Support de cours

Compilation

2013 - 2014
M1 Informatique

Anne BERRY

Cours de Compilation

Anne BERRY

ENT : M 1_INFO_COMPIL

Bibliographie

1. Cours en ligne Keith Schwarz http://www.stanford.edu/class/cs143/
2. Des WATSON - High-Level Languages and Their Compilers - ISBN 0-201-18489-3
3. Daniel I.A. COHEN - Introduction to Computer Theory ISBN 0-471-54841-3
4. John HOPCROFT and Jeffrey ULLMAN - Introduction to Automata Theory, Languages and Computation
5. Alfred AHO, Monica S. Lam, Ravi SETHI and Jeffrey ULLMAN - Compilers, Principles, Techniques and Tools
6. Andrew W. APPEL - Modern compiler implementation in C
7. R. WILHELM and D. MAURER - Compiler Design
8. W. WAITE and G. GOOS - Compiler Construction
9. S. MUCHNICK - Advanced Compiler design Implementation
10. D. GRUNE, H. BAL, C. JACOBS et K. LANGENDOEN - Compilateurs

Plan du cours :

1 : Introduction: Qu'est-ce que la Compilation
2 : Quelques rappels de théorie des langages
3 : L'analyse lexicale
4 : L'analyse syntaxique
5 : Les fonctions FIRST et FOLLOW
6 : L'analyse descendante (langages LL)
7 : L'analyse ascendante (langages LR)

(+ Génération et optimisation de code : quelques notions)

Table des matières

1. Introduction

"*To all who know more than one language*" (Waite & Goos)

Remarque : la plupart des documents sont en anglais, ainsi que des termes techniques utilisés.

1.1. Définition de la compilation

Un compilateur est un programme de traduction.

De façon générale :
- On part d'un code source
écrit dans un 'langage de haut niveau' (langage de programmation, C, C++, Java, LISP...) qui est “facile” à manipuler pour un informaticien.
- Le compilateur traduit dans un langage cible (de type Assembleur), facile à manipuler pour l'ordinateur.

Ce langage cible est ensuite optimisé.

Remarque : Il existe d'excellents compilateurs, mais il n'existe pas d'excellent traducteur d'une langue naturelle à une autre !

Buts de la compilation :
- changer la forme tout en conservant la sémantique.
- Signaler les erreurs rencontrées et aider l'informaticien à les corriger.

La traduction s'effectue en plusieurs phases.

Chaîne de développement d'un programme

Enoncé d'un problème à résoudre
Analyse du problème (décidabilité, classe de complexité)

↓

Algorithme

↻ Analyse de complexité (moyenne, pire cas, expérimentale; recherche de meilleurs algorithmes)

↓

Choix d'un langage de programmation

↓

Ecriture d'un code source

↻ **débuggage**

↓ **Compilation**

Génération du code cible

↻ **Optimisation**

1.2. Quelques problèmes posés

le choix du langage

portabilité

lisibilité

optimisation du code

Un **méta-langage** est un langage utilisé pour définir un autre langage (par exemple des règles de grammaire).

Le langage peut servir à se décrire lui-même.

"Un des buts de l'étude de la compilation pour un programmeur est de savoir choisir le bon langage." (Watson)

La gestion des erreurs

- Problèmes d'IHM (Interface Homme/Machine) : clarté des messages d'erreur, aide efficace au débuggage.
- Quelles 'initiatives' peut-on laisser au compilateur ? (reparenthésage, retypage)
- Quelles erreurs le compilateur peut-il 'ignorer' tout en continuant à travailler? (problèmes de 'error recovery')

IHM (Interface Homme-Machine)

permissivité du langage

initiatives que peut prendre un compilateur

problèmes de 'trade-off' (équilibre entre des compromis)

Exemple : Plus la compilation est rapide, moins l'optimisation du code est facile.

Problème de l'ambiguïté

Une phrase ambiguë est une phrase à laquelle on peut attribuer plusieurs sens.

Exemple :

2 + 3 * 4 s'interprète comme (2 + 3) * 4 ou 2 + (3 * 4)}

A quelle branche appartient la compilation ?

C'est une matière fondamentale pour l'informatique

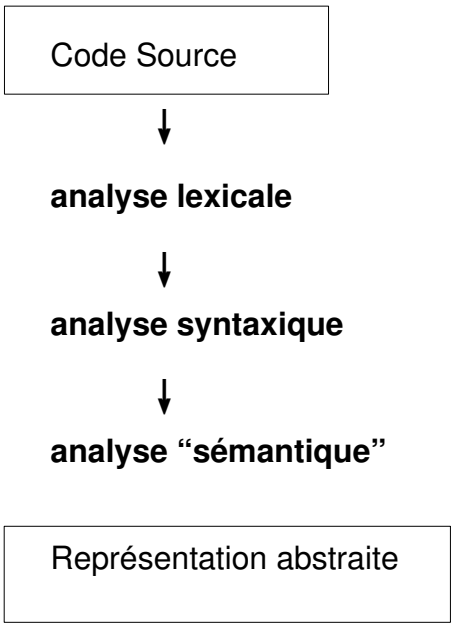
Cela met en jeu :

- la théorie des langages
- la programmation
- l'algorithmique
- les structures de données
- l'architecture des machines

But de ce cours :

- Comprendre ce que fait le compilateur (moteur sous-jacent)
- Savoir écrire des grammaires adaptées
- Apprendre des outils qui permettent d'écrire un compilateur (En TP : logiciels LEX et YACC)

1.3. Un aperçu des phases de la compilation



↓

génération de code intermédiaire

↓

optimisation du code intermédiaire

↓

génération et optimisation du code cible

Différentes phases peuvent s'exécuter en même temps (suivant les langages et les compilateurs).

A. La phase d'analyse

Cette phase traduit le code source en une représentation intermédiaire.

Elle constitue le "**front end**".

L'analyse lexicale

C'est la phase la plus simple.

But : prendre en entrée un flot de caractères et procéder à la segmentation

(séparation en "mots").

Outil TP : LEX.

- On trouve les mots (lexèmes, tokens)
- On vérifie que chaque mot trouvé appartient bien au lexique (dictionnaire)
- On retourne (éventuellement) des messages d'erreur.

Ces 3 étapes se font simultanément, en un seul parcours linéaire du flot d'entrée.

Exemple :

létudiantatroischiens → l étudiant a trois chiens.

les @@2 → ERREUR

les gozbuku → ERREUR

les étudiant → OK

((1+3*2+4) → OK

Certains compilateurs acceptent des mots sans délimiteur :

lavoiture : lav lave? la voiture?

Nécessite un travail de '**look-ahead**' (on regarde ce qui vient après)

L'analyse syntaxique

(en anglais : **parsing**)

C'est une analyse de la correction de la structure.

Outil TP : YACC.

On se donne : un flot de mot et on veut vérifier la consistance de l'assemblage de ces mots en une “phrase”.

On construit un arbre syntaxique

On détecte des erreurs.

Exemple :

* * 43 - 12 $\rightarrow$ ERREUR : * * est interdit.

L'analyse dite "sémantique"

C'est l'analyse du "sens", elle gère par exemple les problèmes de typage.

Certains langages sont capables de convertir automatiquement les types (**mécanisme de coercition**).

On prend en entrée un arbre et on procède à des vérifications.

B. La phase de synthèse (ou génération de code)

Elle constitue le "**back end**".

Elle consiste à prendre en entrée la représentation intermédiaire (de nature arborescente) engendrée par la phase d'analyse, pour la traduire en Assembleur ou en code machine spécifique.

Elle comporte trois étapes :

- génération d'un code intermédiaire (par exemple en "code 3-adresses") proche de la machine mais "portable"

- **génération du code machine spécifique** à partir de ce code intermédiaire

- **optimisation du code** qui se fait en concurrence avec les phases 1 et 2 (factorisation de variables, de code, "minimisation" du nombre de registres nécessaires, etc)

2. Quelques rappels de théorie des langages (voir Annexe)

2.1. Feuille d'exercices 1 : révisions de théorie des langages

Grammaires

1. Soit G la grammaire définie comme suit :

$S \rightarrow bA \mid aB$
$A \rightarrow bAA \mid aS \mid a$
$B \rightarrow aBB \mid bS \mid b$

Examiner si les mots suivants appartiennent à L(G), et si oui, donner une dérivation droite, une dérivation gauche, et l'arbre syntaxique correspondants :
ω_1 = bbaaba ω_2 = babbab ω_3 = bbaaba

2. Montrer que la grammaire :

$S \rightarrow aS \mid Sa \mid a$

est ambiguë et trouver une grammaire équivalente G' non-ambiguë.

3. Soit G la grammaire sur {a,b} définie comme suit :

$S \rightarrow SS \mid XaXaXa \mid \varepsilon$ $X \rightarrow bX \mid \varepsilon$

Montrer que le mot ω = abbaba est dans L(G).

4. On considère la grammaire définie par :

$S \rightarrow aB \mid bA$
$A \rightarrow a \mid aS \mid bAA$
$B \rightarrow b \mid bS \mid aBB$

Examiner si les mots suivants appartiennent à L(G), et si oui, donner une dérivation droite, une dérivation gauche, et l'arbre syntaxique correspondant :
ω_1 = aaabbabbba ω_2 = babbab
Existe-t-il une expression régulière décrivant le langage engendré par G ? Pourquoi ?
Si oui, donner cette expression régulière.

