Parseurs

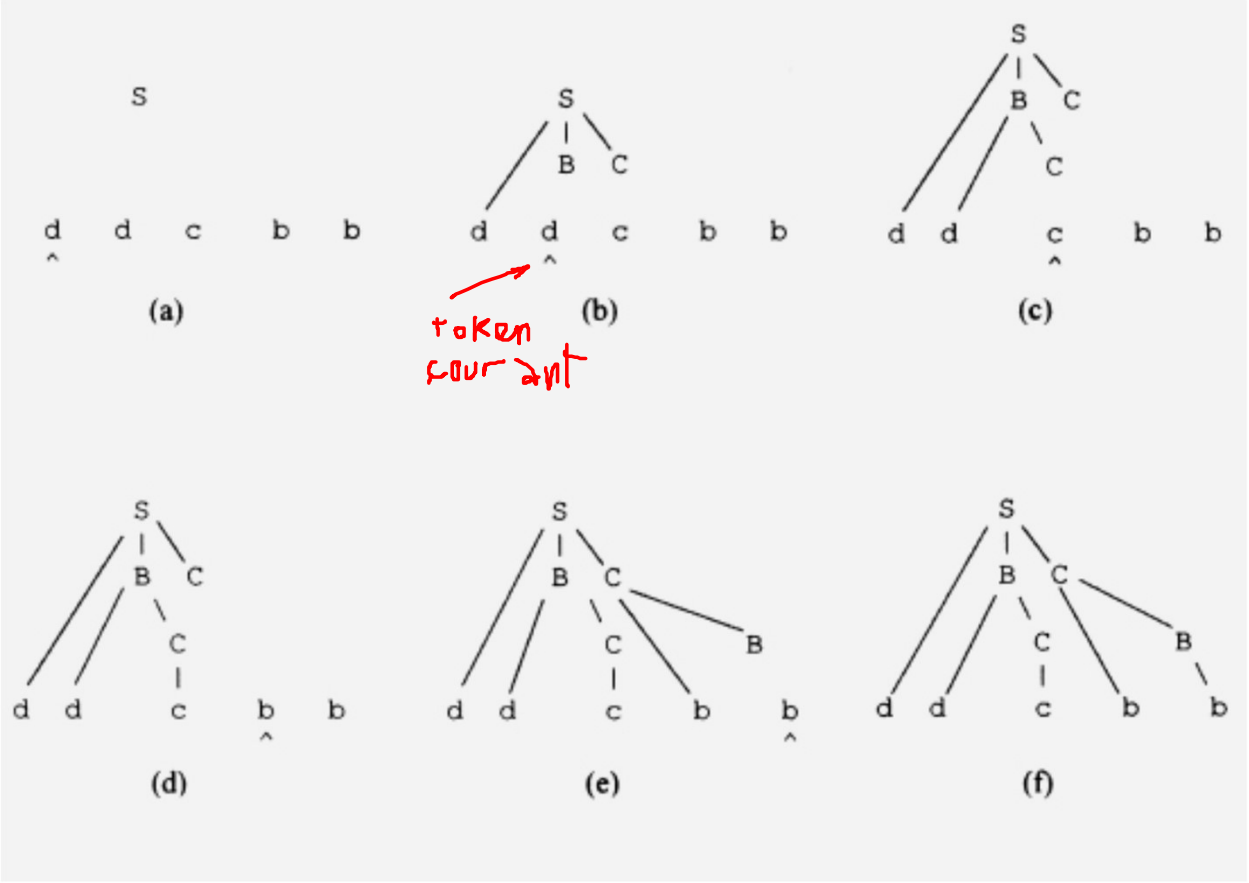
Analyse syntaxique descendante (Top down)

Principe:

en même temps que l'analyseur syntaxique lit la chaîne en entrée token après token, il essaie de prédire quelle est la dérivation la plus à gauche issue de l'axiome à utiliser. L'analyseur peut utiliser un mécanisme essai-erreur avec retour en arrière (lookahead) en cas d'erreur.

1. Construction de l'arbre d'analyse par analyse descendante
   - Grammaire algébrique qui ne possède aucune ε-production et dont les parties droites des productions commencent par un terminal.
   - Algorithme
     1. Toujours développer le non terminal le plus à gauche non encore développé Si le non terminal à développer se trouve dans la partie gauche de plusieurs productions, alors utiliser le token courant (le plus à gauche dans la chaîne en entrée) qui n'est pas encore généré, pour déterminer la production à utiliser.
     2. Si le token courant est généré, alors il faut faire avancer le token courant vers le token suivant
   - Exemple:
     - Soit G1 et la chaîne "ddcbb" (Chaîne en entrée, dbb, dbcc)
       1. S →dBC
       2. B → dC
       3. B → *b*
       4. *C → bB*
       5. C → c

Arbre d’analyse de la chaîne "ddcbb"



- G2
  1. S -> bS
  2. S -> b

  LL(2) (L: la chaîne est lue de gauche à droite, L: une dérivation gauche est utilisée, 2: il faut au plus lire deux tokens pour déterminer les productions à utiliser).
- G3
  1. S→dBC
  2. B →dC
  3. B →d
  4. C →bB
  5. C →c

