Table des matières

Théorie des Langages et Compilation (TLC)

ECL - MI - 2A - 2003/2004

Modalités

Responsable : Alexandre.Saidi@ec-lyon.fr

* Nb cours et TD : 6 à 7 séances de 2h.

* Nb TP : 3 à 4 séances de 2h.

* Chargés de TD : Moi même.

* Projets: à définir

* Examen final : voir la date !

Contrôles de connaissances:

Note (Projet/TP+Examen), syllabus cours

Plan du cours

1. **Introduction :** Notions préliminaires en théorie des langages, structure d'un compilateur, bootstrap, ...
2. **Eléments théoriques** : Grammaires et Langages Formels, types des langages, ...
3. **Analyse Lexicale et syntaxique :** éléments théoriques, analyse descendante, analyse ascendante, Lex, Yacc
4. **Arbre de syntaxe abstraite :** structure et représentation
5. **Analyse statique :** typage
6. **Blocs l'activation** pour fonctions/variables locales
7. **Structure de** la machine d'exécution
8. **Génération de code** intermédiaire, optimisation , assembleur, allocation des registres
9. **Outils (pour les TPs, Projet) : programmation, Lex, Yacc**

Bibliographie

Alfred AHO, Ravi SETHI, Jeffrey ULLMAN *Compilers: Principles, Techniques and Tools (Dragon Book)*

Alfred AHO, Ravi SETHI, Jeffrey ULLMAN *Compilateurs, principes, techniques et outils* - InterEditions - 1986

R. FLOYD, R. BIEGEL *Le langage des machines, introduction à la calculabilité et aux langages formels*

Thomson Publishing - 1995

R. WILHELM, D. MAURER

Les compilateurs : Théorie, construction, génération

Masson - 1994

B. GROC, M. BOUHIER

La Programmation Par Syntaxe

Dunod - 1990

WOLPER *Introduction à la calculabilité*

InterEditions - 1991

THALMANN, LEVRAT *Conception et implantation de langages de programmation*

Gaëtan Morin - 1979

A. AHO, J. ULLMAN *Concepts fondamentaux de l'informatique*

GROSS, LENTIN *Notions sur les grammaires formelles*

Gauthier Villars 1967

N. WIRTH *Algorithmes + structures de données = programmes*

1° édition - Masson - 1976

PEMBERTON *The P4 Compiler*

Prentice Hall

STEHLÉ, HOCHARD *Ordinateurs et langages*

Ellipses 1989

KNUTH *The Art of Computer Programming*

3 vol. - Addison Wesley - 1973

Et autres Accepteurs, Programmation par la syntaxe, ...

etc.

Quelques exemples de sujet de projet :

– Langages et automates cellulaires

– Langages et algorithmes génétiques, vie artificielle

– Egrep et les expressions régulières

– Langages objets et compilation

– Langages fonctionnels et compilation

– Langages logiques et compilation

– Les langages du web (HTML/XHTML)

– Les langages du web (XML)

– Langage Postscript

– Mécanismes d'exécution, les messages d'erreur sous Unix; optimisation du code généré (Dragon Book)

– Evolution des compilateurs avec les architectures

– Audit d'un compilateur

– La machine virtuelle Java

– Programmation grammaticale Logique, grammaires Affixes/Att

– DCG

– Outils : Lex et Yacc, Xref, lint,

– Méta Compilateurs : MIRANDA, FNC2,..

– OcamlLex/OcamlYacc

– ...

Chapter 1

Introduction

1 Généralités

compilateur *Personne qui réunit des documents dispersés"*

(Le Petit Robert)

compilation *Rassemblement de documents (Le Petit Robert)*

Mais le programme qui "réunit des bouts de code dispersés" (binaires) s'appelle un *Editeur de Liens*

$\Longrightarrow$ On appelle "compilateur" une autre chose :

compiler (Webster)

1: one that compiles

2: a computer program that translates an entire set of instructions written in a higher-level symbolic language (as COBOL3) into machine language before the instructions can be executed

Langage machine?

- Natif

- Intermédiaire

- Emulé

1.1 Notions de base

machine hardware : le processeur

machines virtuelle une machine qui reconnaît un certain nombre d'instructions qui ne sont pas toutes "natives" pour la machine hardware.

EXAMPLE 1. *le compilateur C produit du code assembleur qui fait appel à un ensemble de fonctions système (e.g. les E/S). Donc il produit du code pour la machine virtuelle définie par les instructions hardware, plus les appels système.*

Lorsque l'on écrit et exécute un programme par exemple en C, on dit que le programme s'exécute sur une machine virtuelle (étendue) C .

Quelques exemples de machine virtuelle :

* Programmes d'application

* Langage de programmation évolué

* Langage d'assemblage

* Noyau du système d'exploitation

* Langage machine

Qu'est-ce qu'un interpréteur?

Un programme qui prend en entrée un autre programme, écrit pour une machine virtuelle, et l'exécute sans le traduire.

Exemples :

* l'interpréteur VisualBasic,

* le toplevel Prolog/Lisp/Caml,

* l'interpréteur C/PERL/...,

* etc.

Qu'est-ce qu'un compilateur?

Un traducteur, en général d'un langage évolué vers un langage moins expressif.

Exemples:

- Compilation des expressions régulières (utilisées dans le shell Unix, les outils sed, awk, l'éditeur $Emacs$ et les bibliothèques des langages C, $Perl$ et $Ocaml$)

- Minimisation des automates (compilation d'un automate vers un automate plus efficace)

- De *LaTex vers Postscript ou HTML (latex2html/Hevea)*

- De C vers l'assembleur x86 plus les appels de système Unix/Linux

- De C vers l'assembleur x86 plus les appels de système Win32

⇒ On peut "compiler" vers une machine... interprétée (e.g. bytecode Java, bytecode Ocaml)

⇒ *Parfois, une partie de code compilé a besoin d'interprétation ! (e.g. printf direct/indirect)*

Exemples :

```
printf("il n'y a que %d de vrai !!", 42);
```

Mais (autre version peu compilable !)

```
int pas_vraiment_compilable(char * ch, int i)
       {return printf(ch, i); }
```

Appel :

```
pas_vraiment_compilable
    ("il n'y a que %d de vrai !!", 42);
```

La décompilation :

On va d'un langage moins structuré vers un langage évolué (sens inverse).

Exemples: retrouver le source C à partir d'un code compilé (d'un programme C).

Applications:

* Récupération de vieux logiciels

* découverte d'API cachées

* . . .

La décompilation est autorisée en Europe (sous conditions).

Pouvoir d'expression des langages

De moins évolué vers plus évolué :

0- Microcode, Langage Machine

1- Langage assemblé (assembleur)

2- Langage Fortran, C, Pascal, ADA, ...

3- Langages Objets?

4- L4G : SQL, Langages Logiques , méta langages divers

5- Langues Naturelles

Exemple :

En français : *trouver les produits dont le prix >100 Euros*

SQL : *select Nom from $\mathcal{R}$ where Prix >100*

Prolog : *$\mathcal{R}$ (Nom,Prix, ...), Prix >100.*

C : *while(! feof(f))*

{read(f,Enreg);

if (Enreg.Prix >100) output(Enreg.nom); }

Autres : *Compiler le code $C \Rightarrow$ Assembleur $\Rightarrow$ binaire ...*

1.2 Bootsrapping (tuiles)

Procédure de bootsrap pour un compilateur de L :

- Ecrire (en assembleur) un mini compilateur C_0 du langage P_0 ($P_0 \subset P_1$)

Par exemple, la langage P_0 comprend *if* + *goto* (qui seront traduits en assembleur par C_0)

- Ecrire en langage P_0 le compilateur (optimisé) C_1 de P_1 (e.g. *while* traduite en *if* + *goto*)

- Compiler C_1 avec $C_0 \Longrightarrow$ le compilateur C_1

- Compiler C_1 avec C_1

"portage" des compilateurs d'une machine à une autre?

- Avec bytecode : réécrire la machine virtuelle (interpréteur de bytecode)

- Sans bytecode : bootstrap ? noyau ? émulateur ? réécrire tout ?

2 Structure logique d'un compilateur

Un compilateur est un logiciel très complexe:

– On essaye de réutiliser au maximum ses composantes,

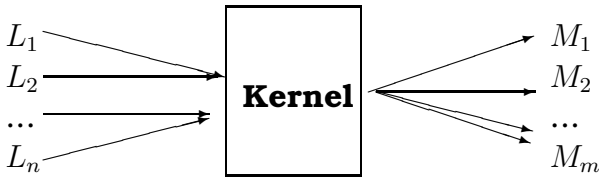
– On identifie :

  – un **front-end** lié au langage source

  – un **back-end** lié à la machine cible

  – un **code intermédiaire** commun sur lequel travaille le *coeur* du système

$\Rightarrow$ Cela permet d'écrire les nm compilateurs de n langages source à m machines cible en écrivant seulement un kernel (noyau, $coeur$), n front-ends et m back-ends :



2.1 Structure détaillée d'un compilateur

Front-end :

Analyse lexicale (flot de caractères → lexèmes)

– théorie: langages réguliers

– outils: automates finis

– logiciels: *Lex* (et similaires)

Analyse syntaxique (flot de lexèmes→ accept/rejet)

– théorie: langages algébriques

– outils: automates à pile

– logiciels: *Yacc* (et similaires)

Analyse sémantique (souvent en même temps que l'analyse syntaxique)

– Objectifs :

  – construction de l'arbre de syntaxe abstrait AST, ajout des actions sémantiques;

  – vérification des types, portée des variables, tables des symboles, gestion des environnements etc.;

  – production de l'AST décoré et les tables des symboles

– outils: grammaires attribuées, ou à la main

– logiciels: *Yacc, FNC2, MIRANDA*, ...

Traduction en code intermédiaire

(souvent sous forme d'un arbre, indépendant de l'architecture cible)

Kernel/Noyau/Coeur:

– linéarisation du code intermédiaire (transformation en liste d'instructions du code intermédiaire – e.g. expression infixe en postfixe –, etc.)

– différentes optimisations

  – analyse de vie des variables et allocation de registres

  – transformation des boucles (déplacement des invariants, transformations affines)

  – function inlining, dépliage des boucles, etc.

Back-end:

Les seules phases dépendantes de la machine assembleur cible :

- sélection d'instructions (passage du code intermédiaire à l'assembleur de la machine cible, éventuellement abstrait par rapport aux noms des registres)
- émission du code (production d'un fichier écrit dans le langage assembleur de la machine cible)
- assemblage (production des fichiers contenant le code machine)
- édition des liens (production du fichier exécutable)

2.2 La machine assembleur

Il y a plusieurs microprocesseurs, classés en deux grandes classes :

CISC (Simple Instruction Set, e.g. Intel, Motoral 68xxx),

RISC : (Reduce Instruction Set, e.g. MIPS, PPC, SPARC, etc).

Pour ce cours, on utilisera comme machine cible le processeur Intel/Motorola avec l'écriture d'un émulateur de code intermédiaire.

3 Exemples de compilation

3.1 Code généré pour un exemple simple (I)

Les phases cachées d'un compilateur: l'apparence

```
alex> cat fact.c

#include <stdio.h>
int fact(int i)
{if (i==0) return(1);
 else {return(i*fact(i-1));};
}
int main()
{ printf ("%d",fact(3)); return(0);}
```

Les fichiers produits par *gcc* :

```
alex> gcc -o fact fact.c
alex> ls -l fact*
-rw-r--r--  1 alex users  3744 <date/heure> fact.c
-rwxr-xr-x  1 alex users 11142    ....        fact
```

Les phases cachées d'un compilateur: la réalité

```
alex>gcc -v -o fact -save-temps fact.c

[...] beaucoup de détails,
      appel de cpp0, cc1, as, ld [...]
```

Les fichiers produits :

```
alex> ls -l fact*
-rw-r--r-- 1 alex users  3744 ... fact.c
-rw-r--r-- 1 alex users   760 ... fact.s
-rw-r--r-- 1 alex users   896 ... fact.o
-rw-r--r-- 1 alex users 16644 ... fact.i
-rwxr-xr-x 1 alex users 11142 ... fact
```

▷ 4 étapes:

1) préprocesseur: **cpp0** traduit **fact.c** en **fact.i**

2) compilateur: **cc1** traduit **fact.i** en **fact.s**

3) assembleur: **as** traduit **fact.s** en **fact.o**

4) éditeur de liens: **collect2** (un enrobage de **ld**) transforme **fact.o** en exécutable **fact**, en résolvant les liens externes

3.2 Code généré pour un exemple simple (II)

Soit le fichier *essai.c* suivant :

```
#include <stdio.h>
int main ()
{int i=2; // pour distinguer en assembleur
 int j=1;
 for (i=0;i<10;i++) {j=6*i;};
 printf("Resultat: %d", j);
 exit(0);
}
```

Le fichier *affine.s* produit (compilation sous *Suse*):

```
.file   "essai.c"
    .section    .rodata
.LC0:
    .string "Resultat: %d"
    .text
.globl main
    .type   main, @function
main:
    pushl   %ebp                -- préparations et
    movl    %esp, %ebp          -- sauvegardes
    subl    $8, %esp            -- allocations des ptrs
    andl    $-16, %esp          -- et vars dans la pile
    movl    $0, %eax            -- 3 entiers et un ptr?
    subl    %eax, %esp          -- Compilateur bête!
    movl    $2, -4(%ebp)        -- i=2
    movl    $1, -8(%ebp)        -- j=1
    movl    $0, -4(%ebp)        -- i=0

.L2:                            -- Condition de "for"
    cmpl    $9, -4(%ebp)        -- comparer i et 9
    jle .L5                     -- goto .L5 si <=
    jmp .L3                     -- goto .L3 sinon
```

```
.L5:                                -- Corps Boucle "for"
    movl    -4(%ebp), %edx          -- edx = i
    movl    %edx, %eax              -- eax = edx (eax=i)
    addl    %eax, %eax              -- eax = 2* eax (2*i)
    addl    %edx, %eax              -- eax = 3 * i
    addl    %eax, %eax              -- eax = 6*i ('*' par '+')
    movl    %eax, -8(%ebp)          -- j = eax
    leal    -4(%ebp), %eax          -- eax = i
    incl    (%eax)                  -- i++
    jmp .L2                         -- goto "condition FOR"
.L3:                                -- fin boucle "for"
    subl    $8, %esp
    pushl   -8(%ebp)                -- empiler j
    pushl   $.LC0                   -- empiler  "Resultat..."
    call    printf
    addl    $16, %esp
    subl    $12, %esp
    pushl   $0                      -- empiler 0
    call    exit                    -- retour à l'appelant
    .size   main, .-main
    .ident  "GCC: (GNU) 3.3 20030226 (prerelease) (SuSE Linux)"
```

The End of the Introduction!

Coming up : l'Analyse Lexicale!

Chapter 2

Analyse Lexicale

4 Analyse Lexicale

1. Rappels sur les langages Rationnels
2. Rôle d'un analyseur lexical
3. Exemples de Lex
4. Utilisation de Lex
5. D'autres Exemples

4.1 Rappels sur les langages rationnels

expressions rationnelles (regular expressions) : $\epsilon, +, ., *$

automates finis non déterministes : (Q,I,T,F)

automates finis déterministes : (Q,I,T,F) avec F fonctionnelle et I singleton

déterminisation : construction de l'ensemble puissance

minimalisation : construction de Moore

propriétés : lemme de l'étoile; fermeture par complément, intersection, union; décidabilité du langage vide, fini ou infini

4.2 Le rôle d'un analyseur lexical

Un analyseur lexical doit séparer un flot de caractères en une série de "tokens" (unités lexicales), chacune définie par une expression régulière.

Cela est légèrement différent du problème de la reconnaissance d'un langage rationnel :

- on recherche le plus long préfixe possible de l'entrée qui corresponde à une des définitions d'unités lexicales,
- on doit pouvoir retourner à la fois le préfixe reconnu et identifier le lexème
- certains analyseurs propose des extensions :
  - gestion des conditions et contextes
  - présence d'information supplémentaire sur les états finaux (le 1^{er} en cas d'ambiguïté) (cf. OcamlLex) qui maintient des variables supplémentaires de gestion

5 Utilisation de Lex

Un fichier source pour Lex a extension **.l** avec le format:

```
<définitions>
%%
<Règles>
%%
<Procédures utilisateurs>
```

5.1 Exemple-1 de Lex : test.l

Reconnaissance de nombres et d'identificateurs :

```
// lex test.l; gcc -o monprog lex.yy.c; ./monprog
%{
#include <ctype.h>
#include <stdio.h>
%}
%%
[a-zA-Z]+ {printf("Ident : %s\n",yytext);}
[0-9]+    {printf("Nbr : %s\n",yytext);}
[ \t\n]   ;      /* un séparateur */
.         {ECHO;  yyerror("Inconnu !",yytext);}
%%

int yywrap() {return 1;}
int yyerror(char* ch,char * text)
  {printf("Error %s: %s\n",ch,text);}
int main()
{yylex();
 printf( "fini \n");
}
```

Explications :

yylex() la fonction principale d'analyse lexicale (générée par *Lex*).

yytext le token courant (de type *char **)

yyin le fichier sur lequel *yylex* lira (de type *FILE **)

yywrap() (non fourni) l'action à entreprendre lorsque *yylex()* arrive à la fin de l'input. *yylex()* appelle *yywrap()* qui doit renvoyer 0 ou 1 (1 par défaut). Si 1, le programme termine et il n'y a plus d'input. Si 0, *yylex()* suppose que *yywrap()* a ouvert un autre fichier input pour lecture. *yylex()* continue donc à lire sur *yyin*. On peut ainsi lire tous les fichiers que l'on voudra passer en paramètre (à la fonction *main()*).

yyerror() procédure à définir pour gérer les erreurs.