5. Quel est le langage engendré par la grammaire :

$S \rightarrow AA$
$A \rightarrow AAA$
$A \rightarrow bA \mid Ab \mid a$

Déterminer si ce langage est rationnel et si oui, donner un automate d'états finis, une expression régulière et une grammaire régulière correspondants.

6. Améliorer la grammaire :

$S \rightarrow aA \mid bB \mid cdD$

$A \rightarrow aB \mid \varepsilon$

$B \rightarrow aB \mid abB$

$D \rightarrow aD \mid bS \mid \varepsilon$

Expressions régulières

7. Montrer que les deux expressions régulières r et s sont équivalentes :

r = (a+b)*a(a+b)*a(a+b)*

s = b*ab*a(a+b)*

8. Donner une grammaire engendrant le langage des mots sur {a,b} comportant au moins une occurrence de 'a' et au moins une occurrence de 'b', ainsi qu'une expression régulière correspondante.

9. Trouver une grammaire G engendrant le langage décrit par l'expression régulière

r = ab*ab(a+b)*

10. Donner une expression régulière décrivant le même langage que la grammaire :

$S \rightarrow AaB$

$A \rightarrow bA \mid \varepsilon$

$B \rightarrow aB \mid bB \mid \varepsilon$

Automates à pile

11. Donner une grammaire et un automate à pile pour le langage des mots 'bien parenthésés' sur {a,b}.

12. Donner une grammaire et un automate à pile pour le langage des palindromes sur {a,b}.

3. L'analyse lexicale

L'analyse lexicale est un prétraitement très important :

- segmentation des caractères du flot d'entrée (séparation des 'mots' ou 'tokens' les uns des autres)

- repérage et séparations des mots clés du langage

- détection d'erreurs ('syntax error')

En TP, on utilisera l'outil 'LEX' (ou flex)

Transparents :

http://www.keithschwarz.com/cs143/WWW/sum2011/lectures/010_Lexical_Analysis.pdf
Keith Schwarz

4. L'analyse syntaxique

Le procédé qui consiste à trouver une dérivation d'un mot ω d'un langage algébrique s'appelle l'analyse syntaxique.

Il existe 2 grandes méthodes d'analyse :
- La méthode **ascendante** (bottom-top)
- La méthode **descendante** (top-to-bottom)

Méthode ascendante
Principe : On part d'un mot ω et on essaye de procéder à des remplacements successifs qui permettent d'arriver à l'axiome S.

Méthode descendante
Principe : On part de l'axiome S et on essaye de trouver une dérivation qui aboutit à ω.

Exemple 1 : Grammaire G1 :

$S \rightarrow aA \mid b$
$A \rightarrow dA \mid e$
$B \rightarrow f \mid g$

mot à analyser : ω = ade

Analyse descendante :

$S \Rightarrow_{(r1)} aA \Rightarrow_{(r3)} ad\,A \Rightarrow_{(r4)} ade$

Analyse ascendante :

$\underline{ade} \Leftarrow_{(r4)} \underline{adA} \Leftarrow_{(r3)} \underline{aA} \Leftarrow_{(r1)} S$

Exemple 2 : Grammaire G2 :

$S \rightarrow AB$
$A \rightarrow BA \mid a$
$B \rightarrow b$

mot : ω : bbab

Analyse descendante :

$S \Rightarrow_{(r1)} \underline{A}B \Rightarrow_{(r2)} \underline{B}AB \Rightarrow_{(r4)} b\underline{A}B \Rightarrow_{(r2)} b\underline{B}AB \Rightarrow_{(r4)} bb\underline{A}B \Rightarrow_{(r3)} bba\underline{B} \Rightarrow_{(r4)} bbab$

Analyse ascendante :

$\underline{b}bab \Leftarrow_{(r4)} B\underline{b}ab \Leftarrow_{(r4)} BB\underline{a}b \Leftarrow_{(r3)} B\underline{BA}b \Leftarrow_{(r2)} \underline{BA}b \Leftarrow_{(r2)} A\underline{b} \Leftarrow_{(r4)}\underline{AB} \Leftarrow_{(r1)} \Leftarrow S$

Analyse descendante : on obtiendra une **dérivation gauche** de ω.

Analyse ascendante : on obtiendra une **dérivation droite** de ω.

Analyse descendante :
S $\Rightarrow$(r1) $\underline{A}$B $\Rightarrow$(r2) $\underline{B}$AB $\Rightarrow$(r4) b$\underline{A}$B $\Rightarrow$(r2) b$\underline{B}$AB $\Rightarrow$(r4) bb$\underline{A}$B $\Rightarrow$(r3) bba$\underline{B}$ $\Rightarrow$(r4) bbab
Suite de règles : (1,2,4,2,4,3,4)

Analyse ascendante :
S $\Leftarrow$(r1) A$\underline{B}$ $\Leftarrow$(r4) $\underline{A}$b $\Leftarrow$(r2) $\underline{B}$Ab $\Leftarrow$(r2) BB$\underline{A}$b $\Leftarrow$(r3)
B$\underline{B}$ab $\Leftarrow$(r4) Bbab $\Leftarrow$(r4)bbab
Suite de règles : (1,4,2,2,3,4,4)

Remarque : l'analyse ascendante marche avec une classe de langages plus restreinte que l'analyse descendante.

Analyse ascendante : langages LR
Analyse descendante : langages LL

5. Les fonctions FIRST et FOLLOW

Ces fonctions seront nécessaires pour construire les analyseurs étudiés.

5.1. Symboles nullables

Un symbole $A \in N$ est **nullable** si $A \Rightarrow^* \varepsilon$

Calcul des symboles nullables
- Initialisation : pour chaque ε-production $A \rightarrow \varepsilon$, A est nullable.
- Répéter jusqu'à obtention de stabilité :
  Pour chaque règle de production $X \rightarrow Y1Y2 \ldots Yn$,
  (où chaque Yi est un caractère non-terminal différent)
  si tous les Yi sont nullables, alors X est nullable.

5.2. Définition de FIRST

Soit α une chaîne de caractères ($\alpha \in N \cup T$).

FIRST(α) est l'ensemble de tous les terminaux qui peuvent commencer une chaîne qui se dérive de α, avec en plus ε si α est nullable :

$FIRST(\alpha)=\{x \in T \mid \alpha \Rightarrow^* x\beta\} \cup \{\varepsilon \text{ si } \alpha \text{ est nullable}\}$.

Exemple :
G3 :
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid n$

pour α =TE' , '(' est dans FIRST (α).
En effet : $TE' \Rightarrow$(r2) $T + TE' \Rightarrow$(r4) $FT'+TE' \Rightarrow$(r7) **(**$E)T'+TE'$.

5.3. Définition de FOLLOW

Soit A un symbole non-terminal.

FOLLOW(A) est l'ensemble de tous les terminaux qui peuvent apparaître immédiatement à droite de A dans une dérivation $S \Rightarrow^* \alpha A \beta$.

FOLLOW(A)=$\{x \in T \mid S \Rightarrow^* \alpha A \beta$, avec $x \in$ FIRST(β)$\}$.

Remarque : le mot vide n'appartient jamais à FOLLOW.

Exemple sur G3 :
')' est dans FOLLOW(E) :
en effet, $E \Rightarrow$(r1) $TE' \Rightarrow$(r4) $FT'E' \Rightarrow$(r7) (E**)**T'E'.

On ajoutera un symbole de fin de mot, $; $ sera dans FOLLOW(S).

5.4. Calcul du FIRST

On aura besoin de calculer FIRST(α) pour chaque α qui est une partie droite d'une règle de production de la grammaire considérée.

- Pour un caractère terminal x, FIRST(x)={x}.
- Pour les caractères non-terminaux X, on calcule FIRST(X).
- Pour α partie droite d'une règle on calcule FIRST(α).

Calcul de FIRST(X), pour tout $X \in N$
On va construire un graphe d'héritage (orienté) dont les sommets sont les éléments de N; chaque sommet X aura une étiquette qui contiendra les éléments de FIRST(X). Les arcs permettront de faire hériter de symboles terminaux (mais pas de ε).
Au départ, le graphe n'a pas d'arc et les étiquettes sont toutes vides.

- Pour chaque règle de production de la forme $X \rightarrow Y_1 Y_2 \ldots Y_n$, (où chaque Y_i est un caractère différent de $N \cup T$) :
  Soit Y_K le premier Y_i non nullable (si les Y_i sont tous nullables, alors k=n)
  Si Y_K est terminal, AJOUTER Y_K à FIRST(X); AJOUTER les arcs (Y_i ,X) de i=1 à k-1.
  Sinon (Y_K est non-terminal), AJOUTER les arcs (Y_i ,X) de i=1 à k.
- Faire hériter.
- AJOUTER ε est dans FIRST de tous les non-terminaux nullables.

Exemple : reprenons la grammaire G3 :
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid n$

Sommets du graphe d'héritage : E, E', T, T', F
Symboles nullables : E', T'.

$E \rightarrow TE'$: on ajoute l'arc (T,E)

$E' \rightarrow +TE'$: on ajoute '+' dans FIRST(E')

$T \rightarrow FT'$: on ajoute l'arc (F,T)

$T' \rightarrow *FT'$: on ajoute '*' dans FIRST(T')

$F \rightarrow (E)$: on ajoute '(' dans FIRST(F)

$F \rightarrow n$: on ajoute 'n' dans FIRST(F)

Graphe : E' {+}, T'{*}, F{(,n}, arcs (T,E), (F,T)
T hérite de {(,n} de F, et E hérite de {(,n} de T.
On ajoute ensuite ε aux étiquettes de T' et de E'.
A la fin :

FIRST(E)={(,n}, FIRST(E')= { ε ,+}, FIRST(T)={(,n}, FIRST(T')={ ε ,*},
FIRST(F)={(,n}.