  LL(2).
- G4
  1. S→bS
  2. S→bb

- La grammaire
  - S → Ab
  - **2.** S → Ac
  - **3.** A → bA
  - **4.** A → ε

  N'est pas LL(k).
- Analyse prédictive (Automate à pile)
  - $: fin de la pile
  - #: fin de la chaîne
  - Etat initial de la pile: S$
  - Avancer : déplacer le pointeur Token_Courant vers le token suivant.
- Exemple G1 et)à
- Etape1: pop,push(C),push(B),avance. (S→dBC)
- Etape2:pop,push(B), avance (B→dB). pop et push(B) s'annulent, donc:
  - Etape2: avancer
- Etape3: pop, …

opérations

Entrée	Pile	Opérations	Production utilisée
d'dcbb#	S$	pop,push(C),push(B),avancer	S→dBC
dd‘cbb#	BC$	pop,push(C),avancer	B→dC
ddc‘bb#	CC$	pop,avancer	C→c
ddcb‘b#	C$	pop,push(B),avancer	C→bB
ddcbb’#	B$	pop,avancer	B→b
ddcbb#’	$	acceptation	

Remarque:
B→cCB pop et push(B) s’annulent

3 Table d'analyse

1. S →dBC
2. B → dC
3. B → *b*
4. *C → bB*
5. C → c

Token courant					
Sommet de la pile		b	c	d	#
	S			pop,push(C),push(B),avancer	
	B	pop, avancer		pop,push(C), avancer	
	C	pop, push(B),avancer	pop, avancer		
	$				accepter

Implémentation d'un parseur en Java

Soit la grammaire G5

1. S →bScA
2. S → cbd
3. A → bcA
4. A → d

Table d'analyse

		b	c	d	#
	S	pop,push(A), push(C), push(S), avancer	pop,push(d), push(b), avancer		
	A	push(c), avancer		pop, avancer	
	b	pop, avancer			
	c		pop, avancer		
	d			pop, avancer	
	$				acceptation

Parseur Java

```
class TokenManager {
int tCourant;// token en cours
String entree;

public TokenManager(String[] args) {
if (args.length > 0)
entree = args[0];
else
entree = "";
tCourant = 0;
System.out.println("Entrée:" + entree);

}

// Normalement cette méthode doit retourner un token -> une chaine
// de caractères
public char getTokenSuivant() {

if (tCourant < entree.length())
return entree.charAt(tCourant++);
else
return '#';
}}
```

```
class Parseur {
private TokenManager tm;
private Stack<Character> pile;
private char tokenCourant;

public Parseur(TokenManager tm) {
this.tm = tm;
tokenSuivant();
pile = new Stack<>();
pile.push('$');
pile.push('S');

}
private void tokenSuivant() {
tokenCourant = tm.getTokenSuivant();
}
```

```
if (tokenCourant == 'b') {
pile.pop();
pile.push('A');
pile.push('c');
pile.push('S');
tokenSuivant();

} else if (tokenCourant == 'c') {
pile.pop();
pile.push('d');
pile.push('b');
tokenSuivant();
} else
termine = true;
break;
case 'A':
if (tokenCourant == 'b') {
pile.push('c');
```

```
} else
termine = true;
break;
case 'b':
case 'c':
case 'd':
if (pile.peek() == tokenCourant) {
pile.pop();
tokenSuivant();
} else
termine = true;
break;
case '$':
termine = true;
break;
} // Fin switch
} // Fin Loop
return (tokenCourant == '#'
```

Exercices

1. Construire la table d'analyse pour:
    - 1. S → bSb
    - 2. S → cAc
    - 3. A → bAA
    - 4. A → cASAb
    - 5. A → dcb
   - Implémenter le parseur en Java.
   - Tester le programme avec les chaînes: cdcbc, bcdcbcb, cbdcbdcbc, ccdcbcdcbcdcbbcr, cdcbbb, cdcb et ε.
2. Construire la table d'analyse
    - S → bSc
    - S → d
   - Implémenter le parseur en Java
   - Tester avec les chaînes: d, bdc, bbdcc, b, c, bbcd et bcdd
3. Mêmes questions pour les grammaires suivantes:
   - G3:
     - S → bcdefg
   - G4
     - S → bABCD
     - **2.** S → c
     - **3.** A → bA
     - **4.** B → bB
     - **5.** D → d

Les grammaires LL(1)

- Soit la grammaire G1:
  1. S→BC {b,d}
  2. S→CB {c,e}
  3. B→bB {b}
  4. B →d {d}
  5. C →cC {c}
  6. C →e {e}

- Les productions de chaque groupe ayant le même non terminal à gauche ont des ensembles de sélection disjoints (1 et 2, 3 et 4, 5 et 6, alors nous pourrons conclure que G1 est une grammaire LL(1)

On définit Premierss(X): X une proto phrase(partie droite d'une production), par l'ensemble des terminaux que peut générer X.

Exemple Premiers (BC), on peut remplacer B par bB ou par d. donc Premier (B,C)={b,d}

- On définit l'ensemble de sélection pour une production, l'ensemble des terminaux pour lesquels cette production peut être utilisée.

- Soit P : a-->b (b dans V*) une production, (si la grammaire ne contient aucune ε-production alors Sélection(P)=Premier(b).