ECHO écrit le token (contenu de *yytext*)

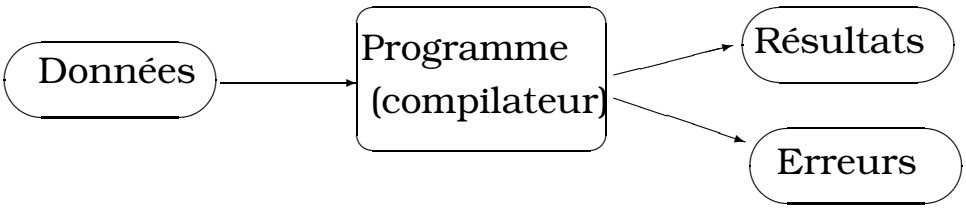
• Les commentaires sont délimités par /* ... */ sur une même ligne et seront reproduits dans le code généré. Les commentaires avec // (à la C++) sont permis.

6 Grammaires et Compilation (I)

☒ **Rappel de la problèmatique :**

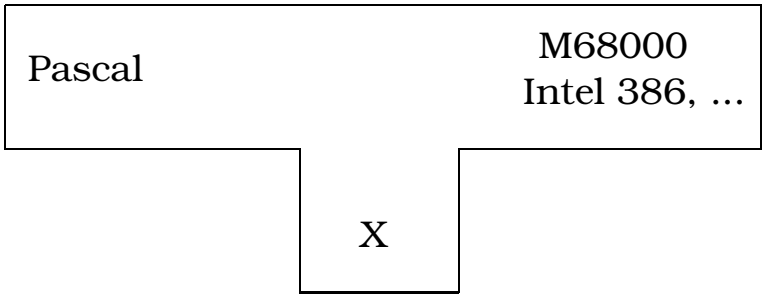
Un compilateur peu être conçu comme un traducteur du langages des données (flot de caractères) vers celui des résultats (langage machine, autre langage).

Les calculs sémantiques sont alors des traitements sur les langages intermédiaires.

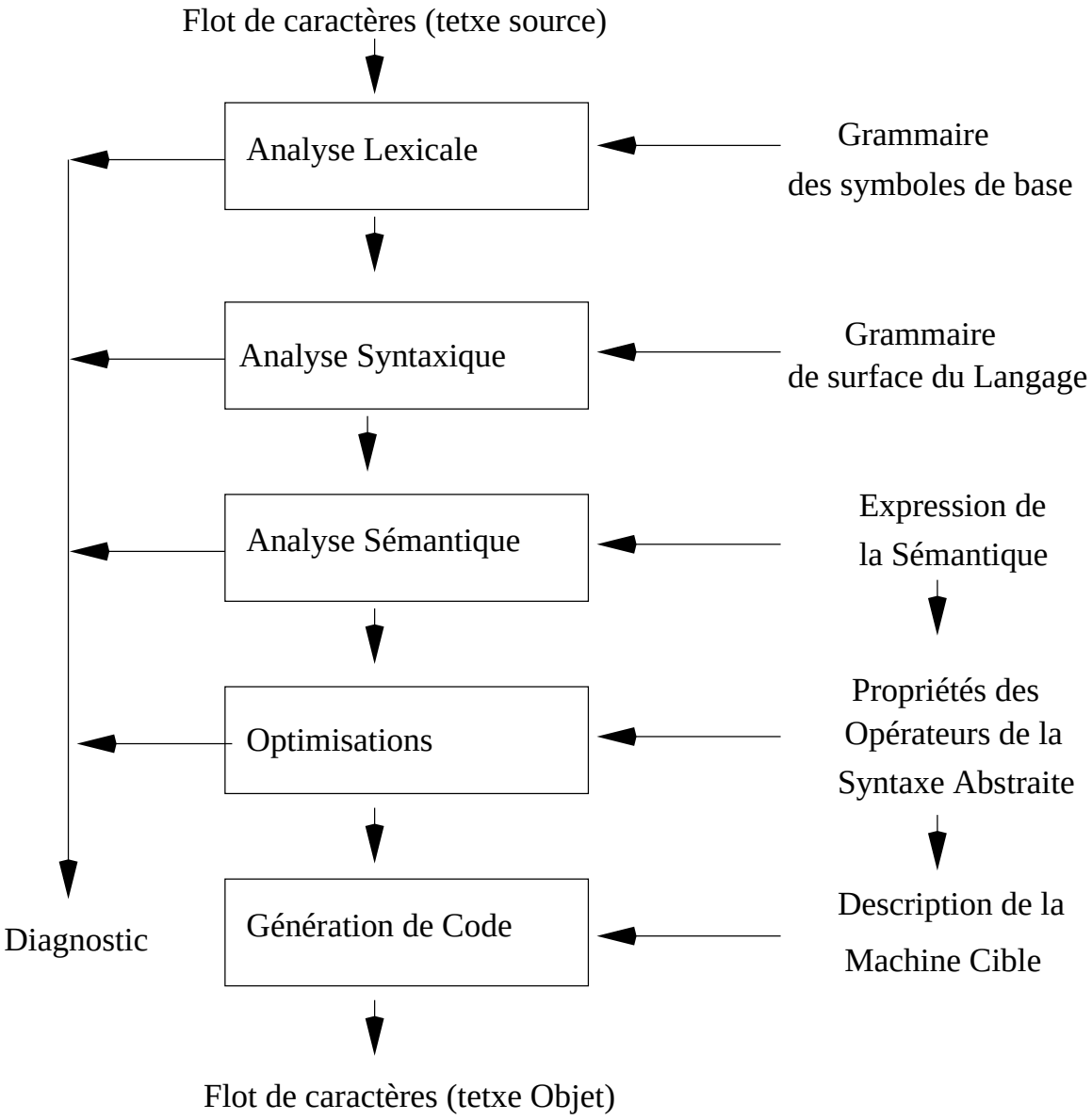


Dans cette vision, une grammaire est un outil de définition de structures et un support de calcul.

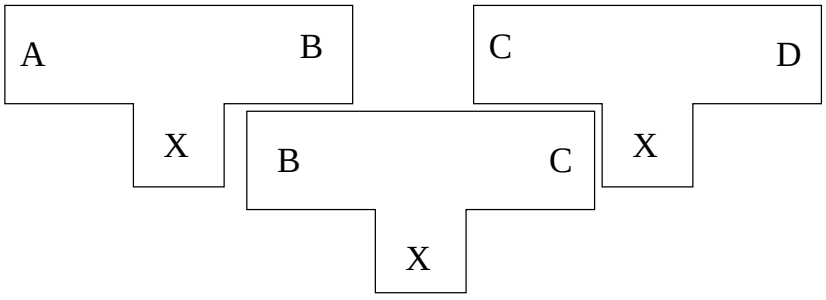
☒ **Les modèles de compilation :**



☒ Etapes de Compilation:



☒ Exemple d'architecture conceptuelle :



6.1 Formalismes Grammaticaux

Σ **:** vocabulaire ou alphabet

ensemble non vide de symboles

e.g. {a,b,c}, caractères ASCII, lettres de l'alphabet latin (français), ...

Σ^* **:** ensemble Σ muni de la concaténations (le point '.')

i.e. ensemble des séquances finies de symboles de Σ

La concaténation :

- $\forall x \in \Sigma, \quad x \in \Sigma^*$

- $\forall x \in \Sigma, \forall y \in \Sigma^*, \quad x.y \in \Sigma^*$

- ε est l'éléments neutre de la concaténation :

$$\forall x \in \Sigma^*, \quad x.\varepsilon = \varepsilon.x = x$$

- le '.' est un opérateur produit associatif :

$a^n = aa...a$ et $a^0 = \varepsilon$

$a,b \in \Sigma, ab \in \Sigma^*, aabab \in \Sigma^*$

Σ^* **:** ensemble de chaînes (mots) constructibles sur Σ, y compris la chaine vide ε

Σ^+ **=** $\Sigma^* - \{\varepsilon\}$

Un **langage (formel)** est une partie de Σ^*

Exemples :

- T={a,b} , $L_T = \{a^n b^m | n > m \geq 0\}$ ou $L_T = \{a^n b^n | n \geq 0\}$

- D={0,1} , $L_D = \{x \in D^* | \quad |x| \ mod \ 2 = 0\}$

- F={a,...,z, A, ..., Z} , $L_F = \{x \in F^* | \quad |x| = 8\}$

- C={0, ...,9},

$$L_C = \{xzy, \ x,y \in C^*, z \in C | xzy = yzx\}$$

NB : Pour le dernier exemple, peut-on déduire?:

$\{|x| = |y| \geq 0\}$

Cas de 0000 où $x = 0, z = 0, y = 00$

7 Opérations sur les langages formels

Soit les langages L_1,$L_2 \subseteq \Sigma^*$

⊠ Opérations ensemblistes :

1) Union de deux langages (somme)

$$L_1 \cup L_2 = L_1 + L_2 = \{x \in \Sigma^* | x \in L_1 \, ou \, x \in L_2\}$$

2) Intersection de langages

$$L_1 \cap L_2 = \{x \in \Sigma^* | x \in L_1 \, et \, x \in L_2\}$$

3) Complémentation ...

Exemple : $L = \{a^n b^n | n \geq 0\}$

$\bar{L} = \{a^n b^m | n \neq m\}$

⊠ **Opérations spécifiques :**

1 Produit de deux langages (concaténation) :

$L_1L_2 = \{xy|x \in L_1, y \in L_2\}$

$\Rightarrow$ opérateur associatif et non commutatif :

$$\Sigma\Sigma^* = \Sigma^* - \{\varepsilon\} = \Sigma^+$$

2) Fermeture d'un langage :

On utilise cette propriété pour prouver que certains langages ne sont pas de telle classe (voir plus loin).

$L^0 = \{\varepsilon\}$ et L le langage sur Σ^*

$L^n = LL^{n-1}, n \geq 1$

$L^* = \bigcup_{n\geq 0} L^n = \{\varepsilon\} \cup L^1 \cup L^2 \cup L^3 ...$

$L^+ = LL^* = L^*L$

Voir aussi les propriétés des grammaires et Langages.

8 Définition formelle des langages

8.1 Expressions Régulières (ER)

ER sur un alphabet Σ :

- $\emptyset$ est une ER décrivant $\emptyset$
- ε est une ER décrivant ε
- $a \in \Sigma$ est une ER décrivant {a}
- a **et** b **deux ER sur** Σ **(***soient* L_a,L_b**) :**

  $(a+b)$ est une ER décrivant $L_a \cup L_b$

  (ab) est une ER décrivant $L_a L_b$

  (a^*) est une ER décrivant L_a^*

N.B. $(a^+) = a(a^*)$ est un racourcie décrivant $L_a^+ = L_a L_a^*$

Exemples :

$l = \{A..Z,a..z\}, \quad c = \{0..9\}$

$\Rightarrow$ N.B. : on peut écrire $c = (0+1+..+9)$

c^+ est une ER décrivant les constantes entières

$l(l+c)^*$ est une ER décrivant les identificateurs

Priorités des opérateurs :

* < produit (concaténation) < somme (union)

Simplifications et calculs algébriques :

$r+s = s+r$	$r+(s+t) = (r+s)+t$
$(rs)t=r(st)$	$r(s+t)=rs+rt$
$(s+t)r = sr+tr$	$r^*=(r+\varepsilon)^*$
$r^{**} = r^*$	$r^+ = r\, r^*$ ...

$\Rightarrow$ Le formalisme ER est apart (des grammaires algébriques).

A ne pas confodre avec les grammaires régulières.

$\Rightarrow$ Il permet de décrire les langages régulièrs et finis.

L'outil Lex accepte les expressions régulières (étendues)

9 Utilisation de Lex: expressions rationnelles

Lex utilise une forme étendue des expressions régulières.

Un expression de base est de l'une des formes suivantes :

Formes de base :

x le caractère 'x'

. un caractère quelconque (mais pas eof)

«**EOF**» la fin d'input

[**xyz**] un de ces 3 caractères

[^ **aeiou**] aucun voyelle

[**a-z**] un caractère entre 'a' ... 'z'

[^ **a-z**] aucun minuscule

[**a-zA-Z**] une des lettres

[**0-9**] un chiffre

"une chaine" la chaîne de caractères

(Voir aussi la documentation ou le *man* de lex).

Lex : Les Formes Rationnelles :

r* 0 ou plus expression régulière r

r+ 1 ou plus expression régulière r

r? expression régulière r optionnelle (0 ou 1 fois)

r{2,5} de 2 à 5 occurrences de l'expression r

r{2,} de 2 à plus occurrences de l'expression r

r{2} exactement 2 occurrences de l'expression r

{nom} une expansion appelée 'nom' (voir ci-dessous)

x si x est l'un des caractères {*a,b,f,n,r,t,v*} alors ce sera l'interprétation ANSI de ce caractère (e.g. \n pour le retour chariot) sinon ce sera le caractère *x* (e.g * pour l'opérateur *, \\ pour \, etc.).

(r) même chose que r. On utilise les () pour la priorité.

rs l'expression régulière r suivie de l'expression s.

r | s l'expression régulière r ou de l'expression s.

r / s l'expression régulière r seulement si elle est suivie de s. Seule r est consommée en entrée.

^ r l'expression régulière r si elle est en début d'une ligne.

r$ l'expression régulière r si elle est à la fin d'une ligne.

Remarques :

• On peut définir des noms, par exemple :

```
DIGITS    [0-9]
```

et des les utiliser entre {} :

```
ID     [a-zA-Z_][a-zA-Z0-9_]*
DIGITS [0-9]+

%%
{DIGITS}  {printf("un nombre : %s", yytext);
           val=atoi(yytext));}
{ID}      {printf("un ident :"); ECHO;
           add_to_tab_sym(tab, yytext);}
```

• Les caractères spéciaux doivent être précédés de \, par exemple \" pour les guillemets.

• La précédence : '*' et '+' précèdent '?'.

• Les caractères spéciaux : {*a,b,f,n,r,t,v*}, \0 (nul ASCII), \0123 (code octal 123), \x2a (code hexa 2a),

• Il existe d'autres possibilités. Consulter les documents de Lex. *man lex* donnera également des explications utiles.

• Voir également les documents pour les actions que l'on peut mettre en partie droite d'une reconnaissance lexicale.

Les mots clefs :

yytext le texte du token analysé

yyleng la longueur du contenu du *yytext*

unput(c) remettre la chaîne yytext ou le caractère c dans l'entrée?

yylex() la fonction principale de lex. Elle peut être surchargée.

yymore() ajouter le prochain yytext à l'actuel au lieu de la remplacer

ECHO copie yytext sur output=affichage

REJECT procéder au second meilleur candidat, voir exemple ci-dessous

yyless(n) remettre sur l'input tout sauf les n premières caractères de yytext. Ils pourront ainsi être ré-analysés.

BEGIN suivi éventuellement d'un nom de *condition* démarre l'analyse avec cet état (voir la documentation). Les conditions permettent de guider l'analyse et de décider en fonction des états (par exemple, un *ident* dans un commentaire n'a pas le même rôle que hors commentaire).

yyrestart() pour recommencer l'analyse sur input (input peut avoir été changé à la fin d'un flux d'entrée

...

• Lex : Les actions :

D'une manière générale, les instruction du langage C peuvent être mis entre {} et être exécutées.

Exemples :

```
%%
[ \t]+    putchar(' '); //  {} non nécessaire ici

[ \t]+$   ; /* rien, ignorer espaces ou tabs */
            /* à la fin d'une ligne $ */

bidon   {faire_qq_chose(); REJECT;}
          /* continuer avec la règle suivante */
[^ \t\n]+ ++nb_words;  /* ici, on prend tout */
           /* sans REJECT, bidon ne sera pas */
           /*compté comme un mot*/

...
```

9.1 Un exemple complet

Reconnaissance d'entiers, réels, commentaires, etc. :

```
%{
#include <stdio.h>
#include <math.h>
int valeur; float reel;
int nb_lignes=0;
int nb_mots=0;
%}

ID     [a-zA-Z][a-zA-Z0-9]*
REEL   [0-9]+"."[0-9]+
ENTIER [0-9]+

/* une condition */
%s comment

%%

"/*"                    {printf("état = comment\n");
                         BEGIN(comment);}
<comment>[^*\n]*        ; /* ignorer si état=comment */
<comment>"*"+[^*/\n]*   ; /*  '*' non suivi par '/' */
<comment>[\n]           nb_lignes++;
<comment>"*"+"/"        {printf("fin etat comment\n");
                         BEGIN(INITIAL);}

[\n]     ++nb_lignes;
[ \t]    ;     /* ignorer espaces et tab */
{ID}     {printf("Ident : %s\n",yytext);
          nb_mots++}
// IL FAUT placer REEL avant ENTIER
{REEL}   {printf("reel : %s, %f\n",yytext,
            reel=atof(yytext));}
{ENTIER} {printf("entier : %s, %d\n",
           yytext,valeur=atoi(yytext));}

.        {ECHO;  yyerror("Inconnu !",yytext);}
```

%%

```
int yywrap() {return 1;}
int yyerror(char* ch,char * text)
      {printf("Error %s: %s\n",ch,text);}
int main()
 {yylex();
  printf( "nbr de lignes %d \n",nb_lignes);
  printf( "nbr de mots (idents) %d \n",nb_mots);
  printf( "le dernier entier lu  %d \n",valeur);
  printf( "le dernier reel lu  %f \n",reel);
  printf( "<--<Terminé>--> \n");
}
```

Remarques :

Si l'on trouve "·*", on met l'analyseur dans l'état "comment". Dans cet état, tout ce qui est lu est ignoré jusqu'à l'analyse de "*/"; ce qui remettra l'analyseur dans l'état normal (INITIAL).