<u>Calcul de FIRST(α), pour α une partie droite de règle</u>
$\alpha = Y_1 Y_2 \ldots Y_n$, (où chaque Y_i est un caractère différent de $N \cup T$) :

Si les Y_i sont tous nullables, alors k=n, sinon Y_K est le premier Y_i non nullable.
$FIRST(\alpha) = FIRST(Y_1) \cup \ldots \cup FIRST(Y_K)$.
Si k=n, AJOUTER ε à FIRST (α).

Exemple sur G3:
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid n$

α_1 = TE' : FIRST(TE') = FIRST(T) = {(,n}.
α_2 = +TE' : FIRST(+TE') = {+}
α_3 = ε : FIRST(ε) = {ε}
α_4 = FT' : FIRST(FT') = FIRST(F) = {(,n}.
α_5 = *FT' : FIRST(*FT') = {*}
α_6 = ε : FIRST(ε) = {ε}
α_7 = (E) : FIRST((E))={(}
α_8= n : FIRST(n)={n}

5.5. Calcul du FOLLOW des non-terminaux

De même que pour FIRST, on construit un graphe d'héritage, dont les sommets sont les éléments de N; chaque sommet X aura une étiquette qui contiendra les éléments de FOLLOW(X).

Initialisation : mettre $ dans FOLLOW(S).

On va chercher 3 types de décomposition des parties droites de règles pour un symbole non-terminal B de cette partie droite.

1. $A \rightarrow \alpha B \beta$, ($\alpha \in (N \cup T)^*$, $\beta \neq \varepsilon$, $\beta \in (N \cup T)^+$, mais β peut être nullable) : on met FIRST(β)-ε dans FOLLOW(B).
2. $A \rightarrow \alpha B$ ou $\alpha B \beta$ avec β nullable ($\alpha \in (N \cup T)^*$, $\beta \in (N \cup T)^*$) : ajouter l'arc (A,B) au graphe d'héritage.

Exemple sur la grammaire G3 :

1. règles de type α B β (β ≠ ε)

règle	*B*	β	*FIRST(β)*	*FOLLOW(B) (avant)*	*FOLLOW(B) (après)*
E → TE'	T	E'	{ε,+}	∅	{+}
E' → +TE'	T	E'	idem	idem	idem
T → FT'	F	T'	{ ε ,*}	∅	{*}
T' → *FT'	F	T'	idem	idem	idem
F → (E)	E	)	{)}	{$}	{$,)}

règle	*B*	β	*FIRST(β)*	*FOLLOW(B) (avant)*	*FOLLOW(B) (après)*

2. règles de type α B

règle	*B*	*A*	*arc (A,B)*
E → TE'	E'	E	(E,E')
E' → +TE'	E'	E'	rien
T → FT'	T'	T	(T,T')
T' → *FT'	T'	T'	rien

3. règles de type α B β (β nullable)

règle	*B*	β	*A*	*arc (A,B)*
E → TE'	*T*	*E'*	*E*	*(E,T)*
E' → +TE'	T	E'	E'	(E',T)
T → FT'	F	T'	T	(T,F)
T' → *FT'	F	T'	T'	(T',F)

On obtient le graphe où T est initialisé à {+}, F à {*}, E * {$,)},
avec les arcs : (E,E'), (E,T), (E',T), (T,T'),(T,F), (T',F)

On obtient :
FOLLOW(E) = { $,)}
FOLLOW(E') = {$,) }
FOLLOW(T) = { +,$,)}
FOLLOW(T') = {+,$,) }
FOLLOW(F) = { +, *, $,)}

6. L'analyse descendante (top-down parsing)

6.1. Les langages LL

Rappels

Principe : on se donne une grammaire G et un mot ω, et on construit un arbre de dérivation en partant de l'axiome.

Exemples :

Grammaire G1 :

$S \rightarrow aA \mid b$

$A \rightarrow dA \mid e$

$B \rightarrow f\,B \mid g$

mot à analyser : ω = ade

Analyse descendante :

$S \Rightarrow_{(r1)} aA \Rightarrow_{(r3)} adA \Rightarrow_{(r4)} ade$

Grammaire G2 :

$S \rightarrow AB$

$A \rightarrow BA \mid a$

$B \rightarrow b$

mot : ω : bbab

Analyse descendante :

$S \Rightarrow_{(r1)} \underline{A}B \Rightarrow_{(r2)} \underline{B}AB \Rightarrow_{(r4)} b\underline{A}B \Rightarrow_{(r2)} b\underline{B}AB \Rightarrow_{(r4)} bb\underline{A}B \Rightarrow_{(r3)} bba\underline{B} \Rightarrow_{(r4)} bbab$

On a construit une dérivation **gauche** de ω, représentable par la suite de règles (r1,r2,r4,r2,r4,r3,r4), en utilisant un symbole de "look-ahead".

Cette technique est utilisée pour des grammaires particulières, appelées LL(k) s'il y a k symboles de "look-ahead" à utiliser.

LL(k)="scanning the input from **L**eft to right producing a **L**eftmost derivation, with k symbols of look-ahead".

- Avantages : cette technique est simple, très efficace et facile à implémenter.

- Inconvénient : elle ne fonctionne sans conflits que sur une sous-classe des langages algébriques déterministes.

- On dit qu'un langage L est LL(k) s'il existe une grammaire LL(k) qui engendre L.
- Un langage est LL(k) si il est LL(1).
- On parlera de la classe des langages LL.

Exemple de grammaire LL(1) : la grammaire G3 est une version LL de la grammaire LALR(1) des expressions arithmétiques simples.
G3 :
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid n$

6.2. Caractérisation des grammaires LL(1)

Rappel : on dit qu'un symbole non-terminal A est nullable si $A \Rightarrow^* \varepsilon$, on dit qu'une chaîne de caractères α est nullable si $\alpha \Rightarrow^* \varepsilon$.

Pour définir une grammaire LL(1), on utilise les fonctions FIRST et FOLLOW, qui permettront de choisir la règle de grammaire à appliquer.

Caractérisation
Une grammaire algébrique G est dite LL(1) si $\forall A \in N$, associé aux règles de production
$A \Rightarrow \alpha_1 \mid \alpha_2 \mid .... \mid \alpha_n$
on a, $\forall i,j , i \neq j$,
- $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing$
- et, pour tout α_i nullable : $FIRST(\alpha_j) \cap FOLLOW(A) = \varnothing$

Cette définition peut s'étendre aux grammaires LL(k).
FIRST et FOLLOW serviront à construire la table d'analyse LL.

6.3. L'analyseur LL

On utilise encore une fois une table d'analyse, et la grammaire est LL(1) si et seulement si la table est sans conflit. Une case vide correspond à un 'reject'.

Exemple : voici la table d'analyse de la grammaire G3 :

G3 :
E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | n

non-être	n (nombre)	+	*	(	)	$
E	r1E → TE'			r1E → TE'		
E'		r2E' → +TE'			r3E' → ε	r3E' → ε
T	r4T → FT'			r4T → FT'		
T'		r6T' → ε	r5T' → *FT'		r6T' → ε	r6 T' → ε
F	r8F → n			r7F → (E)		

colonnes : les symboles terminaux et $
lignes : les symboles non-terminaux
cases : les règles de production

On utilise une pile initialisée à Z0, S, et un symbole de "look-ahead" p qui pointe sur le caractère courant du mot à analyser.

Etape d'analyse :
On compare le symbole de haut de pile X avec p.

1er cas : X est un symbole terminal ($X \in T$)
- si X = p, on dépile X et on décale p.
- sinon, REJECT.

2e cas : X est un symbole non-terminal ($X \in N$)
- si [X,p] est une règle X → ε, on dépile X (et on n'empile rien).
- si [X,p] est une règle X → Y1Y2 ... Yn, (où chaque Yi est un caractère différent), on dépile X et on empile Yn....Y1 (sauf ε).
- si [X,p] est une case vide : REJECT.

3e cas : X=Z0
- si p=$, ACCEPT.
- sinon, REJECT.

Exemple : on va analyser le mot ω = 2 + 3 * 4.