- Opérations dans le cas d'une opération qui commence par non terminal:
  - pop, si la partie gauche est différente du symbole le plus à droite de la partie droite.
  - push, dans l'ordre inverse les symboles de la partie droite (sauf le premier si c'est un terminal ou le dernier s'il apparaît aussi dans la partie gauche.
  - Avancer si la production commence par un terminal.
- Table d'analyse

	b	c	d	e	#
S	pop,push(C) ,push(B)	pop,push(B) ,push(C)	pop,push(C) ,push(B)	pop,push(B) ,push(C)	
B	avancer		pop,avancer		
C		avancer		pop,avancer	
$					accepter

Exemple G2

G2	Premiers
1. S → BC	Pre{b, c}
2. S → CB	{c, e}
3. B → bB	{b}
4. B → c	{c}
5. C → cC	{c}
6. C → e	{e}

- Les ensembles Premiers(BC) et Premiers(CB) ont c en commun, si on doit développer S et le token courant est ‘c’, alors on ne peut pas décider laquelle des deux productions 1 ou 2 appliquer.
- Donc la grammaire G2 n’est pas LL(1)

Exercices

- Déterminer l'ensemble de sélection et la table d'analyse:
  1. S -> bS
  2. S -> CS
  3. S -> c
  4. C ->d

Exercice

- Déterminer les ensembles de sélection pour chaque production et construire la table d'analyse
- Gex1
  1. S → bS
  2. S → CS
  3. S → c
  4. C →d

Gex2

1. S → BC	{b, c}
2. B → bB	{b}
3. B → ε	{c}
4. C → c	{c}

- Premier(B) ={b,ε} => Premier(C) inclus dans Premier(BC).
- Si X non vide, Premier(XY)=Premier(X), et X n'est pas dans la partie gauche d'une ε-production

Ensembles des successeurs :

- G3
  - 1. S →BC {b, c}
  - 2. B → bB {b}
  - 3. B →ε {c}
  - **4. C → c {c}**

- Suivant(N): ens. des terminaux qui peuvent être à droite de N dans une dérivation
- L'ensemble de sélection d'une ε-production est l'ensemble Suivant du non terminal à gauche de la production.
- Le symbole # est toujours dans l'ensemble Suivant de l'axiome.

- Premier (c)={c}
- Premier(bB)={b}
- Premier (BC) = Premier(B) U Premier(C), à cause de la production 3.={b,c}
- Suivant(B)=Premier(C)=Premier(c)={c}

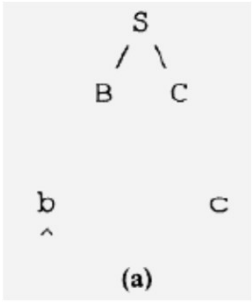
Ensemble de sélection pour une ε-production

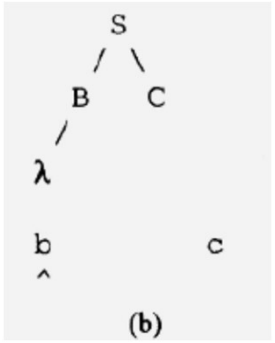
Remarque:
Utiliser une ε-production si le token courant peut être généré par un non terminal qui suit le non terminal à gauche de la ε-production

Grammaires avec des ε-productions

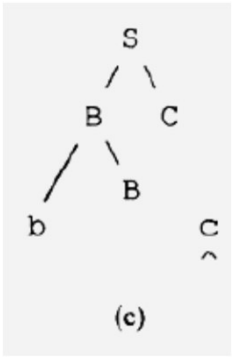
G4

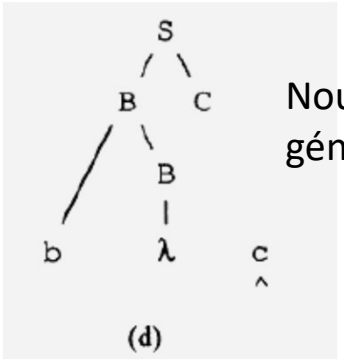
1. S → BC {b, c}
2. B →bB {b}
3. B → ε {c}
4. C →c {c}



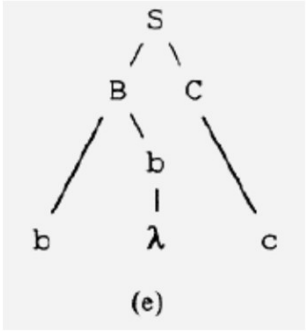


C ne peut pas générer b





Nous utilisons la pr 3, pour générer un c par BC



. S -> AD

2. A → BC

3. B -> b

4. B →ε

5. C→c

6. C → ε

7. D → d

G5 Sélection

1, S -> BCd Premier(BCd)=Premier (B) U Premier(Cd) et Premier (Cd)=Premier(C)U Premier (d)={b,c,d}

2. B -> bb {b}

3. B →ε Suivant(B)=Premier(Cd)={c,d}

4. C —> cc {c}

5. C→ε Suivant(C)={d}

G5 est LL(1).