Test : A l'exécution, on tape sur une même ligne

```
ceci est une ligne contenant 12 et 34.5
    etabcd123 et /* blab
```

Produit les résultats suivants :

```
Ident : ceci
Ident : est
Ident : une
Ident : ligne
Ident : contenant
entier : 12, 12
Ident : et
reel : 34.5, 34.500000
Ident : etabcd123
Ident : et
on rentre en etat comment
```

Puis on donne la chaîne (suivi de contrôle–D):

```
fin comment */ 12000 fini 23.1
```

qui produit les résultats suivants :

```
fin etat comment
entier : 12000, 12000
Ident : fini
reel : 23.1, 23.100000
nbr de lignes 2
nbr de mots (idents) 9
le dernier entier lu  12000
le dernier reel lu  23.100000
<--<Terminé>-->
```

Chapter 3

Analyse Syntaxique descendante

Plan du chapitre :

- Rappels sur les grammaires
- Définition d'analyse descendante
- Un exemple simple en *yacc*
- Définition et construction de Nullable, First, Follow
- Construction de la table LL(1)
- Transformations de grammaires:
  - levage de l'ambiguïté par la précédence d'opérateur
  - derecursivisation gauche
  - factorisation à gauche
- Un exemple complet

10 Les Grammaires (II)

• Une grammaire est un quadruplet G = $(\Sigma,V_N,\mathcal{S},\mathcal{P})$ où:

Σ est un alphabet fini de "terminaux",

V_N est un ensemble fini de symboles "non terminaux", avec $\Sigma \cap V_N = \emptyset$, $V = \Sigma \cup V_N$;

$\mathcal{S}$ est un non terminal distingué nommé le "symbole de départ" (start symbol)

$\mathcal{P}$ est une relation finie sur $V_N^+ \times \mathrm{V}^*$ qui définit les "règles" (de "production") de la grammaire.

Une règle $(\alpha,\beta) \in P$ s'écrit aussi $\alpha \rightarrow \beta$.

Exemple (Grammaire G):

$< Phrase >$ $\rightarrow < Groupe\_nominal >< Groupe\_verbal > .$

$< Groupe\_nominal >$ $\rightarrow < Determinant >< Nom. >$

$< Groupe\_verbal >$ $\rightarrow < Verbe\_nt > .$

$< Groupe\_verbal >$ $\rightarrow < Verbe\_t >< Groupe\_nominal > .$

$< Verbe >$ $\rightarrow$ $'mange'|$ $'danse'|...$

$< Nom >$ $\rightarrow$ $'chien'|$ $'soupe'|...$

10.1 Définitions :

On peut appliquer la règle $\alpha \rightarrow \beta$ à n'importe quelle séquence $x\alpha y$ pour obtenir $x\beta y$.

On écrit alors $x\alpha y \Rightarrow x\beta y$.

On dira que l'on a *réécrit* $x\alpha y$ en $x\beta y$ (à l'aide de la règle $\alpha \rightarrow \beta$).

On écrira $\omega \Rightarrow^* z$ s'il existe $\omega 1 \Rightarrow \omega 2... \Rightarrow \omega n$, $n \geq 1$,avec $\omega = \omega 1 \Rightarrow \omega 2... \Rightarrow \omega n = z$.

L'opérateur $\Rightarrow^*$ (appelée une *dérivation*) est la fermeture réflexo-transitive de l'opérateur $\Rightarrow$.

Le *langage engendré* par une grammaire G est

$$L(G) = \{\omega | S \Rightarrow^* \omega\}$$

Exemple : Soit la grammaire G des expressions arithmétiques constantes simplifiée où E est le symbole de départ :

$$\begin{aligned} E &\rightarrow T\ '+'\ E. \\ E &\rightarrow T. \\ T &\rightarrow F\ '*'\ T. \\ T &\rightarrow F. \\ F &\rightarrow 'x'. \\ F &\rightarrow '5'. \end{aligned}$$

Une dérivation pour $x * 5 + x \in L_G$:

$$\underline{E} \Rightarrow \underline{T} + E \Rightarrow \underline{F} * T + E \Rightarrow x * \underline{T} + E \Rightarrow x * \underline{F} + E$$

$$\Rightarrow x * 5 + \underline{E} \Rightarrow x * 5 + \underline{T} \Rightarrow x * 5 + \underline{F} \Rightarrow x * 5 + x$$

On a l'équivalence : $\boxed{(x * 5 + x \in L_G) \Leftrightarrow (E \Rightarrow^* x * 5 + x)}$

Remarque sur la signification de "LL1"

Le non terminal souligné à chaque pas de dérivation est celui utilisé pour cette étape de dérivation.

On a utilisé le non terminal le plus à gache (L : left) et on a appliqué les règles de gauche à droite (L: left).

Pour le rôle de "1", voir plus loin (tables LL).

10.2 Définitions (suite) :

• **Arbre de dérivation :** On peut représenter une dérivation $A \Rightarrow^* w$ (avec $w \in \Sigma^*$) comme un arbre ayant pour noeuds internes des non terminaux, pour feuilles des terminaux et des arcs reliant un noeud X a des fils W_1,...W_n s'il y a une règle $X \rightarrow X_1...X_n$.

Exemple :

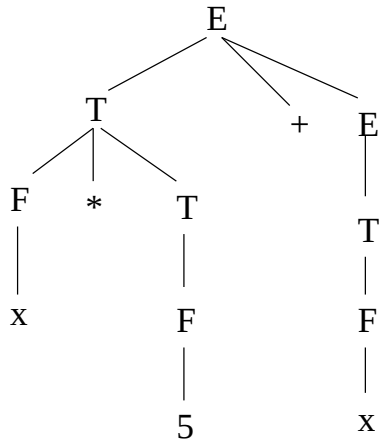
Pour l'expression $x * 5 + x \in L_G$, on a eu la dérivarion :

$E \Rightarrow T + E \Rightarrow F * T + E \Rightarrow x * T + E$

$\Rightarrow x * F + E \Rightarrow x * 5 + E \Rightarrow x * 5 + T$

$\Rightarrow x * 5 + F \Rightarrow x * 5 + x \Rightarrow$

L'arbre de dérivation :



• **Grammaire ambiguë :** Une grammaire G est ambiguë s'il existe un mot $w \in L(G)$ avec au moins deux arbres de dérivation différents.
Exemple: le mot $a + b + a$ dans la grammaire

$$E \rightarrow E + E.$$
$$E \rightarrow a.$$
$$E \rightarrow b.$$

De façon équivalente, une grammaire est ambiguë si elle permette deux dérivations $gauches$ différentes pour un même mot w.

• **Backus Naur Form (BNF)** La "forme normale de Backus" est une extension du formalisme des grammaires hors contextes qui n'ajoute pas de pouvoir expressif (on ne définit pas une classe plus large de langages), mais qui permet des définitions plus concises.

A la structure des règles on ajoute les notations suivantes:

l'opérateur étoile de Kleene:

on peut écrire $A \rightarrow \alpha\beta^*\gamma$

à la place de :

$A \rightarrow \alpha B\gamma$

$B \rightarrow \beta B$

$B \rightarrow \epsilon$.

optionel :

on peut écrire $A \rightarrow \alpha[\beta]\gamma$

à la place de :

$A \rightarrow \alpha B\gamma$

$B \rightarrow \beta$

$B \rightarrow \epsilon$.

choix :

on peut écrire $A \rightarrow \alpha_1|\alpha_2...|\alpha_n$

à la place de :

$A \rightarrow \alpha_1$

$A \rightarrow \alpha_1$

...

$A \rightarrow \alpha_n$

Nota Bene : en s'inspirant des expressions régulières (cf. Lex), parfois :

β^* est noté $\{\beta\}$ ou encore $[\beta]^*$

$\beta(\beta)^*$ est noté $(\beta)^+$ ou encore $[\beta]^+$.

Attention a ne pas mélanger les symboles BNF avec les terminaux de Σ.Pour éviter ce cas, les éléments $s \in \Sigma$ sont notés par $'s'$ (sauf s'il n'y a apas d'ambiguïté).

10.3 Extension du BNF (EBNF = Extended BNF)

$X \to AC.$ $X \to BC.$ **donne** $X \to (A\ ;\ B)\, C.$

$A \to BC.$ $A \to B.$ **donne** $A \to B(C\ ;\ \varepsilon)$

$B \to CB.$ $B \to \varepsilon.$ **donne** $B \to \{C\}$ **ou** $(C)^*$ **ou** $[C]^*$

$C \to aC.$ $C \to a.$ **donne** $B \to (a)^+$ **ou** $[a]^+$

10.4 Grammaire du BNF

$$
\begin{array}{lcl}
\mathit{Grammaire} & \rightarrow & \mathit{Regles} \\
\mathit{Regles} & \rightarrow & \mathit{Regle}\ \ \mathit{Regles} \mid \varepsilon. \\
\mathit{Regle} & \rightarrow & \mathit{SymboleDeV_N}\text{ ":: "}\ \mathit{Alternants}\text{"."}. \\
\mathit{Alternants} & \rightarrow & \mathit{Sequence}\ \mathit{Sommes}\ . \\
\mathit{Sequence} & \rightarrow & \mathit{Element}\ \ \mathit{Produit}\ . \\
\mathit{Sommes} & \rightarrow & \text{";"}\ \mathit{Alternants} \mid \varepsilon. \\
\mathit{Produit} & \rightarrow & \text{","}\ \mathit{Sequence} \mid \varepsilon. \\
\mathit{Element} & \rightarrow & \text{"("}\ \mathit{Alternants}\text{")"}. \\
\mathit{Element} & \rightarrow & \text{"\{"}\ \mathit{Alternants}\text{"\}"}. \\
\mathit{Element} & \rightarrow & \text{"["}\ \mathit{Alternants}\text{"]"}. \\
\mathit{Element} & \rightarrow & \text{"}\varepsilon\text{"}. \\
\mathit{Element} & \rightarrow & \mathit{SymboleDe\Sigma} \mid \mathit{SymboleDeV_N}. \\
\mathit{SymboleDeV_N} & \rightarrow & \text{élément de } V_N. \\
\mathit{SymboleDe\Sigma} & \rightarrow & \text{élément de } \Sigma.
\end{array}
$$

11 Classification de Chomsky

Restriction sur la forme des productions.

$\Rightarrow$ Pouvoir de description, propriétés formelles.

• On classifie les grammaires selon la forme de $\mathcal{P}$:

type 4 (finie) :

règle de la forme $A \rightarrow b$, $A \in V_N$,$b \in \Sigma$.

type 3 (rationnelle, linéaire, régulier, Kleen) :

règle de la forme $A \rightarrow bB$ ou $A \rightarrow b$, A,$B \in V_N$,$b \in \Sigma$.

NB : on distingue linéaire à *gauche* ou à *droite*.

- linéaire à gauche (cf. ci-dessus)

- linéaire à droite : règle de la forme

$A \rightarrow Bb$ ou $A \rightarrow b$, A,$B \in V_N$,$b \in \Sigma$.

type 2 (hors contexte) :

règle de la forme $A \rightarrow \beta$, $A \in V_N$,$\beta \in V^*$

type 1 (contextuelle):

pour toute règle $\alpha \to \beta$, $|\alpha| \leqslant |\beta|$, $\alpha,\beta \in V^+$

type 0 :

$\mathcal{P}$ quelconque ($\alpha \to \beta$, $\alpha \in V^+$,$\beta \in V^*$)

Pour les langages de programmation, on s'intéresse aux grammaires de type 2 (hors contexte = CFG).

11.1 Exemples

Notation : distinction terminaux et non terminaux,

on note $A \rightarrow C\,'a'\,B$ (terminaux entre '')

ou $A \rightarrow < C > \ a \ < B >$ (non terminaux entre <>)

ou même $A \rightarrow < C > \ 'a' \ < B >$

• **type 4 : langages finis :**

amphi → *'A1'* | *'A2'* | *'A3'* | *'A201'* | *'A202'* .

• **type 3 : langages réguliers :**

Exemple : identificateur avec la lettre 'a' et le chiffre '0' :

$< identificateur > \rightarrow \ 'a' < chiffres\_ou\_lettres >.$

$< chiffres\_ou\_lettres > \rightarrow \ 'a' < chiffres\_ou\_lettres >.$

$< chiffres\_ou\_lettres > \rightarrow \ '0' < chiffres\_ou\_lettres >.$

$< chiffres\_ou\_lettres > \rightarrow \ '0'$.

$< chiffres\_ou\_lettres > \rightarrow \ 'a'.$

• type 2 : langages Hors Contextes:

• Exemple-1 : L=$\{a^n b^n | n \geq 1\}$:

$S \quad \rightarrow \ 'a'\ E\ 'b'.$

$E \quad \rightarrow \ 'a'\ E\ 'b'.$

$E \quad \rightarrow \ \varepsilon.$

• Exemple-2 : L=$\{a^n b^m | n,m \geq 0\}$:

$S \quad \rightarrow \ A\ B.$

$A \quad \rightarrow \ 'a'\ A.$

$A \quad \rightarrow \varepsilon.$

$B \quad \rightarrow \ 'b'\ B.$

$B \quad \rightarrow \varepsilon.$

• type 1 : langages Contextuels

Exemple : L= $\{a^n b^n c^n | n \geq 1\}$ avec $\Sigma = \{a,b,c\}$:

(1) $S \rightarrow a\ S\ B\ C.$

(2) $S \rightarrow a\ B\ C.$

(3) $C\ B \rightarrow B\ C.$

(4) $a\ B \rightarrow a\ b.$

(5) $b\ B \rightarrow b\ b.$

(6) $b\ C \rightarrow b\ c.$

(7) $c\ C \rightarrow c\ c.$

Un exemple de dérivation de $a^2b^2c^2 \in L$:

$\underline{S} \Rightarrow_1 a\underline{S}BC \Rightarrow_2 a\ \underline{a\ B}CBC \Rightarrow_4 a\ a\ b\ \underline{CB}C$

$\Rightarrow_3 a\ a\ \underline{b\ B}CC \Rightarrow_5 a\ a\ b\ \underline{b\ C}C$

$\Rightarrow_6 a\ a\ b\ b\ \underline{c\ C} \Rightarrow_7 a\ a\ b\ b\ c\ c$

• type 0 : langages formels

• Aucune restriction sur la taille de la partie gauche (ou droite) de $\rightarrow$.

Formalisme non exploitable !

Exemple (semblable au type 1) :

L= $\{a^i | i = 2^n, n \geq 1\}$ avec $\Sigma = \{a\}$:

$S \rightarrow a\ C\ aB.$

$C\ a \rightarrow a\ a\ C.$

$C\ B \rightarrow D\ B.$

$C\ B \rightarrow E.$

$a\ D \rightarrow D\ a.$

$A\ D \rightarrow A\ C.$

$a\ E \rightarrow E\ a.$

$A\ E \rightarrow \varepsilon.$

Remarque : les grammaire de type 0 sont équivalentes à la machine de Turing.

11.2 Quelques propriétés des grammaires

Pumping Lemme pour les CFL

Soit L un langage hors contexte. Il existe une constante n dépendante seulement de L tel que pour tout mot $z \in L$, $|z| \geq n$, il existe une décomposition $z = uvwxy$ avec $|vwx| \leq n$, $|vx| \geq 1$ et pour tout $i \geq 0$, uv^iwx^iy est dans L.

Homomorphisme sur les CFL

Les CFL sont clos sous l'homomorphisme (et sous l'homomorphisme inverse).

C'est à dire, pour $h : \Sigma \rightarrow \Delta$ (h appliqué à $a \in L$ génère une chaîne $h(a) \in \Delta$) ;

et pour un langages Hors Contexte L, $L' = h(L)$ est CFL (idem pour $L'' = h^{-1}(L)$).

i.e., si $a_1a_2...a_n \in L$ alors $h(a_1a_2...a_n) \in h(L)$

Avec :

- $h(a) = a', a \in \Sigma, a' \in \Delta$

- $h(a_1a_2...a_n) = h(a_1)h(a_2)...h(a_n)$

11.3 La fermeture et les CFL (langages Hors Contextes)

- Les CFL sont clos sous la substitution.

- Aussi, les CFL sont clos sous l'union, la concaténation et la fermeture de Kleene (i.e. le résultat est un CFL)

- Mais les CFL ne sont pas clos sous l'intersection. Par contre, CFL ∩ ER = CFL.