Pile	p	case		
Z0,E	2	[E,n]	r1: E → TE'	
Z0, E', T	2	[T,n]	r4 : T → FT'	
Z0, E', T',F	2	[F,n]	r8 : F → n	
Z0, E', T',n	2		n=2	décalage
Z0, E', T'	+	[T',+]	r6 : T' → ε	
Z0, E'	+	[E',+]	r2 : E' → +TE'	
Z0, E',T,+	+		+ = +	décalage
Z0, E',T	3	[T,n]	r4 : T → FT'	
Z0, E',T',F	3	[F,n]	r8 : F → n	
Z0, E',T',n	3		n=3	décalage
Z0, E',T'	*	[T',*]	r5 : T' → *FT'	
Z0, E',T',F,*	*		* = *	décalage
Z0, E',T',F	4	[F,n]	r8 : F → n	
Z0, E',T',n	4		n = 4	décalage
Z0, E',T'	$	[T',$]	r6 : T' → ε	
Z0, E'	$	[E',$]	r3 : E' → ε	
Z0	$	[Z0,$]	ACCEPT	

Suite de règles : (r1, r4, r8, r6, r2, r4, r8, r5, r8, r6, r3).
Dérivation gauche correspondante :

E ⇒(r1)TE' ⇒(r4) FT'E' ⇒(r8)nT'E' ⇒(r6) nE' ⇒(r2) n+TE' ⇒(r4) n+FT'E' ⇒(r8) n+nT'E' ⇒(r5) n+n*FT'E'⇒(r8) n+n*nT'E' ⇒(r6) n+n*nE' ⇒(r3) n+n*n

6.4. Calcul de la table d'analyse LL

On peut vérifier que G3 est bien une grammaire LL(1) :

Règle de production factorisée	***FIRST(αi)∩ FIRST(αj)***	**Aj nullables**	***si oui, FOLLOW(A)∩FIRST (αj)***
E → TE'	rien	rien	
E' → +TE' | ε	FIRST(+TE')={+}, FIRST(ε)={ε} OK	ε	FOLLOW(E')= {$,) } OK
T → FT'	rien	rien	
T' → *FT' | ε	FIRST(*FT')={*}, FIRST(ε)={ε} OK	ε	FOLLOW(T')= {+,$,) } OK
F → (E) | n	FIRST((E))={(}, FIRST(n)={n} OK	rien	

Construction de la table d'analyse LL(1)

Pour chaque règle de production A→α
    Pour chaque a dans FIRST(α)
        Mettre A→α dans la case [A,a]
    Si en plus α est nullable, pour chaque b ∈ FOLLOW(A)
        Mettre A→α dans a case [A,b]

Exemple sur la grammaire G3 :
E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | n

FOLLOW(E) = { $,)}
FOLLOW(E') = {$,) }
FOLLOW(T) = { +,$,)}
FOLLOW(T') = {+,$,) }
FOLLOW(F) = { +, *, $,)}

α1 = TE' : FIRST(TE') = FIRST(T) = {(,n}.
α2 = +TE' : FIRST(+TE') = {+}
α3 = ε : FIRST(ε) = {ε}
α4 = FT' : FIRST(FT') = FIRST(F) = {(,n}.
α5 = *FT' : FIRST(*FT') = {*}
α6 = ε : FIRST(ε) = {ε}
α7 = (E) : FIRST((E))={(}
α7 = n : FIRST(n)={n}

Règle r1 : E → TE' : FIRST(TE')={(,n} : on met r1 dans [E,(] et dans [E,n]

Règle r2 : E' → +TE' : FIRST(+TE')={+} : on met r2 dans [E',+]

Règle r3 : E' → ε : FIRST(ε)={ ε } ; ε est nullable ; FOLLOW(E')={$,) } ; on met r3 dans [E',$] et dans E',)]

Règle r4 : T → FT' : FIRST(FT')={(,n} ; on met r4 dans [T,(] et dans [T,n].

Règle r5 : T' → *FT' ; FIRST(*FT') = {*} ; on met r6 dans [T'*].

Règle r6 : T' → ε : ε est nullable ; FOLLOW(T') = {+,$,) } ; on met r6 dans [T', +], [T',$] et dans [T',)].

Règle r7 : F → (E) ; FIRST((E)) = {(} ; on met r7 dans [F,)].

Règle r8 : F → n ; FIRST(n)={n} ; on met r8 dans [F,n].

On retrouve la table utilisée ci-dessus.

6.5. Procédés de réécriture d'une grammaire non LL(1)

Rappel : on dit que l'on réécrit une grammaire G si on calcule une grammaire G' équivalente à G.

6.5.1. Récursivité

Une grammaire algébrique G est dite **récursive à gauche** (left recursive) si et

seulement si elle contient un non-terminal A tel que
$A \Rightarrow^* A\,\alpha$, $\alpha \in N \cup T$.

Exemple :
G1 :

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid n$

G1 est récursive à gauche.

Propriété :

Une grammaire récursive à gauche n'est pas LL(1).

On peut parfois éliminer la récursivité en réécrivant la grammaire.

On distingue :

La récursivité à gauche immédiate

Il existe une règle de type $A \rightarrow A\,\alpha$.

Elimination de la récursivité gauche immédiate : pour chaque non-terminal A, on remplace
$A \rightarrow A\,\alpha 1 \mid .... \mid A\,\alpha m \mid \beta 1 \mid ... \mid \beta n$

par :

$A \rightarrow \beta 1\ A' \mid .... \mid \beta n\ A'$

$A' \rightarrow \alpha 1\ A' \mid .... \mid \alpha m\ A' \mid \varepsilon$

On obtient une grammaire équivalente.

Exemple :

G1 :

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid n$

Traitement de

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

$A \rightarrow A\,\alpha 1 \mid \beta 1$

on remplace par :

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$.

Traitement de

$T \rightarrow T*F \mid F$

on remplace par :

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

On obtient la grammaire G3 :

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid n$

La récursivité à gauche non immédiate

Une grammaire est récursive à gauche (non immédiatement) si elle contient

un non-terminal A tel que
$A \Rightarrow^{+} A\alpha$, $\alpha \in N \cup T$.

Exemple : G
$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \varepsilon$

On a : $S \Rightarrow Aa \Rightarrow Sda$, donc G est récursive à gauche mais non immédiatement.

Elimination de la récursivité gauche :

1. Attribuer un ordre arbitraire aux non-terminaux : (X1, X2, ...Xn)
2. Pour i=1 à n faire
   Pour j=1 à i-1 faire
   Remplacer chaque production de la forme $Xi \rightarrow Xj\ \alpha$, avec
   $Xj \rightarrow \beta 1 \mid .... \mid \beta r$
   par :
   $Xi \rightarrow \beta 1\ \alpha \mid .... \mid \beta r\ \alpha$
   Eliminer les récursivités gauches immédiates pour Xi.

On obtient une grammaire équivalente.

Exemple :

S → Aa | b
A → Ac | Sd | ε

i=1 : Xi = S : pas de récursivité

i=2 et j=1 : Xi = A et Xj = S : remplacer A → Sd, où S → Aa | b

par :

A → Aad | bd

Eliminons ensuite la récursivité gauche immédiate de :

A → Ac| Aad | bd | ε

on remplace par :

A → bdA' | εA'

A' → cA' | adA' | ε

On obtient finalement :

S → Aa | b

A → bdA' | A'

A' → cA' | adA' | ε

Remarque : ce procédé de réécriture ne supprime pas toujours la récursivité, par exemple pour :

S → Sa | Tsc | d
T → TbT | ε

6.5.2. Factorisation

Factorisation à gauche

On dit qu'un grammaire algébrique G est non factorisée à gauche si elle contient un non-terminal A dont l'ensemble des parties droites de productions est de la forme
A → α β1 | α β2....|γ1|....|γp

Propriété : Une grammaire non factorisée à gauche n'est pas LL(1).

Procédé de factorisation gauche

Pour chaque non-terminal A, trouver un plus long préfixe α commun à plusieurs parties droites de production dont A est partie gauche.

$A \rightarrow \alpha\ \beta1 \mid \alpha\ \beta2 ....| \alpha\ \beta n|\gamma 1|....|\gamma p$

Remplacer par :

$A \rightarrow \alpha\ A' \mid \gamma 1|....|\gamma p$

$A' \rightarrow \beta 1 \mid \beta 2....| \beta n$

Recommencer jusqu'à ce qu'il n'y ait plus de préfixe propre commun.

On obtient une grammaire équivalente.

Exemple :

Voici une grammaire G4 (encore une version de la grammaire G1 des expressions arithmétiques simple)

G4 :

$E \rightarrow T + E \mid T$

$T \rightarrow F^* T \mid F$

$F \rightarrow (E) \mid n$

Réécriture de la règle :

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

on obtient :

$E \rightarrow TE'$

$E' \rightarrow +E \mid \varepsilon$

Réécriture de la règle :

$T \rightarrow F^* T \mid F$

on obtient :

$T \rightarrow FT'$

$T' \rightarrow *T \mid \varepsilon$

Soit globalement, la grammaire équivalente :

$E \rightarrow TE'$

$E' \rightarrow +E \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *T \mid \varepsilon$

$F \rightarrow (E) \mid n$

6.6. Feuille d 'exercices 2 : les langages LL

1. **Calculer la table d'analyse LL(1) pour :**
   S → iBae

   B → TB | ε

   T → [eD] | di
   D → ed | ε

2. **Calculer la table d'analyse LL(1) pour (après avoir factorisé) :**
   S → aB | bA
   A → a | aS | bAA
   B → b | bS | aBB

3. **Considérons la grammaire G2 :**

   E → E+E | E*E | (E) | n

1. Montrer que G2 est récursive à gauche et non factorisée à gauche.
2. Réécrire G2 en commençant par la factoriser, puis en traitant la récursivité, obtenant la grammaire G'2.
3. Réécrire G2 en commençant par traiter la récursivité, puis en factorisant, obtenant la grammaire G”2.
4. Construire les deux tables d'analyse. G'2 et G”2 sont-elles LL(1)?

Procéder à l'analyse du mot ω = 4+5 pour les deux grammaires.

7. L'analyse ascendante et les langages LR

7.1. Principe du shift-reduce parsing

Principe : On se donne un mot ω, on le lit caractère par caractère et on tente de reconstituer une dérivation.

On va utiliser une pile qui va aider à repérer des parties droites de règles. On va donc empiler jusqu'à ce qu'on trouve une partie droite, puis on va dépiler cette partie droite (à l'envers) et rempiler la partie gauche à sa place.