G6

1. S → Bd {b}
2. B → bC {b}
3. C → cC {c}
4. C → ε Suivant(C)= Suivant(B)={d}

Remarque

Si A→pB (p une proto phrase), alors suivant(A) est inclus dans Suivant(B)

L'ensemble Suivant de la partie gauche d'une production est inclus dans l'ensemble suivant du non terminal à droite de la partie droite de la production.

G7

1. S → Be Premier(Be)={b}
2. B → bCD {b}
3. C → cC {c}
4. C→ε Suivant(C)=Premier(D) U Suivant(B)={d,e}
5. D → dD {d}
6. D →ε Suivant(D)=Suivant(B)={e}

1→Suivant (B)={e}

2→Suivant(B) inclus dans Suivant (D)

Comme D s'annule par 6, alors 2→Suivant(B) inclus dans Suivant(C)

Cas d'une production avec une partie droit annullable.

- G8
  1. S -> AD
  2. A → BC
  3. B -> b
  4. B →ε
  5. C→c
  6. C → ε
  7. D → d

Production	Sélection
P1	b,c,d (Premier(AD)=Premier(A) U Premier (D) ; A peut aussi s'annuler)
P2	b,c,d (Premier(BC) U Suivant(A) et Premier(BC)=Premier(B) U Premier(C))
P3	b
P4	c,d (Suivant(B)=Premier(C) U suivant (A))
P5 P6	c d (Suivant(C)=Suivant(A))
P7	d

Exemples les chaînes : bd et d pour expliquer l'ens. de sélection de p2.

Règles pour les ensembles de sélection

1. L'ensemble de sélection d'une production avec une partie droite non annulable est l'ensemble Premier de la partie droite.
2. L'ensemble de sélection d'une production avec une partie droite annulable (qui inclus les ε-productions) est l'union de l'ensemble premier de la partie droite et l'ensemble suivant de la partie gauche.

Exemple

- **G9**
  1. **S → feS**
  2. **S → BCD**
  3. **B → b**
  4. **B → ε**
  5. **C → c**
  6. **C → ε**
  7. **D → d**
  8. **D → ε**

Production	Sélection
P1	f
P2	b,c,d, # (Premier(BCD) U Suivant(S)=P(B) U P(C) U P(D) U S(S))
P3	b
P4	c,d,#
P5	c
P6	d,#
P7	d
P8	#

- **arbre d'analyse de : fefe**

G10

1. S →feS
2. S → BCD
3. B →b
4. B→ε
5. C → c
6. C →ε
7. D → d

Production	Sélection
P1	f
P2	b,c,d
P3	b
P4	c,d
P5	c
P6	d
P7	d

Table d’analyse d’une grammaire avec ε-productions

- G11

S -> BC	{b,c,#}
2. B -> bB	{b}
3. B --> ε	{c,#}
4. C --> cCb	{c}
5. C --> ε	{b,#}

	b	c	#
S	pop,push(C),push(B)	pop,push(C),push(B)	pop,push(C),push(B)
B	avancer	pop	pop
C	pop	Pop,push(b),push(C),avancer	pop
b	Pop,avancer		
$			acceptation

- Ex5
  1. S --> Ae
  2. A --> fBCD
  3. B --> b
  4. B --> ε
  5. C --> c
  6. C --> ε
  7. D --> d
  8. D --> ε

Non terminal	Premier
S	f
A	f
B	b
C	c
D	d

Non terminal	Suivant
S	#
A	e
B	c,d,e
C	d,e
D	e

Production	Sélection
P1	f
P2	f
P3	b
P4	c,d,e
P5	c
P6	d,e
P7	d
P8	e

- Ex6
  1. S -->Ad
  2. A -->B
  3. B --> C
  4. C -->c
  5. C-->ε

Production	Sélection
P1	c,d (Premier(Ad) =Premier(A) U {d})
P2	c,d
P3	c,d
P4	c
P5	d

- Ex7.1
  1. S →dA
  2. A →BC
  3. B→ b
  4. B →ε
  5. C →c
  6. C →ε

Production	Sélection
P1	d
P2	b,c,#
P3	b
P4	c,#
P5	c
P6	#

- Ex7.2
  1. S → ABC
  2. A → dSd
  3. A→ε
  4. B-->bBe
  5. B-->ε
  6. C-->c
  7. C-->ε