Utilisation:

On utilise ces lemmes pour prouver que certains langages ne sont pas CFL (+ lemme de Ogden).

⊡ $\{wcw|w \in (a|b)^*\}$ n'est pas hors contexte.

⊡ $\{a^n b^m c^n d^m | n,m \geqslant 1\}$ n'est pas hors contexte.

⊡ $\{a^n b^n c^n | n \geqslant 0\}$ n'est pas hors contexte.

En utilisant ces propréiéts on démontre par exemple que :

- déclaration avant utilisation d'identificateurs;

- correspondance entre paramètres formels et actuels d'une procédure

- indication des mots soulignés (ancienne technique)

⊡ **Un exemple complet :**

L = $\{ww|w \in (a|b)^*\}$ n'est pas hors contexte (proche de l'exemple ci-dessus)

C'est à dire, L contient des mots (feuilles de l'arbre de dérivation) dont le début et la fin sont identiques.

Preuve : supposons que L est CFL.

Dans ce cas, $L_1 = L \cap a^+b^+a^+b^+$ est aussi CFL (intersection CF et ER).

Mais $L1 = \{a^ib^ia^ib^i\}$ n'est pas CFL (preuve à l'aide de Pumping lemme).

Donc, L n'est pas CFL.

Nota Bene : Si l'on ne veut pas utiliser le Pumping lemme, on pose L_2 une réduction de L1, $L2 = \{a^ib^jc^id^j|i,j \geq 1\}$.

On sait que si L_1 est CFL, alors L_2 l'est aussi. Or, si h est un homomorphisme tel que $h(a) = h(c) = a$ et $h(b) = h(d) = b$, on a $L_2 = h^{-1}(L_1) \cap a^*b^*c^*d^*$.

Or on sait que $L2$ n'est pas CFL, donc L_1 ne l'est pas non plus.

12 Autres Formalismes

12.1 Automates d'états finis

Les automates d'états finis (AEF) sont équivalents aux langages (et expressions) régulières.

Un AEF se définit par $A = (V,Q,Q_0,F,\delta)$ où :

Σ vocabulaire

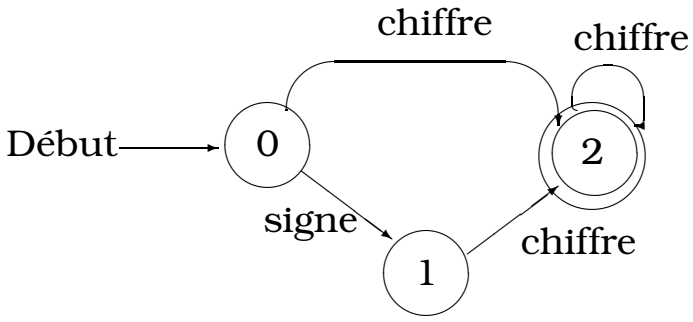
Q ensemble d'états

Q_0 état initial,$Q_0 \in Q$

F état finaux,$F \subseteq Q$

δ fonction de transition, $\delta : \Sigma \times Q \rightarrow Q$

Exemple : structure d'un entier signé :



Remarques : *Début* désigne l'état initial, un double cercle désigne l'état final, *chiffre* et *signe* ont leur signification habituelle. Ici, le vocabulaire={+,-,0,...,9}.

Les transitions dans un AEF peuvent être enrichies d'une *action sémantique* qui aura lieu si la transition est appliquée.

Dans l'exemple des entiers signés, on pourra écrire pour chaque transition:

init : *int sg= -1, val=0;*
$\delta(0,signe) = 1$ et S_{01}: if ($signe == '-'$) $sg = -1$;
$\delta(0,chiffre) = 2$ et S_{02}: $val = atoi(chiffre)$; %chiffre reconnu
$\delta(1,chiffre) = 2$ et S_{12}: $val = atoi(chiffre)$;
$\delta(2,chiffre) = 2$ et S_{22}: $val = val * 10 + atoi(chiffre)$;
S_f: $val = sg * val$;

L'action finale S_f sera appliquée à la fin de la reconnaissance (sortie de l'automate).

La compilation d'un AEF en un analysuer est assez simple et directe. Une table $Q \times \Sigma$ permet de décider de la transition à appliquer.

Les AEF permettent de mieux visualiser les expressions (grammaires) régulières.

Problème des AEF :

certains peuvent être non déterministes.

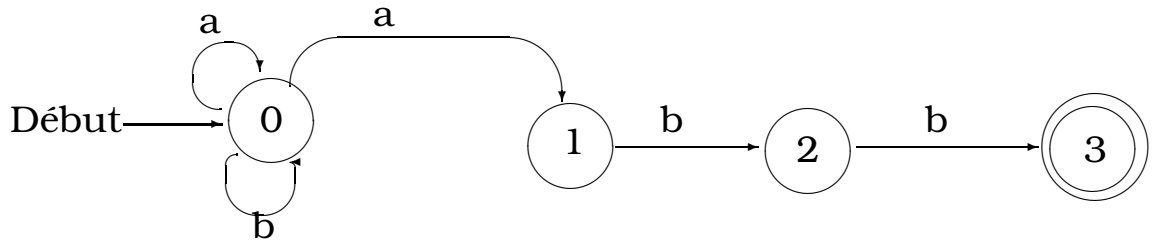
C'est à dire, $\exists q,q',q'' \in Q,\alpha \in \Sigma$ avec :

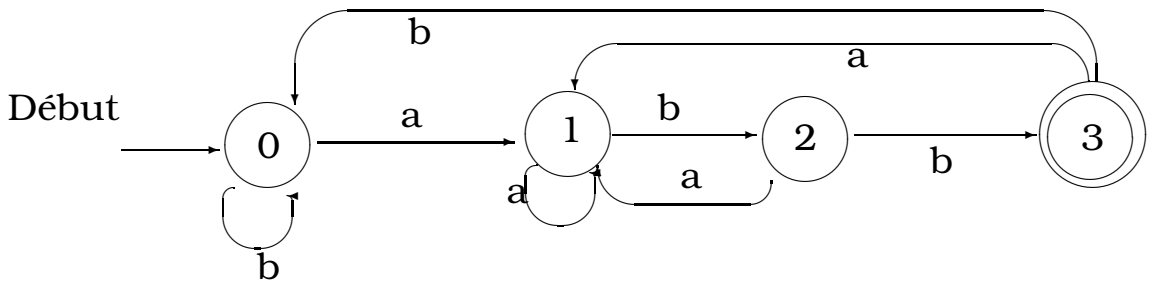
$$\delta(q,\alpha) = q' \qquad et \qquad \delta(q,\alpha) = q''$$

Dans ce cas, lors de la reconnaissance du symbole α, on ne peut pas décider (comme cela serait le cas dans un automate **déterministe** comme celui de l'exemple) quel transition choisir.

Exemple :

L'exemple suivant montre un AEF non déterministe reconnaissant le langage $(a+b)^*abb$ ainsi que l'AEF déterministe équivalent (reconnaissant le même langage).





Solutions aux AEF non déterminsistes:

- Retour arrière pendant l'analyse (avec problème des actions sémantiques qu'il faudra pouvoir défaire);

- Rransformation de l'AEF en un automate déterministe.

Equivalence AEF et grammaires régulières:

Une transition de la forme $\delta(A,\alpha) = B$ devient $A \rightarrow \alpha\, B$.

Aussi, $A \rightarrow \alpha$ représente $\delta(A,\alpha) = q_f$.

Attention : une règle d'une grammaire régulière de la forme $A \rightarrow \alpha$ ne débouche pas toujours sur un état final (mais l'inverse est vrai). Par exemple, dans :

$$S \rightarrow Aa$$
$$A \rightarrow Ab$$
$$A \rightarrow b$$

on voit bien que la règle $A \rightarrow b$ ne doit pas aboutir à un état final.

Autres exemples d'AEF : métro, chèvre et loup, ...

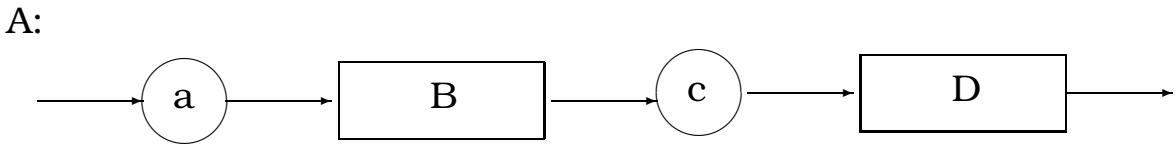
12.2 Cartes Syntaxiques

Les cartes syntaxiques permettent de spécifier une grammaire (Hors Contexte ou régulière) sous forme graphique. Ce qui permet une visualisation plus simple de la grammaire (mais occupe plus d'espace et sera difficile à traiter automatiquement car la grammaire est sous une forme non textuelle !).

Dans une carte syntaxique, une règle de la forme :

$$A \rightarrow a\ B\ c\ D$$

deviendra :

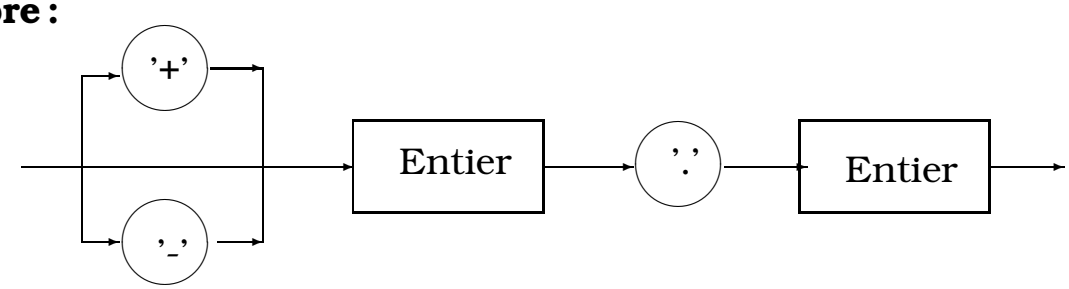


Comme on peut le remarquer, les éléments terminaux (éléments de Σ) s'inscrivent dans un cercle (ou une ellipse) alors que les non terminaux sont inscrits dans un rectangle. Le sens gauche-droite est donné par les flêches.

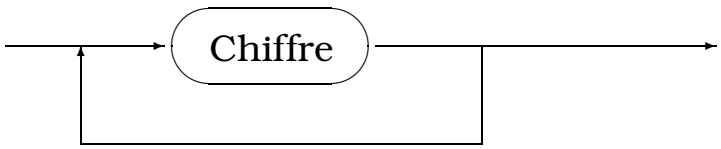
Un exemple :

La grammaire des nombres réels signés :

Nombre :



Entier :



Dans la 2e règle, la syntaxte d'un *Entier* traduit une règle telle que :

$$Entier \rightarrow Un\_Chiffre \quad Chiffres.$$
$$Chiffres \rightarrow Un\_Chiffre \quad Chiffres \mid \varepsilon.$$

On peut exprimer la même chose par l'expression rgulière $(Chiffre)^+$ ou par la règle BNF :

$$Entier \rightarrow Chiffre \quad \{Chiffre\}.$$

Les cartes syntaxiques permettent d'exprimer les mêmes possibilités que le formalisme EBNF.

13 Analyse Descendante

La Problèmatique de l'analyse : étant donné une grammaire G et une chaîne $w \in \Sigma^*$, construire un algorithme permettant de décider si w appartient ou non a $L(G)$.

De plus, dans la positive, on demande de construire un arbre de dérivation pour w dans G.

Certains langages permettent de résoudre ce problème a l'aide d'une méthode simple dite *d'analyse descendante*.

Cette méthode cherche a construire une *dérivation gauche* directement a partir du symbole initial, ou de façon équivalente, cherche a construire un arbre de dérivation à partir de la racine avec une exploration en *profondeur d'abord* et *de gauche a droite*. On dira alors que l'on procède à une **analyse LL**.

Pour cela, l'analyseur doit pouvoir décider, en s'aidant éventuellement des prochains k symboles en entrée, quelle prochaine production utiliser dans la dérivation gauche que l'on construit.

La difficulté : étant donné le symbole terminal en entrée et les règles de production, il faudra de la chance pour que la méthode fonctionne.

Dans l'absolu, il faut tenter chaque règle pour appliquer celle qui "marche" mais toute grammaire ambiguë posera des problèmes (retour arrière nécessaire).

Mais si l'on réussit, alors le langage de la grammaire est dans la classe *LL(k)* (langages analysables en parcourant l'entrée, de gauche (L) à droite et en construisant une dérivation gauche (L), en réécrivant α en β à l'aide d'une règle de la fome $\alpha \rightarrow \beta$).

La valeur de k désigne le nombre de symboles (lexèmes) qu'il faudra examiner pour décider de la règle à appliquer.

NB : même dans les méthodes naïves, on se fait guider par le symbole de départ, les terminaux dans les règles et les k lexèmes en entrée.

13.1 Exemple : analyseur LL1 basic

Prenons un exemple de grammaire *LL(1)* et un analyseur construit a la main (# dénote la fin d'un S) :

$$S \rightarrow E\#$$
$$E \rightarrow INT\ E'$$
$$E' \rightarrow +\ INT\ E' \mid \epsilon$$

Admettons (cf. Lex) qu'un analyseur lexicale nous donne les tokens *PLUS* (pour +) , *INT* (entier), Fin_S (pour #) et EOF (fin de fichier) (cf. voir TD0 et TD1).

Protocole à suivre pour l'écriture de l'analyseur :

- Ecrire une fonction par symbole non terminal (ici, {S, E, E'})

- Pour un terminal α en partie droite d'une règle, tester la présence de ce symbole sur le flot d'entrée (*if token*==α); si succès alors lire le prochain symbole (appel à yylex()), échec si token $\neq \alpha$

- En présence d'un non terminal X, appeler la fonction $X()$.

```
// .... La partie lexical ....

token t;
bool descente()
{t=get_token();
 if (S())
    {if (t  !=  EOF)
        {perror("EOF attendu"); return false;}
     else {printf("succès"); return true;}
    }
 else return false;
}
bool S()
{if (E())
    {if (t  ==  Fin_S)  // le '#'  final
       {t=get_token();return true;}
    }
 else return false;
}
bool E()
{if (t == INT)
    {t=get_token();
     return E_prim();
    }
 else return false;
}
bool E_prim()
{if (t == PLUS)
    {t=get_token();
     if (t == INT)
        {t=get_token();
         return E_prim();
        }
     else {perror("INT attendu"); return false;}
    }
 else return true;
}
```

13.2 Table LL(1)

Lors d'une analyse LL(1), on peut choisir la règle de production en regardant le prochain token (k=1 dans *LL(k)*). Pour réaliser un analyseur LL(1), il faut remplir une table de la forme :

	t_1	...	t_k	...	t_m
X_1		...	...	...	
...	...		...	...	
X_i	...		$X_i \to \gamma$	...	
...	...		...	...	
X_n	...		...	...	

On mettra la règle $X_i \to \gamma$ dans la case (X_i, t_k) si la présence de t_k et la règle X_i (le contexte) peuvent faire appliquer $X_i \to \gamma$.

Remarque : l'exemple de calcul de la table donné plus loin détaille l'intérêt de cette construction dont l'objectif est d'éviter une analyse non informée (aveugle)..

13.3 ε-règles, Premier, Suivant, Directeur

Pour la construction de la table LL1, on a besoin de calculer trois ensembles pour les symboles non terminaux (éléments de V_N de la grammaire).

ε-règles(X) (Nullable rules) : vrai ssi X peut produire la chaîne vide (règle de la forme $X \rightarrow^* \varepsilon$, i.e. directement ou indirectement)

Premier(X) (First) : ensemble de symboles terminaux qui peuvent paraître en première position dans une chaîne γ dérivable à partir de X.

La présence de l'un de ces symboles permettra d'activer la règle $X \rightarrow \gamma$.

Suivant(X) (Follow) : ensemble de symboles terminaux t qui peuvent paraître immédiatement après X dans une dérivation.

C'est à dire, il existe une dérivation contenant la chaîne Xt (ce qui peut aussi arriver si la dérivation contient la chaîne $XYZt$ où Y et Z sont des ε-règles).

Directeur(X) : déclencheur de règle (voir plus loin).

13.4 Définitions pour la table LL1

Formulation de la propriété ε-règle(X) et des ensembles *Premier(X), Suivant(X), Directeur(X)*, $X \in V_N$.

