code à 3-adresses :*** Exercice

- Soit l'expression arithmétque suivante « d := b*b – 4*a*c »
  a. Proposer une solution de pseudo-code avec le jeu d'instruction à 3-adresses LOAD, MULT, SUB

Solution de B2 – 4AC à 3-adresse

1. Conventions :
    1. 4 registres
2. LOAD B,R1
3. LOAD B,R2
4. MULT R1,R2, **R3** // **R3** = b^2, R1 et R2 sont libres
5. LOAD 4,R1
6. LOAD A,R2
7. MULT R1,R2,**R4** //**R4** = 4A, R1 et R2 sont libres
8. LOAD C,R1
9. MULT R1,R4, **R2** // **R2** = 4AC, R1, R4 est libre
10. SUB **R3**,**R2**,**R1** // **R1** = R3 – R2
11. **STORE R1, D**

Compilation

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Lectures recommandées - livres

1. « Engineering a Compiler » ©, K. D. Cooper & L. Torczon, Morgan Kaufmann 2004
2. « Introduction to Automata Theory Languages and Computation » ©, J. E. Hopcroft, R. Motwani, J. D.Ullman, 2001, Addison Wesley
3. « Compilers and Compiler Generators » ©, P.D. Terry, 2000
4. Cours « Compilation et interprétation », Danny Dubé, 2006
5. Cours « Compilation », C. Paulin (Université Paris Sud), 2009-2010

Plan du cours

1. Module initial Motivation & Définitions
2. Module 1 Analyseur lexical *(langages réguliers et théorie des automates)*
3. Module 2 Analyseur syntaxique *(langages hors contexte)*
4. Module 3 Analyseur sémantique *(grammaires attribuées)*
5. Module 4 Représentations Intermédiaires
6. Module 5 Génération de code intermédiaire
7. **Module 6** Optimisation et génération de code
8. Conclusion et perspectives

Module 6
Optimisation et génération de code

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Optimisation et génération de code

- À partir de la RI linéaire (pseudo-code), une nouvelle passe va optimiser ce code : choisir une implémentation
  - Plus rapide
  - Utilisant moins de mémoire
  - Utilisant moins de registres
  - Utilisant moins de puissance durant l'exécution.

  - Ces optimisations ne sont pas toutes corrélées positivement. Il faut donc appliquer des heuristiques.

Types d'optimisations *(entre autres)*

1. Simplification d'opérations
2. Simplification de suite d'instructions
3. Élimination des calculs redondantes
4. Propagation de copies
5. Élimination du code mort
6. Optimisation des régions de calcul intensif
   - Les boucles, la récursivité

Compromis d'une optimisation

- coût d'implantation d'une optimisation
- *Versus*
- amélioration espérée des programmes

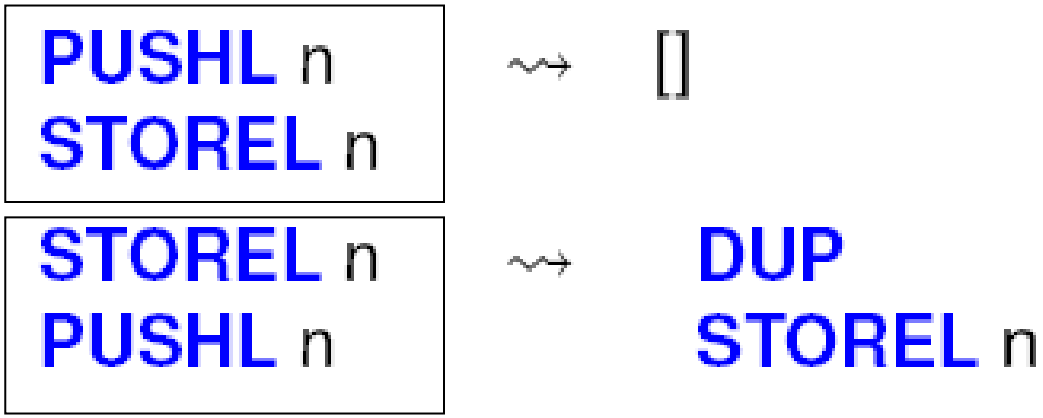
(1) Simplification d'opérations

- Certaines opérations arithmétiques peuvent être implantées par des opérations machines différentes:
  - 2 * x est équivalent à x + x ou à un shift right de x
  - x / 2 est équivalent à x * 0.5 ou à un shift left de x
  - x**2 est équivalent à x * x

- On choisit les opérations les moins coûteuses.