Production	Sélection
P1	d,b,c,#
P2	d
P3	b,c,d,#
P4	b
P5	c,d,e,#
P6	c
P7	d,#

Ex8

1. S --> **SSb** **{c}**
2. S --> c {c}

S→cL {c}
L→SbL {c}
L-->ε {#}

1. S -> eBd {e}
2. S -> eCd {e}
3. B -> bB {b}
4. B -> ε {d}
5. C --> c {c}

Utiliser la technique de factorisation gauche

1. S-->eR {e}
2. R-->Bd {b,d}
3. R-->Cd {c}
4. B-->bB {b}
5. B-->ε {d}
6. C-->c {c}

Ou bien

1. S-->eMd {e}
2. M-->B {b,d}
3. M-->C {c}
4. B-->bB {b}
5. B-->ε {d}
6. C-->c {c}

8.3

1. S --> b,S {b}
2. 2. S --> b {b}

8.3

1. S --> bR {b}
2. R-->,S {,}
3. R--> ε {#}

8.4

1. S -->bA	{b}
2. S -->b	{b}
3. A -->bb	{b}
4. A -->bc	{b}

1. S -->bM	{b}
2. M -->ε	{#}
3. M -->A	{b}
4.A-->bR	{b}
5.R-->b	{b}
6.R-->c	{c}

8.5

1. S -> bA	{b}
2. S -->Ac	{b,c}
3. A--> bA	{b}
4. A --> ε	{c,#}

8.5 Etape 1

1. S -> bA	{b}
2. S -> bAc	{b}
3. S -> c {c}	
4. A -> bA	{b}
5. A -> ε	{c,#}

Étape 2

1. S -> bAM	{b}
2. M -> c	{c}
3. M-->ε	{#}
4. S -> c {c}	
5. A -> bA	{b}
6. A -> ε	{c,#}

Exemple : La grammaire

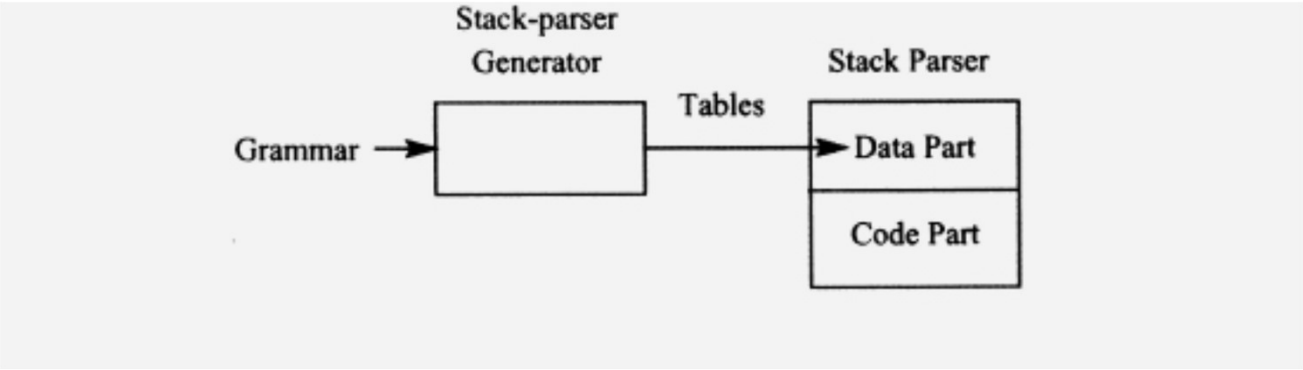
1. E→T listeT
2. listeT→+ T listeT
3. listeT→ε
4. T→F listeF
5. listeF→* F listeF
6. listeF→ε
7. F→ID
8. F→(E)

Correction des exercices

- ex1

Table driven stack parser

- Le programme sera décomposé en deux parties
  - Tables contenant la description d'une grammaire
  - Le code du parseur



Parseur Récursif descendant

- écrit une méthode pour chaque non terminal.
- Si des productions associées un terminal sont récursives alors, les méthodes associées seront aussi récursives (d'où le nom récursif descendant).

Gr1 (langage engendré b*cde)

1. S → BD {b, c}
2. B → bB {b}
3. B → c {c}
4. D → de {d}

Chaque non terminal peut générer un ensemble de chaînes

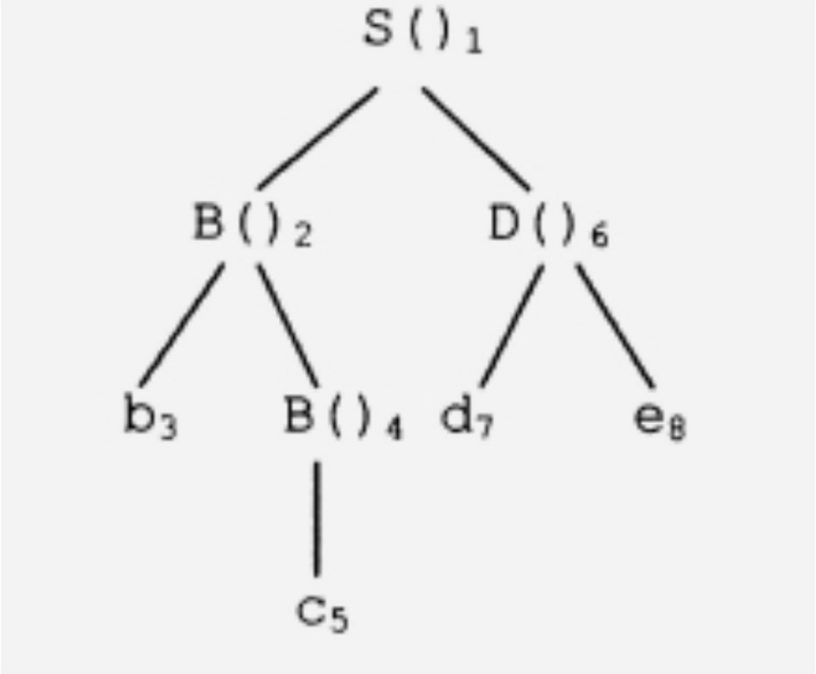
Soit la chaîne « bbcde », bbc est générée par B, de est générée par D, et « bbcde » est généré par S

```
package com.parseurs.recursif;

public class Parseur {
private TokenManager tm;
  private char tokenCourant;
  //-----------------------------------
------
  public Parseur(TokenManager tm)
  {
    this.tm = tm;
    avancer();
  }
  //-----------------------------------
------
  private void avancer()
  {
    tokenCourant = tm.tokenSuivant();
  }
  //-----------------------------------
------
  private void consommer(char attendu)
  {
    if (tokenCourant == attendu)
      avancer();
    else
      throw new RuntimeException(
                    "Token attendu \""  +
attendu  +  "\"");
  }
  //-----------------------------------
------
  public void parse()
  {
    S();
    if (tokenCourant != '#')
      throw new RuntimeException(
"Expecting end of input");
```

```
  }
  //----------------------------------------
  private void S()
  {
    B(); //  S -> BD
    D();
  }
//----------------------------------------
  private void B()
  {
    switch(tokenCourant)
    {
      case 'b':
        consommer('b'); //  B -> bB
        B();
        break;
      case 'c':
        consommer('c'); //  B -> c
        break;
      default:
        throw new RuntimeException(
"Token attendu \"b\" or \"c\"");
    }
  }
  //----------------------------------------
  private void D()
  {
    consommer('d'); //  D -> de
    consommer('e');
  }
}
```