ε-règle (Nullable): $X \vdash^* \varepsilon$ Si $X = \varepsilon$ ou $X \rightarrow^* \varepsilon$

et pour une séquences de symboles : ε-règle$(X_1...X_n)$ est vrai ssi ε-règle(X_i) est vrai pour $1 \leq i \leq n$;

Premier(X) (First) :

– Si $X \in \Sigma$ alors Premier(X) = $\{X\}$ (ensembliste)

– Si $X = \varepsilon$ alors Premier(X) = $\emptyset$

– Si $X \in V_N$,la règle $X \rightarrow Y \in \mathcal{P}$, $Y \in V^*$ alors

Premier(X) = Premier(Y)

– Si $X \in V^*$, $X = \alpha_1,\alpha_2,...,\alpha_n$ alors

– Si $\alpha_1 \vdash^* \epsilon$ (Nullable) alors

Premier(X) = Premier(α_1) $\cup$ Premier($\alpha_2,...,\alpha_n$)

– Sinon Premier(X) = Premier(α_1)

– Si $X \in V^*$,$X = \alpha_1|\alpha_2|...|\alpha_n$ alors

Premier(X) = Premier(α_1) $\cup$ Premier($\alpha_2|...|\alpha_n$)

– Si X=$\{\beta\}$ (itération) alors Premier(X) = Premier(β)

Suivant(X) (Follow) :

Soit $\mathcal{R}$ (une partie de $\mathcal{P}$) l'ensemble de règles où $X \in V_N$ figure à droite : $\forall r \in \mathcal{R}$, $r : A \rightarrow \alpha\mathbf{X}\beta$:

- Si $\beta \vdash^* \varepsilon$ (i.e. $\beta = \varepsilon$ ou $\beta \rightarrow^* \varepsilon$) alors

  Suivant(X) = Premier(β) $\cup$ Suivant(A)

- Sinon Suivant(X) = Premier(β)

NB : on peut facilement étendre cette définition à n'importe quel symbole de V_N figurant à gauche d'une règle.

Directeur(X), $X \in V_N$

On considère les cas de figures similaires au *Premier(X)*.

A titre d'exemple :

- Soit la règle $X \rightarrow \alpha_1,...,\alpha_n$
  - Si $\alpha_1 \vdash^* \varepsilon$ alors $Directeur(X) = Premier(\alpha_1) \cup Suivant(\alpha_1)$
  - Sinon $Directeur(X) = Premier(\alpha_1)$
- Soit la règle $X \rightarrow \alpha_1|...|\alpha_n$

  $Directeur(X) = Directeur(\alpha_1) \cup Directeur(\alpha_2|...|\alpha_n)$

N.B. : dans une règle $A \rightarrow X\alpha$, *Director(A)* dépendra du *Suivant(X)* si *nullable(X)*.

13.5 Un Exemple

Considérons la grammaire G={Σ, V_N, S,$\mathcal{P}$) avec :
Σ={a,b,d,e,f}, V_N={S,A,B}, S=S et $\mathcal{P}$ donné ci-dessous :

$$S \rightarrow Ad \mid B\{f\}.$$
$$A \rightarrow aA \mid e.$$
$$B \rightarrow \varepsilon \mid bB.$$

On peut facilement constater les problèmes posés par une analyse descendante habituelle utilisée dans les exemples précédents (sans la table LL1). Par exemple, dans la règle $S \rightarrow Ad|B\{f\}$. l'analyse commencera par essayer l'alternative Ad et rien n'indique que l'appel de A pourra réussir et que l'on trouvera ensuite le non-terminal d sur le flot d'entrée.

Du fait de cet essai aveugle (non informé), il faut s'interdire tout effet de bord (modification de l'environnement, tables de symboles, etc.) car l'essai de A peut échouer.

Si l'alternative Ad ne réussit pas, l'on fera de même avec $B\{f\}$., ce qui posera les même problèmes. Et si l'on avait n alternatives et si les $n-1$ premières ne devaient pas aboutir !? De plus, le processus étant non déterministe, il faudra mettre en place un mécanisme de retour arrière élaboré.

Il est évident qu'un processus déterministe (irrévocable) sera bien plus efficace. Pour cela, nous avons besoin de nous engager dans une branche de l'arbre de dérivation avec les informations (la présence de terminaux) nous permettant d'éviter des tentatives infructueuses (échec suivi de retour arrière).

Une autre source de problèmes est l'analyse de règles *nullables* comme dans $B \rightarrow \varepsilon \mid bB.$ qui, dans le cas d'une analyse sans table LL(k), réussira si les autres alternatives de la règle échouent. La table LL(k) donnera également des indications dans le cas de ces règles : si les autres alternatives ne peuvent pas être essayées (décision *a priori*), la présence de certains terminaux permet d'accepter l'alternative ε (voir dans la table de cet exemple).

Les calcules :

Pour procéder aux calculs, on considère en premier lieu les parties droites des règles.

On a :

• Premier($\varepsilon \mid bB$)=Premier(ε) $\cup$ Premier(bB)={b}

• Premier($aA \mid e$)=Premier(aA) $\cup$ Premier(e)={a,e}

• Premier($Ad \mid B\{f\}$)=Premier(Ad) $\cup$ Premier($B\{f\}$)

=Premier(A) $\cup$ Premier(B) $\cup$ Premier({f})

=Premier(A) $\cup$ Premier(B) $\cup$ {f}

On a :

• par la règle $A \rightarrow aA \mid e.$, Premier(A)=Premier($aA \mid e$)={a,e}

• par la règle $B \rightarrow \varepsilon \mid bB.$, Premier(B)= Premier($\varepsilon \mid bB$) = {b}

• Premier({f}) = {f}

Remarque : **B** et par conséquent **S** sont nullables;

Pour récapituler : les ensembles *Premier(X)* :

Expression γ	γ'=Premier(γ)
ε	$\emptyset$
bB	$\{b\}$
aA	$\{a\}$
e	$\{e\}$
$\{f\}$	{f}
A $A \rightarrow aA \mid e.$	$\{a,e\}$
B $B \rightarrow \varepsilon \mid bB.$	$\{b\}$
Ad	$\{a,e\}$
$B\{f\}$	$\{b,f\}$
S $S \rightarrow Ad \mid B\{f\}.$	$\{a,e,b,f\}$

Remarque : le calcul du *Premier(S)* est peu utile car toute dérivation commencera par ce symbole particuler (de départ).

On ajoutera la règle $SS \rightarrow S\#.$ pour compléter la grammaire. L'ajout de # (qui joue le même rôle que le point-virgule dans les langages de programmation) permet d'éviter d'accepter des constructions (*statement*) partiellement correctes (e.g. dans *(a+5))))*) alors que l'ensemble n'est pas une expression correcte. De plus, on pourra donner plusieurs constructions sur la même ligne.

Calcul de X" = Suivant(X) pour X figurant à droite d'une règle.

Rappel : ajout de la règle $SS \rightarrow S\#$.

Expression (γ)	Figurant à dte. de	γ"=Suivant(γ)	γ" final
$Ad \mid B\{f\}.$	$S \rightarrow Ad \mid B\{f\}.$	S"	$\{\#\}$
$\varepsilon \mid bB$	$B \rightarrow \varepsilon \mid bB.$	B"	$\{f,\#\}$
S	$SS \rightarrow S\#.$	$\{\#\}$	$\{\#\}$
B	$S \rightarrow B\{f\}$ $B \rightarrow bB$	$\{f\} \cup S$" B"	$\{f,\#\}$ $\{f,\#\}$

Remarques :

• S (dans la règle $S \rightarrow Ad \mid B\{f\}.$) est nullable d'où S" en colonne $Suivant(\gamma)$ dans la 1^e ligne.

• B (dans la règle $B \rightarrow \varepsilon \mid bB.$) est nullable d'où B" en 2e ligne.

Tableau final

d(X) = Directeur(X) pour X figurant à droite d'une règle.

Les ensembles $X' = Premier(X)$ et $X" = Suivant(X)$ sont rappelés ici.

Remarque : le mot **inutile** figure dans les cases lorsque l'expression concernée n'est pas nullable et donc l'on n'a pas pas besoin de calculer γ".

Expression	choix de γ	γ'	γ"	d(γ)
$Ad \mid B\{f\}$	Ad	$\{a,e\}$	inutile	$\{a,e\}$
règle : $S \rightarrow Ad \mid B\{f\}$	$B\{f\}$	$\{b,f\}$	$\{\#\}$	$\{b,f,\#\}$
$aA \mid e$	a	$\{a\}$	inutile	$\{a\}$
règle : $A \rightarrow aA \mid e$	e	$\{e\}$	inutile	$\{e\}$
$\varepsilon \mid bB$	ε_B	$\{\}$	$\{f,\#\}$	$\{f,\#\}$
règle : $B \rightarrow \varepsilon \mid bB$	bB	$\{b\}$	inutile	$\{b\}$

On constate que l'essai des alternatives (e.g. Ad, $B\{f\}$) est conditioné par la présence de certains terminaux.

Aussi, dans la cas de la règle $B \rightarrow \varepsilon \,|\, bB$, si le terminal b n'est pas présent, on n'activera la branche ε que si l'un des terminaux $\{f,\#\}$ est présent. Rappelons que dans le cas d'un parcours non informé (sans la table), l'alternative ε réussiara sans condition ! (voir la traduction en code, fonction B).

Une autre remarque : les ensembles Directeurs dans cette table ont une intesection vide; dans le cas contraire, il faudra considérer plus d'un symbole : dans $LL(k),K > 1$.

13.6 Traduction en règles

A partir du tableau précédent, on peut écrire les règles d'analyse syntaxique :

```
bool S()
{if (dans(token, {'a','e'})   //test sur d(Ad)
         && A() && token=='d'
    )
    {token=get_token(); return true;}
 else
    if (dans(token, {'b','f','#'}) //test sur d(B{f})
          && B()
       )
       {while (token=='f') token=get_token();
        return true;
       }
    else return false;
}
bool A()
{if (dans(token, {'a'}))   //test sur d(aA)
    {token=get_token();
     return A();
    }
 else
    if (dans(token, {'e'})) //test sur d(e)
       {token=get_token();
        return true;
       }
    else return false;
}
bool B()
{if (dans(token, {'b'}))   //test sur d(bB)
    {token=get_token();
     return B();
    }
 else
    if (dans(token, {'f','#'})) //test sur d(epsilon)
        return true;
    else return false;
}
```

Exercice 1 : faire les calculs pour la grammaire G=({a,b,c}, {A,B,C}, A, $\mathcal{P}$) avec $\mathcal{P}$:

$$A \rightarrow [a|cc]BC.$$
$$B \rightarrow \varepsilon|bB.$$
$$C \rightarrow \varepsilon|cCc.$$

Exercice 2 : faire les calculs pour la grammaire G=({a,b,c}, {S,A,B,C}, S, $\mathcal{P}$) avec $\mathcal{P}$:

$$S \rightarrow AB|BC|CA.$$
$$A \rightarrow b|aA.$$
$$B \rightarrow a|bB.$$
$$C \rightarrow \varepsilon|cC.$$

13.7 Une définition plus formelle des ensembles

Une autre spécification (plus formelle) de la propriété ε-règle(X) et des ensembles *Premier(X)*, *Suivant(X)*, *Directeur(X)*, $X \in V_N$ est donnée ci-dessous.

On peut voir ces ensembles comme des relations :

⊡ **Nullable** est la plus petite relation Nu telle que :

- $Nu = Nu \cup \{(X,vrai) | X \rightarrow \varepsilon \in P\}$
- $Nu = Nu \cup \{(X,vrai) | X \rightarrow Y_1...Y_n \in P \ , (Y_i,vrai) \in Nu\}$

⊡ **First** est la plus petite relation F_i telle que :

- $F_i = Fi \cup \{(a,a) \mid a \in \Sigma\}$
- $F_i = F_i \cup \{(X,a) \mid X \rightarrow Y_1...Y_n \in P,$
  $\exists i,Y_1,...,Y_{i-1}\ Nullable\ et\ (Y_i,a) \in F_i\}$

⊡ **Follow** est la plus petite relation F_o telle que

- $F_o = F_o \cup \{(Y,a) | X \rightarrow Y_1...Y_i \mathbf{Y} Y_{i+1}...Y_n \in P,$
  $Y_{i+1},...,Y_n\ Nullable\ et\ (X,a) \in F_o\}$
- $F_o = F_o \cup \{(Y,a) | X \rightarrow Y_1...Y_i \mathbf{Y} Y_{i+1}...Y_j Y_{j+1}...Y_n \in P,$
  $\mathbf{Y}_{i+1},...,Y_j\ Nullable\ et\ (Y_{j+1},a) \in F_o\}$

Modus operandi de ces calculs :

Pour calculer le plus petit ensemble récursivement définis avec ces équations, on pourra trouver la plus petite solution en utilisant une itération jusqu'au plus petit point fixe en partant de l'ensemble vide.

L'existance et la calculabilité de ce point fixe sont assurées par la propriété de continuïté (par rapport à une topologie bien choisie) des opérations ensemblistes utilisées ci-dessus.

13.8 Remplissage de la table LL(1)

Dans la table :

	t_1	...	t	...	t_m
X_1		...	...	...	
...	...		...	...	
X_i	...		$X_i \to \gamma$	...	
...	...		...	...	
X_n	...		...	...	

On analyse toute production $X_i \to \gamma$ et (ensemble *Directeur*) et on met $X_i \to \gamma$ dans la case (X_i,t) **si** $t \in Directeur(\gamma)$

Dans les cas les plus simples, cela revient (cf. la définition de l'ensemble *Directeur*) à :

* mettre $X_i \to \gamma$ dans la case (X_i,t) **si** $t \in Premier(\gamma)$

* mettre $X_i \to \gamma$ dans la case (X_i,t) **si** $t \in Suivant(X_i)$ et Nullable(γ).

Si la table n'a pas d'entrée multiple, alors le langage est LL(1) et la grammaire est analysable par un analyseur LL(1).

14 Transformations des grammaires

Quelques transformations préliminaires sont appliquées aux grammaires pour les rendre exploitable.

14.1 Elimination de l'ambiguïté

L'existence d'une grammaire non ambiguë pour un langage donné L n'est pas décidable.

Cependant, dans les cas les plus fréquents, on sait éliminer l'ambiguïté assez facilement :

Exemple: on peut réécrire la grammaire (ambiguë) :

$$E \rightarrow E + E \mid E * E \mid (E) \mid int \mid id$$

en une grammaire équivalente (prouvable) où une priorité et une associativité sont imposées sur les opérateurs :

$$E \rightarrow E + T | T$$
$$T \rightarrow T * F | F$$
$$F \rightarrow (E) | int | id$$

Ici, les priorités deviennent : '()' > '$*$' > '+'

et '$*$' et '+' deviennent associatifs à gauche.

14.2 Dérécursivisation

Une grammaire recursive a gauche pose toujours de problèmes pour une analyseur LL(1).

Si l'on a deux productions $E \rightarrow E\alpha$ et $E \rightarrow \beta$, alors il y aura toujours une double entrée dans les cases (E,t) pour $t \in First(\beta)$. C'est aussi le cas dans la grammaire des expressions précédente.

Elimination de la récursivité a gauche

Toute production

$$X \rightarrow X\gamma_1|...|X\gamma_n|\alpha_1|...|\alpha_m$$

peut être remplacée par les productions (équivalentes, $X' \notin V_N$ est un nouveau symbole) :

$$X \rightarrow \alpha_1 X'|...|\alpha_m X'$$

$$X' \rightarrow \gamma_1 X'|...|\gamma_n X'|\epsilon$$

Exemple :

$$E \rightarrow E\ \ '+'\ id \mid\ '1'$$

deviendra :

$$E \rightarrow\ '1'\ E'$$

$$E' \rightarrow\ '+'\ id\ E' \mid \epsilon$$

Remarque : si l'on écrite la règle

$$E \rightarrow E + id \mid\ '1'$$

sous la forme de l'équation (sur E)

$$E = E\alpha \mid\ \beta$$

Une solution pour E (sous forme d'une ER) est :

$$E = \beta\alpha^*$$

Ce qui dans l'exemple ci-dessus donnera l'expression régulière :

$$E =\ '1'\ (\ '+'\ id)^*$$

qui est identique au résultat de la dérécursification.

14.3 Factorisation gauche

Un autre exemple de problème pourles grammaires LL(1) apparaît si l'on a deux productions $E \rightarrow \alpha E'$ et $E \rightarrow \alpha E''$ avec le même préfixe α.

Dans ce cas, l'analyseur ne saura pas faire le bon choix.

On améliore la situation en "factorisant" à gauche le préfixe commun.

Toute production

$$X \rightarrow \alpha\beta_1 | \ldots | \alpha\beta_k | \gamma$$

peut être remplacée par les productions (équivalentes, $X' \notin V_N$ est un nouveau symbole) :

$$X \rightarrow \alpha X' | \gamma$$

$$X' \rightarrow \beta_1 | ... | \beta_k$$

Un exemple de factorisation gauche : le cas *if then else*

$$X \rightarrow \ if\ E\ then\ S\ else\ S \mid if\ E\ then\ S$$

devient

$$X \rightarrow if\ E\ then\ S\ X'$$

$$X' \rightarrow else\ S \mid \epsilon$$

N.B.: cette partie de la règle *if then else* n'est toujours pas LL(1) mais on sait mieux traiter le problème après la factorisation (e.g. en donnant priorité au cas "else")

15 Un exemple complet

Soit le langage des expressions arithmétiques :

$$S \rightarrow E\#$$

$$E \rightarrow E + E \mid E * E \mid (E) \mid int \mid id$$

Cette grammaire est ambiguë et récursive à gauche. On applique la technique de suppression de l'ambiguïté (désambiguation).

Désambiguation : en imposant que l'opérateur $*$ soit plus prioritaire (*lie plus*) que $+$ et qu'on associe les opérateurs à gauche, on obtient la grammaire non ambiguë:

$$S \rightarrow E\#$$

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

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

$$F \rightarrow (E) \mid int \mid id$$

Cette grammaire est recursive a gauche.