On utilise 2 opérations élémentaires :

- L'opération de **décalage** (**shift**) qui consiste empiler le caractère courant, puis à lire le caractère suivant du mot à analyser.

- L'opération de **réduction** (**reduce**), qui consiste à remplacer la partie droite d'une règle par sa partie gauche (on dépile la partie droite de règle correspondante et on rempile la partie gauche).

Exemple : Grammaire G3 :

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

mot ω = abbcde

a<u>a</u>bbcde ⇐(shift a) a<u>ab</u>bcde ⇐(shift b) ab<u>bc</u>de ⇐(red 3 : A → b) aAbcde ⇐(shift b) aAb<u>cd</u>e
⇐(shift c) aAbc<u>de</u>⇐(red 2 : A → Abc) aA<u>de</u> ⇐(shift d) aAd<u>e</u>
⇐(red 4 : B → d)aAB<u>e</u> ⇐(shift e) aAB<u>e</u> ⇐(red 1 : S → aABe) S

On a construit la dérivation droite :
S ⇒(r1)aABE ⇒(r4)aAde ⇒(r2) aAbcde ⇒(r3)abbcde

mot	**dériv.**	**pile_avt**	**action**	**pile après**
a<u>a</u>bbcde$	abbcde	[	shift (a)	[a
a<u>ab</u>bcde$	abbcde	[a	shift (b)	[a,b
ab<u>bc</u>de$	abbcde	[a,b	red 3 A → b	[a,A
ab<u>bc</u>de$	aAbcde	[a,A	shift (b)	[a,A,b

abbcde$ aAbcde [a,A,b **???? CONFLIT!**
ON NE SAIT PAS SI ON shift (pour aller vers Abc) ou si red 3 (A → b)
ON CHOISIT SHIFT

		shift (c)	[a,A,b,c	
abbcde$	aAbcde	[a,A,b,c	red 2 : A → Abc	[a,A
abbcde$	aAde	[a,A	shift (d)	[a,A,d
abbcde$	aAde	[a,A,d	red 4 : B → d	[a,A,B
abbcde$	aAde	[a,A,B	shift (e)	[a,A,B,e
abbcde$	aAde	[a,A,B,e	red 1 : S → aABe	[S

On a S et $: on a gagné !

On a défini par l'analyse une suite de règles qui définit la dérivation droite obtenue.
Cette méthode d'analyse s'appelle le 'shift-reduce parsing'.

Lorsque l'on ne sait pas s'il faut choisir un shift ou un reduce, on rencontre un **conflit shift-reduce** ; lorsqu'on a les choix entre l'applications de plusieurs règles, on rencontre un **conflit reduce-reduce.**

7.2. Les grammaires LR et LALR

Le shift-reduce parsing peut se faire sans conflit pour les grammaires LR(k).

LR(k)=Left-to-Right scan of the input producing a Rightmost derivation using k symbols of look-ahead.

Définition :
On appelle grammaire LR(k), une grammaire algébrique G qui permet, en connaissant les k caractères suivants du mot à analyser, de décider (de façon déterministe) quelle opération appliquer pour arriver à la dérivation d'un mot de L(G).

Définition :
On appelle langage LR(k) un langage algébrique tel qu'il existe une grammaire LR(k) qui l'engendre.

Théorème :
Tout langage LR(k) est aussi LR(1).

On appelle LR un langage qui est LR(1).

Théorème :
Un langage est LR si et seulement si il est algébrique et déterministe.

Pour décider si un mot ω appartient au langage défini par une grammaire LR, on utilise un automate à pile déterministe particulier (analyseur LR) présenté sous forme d'une table d'analyse.

- Une table LR(k) est de dimension k+1.
- Dans la pratique, on n'utilise que les tables LR(1), qui sont des tables à double entrée.

La classe des langages LL est strictement incluse dans la classe des langages LR :

langages rationnels ⊊ langages LL ⊊ langages LR = langages algébriques déterministes ⊊ langages algébriques

7.3. L'analyseur LR(1)

On se donne :

- une table d'analyse
- une pile, qui contient une alternance d'états numérotés et de caractères de N U T
- un mot à analyser ω
- un pointeur p sur la lettre courante de ω

Initialisation :
On empile 0 et p pointe sur le premier caractère de ω.

Etape d'analyse :
On lit le haut de la pile (1 ou 2 caractères), et avec p, on en déduit une case de la table qui va contenir des instructions.

1er cas :
Le symbole de haut de pile est un non-terminal F, on lit les 2 caractères de haut de pile [i,F] qui donne un état j, que l'on empile.

2e cas :
Le symbole de haut de pile est un état i, on lit la case [i,p] de l'analyseur.

cas 2.1 : T[i,p]=rj (une règle de la forme A → α), on dépile jusqu'à obtenir (à l'envers) tous les caractères consécutifs de α, puis on empile A.

cas 2.2 : T[i,p]=sj, on va effectuer un shift : on décale p; on empile l'ancien p;
on empile l'état j.

cas 2.3 : T[i,ps] est vide : REJECT : le mot n'appartient pas au langage.

cas 2.4 : T[i,ps] = ACCEPT, le mot est accepté, et on obtient une dérivation droite de ce mot en reprenant (à l'inverse) la suite des règles utilisées.

Exemple : Grammaire G1 des expressions arithmétiques :
r1 : E → E + T
r2 : E → T
r3 : T → T * F
r4 : T → F
r5 : F → (E)
r6 : F → 0|1|2|3|4|5|6|7|8|9
ou :
r6 : F → nb

Table d'analyse :

Etat	nb	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				ACC			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

mot à analyser : 3+4*2$

Pile	p	case	action
0	3	0,nb	s5
0,3,**5**	+	5,+	**r6** : F → nb
0,F	+	0,F	3
0,F,**3**	+	3,+	**r4** : T → F
0,T	+	O,T	2
0,T,**2**	+	2,+	**r2** : E → T
0,E	+	0,E	1
0,E,**1**	+	1,+	s6
0,E,**1**,+,**6**	4	6,nb	s5
0,E,**1,**+,**6**,4,**5**	*	5,*	**r6** : F → nb
0,E,**1,**+,**6**,F	*	6,F	3
0,E,**1,**+,**6**,F,**3**	*	3,*	**r4** : T → F
0,E,**1,**+,**6**,T	*	6,T	9
0,E,**1,**+,**6**,T,**9**	*	9,*	s7
0,E,**1,**+,**6**,T,**9**,*,**7**	2	7,nb	s5
0,E,**1,**+,**6**,T,**9**,*,**7**,2,**5**	$	5,$	**r6** : F → nb
0,E,**1,**+,**6**,T,**9**,*,**7**,F	$	7,F	10
0,E,**1,**+,**6**,T,**9**,*,**7**,F,**10**	$	10,$	**r3 :T → T * F**
0,E,**1,**+,**6**,T	$	6,T	9
0,E,**1,**+,**6**,T,**9**	$	9,$	**r1 : E → E + T**
0,E	$	0,E	**1**
0,E,**1**	$	1,$	**ACCEPT**

Suite de règles utilisées :
r6,r4,r2,r6,r4,r6,r3,r1

Dérivation droite correspondante :
(1,3,6,4,6,2,4,6)

*7.4. Les gra*mmaires LALR

On a vu le fonctionnement de l'analyseur LR(1).
Dans la pratique, on utilise des tables LALR(1), qui sont obtenues en contractant certains états de la table LR(1) correspondante.

Avantages :
- La table obtenue est nettement plus petite : on gagne du temps.
- Ces grammaires 'marchent' de façon satisfaisante.

Inconvénients :
- Les langages LALR sont une classe restreinte de langages LR.
- On peut introduire des conflits alors que la grammaire de départ était bien LR(1).

YACC construit des tables LALR(1).

Gestion des conflits par YACC

conflit shift/reduce : YACC choisit le shift

conflit reduce/reduce : YACC choisit de réduire par la règle de plus petit numéro

Remarque : la façon de gérer les conflits correspond à des interprétations différentes du flot d'entrée.

7.5. Feuille d'exercices 3 : l'analyseur LALR

Voici des grammaires avec pour chacune un mot à analyser.
Vous trouverez les exécutions YACC dans l'annexe 3.
Pour chaque grammaire, construire la table d'analyse. Pour chaque mot à analyser, signaler les éventuels conflits et la façon dont ils ont été résolus, donner la suite de règles obtenue, ainsi que la dérivation droite et l'arbre syntaxique correspondant.

1. Grammaire G1 des expressions arithmétiques : mot (2+3)

2. Grammaire régulière engendrant le langage $(aa + bb)^+$: mot bbaabb

3. Mots bien parenthésés sur {a,b} : mot aababb

4. Langage 'EQUAL' (autant de 'a' que de 'b') : mot abba

5. Grammaire de $a^n b^n$: mot aaabbb

6. Trailing Count : mot : abaaaa

7. Even-Even (un nombre pair de 'a' et un nombre pair de 'b') version régulière : mot abbabb

8. Even-Even (un nombre pair de 'a' et un nombre pair de 'b') version non régulière : mot abbabb

7.6. Feuille d'exercices 4 : étude des conflits dans les tables LALR(1)

On va comparer deux grammaires des expression algébriques simples suivantes (n est un nombre). La sortie YACC pour G2 est donnée à la fin.