Arbre des appels



Code associé aux productions

- Contenu dans la partie droite de la production
  - Terminal t → consommer (t);
  - Non terminal N : N();
  - Epsilon → ;

Exercices

Implémenter un parseur récursif descendant pour

1. E → +EE

2. E → -EE

3. E → *EE

4. E →/EE

5. E → b

6. E → c

7. E → d

Tester avec

b, c, d, +bc, - /bc* cd, +-*/bcbcd, bc, +b, *bcd,

and λ.

Implémenter un parseur récursif descendant pour

1. S → BCD

2. B → bB

3. B → epsilon

4. C → cC

5. C → epsilon

6. D → dD

7. D → epsilon

Tester avec “”, b, c, d, bc, bd, cd, ccc, bcb, db, and dc.

Traduction: génération de code

Ga (On veut afficher toute occurrence de c qu'on rencontre dans une chaîne

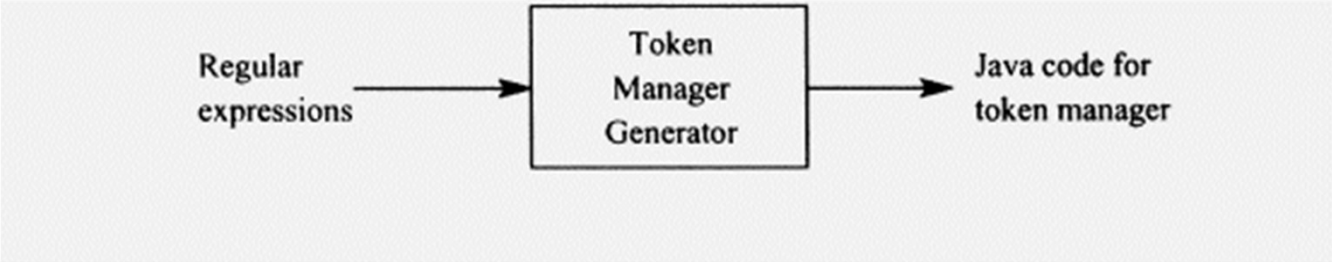
1. S → bCd {b}
2. C → cC {c}
3. C → λ {d}