Dérécursivisation

En dérécursivisant, on obtient :

$$S \rightarrow E\#$$

$$E \rightarrow T\ E'$$

$$E' \rightarrow +TE'|\epsilon$$

$$T \rightarrow F\ T'$$

$$T' \rightarrow *FT'|\epsilon$$

$$F \rightarrow (E)|int|id$$

15.1 Calcul de Nullable, First et Follow

Ces calculs se feront dans cet ordre (dépendance des résultats).

Calcul des ε-règles (Nullable) :

Iteration	*S*	*E*	*E '*	*T*	*T '*	*F*
0	*?*	*?*	*?*	*?*	*?*	*?*
1	*?*	*?*	*true*	*?*	*true*	*?*
2 = 1	*false*	*false*	***true***	*false*	***true***	*false*

NB : à l'étape 1 (et en partant de l'ensmble vide de l'étape initiale 0), on trouve E ' et T ' Nullable (règles de la forme $A \rightarrow \epsilon$) puis aucun non terminal nullable.

Une autre manière de calculer la même information :

- $\text{Nu}_0 = \{\}$ i.e. à l'étape initiale, aucun symbole nullable.

- $\text{Nu}_1 = \text{Nu}_0 \cup \{E',T'\} = \{E',T'\}$,

- $\text{Nu}_2 = \text{Nu}_1 \cup \{\} = \text{Nu}_1$ i.e. à l'étape 2, rien est ajouté; l'agorithme s'arrête.

A la dernière ligne, on ajoutera $false$ au lieu de '?' car ici, ce qui n'est pas $true$ est $false$.

Calcul des Premiers (First) :

On utilise les résultats de Nullables :

	S	*E*	*E'*	*T*	*T'*	*F*
Nullable	*false*	*false*	*true*	*false*	*true*	*false*

Pour l'aspet pratique de cet algorithme, on choisira d'abord les règles dont la partie droite commence par un terminal (e.g. F, T ' et E '):

Iteration	*S*	*E*	*E'*	*T*	*T'*	*F*
0						
1			*{'+' }*		*{'*' }*	*{'(', int, id}*
2			*{'+' }*	*{'(', int, id}*	*{'*' }*	*{'(', int, id}*
3		*{'(', int, id}*	*{'+' }*	*{'(', int, id}*	*{'*' }*	*{'(', int, id}*
4	*{'(', int, id}*	*{'(', int, id}*	*{'+' }*	*{'(', int, id}*	*{'*' }*	*{'(', int, id}*
5=4	*{'(', int, id}*	*{'(', int, id}*	*{'+' }*	*{'(', int, id}*	*{'*' }*	*{'(', int, id}*

Calcul de Suivant (Follow) :

Les tables Nullable et First sont :

	S	*E*	*E'*	*T*	*T'*	*F*
Nullable	*false*	*false*	*true*	*false*	*true*	*false*
First	*{'(', int, id}*	*{'(', int, id}*	*{'+' }*	*{'(', int, id}*	*{'*' }*	*{'(', int, id}*

On procède au calcul de Follow :

Iteration	*S*	*E*	*E'*	*T*	*T'*	*F*
0						
1		*{'#',')'}*		*{'+'}*		*{'*'}*
2		*{'#',')'}*	*{'#',')'}*	*{'+','#',')'}*	*{'+'}*	*{'+','*"#',')'}*
3		*{'#',')'}*	*{'#',')'}*	*{'+','#',')'}*	*{'+','#',')'}*	*{'+','*"#',')'}*
4=3		*{'#',')'}*	*{'#',')'}*	*{'+','#',')'}*	*{'+','#',')'}*	*{'+','*','#',')'}*

L'ensemble des résultats (+ *Directors*) :

Rappel sur le calcul des *Directors* :

Directeur(X), $X \in V_N$. Soit la règle $X \rightarrow \alpha_1 ... \alpha_n$

- Si $\alpha_1 \vdash^* \varepsilon$ alors $Dir(X) = First(\alpha_1) \cup Follow(X)$

- Sinon $Dir(X) = First(\alpha_1)$

Les résultats de tous les calculs :

	S	E	E'	T	T'	F
Nullable	*false*	*false*	*true*	*false*	*true*	*false*
First	*{'(', int, id}*	*{'(', int, id}*	*{'+' }*	*{'(', int, id}*	*{'*' }*	*{'(', int, id}*
Follow		*{'#',')'}*	*{'#',')'}*	*{'+','#',')'}*	*{'+','#',')'}*	*{'+','*','#',')'}*
Director	*{'(', int, id}*	*{'(', int, id}*	*{'+',')' ,'#'}*	*{'(', int, id}*	*{'*','+',')' ,'#' }*	*{'(', int, id}*

On constate que Director=First pour *S,E,T,F*.

E ' et T ' sont Nullables. Pour E ' :

Director(E ')={'+'} $\cup$ *Follow(E ')*

Follow(E ') = Follow(E) ={')' ,'#'}, d'où,

Director(E ')= {'+',')' ,'#'}.

De même,

Director(T ')={''}* $\cup$ *Follow(T ')* qui donne

Director(T ')={'','+',')' ,'#' }* .

15.2 La table LL1

La table LL1 reflète l'ensemble des Directeurs.

$V_N \setminus Dir$	'+'	'*'	int = id	'('	')'	#
S			$S \to E\#$	$S \to E\#$		
E			$E \to TE'$	$E \to TE'$		
E'	$E' \to +TE'$				$E' \to \varepsilon$	$E' \to \varepsilon$
T			$T \to FT'$	$T \to FT'$		
T'	$T' \to \varepsilon$	$T' \to *FT'$			$T' \to \varepsilon$	$T' \to \varepsilon$
F			$F \to int$	$F \to int$		

Règle d'écriture des fonctions :

- On écrit une fonction par non terminal
- Etant placé dans $X \in V_N$ (la fonction $X()$), pour une case $(X,a), X \in V_N, a \in \Sigma$ de la ligne X de la table LL1 :
  - Si la case contient une règle de la forme $X \to ABC$
    - **tester la présence** de α
    - si $A = a \in \Sigma$, consommer(α) et appeler la fonction $B()$ (si $B = b \in \Sigma$, consommer(b) et passer à C ...)
    - si $A \in V_N$, appeler la fonction $A()$
    - si $A = \varepsilon$, retourner succès (pas de BC)
- S'il n'y a rien dans la case alors rien à faire.

Un exemple :

Nous avions calculé la table LL1 pour l'exemple précédent. Pour la table LL1 actuelle qui, comme on peut le constater, attribue une colonne par symbole terminal, nous aurons la forme générale de la fonction de l'analyse LL1.

	...	α	....	...
VN		$X \rightarrow ABC$		

Variable *token_courant : token;*
fonction VN() renvoie booléen
Dédut
- *Observer la ligne VN dans la table LL1*
- *Pour toute colonne concernant un symbole terminal* α
  - *Soit* $X \rightarrow ABC$ *la règle contenue dans la case* (VN,α)
  - *Si token_courant =* α *alors*
    - *Si la case (VN,* α*) contient la règle* $X \rightarrow ABC$ *alors Renvoyer* $\bigwedge_{\phi(X_i)}$ *pour* $X_i \in \{A,B,C\}$ *avec* $\phi(X_i)$ *:*
      - *Si* X_i *est un non-terminal alors* $\phi(X_i) \leftarrow call\ X_i()$
      - *Si* X_i *est un terminal alors* β*,* $\phi(X_i) \leftarrow (token\_courant = \beta)$*;*
      - *Si* X_i *est* ε *alors* $\phi(X_i) \leftarrow vrai$*; quitter VN;*
      - *Sinon* $\phi(X_i) \leftarrow vrai$*; token_courant* $\leftarrow$ *get_next_token();*
  - *Si token_courant ne correspond à aucun* α *alors renvoyer faux;*

Fin VN

15.3 L'Analyseur

Le code de l'analyseur des expression arithmétiques est donnée ci-dessous. On utlise ici Lex pour la partie lexicale.

```
/* analyseur d'expressions arithmétiques à base de la table LL1
/* la table LL1 tient compte des priorités/associations des opéra
/* donc, je ne dis rien ici la dessus (cf. yacc) */

/* Utilisation */
/* lex -v ce_fic.l */
/* gcc lex.yy.c */
/* a.out */

%{
#include <stdio.h>
#include <math.h>
int valeur;
typedef enum  {BIDON,ERR,ID,INT,PLUS,MULT,PARG,PARD,EOS} type_tok
/* je mets BIDON (jamais renvoyé)  au debut (code = 0) pour  */
/* renvoyer false (cf. appel yylex() ) */
/* on ne peut pas mettre EOF dans un enum  (réservé) */
%}

ident [a-zA-Z][a-zA-Z0-9]*
entier [0-9]+

%%
[ \t\n] ;
{ident}  {return ID;}
{entier} {return INT;}
"#"  {return EOS;}
"+"  {return PLUS;}
"*"  {printf("*");return MULT;}
"("  {return PARG;}
")"  {return PARD;}

<<EOF>> {return EOF;}
.  {ECHO; return ERR;}
```

```
%%

int yywrap() {return 1;}
int yyerror(char* ch,char * text)
   {printf("Error %s: %s\n",ch,text);}
#define bool int
#define true 1
#define false 0
type_token token; // globale sinon le passer en tt param
bool S(); bool E(); bool E1(); bool T(); bool T1(); bool F();
void erreur(char*);  bool check_eos();
int main()
{do
  {printf("donner des exepression terminées par # (ctrl-D=fin)\
   token=yylex();
   if (token==EOF) break;
   if (S()) printf("succes \n");
   else  printf("Echec\n");
  } while(true);
 printf( "fini \n");
 return 0;
}
```

Rappel de la table LL1 :

$V_N \setminus Dir$	'+'	'*'	int = id	'('	')'	#
S			$S \to E\#$	$S \to E\#$		

```
bool S()
{if(token==INT || token==ID)
   {E(); return check_eos();} // ne pas consommer
 if(token==PARG) {E(); return check_eos();}
 erreur("Err dans S"); return false;
}
```

Rappel de la table LL1 :

$V_N \setminus Dir$	'+'	'*'	int = id	'('	')'	#
E			$E \to TE'$	$E \to TE'$		

```
bool E()
{if(token==INT || token==ID) {return (T() && E1());}
 if(token==PARG) {return (T() && E1());}
 erreur("Err dans E"); return false;
}
```

Rappel de la table LL1 :

$V_N \setminus Dir$	'+'	'*'	int = id	'('	')'	#
E'	$E' \rightarrow +TE'$				$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$

```
bool E1()  /* pour E' */
{if(token==PLUS) {token=yylex(); return (T() && E1());}
 if(token==PARD) {return true;} // transition vide
 if(token==EOS) {return true;} // transition vide
 erreur("Err dans E1"); return false;
}
```

Rappel de la table LL1 :

$V_N \setminus Dir$	'+'	'*'	int = id	'('	')'	#
T			$T \rightarrow FT'$	$T \rightarrow FT'$		

```
bool T()
{if(token==INT || token==ID) {return (F() && T1());}
 if(token==PARG) {return (F() && T1());}
 erreur("Err dans T"); return false;
}
```

Rappel de la table LL1 :

$V_N \setminus Dir$	'+'	'*'	int = id	'('	')'	#
T'	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$			$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$

```
bool T1()
{if(token==MULT) {token=yylex(); return (F() && T1());}
```

```
 if(token==PLUS) {return true;} // transition vide
 if(token==PARD) {return true;} // transition vide
 if(token==EOS) {return true;} // transition vide
 erreur("Err dans T1"); return false;
}
```

Rappel de la table LL1 :

$V_N \setminus Dir$	'+'	'*'	int = id	'('	')'	#
F			$F \rightarrow int$	$F \rightarrow int$		

```
bool F()
{if(token==INT || token==ID) // consommer
    {token=yylex();return true;}
 if(token==PARG)   // détailler pour les erreurs
    {token=yylex();
     if (E())
       {if (token==PARD) {token=yylex();return true;}
        else {erreur("PARD attendue"); return false;}
       }
     else {erreur("Err de E dans F après PARG");
           return false;
          }
    }
 else {erreur("Err dans F"); return false;}
}

void erreur(char * ch)
{printf("ERREUR : %s, token=%s(code %d)\n",
         ch, yytext,token);
    }
bool check_eos() {return (token==EOS) ;}
```

Chapter 4

Analyse Syntaxique ascendante

Plan du Chapiter :

– Définition d'analyse ascendante LR

– Définition de poignée et préfixe viable

– Structure d'un analyseur ascendant

– Définition et construction des Items LR(k)

– Définition et construction des tables LR(k)

– Un exemple LR(0)

– Un exemple LR(1)

– Construction des tables SLR

– Construction des tables LALR(1)

16 Analyse Syntaxique ascendante

Objectif : on cherche a construire une dérivation droite à l'envers à partir de la chaîne de terminaux vers le symbole initial de la grammaire. En d'autres termes, on cherche à construire un arbre de dérivation a partir des feuilles selon un parcours qui est l'inverse d'un parcours en profondeur d'abord et de gauche a droite (LL). Dans l'appélation LR(k), le R vient du fait de considérer les règles de la grammaires de leur partie droite (Right) vers leur partie gauche.

Pour savoir simplement comment et quand arrêter l'analyse avec succès pour une grammaire G, on travaille toujours sur une grammaire dite *augmentée* G0 qui est G avec un nouveau symbole S_0 comme symbole de départ, un nouveau symbole terminal # et une transition supplémentaire :

$$S_0 \rightarrow S\#$$

Terminologie

Dans ce qui suit, on utilisera les conventions suivantes:

- des lettres majuscules A,B,...,Z indiquent des symboles non-terminaux (éléments de V_N),

- des lettres grecques α,β,γ,... $\in V = V_N \cup \Sigma$. Elles indiquent donc des séquences de symboles terminaux (éléments de Σ) ou non-terminaux (éléments de V_N),

- des lettres minuscules a,b,c,... indiquent des symboles terminaux (éléments de Σ)

- des lettres minuscules u,v,x,y,w,z indiquent des séquences de symboles terminaux (éléments de Σ^*)

Analyseurs LR: les analyseurs ascendants le plus connus sont dans la classe LR des analyseurs qui lisent le flot de tokens en entrée de gauche a droite (le L dans LR) pour reconstruire une dérivation droite (le R dans LR).

Dérivation droite: une dérivation droite est une dérivation qui remplace à chaque étape le symbole non terminal le plus à droite.

On notera $\alpha \rightarrow_d \beta$ une étape de dérivation droite entre α et β.

On notera $\alpha \rightarrow_d^* \beta$ une série (même vide) d'étapes de dérivation droite entre α et β.

Protophrase d'une grammaire G : une protophrase est une séquence de symboles terminaux et non terminaux qui peut apparaître en cours d'une dérivation du symbole initial S d'une grammaire G. On parle de *protophrase droite* (resp. *gauche*) lorsque cette séquence peut apparaître dans une dérivation droite (resp. gauche) de G.

Poignée (handle) : dans une protophrase ϕ, la séquence γ est une poignée à la position n pour la grammaire G si elle est la partie gauche d'une production $X \rightarrow \gamma$, et que cette production soit appliquée à ϕ en position n pour construire la protophrase précédente dans une dérivation droite à partir de S vers ϕ avec la grammaire G.

Préfixe viable : Une séquence γ est un préfixe viable pour une grammaire G si γ est un préfixe de $\alpha\beta$, où ϕ = $\alpha\beta\omega$ est une protophrase droite de G et β est une poignée dans cette protophrase. Autrement dit, un préfixe viable est un préfixe γ d'une protophrase ϕ, mais qui ne s'étend pas plus à droite d'une poignée β de ϕ.

Un exemple Sur la grammaire augmentée

$$S \to E\#$$
$$E \to E + T | T$$
$$T \to id$$

On a une unique dérivation droite pour $id + id + id$# (lire de bas en haut) :

$\bar{S}$	S
$\bar{E}\#$	$\underline{E\#}$
$E + \bar{T}\#$	$\underline{E + T}$#
$\bar{E} + id\#$	$E + \underline{id}\#$
$E + \bar{T} + id\#$	$\underline{E + T} + id\#$
$\bar{E} + id + id\#$	$E + \underline{id} + id\#$
$\bar{T} + id + id\#$	$\underline{T} + id + id\#$
$id + id + id\#$	$\underline{id} + id + id\#$

Chaque ligne est une protophrase droite, en surligné les symboles produits, et en souligné les poignées. Les préfixes viables sont les préfixes qui ne s'étendent pas plus loin qu'une poignée. Par exemple, sur la protophrase $E + \underline{id} + id\#$ ce sont ε, *E*, *E*+, *E* + *id*.

FIRST$_k(\gamma)$: étant donnée une grammaire G, l'ensemble $FIRST_k(\gamma$ contient les préfixes de longueur k des séquences de terminaux de longueur au moins k dérivables à partir de γ dans G, et les séquences de terminaux de longueur inférieur à k dérivables depuis γ.

EFF$_k(\gamma)$ (ε-free $FIRST_k$)) : étant donnée une grammaire G, EFF_k est le sous ensemble de $FIRST_k$ obtenu en considérant seulement les dérivations qui ne réduisent pas un non terminal en tête de chaîne sur ε .