(2) Simplification d'une suite d'instructions

- On peut regarder linéairement le code engendré et repérer des suites d'instructions qui peuvent être simplifiées.



- Attention, on doit s'assurer que les instructions sont toujours exécutées séquentiellement (pas d'étiquette arrivant sur la seconde instruction).

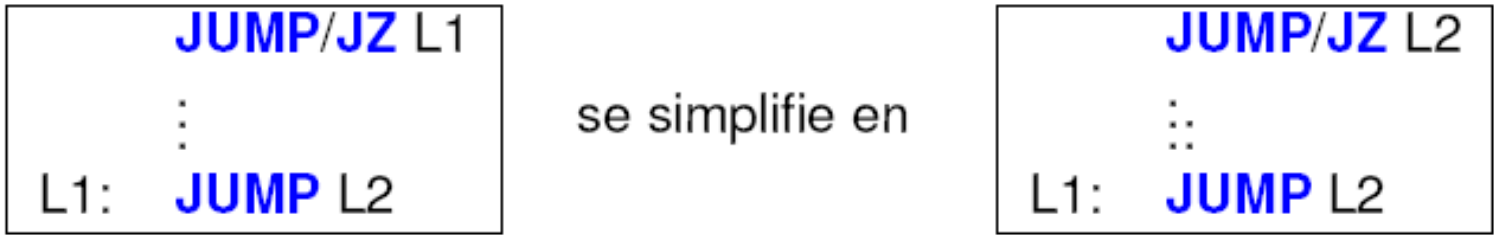
(2) Simplification d'une suite d'instructions

- Une instruction non étiquetée après un branchement inconditionnel est
- du code mort.

- Cette situation peut arriver après propagation de constante:
- **if** debug **then** code1
- code2
- est compilé en :
  - **PUSHI** debug
  - **JZ** endif
  - code1
  - endif : code2
- Si debug=false alors on peut simplifier
  - **PUSHI** 0
  - **JZ** endif
- est compilé en
  - **JUMP** endif
  - endif : code2
- la partie code1 est alors du code mort non généré

(2) Simplification d'une suite d'instructions

- Les branchements chainés peuvent être factorisés:



- Schéma plus complexe

- Si aucune instruction ne mène plus à une étiquette L1, le bloc pourra être supprimé.

(3) Élimination des calculs redondants

- Considérons l'extrait de programme suivant:
  - *ti := E*
  - *. . .*
  - *tj := E*

- L'expression E dans la deuxième instruction est considérée comme redondante si le contrôle doit toujours passer par la première instruction avant de se rendre à la seconde et si ni les variables dans *E* ni *ti* ne sont modifiées sur le chemin entre les deux.

- On peut alors éliminer un des deux calculs de E en remplaçant la seconde instruction par:

  - *tj := ti*

(4) Propagations de copies (des constantes)

- Considérons l'extrait de programme suivant:
  - *ti := ...*
  - *. . .*
  - *tj := ti*
  - *. . .*
  - *tk := E(tj)*

- Si la variable *ti* originalement calculée par la première instruction demeure disponible, alors on peut récrire la dernière instruction ainsi:

  - *tk := E(ti)*

- Ultimement, grâce à ce type de transformation, on pourrait rendre l'instruction de copie inutile. Similairement, on peut remplacer une référence à une variable affectée à une valeur constante par la constante elle-même.

(5) Élimination du code mort

- Des instructions qui sont inaccessibles à cause d'instructions de saut sont considérées comme du code mort et peuvent être éliminées :
  - *if (1 <> 0) goto label*
    - *i := a[k]*
    - *k := k + 1*
  - *label:*

- Des instructions qui calculent la valeur de variables qui sont mortes (*dont la valeur n'est pas nécessaire pour le reste du calcul*) sont aussi considérées comme du code mort et peuvent être éliminées, en autant que le calcul n'a pas d'effets de bord.

(6) Optimisations de boucles

- Beaucoup de programmes passent une grande partie de leur temps dans des boucles internes.
- Intérêt d'optimiser ces boucles
  - Faire sortir des instructions constantes (invariants)
  - Éviter des calculs redondants
- L'optimisation des boucles est particulièrement intéressante car l'intérieur des boucles est typiquement exécuté intensivement.
- Les transformations que l'on peut appliquer aux boucles incluent:
  - l'identification et l'élimination des variables d'induction
  - le déplacement de code hors des boucles et
  - l'affaiblissement d'instructions

(6.1) Optimisations de boucles : l'identification et l'élimination des variables d'induction

- Une variable dont la valeur augmente ou diminue de façon régulière à chaque itération d'une boucle est appelée variable d'induction.