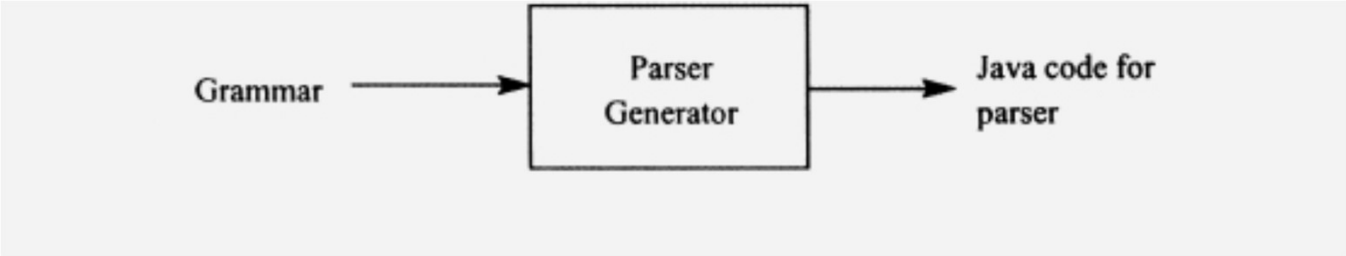
BNF

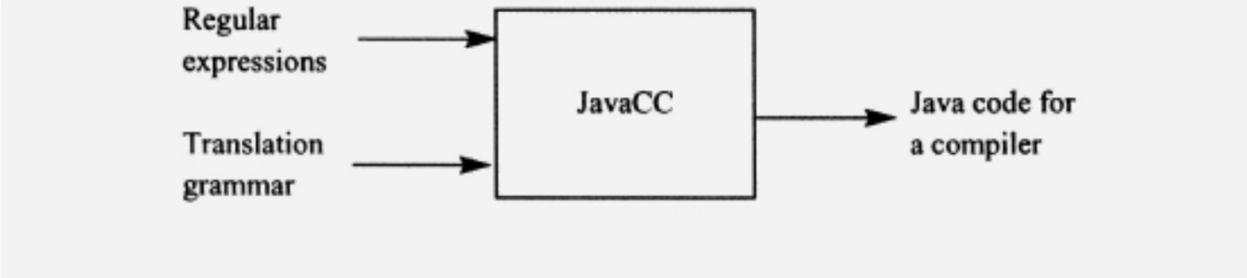
- S : "b" C "d" avec une action (« afficher les c »)
- C : "c" C | λ

```
void S():{}
{
« b » C() « d »
}
void C():{}   ← pour les var locales
{
« c »
{System.out.printf(« c »);}
C()
|
{}
}
```

JavaCC







- Génère des parseurs TOP-DOWN
- Génère des parseurs descendant récursifs, écrit une méthode pour chaque non terminal
- http://javacc.org/

Composants d’un compilateur

- Token Manager
- Parseur
- Générateur de code

Expressions rationnelles étendues

- ‘‘a’’ | ‘’b’’ (les espaces qui ne sont pas entre guillemets sont ignorés)
- (‘’a’’)+ (et non ‘’a’’+)
- (‘’a’’)*
- (‘’a’’)?
- ~[“b”,"d"]: un caractère n’appartenant pas à la classe de caractères
- ~[]: représente tous les caractères

Section du fichier jj

- Options
- Classe principale du parseur.
- Déclarations Token Manager
- Skip
- Token
- Les méthodes

options

```
options {
STATIC =false;
JDK_VERSION = "1.8";
COMMON_TOKEN_ACTION=true;
DEBUG_TOKEN_MANAGER=true;
DEBUG_PARSER=true;
}
```

Classe principale du parseur

```
PARSER_BEGIN (Parseur)
/* Code java doit contenir au moins la classe du parseur.
ici */
package comp2.com;
import java.io.*;
 public class Parseur {

/*private   G2SymTab st;
private G2GenCode cg;
private  PrintWriter pw;*/
  public static void main(String []  args) throws IOException {
    if (args.length!=1)
    { System.out.println ("il faut fournir le nom du fichier");
    System.exit(1);
   }

      //Fichiers en entrée et en sortie
   String inFichier= args[0] + ".ss";
   String outFichier =args[0] + ".aa";

   //Les flux
   FileInputStream fis=new FileInputStream(inFichier);
PrintWriter    pw=new PrintWriter (outFichier);
   //Construction des composants du compilateur
/*G2SymTab   st=new G2SymTab();
G2GenCode  cg=new G2GenCode();*/


//Initialisation des variables du parseur
/*parseur.outFile=pw;
parseur.st.st;
parseur.cg=cg;*/

  }
}
/*
public class G2SymTab {
}


public class G2GenCode {
}
  */

PARSER_END(Parseur)
```

```
TOKEN:
{
/* Expressions qui décrivent les tokens à passer au parseur*/
/* Si on donne un nom à une expression, on peut l'utiliser dans d'autres expressions <NOM >*/

/*On pourrait lister directement "println" sans lui associer un nom et dans ce cas
là un sera un token anonyme*/
 <PRINTLN:"println" >
|
< ENTIER:(["0"-"9"])+ >
{
  System.out.println("Entier:" + matchedToken.image);
}
|
< ID: ["A"-"Z","a"-"z"](["A"-"Z","a"-"z","0"-"1"])* >
|

"="
|
";"
|
"("
|
")"
|
"+"
|
"-"
|
"*"
|
< ERROR:~[] > //pour laisser le traitement des erreurs au parseur.
}
```

- Une expression régulière peut avoir un nom:
  - <UNSIGNED: (["0"-"9"])+>

```
TOKEN_MGR_DECLS:
{
  /* déclarations des variables et des méthodes qui seront
utilisées par le token manager
  */

//sera appelée avant chaque token
public void CommonTokenAction(Token t)
 {
System.out.println("Commun:" + t.image);
 }
}
```

skip

```
/*toute alternative dans skip ou token doit être entre < > si elle n'est pas entre ""
exemple <("a")+>*/

SKIP:
{
/*Expressions  qui décrivent les tokens à ignorer par le token manager*/
" "
|
"\n"
|
"\r"
|
"\t"
}
```

Expressions régulières

- **<u>Les métacaractères :</u>**
- Un métacaractère est un caractère spécial qui a une signification particulière dans une expression régulière.
- Les métacaractères suivants sont supportés par Flex : ^ . [ ] $ ( ) * + ? | { } \
  - [ ] : définit une classe de caractères.