Exemple : pour la grammaire:

$$S \rightarrow AB$$

$$A \rightarrow Ba|\varepsilon$$

$$B \rightarrow Cb|C$$

$$C \rightarrow c|\varepsilon$$

on a:

$FIRST_2(S) = \{\varepsilon,a,b,c,ab,ac,ba,ca,cb\}$

$EFF_2(S) = \{ca,cb\}$

17 Grammaires LR(k)

Définition :

une grammaire G est dans $LR(k)(k \geq 0)$ si les trois conditions :

– $S \rightarrow_d^* \alpha A\omega \rightarrow_d \alpha\beta\omega$

– $S \rightarrow_d^* \gamma Bx \rightarrow_d \alpha\beta y$

– $FIRST_k(\omega) = FIRST_k(y)$

impliquent $\alpha = \gamma$,$A = B$,$x = y$

18 Structure de l'analyseur

Les grammaires $LR(k)$ sont celles dont le langage est reconnu par un analyseur **déterministe** $LR(k)$. Cet analyseur utilise une pile et le flot d'entrée qui décrivent une *configuration* de l'analyseur.

Cette *configuration* est notée :

$$[X_1...X_j \qquad , \qquad a_i...a_n]$$

où les X sont de symboles de V_N (terminaux ou non terminaux) stockés sur la pile, alors que les $a \in \Sigma$ sont seulement des symboles terminaux et correspondent aux terminaux non encore lus sur le flot d'entrée.

L'analyseur travaille en effectuant quatre actions possibles:

shift (décalage) : on transfert le terminal a_i du flot d'entrée vers la pile

reduce (réduction) : on reconnaît sur le sommet de la pile une partie droite d'une production $Y \rightarrow X_{j-k}...X_j$.
On enlève celle-ci de la pile et on la remplace par la partie gauche Y.

erreur : l'analyseur s'arrête et signale un erreur

accept : l'analyseur s'arrête et signale que la phrase a été reconnue.

Pour choisir les actions, on utilise une table d'analyse que l'on verra plus loin.

18.1 Un exemple

Pour la grammaire augmentée :

$$S \rightarrow E\#$$

$$E \rightarrow E + T | T$$

$$T \rightarrow id$$

Une séquence possible de reconnaissance pour $id + id + id\#$ pour un analyseur ascendant pourrait être:

Configuration			*Action*
[	,	$id + id + id\#$]	$shift$
[$\underline{id}$	,	$+id + id\#$]	$reduce$
[$\underline{T}$	,	$+id + id\#$]	$reduce$
[E	,	$+id + id\#$]	$shift$
[$E+$	,	$id + id\#$]	$shift$
[$E + \underline{id}$	,	$+id\#$]	$reduce$
[$\underline{E + T}$	,	$+id\#$]	$reduce$
[E	,	$+id\#$]	$shift$
[$E+$	,	$id\#$]	$shift$
[$E + \underline{id}$	,	$\#$]	$reduce$
[$\underline{E + T}$	,	$\#$]	$reduce$
[E	,	$\#$]	$accept$

Remarques sur la configurations et protophrases:

la concatenation de la partie gauche et droite d'une *configuration* (entre []) d'un analyseur ascendant pour une grammaire G est toujours une protophrase droite de G (si l'analyse se termine avec succès).

Remarques sur les préfixes viables: un préfixe viable peut toujours se compléter en une protophrase droite. En d'autre termes, il n'y a pas d'erreur au cours de l'analyse tant que l'on a sur la pile un préfixe viable.

18.2 Un autre exemple, avec look-ahead

Pour la grammaire augmentée (non récursive à gauche):

$$S \rightarrow E\#$$

$$E \rightarrow T + E | T$$

$$T \rightarrow id$$

Une séquence possible de reconnaissance pour $id + id + id\#$ pour un analyseur ascendant pourrait être:

Configuration			*Action*
[	,	$id + id + id\#$]	$shift$
[$\underline{id}$	,	$+id + id\#$]	$reduce$
[T	,	$+id + id\#$]	$shift$
[$T+$	,	$id + id\#$]	$shift$
[$T + \underline{id}$	,	$+id\#$]	$reduce$
[$T + T$	,	$+id\#$]	$shift^*$
[$T + T+$	,	$id\#$]	$shift$
[$T + T + \underline{id}$	,	$\#$]	$reduce$
[$T + T + \underline{T}$	,	$\#$]	$reduce^*$
[$T + \underline{T + E}$	,	$\#$]	$reduce$
[$\underline{T + E}$	,	$\#$]	$reduce$
[E	,	$\#$]	$accept$

(*) : Look-ahead pour décider.

19 Analyseurs LR

Un analyseur LR est composé de :

une pile et un flot d'entrée comme nous l'avons vu dans les exemples

une table d'analyse qui décrit un automate à états fini augmenté avec des actions à effectuer éventuellement sur la pile (*shift, reduce, accept, error*)

L'exécution de l'automate conduit à déplacer (shift) sur la pile des symboles jusqu'à atteindre un *préfixe viable maximal*, puis réduire la poignée en la remplaçant par X avec la partie droite de la production $X \rightarrow \gamma$ concernée.

Remarque sur le préfixe viable maximal : c'est un préfixe non extensible, i.e. contenant une poignée γ au sommet de la pile.

Fonctionnement d'un analyseur LR :

Sur un état d'analyseur [α , $x\omega$], le fonctionnement de l'analyseur LR est le suivant:

- exécuter l'automate à partir de l'état initial s_1 sur la pile α, ce qui nous laisse sur un état s_k
- exécuter l'action décrite dans la table d'analyse associée au symbole terminal x en entrée pour l'état s_k. Cette action peut être :

  shift (noté s) déplacer le symbole d'entrée x sur la pile,

  reduce (noté rn**)** sur le sommet de la pile se trouve la partie droite de la règle $X \rightarrow \gamma$ numérotée n; dépiler γ et empiler X

  accept (noté a) arrêter avec succès

  error (noté par une case vide) arrêter sur erreur

- recommencer avec le nouvel état d'analyseur

19.1 Exemple d'exécution avec une table d'analyse LALR(1)

$(0) S \rightarrow E\#$ $(2) E \rightarrow T$

$(1) E \rightarrow T + E$ $(3) T \rightarrow id$

Ci-dessous, on note (dans la seconde table) les états de l'automate après lecture de chaque symbole sur la pile.

Num.	**Action**			*Transition*				
règle	id	+	$\#$	id	$+$	#	E	T
1	S			5			2	3
2			a					
3		S	r_2		4			
4	S			5			6	3
5		r_3	r_3					
6			r_1					

Les configurations :

Configuration			*Action*
$[_1$	,	$id + id\#]$	$shift$
$[\ _1\underline{id}_5$	,	$+id\#\ ]$	$reduce$
$[\ _1T_3$	,	$+id\#\ ]$	$shift^*$
$[\ _1T_3{+}_4$	,	$id\#\ ]$	$shift$
$[\ _1T_3 +_4 \underline{id}_5$	,	$\#\ ]$	$reduce$
$[\ _1T_3 +_4 \underline{T}_3$	,	$\#\ ]$	$reduce^*$
$[\ \underline{_1T_3 +_4 E_6}$	,	$\#\ ]$	$reduce$
$[\ _1T_3$	,	$\#\ ]$	$accept$

19.2 Analyseur avec états sur la pile

Si l'on garde les états sur la pile (en modifiant la notion de configuration pour que chaque symbole soit suivi par un état), on peut éviter de relire toute la pile à chaque fois.

Dans la configuration $[s_1 X_1 .. s_{k-1} X_{k-1} s_k \quad , \quad x\omega]$, l'analyseur LR exécute pour l'état s_k, à l'aide de la table d'analyse, l'action associée au symbole terminal x en entrée :

shift k déplacer le symbole d'entrée x sur la pile, et empiler l'état numéro k

reduce n sur le sommet de la pile, il y a la partie droite de la règle $X \to \gamma$ numérotée n; dépiler γ et tous les états associés, en découvrant l'état s'; empiler X et l'état $s"$ contenu dans la table à la ligne s', colonne X

accept arrêter avec succès

error arrêter sur erreur

Cela produit la nouvelle configuration.

20 Production d'une table d'analyse

Il faut identifier les préfixes viables et déterminer quelles productions utiliser pour les réductions, éventuellement en utilisant k tokens en entrée pour aider dans la décision. Pour reconnaître les préfixes viables, on définit la notion de *ITEM LR(k)*.

ITEM LR(k) : Un *ITEM LR(k)* pour une grammaire G est une production $X \to \gamma$ de G plus une position j dans γ et une séquence de longueur $\leq k$.

Si γ = $\alpha\beta$ avec j la longueur $|\alpha|$, on note :

$$X \to \alpha \bullet \beta, \omega$$

sauf dans le cas LR(0) pour lequel on écrit simplement $X \to \alpha \bullet \beta$

Par intuition, ($X \to \alpha \bullet \beta$,$\omega$) veut dire que l'on a déjà vu en entrée le préfixe α d'une protophrase et l'on attend sur l'entrée une séquence dérivable à partir de $\beta\omega$.

Reconnaissance des préfixes viables: la fermeture :

Si l'on a ($A \to \alpha \bullet X\beta$,$z$), i.e. on a déjà vu en entrée le préfixe α et on attend une séquence dérivable à partir de $X\beta z$, on peut aussi attendre une séquence dérivable depuis X, suivie d'une séquence dérivable depuis βz.

Cette notion est exprimée par la définition suivante de la *fermeture* :

Fermeture (Closure) LR(k) :

Fermeture(I) =

répéter tant que I grandit

  pour tout item (A $\to \alpha \bullet$ Xβ,z) dans I

    pour toute production X $\to \gamma$

      pour tout $\omega \in$ FIRST$_k(\beta$ z)

        I $\leftarrow$ I $\cup \{$(X $\to \gamma$, ω)$\}$

retourner I

Reconnaître les préfixes viables: GOTO :

Supposons avoir ($A \to \alpha \bullet X\beta$,$z$), pour un symbole terminal ou non terminal X: on a donc déjà vu en entrée le préfixe α et on attend une séquence dérivable à partir de $X\beta z$. Si

maintenant l'on reconnaît X, alors on a vu αX et on attend une séquence dérivable à partir de βz.

Cette notion est exprimée par *GOTO* :

Goto(I,X) =

J $\leftarrow$ 0;

pour tout item(A $\rightarrow \alpha \bullet$ Xβ, z) dans I

    J $\leftarrow$ J $\cup$ {(A $\rightarrow \alpha$X $\bullet \beta$,z)}

retourner Fermeture(J)

L'automate qui reconnaît les préfixes viables:

Soit G une grammaire augmentée et $\mathcal{I}$= $\{s_0; s_1; ...; s_k\}$ la collection d'ensembles d'*ITEMS LR(k)* attaignables depuis la fermeture de l'item $s_0 = (S' \rightarrow \bullet S\#,\varepsilon)$ par la fonction *GOTO*.

On peut alors construire l'automate d'états fini suivant:

états : $\{s_0; s_1; ...; s_k\}$

état initial : s_0

états finaux : ceux qui contiennent au moins un ITEM LR(k) avec le point au fond à droite (i.e. de la forme $(X \rightarrow \gamma \bullet ,\omega)$)

transitions : on a une transition de l'état s_i vers l'état s_j sur le symbole X si $GOTO(s_i,X) = s_j$

21 La construction de la table LR(k)

Soit G une grammaire augmentée pour laquelle on a construit l'automate précedent.

La table d'analyse a une ligne par état et un colonne par séquence de symboles terminaux de longueur $\leq k$ (le *look-ahead*) et une colonne par symbole terminal et nonterminal.

On remplit cette table de la façon suivante:

pour tout état $s_i \in \mathcal{I}$,

- inscrire $reduce\ n$ dans la case (s_i,u) si $(A \rightarrow \beta \bullet u) \in s_i$ et $A \rightarrow \beta$ est la production numéro $n \geq 1$
- inscrire *accept* dans la case (s_i,#) si $(S' \rightarrow \beta \bullet ,\#) \in s_i$
- inscrire *shift* dans la case (s_i,u) si $(A \rightarrow \beta_1 \bullet \beta_2,v) \in s_i$ et $u \in EFF_k(\beta_2 v)$
- laisser vide (i.e. on signale *erreur*) autrement

21.1 Théorème fondamental de l'analyse ascendante

Considérons le processus de construction de la table précédente.

Si une grammaire est LR(k), alors l'automate construit reconnaît les préfixes viables de G et tout état contenant un ITEM LR(k) de la forme ($X \rightarrow \gamma \bullet ,\omega$) ne contient pas un ITEM ($X' \rightarrow \gamma_1 \bullet \gamma_2,u$) avec $\omega \in EFF_k(\beta_2 v)$.

En d'autre terme, dans la table, on n'aura pas d'entrées multiples *shift* et *reduce* consecutives ou entrelacées.

Comment l'analyseur peut-il choisir l'action à effectuer?

Un analyseur LR dispose de plus d'information qu'un analyseur LL pour déterminer la prochaine action.

Imaginons avoir en entrée une chaîne uvw, et avoir déjà lu u.

Pour déterminer la production à appliquer

• un analyseur LL(k) connaît u et $FIRST_k(vw)$

• un analyseur LR(k) connaît uv (en effet, il connaît un préfixe viable γ obtenu à partir de uv) et $FIRST_k(w)$

21.2 Un exemple LR(0)

La grammaire $G1$ suivante est LR(0)

$(0)S' \rightarrow S \quad (1)S \rightarrow (L) \quad (3)L \rightarrow S \quad (2)S \rightarrow x \quad (4)L \rightarrow L,S$

La construction complète de la table LR(0) de $G1$:

Etats et transitions (2,4,6,7,9 sont des terminaux) :

1. $\{(S' \rightarrow \bullet S\#)$, $(S \rightarrow \bullet(L))$, $(S \rightarrow \bullet x)\}$
   $\mathbf{goto}(1,S) = 4 \quad goto(1,() = 3 \quad goto(1,x) = 2$
2. $\{(S \rightarrow x\bullet)\}$
3. $\{(S \rightarrow (\bullet L)),(L \rightarrow \bullet S),(L \rightarrow \bullet L,S),(S \rightarrow \bullet(L)),(S \rightarrow \bullet x)\}$
   $goto(3,() = 3 \quad goto(3,x) = 2 \quad goto(3,S) = 7 \quad goto(3,L) = 5$
4. $\{(S' \rightarrow S \bullet \#)\}$
5. $\{(S \rightarrow (L\bullet)),(L \rightarrow L \bullet ,S)\}$
   $goto(5,)) = 6 \quad goto(5,,) = 8$
6. $\{(S \rightarrow (L)\bullet)\}$
7. $\{(L \rightarrow S\bullet)\}$
8. $\{(L \rightarrow L, \bullet S),(S \rightarrow \bullet(L)),(S \rightarrow \bullet x)\}$
   $goto(8,x) = 2 \quad goto(8,() = 3 \quad goto(8,S) = 9$
9. $\{(L \rightarrow L,S\bullet)\}$

La table d'analyse LR(0) :

Pour remplir la table LR(0), on écrit la table de transition de l'automate et on introduit les actions de décalage (s pour *shift*) comme décrit plus haut.

Pour les réductions (rk pour *reduce avec la production k*), n'ayant pas de look-ahead dans les états, on met rk dans toute la ligne action de l'état j si l'état j contient un ITEM LR(0) $X \to \gamma\bullet$, et que $X \to \gamma$ est la production numéro j.

Num.	**Action**					$Transition$						
règle	(	)	x	,	#	(	)	x	,	#	S	L
1	S		S			3		2			4	
2	r_2	r_2	r_2	r_2	r_2							
3	S		S			3		2			7	5
4					a							
5		S		S				6		8		
6	r_1	r_1	r_1	r_1	r_1							
7	r_3	r_3	r_3	r_3	r_3							
8	S		S			3		2			9	
9	r_4	r_4	r_4	r_4	r_4							

La table d'analyse LR(0), versions compacte :

Remarque : Les générateurs d'analyseurs, comme *Yacc*, fusionnent les colonnes des *Actions* et des transitions pour les terminaux: plutôt que d'avoir une case case $(1,x)$ qui contient *s(hift)* pour les actions, et une *case (1, x)* qui contient 2 pour les transitions; on préfère avoir une seule case $(1,x)$ qui contient $s2$, pour exprimer : *l'action est un shift et la transition est vers l'état 2.*

Dans ce cas, on écrit gk dans les colonnes *Transitions* restantes (celles des non-terminaux). C'est une abréviation pour *"goto k"*, plus lisible que juste k.