G1 :

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid n$

G2 :

$E \rightarrow E+E \mid E*E \mid (E) \mid n$

Question 1

- Pour les mots ω1= 1+2+3 et ω2 = 4+5*6, donner tous les arbres syntaxiques possibles. Que remarque-t-on?

- Pour chaque arbre syntaxique, donner la dérivation droite ainsi que l'interprétation correspondante. (Rappel : l'interprétation est le parenthésage de l'expression).

Question 2

Justifiez vos réponses aux questions suivantes :

- G1, G2 sont-elles ambiguës ?

- G1 et G2 sont-elles équivalentes?

- G1 et G2 sont-elles LALR(1)? LR(1)?

- Quels sont les avantages et les inconvénients de G1 et de G2 ?

Question 3

Interprétations des formules incomplètement parenthésées :

Rappels :

associativité droite de + : 1+2+3 s'interprète comme (1+(2+3))
associativité gauche de + : 1+2+3 s'interprète comme ((1+2)+3)
précédence de + sur * : 4+5*6 s'interprète comme ((4+5)*6)
précédence de * sur + : 4+5*6 s'interprète comme (4+(5*6))

Quelle est la précédence entre + et * définie par G1 ? G1 utilise-t-elle l'associativité gauche ou droite? Comment pourrait-on modifier ces paramètres?

Question 4

On va maintenant étudier la précédence et l'associativité pour G2.

En se servant de la question 1, donner la suite de règles attendue dans une analyse syntaxique du mot ω1= 1+2+3 pour que + soit associative à gauche, à droite? Et de même, donner la suite de règles attendue dans une analyse syntaxique du mot ω2 = 4+5*6, pour que * ait précédence sur +, ou que + ait précédence sur *?

Question 5

Résolutions de conflits dans G2

Comment doit-on résoudre les conflits dans G2 (états 9 et 10) pour que :

- * ait précédence sur +
- + ait précédence sur *
- + soit associatif à gauche
- + soit associatif à droite.

Décrire les 8 cas rencontrés pour les 4 conflits décalage/réduction , et expliquer à quel choix sur la précédence ou l'associativité ils correspondent.

Question 6

Gestion des erreurs

Dans quelles cases vides de la table d'analyse de G2 pourrait-on mettre les messages d'erreurs suivants :

- err 1 : 'opérande manquant'

- err 2 : 'parenthèse ouvrante manquante'

- err 3 : 'opérateur manquant'

- err 4 : 'parenthèse fermante manquante'

Expliquez pourquoi.

Sortie YACC pour G2

```
 0  $accept : E $end
   1  E : E '+' E
   2    | E '*' E
   3    | '(' E ')'
   4    | 'n'
state 0
        $accept : . E $end  (0)

        '('  shift 1
        'n'  shift 2
        .  error
        E  goto 3
state 1
        E : '(' . E ')'  (3)

        '('  shift 1
        'n'  shift 2
        .  error
        E  goto 4
state 2
        E : 'n' .  (4)

        .  reduce 4
state 3
        $accept : E . $end  (0)
        E : E . '+' E  (1)
        E : E . '*' E  (2)

        $end  accept
```

```
        '+'  shift 5
        '*'  shift 6
        .  error
state 4
        E : E . '+' E  (1)
        E : E . '*' E  (2)
        E : '(' E . ')'  (3)

        '+'  shift 5
        '*'  shift 6
        ')'  shift 7
        .  error
state 5
        E : E '+' . E  (1)

        '('  shift 1
        'n'  shift 2
        .  error
        E  goto 8
state 6
        E : E '*' . E  (2)

        '('  shift 1
        'n'  shift 2
        .  error
        E  goto 9
state 7
        E : '(' E ')' .  (3)

        .  reduce 3

8: shift/reduce conflict (shift 5, reduce 1) on '+'
```

```
8: shift/reduce conflict (shift 6, reduce 1) on '*'
state 8
        E : E . '+' E  (1)
        E : E '+' E .  (1)
        E : E . '*' E  (2)

        '+'  shift 5
        '*'  shift 6
        $end  reduce 1
        ')'  reduce 1

9: shift/reduce conflict (shift 5, reduce 2) on '+'
9: shift/reduce conflict (shift 6, reduce 2) on '*'
state 9
        E : E . '+' E  (1)
        E : E . '*' E  (2)
        E : E '*' E .  (2)

        '+'  shift 5
        '*'  shift 6
        $end  reduce 2
        ')'  reduce 2

State 8 contains 2 shift/reduce conflicts.
State 9 contains 2 shift/reduce conflicts.

7 terminals, 2 nonterminals
5 grammar rules, 10 states
```

8. Construction des tables LALR(1)

8.1. Notations et définitions

Dans toute la suite, on considérera une grammaire G=(N,T,P,S).

Une règle de P sera désignée par :

$A \rightarrow \alpha$

ou bien par :

$A \rightarrow X_1X_2...X_n$, où chaque X_i est un caractère de N∪T .

♣ Items

Etant donnée un règle $A \rightarrow X_1X_2...X_n$, un **item** est cette règle avec un point inséré dans la partie droite. Par exemple, $A \rightarrow X_1 \bullet X_2...X_n$ est un item.
Ce point peut aussi être positionné au début ou à la fin de la chaîne.
$A \rightarrow \bullet X_1X_2...X_n$ et $A \rightarrow X_1X_2...X_n \bullet$ sont des items.
Le point symbolise la pile : ce qui est à gauche du point est empilé.
Le but est d'empiler une partie droite de règle, pour pouvoir procéder à une réduction.

♦ Grammaire augmentée

On ajoute une règle fictive $S' \rightarrow S$ à G.

On va construire un automate (un graphe de transitions) dont les sommets sont des états et où les arcs d'un sommet vers un autre sont étiquetés par des transitions.

Les états contiennent des items.

Principe général :

- On définit un procédé de fermeture d'un état
- On initialise avec un état I_0.
- On ferme I_0.
- On définit les transitions à partir de I_0, créant de nouveaux états.
- On ferme chaque nouvel état et on définit les transitions dont il est l'origine.
- et ainsi de suite...

8.2. Cas particulier : les tables LR(0)

Etat initial : I_0= S'→ •S

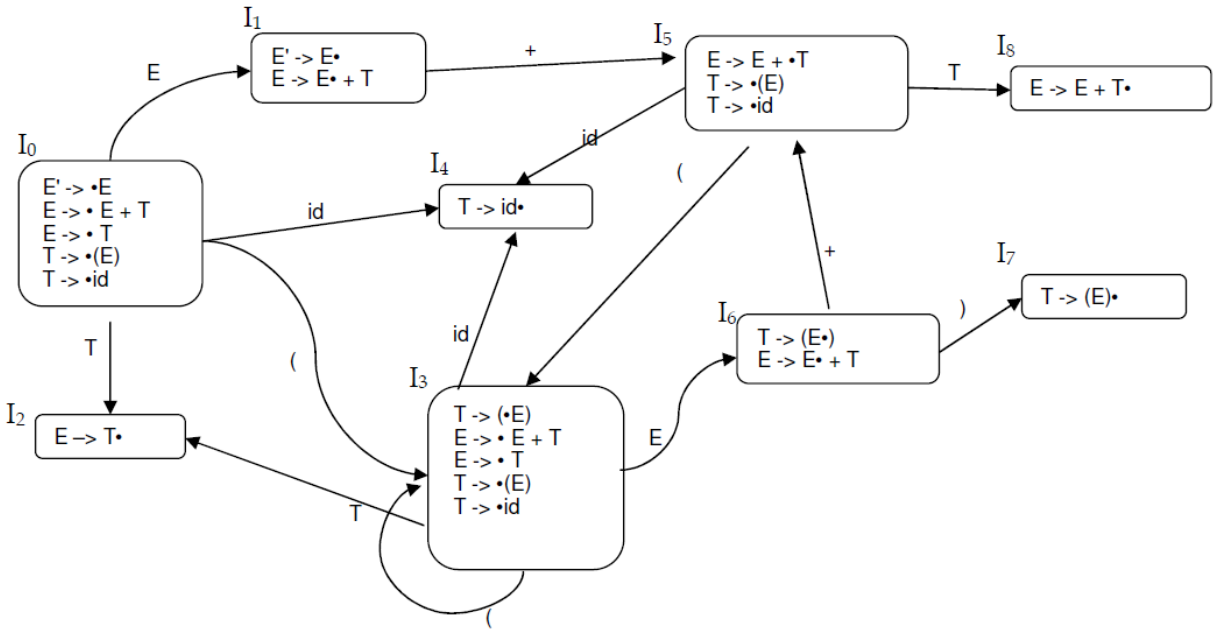
♥ Procédé de fermeture d'un état

Pour chaque règle de type $A \rightarrow X_1 X_2 ... X_i \bullet X_{i+1} ... X_n$
(le point n'est pas à la fin) telle que X_{i+1} est dans N,
pour chaque règle de type $X_{i+1} \rightarrow \alpha$, on ajoute l'item $X_{i+1} \rightarrow \bullet\alpha$.

On recommence sur les nouveaux items jusqu'à stabilité.