Les symboles supportés dans une expression régulière

c	le caractère c
.	Un caractère quelconque à l'exception du caractère "nouvelle ligne".
^	Indique le début d'une ligne.
$	Indique la fin d'une ligne
[abg]	un caractère appartenant à l'ensemble {a,b,g}.
[f-m]	un caractère compris entre f et m
[^a-z\n]	un caractère n'appartenant pas à la classe [a-z] et différent du caractère nouvelle ligne
exp*	0 ou plusieurs occurrences de l'expression règulière exp
Exp?	0 ou 1 occurrence de l'expression rationnelle exp
exp+	1 ou plusieurs occurrences de l'expression rationnelle exp
exp{3,8}	Une concaténation de trois à huit occurrences de l'expression exp
exp{3,}	au minimum une concaténation de trois occurrences de l'expression exp
'exp{4}	exactement quatre occurrences de l'expression exp
\x	Une séquence d'échappement ANSI-C si x est un 'a', 'b', 'f', 'n', 'r', 't', ou 'v', sinon le caracètre \ est utilisé pour échapper un caractère spécial, par exemple pour avoir le caractère ^ dans une expression régulière il faut le faire précéder du caractère \^).
exp1/exp2	exp1 mais seulement si elle est suivie de **exp2**.

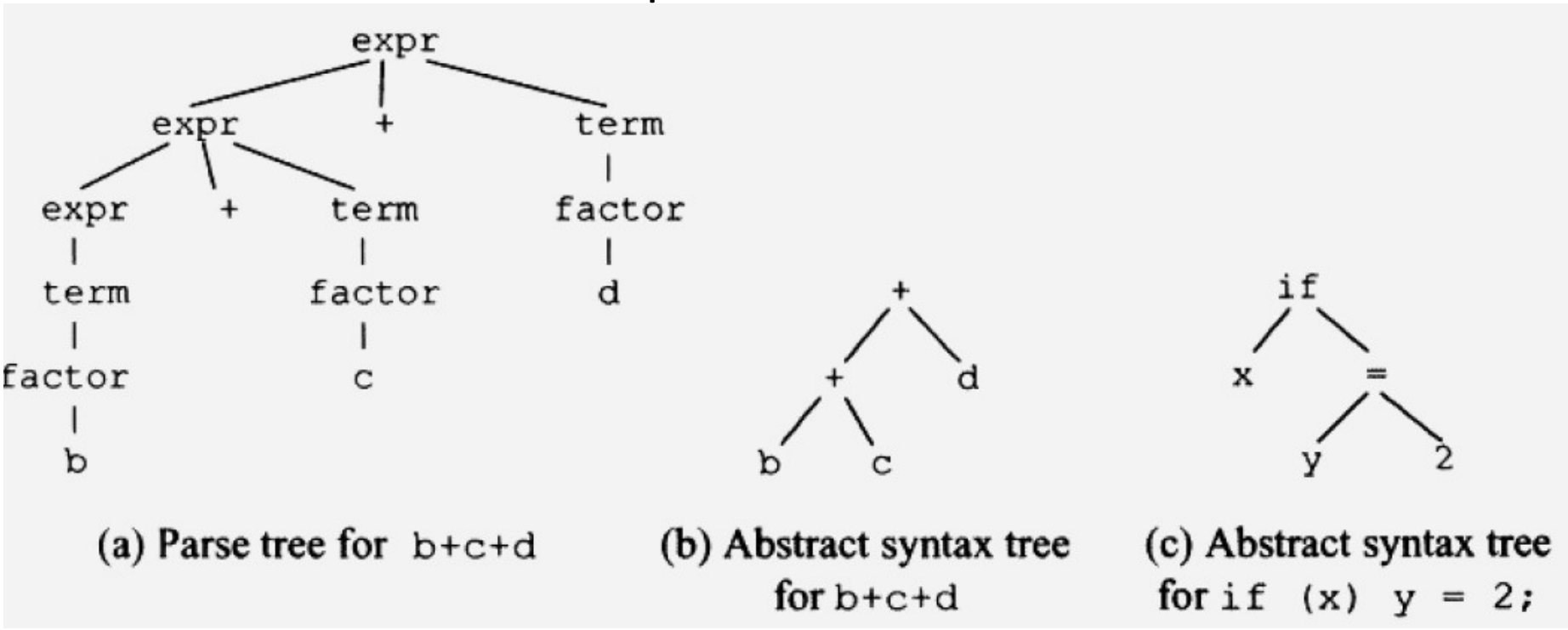
AST TAC

- Les compilateurs utilisent deux représentations internes comme langage intermédiaire d'un code source.
  - AST (Abstract Syntax Tree)
  - TAC (Three Addresses Code)
- Source → AST | TAC → Sémantique → Optimisation → Génération du code.

AST

Une branche contient un opérateur ou un symbole (+ , - , if, …)

Les feuilles contiennent les opérandes.



(a) Parse tree for `b+c+d`

(b) Abstract syntax tree for `b+c+d`

(c) Abstract syntax tree for `if (x) y = 2;`

TAC

- Une instruction est constituée d'une opération et de trois adresses au max (une pour le résultat et deux pour les opérandes)

```
(+, @t1, b, c)      // add b and c and assign to @t1
(+, @t2,@t1,d)      // add @t1 and d and assign to @t2
```

where `@t1` and `@t2` are compiler-generated temporaries. For Figure 4.8c, the three-address code is

```
(if_false, x, @L0, null)    // if x is false, goto @L0
(=, y, 2, null)             // assign 2 to y
@L0:
```