Num.	**Action**					$Transition$	
règle	(	)	x	,	#	S	L
1	S_3		S_2			$g4$	
2	r_2	r_2	r_2	r_2	r_2		
3	S_3		S_2			$g7$	$g5$
4					a		
5		S_6		S_8			
6	r_1	r_1	r_1	r_1	r_1		
7	r_3	r_3	r_3	r_3	r_3		
8	S_3		S_2			$g9$	
9	r_4	r_4	r_4	r_4	r_4		

21.3 Un exemple LR(1) non LR(0)

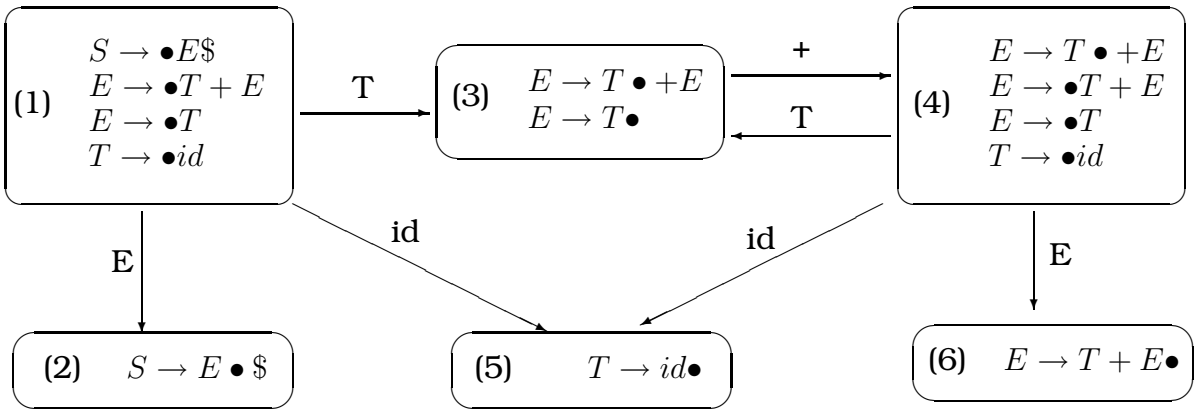
La grammaire $G2$:

$$(0)S \to E\# \qquad (2)E \to T$$

$$(1)E \to T + E \qquad (3)T \to id$$

est une grammaire LR(1) mais pas LR(0).

En effet, dans l'automate on a un problème pour l'état $\{(E \to T \bullet + E),(E \to T\bullet)\}$.



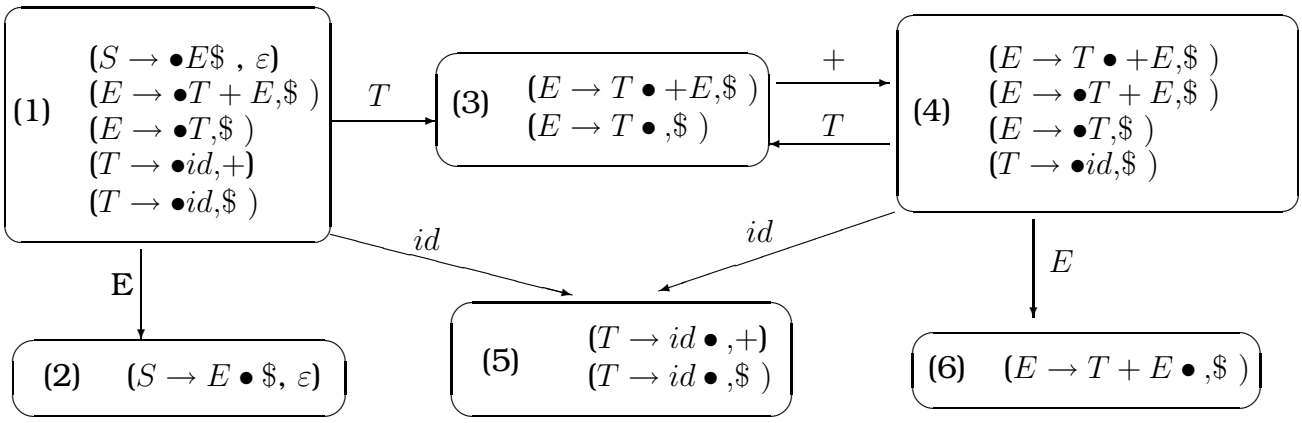
Dans la table d'analyse LR(0) on trouve un conflit $shift/reduce$ dans la case $3,+$

Num. règle	**Action**			$Transition$	
	$+$	id	$\#$	E	T
1		S_5		$g2$	$g3$
2			a		
3	$\mathbf{r_2}$,$\mathbf{S_4}$	r_2	r_2		
4		$S_7?$		$g6$	$g3$
5	r_3	r_3	r_3		
6	r_1	r_1	r_1		

Les états LR(1) de G2 :

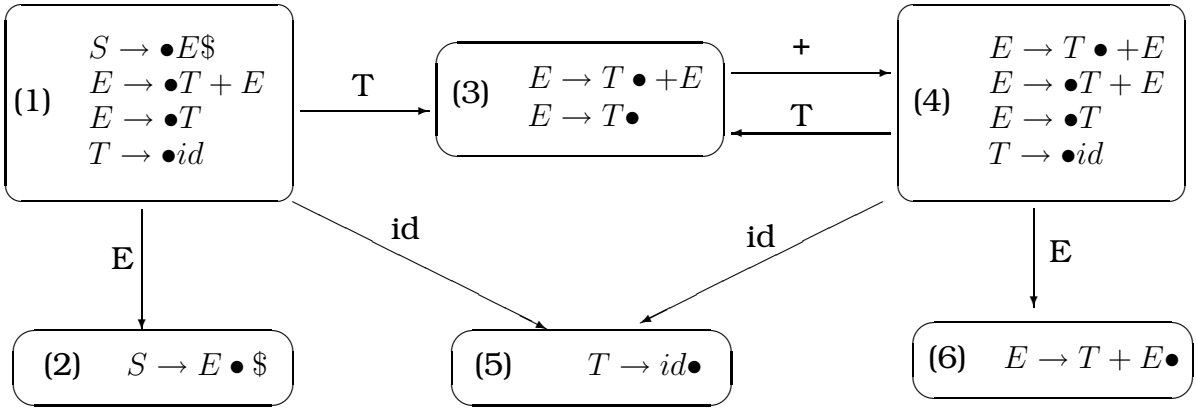
Regardons alors la construction LR(1) qui garde la trace des look-aheads dans les états

Etats et transitions (2, 5 et 6 sont des terminaux)

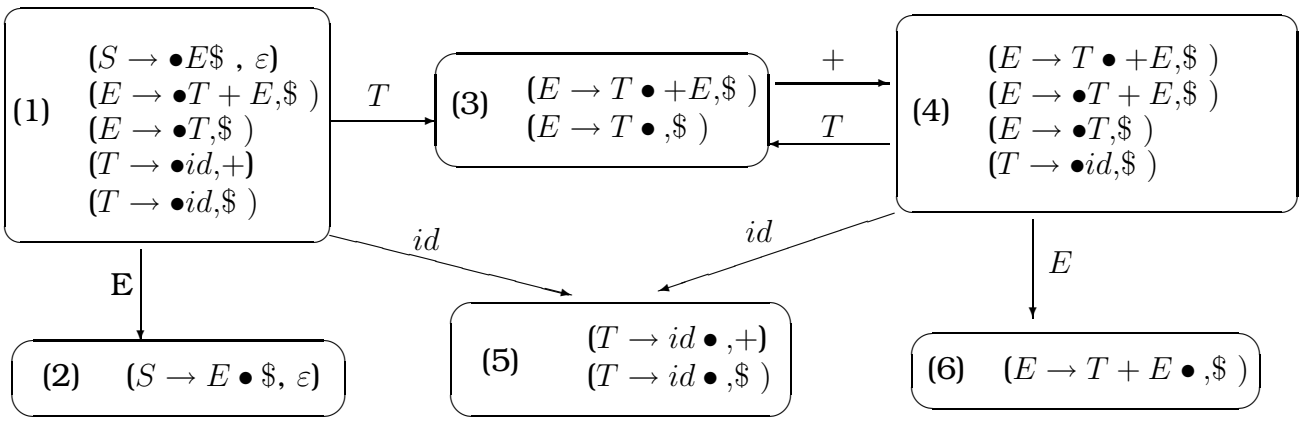


Comparaison des automates LR(0) et LR(1) pour G2 :

LR(0):



LR(1):



La table d'analyse LR(1) de G2 :

On garde trace des look-ahead pour introduire les actions *reduce* dans la table. Il y aura moins d'action et le conflit disparaît !

Num.	**Action**			$Transition$				
règle	$+$	id	$\#$	$+$	id	$\#$	E	T
1		s			5		2	3
2			a					
3	s		r_2	4				
4		s			5		6	3
5	r_3		r_3					
6			r_1					

La table d'analyse LR(1) de G2, version compacte :

Num.	**Action**			$Transition$	
règle	$+$	id	$\#$	E	T
1		S_5		$g2$	$g3$
2			a		
3	S_4		r_2		
4		S_5		$g6$	$g3$
5	r_3		r_3		
6			r_1		

Comparaison des tables LR(0) et LR(1) pour G2 :

LR(0):

Num. règle	**Action**			$Transition$	
	$+$	id	$\#$	E	T
1		s_5		$g2$	$g3$
2			a		
3	$\mathbf{r_2}$,$\mathbf{s_4}$	r_2	r_2		
4		$S_7?$		$g6$	$g3$
5	r_3	r_3	r_3		
6	r_1	r_1	r_1		

LR(1):

Num. règle	**Action**			$Transition$	
	$+$	id	$\#$	E	T
1		s_5		$g2$	$g3$
2			a		
3	s_4		r_2		
4		s_5		$g6$	$g3$
5	r_3		r_3		
6			r_1		

Quel k choisir dans les LR(k) :

La classe d'analyseurs LR(0) est trop faible pour traiter les langages de programmation: même le simple langage des expressions pose problème.

Les classes LR(2), LR(3),... ont par contre une table d'analyse trop grande dans la pratique et ce en raison du nombre de colonnes pour le "look-ahead": un analyseur moderne utilise plusieurs dizaines de tokens, et une colonne pour chaque séquence de tokens de longueur inférieur ou égale à k. De ce fait, pour $k \geq 2$, la construction est déraisonnable ttable trop grande).

N.B. : le nombre de séquences de longueur inférieure ou égale à k si l'on se donne n tokens différents est $n.\sum_1^k C_k^i$?

Choisir entre LR(0) et LR(1) :

Heureusement, la classe LR(1) est largement suffisante pour les langages modernes, et la table n'a qu'une colonne par token. Mais là, c'est le nombre d'états qui grandit trop, en raisons de la présence de look-ahead qui départage les états qui sont très peu différents.

Pour cette raison, on utilise dans la pratique deux types d'analyseurs dont le pouvoir d'expression est comprise entre celle de LR(0) et celle de LR(1): **SLR** et **LALR(1)**.

22 Analyseurs SLR

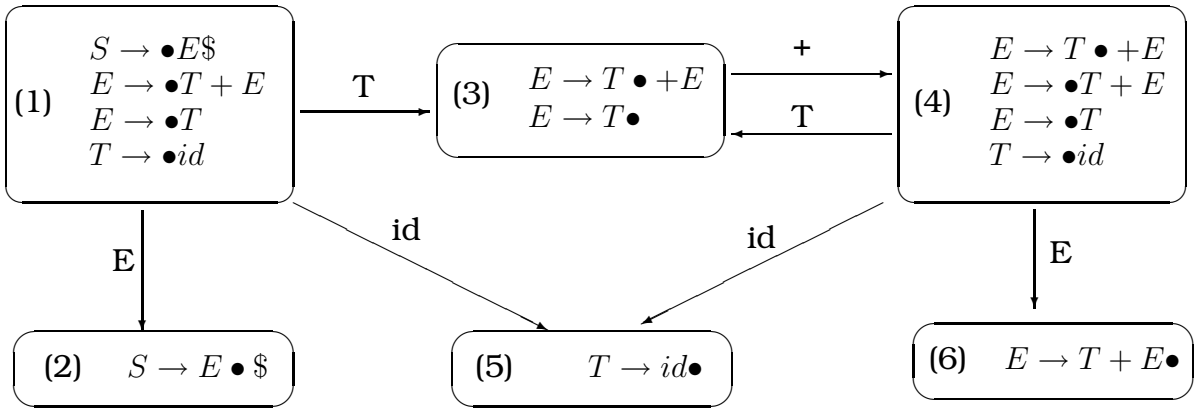
SLR (Simple LR) est un analyseur dont l'automate est celui de LR(0), donc la partie transition est la même que LR(0) ainsi que les actions de décalage (shift), mais la table d'analyse est construite de façon plus fine:
on pallie l'absence de look-ahead dans les états par l'information contenue dans les ensembles $Suivant$ (Follow) construits à partir de la grammaire.

La règle de placement des réductions (reduce) devient alors:

si l'état j contient un ITEM LR(0) $X \to \gamma\bullet$, et que $X \to \gamma$ est la production numéro $j \geq 1$, on met rk dans toutes les cases (j,t) telles que $t \in Follow(X)$.

SLR pour la grammaire G2 :

L'automate SLR étant le même que celui LR(0) :



Maintenant la table SLR contiendra des entrées *reduce* seulement sur certains non terminaux (pas tous!).
En particulier, on pourra éviter le conflit $shift/reduce$ dans l'état $\{(E \rightarrow T \bullet +E),(E \rightarrow T\bullet)\}$.

Num.	**Action**			$Transition$	
règle	$+$	id	$\#$	E	T
1		s_5		$g2$	$g3$
2			a		
3	s_4		r_2		
4		S_5		$g6$	$g3$
5	r_3		r_3		
6			r_1		

Dans ce cas précis, SLR fait aussi bien que LR(1), avec moins d'effort.

23 Analyseurs LALR(1)

LALR(1) (Look-Ahead LR(1)) est une classe d'analyseurs dont l'automate est obtenu de l'automate LR(1) en fusionnant les états qui diffèrent seulement par leur look-ahead.

On dit aussi que l'on fusionne les états ayant le même *coeur*, le coeur d'un état étant l'ensemble des parties gauches des ITEMS LR(1) qu'il contient, i.e. sans le look-ahead, i.e. des ITEMS LR(0).

Un analyseur LALR(1) a donc autant d'états qu'un LR(0) ou SLR. Les analyseurs LALR(1) sont les plus utilisés parce que, même s'ils ont moins d'états qu'un analyseur LR(1), il est très rare que l'on retrouve un conflit dans la table LALR(1) quand il n'y en a pas dans la table LR(1). En particulier, on peut prouver que si un analyseur LR(1) n'a pas de conflit *shift/reduce*, l'analyseur LALR(1) n'en a pas non plus. Par contre, on peut introduire des conflits *reduce/reduce*.

23.1 LALR(1) pour G2

Dans le cas précis de cette grammaire, l'automate LR(1) pour G2 n'ayant pas d'états différents avec le même coeur, la table d'analyse LALR(1) de G2 est la même que celle LR(1).

La grammaire LR(1) $G2$:

$$(0)S \rightarrow E\# \qquad (2)E \rightarrow T$$

$$(1)E \rightarrow T + E \qquad (3)T \rightarrow id$$

La version compact de la table :

Num.	**Action**			$Transition$	
règle	$+$	id	$\#$	E	T
1		S_5		$g2$	$g3$
2			a		
3	S_4		r_2		
4		S_5		$g6$	$g3$
5	r_3		r_3		
6			r_1		

23.2 Un exemple LALR(1) mais non SLR

La grammaire suivante qui engendre un sous-ensemble des expressions du langage C est un exemple de grammaire LALR(1) qui n'est pas SLR et pour laquelle l'automate LALR(1) est plus petit que l'automate LR(1).

(0) $S' \rightarrow S\#$ (3) $E \rightarrow V$

(1) $S \rightarrow V = E$ (4) $V \rightarrow id$

(2) $S \rightarrow E$ (5) $V \rightarrow *E$

LR préfère l'associativité à gauche :

Contrairement aux analyseurs LL, dans le cas de l'analyse ascendante, **on a plutôt intérêt** à utiliser des grammaires récursives à gauche.

Considérons les analyseurs LR pour la grammaire récursive à droite :

$S \rightarrow E\#$ $\mathbf{E} \rightarrow T$ $E \rightarrow T + E$ $T \rightarrow id$

Comme nous l'avons déjà vu, pour reconnaître $id+id+...+id$, l'nalyseur empile toute la suite de symboles (en réduisant id

sur T à chaque coup) avant de faire la prémière reduction non triviale. Par contre, la grammaire recursive à gauche :

$S \to E\#$ $\quad$ $\mathbf{E} \to T$ $\quad$ $E \to E + T$ $\quad$ $T \to id$

mantiendra la dimension de la pile à un minimum.

24 Utilisation de grammaires ambiguës

Une grammaire ambiguë n'est jamais LR(k), quelque soit k. Pourtant, on a intérêt à essayer d'utiliser une grammaire ambiguë, quitte à manipuler l'automate LR(k), si l'on peut.

Efficacité dans une grammaire obtenue par désambiguation, l'analyseur passe beaucoup de temps à reduire des productions triviales (comme $E \rightarrow T$ dans l'exemple précédent), dont le seul but était d'expliciter dans la grammaire les priorités entre opérateurs et leur associativité droite ou gauche.

Considérations pratiques si ol'n peut décrire de façon concise ces priorités entre opérateurs et leur associativité droite ou gauche, sans toucher à la grammaire, on obtient une description plus modulaire du langage qui nous intéresse (cf *yacc*).

24.1 Exemple de grammaire ambiguë

Considérons la grammaire (ambiguë) suivante:

$S \rightarrow E\#$ $E \rightarrow E * E$ $E \rightarrow E + E$ $E \rightarrow id$

et sa table d'analyse SLR.

Nous allons considérer comment les nombreux conflits apparents peuvent s'expliquer en terme d'associativité et précedence d'opérateurs, que l'on peut résoudre en travaillant directement sur les entrées de la table. . .

A faire....