♠ Transition de l'état I_i à l'état I_j
Principe : on crée un nouvel item en décalant le point d'un cran vers la droite.
Par exemple, $A \rightarrow X_1 \bullet X_2 ... X_n$ devient $A \rightarrow X_1 X_2 \bullet X_3 .. X_n$
La transition associée est étiquetée par X_2

Exemple (extrait du cours de Keith Schwarz) :
G :
E' → E
E → T
E → E+T
T → int
T → (E)



Construction de la table à partir de l'automate :

- Si S' –> S• est dans l'état I_i : Action[i,$] = ACCEPT
- Si $A \rightarrow \alpha$ • (A ≠ S) est dans l'état I_i : Action[i,a] = reduce avec $A \rightarrow \alpha$ pour tout a.
- Si $A \rightarrow X_1 X_2 ... X_i \bullet X_{i+1} ... X_n$ est dans l'état I_i et qu'il y a une transition vers l'état I_j par le terminal X_{i+1} : Action[i,X_{i+1}] = shift j
- Si $A \rightarrow X_1 X_2 ... X_i \bullet X_{i+1} ... X_n$ est dans l'état I_i et qu'il y a une transition vers l'état I_j par le non-terminal X_{i+1} : Goto (i,X_i) = j

Sur l'exemple :

	+	(	)	int	$	E	T
I_0		s3		s4		1	2
I_1	s5				ACCEPT		
I_2	r1	r1	r1	r1	r1		
I_3		s3		s4		6	2
I_4	r3	r3	r3	r3	r3		

I_5		s3		s4			8
I_6	s5		s7				
I_7	r4	r4	r4	r4	r4		
I_8	r2	r2	r2	r2	r2		

Le langage est LR(0) si sa table LR(0) est sans conflit.

Malheureusement, la classe des langages LR(0) est très limitée.

(par exemple, une partie droite ne doit pas être préfixe d'une autre partie droite ;

deux parties droites ne doivent pas être identiques, etc.)

8.3. Construction des tables LR(1)

On va insérer un caractère de 'look-ahead' pour passer à la classe LR(1).

Pour cela, on va ajouter à chaque item une liste de terminaux qui indiqueront quelles sont les transitions autorisées lors d'une opération de réduction (reduce).

Notation : $A \rightarrow X_1 X_2 ... X_i \bullet X_{i+1} ... X_n , [a_1, a_2, ...]$

$X_1 X_2 ... X_i \bullet X_{i+1} ... X_n$ s'appelle le **coeur** de l'item ; on appellera $[a_1, a_2, ...]$ le **look-ahead** de l'item.

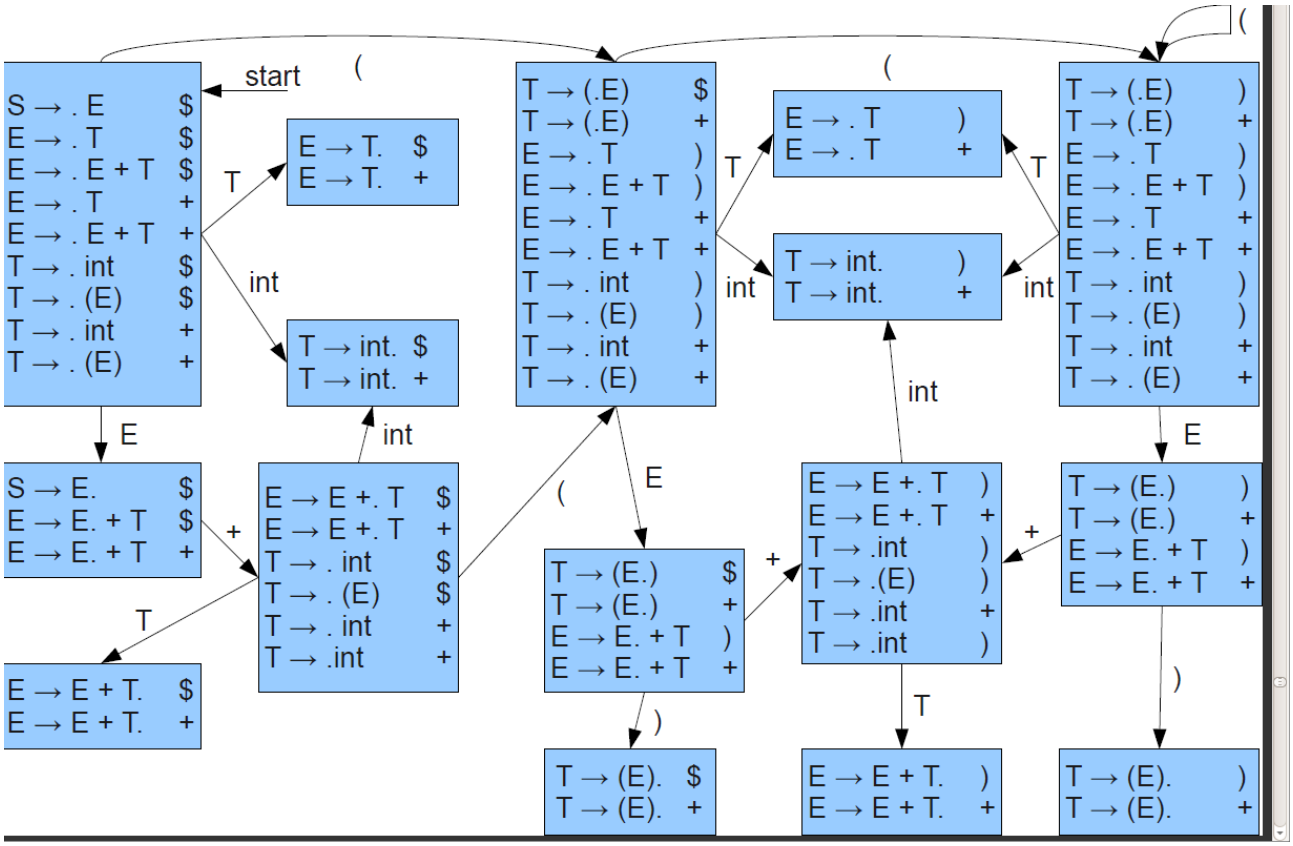
Abus de langage : on dira que $A \rightarrow X_1 X_2 ... X_i \bullet X_{i+1} ... X_n , a_i$ est un item.

Etat initial : $I_0 = S' \rightarrow \bullet S , [\$]$

♥ Procédé de fermeture d'un état

Pour chaque item de type $A \rightarrow \alpha \bullet B \beta , a$ (avec B dans N)
pour chaque règle de type $B \rightarrow \gamma$,
pour chaque terminal b dans FIRST (βa) ,
on ajoute l'item $B \rightarrow \bullet \gamma , b$

Exemple sur G (extrait du cours de Keith Schwarz) :



Calcul de la table LR(1) : Comme pour les tables LR(0) sauf que

- Si $A \rightarrow \alpha \bullet, x$ $(A \neq S)$ est dans l'état I_i : Action[i,x] = reduce avec $A \rightarrow \alpha$

Sur l'exemple :

	int	(	)	+	$	T	E
1	s5					s4	s2
2				s6	ACCEPT		
3				r3	r3		
4				r2	r2		
5				r5	r5		
6	s5	s7				s3	
7	s10	s14				s10	s8
8			s9	s12			
9				r5	r5		
10			r2	r2			
11			r4	r4			
12	s11					s13	
13			r3	r3			
14	s11		s14			s10	s15
15			s16	s12			
16			r5	r5			

S → E (1)
E → T (2)
E → E + T (3)
T → int (4)
T → (E) (5)

8.4. Tables LAR(1)

Inconvénient : les tables LR(1) peuvent être énormes.

Souvent, l'automate comporte des états presque identiques.

Exemple :

T → (·E)	$
E → ·E + T	)
E → ·T	)
T → ·int	)
T → ·(E)	)

T → (·E)	)
E → ·E + T	)
E → ·T	)
T → ·int	)
T → ·(E)	)