- Parfois, deux variables d'induction '*i*' et '*j*' sont corrélées (l'une est fonction linéaire de l'autre)
  - $j = a \times i + b$

- Si plusieurs variables d'induction sont corrélées, on en garde parfois une seule (disons '*j*') en réinterprétant les opérations sur '*j*' en fonction de '*i*'.

(6.2) Optimisations de boucles : Déplacement de code

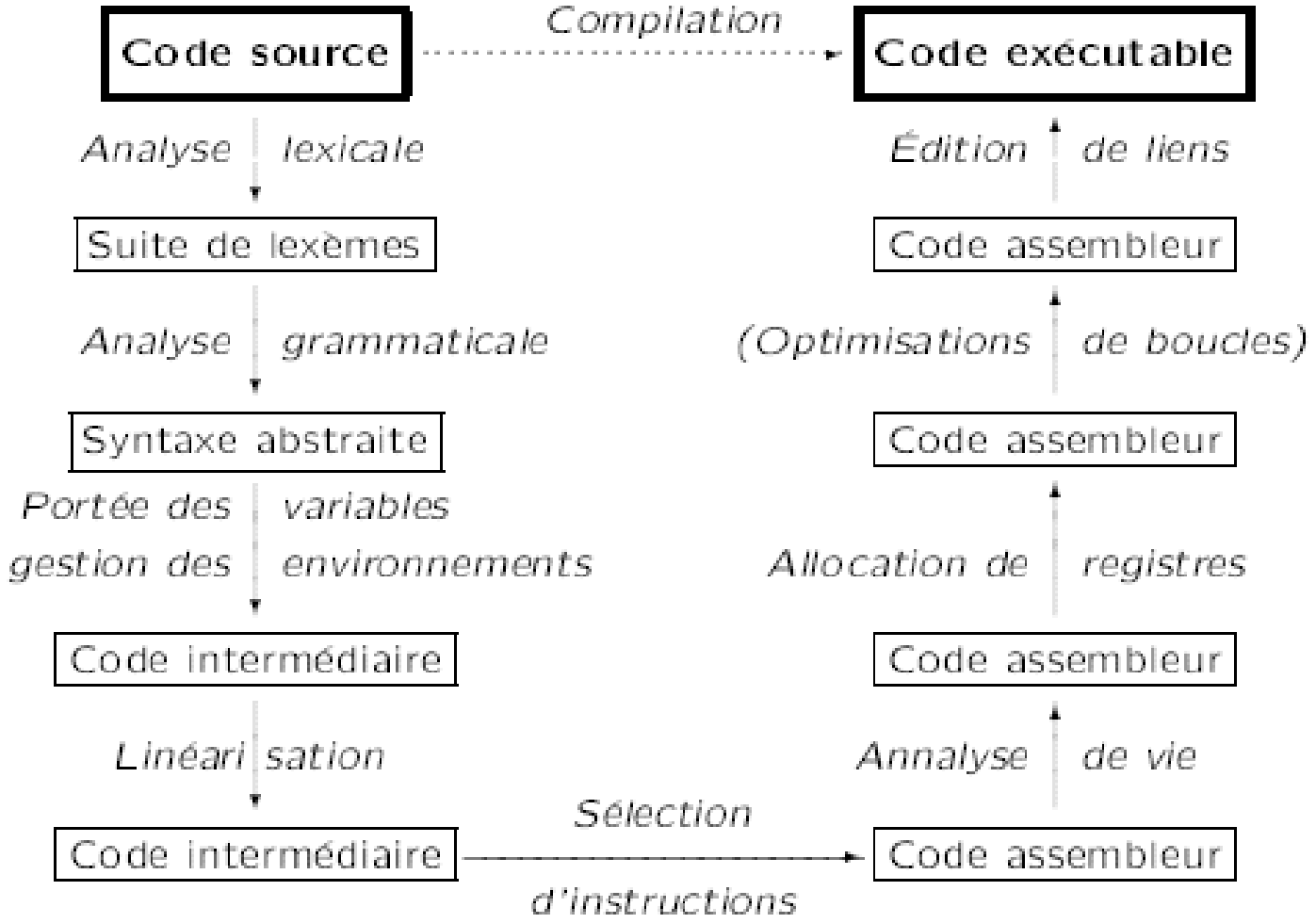
- Un calcul interne à une boucle et dont les opérandes sont constants dans la boucle est appelé invariant de boucle et peut être déplacé en dehors de la boucle.

- Supposons que *E* soit un calcul invariant dans la boucle suivante:
  - *while (...)*
    - *. . .*
    - *x := E*
    - *. . .*

- Alors, on peut déplacer le calcul de E en dehors de la boucle:
  - *t123 := E*
  - *while (...)*
    - *. . .*
    - *x := t123*
    - *. . .*

(6.3) Optimisations de boucles : Affaiblissement d'instruction

- Lorsque l'on a deux variables d'induction qui sont liées par un calcul considéré coûteux comme dans :
  - while (...)
    - . . .
    - i := i + 1
    - x := 4 * i
    - . . .
- alors on affaiblir l'instruction utilisée pour faire le calcul et espérer accélérer le code (l'incrémentation est moins coûteuse que la multiplication) :
  - x := 4 * i
  - while (...)
    - . . .
    - i := i + 1
    - x := x + 4
    - . . .


Code source
Compilation
Code exécutable
Analyse lexicale
Suite de lexèmes
Analyse grammaticale
Syntaxe abstraite
Portée des variables
gestion des environnements
Code intermédiaire
Linéarisation
Code intermédiaire
Sélection d'instructions
Code assembleur
Annalyse de vie
Code assembleur
Allocation de registres
Code assembleur
(Optimisations de boucles)
Code assembleur
Édition de liens
Code assembleur

Compilation

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Lectures recommandées - livres

1. « Engineering a Compiler » ©, K. D. Cooper & L. Torczon, Morgan Kaufmann 2004
2. « Introduction to Automata Theory Languages and Computation » ©, J. E. Hopcroft, R. Motwani, J. D.Ullman, 2001, Addison Wesley
3. « Compilers and Compiler Generators » ©, P.D. Terry, 2000

Plan du cours

1. Module initial Motivation & Définitions
2. Module 1 Analyseur lexical *(langages réguliers et théorie des automates)*
3. Module 2 Analyseur syntaxique *(langages hors contexte)*
4. Module 3 Analyseur sémantique *(grammaires attribuées)*
5. Module 4 Représentations Intermédiaires
6. Module 5 Génération de code intermédiaire
7. Module 6 Optimisation de code
8. Module 7 Génération de code
9. Module final Conclusion et perspectives

Module 8 Génération de code

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Code final

- Une dernière passe va transformer ce pseudo-code en code final
  - code pour une autre machine abstraite,
  - code assembleur natif,
  - code parallélisé (threaded code),
  - code machine exécutable, etc.

Compilation

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

Lectures recommandées - livres

1. « Engineering a Compiler » ©, K. D. Cooper & L. Torczon, Morgan Kaufmann 2004
2. « Introduction to Automata Theory Languages and Computation » ©, J. E. Hopcroft, R. Motwani, J. D.Ullman, 2001, Addison Wesley
3. « Compilers and Compiler Generators » ©, P.D. Terry, 2000

Plan du cours

1. Module initial Motivation & Définitions
2. Module 1 Analyseur lexical *(langages réguliers et théorie des automates)*
3. Module 2 Analyseur syntaxique *(langages hors contexte)*
4. Module 3 Analyseur sémantique *(grammaires attribuées)*
5. Module 4 Représentations Intermédiaires
6. Module 5 Génération de code intermédiaire
7. Module 6 Optimisation de code
8. Module 7 Génération de code
9. Module final Conclusion et perspectives

8. Conclusions et perspectives

Prof. Habilité Karim Baïna
baina@ensias.ma
www.ensias.ma/ens/baina

***ENSIAS*,**
Mohammed V - Souissi University,
Rabat, *Morocco*

Prof. K. Baïna 2010 ©, 7ème édition

L'avenir de la compilation

1. Au tout début, il y a eu les langages machines,
2. Puis les assembleurs,
3. Ensuite les langages de haut niveaux,
4. Et maintenant, on parle d'approche orientée modèles Model-Driven Approach (MDA)
   - Travail sur les modèles et non sur le code
     - Modèles : Physique, Mathématique, Logique, Objet, Relationnel, Pétri, Workflow, etc.
   - Analyse et vérification des modèles
   - Exécution des modèles
   - Transformations entre Modèles
   - Génération de code à partir de Modèles
   - Extraction du modèle à partir de code (reverse engineering)