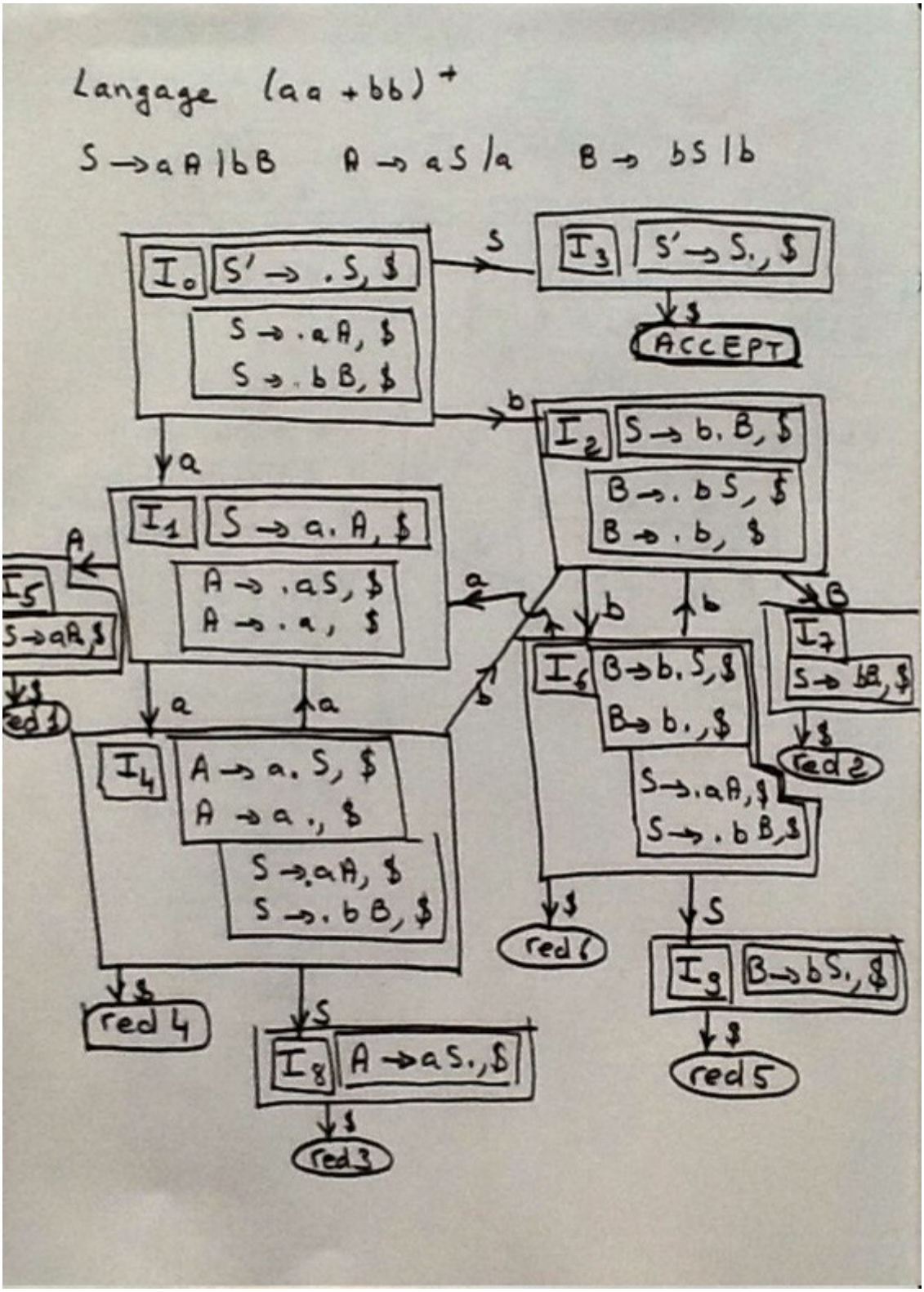
Seule différence : le look-ahead.

On va fusionner les états qui ont le même cœur, et on fera l'union des look-aheads correspondants.

Cette fusion peut engendrer des conflits supplémentaires de type reduce/reduce.

Exemple 1 : langage (aa + bb) $^{+}$
G1 : S $\rightarrow$ aA | bB A $\rightarrow$ aS | a B $\rightarrow$ bS | b

graphe de transitions LALR(1) de G1 :

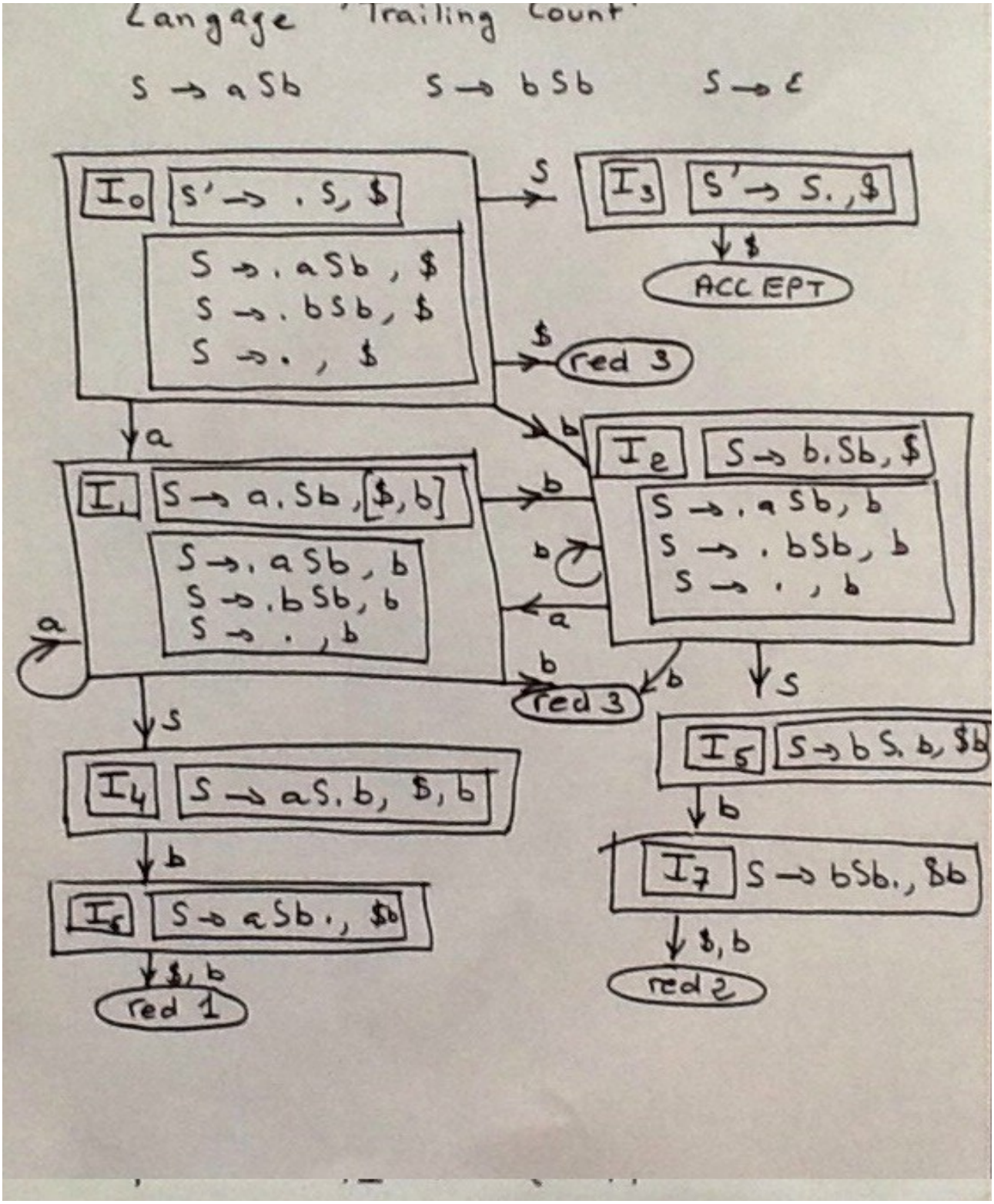


```
Sortie YACC pour G1 :
   0  $accept : S $end
   1  S : 'a' A
   2    | 'b' B
   3  A : 'a' S
   4    | 'a'
   5  B : 'b' S
   6    | 'b'
state 0
        $accept : . S $end  (0)
        'a'  shift 1
        'b'  shift 2
        .  error
        S  goto 3
state 1
        S : 'a' . A  (1)
        'a'  shift 4
        .  error
        A  goto 5
state 2
        S : 'b' . B  (2)
        'b'  shift 6
        .  error
        B  goto 7
state 3
        $accept : S . $end  (0)
        $end  accept
state 4
        A : 'a' . S  (3)
        A : 'a' .  (4)
        'a'  shift 1
        'b'  shift 2
        $end  reduce 4
        S  goto 8
state 5
        S : 'a' A .  (1)
        .  reduce 1
state 6
        B : 'b' . S  (5)
        B : 'b' .  (6)
```

```
        'a'  shift 1
        'b'  shift 2
        $end  reduce 6
        S  goto 9
state 7
        S : 'b' B .  (2)
        .  reduce 2
state 8
        A : 'a' S .  (3)
        .  reduce 3
state 9
        B : 'b' S .  (5)
        .  reduce 5
4 terminals, 4 nonterminals
7 grammar rules, 10 states
```

Exemple 2 : langage 'trailing count'
G2 : S → aSb | bSb | ε

graphe de transitions LALR(1) de G2 :



```
Sortie YACC pour G2 :
   0  $accept : S $end
   1  S : 'a' S 'b'
   2    | 'b' S 'b'
   3    |
state 0
        $accept : . S $end  (0)
        S : .  (3)
        'a'  shift 1
        'b'  shift 2
        $end  reduce 3
        S  goto 3
1: shift/reduce conflict (shift 2, reduce 3) on 'b'
state 1
        S : 'a' . S 'b'  (1)
        S : .  (3)
        'a'  shift 1
        'b'  shift 2
        S  goto 4
2: shift/reduce conflict (shift 2, reduce 3) on 'b'
state 2
        S : 'b' . S 'b'  (2)
        S : .  (3)
        'a'  shift 1
        'b'  shift 2
        S  goto 5
state 3
        $accept : S . $end  (0)
        $end  accept
state 4
        S : 'a' S . 'b'  (1)
        'b'  shift 6
        .  error
state 5
        S : 'b' S . 'b'  (2)
        'b'  shift 7
        .  error
state 6
        S : 'a' S 'b' .  (1)
        .  reduce 1
```

```
state 7
        S : 'b' S 'b' .  (2)
        .  reduce 2
State 1 contains 1 shift/reduce conflict.
State 2 contains 1 shift/reduce conflict.

4 terminals, 2 nonterminals
4 grammar rules, 8 states
```

8.5. Feuille d'exercices 5 : construction de tables LALR(1)

Construire les tables LALR(1) pour :

1. Le langage $a^n b^n$: $S \rightarrow aSb \mid ab$
   Puis vérifier à partir de l'éxécution donnée en annexe.

2. Le langage $S \rightarrow iSeS \mid iS \mid a$

3. Le langage des mots bien parenthésés $S \rightarrow aSb \mid SS \mid \varepsilon$
   Puis vérifier à partir de l'éxécution donnée en annexe.